ORACLE®

ATG WEB COMMERCE

Version 10.0.2

Programming Guide

Oracle ATG
One Main Street
Cambridge, MA 02142
USA

ATG Programming Guide

Document Version

Doc10.0.2 DYNPROGv1 4/15/2011

Copyright

Contents

1 Introduction

ATG provides an open, server-side environment for building and deploying dynamic, personalized applications for the web and other communication channels, such as email and wireless devices. ATG applications implement a component development model based on JavaBeans and JSPs. Developers assemble applications out of component beans (based on standard ATG classes or custom Java classes) by linking them together through configuration files in Nucleus, ATG's open object framework. Page designers build the front-end interface for the application out of JSPs that use ATG's DSP tag library. The DSP tag library makes it possible to embed Nucleus components in JSPs, and use those components for rendering dynamic content.

Each chapter in this manual focuses on a fundamental aspect of ATG application architecture. For specific information about ATG applications such as ATG Commerce or ATG Merchandising, see their online product documentation.

2 Nucleus: Organizing JavaBean Components

Nucleus is ATG's component model for building applications from JavaBeans. Nucleus lets you assemble applications through simple configuration files that specify what components are used by the application, what parameters are used to initialize those components, and how those components hook up to each other.

Nucleus by itself provides no application-specific functions. The JavaBean components implement all of an application's functionality. Nucleus is the mechanism that gives those components a place to live, and a way for those components to find each other.

Nucleus organizes application components into a hierarchy, and assigns a name to each component, based on its position in the hierarchy. For example, a component named `/services/logs/FileLogger` represents a component called `FileLogger`, contained by the container component called `logs`, which is itself contained by the container component called `services`. The `services` component is contained by the root component of the hierarchy, which is Nucleus. Components in the hierarchy can refer to each other by name. This includes both absolute names, such as `/services/logs/FileLogger`, and relative names such as `../servers/HttpServer`.

Nucleus also takes on the task of creating and initializing components. An application does not need to contain the code that creates a component and adds it to the Nucleus namespace. Instead, you can write a configuration file that specifies the class of the component and the initial values of the component's properties. The first time that component is referenced by name, Nucleus finds the component's configuration file, creates the component based on the values in that configuration file, and adds the component to the Nucleus namespace.

Nucleus provides a simple path for writing new components. Any Java object with an empty constructor can act as a component in Nucleus, so writing a new Nucleus component is as easy as writing a Java class. By adhering to JavaBeans standards for defining properties and events, a Java class can take advantage of Nucleus's automatic creation and configuration mechanism. By implementing various interfaces, a Nucleus component can also take advantage of Nucleus services and notifications.

In this chapter

The sections of this chapter help you understand and use Nucleus as a framework for application components:

Building Applications from JavaBeans

A typical Internet application usually begins with an architectural diagram. For example, a database connection component might be connected to a data cache, which is accessed by a search engine that is attached to some UI component. When you build an architectural plan, you typically follow several rules:

- Use existing components where appropriate. If no component exists to do the job, try subclassing an existing component.
- Break down large components into smaller components. Smaller components are easier to test, reuse, and inspect at runtime. This might result in a larger number of components, but Nucleus is designed to handle large numbers of components. Large monolithic components are sometimes difficult to spot, so always be on the lookout. It is generally good practice to design each component to perform a single function that can be described in a short paragraph.
- Centralize functions that are shared by multiple components. For example, one component might spin off a thread that causes email to be sent every hour, while another component might spin off another thread that archives a log file each day. Both timing threads can be eliminated if the components take advantage of a centralized Scheduler component.
- If a component is not completely self-contained—usually the result of following the previous point—be sure that its dependencies on other components are clearly enumerated. These dependencies are usually listed as properties of the component (see below). For example, a component might require a pointer to a Scheduler component and a `DatabaseConnection` component, so the component has properties of those types. A component should never need to know about its position in the grand scheme of the architecture—it only needs to know its most immediate dependencies.

When the architectural plan is complete, you can implement it with Nucleus and JavaBeans. If you design each component as a JavaBean, you can rely on Nucleus to create, initialize, and establish the relationship between Beans. You can build the components without regard for their initialization values or how their dependencies on other components are satisfied. These application-specific concerns are contained in configuration files that are read and interpreted by Nucleus.

Using Properties to Connect Components

To be interpreted by Nucleus, a bean’s initialization parameters must be exposed as properties. For example, a server component might wish to expose its TCP port as a configurable parameter. To do so, it implements the following methods:

```
public int getPort();
public void setPort(int port);
```

Defining these two methods allows Nucleus to treat `port` as a property that can be initialized by a configuration file. The implementation of these methods is unimportant; most implementations use a member variable to store the value of `port`, but this is not required by the JavaBeans specification.

Nucleus can also display the values of properties to an administrator at runtime, so it is often a good idea to expose instrumented values as read-only properties—that is, properties that do not have a write method:

```
public int getHandledRequestCount();
```

As mentioned earlier, properties can also satisfy interdependencies among components. For example, if a component needs a pointer to a Scheduler component to operate, it simply exposes that dependency as a property:

```
public Scheduler getScheduler();
public void setScheduler (Scheduler scheduler);
```

The configuration file can specify which Scheduler to be used by the component, and Nucleus automatically sets the `scheduler` property accordingly. In this way, properties can express the strong dependencies between components—that is, dependencies where one component must have a pointer to another kind of component in order to operate.

Using Events to Connect Components

In addition to dependencies, applications often use a weak relationship to describe notifications and messages. These are encapsulated by JavaBean events. An event is designed into a source bean when one or more listener Beans wish to be notified of some event that takes place on the source bean. This is described as a weak relationship because neither bean needs to know about the other in order to run. While the application might require the connection in order to work properly, the components themselves do not require it. Logging, for instance, uses JavaBean events: individual components do not require listeners for their log events, but the application as a whole usually requires certain logging connections to be in place. For more information, see Events and Event Listeners in the Core ATG Services chapter.

Nucleus configuration files can establish event source/listener relationships. The event source configuration file specifies which components act as event listeners, and Nucleus automatically makes the connections.

After establishing the architecture, components, dependencies, and configuration of the application, the developer can then hand the whole application over to Nucleus and watch it run.

Basic Nucleus Operation

Nucleus performs one basic operation: resolving component names. Given the name of a component, Nucleus does its best to find or create that component and return it.

Within Nucleus, certain components can contain other components, forming a component hierarchy. For example, given the name `/services/logs/FileLogger`, Nucleus resolves it by looking for components in the following locations:

1. Looks in the root container for the `services` component.
2. Looks in the `services` container for the `logs` component.
3. Looks in the `logs` container for the `FileLogger` component.

Nucleus recognizes any component that implements `atg.naming.NameContext` as a container of other components, thereby allowing that component to participate in the naming hierarchy.

Nucleus can also resolve names relative to some `NameContext`. For example, Nucleus can resolve the name `../db/Connections` relative to the logs `NameContext`, which in the end translates to `/services/db/Connections`.

Name resolution is not a difficult task. Nucleus shows its real power by creating components and hooking them up to other components automatically. This function is invoked when Nucleus is asked to resolve a name of a component that does not yet exist. In this case, Nucleus looks for the appropriate configuration file that describes how to create that component and any other components that it requires.

On startup, Nucleus is given a configuration path—a list of configuration directories that contain the configuration files of various components. Within the configuration directories, Nucleus expects to find a configuration file that uses the component name.

For example, to create the component `/services/logs/FileLogger`, where the configuration root directory is `<ATG10dir>/DAS/config`, Nucleus looks for `FileLogger`'s configuration in:

```
<ATG10dir>/DAS/config/services/logs/FileLogger.properties
```

Configuration File Format

The configuration file is a properties file that follows the `key=value` format expected by the class `java.util.Properties`. For example:

```
$class=somepackage.FileLogger
fileName=/work/logs/log1
maximumFileSize=20000
```

The properties file lists the property values used to initialize the new component. For example, when this component is created, its `fileName` property is set to `/work/logs/log1`, and its `maximumFileSize` property is set to `20000`.

The properties file also includes special properties that are read only by Nucleus. These special properties begin with a $ character. In the previous example, the `$class` property is required by Nucleus to determine what class of object to create for the component. So when Nucleus is asked to resolve the name `/services/logs/FileLogger`, it creates an object of class `somepackage.FileLogger`, binds that object into the naming hierarchy, and sets its `fileName` and `maximumFileSize` properties. The new component remains in the namespace, so the next time Nucleus resolves that same name it retrieves the same component without having to create it again.

The previous example shows how Nucleus sets simple property values such as Strings and integers. Nucleus can also set properties to other Nucleus components. For example, the `FileLogger` component might require a pointer to a `Scheduler` component; in this case, it sets a `scheduler` property to a Scheduler component, as follows:

```
$class=somepackage.FileLogger
fileName=/work/logs/log1
maximumFileSize=20000
scheduler=/services/Scheduler
```

In order to initialize this `FileLogger`, Nucleus must resolve the component name `/services/Scheduler`. This might require Nucleus to create a `Scheduler` component, which might further require initialization of other components. After all components are resolved and created, the `scheduler` property is set and the initialization of the `FileLogger` component is complete.

For more information about how Nucleus sets component properties, see Managing Properties Files.

Using Nucleus

This section explains the various ways you can assemble an application with Nucleus. In this section, you work through a series of exercises that demonstrate Nucleus capabilities.

Before you begin

1. Make sure that the ATG platform is properly installed. See the *ATG Installation and Configuration Guide*.
2. Make sure that the Java Software Development Kit binaries are in your path. The JSDK binaries are usually found in the JSDK distribution under the JSDK's `bin` directory.
3. Start up an ATG application that has been assembled in development mode.
4. Create a directory to hold your Java class files. Add this directory to your CLASSPATH environment variable by editing your ATG environment file:
   - Windows: `<ATG10dir>\home\localconfig\environment.bat`:

     `set CLASSPATH=%CLASSPATH%;`*`class-directory-path`*
   - UNIX: `<ATG10dir>/home/localconfig/environment.sh`:

     `CLASSPATH=${CLASSPATH}:` *`class-directory-path`*

Alternatively, put the class directory in `<ATG10dir>/home/locallib`, which is already part of the default CLASSPATH when you run the ATG platform.

5. Create a tutorial directory `<ATG10dir>/home/localconfig/test` where you can run the tutorial exercises.
6. Change directories to the tutorial directory so the tutorial directory becomes the current directory.
7. Set the `DYNAMO_HOME` environment variable to `<ATG10dir>/home`.
8. Set your environment variables in the command line console by running `<ATG10dir>/home/bin/dynamoEnv`. On UNIX platforms, you can do this by starting an instance of `sh` (if you are using any other shell) and then executing `dynamoEnv.sh` directly into that shell:

   ```
   . bin/dynamoEnv.sh
   ```

Creating a Nucleus Component

In this section, you define a class that is used as a Nucleus component, then configure the component.

Define the Class

Define a simple `Person` class that has two properties: `name` (a String) and `age` (an integer):

```
public class Person {
  String name;
  int age;

  public Person () {}
  public String getName () { return name; }
  public void setName (String name) { this.name = name; }
  public int getAge () { return age; }
  public void setAge (int age) { this.age = age; }
}
```

Put this class definition in your `classes` directory with the name `Person.java`. The ATG platform includes an `<ATG10dir>/home/locallib` directory that you can use for any Java class files you create. This directory is included in the ATG CLASSPATH by default, so any classes stored there are picked up automatically.

Note: The `locallib` directory is intended for evaluation and development purposes only. For full deployment, you should package your classes in an application module, as described in Working with Application Modules.

Use the `javac -d` command to compile your `Person.java` source file and add the resulting `.class` files to the `locallib` directory:

```
javac -d <ATG10dir>/home/locallib Person.java
```

Configure a Component

Now create an instance (or component) of a `Person` class,`/test/services/Person`:

1. In the `test` directory, create a `services` directory.
2. In the `services` directory create a file called `Person.properties` with the following contents:

   ```
   $class=Person
   name=Stephen
   age=20
   ```

You can now view the `Person` component in the Components window. Select `test/services/Person` and click Open Component. The Component Editor should display `Person` and its two properties.

Starting a Nucleus Component

When you start up an application, Nucleus reads the configuration path, which is a list of directories to use to find configuration files. Within one of those directories is a file called `Nucleus.properties` that contains the name of the first component to create. In the standard ATG platform configuration, the start of the `Nucleus.properties` file looks like this:

```
$class=atg.nucleus.Nucleus
initialServiceName=/Initial
```

The `initialServiceName` property instructs Nucleus to configure and start up its initial service using `Initial.properties`, which in the standard ATG platform configuration looks like this:

```
$class=atg.nucleus.InitialService
initialServices=\
     /atg/Initial,\
     VMSystem,\
     /atg/dynamo/StartServers
```

If you want to add another service to the list of initial services, you can edit the `/Initial` component in the Components window:

1. Select the `Initial` component and click Open Component.
   A Component Editor opens, displaying the properties of the `Initial` component.
2. Select the first property, `initialServices`. This property displays for its values the services listed in the `Initial.properties` file.
3. Click **...** to view the complete list of the values for the `initialServices` property.
4. From that list of values, select the last value, `/atg/dynamo/StartServers`, and click Insert After. A new blank value field appears, with an @ button.
5. Click the @ button. A dialog appears, displaying the available components.
6. Select the `test/services/Person` component and click OK. The new component appears in the list of values for the `initialServices` property.

Now, the next time you start your application, the `test/services/Person` component is run as an initial service.

Most components do not need to be started from the `Initial` service when an application starts up; they can be instantiated by Nucleus when they are needed, typically in response to a page request from a user. A component started through the `initialServices` property must be globally scoped.

To show that Nucleus really is doing something, change the `Person` class to print some output:

```
public class Person {
  String name;
  int age;

  public Person () {
    System.out.println ("constructing Person");
  }
  public String getName () { return name; }
  public void setName (String name) {
    System.out.println ("setting name to " + name);
    this.name = name;
  }
  public int getAge () { return age; }
  public void setAge (int age) {
    System.out.println ("setting age to " + age);
    this.age = age;
  }
}
```

Compile this class, reassemble your application, and restart it. On the console you should be able to watch the class get constructed and initialized.

Note: The forward slash / in `/test/services/Person` is always used when naming Nucleus components. It is independent of the file separator character that varies among operating systems.

Public Constructor

When Nucleus creates a component from a properties file, Nucleus calls the component's constructor, which takes no arguments. This means that the component must be declared as a `public` class, and the component must have a `public` constructor that takes no arguments. The `Person` class, for example, defines such a constructor:

```
public Person () {}
```

Even if a component does nothing, this constructor must be defined in order for Nucleus to be able to create the component from a properties file.

Property Names

In the previous example, the `Person` class defined properties name and age, of types `String` and `int` respectively. The properties were defined by the fact that the class defined methods `getName`, `setName`, `getAge`, and `setAge`.

The JavaBeans specification details how to define properties; however, the basic rules are as follows:

- To define a configurable property, a class defines a `getX` method that takes no arguments and returns a value, and a `setX` method that takes one argument and returns `void`. The type returned by the `getX` method must be the exact same type as the type taken as an argument by the `setX` method, and can be any Java type. Both the `getX` and `setX` methods must be declared `public`.

  One exception applies: the `getX` method for a Boolean property can be replaced by `isX`. For example, the Boolean property `running` can be defined by the method `getRunning()` or `isRunning()`.

- The property name is formed by removing `get` or `set` from the method name and changing the first letter of the remaining string to lower case. For example, the method `getFirstName()`defines a property called `firstName`.

  One exception applies: if the first two letters of the remaining string are both capitalized, no letters are changed to lower case. For example, the method `getURL()` defines a property called `URL`.

Property names are case-sensitive. Thus, the entry `Age=20` does not set the property `Person.age`.

Property Types

In the `Person` example, Nucleus creates a component of class `Person`, and sets the values of a `String` and `int` property from the values found in the properties file. Nucleus can parse these values from the properties file because it is configured to recognize `String` and `int` property types. Nucleus can parse any property type for which a property editor is registered using the `java.beans.PropertyEditorManager` class.

Simple Property Types

The following is a list of the simple Java data types that Nucleus can parse from properties files:

```
boolean
byte
char
short
int
long
float
double
java.lang.Boolean
java.lang.Byte
java.lang.Character
java.lang.Short
java.lang.Integer
java.lang.Long
java.lang.Float
java.lang.Double
java.lang.String
java.util.List
java.util.Map
java.util.Locale
```

Complex Property Types

The following table lists more complex Java data types that Nucleus can parse from properties files, and describes how Nucleus interprets each value:

Data Type	Value
`java.io.File`	Read from the properties file as a String, then converted into a file. For example, if the value is `c:\docs\doc1` then the `File` object is equivalent to calling `new File ("c:\docs\doc1")`.
`atg.xml.XMLFile`	An absolute configuration pathname—for example, `/atg/dynamo/service/template.xml`
`java.util.Date`	Parsed according to the rules of the default `java.text.DateFormatter`.
`java.util.Properties`	Read as a comma-separated list of `key=value` pairs. For example, `a=17,b=12,c=somestring`.
`java.lang.Class`	Read as a class name, and `Class.forName()` is called to convert the name to a `Class`. If there is no such class, an exception is thrown.
`java.net.InetAddress`	See IP Addresses in Properties Files in this section for information about setting numeric `InetAddress` properties.
`atg.repository.rql.RqlStatement`	A Repository Query Language statement (see the *ATG Repository Guide*).
`atg.nucleus.ServiceMap`	Read as a comma-separated list of `key=serviceName` pairs. For example, `a=/cart,b=../foreach` (see ServiceMap Properties).
`atg.nucleus.ResolvingMap`	A map property whose values are linked to a property of another component (see Linking Map Properties)

Arrays as Property Values

Nucleus can parse arrays of any of the types shown earlier. Nucleus parses an array by assuming that the element values are separated by commas. For example, a property named `heights` might be defined by the following methods:

```
public double [] getHeights ();
public void setHeights (double [] heights);
```

The `heights` property might be set by the following properties file entry:

```
heights=3.2,-12.7,44.6
```

In this case, Nucleus creates an array of three `doubles`, and assigns that array to the component by calling `setHeights()`.

Leading and trailing white spaces are included in element values. For example, given the following property setting:

```
name=Mary,Paul, Peter
```

array elements Paul and Peter embed a trailing space and a leading space, respectively.

Hashtables as Property Values

Hashtable property values are parsed in the same way as `java.util.Properties` and `atg.nucleus.ServiceMap` type properties, as a comma-separated list of `key=value` pairs.

Defining Property Types

You can define and register additional property types, using the `setAsText` method of the JavaBean `PropertyEditor` interface. See the JSDK API documentation for `java.beans.PropertyEditor`.

Properties File Format

The properties files read by Nucleus must conform to a format that is recognized by the class `java.util.Properties`, as described in the following sections.

Properties files created by the ATG Control Center automatically use the correct format. The ATG Control Center also checks whether a property value you enter is valid for the property's data type. The Components editor presents array type properties in a table, with a separate row for each property value entry, so you do not need to continue lines with backslashes .

Note: Nucleus-specific properties are prefixed by the $ character. See Special ($) Properties.

Single-line Property Settings

A property setting must use one of the following formats:

- *`propertyName`*`=`*`propertyValue`*
- *`propertyName`*`:`*`propertyValue`*

A property value can span multiple lines if each line is terminated by a backslash (\) character. For example:

```
targetCities=\
        Detroit,\
        Chicago,\
        Los Angeles
```

This is equivalent to `targetCities=Detroit,Chicago,Los Angeles` (white space at the beginning of lines is ignored).

White Space

White space that follows the property value is treated as part of the property value.

White space is ignored in the following cases:

- Beginning of a line
- Between the property name and property value, so the following are equivalent.

  ```
  name=Stephen
  name = Stephen
  ```

- Blank lines

Special Characters

Certain characters and strings are given special treatment, as described in the following table.

! #	If placed at the beginning of a line, comments out the line.
`\n`	Newline character
`\r`	Carriage return
`\t`	Tab
`\\`	Inserts a backslash character. For example: `path=c:\\docs\\doc1`
`\u`	Prefixes a UNICODE character—for example, `\u002c`

Class versus Component Names

It is important to differentiate class names from Nucleus component names. Multiple components in Nucleus cannot have the same absolute name, but they can have the same class. For example, in the previous section the class `Person` is instantiated as the component `/services/Person`. It might also be instantiated as another component—for example, `/services/Employee`.

It is especially important to differentiate Java source files from the properties files required to build an application. For both types of files, the file's position in the namespace also determines its position in the file directory. For example:

- A Java file for the class `atg.nucleus.Nucleus` should live at `{SOURCEDIRECTORY}/atg/nucleus/Nucleus.java`
- The properties file for component `/services/log/fileLogger` should live at `{CONFIGDIR}/services/log/fileLogger.properties`.

A component name and class name are sometimes the same—typically, when an application instantiates a single component from a given class. For example, the class `atg.service.scheduler.Scheduler` might be instantiated as the component `/services/Scheduler`.

Specifying Components as Properties

Previous examples show how Nucleus creates and initializes components from properties files. Nucleus also allows components to point to each other through configuration file properties.

For example, a `Weather` component might be defined in Nucleus, and the `Person` component needs a pointer to that `Weather`. The `Weather` class might look like this:

```
public class Weather {
  String currentWeather;

  public Weather () {
    System.out.println ("constructing Weather");
  }
  public String getCurrentWeather () {
    return currentWeather;
  }
  public void setCurrentWeather (String currentWeather) {
    System.out.println ("setting currentWeather to " + currentWeather);
    this.currentWeather = currentWeather;
  }
}
```

This example requires instantiation of a Nucleus `Weather` component, `/services/Weather`. You should compile the `Weather` Java class and create a `Weather` class component with a `Weather.properties` file in the same directory as `Person.properties`. The properties file might look like this:

```
$class=Weather
currentWeather=sunny
```

Next, modify the `Person` class so it defines a property that is set to a `Weather` component:

```
public class Person {
  String name;
  int age;
  Weather weather;

  public Person () {
    System.out.println ("constructing Person");
  }
  public String getName () { return name; }
  public void setName (String name) {
    System.out.println ("setting name to " + name);
    this.name = name;
  }
  public int getAge () { return age; }
  public void setAge (int age) {
    System.out.println ("setting age to " + age);
    this.age = age;
  }
  public Weather getWeather () { return weather; }
  public void setWeather (Weather weather) {
    System.out.println ("setting weather to " + weather.getCurrentWeather());
    this.weather = weather;
  }
}
```

Finally, modify the `Person` component's properties file so it has a `weather` property that points to the `weather` component:

```
$class=Person
name=Stephen
age=20
weather=Weather
```

If you include the `Person` component as an initial service (described in the earlier section Starting a Nucleus Component), when you start your application, the `Person` component is created and initialized. Its `name` and `age` properties are set from the values found in the properties file. In order to set the `weather` property, Nucleus resolves the name `Weather` by creating and initializing the `Weather` component before assigning it to the `Person` property. The output should look something like this:

```
constructing Person
setting name to Stephen
setting age to 20
constructing Weather
setting currentWeather to sunny
setting weather to sunny
```

The first two lines of the output show that Nucleus created the `/services/Person` component and set the `age` property. Then Nucleus attempts to set the `weather` property. In doing so, it searches for the component named `Weather`. This is a relative name, and so it is resolved relative to the current context `/services`, resulting in `/services/Weather`.

Nucleus searches its existing components and, finding that there is no /services/Weather, it tries to create one from the configuration file services/Weather.properties. This causes Nucleus to construct an instance of the Weather class and initialize its currentWeather property, thereby resulting in the third and fourth lines of output.

Now that a /services/Weather component is created and initialized, Nucleus can initialize the rest of the Person component, by setting its weather and name properties. This results in the last two lines of output.

Nucleus does not limit the number of components that refer to each other through properties. For example, component 1 can refer to component 2, which refers to component 3, and so on. Nucleus can even resolve circular references without spiraling into infinite loops. For example, component 1 might have a property that points to component 2, which has a property that points back to component 1. However, you should try to avoid circular references as they can result in deadlocks. See Enabling Deadlock Detection for information about avoiding deadlocks.

Application errors can also occur if you reference a property of a component before that component is completely configured. To diagnose this type of error, set the loggingInfo property of the / Nucleus service to true, and the ATG platform prints information messages for this situation.

Arrays of components can also be specified in the same way that other array values are specified: as a comma-separated list. For example, the Person component might have a property called cityWeathers that contains an array of Weather components:

```
public Weather [] getCityWeathers ();
public void setCityWeathers (Weather [] cityWeathers);
```

This property might be initialized in the configuration file like this:

```
cityWeathers=\
        /services/weather/cities/atlanta,\
        /services/weather/cities/boston,\
        /services/weather/cities/tampa,\
        /services/weather/cities/phoenix
```

Nucleus handles this by finding each of the components in the list, arranging the found components into a 4-element array, then assigning that array to the cityWeathers property of the Person component.

ServiceMap Properties

It is often useful to have a property that maps Strings to other components. In Nucleus, properties of type atg.nucleus.ServiceMap are assumed to perform this mapping. For example, a cities property might map a city name to the weather component monitoring it:

```
import atg.nucleus.*;

public ServiceMap getCities ();
public void setCities (ServiceMap cities);
```

The corresponding properties file might initialize the `cities` property as follows:

```
cities=\
        atlanta=/services/weather/cities/atlanta,\
        boston=/services/weather/cities/boston,\
        tampa=/services/weather/cities/tampa,\
        phoenix=/services/weather/cities/phoenix
```

The `ServiceMap` class is a subclass of `java.util.Hashtable`, so you can access it with all the normal `Hashtable` methods such as `get`, `keys`, `size`, and so on. In this case, the key `atlanta` maps to the component found at `/services/weather/cities/atlanta`, and so on for the other cities. The following code accesses the `Weather` component for a particular city:

```
Weather w = (Weather) (getCities ().get ("tampa"));
```

Component Names in Properties Files

When a name is resolved in a properties file, it is resolved one element at a time. In the previous example, a component was specified as `Weather`. The name resolution begins at the context where the name was found. You can think of this as the directory containing the properties file, which in this case was `/services`. The name is then resolved one element at a time. Because this name consists of only one element, the result is `/services/Weather`.

The name `Weather` is a relative name, meaning that its resolution starts with the directory where it was found. Any name that does not begin with a `/` is considered a relative name. For example, `Weather`, `../service1`, `logger/FileLogger`, and `..` are all relative names.

On the other hand, any name that begins with a `/` is considered an absolute name. For example, the following are all treated as absolute names:

```
/services/Weather
/services/somedir/../Weather
/
```

Absolute names are resolved by starting from the root and resolving each element of the name in order.

Dot Names

In both absolute and relative names, dot names have special meanings. These dot names can be used anywhere in a name, relative and absolute:

Notation	Description
`.` (single)	Refers to the current component, and usually has no effect on the name resolution process. For example, `Person` and `./Person` are equivalent, as are `/services/log/FileLogger` and `/services/log/./FileLogger`.

Notation	Description
`..` (double)	Refers to the parent of the current component. For example, `/services/log/../tests/BigTest` is equivalent to `/services/tests/BigTest`.

`...` (triple)	Initiates a search up the component hierarchy for the name specified after the triple dot. For example, the name `.../Adder` searches the current context for a component called `Adder`, then searches the current component's parent. It continues its search up the hierarchy until the component is found, or the root is reached—that is, no more parents can be found.
	
	The triple dot can also be used in more complex names. For example, given this name:
	
	`/services/logs/.../files/TestFile`
	
	these names are searched in the following order:
	
	`/services/logs/files/TestFile`
	`/services/files/TestFile`
	`/files/TestFile`
	
	In summary, Nucleus searches for everything after the triple dot by walking up the hierarchy defined by everything before the triple dot. If Nucleus cannot find the component and must try to create it, Nucleus uses the same search algorithm to find the component's property configuration file.

Aliases for Nucleus Components

ATG includes a class that lets you use an alias for Nucleus components. This class, `atg.nucleus.GenericReference`, lets you use a name of a Nucleus component to reference another component instance in Nucleus. This is useful if you want systems to have separate names, but be backed by the same service instance. If necessary, someone can later change the configuration of the referencing service to have its own instance. All other systems that utilize the original Nucleus name do not need to be reconfigured. Note that the aliased component must have global scope.

To use the `GenericReference` class:

1. Create an instance of `atg.nucleus.GenericReference`.
2. Give the `GenericReference` the alias name you want to use.
3. Set the `GenericReference's` `componentPath` property to the Nucleus address of the globally scoped component you want to reference.

For example, an application might use a customized pricing model for each customer. The pricing model is not actually a separate component, but is contained within the profile repository. You can refer to the pricing model as if it were a separate component with a Nucleus address like `/atg/commerce/pricing/PricingModels`. The Nucleus component at `/atg/commerce/pricing/PricingModels` is a `GenericReference` whose `componentPath` property points to the profile repository as follows:

```
componentPath=/atg/userprofiling/ProfileAdapterRepository
```

If you later decide to move pricing models out of the user repository and set them up as a separate component, you only need to change the configuration of `/atg/commerce/pricing/PricingModels` to use the class of the new separate component instead of `atg.nucleus.GenericReference`.

Pre-Parsed Component and Parameter Names

The `atg.nucleus.naming` package includes two classes that can pre-parse often used component names and parameter names:

- ComponentName
- ParameterName

You can use these classes to assign a name to a component or parameter and store the name and its corresponding component or parameter in a hashtable. This typically speeds up name resolution for components and parameters.

ComponentName

A `ComponentName` object of class `atg.nucleus.naming.ComponentName` can represent any Nucleus component. Use this class to create unique component names that you can reference elsewhere. The component names are stored in a global hashtable that is keyed by strings. Using this class provides better performance by pre-parsing the component name.

To get the unique `ComponentName` for a given String, call the static method `getComponentName()`. This method looks up the given string in a hashtable of component names and returns the value or creates one with the supplied string. For example, you might set a `ComponentName` value as follows:

```
public final static ComponentName PEACH =
  ComponentName.getComponentName("/atg/fruits/Peach");
```

You can pass a component name to the `resolveName()` method of `atg.servlet.DynamoHttpServletRequest`:

```
public Object resolveName(ComponentName pName);
```

This technique can help limit the amount of parsing required to resolve the same component name repeatedly. The ATG page compiler uses `ComponentNames` wherever possible to reduce the memory cost and parsing time to resolve components. `GenericService` implements the `atg.nucleus.naming.ComponentNameResolver` interface, which makes available a `resolveName()` method that takes a `ComponentName`:

```
public Object resolveName(ComponentName pName);
```

ParameterName

You can use a `ParameterName` object of class `atg.nucleus.naming.ParameterName` to represent any request parameter name used in the ATG platform. `ParameterNames` are used when you want to look up a request parameter quickly. Use this class to create unique parameter names when building your own servlet beans. The parameter names are stored in a global hashtable, keyed by strings. Using this class makes the parameters of a servlet bean publicly available. You can use this class not only to enhance performance by pre-parsing the parameter name, but also to enforce good coding standards, by ensuring that the parameter name string appears once only in your Java code.

To get the unique `ParameterName` for a given String, call the static method `getParameterName()`. This method looks up the given string in the hashtable of parameter names and returns the value or creates one with the supplied string. For example, you can set a `ParameterName` value as follows:

```
public final static ParameterName EMPTY = ParameterName.getParameterName("empty");
```

Later, you can reference that parameter name through the string EMPTY as follows:

```
request.serviceLocalParameter(EMPTY, request, response);
```

You can pass a `ParameterName` to the following methods of `atg.servlet.DynamoHttpServletRequest`:

```
public Object getParameter (ParameterName pName);
public Object getLocalParameter (ParameterName pName);
public Object getObjectParameter (ParameterName pName);
```

This technique is useful when you want to resolve the same parameter repeatedly. The ATG page compiler uses `ParameterNames` wherever possible to reduce the memory cost and parsing time of accessing request parameters. You do not need to use a `ParameterName` for parameters that are found in standard ATG servlet beans or in `<valueof>` or `<setvalue>` tags; the page compiler takes care of that for you. If you create your own servlet bean, however, you can obtain better performance if you use `ParameterName` for its parameters.

File Names in Properties Files

Sometimes a property refers to a file name, rather than a component or component's property. Properties of type `java.io.File` can use ATG system properties as part of the file's pathname, with the system property name in curly braces. You can use this notation with the following system properties:

System property notation	Description
`{atg.dynamo.home}`	Resolves to `<ATG10dir>/home` Example: `errorLog={atg.dynamo.home}/logs/error.log`
`{atg.dynamo.root}`	Resolves to `<ATG10dir>` Example: `helpDir={atg.dynamo.root}/DAS/help`

`{atg.dynamo.server.home }`	Resolves to the home directory of the specified server.
	
	Example:
	
	`archiveDir=\`
	`  {atg.dynamo.server.home}/`*`servername`*`/logs/archiv e`

IP Addresses in Properties Files

You can persistently set a property to an IP address. Java tries to convert a numeric `InetAddress` to a host name, and if it succeeds, the host name alone is employed when Java saves or transmits the value of the address. If no host name is available, the numeric form used.

ATG server References

If a component needs a name for the current instance of an ATG server, it can refer to the `serverName` property of the `/atg/dynamo/service/ServerName` component. The server can be named explicitly by setting the `serverName` property directly, or the name can be built from the combination of the server hostname (obtained dynamically) and the `DrpServer` port.

Services that require a server name should not set a server name property directly from this services `serverName` property. Instead, they should obtain a reference to the `/atg/dynamo/service/ServerName` component and call the `serverName()` method. This forces the `ServerName` component to be fully started, allowing the name to be built properly if necessary.

For more information about ATG servers, see the *ATG Installation and Configuration Guide*.

dynamosystemresource

The ATG platform includes a URL protocol named `dynamosystemresource`. You can use this protocol to refer to any file in the system CLASSPATH. Just as Nucleus makes components available through a Nucleus address relative to the ATG configuration path, the `dynamosystemresource` protocol makes files available through an address relative to the CLASSPATH. For instance, the following notation identifies a file with a path relative to the CLASSPATH of `somepackage/file.txt`:

```
dynamosystemresource:/somepackage/file.txt
```

You can use a URL in this form as a property value in components. For example:

```
fileLocation=dynamosystemresource:/somepackage/file.txt
```

Starting Multiple Components

Many applications require creation of multiple components when Nucleus starts. For example, an application might be running three different server components. It is unlikely that these server components refer to each other, so starting one of the servers does not necessarily start the other two.

You can start multiple components through a single component that references all components that must be started, then start that component. The `Initial` component of class `atg.nucleus.InitialService` that exists specifically for this purpose. Because it is specified in `Nucleus.properties`, it is always guaranteed to start, and in turn starts other services that are specified in its `initialServices` property:

```
$class=atg.nucleus.InitialService
initialServices=\
     /atg/Initial,\
     VMSystem,\
     /atg/dynamo/StartServers
```

The `initialServices` property specifies three services that start when Nucleus starts. You can use this technique to initialize entire sections of an application.

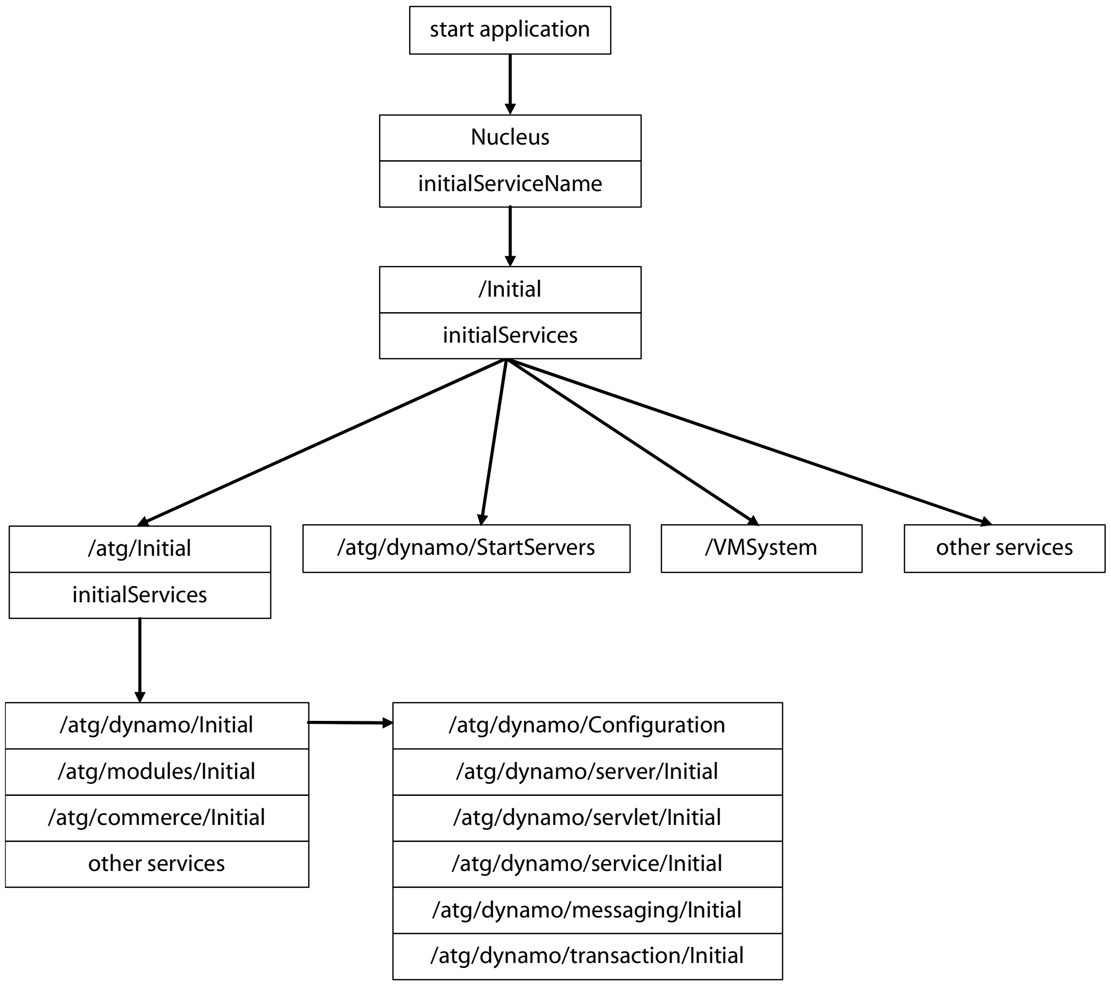
For example, an application might include multiple servers and loggers, where servers and loggers are started by two Initial components:

- `/servers/Initial` starts the servers.
- `/loggers/Initial` starts the loggers.

The `initialServices` property of the master `/Initial` component references these two components. This lets you manage each set of services separately, while ensuring that they are always included in the overall startup process.

Note: A component that is started through the `initialServices` property must be globally scoped.

The following diagram shows how an ATG configuration can ensure that startup of a Nucleus-based application precipitates startup of multiple initial services:



You can configure the ATG platform to send logging info messages for each component that is started by setting the following property in the `Nucleus` component:

```
loggingInfo=true
```

Linking Property Values

In a system that contains many components, it is common that multiple components are initialized with the same property values. In order to maintain consistent property settings among these components, you can specify common property settings in one place, which the various components can all reference. Nucleus lets you link the property of one component to the property in another component through the ^= operator, as follows:

`property-name`^=*`component-name.property-name`*

Note: No white space should precede or follow the ^= operator.

For example, you might want to initialize the `currentWeather` property in the `Sunny` component from the `currentWeather` property in the `RainyWeather` component. To do this, set the `Sunny` component's properties file as follows:

```
$class=Weather
currentWeather^=RainyWeather.currentWeather
```

When Nucleus starts, the `Sunny. currentWeather` property obtains its value from `RainyWeather.currentWeather`.

Note: Property linkage only occurs when the related components are initialized. After initialization, the linked properties are completely independent. So, given the previous example, changes to `RainyWeather.currentWeather` have no effect on `Sunny.currentWeather`.

Typically, frequently modified configuration values are placed into properties of a single component, which only serves to hold the property values referenced by other component. All components in the application that need those values link their properties to this component.

Linking Map Properties

The Nucleus class `atg.nucleus.ResolvingMap` lets you link the value of a map key to another component property, through the ^= operator:

```
myResolvingMapProperty=\
   key1^=/component-name.property-name
   key2^=/component-name.property-name
 ...
```

Note: No white space should precede or follow the ^= operator.

For example, the following property setting links the value of `activeSolutionZones` to another component property `textActiveZones` property:

```
relQuestSettings=\
  activeSolutionZones^=/MyStuff/MyIndexingOutputConfig.textActiveZones
```

Debugging Nucleus Configuration

To help you identify configuration problems, Nucleus displays messages about the configuration process. You configure the level of logging messages for Nucleus through the configuration file `Nucleus.properties`. By default, Nucleus is configured to display warning and error messages; however, it can be configured to display informative and debugging messages also For example, the default configuration looks like this:

```
initialServiceName=/Initial
loggingError=true
loggingWarning=true
```

```
loggingInfo=false
loggingDebug=false
```

If you have problems with configuration, it can be helpful to set the `loggingInfo` and `loggingDebug` properties to `true` and examine the messages for indications where the problems might be.

Note: These settings apply only to Nucleus, and do not affect the logging settings of other components.

For more information about using the logging system in the ATG platform, see the ATG Logging section of the Logging and Data Collection chapter.

Enabling Deadlock Detection

When Nucleus starts up a component, it also starts up other components referenced by that component's properties. Nucleus can resolve circular references, where a component referred to by another component has properties that refer back to the first component. Circular references can cause deadlocks, which result from multiple threads trying to lock the two components in different orders. To debug deadlock problems, add this setting to the `Nucleus.properties` file:

```
debugComponentLock=true
```

If `debugComponentLock` is set to `true` and a potential deadlock is detected, a `DeadlockException` is thrown and displayed in the console. The exception indicates the lock in question, the thread it conflicts with, and the locks held on each side. Locks are named after the components involved, which can help determine which component is causing the problem. One or more of the components is incompletely initialized, so you need to fix the source of the problem and restart the ATG platform.

Component Scopes

An application component can be set to one of the following scopes:

- Global: Component is shared among all users.
- Session: Separate instances of the component are provided to each user.
- Request: Separate instances of the component are provided to each active request.
- Window: Separate instances of the component are provided to each browser window.

Specifying Component Scopes

You specify a component's scope by setting its `$scope` property to `global`, `session`, `request`, or `window`. For example, a NewPerson component might be set to session scope as follows:

```
$class=Person
$scope=session
name=Bill
age=28
```

If a component's `$scope` property is not explicitly set, it automatically has global scope.

Property Object Scopes

A component's properties should always point to objects whose scope is equal to or greater than its own. Thus, global-scope component properties should only point to objects that also have global scope; session-scope component properties should only point to objects that have global or session scope; while request-scope component properties can point to objects of any scope, including request.

Global Scope

If you set a component to global scope, it is accessible to all users across multiple sessions. For example, multiple users might simultaneously access an input form that updates a NewPerson component, initially set as follows:

```
$class=Person
name=Bill
age=28
```

If the NewPerson component has global scope, each user can update the values of this component from their respective browser sessions, and thereby overwrite changes posted by other users. In general, in an application that is accessed by multiple users, components like this are set to session or request scope, in order to guarantee data integrity and consistency within the current session or request.

Session Tracking

ATG provides a session-tracking mechanism that automatically matches browser requests to a server session. The ATG platform uses cookies or rewritten URLs to identify requests from a given browser as belonging to the same session.

If the browser makes no requests after a period of time, the ATG platform assumes that the user has left the application. It invalidates the session and removes the components associated with that session. Component data that was not copied to permanent storage is lost.

For more information about session tracking, see Session Management in ATG Applications in the *ATG Installation and Configuration Guide*.

Multiple Scopes in the Same Namespace

If a component has session or request scope, separate instances of the component are distributed to the various sessions or requests that access it. The Component Browser can show the multiple scopes of a given component. If you click on `/atg/dynamo/servlet/sessiontracking/SessionManager`, the Component Browser displays components with unique identifiers that correspond to the sessions associated with those components. In each component, you should see separate instances of the entire component tree.

When Nucleus needs to resolve a component name, it merges the global tree of components with the tree of components for a specific session. This allows the two scopes to appear in the same namespace, but still be separated in the real tree of components.

Request Scope

If a component is marked with `request` scope, simultaneous requests each see a different instance of the component. This is true even when the same session sends two requests simultaneously; each request gets a pointer to a separate object. Each instance is handled independently and has no effect on the others.

Request scope can be especially useful for components whose properties are set by a form. When a form is submitted, the component values are set by the appropriate `setX` methods, and actions are performed by `handleX` methods.

If two forms are submitted at the same time to the same component, one submission might overwrite the `setX` methods of the other. This is especially true for globally-scoped components, which are highly vulnerable to multiple simultaneous requests from different sessions; with a session-scoped component, multiple simultaneous requests occur only if the user submits the form twice in very rapid succession. As a general rule, it is a good idea for forms to use request-scoped components; this ensures that only one request at a time can set their properties.

Note: To ensure that multiple requests do not access the same component simultaneously, you can also set the `synchronized` attribute in the `form` tag. With this attribute, the ATG platform locks the specified component before setting any properties, and releases the lock only after form submission is complete. Other form submissions can set the component's properties only after the lock is released. For more information, see the Forms chapter in the *ATG Page Developer's Guide*.

Preserving Request Scoped Objects on Redirects

If a request results in a redirect to a local page through the method `HttpServletResponse.sendLocalRedirect()`, the ATG platform treats the redirect request as part of the original request, and maintains any request-scoped objects associated with that request. To implement this, the ATG platform adds an additional query parameter named `_requestid` to the redirected URL.

Setting Properties of Session and Request-Scoped Components

At any given time, a session-scoped or request-scoped component might have multiple instances. For example, a session-scoped component instance might exist for each user logged on to the site.

When a component instance is created, the ATG platform does not create objects for its properties. In order to minimize memory use, new component properties are set as pointers to existing instances of those objects. Consequently, be careful how you set properties of a session-scoped or request-scoped component; changing the value of a property is liable to affect other component instances, depending on the property data type:

- You can safely set the value of an immutable object such as a `String` property. In this case, ATG platform creates a `String` object and sets the property to it. The property has a unique reference to the `String` object which other component instances cannot affect.
- If you change the value of a mutable object such as an array, always replace the object rather than modify the object in place.

For example, given an array property `myArray String[]` set to `{"a", "b", "c"}`, you should change its last element by creating an array with the desired change and setting the property to it:

```
setMyArray(new String[] {"a", "b", "z"}
```

Conversely, the following code incorrectly modifies the array in place, and is liable to affect other component instances:

```
String[] arr = getMyArray()
arr[2] = "z";
```

Managing Properties Files

Nucleus provides several ways to manage an application's properties files. Multiple configuration directories can set properties differently for various modules and their components. You can also use global properties files to set a property value to the same value in different components.

Setting the Configuration Path

On assembly, an application's configuration path is set to one or more configuration directories. These directories are set from the configuration path attributes in module manifest files. Precedence of configuration path directories determines how component properties are set, and is generally determined by two factors:

- Precedence of configuration path attributes
- Module list order and dependencies

Configuration Path Attributes

On application assembly, each module adds to the configuration path the directories that are set in the module's manifest file. Each configuration path attribute can set one or more directories in the following format, where spaces delimit multiple directories, and directory paths are relative to the module's root directory:

```
config-path-attr: config-dir[ config-dir]...
```

For example, the DAS module manifest `<ATG10dir>/DAS/META-INF/MANIFEST.MF` sets the attribute `ATG-Config-Path` as follows:

```
ATG-Config-Path: config/config.jar
```

On application assembly, the directory's absolute pathname is added to the configuration path as follows:

```
<ATG10dir>/DAS/config/config.jar
```

The following table lists the configuration path attributes that a module's manifest file can set:

Attribute	Specifies directories of….
`ATG-Config-Path`	Configuration files that are required by module application components.
`ATG-`*`cfgName`*`Config-Path`	Configuration files that are associated with the named configuration *`cfgName`*. These files are enabled when the application is assembled with the switch `-layer` *`cfgName`*. For example, the following configuration path attribute is used and its directories are added to the configuration path when the application is assembled with the switch `-layer Management`: `ATG-ManagementConfig-Path: mgmt_config/`
`ATG-`*`app-svr`*`Config-Path`	Configuration files that are specific to the third-party application server specified by *`app-svr`*. For example, `ATG-JBossConfigPath` points to configuration files that are specific to the JBoss platform.
`ATG-`*`app-svrCfgName`*`Config-Path`	Platform-specific configuration files that are associated with the named configuration *`CfgName`*. These files are enabled when the application is on the application server *`platform`* and is assembled with the switch `-layer` *`CfgName`*. For example, the following configuration path attribute is used and its directories are added to the configuration path when the application runs on JBoss, and the application is assembled with the `-layer Management` switch: `ATG-JbossManagementConfig-Path: mgmt_config/`
`ATG-LiveConfig-Path`	Module resources that provide Nucleus configuration files. The specified directories are added to the configuration path when the `-liveconfig` switch is supplied during application assembly.
`ATG-`*`platform`*`LiveConfig-Path`	Platform-specific configuration files. The specified directories are added to the configuration path when two conditions are true: the application runs on the platform specified by *`platform`*; and the `-liveconfig` switch is supplied during application assembly.

Precedence of Configuration Path Attributes

The directories specified by the configuration path attributes of each module are appended to the configuration path in the following order (left-to-right), where the left-most path (set by `ATG-Config-Path`) has lowest precedence:

1. `ATG-Config-Path`
2. `ATG-`*`cfgName`*`Config-Path`
3. `ATG-`*`platform`*`Config-Path`
4. `ATG-`*`app-svrCfgName`*`Config-Path`
5. `ATG-3rdPartyConfig-Path`
6. `ATG-LiveConfig-Path`
7. `ATG-`*`platform`*`LiveConfig-Path`

Module List Order and Dependencies

The previous section describes how the configuration path is set from a single module. Because an ATG application is assembled from multiple modules, the assembly process must determine precedence among them when it orders their respective configuration directories in the configuration path. These modules include application modules that are explicitly specified in the assembly module list, and ATG modules such as DAS and DSS. Together, these comprise the expanded module list, and the assembly process must resolve dependencies among them when it creates the configuration path.

The ordering of directories from various modules in the configuration path is generally determined by two factors:

- The order of the module list that is explicitly supplied for application assembly
- Dependencies among modules within the expanded module list

Unless inter-module dependencies mandate otherwise, Nucleus sets module configuration directories in the configuration path in the same order as the module list. For example, a startup script might be supplied the following module list:

```
-m foo bar
```

In this case, you can generally expect that the configuration directories specified by module `foo` are set in the configuration path before those in module `bar`. Thus, given overlapping component properties, settings in `bar` have precedence over those in `foo`. However, if `foo` is directly or indirectly dependent on `bar`, their order in the configuration path is reversed to reflect this dependency. In that case, `bar` precedes `foo`, so `foo` settings have precedence.

The ATG modules in the expanded module list might also have dependencies; these are resolved during application assembly, before the configuration path is created. Dependencies are not always obvious; you can view their resolution during application startup, when Nucleus outputs the application's configuration path.

Reading the Configuration Path

Component properties are set according to the precedence of configuration path directories. For example, a configuration path might look like this:

```
<ATG10dir>/DAS/config/config.jar:<ATG10dir>/DAS/home/localconfig
```

Given this configuration path, properties that are set in `localconfig` always have precedence over those set in `config`. So, when Nucleus needs to configure the component `/services/Sunny`, it looks for `Sunny.properties` as follows:

1. `<ATG10dir>/DAS/config/services/Sunny.properties`
2. `<ATG10dir>/DAS/home/localconfig/services/Sunny.properties`

If Nucleus fails to find `Sunny.properties` in the configuration path, it generates an error.

Configuration Path versus CLASSPATH

An application's configuration path and Java's CLASSPATH behave differently as follows:

- Configuration files found in the configuration path are merged, not replaced; and the last-found properties in configuration files have precedence over those found earlier.
- `.class` files found earlier in CLASSPATH supersede files found later.

Note: Never place `.class` files in the configuration path directory path. Doing so can yield errors, as the `.class` files might interfere with Nucleus' ability to resolve component names.

Configuration Directories

As installed, ATG sets the configuration path to the following directories:

- `config` is the module base configuration directory, specified by the configuration path attribute `ATG-Config-Path.config`. This directory contains configuration files that are used to start up components required by ATG products.

  Because each new ATG distribution overwrites the configuration files in `config`, you should not edit their properties.
- `localconfig` contains custom properties settings, and have the highest priority in the configuration path. Settings in `localconfig` are preserved across product upgrades; changes to base configuration properties can be safely set here.

Depending on application requirements, you can set the configuration path so it includes settings for specific configurations:

- Application server configuration
- Production environment configuration
- Named configuration

Application Server Configuration

During application assembly, you can add to the configuration path directories that are specific to your application server. You do so by setting the configuration path attribute `ATG-`*`platform`*`Config-Path`, where the string *`platform`* denotes the application server—for example, `ATG-jbossConfig-Path`.

Production Environment Configuration

Two configuration path attributes can be used to configure an application for production:

- `ATG-LiveConfig-Path` contains settings appropriate for applications that are deployed for production.
- `ATG-`*`platform`*`LiveConfig-Path` contains production-ready settings that are specific to your application server. The string *`platform`* denotes the application server—for example, `ATG-jbossLiveConfig-Path`.

The directories specified by these attributes are included in the configuration path only when the application is assembled with the `-liveconfig` switch (see the *ATG Installation and Configuration Guide*).

Named Configuration

Two configuration path attributes let you associate a set of properties files with a user-defined name:

- `ATG-`*`cfgName`*`Config-Path` specifies configuration directories that are associated with the named configuration *`cfgName`*—for example, `ATG-ManagementConfig-Path`.
- `ATG-`*`app-svrCfgName`*`Config-Path` specifies configuration directories that are associated with the named configuration *`CfgName`* and are specific to the application server denoted by *`app-svr`*—for example, `ATG-JBossManagementConfig-Path`.

In order to set the configuration path with the configuration directories associated with *`cfgName`*, the application must be assembled with the `-layer` *`cfgName`* switch.

Named configurations are useful for associating related configuration settings that span multiple modules, which can be simultaneously invoked under specific conditions.

For example, given the following configuration path attributes—

```
ATG-ManagementConfig-Path: management_config/
ATG-JbossManagementConfig-Path: jboss_management_config/
```

—the configuration path includes these directories if you assemble the application as follows:

```
runAssembler MyApp.ear –layer Management –m Service.admin
```

Note: You can assemble a single EAR file that contains all named configuration layers that are required across various servers, then activate the desired named configuration layers on each server by setting the system property `atg.dynamo.layers` on server startup. For more information, see Specifying Configuration Layers on Server Startup.

Setting Properties from Multiple Configuration Directories

Nucleus can configure a component from properties that are set in multiple configuration directories. If configuration files in multiple directories set the same property, Nucleus can resolve this in two ways:

- Override property settings from previously-read configuration files.
- Combine multi-value property settings from multiple configuration files.

Override Property Settings

You can override the settings in any properties file from another properties file that has higher precedence. For example, any settings in a module’s base configuration can be overridden by a properties file of the same name in the application’s `localconfig` directory.

Combine Multi-Value Property Settings

You can concatenate settings from multiple files for a given multi-value property (such as an array or list) by using the += operator, as follows:

`property-name``+=`*`property-setting[, property-setting]...`*

Note: No white space should precede or follow the += operator.

This can be especially useful when you need to supplement existing settings that are supplied by the base configuration of an ATG module. For example, the standard ATG distribution provides the base `Initial.properties`, which starts a number of services:

```
$class=atg.nucleus.InitialService
initialServices=\
     servers/Initial,\
     statistics/Sampler,\
     sessiontracking/SessionManager,\
     snmp/Initial
```

It is likely that your application also starts its own set of services. Because installation of any later ATG distribution always overwrites the base `Initial.properties`, you should not modify this file so it includes application-specific services. You also should not override it with another copy of `Initial.properties` elsewhere in the configuration path—for example, in `localconfig`. If the next ATG distribution changes the installed list of initial services, those changes are shadowed by the `localconfig` version of `Initial.properties`.

Instead, you can concatenate settings for the `initialServices` property from the base and `localconfig` versions of `Initial.properties`. For example, you might modify `localconfig/Initial.properties` as follows:

```
initialServices+=store/CatalogManager
```

Given the previous settings in the base `Initial.properties`, this yields the following settings for `initialServices`:

```
servers/Initial
statistics/Sampler
sessiontracking/SessionManager
snmp/Initial
store/CatalogManager
```

By using the += operator, the `store/CatalogManager` entry is appended to the list of services already set by the base version of `Initial.properties`. When Nucleus reads the configuration path, it finds two `Initial.properties` files in `/config/config.jar` and `/localconfig`, and combines `initialServices` settings from both files. If product updates change the base version's set of services, the `/localconfig` setting is appended to the new set.

Global Properties Files

A global properties file can set the same property in multiple components. The property settings in a `GLOBAL.properties` file apply to all components in the file's configuration directory and subdirectories.

For example, `/localconfig/services/GLOBAL.properties` might have the following setting:

```
currentWeather=miserably hot
```

This setting is applied to any component in `/services` and its subdirectories that contains the `currentWeather` property.

Precedence of Global and Component Settings

A component's own property settings have precedence over global property settings. For example, if the component `/services/Sunny` sets the `currentWeather` property, that value overrides the global setting; if the component omits the `currentWeather` property, it uses the global setting. A component can also be set by multiple global properties files, where the global properties file that is most proximate to the component has precedence over other global properties files.

In the following example, the component `/services/Sunny` is configured by two global properties files and its own properties file, listed in ascending order of precedence:

```
localconfig/GLOBAL.properties
localconfig/services/GLOBAL.properties
localconfig/services/Sunny.properties
```

Combining Global and Component Settings

Property files can append values to those set by a global properties file. For example, a global properties file might declare the property `affectedCities`:

```
affectedCities=Detroit,Boston,Los Angeles
```

A contained component can append a single value to the same property as follows:

```
affectedCities+=Chicago
```

This yields the following composite of settings for the component:

```
affectedCities=Detroit,Boston,Los Angeles,Chicago
```

Site-Specific Component Properties

In a multisite environment, an ATG component can be configured to substitute site configuration values for the values otherwise provided by the component's own properties, by setting two Nucleus properties:

- `$instanceFactory`: Set to the component `/atg/multisite/SiteSourcedPropertyGetterSubClasser` (class `atg.service.subclasser.SiteSourcedPropertyGetterSubClasser`).

  This component provides a `cglib2`-based proxying mechanism, which enables site-specific behavior for this component.

- `$overridePropertyToValuePropertyMap`: Map component properties to the corresponding site properties.

For example, you can configure a component so one of its properties is set from a site's custom configuration property `cssFile` . The component is defined with the following methods:

```
package my;
import atg.service.GenericService;
public class PageConfiguration extends GenericService {
...
String mStyleSheet;
  public void setStyleSheet(String pStyleSheet) {
    mStyleSheet = pStyleSheet;
  }

  public String getStyleSheet() {
    return mStyleSheet;
  }
  ...
}
```

The PageConfiguration component is configured as follows:

```
$class=my.PageConfiguration
$instanceFactory=/atg/multisite/SiteSourcedPropertyGetterSubClasser
$overridePropertyToValuePropertyMap=styleSheet=cssFile

styleSheet=default.css
```

At startup, Nucleus sets the component's `styleSheet` property to `default.css`. However, on processing each request for this property, Nucleus uses the current site context to look up the site configuration property `cssFile`. If a value is found, Nucleus overrides the component's `styleSheet` property setting and returns the site-specific value in `cssFile`. If `cssFile` is null, Nucleus returns `default.css`.

You can also specify to return null values from the site configuration by setting the Nucleus property `$nullAsOverridePropertyNames`. This property specifies to return null if the designated site configuration settings are also null. It disregards any settings that are set within the properties file itself or derived from other configuration directories. For example, you might modify the previous configuration with this setting:

```
$nullAsOverridePropertyNames=styleSheet
```

So, if `cssFile` is `null`, Nucleus overrides all other settings for `styleSheet` and returns `null`.

Tracing Component Property Settings

When an application has multiple configuration directories, a component can get its properties from multiple sources. You can use the ATG Dynamo Server Admin Component Browser to determine how a given component is configured:

1. Navigate to the target component.
2. Click View Service Configuration to view a hierarchical listing of the properties files for that component.

Setting Properties to Null

A null value does not override a non-null value set earlier in the configuration path. For example, a component with the property `/atg/foo/boo` with a `smells` might be set as follows:

```
$class=atg.service.funkystuff.foo.Boo
smells=spicy
```

A configuration file with higher precedence in the configuration path cannot override this property value by setting it to blank or `null` as follows:

```
smells=
smells=null
```

The ATG platform provides a `Constants` Nucleus service that lets you set null values by reference. This service, with a Nucleus address of `/Constants`, has a single `null` property that is set to `null`. Thus, you can set a property value to `null` as follows:

```
smells^=/Constants.null
```

Decoding Encrypted Properties in Nucleus Components

You might decide to encode or encrypt sensitive information that is stored in properties files with the class `atg.core.util.Base64`, or another method. In this case, you must be able to access the encrypted information.

Note: The ATG distribution currently supports BASE64 decryption only.

The `atg.nucleus.PropertyValueDecoder` class defines a component that can decode the value of properties encoded with Base64. You can use a component of this type to protect properties that should remain encoded until their value is used. For example, `DataSource` components can use a

PropertyValueDecoder component to decrypt user and password properties before using them to create a database connection. These sensitive pieces of information are protected in the `DataSource`'s properties file until they are needed

To use a PropertyValueDecoder, modify the original component to use the decoder for the encoded property. Do not make the decoded value visible to any public method or property. The PropertyValueDecoder's `decode()` method should be called and its return value used directly (apart from type casting). This lets your component use different implementations of the `PropertyValueDecoder` interface without modification.

As installed, the `atg.service.jdbc.FakeXADataSource` class supports this feature. To use it, follow these steps:

1. Create an `atg.service.jdbc.SimpleLoginDecoder` component that implements `PropertyValueDecoder`—for example, `MySimpleLoginDecoder`.
2. Set the `loginDecoder` property of `FakeXADataSource` to `MySimpleLoginDecoder`.
3. Set the values of the `user` and `password` properties in `FakeXADataSource` with Base64-encoded values. You can rely on the decoder to pass the decoded login to the database when connections are created.

If you need more robust security, you can subclass `LoginDecoder` and override its `decode()` methods, or implement your own `PropertyValueDecoder`.

Loading Serialized Beans

In addition to specifying a class name, the `$class` property in a properties file can be used to define an instance of a serialized JavaBean. A serialized JavaBean can be obtained from an IDE tool; you can also create one with the `ObjectOutputStream` class. These files have a `.ser` suffix, and are stored in the CLASSPATH.

The value of the `$class` attribute should be a name of the form `x.y`. This first looks for a file `x/y.ser` in your CLASSPATH. If that file does not exist, Nucleus loads the class `x.y`. Nucleus uses the standard JavaBean utility method `Beans.instantiate` to implement this feature.

If you do not use any serialized JavaBeans, you can improve performance by disabling checks for `.ser` files. To disable checking, set the `checkForSerFiles` property of the top-level Nucleus component to `false`. For example, your `<ATG10dir>/home/localconfig/Nucleus.properties` might include this setting:

```
checkForSerFiles=false
```

Checking File Name Case on Windows

Nucleus component names are case-sensitive. An operating system that supports case-sensitive file names can also support component names that differ only in case. For example, UNIX can differentiate `Person.properties` and `person.properties`, which configure components `Person` and `person`, respectively.

Windows does not support case-sensitive file names, so it cannot distinguish between properties files `Person.properties` and `person.properties`. Consequently, attempts to create creating components `Person` and `person`, cause one configuration file to overwrite the properties of the other. To avoid this, set the `checkFileNameCase` property of the top-level Nucleus component to true. This setting prevents you from creating components whose names are different only in case.

Note: Setting `checkFileNameCase` to true can slow performance, so set it to true only during development When the application is ready for deployment, be sure to reset this property to false.

XML File Combination

Some Nucleus components use XML files instead of Java properties files for configuration or other initialization tasks. Like properties files, several XML files of the same name can appear along the configuration path. At runtime, the ATG platform combines these files into a single composite file, which is then used by the appropriate component. This allows multiple applications or modules to layer on top of each other, forming a single definition file from multiple definition files.

This section describes the operations and rules that are used to combine two XML files into a new XML file. XML files are combined one tag at a time; in other words, tags are matched up, and the combination rules are applied to each pair of matched tags.

XML file combination is controlled by an XML attribute `xml-combine`. This attribute is used only in the preprocessing stage of XML file combination. Because the `xml-combine` attribute is not included in the file that results from the preprocessing combination of the XML files, it does not need to appear in the document type definition (DTD) for the XML files.

XML Encoding Declaration

The first line of an XML file should begin with this declaration:

```
<?xml version="1.0" ?>
```

An XML file with this declaration is assumed to use UTF-8 encoding for escaped Unicode characters. You can specify another character encoding with a declaration of this form:

```
<?xml version="1.0" encoding="encoding-name" ?>
```

`encoding-name` is the name of a supported XML encoding—for example, `ISO-8859-1` or `SHIFT_JIS`. For a list of the XML encodings supported by the ATG XML parser, go to http://xml.apache.org/xerces-j/faq-general.html.

Note: If you combine files with different encodings, the combined XML file (which exists only as a temporary file) uses a common encoding of UTF-8.

DOCTYPE Declaration

When you combine XML files, only one file can have a DOCTYPE declaration. This file must be earliest in the configuration path of all files to combine. The DOCTYPE that this base file declares determines the DOCTYPE of the resulting file.

Combining Two Tags

When the configuration path contains two XML files of the same name, matching tags are combined according to the `xml-attribute` specified by the tag in the second (last-read) XML file (see the next section, Controlling Tag Combination). If this attribute is omitted, the following combination rules are followed:

- If one of the combined tags contains only a PCDATA section—that is, a text block without embedded tags—the first file's tag is discarded and the tag content of the second file is used (equivalent to `xml-combine=replace`).
- In all other cases, the contents of the tag in the second file are appended to the contents of tag in the first file (equivalent to `xml-combine=append`).

Given these rules, you can combine most XML files without explicitly setting the `xml-combine` attribute, reserving its use for special situations.

The values of XML elements can be set in the DTD. If an XML element has a default setting specified in the DTD, that default setting is applied in any XML file that does not explicitly set the element. For example, the SQL Repository DTD specifies the `expert` attribute of the `property` element as follows:

```
expert           %flag;       "false"
```

If your base SQL repository definition file sets the `expert` attribute of a property to `true`, and if supplemental SQL repository definition files modify that property, you must also explicitly set the `expert` attribute of a property to `true` in the supplemental SQL repository definition files; otherwise the attribute's value reverts to the default specified in the DTD.

Controlling Tag Combination

You can override the default rules for tag combination by setting the `xml-combine` attribute in tags of the last-read configuration file. `xml-combine` can be set to one of the following values:

- `replace`
- `remove`
- `append`
- `append-without-matching`
- `prepend`
- `prepend-without-matching`

Note: The `xml-combine` attribute is removed from the combined file.

replace

Only the tag in `file2.xml` is used; the tag in `file1.xml` is ignored. for example:

File1.xml

```
<people>
  <person name="joe">
    <interests>
      <interest interest="rollerblading"/>
      <interest interest="bass"/>
    </interests>
  </person>
</people>
```

File2.xml

```
<people>
  <person name="joe" xml-combine="replace">
    <interests>
      <interest interest="parenting"/>
    </interests>
  </person>
</people>
```

Result

```
<people>
  <person name="joe">
    <interests>
      <interest interest="parenting"/>
    </interests>
  </person>
</people>
```

remove

The tag is removed from the combined file. For example:

File1.xml

```
<people>
  <person name="joe">
    <interests>
      <interest interest="rollerblading"/>
      <interest interest="bass"/>
    </interests>
  </person>
</people>
```

File2.xml

```
<people>
  <person name="joe" xml-combine="remove"/>
</people>
```

Result

```
<people>
</people>
```

append

The contents of `file2.xml`'s tag are appended to the contents of `file1.xml`'s tag. For example:

File1.xml

```
<people>
  <person name="joe">
    <interests>
      <interest interest="rollerblading"/>
      <interest interest="bass"/>
    </interests>
  </person>
</people>
```

File2.xml

```
<people>
  <person name="joe">
    <interests xml-combine="append">
      <interest interest="parenting"/>
    </interests>
  </person>
</people>
```

Result

```
<people>
  <person name="joe">
    <interests>
      <interest interest="rollerblading"/>
      <interest interest="bass"/>
      <interest interest="parenting"/>
    </interests>
  </person>
</people>
```

Embedded tags are matched and combined recursively.

append-without-matching

Identical to `xml-combine="append"`, except embedded tags are not matched and combined recursively. Tag content is simply appended.

prepend

The contents of `file2.xml`'s tag are prepended to the contents of `file1.xml`'s tag. For example:

File1.xml

```
<people>
  <person name="joe">
    <interests>
      <interest interest="rollerblading"/>
      <interest interest="bass"/>
    </interests>
  </person>
</people>
```

File2.xml

```
<people>
  <person name="joe">
    <interests xml-combine="prepend">
      <interest interest="parenting"/>
    </interests>
  </person>
</people>
```

Result

```
<people>
  <person name="joe">
    <interests>
      <interest interest="parenting"/>
      <interest interest="rollerblading"/>
      <interest interest="bass"/>
    </interests>
  </person>
</people>
```

Embedded tags are matched and combined recursively (see the Recursive Combination).

prepend-without-matching

Identical to `prepend`, except embedded tags are not matched and combined recursively. Tag content is simply prepended.

Recursive Combination

If a tag sets `xml-combine` to append or `prepend`, tags that are embedded in the combined tags also are matched and combined recursively. Before the primary tags are combined, they are searched for matching embedded tags. Given embedded tags `tag1.subtag1` and `tag2.subtag2` in `file1.xml` and `file2.xml`, respectively, the two tags match if all attributes in `tag2.subtag1` have matching attributes in `tag1.subtag1`. The attributes in `tag2.subtag2` can be a subset of the attributes in `tag1.subtag1`.

If a tag embedded in `tag1` matches a tag from `tag2`, the tag from `tag1` is replaced by its combination with the matching `tag2` as defined by `tag2`'s `xml-combine` attribute. That tag is replaced in place—that is, it is not appended or prepended.

For example, each of the following XML files contains a `<people>` tag, where the tag in `file2.xml` sets its `xml-combine` attribute to append:

File1.xml

```
<people>
  <person name="joe" title="CTO">
    <interests>
      <interest interest="rollerblading"/>
      <interest interest="bass"/>
    </interests>
  </person>
</people>
```

File2.xml

```
<people xml-combine="append">
  <person name="jeet" title="CEO">
    <interests>
      <interest interest="parenting"/>
    </interests>
  </person>
  <person name="joe" xml-combine="append">
    <interests xml-combine="prepend">
      <interest interest="parenting"/>
    </interests>
  </person>
</people>
```

Before appending, all tags embedded in `tag1` are searched for matches. The search yields the following match:

```
<person name="joe" title="CTO">
<person name="joe">
```

It does not define all the same attributes found in tag1, but those that it does define match.

Because these tags are a match, the tag embedded in tag1 is modified in place to combine the tag from tag1 and the tag from `tag2`. The tag is then removed from `tag2` so that it is not actually appended.

The embedded tags are then combined by recursively going through the entire combination process. In this example they are combined by using append, but the `<interests>` tag in each matches, so the `<interests>` tags are combined by using prepend, and the final result is:

```
<people>
  <person name="joe" title="CTO">
    <interests>
      <interest interest="parenting"/>
      <interest interest="rollerblading"/>
      <interest interest="bass"/>
    </interests>
  </person>
  <person name="jeet" title="CEO">
    <interests>
      <interest interest="parenting"/>
    </interests>
  </person>
</people>
```

If there are multiple matches for a tag, it is undefined which of the matching tags is used.

Root Tag

The rules described earlier specify how two tags are supposed to be combined. However, an additional rule is required to specify how two XML files must be combined.

The rule for combining two XML files is to act as if each file were completely enclosed in a tag, that tag matched for both files, and the tags are being combined with mode append. For example, consider two XML files:

File1.xml

```
<person name="joe" title="CTO">
  <interests>
    <interest interest="rollerblading"/>
    <interest interest="bass"/>
  </interests>
</person>
```

File2.xml

```
<person name="jeet" title="CEO">
  <interests>
    <interest interest="parenting"/>
  </interests>
</person>
<person name="joe" xml-combine="append">
  <interests xml-combine="prepend">
    <interest interest="parenting"/>
  </interests>
</person>
```

The `<people>` tag has been removed for the purpose of this example. In this case, the XML files should act as if they were defined like this:

File1.xml

```
<pretend-enclosing-tag>
  <person name="joe" title="CTO">
    <interests>
      <interest interest="rollerblading"/>
      <interest interest="bass"/>
    </interests>
  </person>
</pretend-enclosing-tag>
```

File2.xml

```
<pretend-enclosing-tag xml-combine="append">
  <person name="jeet" title="CEO">
    <interests>
      <interest interest="parenting"/>
    </interests>
  </person>
  <person name="joe" xml-combine="append">
    <interests xml-combine="prepend">
      <interest interest="parenting"/>
    </interests>
  </person>
</pretend-enclosing-tag>
```

The enclosing tags are combined as normal, and the enclosing tag is omitted from the generated file.

id Attribute

The matching rules described earlier in this section match two tags on the basis of their attribute values. Sometimes, tags cannot be matched in this way. For example, J2EE deployment descriptors do not typically use attributes. Thus, the matching rules cause too many tag matches because there are no attribute values to distinguish the tags.

In this case, it might be necessary to manufacture an attribute. The J2EE deployment descriptors provide the `id` attribute, which is designed to be used when tags need to be matched at a lexical level.

The `id` tag can be used to hold the value of an embedded value that is known to be unique, as in this example:

```
<session>
  <ejb-name>Account</ejb-name>
  ...
</session>

<session>
  <ejb-name>Payroll</ejb-name>
  ...
</session>
```

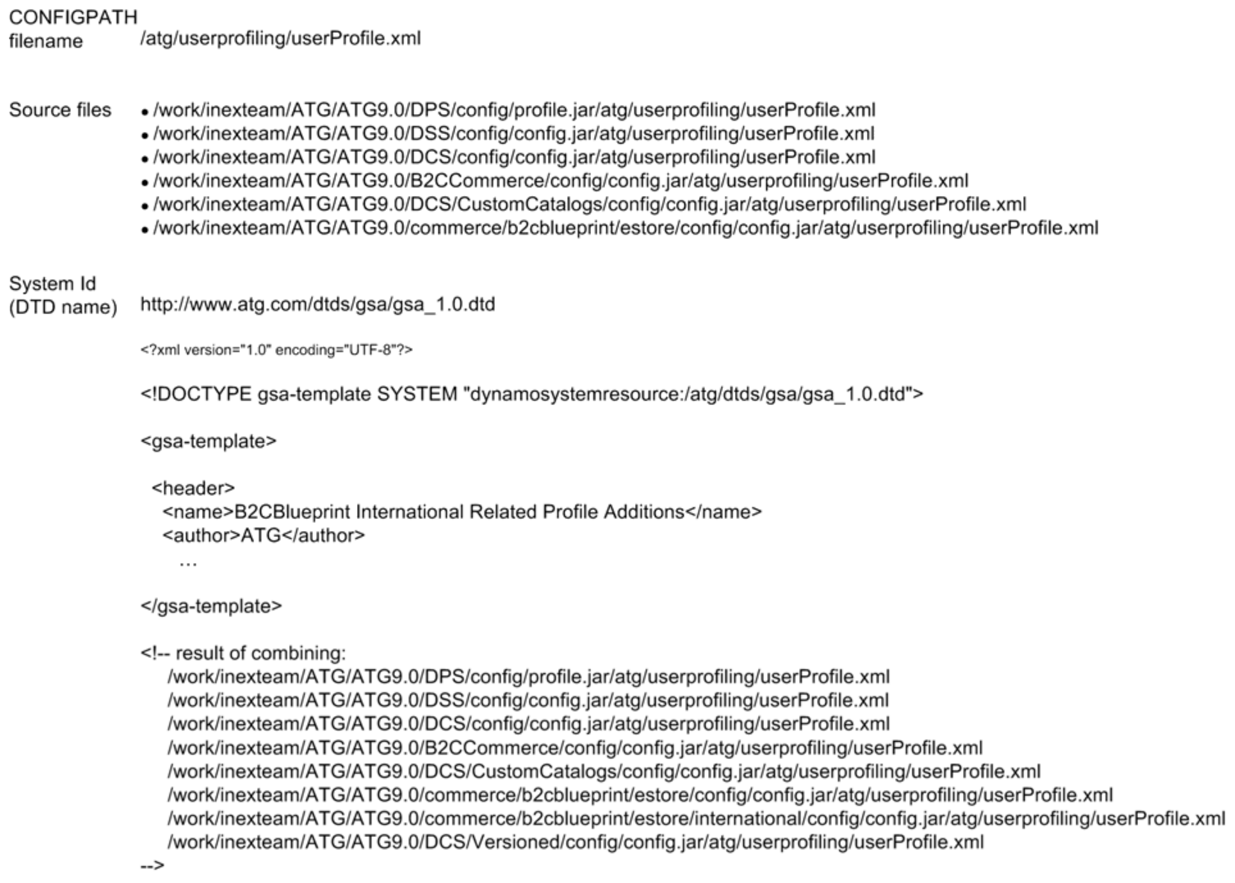
Here, all `<session>` tags are distinguished by the value of their `<ejb-name>` child tag, but that is no help to the XML combination rules. In this case, an `id` attribute is added to facilitate tag matching:

```
<session id="Account">
  <ejb-name>Account</ejb-name>
  ...
</session>
```

```
<session id="Payroll">
  <ejb-name>Payroll</ejb-name>
  ...
</session>
```

Viewing the Combined File

If a running Nucleus component has a property whose value is an XML file, the ATG Dynamo Server Admin Component Browser can show you the configured XML and the source files that combined to create it. For example, the component `/atg/userprofiling/ProfileAdapterRepository` contains the property `definitionFiles`, whose value is set to an XML file. When you click on the property, the Component Browser opens a window that includes the following information:



```
CONFIGPATH
filename      /atg/userprofiling/userProfile.xml

Source files  • /work/inexteam/ATG/ATG9.0/DPS/config/profile.jar/atg/userprofiling/userProfile.xml
              • /work/inexteam/ATG/ATG9.0/DSS/config/config.jar/atg/userprofiling/userProfile.xml
              • /work/inexteam/ATG/ATG9.0/DCS/config/config.jar/atg/userprofiling/userProfile.xml
              • /work/inexteam/ATG/ATG9.0/B2CCommerce/config/config.jar/atg/userprofiling/userProfile.xml
              • /work/inexteam/ATG/ATG9.0/DCS/CustomCatalogs/config/config.jar/atg/userprofiling/userProfile.xml
              • /work/inexteam/ATG/ATG9.0/commerce/b2cblueprint/estore/config/config.jar/atg/userprofiling/userProfile.xml

System Id
(DTD name)    http://www.atg.com/dtds/gsa/gsa_1.0.dtd

              <?xml version="1.0" encoding="UTF-8"?>

              <!DOCTYPE gsa-template SYSTEM "dynamosystemresource:/atg/dtds/gsa/gsa_1.0.dtd">

              <gsa-template>

                <header>
                  <name>B2CBlueprint International Related Profile Additions</name>
                  <author>ATG</author>
                    ...

              </gsa-template>

              <!-- result of combining:
                  /work/inexteam/ATG/ATG9.0/DPS/config/profile.jar/atg/userprofiling/userProfile.xml
                  /work/inexteam/ATG/ATG9.0/DSS/config/config.jar/atg/userprofiling/userProfile.xml
                  /work/inexteam/ATG/ATG9.0/DCS/config/config.jar/atg/userprofiling/userProfile.xml
                  /work/inexteam/ATG/ATG9.0/B2CCommerce/config/config.jar/atg/userprofiling/userProfile.xml
                  /work/inexteam/ATG/ATG9.0/DCS/CustomCatalogs/config/config.jar/atg/userprofiling/userProfile.xml
                  /work/inexteam/ATG/ATG9.0/commerce/b2cblueprint/estore/config/config.jar/atg/userprofiling/userProfile.xml
                  /work/inexteam/ATG/ATG9.0/commerce/b2cblueprint/estore/international/config/config.jar/atg/userprofiling/userProfile.xml
                  /work/inexteam/ATG/ATG9.0/DCS/Versioned/config/config.jar/atg/userprofiling/userProfile.xml
              -->
```

Testing XML File Combination

A reference implementation of the rules described in this section can be found in the scripts `xmlCombine` (for UNIX) and `xmlCombine.bat` (for Windows). These scripts parse a set of input files, combine them and write the result to an output file. These scripts are found in `<ATG10dir>\home\bin`, and are executed with the following syntax:

```
xmlCombine file1 file2 ... -o outputfile
```

Writing Nucleus Components

Any Java class with a constructor that takes no arguments can be instantiated as a Nucleus component, which gives you a wide degree of latitude when you create component classes. However, these classes should implement certain interfaces that give components access to a number of Nucleus capabilities, such as automatic creation, configuration, and notifications.

The easiest way to implement these interfaces is to subclass `atg.nucleus.GenericService`, which implements all interfaces described in the following sections. However, your class might already extend some other class, thereby preventing you from extending `GenericService`. In this case, your class must implement the necessary interfaces.

Note: Your class does not need to implement all interfaces that `GenericService` implements, only the ones with the functionality required.

Public Constructor with No Arguments

Nucleus can create a component automatically from a properties file only if the component class is declared public and it implements a public constructor with no arguments. For example:

```
public class Person {
  public Person () {}
}
```

The constructor is not required to be empty. However, as shown later in this chapter, it is often a good idea to defer initialization functions until after the component starts.

Properties

In order to enable configuration of a class variable from a properties file, the class defines `setX()` and `getX()` methods for that variable, where `X` specifies the variable/property to configure.

For example, to make the variable age configurable from the property `age`, the class defines `setAge()` and `getAge()` methods:

```
int age;
public void setAge (int age) { this.age = age; }
public int getAge () { return age; }
```

Exposing a variable as a property also exposes it to runtime inspection by the Nucleus administrator.

You might also wish to make a property read-only, meaning that it can be inspected at runtime but cannot be configured through a properties file. This technique is often used for properties that contain runtime statistics. To make a property read-only, simply omit the `setX` function:

```
int age;
public int getAge () { return age; }
```

Special $ Properties

Nucleus properties files use several special properties that are indicated by a leading $ character:

`$class`	The component's Java class.
`$scope`	The scope of the component (global, session, or request). The default value is global. See the Component Scopes section in this chapter.
$description	A brief description of the component, for display in the ATG Control Center Components task areas.

$description

Given a large number of components in a typical Nucleus application, it can be helpful to document what each component does. You can document your components using the `$description` property. For instance, you might describe a `Person` component like this:

```
$description=Holds name and age traits of users
```

Event Sources

Your class can be a source for JavaBeans events by following the patterns outlined in the JavaBeans specifications (see Events and Event Listeners in the Core ATG Services chapter). If your class fires events, Nucleus properties files can be used to configure the listeners for those events.

NameContextBindingListener

When Nucleus creates your component from a properties file, it first calls your class constructor with no arguments. It then binds the component into the namespace of the `NameContext` that contains the component. For example, if your component was created with the name `/services/servers/LatteServer`, the component is bound into the `NameContext` at `/services/servers`, using the name `LatteServer`.

If your class implements the `atg.naming.NameContextBindingListener` interface, the component is notified when it is bound into a `NameContext`, and also receives notification when it is unbound from that `NameContext`.

A typical implementation of `NameContextBindingListener` looks like this:

```
import atg.naming.*;

public YourClass implements NameContextBindingListener {
  String name;
  NameContext nameContext;

  public void nameContextElementBound (NameContextBindingEvent ev) {
    if (ev.getElement () == this) {
      nameContext = ev.getNameContext ();
      name = ev.getName ();
    }
  }
  public void nameContextElementUnbound (NameContextBindingEvent ev) {
    if (ev.getElement () == this) {
      nameContext = null;
      name = null;
    }
  }
}
```

Both methods check to verify that the `element` in the event really is the object. This is because the same methods are called if the object is registered as a listener for binding events on other `NameContexts`. For the time being, just remember to include this check before setting the member variables.

Although you can generally assume that these notifications happen all the time, the notifications usually happen only if the `NameContext` also implements `NameContextBindingEventSource`. This is because the `NameContext` is responsible for sending the events, so if a `NameContext` has a less responsible implementation, it might not send the notifications.

NameContextElement

If you implement `NameContextBindingListener`, you might also wish to implement `atg.naming.NameContextElement`. This is a simple extension to the `NameContextBindingListener` interface that exposes the component's `name` and `nameContext` as properties:

```
public NameContext getNameContext () { return nameContext; }
public String getName () { return name; }
```

Exposing these properties gives Nucleus some help when Nucleus is traversing the namespace looking for components. These properties also show up in the Component Browser, which is always a help to the administrator. In general, it is a good idea to implement `NameContextElement` if you already implemented `NameContextBindingListener`.

NameContext

Components that implement `atg.naming.NameContext` are recognized by Nucleus as containers of other components. This means that Nucleus can traverse through these components when it is resolving names. It also means that the Component Browser allows the administrator to walk through the children of that component, in the same way that a web browser allows a user to walk through the files in a directory.

The NameContext interface resembles java.util.Dictionary in that it has methods for getting, putting, removing, and listing elements. One possible implementation of NameContext is to use a Hashtable:

```
Hashtable elements = new Hashtable ();

public Object getElement (String name) {
  return elements.get (name);
}
public void putElement (String name,
                        Object element) {
  removeElement (name);
  elements.put (name, element);
}
public void removeElement (String name) {
  elements.remove (name);
}
public boolean isElementBound (String name) {
  return getElement (name) != null;
}
public Enumeration listElementNames () {
  return elements.keys ();
}
public Enumeration listElements () {
  return elements.elements ();
}
```

Some implementations, however, might not wish to implement all of this functionality. For example, a NameContext can be hard-coded to have three elements: name, price, and availability:

```
public Object getElement (String name) {
  if (name.equals ("name")) return "PowerCenter Pro 180";
  else if (name.equals ("price")) return new Integer (1995);
  else if (name.equals ("availability")) return new Boolean (true);
  else return null;
}
public void putElement (String name,
                        Object element) {
}
public void removeElement (String name) {
}
public boolean isElementBound (String name) {
  return getElement (name) != null;
}
public Enumeration listElementNames () {
  return new Enumeration () {
    int i = 0;
```

```
      public boolean hasMoreElements () {
        return i < 3;
      }
      public Object nextElement () {
        if (!hasMoreElements ()) throw new NoSuchElementException ();
        switch (i++) {
          case 0: return "name"; break;
          case 1: return "price"; break;
          case 2: return "availability"; break;
        }
      }
    };
  }
  public Enumeration listElements () {
    return new Enumeration () {
      Enumeration e = listElementNames ();
      public boolean hasMoreElements () {
        return e.hasMoreElements ();
      }
      public Object nextElement () {
        return getElement (e.nextElement ());
      }
    };
  }
```

Notice how the `putElement` and `removeElement` methods do nothing. Also notice the use of inner classes to implement the methods that return Enumerations. This is a common technique for satisfying these types of interfaces.

Remember that `NameContext` extends `NameContextElement`, so your implementations of `NameContext` must also implement all the methods for `NameContextElement`.

NameContextBindingEventSource

It is often useful for a `NameContext` to fire an event whenever an element is bound or unbound from that `NameContext`. For example, a pricing component might wish to be notified whenever an element is bound into a shopping cart component.

A `NameContext` declares that it fires such events by implementing `NameContextBindingEventSource`. This declares that `NameContextBindingListeners` can be added to the `NameContext`. These listeners receive notifications whenever elements are bound or unbound from the `NameContext`.

If a component declares that it implements `NameContextBindingEventSource`, that component is also expected to perform the following behavior: whenever an element is bound or unbound from the `NameContext`, the `NameContext` should check to see if that element implements `NameContextBindingListener`. If it does, the `NameContext` should send a bound or unbound event to that element. This behavior must be performed in addition to notifying the registered listeners.

A sample implementation of this behavior can be found in the source code for `atg.naming.NameContextImpl`, found in the `<ATG10dir>/DAS/src/Java/atg/naming` directory. This class implements all of the following interfaces:

- `atg.naming.NameContextBindingEventSource`, which extends
- `atg.naming.NameContext`, which extends
- `atg.naming.NameContextElement`, which extends
- `atg.naming.NameContextBindingListener`

If your component implements `NameContext`, you might consider implementing `NameContextBindingEventSource` as well. This is usually only required if arbitrary elements are going to be bound and unbound from your `NameContext`, which is usually not the case for an application-specific component. For example, the last example of the previous section implements a read-only `NameContext`, and implementing `NameContextBindingEventSource` on that object is not very useful.

Naming and Nucleus

All of the previous interfaces dealt with the general naming system interfaces found in `atg.naming`. The rest of the interfaces described in this section are specific to Nucleus. They deal with notifications given to components that are automatically created by Nucleus. They also deal with logging of application-specific messages, and allow components to define their own HTML interfaces for use in ATG Dynamo Server Admin.

ServiceListener

When Nucleus creates a component from a properties file, it goes through the following steps:

- Nucleus constructs the component using the public constructor with no arguments.
- Nucleus binds the component into its parent `NameContext`.
- If the `NameContext` implements `NameContextBindingEventSource` and the component implements `NameContextBindingListener`, the component is notified that it was bound into a `NameContext`.
- Nucleus then configures the component by setting its properties from the values found in the properties configuration file. This might involve resolving names of other components, which can involve creating, binding, and configuring those components as well.
- Nucleus then adds any event listeners defined in the properties file. Again, this involves resolving component names by finding or creating those components.
- After all of the component’s properties are set and its event listeners added, the component is ready to go. Nucleus now notifies the component that it is all set up and ready to start. This notification is only performed if the component implements `atg.nucleus.ServiceListener`.

Notice how the component can receive two notifications—one when it is bound into the `NameContext`, and one when Nucleus is finished configuring its property values. Most application components wait until the second notification before starting their operations. In order for a component to receive this second notification, it must implement `atg.nucleus.ServiceListener`.

The following is a typical implementation of `ServiceListener`:

```
Nucleus nucleus;
Configuration configuration;
boolean running;

public void startService (ServiceEvent ev) throws ServiceException {
  if (ev.getService () == this && !running) {
    nucleus = ev.getNucleus ();
    configuration = ev.getServiceConfiguration ();
    running = true;
    doStartService ();
  }
}
public void stopService () throws ServiceException {
  if (running) {
    running = false;
    doStopService ();
  }
}

public void doStartService () throws ServiceException {}
public void doStopService () throws ServiceException {}
```

First, notice that `startService` checks the service specified by the event to make sure that it is actually this service. Second, notice the use of a `running` flag. This flag is needed because Nucleus might call `startService` multiple times, even after calling `startService` a first time. The use of the `running` flag makes sure that the service performs its initialization functions only once. In this particular implementation, run-once logic is placed in `startService`, while the actual initialization procedures are delegated to another method, such as `doStartService`.

A similar technique is used for the `stopService` method. The `running` flag is used to make sure that the shutdown procedures are executed only once, and the actual shutdown procedures are delegated to the `doStopService` method. A service might be stopped for a variety of reasons: a direct command from the administrator, overall Nucleus shutdown, or service reconfiguration.

A service that has been stopped should be prepared to start again at any time. For example, when reconfiguring a service, the administrator typically stops the service, changes some configuration values, then restarts the service. The service is expected to restart itself using the new configuration values. Thus, a service can expect to be stopped and restarted several times during its lifetime in Nucleus.

Both start and stop methods can throw a `ServiceException` to indicate that some problem has occurred during startup or shutdown.

Service

After a component has implemented `ServiceListener`, it should go on to implement `Service`, which extends `ServiceListener`. The `Service` interface exposes the various properties set by the `ServiceListener` interface:

```
public Nucleus getNucleus () { return nucleus; }
public Configuration getServiceConfiguration () { return configuration; }
public boolean isRunning () { return running; }
```

By implementing this interface, a component exposes these properties for inspection by the Nucleus administrator. The `configuration` property, for example, tells the administrator what properties files were used to configure the component. Your component does not actually need to do anything with the `configuration` property except remember it and return it when asked.

ApplicationLogging

Most application services need a way to report events that occur during the operation of that service. In ATG products, this is handled by having the component fire `LogEvents`. These `LogEvents` are then broadcast to listeners that can handle those events. The ATG platform comes with a set of listener services that can send `LogEvents` to files, to the screen, to databases, and so on.

With this setup, components only have to worry about what logging events they want to report, while other components worry about sending those events to their eventual destinations. Like everything else in Nucleus, the connections between components and their logging destinations are described by properties files.

For the convenience of programmers, the ATG platform uses a standard logging interface called `atg.nucleus.logging.ApplicationLogging`. This interface defines the listener, adding and removing methods needed to define a component as a source of `LogEvents`. This interface also defines a set of properties for indicating what level of logging has been turned on. For example, the `loggingWarning` property describes whether a component should be emitting warning log messages. See Using ApplicationLogging in the Logging and Data Collection chapter for more information.

AdminableService

When the Component Browser in the Administration Interface displays the page for a component, it uses a special servlet that displays the default representation of a component. This servlet shows the name of the component, its contained children, and the values of that component's properties.

Some services might wish to customize this page, perhaps to show more information. The Scheduler service, for example, extends the standard administration servlet to show information about scheduled events.

A component that wishes to present its own administration interface must implement `atg.nucleus.AdminableService`. This interface has a method that allows Nucleus to obtain the servlet that acts as the component's administration interface:

```
public Servlet getAdminServlet ();
```

The component is expected to return a servlet from this method. Inner classes are often used to produce this servlet.

A full description of how to customize the administrative servlet is located in the Customizing the ATG Dynamo Server Admin Interface section of this chapter.

GenericService

As described above, classes used for Nucleus components typically implement a large number of standard interfaces. When you create classes, you can greatly simplify your task by extending the `atg.nucleus.GenericService` class. This class implements most of the key Nucleus interfaces, so classes that extend it also implement those interfaces.

Furthermore, two important interfaces, `atg.naming.NameContext` and `atg.naming.NameContextBindingEventSource`, are implemented by a subclass of `GenericService`, `atg.nucleus.GenericContext`, described later. Thus, a class can implement these interfaces by extending the `GenericContext` class.

When you create a component that extends `GenericService`, you should be aware of the following:

- The method `doStartService` is called after Nucleus creates the component, installed it into the naming hierarchy, set all of its property values from the properties file, and added all of its event listeners from the properties file. Your component must override `doStartService` to perform any required initialization—for example, create server ports and start threads. If initialization problems occur, the method can throw a `ServiceException`.
- The method `doStopService` is called when the service stops. The component must override this method to stop any processes that were started by this component—for example, close open file descriptors and server ports, stop any threads that this component started, and remove any event listeners that it added. However, the service should be prepared to start up again, possibly with new configuration values. When it restarts, the component is notified by calling `doStartService`.
- `GenericService` contains an implementation of `atg.nucleus.logging.ApplicationLogging`, thereby providing your service with a simple way to log events. For example, your service might log an error like this:

  ```
  catch (SomeException exc) {
    if (isLoggingError ()) {
      logError ("Something went terribly wrong", exc);
    }
  }
  ```

  The `logError` call might or might not include an exception. There are similar calls for the `Error`, `Warning`, `Info`, and `Debug` logging levels. See the Logging and DataCollection chapter.
- `GenericService` includes a default servlet to use for ATG Dynamo Server Admin. If your component wishes to use a different servlet to display information about itself in the Administration Interface, it should override `createAdminServlet` to create the servlet that should be used. See Customizing the ATG Dynamo Server Admin Interface.

- `GenericService` implements a method called `getAbsoluteName` that returns the absolute name of the component as seen in the Nucleus namespace.
- `GenericService` implements a couple of methods for resolving names in the Nucleus namespace. The method `resolveName` resolves a name relative to the component's parent. For example:

  ```
  resolveName ("Pricer")
  ```

  returns a pointer to the component named `Pricer` that is a child of the component's parent. The method tries to create the component from a configuration file if it cannot be found. `GenericService` also implements the `ComponentNameResolver` interface, so that you can pass the `resolveName` method a `ComponentName` key string, as well as a component's real relative name. See ComponentName.

Your component might not need to use all or any of this functionality. You should, however, be aware that all of these functions are available if your component extends `GenericService`.

Note: Be sure not to have a `getName` or `setName` method in a component that subclasses `GenericService`. If you do, your component's Nucleus pathname can become confused.

GenericContext

The `GenericService` class implements all useful Nucleus interfaces, except those that allow it to contain other services: `NameContext` and `NameContextBindingEventSource`.

Nucleus includes a subclass of `GenericService` that goes the extra step and implements those two interfaces. The subclass is called `GenericContext`. If your component extends `GenericContext`, it gets all the benefits of `GenericService`, and it can contain arbitrary objects by calling `putElement`. Those objects appear in the Nucleus namespace as children of your component.

Validateable

A component can implement an interface called `atg.nucleus.Validateable`, indicating that it knows how to check the validity of its own state. Nucleus automatically recognizes components implementing this interface and periodically calls `validateService` on these components to check their validity. If that method throws an `atg.nucleus.ValidationException`, Nucleus prints the exception and immediately shuts down.

Components that extend `GenericService` or `GenericContext` do not automatically implement `Validateable`.

Nucleus API

Earlier sections of this chapter focus on the various interfaces used by naming and Nucleus services. Very little is said about Nucleus itself, because Nucleus itself has little functionality. It exists solely to find components in the hierarchy and create them if they do not exist. This function is offered in the following method:

```
public Object resolveName (String name,
                           NameContext nameContext,
                           boolean create)
```

The name is the name of the component to find. If the name is a relative name, the specified nameContext is the NameContext that is used as a starting point of the name resolution. The create flag determines if Nucleus should attempt to create the component if it cannot be found.

Nucleus also determines the policies for describing the namespace. For example, the fact that forward slash (/) is used to separate the elements of a name is a policy issue implemented by Nucleus, not a general characteristic of NameContext. These policies are exposed in the following methods:

```
public String getAbsoluteName (NameContextElement element);
```

This returns the absolute name of a component in the Nucleus namespace. The component must implement NameContextElement in order to be examined by this method.

```
public String getAbsoluteElementName (NameContextElement element,
                                      String name)
```

This returns the absolute name of a hypothetical child of the specified element. For example, if the element is /services/servers, and the name is HttpServer, this returns /services/servers/HttpServer.

In order for a component to use these functions, the component must have a pointer to Nucleus. If the component implements ServiceListener, the component is passed a pointer to Nucleus when the component is initialized. Nucleus also acts as the root of the name hierarchy.

For more information, see the description of the atg.nucleus.Nucleus class in the *ATG API Reference*.

Dynamic Beans

Dynamic beans extend the standard JavaBeans specification, where a Java object can expose different sets of properties, that are determined at runtime. Unlike Java Beans, any object can be treated as a dynamic bean whose properties can be determined at runtime. This mechanism, for example, allows the Profile object to be used in a DSP tag like the following, though there is no getCity() method in the Profile object.:

```
<dsp:valueof bean="Profile.city"/>
```

Dynamic beans lets you use the Profile class without having to extend it to include as JavaBean properties all the profile attributes your application requires. Dynamic beans can be used in two broad sets of cases, where:

- All beans of a class have the same properties, but the properties are not known at runtime. The Profile object is an example.

- Different instances of a class have different properties from each other. For example, in a `Hashtable` or a `Map`, the properties are whatever keys are in the specific instance you are looking at.

A dynamic bean can be of any class, and need not implement any special interfaces. Before you can access a dynamic bean's properties, an implementation of `DynamicPropertyMapper` must be registered for the bean's class, one of the bean's superclasses, or one of the bean's interfaces. `DynamicPropertyMappers` are registered by default for several classes and interfaces that are most commonly used as dynamic beans, as listed in the next section, Registered DynamicBeans and Dynamic Types. You can register such an implementation by calling `DynamicBeans.registerPropertyMapper()`. See Registering DynamicBeans for more detailed information. After this has been done, you can use the methods `DynamicBeans.getPropertyValue()` and `DynamicBeans.setPropertyValue()` to access dynamic properties of objects belonging to the registered class or interface. This indirect approach permits existing classes like `java.util.Hashtable` or interfaces like `java.sql.ResultSet` to be treated as dynamic beans. If no `DynamicPropertyMapper` is registered, these methods simply access the object's regular JavaBean properties.

For example, because `atg.userprofiling.Profile` is registered as a dynamic bean, one way to access the `Profile.city` value from Java is:

```
String city = (String) DynamicBeans.getPropertyValue(profile, "city");
```

`DynamicBeans` also has `getSubPropertyValue()` and `setSubPropertyValue()` methods, which take a hierarchy property name of the form `propertyName1.subPropertyName2.subSubPropertyName3`. For example:

```
String city = (String) DynamicBeans.getSubPropertyValue(profile,
  "homeAddress.country");
```

Registering Dynamic Beans

You can treat any object as a dynamic bean if a `DynamicPropertyMapper` is registered for its class or for one of its superclasses or interfaces. A `DynamicPropertyMapper` looks like this:

```
public interface DynamicPropertyMapper {
  public Object getPropertyValue(Object pBean, String pName)
    throws PropertyNotFoundException;

  public void setPropertyValue(Object pBean, String pName, Object pValue)
    throws PropertyNotFoundException;

  public DynamicBeanInfo getBeanInfo(Object pBean)
    throws IntrospectionException;
}
```

You can register a `DynamicPropertyMapper` by calling `DynamicBeans.registerPropertyMapper()`. The registration needs to occur in a static initializer of some class that is guaranteed to load before the first use of the kind of dynamic bean being registered. For example, the `ProfileForm` class (which you can find at `<ATG10dir>/DPS/src/Java/atg/userprofiling/ProfileForm.java`) registers a `DynamicPropertyMapper` for the `ProfileFormHashtable` class like this:

```
static {
    DynamicBeans.registerPropertyMapper(ProfileFormHashtable.class,
                                new ProfileFormHashtablePropertyMapper());
  }
```

Example: DynamicPropertyMapper for Maps

For example, the ATG platform includes a `DynamicPropertyMapper` for `java.util.Map` objects, which is implemented like this:

```
package atg.beans;

import java.beans.*;
import java.util.*;

/**
 * <P>Implementation of DynamicPropertyMapper that can work with any
 * subclass of java.util.Map.
**/

public class MapPropertyMapper
implements DynamicPropertyMapper
{
  //-------------------------------------
  // CONSTRUCTORS
  //-------------------------------------

  public MapPropertyMapper() {
  }

  //-------------------------------------
  // METHODS
  //-------------------------------------

  /**
   * Gets the value of the dynamic property from the specified object.
   */
  public Object getPropertyValue(Object pBean, String pPropertyName)
       throws PropertyNotFoundException
  {
    Object value  = ((Map)pBean).get(pPropertyName);
    /*
     * There happens to be one property defined on the "map bean"
     * which we take advantage of in a few places due to the isEmpty
     * method. We treat this as a special case.
     */
    if (value == null && pPropertyName.equals("empty"))
      return ((Map) pBean).isEmpty() ? Boolean.TRUE : Boolean.FALSE;
    return value;
  }

  /**
   * Sets the value of the dynamic property from the specified object.
   */
  public void setPropertyValue(Object pBean, String pPropertyName, Object pValue)
  {
    ((Map)pBean).put(pPropertyName, pValue);
  }

  public DynamicBeanInfo getBeanInfo(Object pBean)
       throws IntrospectionException
}
```

The existence of such a `DynamicPropertyMapper` lets you insert values into Hashtables using a tag like this in a JSP:

```
<dsp:setvalue bean="MyComponent.hashtableProp.foo" value="23"/>
```

This tag looks up the `MyComponent` bean, calls the `getHashtableProp()`, method, and calls the dynamic bean method `setPropertyValue(hashtableProp, "foo", "23")`.

Multiple Property Mappers

What happens if an object has more than one superclass or interface with a registered `DynamicPropertyMapper`? If the definitions are at different levels of the inheritance hierarchy, the `DynamicPropertyMapper` of the most specific definition is used—that is, the one closest to the actual concrete class of the `DynamicBean`. If the object implements multiple interfaces that have `DynamicPropertyMappers`, the one that was declared first in the object class's implements clause wins. For example, an `atg.userprofiling.Profile` object has a `DynamicPropertyMapper` registered for itself, and for its `atg.repository.RepositoryItem` interface. The `Profile` `DynamicPropertyMapper` is used for it, rather than the `RepositoryItem` one.

DynamicBeanInfo

The `atg.beans` package also includes a useful interface named `DynamicBeanInfo`. A `DynamicBeanInfo` object allows a bean to be queried about what properties are available from the object, as well as other descriptive data about the bean. This is very similar to the standard `BeanInfo` objects of JavaBeans, except `DynamicBeanInfo` is based on the instance, not on the class. The `DynamicBeanInfo` allows user interfaces to show dynamically the available properties of an object.

The `DynamicBeanInfo` interface has the following methods:

```
public interface DynamicBeanInfo {
  public DynamicBeanDescriptor getBeanDescriptor();
  public boolean hasProperty(String pPropertyName);
  public String [] getPropertyNames();
  public DynamicPropertyDescriptor getPropertyDescriptor(String pPropertyName);
  public DynamicPropertyDescriptor[] getPropertyDescriptors();
  public boolean isInstance(Object pObj);
  public boolean areInstances(DynamicBeanInfo pDynamicBeanInfo);
}
```

`DynamicBeanDescriptor` is a subclass of the `java.beans.FeatureDescriptor` class, which includes human-friendly descriptive information about a bean. It includes methods like `getName()` and `getShortDescription()`. `DynamicPropertyDescriptor` is also a subclass of `FeatureDescriptor`, and allows individual properties of the properties, including a JavaBeans `PropertyEditor` for the property.

To implement this behavior in a `DynamicPropertyMapper`, use a `getBeanInfo()` method. For example, in the `MapPropertyMapper` discussed earlier, there is a `getDynamicBeanInfo` method that looks like this:

```
public DynamicBeanInfo getBeanInfo(Object pBean)
       throws IntrospectionException
  {
    // A Map acts as its own dynamic bean type, since every map
    // is different.
    return DynamicBeans.getBeanInfoFromType(pBean);
  }
```

Using DynamicPropertyMappers

When you call the `DynamicBeans.getPropertyValue()` method, the ATG platform first looks for a `DynamicPropertyMapper` registered for the class. If there is no other applicable `DynamicPropertyMapper`, the ATG platform uses the default `DynamicPropertyMapper` (registered for the `java.lang.Object` class) that treats its beans as regular JavaBeans, using Introspection to determine the dynamic properties and meta-data. Objects that have no other `DynamicPropertyMapper` registered default to this `DynamicPropertyMapper` and have dynamic properties that correspond one-to-one with their non-dynamic JavaBeans properties. This is a convenient way to manipulate regular JavaBeans easily without knowing their class in advance.

Here are some examples of how you can use the `DynamicBeans.getPropertyValue()` method to look up JavaBean properties:

- To call the `java.util.Date` method `getMonth`:

  ```
  Date d = new Date();
  Integer month = (Integer) DynamicBeans.getPropertyValue(d, "month")
  ;
  ```

There is no DynamicPropertyMapper registered for java.util.Date; the ATG platform uses the default DynamicPropertyMapper for java.lang.Object.

- To look up the DynamicPropertyMapper for DynamoHttpServletRequest and find the parameter named month:

  ```
  Integer month = (Integer) DynamicBeans.getPropertyValue(request, "m
  onth");
  ```

  In this example, the getPropertyValue() method calls the property mapper for DynamoHttpServletRequest, which treats each query or post parameter as a property.

- To look for the month attribute in the user Profile:

  ```
  Integer month = (Integer) DynamicBeans.getPropertyValue(getProfile(
  ),
    "month");
  ```

  The ATG platform uses the registered property mapper for atg.userprofiling.Profile.

Likewise, you can invoke lookup methods like those in these three previous examples, using <dsp:valueof> tags in JSPs:

```
<dsp:valueof bean="CurrentDate.timeAsDate.month"/>
<dsp:valueof bean="/OriginatingRequest.month"/>
<dsp:valueof bean="Profile.month"/>
```

Displaying Information from BeanInfos

The next example shows how you might write an ATG servlet bean that dumps information about any bean. This example servlet bean, named BeanInfoDroplet, sets a parameter named beaninfo with the values of the BeanInfo of the bean passed to it:

```
public class BeanInfoDroplet extends DynamoServlet {
  public void service(DynamoHttpServletRequest pRequest,
                      DynamoHttpServletResponse pResponse)
    throws ServletException, IOException {
    Object bean = pRequest.getObjectParameter("bean");

    try {
      pRequest.setParameter("beaninfo", DynamicBeans.getBeanInfo(bean));
    } catch (IntrospectionException ex) {
       logError(ex);
    }
    pRequest.serviceParameter("output", pRequest, pResponse);
  }
}
```

You might invoke the example `BeanInfoDroplet` servlet bean in a JSP like the following example. You pass the name of the bean in the bean input parameter. This page then displays the `beaninfo` parameter in a table:

```
<%@ taglib uri="/dspTaglib" prefix="dsp" %>
<dsp:page>

<html>
<head><title>BeanInfo</title></head>
<body><h1>BeanInfo</h1>

<dsp:droplet name="BeanInfoDroplet">
  <dsp:param bean="/atg/userprofiling/Profile" name="bean"/>
  <dsp:oparam name="output">
    <b><dsp:valueof param="beaninfo.name"/></b><p>
    <i><dsp:valueof param="beaninfo.shortDescription"/></i><p>

    <dsp:droplet name="ForEach">
      <dsp:param name="array" param="beaninfo.propertyDescriptors"/>
      <dsp:oparam name="outputStart">
        <table>
        <tr>
          <td>Property</td>
          <td>Type</td>
          <td>Value</td>
          <td>Readable?</td>
          <td>Writable?</td>
          <td>Required?</td>
          <td>Expert?</td>
          <td>Hidden?</td>
          <td>Preferred?</td>
          <td>Description</td>
        </tr>
      </dsp:oparam>
      <dsp:oparam name="output">
        <tr>
        <td><dsp:valueof param="element.name"/></td>
        <td><dsp:valueof param="element.propertyType.name"/></td>
        <td>
          <dsp:valueof param='<%="bean." +
            request.getParameter("element.name")%>'/>
        </td>
        <td><dsp:valueof param="element.readable"/></td>
        <td><dsp:valueof param="element.writable"/></td>
        <td><dsp:valueof param="element.required"/></td>
        <td><dsp:valueof param="element.expert"/></td>
        <td><dsp:valueof param="element.hidden"/></td>
        <td><dsp:valueof param="element.preferred"/></td>
        <td><dsp:valueof param="element.shortDescription"/></td>
        <tr>
      </dsp:oparam>
      <dsp:oparam name="outputEnd">
        </table>
      </dsp:oparam>
    </dsp:droplet>
  </dsp:oparam>
</dsp:droplet>


</body></html>

</dsp:page>
```

Dynamic Types

Often you might like to get access to information about a DynamicBean that has not yet been instantiated. For instance, you might have a DynamicBean based on JDBC ResultSets. You want to know what properties a ResultSet for some query might have. Using the above techniques, there is no way to do this; where would the DynamicBeanInfo come from?

You might have a Query class or interface, which describes a query that generates a ResultSet when executed. It would be nice to have a way to get a DynamicBeanInfo from the Query without executing it. We'd like to use the Query (apart from its other functions) as a dynamic type: it can provide information about the dynamic beans that it is capable of generating.

Dynamic beans provides an interface called DynamicBeanTyper. It contains a single method:

```
public DynamicBeanInfo getBeanInfoFromType(Object pDescription)
        throws IntrospectionException;
```

The purpose of this method is to return a DynamicBeanInfo from an object (such as the imagined Query) that plays the role of a dynamic type. You register a DynamicBeanTyper by calling DynamicBeans.registerBeanTyper(Class, DynamicBeanTyper). The class parameter is the class of a dynamic type, not the class of a DynamicBean. In this example, it is Query.class.

After the example DynamicBeanTyper is registered, the static method DynamicBeans.getBeanInfoFromType(Object) can be used to obtain a DynamicBeanInfo for any Query.

One final, useful twist: instances of java.lang.Class—that is, static types—act as dynamic types. In other words, there is a DynamicBeanTyper registered for Class.class. Its function is to return a DynamicBeanInfo that describes an instance of the given class, as analyzed by JavaBeans introspection. You might, for instance, call DynamicBeans.getBeanInfoFromType(Date.class), and the result is a DynamicBeanInfo describing an instance of Date. This is the same result you get by calling DynamicBeans.getBeanInfo() on an instance of Date.

Registered DynamicBeans and Dynamic Types

By default, the ATG platform registers the following classes and interfaces automatically as `DynamicBeans`:

- `java.lang.Object`
- `java.util.Map`
- `atg.repository.RepositoryItem`
- `atg.userprofiling.Profile`
- `atg.servlet.DynamoHttpServletRequest` (parameters are treated as properties)
- `org.w3c.xml.Node` (attributes and bean properties are treated as properties)

The following are automatically registered as dynamic types:

- `java.lang.Class` (introspection determines properties)
- `java.util.Map` (treated as a prototype instance of a `Map` bean)

Customizing the ATG Dynamo Server Admin Interface

The ATG platform includes ATG Dynamo Server Admin, an HTML-based interface that lets you inspect individual components at runtime. The default interface displays a component by listing its contained children and the values of its properties. Individual components can override this default behavior and provide their own servlets to act as the interface for that component. These custom servlets can also subclass the default servlet, so that it looks like the normal default servlet, possibly with some additional information.

In order for a component to define a custom administration interface, it must implement `atg.nucleus.AdminableService`. When the administration interface is asked to display a particular component, it first checks to see if the component implements `AdminableService`. If it does, Nucleus calls `getAdminServlet` to get the servlet that displays the component's administration interface, then passes the call off to that servlet. If the component does not implement `getAdminServlet`, Nucleus uses the default administrative servlet to display the component and its properties in the Component Browser.

The default administrative servlet is called `atg.nucleus.ServiceAdminServlet`. It contains a number of methods that can be overridden, as well as a number of methods that perform useful functions, such as formatting object values into HTML.

Creating Administration Servlets

`GenericService` already implements `AdminServlet` by creating an instance of `ServiceAdminServlet` when asked for the administration servlet. Subclasses of `GenericService` should override `createAdminServlet`. For example:

```
protected Servlet createAdminServlet ()
  {
    class AdminServlet extends ServiceAdminServlet {
      public AdminServlet (Object pService) {
        super (pService, getNucleus ());
      }

      protected void printAdmin (HttpServletRequest pRequest,
                                 HttpServletResponse pResponse,
                                 ServletOutputStream pOut)
           throws ServletException, IOException
      {
        pOut.println ("<h1>This is my special admin section</h1>");
      }
    }

    return new AdminServlet (this);
  }
```

This implementation uses an inner class that extends `ServiceAdminServlet`. The inner class overrides `printAdmin` to print out a line of HTML. The default servlet calls `printAdmin` to insert service-specific HTML between the listing of child components and the listing of property values. This is the easiest way for a service to provide its own administrative servlet.

Formatting Object Values

When a custom administrative servlet prints its output, you might want it to format object values into HTML output. The default administrative servlet `ServiceAdminServlet` provides a method that can be called by subclasses to do this formatting:

```
protected String formatObject (Object pObject,
                               HttpServletRequest pRequest);
```

This formatting method takes several steps to format the object:

- It checks whether the object is a component that implements `NameContextElement`. If so, it returns an anchor tag that contains a link to that component.
- It checks whether the object implements `atg.nucleus.ValueFormatter`. If so, it calls the object’s `formatValue()` method and returns the result.
- If the object is of type `java.lang.Class`, it returns an anchor tag that contains a link to the API documentation for that class.
- If the object is of type `java.lang.File`, it returns an anchor tag that points to that file.
- Otherwise, it uses the object’s `toString()` method.

The default administrative servlet uses this method to format the values of the properties that it displays.

ValueFormatter

As mentioned above, objects can customize their HTML representations in ATG Dynamo Server Admin by implementing `atg.nucleus.ValueFormatter`. This interface has two methods:

```
public String formatValue ();
public String formatLongValue ();
```

If you use the Component Browser, you might notice that property values can take on two forms. In the main page that lists all properties, only the short form of the value is shown. But when you then click on the property, the property is shown to you in its own page. On this page, the long form of the value is shown.

For example, the short form of a Hashtable entry might simply declare that it is a Hashtable, while the long form might display all the keys and values in the Hashtable.

Spring Integration

Spring is an open source component framework. Like Nucleus, Spring is based on JavaBeans.

The ATG platform lets you integrate existing Nucleus-based and Spring-based applications. For example, if you have a Spring-based web application that needs to access a property of an ATG user profile, you can use the integration to enable that.

The integration includes two classes:

- `atg.nucleus.spring.NucleusResolverUtil` enables Spring configurations to refer to Nucleus components
- `atg.nucleus.spring.NucleusPublisher` enables Nucleus components to refer to Spring components

NucleusResolverUtil

The `NucleusResolverUtil` class contains a single static `resolveName` method, which attempts to resolve the specified Nucleus path. Because Spring is unaware of Nucleus component scope, `NucleusResolverUtil` first attempts to resolve the name in the current request of the current thread (which should succeed if the component is request- or session-scoped) and if that fails, it then attempts to resolve the name in the global Nucleus scope.

To make a Nucleus component available in Spring, you declare it in your Spring configuration XML file. For example, to resolve the current user profile as a Spring component:

```
<bean name="/Profile" class="atg.nucleus.spring.NucleusResolverUtil"
    factory-method="resolveName" singleton="false">
  <constructor-arg value="/atg/userprofiling/Profile"/>
</bean>
```

Note: Nucleus components that do not have global scope should be specified with the `singleton` attribute set to `false`. If singleton is set to `true`, Spring caches the component, which can result in the wrong instance of a request- or session-scoped component being exposed.

NucleusPublisher

The `NucleusPublisher` class publishes a Spring configuration (that is, a Spring `ApplicationContext`) to a Nucleus path. The `NucleusPublisher` appears in the specified location as a Nucleus `NameContext` (a Nucleus folder) containing the JavaBeans from the Spring `ApplicationContext`. You can view these Spring components in the Component Browser in ATG Dynamo Server Admin.

For example, you can have the `NucleusPublisher` publish an `ApplicationContext` to `/atg/spring/FromSpring` by including the following in the Spring configuration XML:

```
<bean name="/NucleusPublisher" class="atg.nucleus.spring.NucleusPublisher"
    singleton="true">
  <property name="nucleusPath">
    <value>/atg/spring/FromSpring</value>
  </property>
</bean>
```

This enables Nucleus components to refer to Spring components in this `ApplicationContext`. For example, a Spring component called `SpringBean` has this Nucleus address:

```
/atg/spring/FromSpring/SpringBean
```

Because the `NucleusPublisher` itself is a Spring component, it can be referred to within Nucleus as:

```
/atg/spring/FromSpring/NucleusPublisher
```

The `NucleusPublisher` class is not in the main ATG CLASSPATH, but is included in a separate JAR file, `<ATG10dir>/DAF/spring/lib/springtonucleus.jar`. `NucleusPublisher` requires access to the Spring classes, so the `springtonucleus.jar` must be added to the `WEB-INF/lib` directory of the web application containing the Spring configuration to be exported.

Note: The Nucleus `NameContext` created by the `NucleusPublisher` is not be available until the web application containing the Spring configuration has been started, so any Nucleus components that depend on Spring components must be started up after `NucleusPublisher`. Therefore, `NucleusPublisher` has an `initialServicePaths` property, which can be configured with the paths of Nucleus components to start up after `NucleusPublisher` has published the Spring `ApplicationContext`. This property must be configured through the Spring configuration XML file, not through a Nucleus `.properties` file.

Naming Conflicts

Spring names can contain slash (/) characters, which are not legal in Nucleus names (because they are used as path separators in Nucleus addresses). Therefore, when the `NucleusPublisher` publishes Spring components to Nucleus, it replaces each slash character in a component's name with a period. For example, a Spring component named `/my/spring` is named `.my.spring` in Nucleus.

If this character substitution results in multiple components having the same Nucleus name, the names are differentiated by adding `-2` to the name of the second component, `-3` to the third, and so on. For example, if a Spring `ApplicationContext` has components named `.my.spring`, `/my/spring`, and `/my.spring`, their Nucleus names are `.my.spring`, `.my.spring-2`, and `.my.spring-3`.

3 Developing and Assembling Nucleus-Based Applications

Including ATG's Nucleus component framework in your J2EE application gives you access to the personalization, commerce, and content administration features of the ATG platform in applications that are portable across application servers. For example, the Quincy Funds demo application works this way.

To facilitate development, the ATG platform includes an application assembly tool that you can use to build J2EE enterprise archive (EAR) files that run Nucleus. All necessary Nucleus classes and configuration are included in the EAR file, so you can simply deploy the application and run it. You can even deploy the application on a machine that does not have an ATG installation.

In this chapter

This chapter describes how to develop and assemble J2EE applications that include Nucleus. It includes these sections:

Developing Applications

To develop Nucleus-based applications, you can use ATG's development tools such as the ATG Dynamo Server Admin. However, to access these tools, you must connect to an already-running Nucleus-based application. Therefore, when you install the ATG platform, the ATG installer automatically builds and deploys the Quincy Funds demo. When you run your application server, Quincy Funds starts up automatically, and you can connect to it, as described in the *ATG Installation and Configuration Guide*.

Development Mode and Standalone Mode

You can create two types of EAR files with the ATG application assembler, development and standalone. Both types contain all ATG classes needed for the application.

The primary difference is in where the configuration files for Nucleus components are stored:

- In development mode, the application draws its Nucleus configuration information from the ATG installation.
- In standalone mode, the application stores its configuration in the EAR file itself.

The two types of EAR files also differ in how they handle output files such as log files and state information.

Use development mode during the development process, when you are making frequent changes to the application, and standalone mode only when you deploy your production site. Later, if you need to update your production site, make changes in your ATG installation and then reassemble and redeploy the standalone EAR file.

Configuration File Sources and ATG-INF

The main differences between development-mode and standalone-mode EAR files are found in the `WEB-INF/ATG-INF` directory of the `atg_bootstrap.war` J2EE module. In this directory, both types of EAR files have a file called `dynamo.env`, which defines ATG application environment properties.

In a development-mode EAR, the `dynamo.env` file contains the following entry, which tells the application to get its configuration from the `.properties` and XML files in the ATG installation:

```
atg.dynamo.use-install=true
```

You can make configuration changes without having to rebuild the EAR file.

In a standalone EAR file, instead of using the ATG installation properties, the `ATG-INF` directory contains directories corresponding to the ATG modules included in the application. These directories store the configuration data for the modules, making the EAR file totally self-contained.

Additionally, in a standalone EAR the `atg_bootstrap.war/WEB-INF/ATG-INF` directory contains a `home/localconfig` subdirectory, which replicates the `localconfig` directory in the ATG installation. This `localconfig` directory is added to the application's configuration path, so that any settings stored in `localconfig` in the ATG installation are also used by the application.

Therefore, you can deploy a standalone EAR on a machine that does not have an ATG installation, although your ability to modify the application without rebuilding the EAR file (which requires an ATG installation) is limited.

Output Files and ATG-Data

Development-mode EAR files write output files to the ATG installation. For example, log files are written by default to the `<ATG10dir>/home/logs` directory.

Standalone EAR files cannot write to the ATG installation, inasmuch as it might not exist. When a standalone EAR file first starts up, it creates an `ATG-Data` directory. The `ATG-Data` directory contains subdirectories `logs`, `pagebuild`, and `data`, which correspond to directories of the same name in `<ATG10dir>/home`. The `ATG-Data` directory also contains a `localconfig` subdirectory that is the last entry in the configuration path. This directory is used for any configuration files written out by the EAR file itself. You can add properties files to this directory to modify the application configuration for debugging purposes; otherwise, you should make these changes in the ATG installation, then rebuild and redeploy the EAR file.

By default, the `ATG-Data` directory is created in the current working directory of the Java process. If the JVM starts up in different directories, it creates `ATG-Data` directories there as well. To specify a location for the directory, set the `atg.dynamo-data-dir` system property. For example:

```
java <arguments>  -Datg.dynamo.data-dir=/var/data/ATG-Data/
```

These directory structures apply only if you are using the default ATG server. For information about using non-default servers, see Using a Non-default ATG Server later in this chapter.

Nucleus-Based Application Structures

Each EAR file built by `runAssembler` includes the following J2EE modules:

- `atg_bootstrap_ejb.jar`: This module contains a single session EJB. The EJB's class loader is used to load the ATG classes needed by the application. These classes are stored in the `lib` directory (at the top level of the EAR file).
- `atg_bootstrap.war`: This module starts up Nucleus and runs the servlet pipeline.

In addition, the EAR file typically includes one or more additional J2EE modules (generally WAR files), containing the web applications that actually run your site. For example, the `QuincyFunds.ear` file includes a web application named `quincy.war`.

The EAR file can optionally include ATG Dynamo Server Admin, which is packaged as a web application named `atg_admin.war`. See Including ATG Dynamo Server Admin for more information.

Assembling Applications

To assemble your application to run on your application server, use the `runAssembler` command-line script. This script takes a set of ATG application modules and assembles them into an EAR file (or the equivalent exploded directory structure).

The basic syntax of the command follows this format:

```
runAssembler earfilename -m module-list
```

For example, if you develop your application as an application module named `MyApp`, and you want to assemble an EAR file that includes your application plus the `DSS` and `DPS` modules, use the following command:

```
runAssembler MyApp.ear -m MyApp DSS
```

You do not need to specify the DPS module, because the application assembler examines the manifest file for each application module specified, and includes any modules that the specified modules depend on. The DSS module requires the DPS module, so it is included without being specified.

In addition to the modules containing core ATG functionality (such as the DSS module), you can also include ATG demos and reference applications in your EAR file. For example, to assemble an EAR file that includes the Quincy Funds demo, include DSSJ2EEDemo in the list of module for the runAssembler command.

When runAssembler creates an EAR file, unless otherwise instructed, it copies only CLASSPATH entries, the configuration path entries, and J2EE modules. To include other module files, specify them via the ATG-Assembler-Import-File attribute in the module's META-INF/MANIFEST.MF file, as shown in this example from the DafEar.base module:

```
ATG-Required: DSS DAS-UI
ATG-Config-Path: config/dafconfig.jar
ATG-Web-Module: j2ee-components/atg-bootstrap.war
ATG-EJB-Module: j2ee-components/atg-bootstrap-ejb.jar
ATG-Class-Path: ../Tomcat/lib/classes.jar ../WebLogic/lib/classes.jar
 ../WebSphere/lib/classes.jar lib/classes.jar
Name: ../WebLogic
ATG-Assembler-Import-File: True
```

In addition, to include a standalone WAR file in an application you must provide runAssembler with the WAR file's URL and context root. If not provided, runAssembler creates a unique name based on the directory where it found the WAR file. To provide the URL and context root, within your WAR file, create a META-INF/MANIFEST.MF file as follows:

```
Manifest-Version: 1.0
ATG-Enterprise-Nucleus: True
ATG-Module-Uri: atg_bootstrap.war
ATG-Context-Root: /dyn
ATG-Assembler-Priority: -1
```

For a list of modules included in the ATG installation, see Appendix D: ATG Modules. This appendix also contains information about how to access the demos and reference applications.

You can also include ATG Dynamo Server Admin in your EAR file, so you can monitor and change settings in your application. See the Including ATG Dynamo Server Admin section later in this chapter for information.

Command Options

The runAssembler command takes a number of command-line flags that you can use individually or in combination to alter the output of the command. These can be supplied as follows:

```
runAssembler [-liveconfig] [cmd-options]
   earfilename
   [-layer layer-name] [-standalone]
   -m module-list
```

Ordering is significant with respect to the following command options:

- `-liveconfig` must follow the `runAssembler` command
- `-standalone` must precede `-m`
- `-layer` must precede `-standalone` and `-m`

The following table describes `runAssembler` options.

Option	**Description**
`-add-ear-file` *`filename`*	Includes the contents from an existing EAR file in the assembled EAR file. See Including An Existing EAR File, below.
`-classesonly`	Instead of assembling a complete EAR file, creates a JAR file that collapses the JAR files and directories in the CLASSPATH into a single library.
`-collapse-class-path`	Collapses all JAR files and directories in the CLASSPATH into a single JAR file in the assembled EAR file. By default, these JAR files and directories are copied separately to the EAR file's `lib` directory, and placed in the EAR file's CLASSPATH.
`-context-roots-file` *`filename`*	Specifies a Java properties file whose settings are used to override the default context root values for any web applications included in the assembled EAR file. To specify the context root for a web application in this properties file, add a line with the following format: *`module-uri=context-root`* where *`module-uri`* is the module URI of the web application, and *`context-root`* specifies the context root to use.
`-displayname` *`name`*	Specifies the value for setting the display name in the `application.xml` file for the assembled EAR file.
`-distributable`	JBoss requires that your `web.xml` file include the `<distributable/>` tag when running in a cluster. The `-distributable` flag for runAssembler automatically adds the tag to all `web.xml` files in an EAR as it assembles. If the `<distributable/>` tag is not included, JBoss does not enable session failover.
`-jardirs`	Collapses all classpath entries into individual jar files.

`-layer`	Enables one or more named configuration layers for the application. This switch can take multiple arguments, each representing a named configuration layer. This option must immediately precede the `–m` switch.
`-liveconfig`	Enables the `liveconfig` configuration layer for the application. For more information, see the *ATG Installation and Configuration Guide*.

`-nofix`	Instructs `runAssembler` not to fix servlet mappings that do not begin with a leading backslash.
	
	By default the `runAssembler` command attempts to fix any servlet mappings defined in a web.xml that do not start with a leading forward slash (/). JBoss does not allow servlet mappings without starting slashes, so `runAssembler` converts this:
	
	`<url-pattern>foo.bar.baz</url-pattern>`
	
	to
	
	`<url-pattern>/foo.bar.baz</url-pattern>`
	
	The `runAssembler` command does ignore mappings that begin with * or with white space. For example, it does not change this:
	
	`<url-pattern>*.jsp</url-pattern>`

`-overwrite`	Overwrites all resources in the existing EAR file. By default, resources in the assembled EAR are only overwritten if the source file is newer, to reduce assembly time.
`-pack`	Packs the assembled EAR file into the archived J2EE enterprise archive format. By default, the EAR is assembled in an exploded, open-directory format.

`-prependJars`	Includes the comma separated list of jar files on the classpath. This attribute is useful for applying hotfixes. For example:
	
	`runAssembler –`
	`prependJars hotfix1.jar,hotfix2.jar myEarFile.ear –`
	`m DCS`
	
	Note: Special characters appearing in jar file names might cause that file to be ignored. When naming files, use only alphanumeric characters and the underscore.

`-run-in-place`	JBoss only; this option should be used only in development environments. When assembling the EAR file with `-run-in-place`, `runassembler` does not copy `classes.jar` included in the application, but refers to the ATG installation for these resources. If during development you make changes to `classes.jar` in the ATG installation, you do not need to reassemble the EAR in order to see the changes.

`-server` *`servername`*	Specifies the value for the `atg.dynamo.server.name` variable for this EAR file. This variable determines which ATG server directory is used for configuration and logging. If this option is not specified, the default server directory is used. For more information about ATG server directories, see Using a Non-Default ATG Server in this chapter.
`-standalone`	Configures the assembled EAR in standalone mode, so that it contains all application resources, including Nucleus configuration files, and does not refer to the ATG installation directory. By default, the EAR is assembled in development mode, where only classes, libraries, and J2EE modules are imported to the EAR file, and Nucleus configuration and other resources are used directly from the ATG install directory.

Specifying Configuration Layers on Server Startup

In some situations, you might want to deploy the same EAR file across various servers, where each server has different configuration requirements. For example, two servers that run ATG Content Administration both need to deploy to a staging site; however, one requires asset preview, while the other does not. You can assemble a single EAR file for both servers by setting the `-layer` switch as follows:

```
-layer preview staging
```

When you start the applications, you can disable asset preview on one by explicitly specifying the configuration layers you wish to activate on it—in this case `staging` only. For example, you can run the JBoss run script so it activates only the `staging` configuration layer as follows:

```
-Datg.dynamo.layers=staging
```

Including an Existing EAR File

When you assemble an EAR file, the application modules you specify can contain EAR files, WAR files, and other J2EE entities. The application assembler automatically includes these, as well as the Nucleus resources used by the application modules themselves.

You can also have the application assembler include an existing EAR file that is not part of a module. To do this, invoke the `runAssembler` command, and use the `-add-ear-file` flag to specify the EAR file to include. For example:

```
runAssembler -add-ear-file resources.ear MyApp.ear -m MyApp DSS
```

To include more than one existing EAR file, use a separate `-add-ear-file` flag before the name of each EAR file.

Note: Only use this option to include existing EAR files that are not part of ATG application modules. To include an EAR file that is part of an ATG application module, just include the module name in the list of modules specified with the `-m` flag. Including the whole module ensures that any Nucleus resources that the existing EAR file depends on are included in the assembled EAR file.

Including Web Services

You can include any of ATG's prepackaged web services in an assembled EAR file by including the module that contains the desired services. For example, to include the Commerce services, specify the `DCS.WebServices` module when you invoke the `runAssembler` command. To include web services you created through the Web Service Creation Wizard, use the `runAssembler` flag `-add-ear-file` to specify the EAR file that contains the service.

Using a Non-Default ATG Server

If you run the application assembler without specifying a server name, the resulting application uses the default ATG server. This means that the application gets site-specific configuration settings from standard configuration directories, such as `<ATG10dir>/home/localconfig`.

If the application is assembled in development mode, the `localconfig` directory in the ATG installation is set as the last entry in the application's configuration path. If the application is in standalone mode, the assembler copies that directory from the ATG installation into the `atg_bootstrap.war/WEB-INF/ATG-INF` directory in the EAR file (as described in Development Mode and Standalone Mode). This directory is added to the configuration path just before the `ATG-Data/localconfig` directory, which is the last entry in the configuration path . For output files, the application uses the `logs` and `pagebuild` subdirectories in the `<ATG10dir>/home` directory (in development mode) or the `ATG-Data/home` directory (in standalone mode).

If you configure additional ATG server instances (see the *ATG Installation and Configuration Guide* for information), you can build your application with a particular server's configuration. The effect of specifying a server differs depending on whether you are assembling a development-mode or standalone EAR file.

Specifying a Server for a Development-Mode EAR File

To build a development-mode EAR file that uses a server, run the application assembler with the `-server` flag. For example:

```
runAssembler -server myServer MyApp.ear -m MyApp DSS
```

The `localconfig` directory for the server is appended to the application's configuration path. This means that the last two entries in the configuration path are the standard `localconfig` (`<ATG10dir>/home/localconfig`) followed by the server-specific `localconfig`—for example, `<ATG10dir>/home/servers/myServer/localconfig`.

For output files, the application uses the `logs` and `pagebuild` subdirectories in the `<ATG10dir>/home/servers/`*`servername`* directory. You should not use the same server for more than one EAR file. If multiple EAR files are using the same output directories, errors or deadlocks can result.

Specifying a Server for a Standalone EAR File

To build a standalone-mode EAR file that uses a non-default server, you can run the application assembler with the `-standalone` and `-server` flags. For example:

```
runAssembler –standalone –server myServer MyApp.ear –m MyApp DSS
```

Note: If your production environment is clustered, do not specify the ATG server when you build the EAR. Instead, omit the `–server` flag and create a single EAR that can be deployed to all servers in the cluster. When you run the application, supply the Java argument `-Datg.dynamo.server.name=` *`server-name`* to specify the named instance of each server.

There are four `localconfig` directories at the end of the application's configuration path. They appear in the following order:

- `atg_bootstrap.war/WEB-INF/ATG-INF/localconfig` (a copy of `<ATG10dir>/home/localconfig`)
- `ATG-Data/localconfig`
- `atg_bootstrap.war/WEB-INF/ATG-INF/home/servers/`*`servername`*`/localconfig` (a copy of `<ATG10dir>/servers/`*`servername`*`/localconfig`)
- `ATG-Data/servers/`*`servername`*`/localconfig`

For output files, the application uses the `logs` and `pagebuild` subdirectories in the `ATG-Data/servers/`*`servername`* directory. You should not use the same server for more than one EAR file. If multiple EAR files are using the same output directories, errors or deadlocks can result.

Including ATG Dynamo Server Admin

To be able to administer your application through ATG Dynamo Server Admin, you must specify the `DafEar.Admin` module when you run the application assembler. For example:

```
runAssembler QuincyFunds.ear –m DSSJ2EEDemo DafEar.Admin
```

Note: The `DafEar.Admin` module must precede any custom modules that are included in the module list.

The ATG Dynamo Server Admin user interface is included in the EAR file as a web application, `atg_admin.war`. This WAR file includes all pages that comprise ATG Dynamo Server Admin . Its `web.xml` file declares a `NucleusProxyServlet` that points to the Nucleus component `/atg/dynamo/servlet/adminpipeline/AdminHandler`:

```
<servlet>
  <servlet-name>AdminProxyServlet</servlet-name>
  <servlet-class>atg.nucleus.servlet.NucleusProxyServlet</servlet-class>
  <init-param>
    <param-name>proxyServletPath</param-name>
    <param-value>/atg/dynamo/servlet/adminpipeline/AdminHandler</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>AdminProxyServlet</servlet-name>
  <url-pattern>/admin/*</url-pattern>
</servlet-mapping>
```

In this configuration, the ContextPath is `/dyn` and the `ServletPath` is `/admin`, so the URL for accessing the ATG Dynamo Server Admin server is:

```
http://{hostname}:{port}/dyn/admin/
```

To access ATG Dynamo Server Admin, use the listen port for your application server. For example, if an ATG application runs on JBoss with a listen port of 8080, you can access ATG Dynamo Server Admin on your machine at `http://localhost:8080/dyn/admin`.

Note: Your database must be running in order for you to use the Administration UI. If necessary, you can override this requirement by copying `/atg/dynamo/security/AdminUserAuthority.properties` from the `<ATG10dir>\DAS\config\config.jar` file to `<ATG10dir>\home\localconfig\atg\dynamo\security`.

Changing the ATG Dynamo Server Admin Login

By default, ATG Dynamo Server Admin requires password authentication to run. The initial user name and password for this server are set as follows:

User Name: **admin**
Password: **admin**

Users who are members of the Systems Administrators group can modify the user name and password through the ATG Control Center. If the default administrative account has been removed or you lose the password, you can reset the user name and password to the default values.

For more information about modifying the default user name and password and creating user accounts and groups, see *Managing Access Control* in this guide.

Logging Attempts to Access the Administration Server

As a precaution, you might want to log information about attempts to log in to ATG Dynamo Server Admin, such as the IP address from which the login originated. Logging this information can alert you to unauthorized attempts to gain access to your Nucleus-based applications, or simply allow you to track usage of the Admin UI.

The `/atg/dynamo/servlet/adminpipeline/AuthenticationServlet` component has two properties that control what login information is logged:

- `logFailedAuthentications`: If `true`, logs failed attempts to log in (defaults to `true`).

- `logSuccessfulAuthentications`: If `true`, logs all successful authentications (defaults to `false`). Setting this to true causes a great deal of logging information, because each page request is logged.

Invoking the Application Assembler Through an Ant Task

The ATG platform includes two Ant tasks to simplify invoking the application assembler from within Ant build files:

- `CreateUnpackedEarTask` builds an unpacked (exploded) EAR file
- `PackEarFileTask` takes an unpacked EAR file and packages it in a packed (unexploded) EAR file

The classes for these Ant tasks are available as part of your ATG platform installation, in the JAR file located at `<ATG10dir>/home/lib/assembler.jar`. This library contains all the supporting classes necessary to run the tasks.

CreateUnpackedEarTask

Class: `atg.appassembly.ant.CreateUnpackedEarTask`

Description

This Ant task invokes the application assembler, which combines ATG platform libraries, Nucleus component configuration, J2EE applications, and J2EE application components to create a single J2EE application, in the form of an unpacked (open-directory) EAR file.

Required Task Parameters

Attribute	Description
`destinationFile`	Specifies the path of the EAR file to be created
`dynamoModules`	Specifies the ATG modules to include in the EAR file, as a comma-delimited string
`dynamoRoot`	Specifies the path to the ATG installation directory

Optional Task Parameters

Attribute	Description
`addEarFile`	Specifies an existing EAR file whose contents are added to the assembled EAR file.

Attribute	Description
`collapseClasspath`	If `true`, the JAR files and directories in the CLASSPATH are collapsed into a single JAR file in the assembled EAR file. The default is `false`.
`contextRootsFile`	Specifies a Java properties file to be used to override the context-root values for any web applications included in the assembled EAR file. In this properties file, each line has the format: *`module-uri=context-root`* This assigns the specified context root to the web application indicated by the module URI.
`displayName`	Specifies the value to be used for the `<display-name>` tag in the `application.xml` file in the assembled EAR file.
`displayVariable`	Specifies the X Window System variable declaring where any X display should be sent. For example, `:0.0`)
`dynamoEnvPropsFile`	Specifies a file that supplies ATG environment properties to be added to `dynamo.env` in the assembled EAR file.
`layer`	Enables one or more named configuration layers for the application. This switch can take multiple arguments, each representing a named configuration layer. This option must immediately precede the `–m` switch.
`liveConfig`	If `true`, `liveconfig` mode is enabled in the assembled EAR file. The default is `false`.
`overwrite`	If `true`, overwrites an existing EAR file; if false, stops processing if the EAR file already exists. The default is `false` (do not overwrite.

`prependJars`	Includes the comma separated list of jar files on the classpath. This attribute is useful for applying hotfixes. For example:
	
	`runAssembler –`
	`prependJars hotfix1.jar,hotfix2.jar myEarFile.ear –m DCS`
	
	Note: Special characters appearing in jar file names can cause that file to be ignored. When naming files, it is best to use only alphanumeric characters and the underscore.

`serverName`	If set, specifies the ATG server (for `localconfig`, etc.) to be used by the assembled EAR file. If unset, the default server is used.

Attribute	Description
standalone	If true, the EAR file is created in standalone mode, where all necessary resources are imported into the resulting EAR file, and the EAR file does not reference the ATG installation directory. If false, a development-mode EAR file is created, where Nucleus configuration and other runtime resources are used directly from the ATG installation. The default is false (development mode).

Example

To use `CreateUnpackedEarTask` in an Ant build file, you must first declare it, using the `taskdef` element:

```
<taskdef name="assemble-jar"
         classname="atg.appassembly.ant.CreateUnpackedEarTask"
         classpath="C:/ATG/ATG10.0.1/home/lib/assembler.jar">
```

You can then create a target that assembles an ATG application EAR file:

```
<target name="create-quincy-ear">
   <-- It's a good idea to delete any old directories
       before assembling... -->
   <delete dir="QuincyFundsEar"/>

   <assemble-jar dynamoRoot="c:/ATG/ATG10.0.1 "
                 dynamoModules="DSSJ2EEDemo,DafEar.Admin"
                 destinationFile="QuincyFundsEar"
                 overwrite="true" />
</target>
```

PackEarFileTask

Class: `atg.appassembly.ant.PackEarFileTask`

Description

This Ant task takes an EAR file in exploded (open-directory) format, and packs it into the archive-file format specified by the J2EE standard.

Required Task Parameters

Attribute	Description
`sourceFile`	Specifies the staging directory containing the unpacked application.
`destinationFile`	Specifies the filename for the packed EAR file.

Example

To use `PackEarFileTask` in an Ant build file, you must first declare it, using the `taskdef` element:

```
<taskdef name="pack-ear"
         classname="atg.appassembly.ant.PackEarFileTask"
         classpath="C:/ATG/ATG10.0.1/home/lib/assembler.jar">
```

This example is a target that uses `CreateUnpackedEarTask` to create the application in unpacked format, and then uses the `PackEarFileTask` to pack the application in an EAR file:

```
<target name="create-quincy-ear">
   <-- It's a good idea to delete any old directories
       before assembling... -->
   <delete dir="QuincyFundsEar"/>

   <assemble-jar dynamoRoot="C:/ATG/ATG10.0.1"
                 dynamoModules="DSSJ2EEDemo,DafEar.Admin"
                 destinationFile="QuincyFundsEar"
                 overwrite="true" />

   <pack-ear sourceFile = "QuincyFundsEar"
             destinationFile = "Quincy.ear" />

   <-- Delete the open directory, and keep the packed EAR file. -->
   <delete dir="QuincyFundsEar"/>

</target>
```

4 Working with Application Modules

ATG products are packaged as separate application modules. An application module groups application files and configuration files into a discrete package for deployment. Application modules exist in the ATG installation as a set of directories defined by a manifest file. When you assemble an application, these modules are analyzed to determine the CLASSPATH, configuration path, and inter-module dependencies.

Application modules provide the following core features:

- Application components are packaged in a simple, modular directory structure.
- Modules can reference other modules on which they depend, and which can be automatically loaded.
- The correct class path, configuration path, and main Java class for a given set of modules are dynamically calculated for both client and server.
- Updated modules are automatically downloaded from server to client.

The basic module and update mechanisms rely on the standard JAR manifest format; for information, see the Oracle Web site.

In this chapter

This chapter includes the following topics:

Using ATG Modules

An ATG application runs in your application server's J2EE container. When you assemble the application, resources such as class libraries, web applications, and EJB modules are copied from the ATG installation into an EAR file. You deploy and run the EAR file on the application server. If you make changes to resources in your ATG installation, you might need to recreate and redeploy the EAR file to see the changes reflected.

Development versus Standalone Modes

If you assemble an EAR file in development mode, the application draws configuration information for Nucleus components from properties files in the ATG installation. Most of these properties files are stored in application modules; for example, the directory `<ATG10dir>/DSSJ2EEDemo` has a `config` subdirectory, which includes properties files for Nucleus components used by the Quincy Funds application.

If you assemble an EAR file in standalone mode, the properties files that configure your Nucleus components are imported into the EAR file and stored in directories that correspond to the ATG modules included in the application.

Creating an Application Module

When you develop a new application, package your class and configuration files into a new module in the ATG installation. To create a module:

1. Create a module directory within your ATG installation.
2. Create a `META-INF` directory within the module directory.
3. Create a manifest file named `MANIFEST.MF` in the module's `META-INF` directory. The manifest contains metadata that describes the module.

Each of these steps is described in more detail in the sections that follow.

Application Module Directory Structure

An ATG installation must have a single module root directory that contains all available module directories. The module root corresponds to the `<ATG10dir>` or `DYNAMO_ROOT` directory in the ATG installation.

Each application module has its own subdirectory under the module root. The application module name and module directory name are identical. If a module is not directly under the module root, the module name uses this format:

`parent-dir.module-dir`

For example, a module located at `<ATG10dir>/MyModule` is named `MyModule`, and a module located at `<ATG10dir>/CustomModules/MyModule` is named `CustomModules.MyModules`.

Each module directory and its subdirectories can contain any number of module resource files, in any desired organization. Module resources can include any files that you wish to distribute, including:

- EAR files for J2EE applications
- WAR files for web applications
- EJB-JAR files for Enterprise JavaBeans
- JAR files of Java classes

- Platform-dependent libraries
- HTML documentation
- Configuration files

Application Module Manifest File

A module must include a `META-INF` directory containing the manifest file `MANIFEST.MF`. For example, the manifest used by the DPS module is located at:

```
<ATG10dir>/DPS/META-INF/MANIFEST.MF
```

Manifest Attributes

You can set a number of manifest attributes to specify the module's environment and resources:

Manifest attribute	Description
`ATG-Assembler-Class-Path`	The CLASSPATH to use for the assembled EAR file. This attribute overrides attribute `ATG-Class-Path`. If no value is set, `ATG-Class-Path` is used.
`ATG-Assembler-Skip-File`	Files to exclude from the assembled EAR file. By excluding unnecessary files, you can reduce the size of the EAR file.
`ATG-`*`cfgName`*`Config-Path`	The path to the directory that contains the configuration files for the named configuration layer *`cfgName`*. These configuration files are appended to the configuration path when the named configuration layer is enabled by the `-layer` switch. Paths are relative to the module's root directory.
`ATG-Class-Path`	A space-delimited set of paths to module resources that contain classes required by this module, either `.jar` files or directories. As each module is processed, the ATG platform adds the `ATG-Class-Path` value to the beginning of the EAR file's CLASSPATH. Paths are relative to the module's root directory. These libraries and directories of classes are imported into the `lib` directory of the assembled EAR file, and added to the CLASSPATH of the EAR-level class loader.
`ATG-Client-Class-Path`	A space-delimited set of paths to module resources that contain classes required by the module's client-side features.
`ATG-Client-Help-Path`	A space-delimited set of paths to module resources that provide JavaHelp help sets to the module's client application. For example, the `DPS` module has this value for the ATG Control Center help set: `help/dps_ui_help.jar`

Manifest attribute	Description
`ATG-Config-Path`	A space-delimited set of paths to module resources that provide Nucleus configuration files needed by the module's application components. These can be `.jar` files or directories.
`ATG-EAR-Module`	One or more EAR files for this ATG module whose J2EE modules are to be included in the assembled application.
`ATG-EJB-Module`	One or more EJB modules for this ATG module to include in the assembled application.
`ATG-LiveConfig-Path`	A space-delimited set of paths to module resources that provide Nucleus configuration files. These configuration files are appended to the configuration path when you enable the `liveconfig` configuration layer. Paths are relative to the module's root directory. For more information about the `liveconfig` configuration layer, see the *ATG Installation and Configuration Guide*.
`ATG-Nucleus-Initializer`	A Nucleus preinitializer required for this module, the name of a class that implements interface `atg.applauncher.initializer.Initializer`. This class must be in the CLASSPATH specified by attributes `ATG-Class-Path` or `ATG-Assembler-Class-Path`. Before Nucleus starts in the assembled application EAR file, the `initialize()` method for each of the named classes is invoked.
`ATG-Required`	A space-delimited set of module names, specifying modules on which this module depends. If you specify this module when you run the application assembler, the modules listed here also are included in the application. When the application starts, the manifests of the modules listed here are processed before the current module's, in least-dependent to most-dependent order . **Note:** You should usually set this attribute to include `DSS`.
`ATG-Web-Module`	One or more web applications to include in the assembled application for this ATG module.

Individual Module Resource Entries

A module's manifest can contain one or more entries for individual resources. The manifest must include an entry for each resource that is automatically downloaded to the client. For example:

```
Name: help/das_ui_help.jar
ATG-Client-Update-File: true
ATG-Client-Update-Version: 3.0.2 build 42
```

At a minimum, a resource entry must set `ATG-Client-Update-File` to true. The following table shows all attributes that can be set for each resource:

Resource attribute	Description
`ATG-Assembler-Import-File`	Optional, specifies whether to copy the resource into EAR files
`ATG-Client-Update-File`	Required, this attribute must be set to true to enable auto-downloading of the file.
`SHA-Digest`	Optional, a SHA digest of the file in Base-64 form, permitting checking of the resource's version and integrity.
`MD5-Digest`	Optional, a MD5 digest in Base-64 form, permitting checking of the resource's version and integrity.
`ATG-Client-Update-Version`	Optional, a version string that specifies the resource's version, overriding consideration any `SHA-Digest` or `MD5-Digest` hash digest attributes that might be present.

Including ATG-Web-Module

If you include `ATG-Web-Module` in your module's `MANIFEST.MF`, you must declare the `ATG-context-root` and `ATG-Module-uri` for those web modules in the web application's own `MANIFEST.MF` file; otherwise, those settings are not correct.

1. Create a `META-INF/MANIFEST.MF` file in the top level of the web application's WAR file.
2. Add the following lines:

```
Manifest-Version: 1.0
ATG-Module-Uri: myModule.war
ATG-Context-Root: /myContextRoot
```

The next time the application is assembled, it uses the assigned values.

Accessing Module File Resources

Application module code on the server can access file resources relative to the module root by using the `appModuleResource` expression. The value of this expression is evaluated at runtime, so it always maps to a location relative to the current location of the module.

The syntax for this expression is:

```
{appModuleResource?moduleID=module-name&resourceURI=relative-path}
```

For example, to set the value of a property named `myFile` to a file called `resource.txt` in the `lib` subdirectory of the module `MyModule`:

```
myFile={appModuleResource?moduleID=MyModule&resourceURI=lib/resource.txt}
```

Creating an Application Module JAR File

You can package an application module as a JAR file, using the module's manifest file as the JAR file's manifest, by invoking the `jar` command with the `m` option flag. For example, if you have a module's resources and manifest file in your `/work/MyModule` directory, you can make a JAR file named `mymodule.jar` for the module with this command:

```
jar cvfm mymodule.jar MANIFEST.MF -C /work/MyModule .
```

Packaging the module into a single file makes it easier to copy the module to multiple ATG installations. To add the module to an ATG installation, unjar the file in the `<ATG10dir>` directory; this installs the module in the appropriate place in the directory structure. You might need also to copy HTML files into your web server document root directory.

Adding Modules to the ATG Control Center

If you want your module to appear in the Components window of the ATG Control Center, add the following elements to your module:

- CONFIG.properties File
- Module Component
- ModuleManager Entry

The content of each of these elements is described below.

CONFIG.properties File

Create a `CONFIG.properties` file in your module's `config` subdirectory. This file labels and configures a configuration layer in the ATG Control Center. You can set the following properties:

Property	Description	Example
`defaultForUpdates`	If `true`, changes made to components are made in this configuration layer by default. This should typically be set to `false`, so that `localconfig` remains the default update layer.	`false`

`readOnly`	If `true`, locks this configuration layer, preventing users from editing it.	`true`
`name`	The display name to use in the ATG Control Center.	`Cheese Grater`
`module`	The module name.	`CheeseGrater`

Module Component

In the `config/atg/modules` directory of your module, create a Nucleus component of class `atg.service.modules.Module`. Give it the same name as your module name. This component has a properties file like this:

```
$class=atg.service.modules.Module
moduleName=CheeseGrater
```

This creates a component with a Nucleus address of `/atg/modules/CheeseGrater` of class `atg.service.modules.Module` with the `moduleName` property set to `CheeseGrater`.

ModuleManager Entry

Add your module's name to the modules property of the `/atg/modules/ModuleManager` component. Create a `ModuleManager.properties` file in your module's `config/atg/modules` directory with the following property:

```
modules+=CheeseGrater
```

This adds the `CheeseGrater` component to the modules list of the `ModuleManager` and presents your module in the ATG Control Center's list of modules.

Launching a Client Application Against Remote Modules

The client version of the ATG Control Center or any other ATG client application is launched against the set of ATG modules in a development-mode EAR file running on a remote server.

The client launcher displays a dialog box that prompts for the hostname, port, username, and password. The hostname and port are used to resolve an RMI service on the server, which in turn provides the client with HTTP URLs for remote downloading of module and system resources. The username and password are used for basic HTTP authentication of the download.

The client then:

1. Obtains the list of modules running on the server, and their server-side manifests.

2. Determines the `ATG-Client-Update-Directory` manifest attribute of the main module(s). This is taken as the name of a client-side module root subdirectory, immediately below the directory where the client launcher started. It is typically version-specific (such as 10.0.1) to permit the client to maintain multiple downloaded versions of the software at the same time.
3. For each server module, examines whether that module exists in the client module-root and loads its client-side manifest if it is found.
4. Compares the entries in the server manifest marked as `ATG-Client-Update-File: true` with entries in the client manifest of previously downloaded files. It determines a list of all module resources that need downloading, either because they do not exist on the client or because the client versions or digests do not match those on the server.
5. Requests the user to confirm downloading, if the list is not empty.
6. Downloads the appropriate server resources.

When this procedure completes successfully, the client-side module root is in effect a mirror of the server-side module root, but a selective one, covering only resources that were marked for downloading in the manifest. The client application is then launched within a custom class loader, using the same set of modules as on the server, but employing the `ATG-Client-Class-Path` and `ATG-Client-Main-Class` manifest attributes instead of `ATG-Class-Path` and `Main-Class`.

Synchronization of Client and Server

For each file to be downloaded:

1. The client manifest entry is deleted and the client manifest is saved. This ensures that a crash does not result in a manifest inaccurately stating that a partially downloaded file exists with such-and-such a digest hash.
2. The file is downloaded from the server archive to its corresponding location in the client archive.
3. The server manifest entry is copied to the client manifest and the client manifest is saved again.

After the files are downloaded, the main attributes of the server manifest are copied to the client manifest on each connection attempt, regardless of whether any files are downloaded or not. Because files are never removed from the client side, a single client can work with servers whose manifests have a common superset of files, without constant removal and re-updating of files.

5 Creating and Using ATG Servlet Beans

You can use the `dsp:droplet` tag to embed the contents of one JSP file into another HTML file. Most applications, however, require some way to generate JSP from a Java object. The `dsp:droplet` tag also lets you do this, by referring to a Nucleus component rather than another JSP file. In this case, the JSP generated by the Nucleus component is embedded into the JSP. A Nucleus component used in this way is called an ATG servlet bean.

The *ATG Page Developer's Guide* describes how to use the servlet beans that are included with the ATG platform.

In this chapter

This chapter explains how to create and use your own servlet beans, and includes the following topics:

- Creating Custom Servlet Beans
- Using Custom Servlet Beans with the ATG Control Center
- Resolving Component Names

Creating Custom Servlet Beans

By using the `dsp:droplet` tag with a bean attribute, you can embed the output of a Java servlet (an ATG servlet bean) in a JSP. JSP generation is performed according to the standard Java Servlet specifications. This means that ATG servlet beans must implement the `javax.servlet.Servlet` interface.

ATG servlet beans have access to all the parameters visible to the `dsp:droplet` tag. ATG servlet beans also have access to APIs that give them the same kind of `valueof` functionality available to JSPs.

This section explains how to create and use ATG servlet beans in JSPs. The next section explains the functions of ten standard ATG servlet beans that are included in the ATG platform, which can handle common design issues in a web application.

This section covers the following topics:

- Simple ATG Servlet Bean Example
- ATG Servlet Beans and Servlets

- Passing Parameters to ATG Servlet Beans
- Displaying Open Parameters in ATG Servlet Beans
- Setting Parameters in ATG Servlet Beans
- Local Parameters
- Separating JSP Code and Java Code
- Object Parameter Values
- Property Parameter Values
- Processing Servlet Beans
- Limitations in Custom Servlet Beans

Simple ATG Servlet Bean Example

The following is a simple example of using an ATG servlet bean to produce JSP code within a page.

Create the following ATG servlet bean in a file named `DSBTest.java` and save it to `<ATG10dir>/home/locallib`:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import atg.servlet.*;

public class DSBTest extends DynamoServlet {
  public DSBTest () {}
  public void service (DynamoHttpServletRequest request,
                      DynamoHttpServletResponse response)
       throws ServletException, IOException
  {
    ServletOutputStream out = response.getOutputStream ();
    out.println ("<h1>I am generated by a java object.</h1>");
  }
}
```

Define `<ATG10dir>/DAS/lib/classes.jar` in your `CLASSPATH`, and compile `DSBTest.class`. Create an instance of it as a Nucleus component:

1. In the Components by Path task area, create a folder called `test`.
2. Click New Component.
3. Select the Generic Component template and click OK.
4. In the Class field, enter the class name, `DSBTest`.
5. Save the component to the `test` directory, and click Finish.

Now you can access the component from a JSP. In the J2EE Pages task area, create a JavaServer Page named `dsbtest.jsp` in a running application. For example, if you are running the ATG Adaptive Scenario Engine, save this file in the `QuincyFunds` application.

Add this text to `dsbtest.jsp`:

```
<%@ taglib uri="/dspTaglib" prefix="dsp" %>
<dsp:page>

<html>
<head>
  <title>DSBtest</title>
</head>

<body>
<h1>DSB Test </h1>

<p>From a java object:

<p>Did it work?

</body>
</html>

</dsp:page>
```

Now embed the `DSBTest` servlet bean:

1. Move the insertion point after the `<p>From a java object:` line.
2. Click Insert Servlet Bean.
3. Click By Path.
4. Select the `/test/DSBTest` component and click OK.
   The Document Editor inserts the following tag:

   ```
   <dsp:droplet name="/test/DSBTest">
   </dsp:droplet>
   ```

5. Click Preview to save and view the `/test/dsbtest.jsp` file. When you access this page, you should see the output of the ATG servlet bean inserted into the JSP.

Notice how this example uses the `dsp:droplet` tag. When you embed an ATG servlet bean, you use a `name` attribute that specifies the name of the Nucleus component to embed. Nucleus finds the component, makes sure that it implements `Servlet`, then hands the request to the component to satisfy the `dsp:droplet` tag.

To make the `/test/DSBTest` component visible in the Dynamic Element Editor, you can use the `dsp:importbean` tag to import the component into the scope of your page.

ATG Servlet Beans and Servlets

In the previous example, the `DSBTest` servlet was a subclass of `DynamoServlet`. Its `service` method took `DynamoHttpServletRequest` and `DynamoHttpServletResponse` objects as parameters.

These interfaces are subclasses of standard servlet interfaces. `DynamoHttpServletRequest` extends `HttpServletRequest` and adds several functions that are used to access ATG servlet bean functionality. The `DynamoHttpServletResponse` extends `HttpServletResponse` and also adds a couple of useful functions.

The `DynamoServlet` class implements the `javax.servlet.Servlet` interface. It passes requests to the `service` method by passing a `DynamoHttpServletRequest` and `DynamoHttpServletResponse` as parameters.

The `DynamoServlet` class extends `atg.nucleus.GenericService`, which allows an ATG servlet bean to act as a Nucleus component. This means that the ATG servlet bean has access to logging interfaces, can be viewed in the Component Browser, and has all the other advantages of a Nucleus service.

A servlet invoked with the DSP tag library `<dsp:droplet name=...>` tag need not be a subclass of `DynamoServlet`; it only needs to implement the `javax.servlet.Servlet` interface. Any servlets that you write for other application servers can be used inserted in JSPs with DSP tag library tags. However, those servlets lack access to all other facilities available to Nucleus components. If you write ATG servlet beans from scratch, the `DynamoServlet` class provides an easier starting point.

Passing Parameters to ATG Servlet Beans

You can pass parameters to an ATG servlet bean just as you do to an embedded JSP. For example, the following passes the `storename` parameter with a value of `Joe's Hardware` to the ATG servlet bean `/test/DSBStoreTest`:

```
<%@ taglib uri="/dspTaglib" prefix="dsp" %>
<dsp:page>

<html>
<head><title>DSB Store Test</title></head>
<body><h1>DSB Store Test</h1>

<dsp:droplet name="/test/DSBStoreTest">
  <dsp:param name="storename" value="Joe's Hardware"/>
</dsp:droplet>

</body></html>

</dsp:page>
```

You can access these parameters with the `getParameter()` call found in `HttpServletRequest`, as in the following ATG servlet bean, `DSBStoreTest.java`, which prints out a header that includes the `storename` parameter:

```
public void service (DynamoHttpServletRequest request,
                     DynamoHttpServletResponse response)
     throws ServletException, IOException
{
  String storename = request.getParameter ("storename");
  if (storename == null) storename = "No-Name's";

  ServletOutputStream out = response.getOutputStream ();
  out.println ("<h1>Welcome to " + storename + "</h1>");
}
```

If the parameter is not defined, `getParameter()`returns null, so your code should be prepared for that situation.

Your ATG servlet bean can access any parameter that is visible to the `dsp:droplet` tag calling the ATG servlet bean. This includes parameters passed directly to the ATG servlet bean, as well as any parameters visible to the JSP containing the `dsp:droplet` tag.

Because `getParameter()` can return only Strings, this method should be used only for parameters that are not open parameters. The next sections describe a more general way to deal with parameters, both simple and open.

Displaying Open Parameters in ATG Servlet Beans

The previous section mentioned that, although it is possible to pass open parameters to ATG servlet beans, those parameters should not be read with the standard `getParameter()` method. In fact, it is unlikely that your ATG servlet bean wants to see the actual value of an open parameter. In most situations, an ATG servlet bean wants to output the value of an open parameter. To do this, use the `serviceParameter` method of the request, as in the following example:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import atg.servlet.*;

public class DSBTest2 extends DynamoServlet {
  public DSBTest2 () {}
  public void service (DynamoHttpServletRequest request,
                       DynamoHttpServletResponse response)
       throws ServletException, IOException
  {
    ServletOutputStream out = response.getOutputStream ();
    out.println ("Here's the value of the parameter 'storename':");
    request.serviceParameter ("storename", request, response);
  }
}
```

The serviceParameter method obtains the value of the given parameter and displays it.

To demonstrate this, save the previous code sample as DSBTest2.java and compile it into a class file and create a corresponding Nucleus component in much the same way as you did in Simple ATG Servlet Bean Example. Create dsbtest2.jsp with the following contents:

```
<%@ taglib uri="/dspTaglib" prefix="dsp" %>
<dsp:page>

<html>
<head>
  <title>Storename Test</title>
</head>

<body bgcolor="#ffffff">
  <h1>Storename Test</h1>

<dsp:droplet name="/test/DSBTest2">
  <dsp:oparam name="storename">
    <h1>Joe's Hardware</h1>
  </dsp:oparam>
</dsp:droplet>

</body>
</html>

</dsp:page>
```

Preview this page to see how it looks when processed and compiled.

The serviceParameter prints out any parameter including simple Strings and open parameters. If an open parameter includes dynamic elements such as dsp:valueof and dsp:droplet tags, those elements are also generated dynamically.

The serviceParameter method returns a Boolean value indicating whether the specified parameter was found or not (true if the parameter was found).

Setting Parameters in ATG Servlet Beans

When your ATG servlet bean displays an open parameter, that open parameter can itself contain dynamic elements such as dsp:valueof and dsp:droplet tags. As always, when a dynamic element contained in an open parameter is displayed, it draws from the list of visible parameters to display its own dynamic elements.

The parameters visible to those elements are the same as the parameters visible to the dsp:droplet tag. For example:

```
<%@ taglib uri="/dspTaglib" prefix="dsp" %>
<dsp:page>

<html>
<head>
  <title>Store Test</title>
</head>

<body bgcolor="#ffffff">
  <h1>Store Test</h1>

<dsp:droplet name="/test/DSBTest2">
  <dsp:param name="goods" value="Lingerie"/>
  <dsp:oparam name="storename">
    <h1>Joe's <dsp:valueof param="goods"></dsp:valueof></h1>
  </dsp:oparam>
</dsp:droplet>

</body>
</html>


</dsp:page>
```

In this example, the `storename` parameter includes a `dsp:valueof` element that displays the value of goods. The `DSBTest2` object can display this by calling `serviceParameter`. When it is displayed, the `dsp:valueof` tag looks through the visible parameters for a parameter called goods. Because that parameter is defined in the `dsp:droplet` call, that parameter is visible to the `dsp:valueof` tag and is therefore used. Remember that the parameter would be visible if it were defined as a top-level parameter, or if this page were itself included by some other `dsp:droplet` tag that defined the goods parameter.

The ATG servlet bean can also add or change parameters that are visible to displayed elements. This is done by calling `setParameter()`. For example, the ATG servlet bean can set the goods parameter in code:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import atg.servlet.*;

public class DSBTest3 extends DynamoServlet {
  public DSBTest3 () {}

public void service (DynamoHttpServletRequest request,
                     DynamoHttpServletResponse response)
     throws ServletException, IOException
{
  request.setParameter ("goods", "Golf Balls");
  request.serviceParameter ("storename", request, response);
}
}
```

The `setParameter` call brings the goods parameter with value `Golf Balls` into the list of parameters that are visible to displayed objects. If a goods parameter is already visible, this shadows the original definition of the parameter. The original definition of the parameter returns after this method finishes execution.

Local Parameters

You might want to create parameters that are visible only locally to an ATG servlet bean. In that case, use the `getLocalParameter(String paramName)` and `serviceLocalParameter(String paramName, ServletRequest, ServletResponse)` methods. These methods return only values that are defined in the current frame for this invocation of the ATG servlet bean. This includes parameters passed to the ATG servlet bean between the open and close `dsp:droplet` tags and parameters defined at the top level of the called ATG servlet bean.

For example:

```
<dsp:param name="notLocalForA" value="x"/>

<dsp:droplet name="A">
    <dsp:param name="localForA" value="y"/>
</dsp:droplet>
```

In this example `notLocalForA` is not local for the ATG servlet bean A, and `localForA` is local.

Local parameters are useful because they allow the ATG servlet bean to determine which parameters are defined for a particular ATG servlet bean call. Without local parameters, it can be easy to get into an infinite loop by nesting different ATG servlet beans, as the inner ATG servlet bean always sees all parameters defined to the outer one.

Separating JSP Code and Java Code

The previous sections showed how you can write ATG servlet beans that generate JSP code from Java code, while still being able to display parameters defined in a JSP and setting parameters for those displayed parameters.

These functions give you the ability to write applications that completely separate JSP formatting from Java functionality, which is one of the main goals of tag libraries. This separation is essential in large applications because it allows JSP designers and Java coders to work together and maintain autonomy.

As an example, consider the following ATG servlet bean that displays a list of numbers. Name this servlet bean `Counter.java` and create a class and component for it as described in Simple ATG servlet bean Example:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import atg.servlet.*;

public class Counter extends DynamoServlet {
public Counter () {}
public void service (DynamoHttpServletRequest request,
                     DynamoHttpServletResponse response)
     throws ServletException, IOException
{
  ServletOutputStream out = response.getOutputStream ();
  out.println ("<ul>");
  for (int i = 0; i < 10; i++) {
    out.println ("<li>This is number " + i);
  }
  out.println ("</ul>");
}
}
```

This ATG servlet bean might be invoked from a JSP like this:

```
<%@ taglib uri="/dspTaglib" prefix="dsp" %>
<dsp:page>

<html>
<head><title>Counter</title></head>
<body><h1>Counter</h1>

<dsp:droplet name="/test/Counter">
</dsp:droplet>

</body></html>

</dsp:page>
```

At first, this looks like a simple and workable solution. The problem is that the ATG servlet bean now contains formatting information. This formatting information is usually subject to change many times during the course of development. If every change requires a designer to ask a Java developer to change and recompile a Java class, the simple solution becomes an obstacle.

When you use ATG servlet beans, you can rewrite the previous example so that all of the JSP is removed from the Java object, while the functionality is retained by the Java object:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import atg.servlet.*;

public class Counter2 extends DynamoServlet {
public Counter () {}
public void service (DynamoHttpServletRequest request,
                     DynamoHttpServletResponse response)
     throws ServletException, IOException
{
  ServletOutputStream out = response.getOutputStream ();
  for (int i = 0; i < 10; i++) {
    request.setParameter ("number", new Integer (i));
    request.serviceParameter ("lineformat", request, response);
  }
}
}
```

This new ATG servlet bean has no formatting left in it at all. Instead, the ATG servlet bean relies on the formatting to be passed as a parameter called `lineformat`. It then uses the `number` parameter to set the value for each line. The ATG servlet bean is then invoked from a JSP like this:

```
<%@ taglib uri="/dspTaglib" prefix="dsp" %>
<dsp:page>

<html>
<head><title>Counter</title></head>
<body><h1>Counter</h1>

<ul>
<dsp:droplet name="/test/Counter2">
  <dsp:oparam name="lineformat">
    <li>This is number <dsp:valueof param="number"/>
  </dsp:oparam>
</dsp:droplet>
</ul>

</body></html>

</dsp:page>
```

Now all formatting information is concentrated in JSP files, making it much easier for a JSP developer to get at it.

Object Parameter Values

All parameter values described so far are either Strings, or open parameters (whose values are of type Servlet). It is possible for parameters to be assigned values that are of other types, such as Vectors, arrays, or any other Java type. Earlier, you saw how arbitrary objects can be assigned to parameters through the `DynamoHttpServletRequest.setParameter()` method.

Arbitrary objects can also be assigned to parameter values by attaching the parameter values to object properties through JSP files. For example:

```
<dsp:droplet name="/test/counter">
  <dsp:param bean="/test/Person.age" name="maxcount"/>
  <dsp:oparam name="lineformat">
    <li>This is number <dsp:valueof param="number"/>
  </dsp:oparam>
</dsp:droplet>
```

Here the parameter `maxcount` has been assigned a value from the age property of the `/test/person` component. Primitive types such as int, float, short, are converted automatically to the corresponding Java object types Integer, Float, Short, and so on. Because the age property is of type `int`, the resulting property value is of type `Integer`.

Parameters with arbitrary object values can be displayed using the `dsp:valueof` or `paramvalue=...` constructs, just as they are for String parameter values. You can also display arbitrary object values within an ATG servlet bean by calling `DynamoHttpServletRequest.serviceParameter()`.

AN ATG servlet bean often needs to obtain the value of an object parameter without actually displaying that parameter. For example, an ATG servlet bean might use the `maxcount` parameter to specify some sort of limit.

The `HttpServletRequest.getParameter()` method is not suitable for this because it can only access parameters that are of type String. To access arbitrary objects, you must use another method from `DynamoHttpServletRequest` called `getObjectParameter()`. For example:

```
public void service (DynamoHttpServletRequest request,
                     DynamoHttpServletResponse response)
     throws ServletException, IOException
{
  ServletOutputStream out = response.getOutputStream ();

  int maxcount = 0;
  Object maxcountval = request.getObjectParameter ("maxcount");
  if (maxcountval instanceof Integer)
    maxcount = ((Integer) maxcountval).intValue ();

  for (int i = 0; i < maxcount; i++) {
    request.setParameter ("number", new Integer (i));
    request.serviceParameter ("lineformat", request, response);
  }
}
```

In this example, the `maxcount` parameter, assigned to an integer property, is used to specify the upper bound for counting.

Property Parameter Values

The previous section demonstrated how a parameter can point to any object. Those objects might themselves have property values that you want to access from a JSP.

For example, say that you wanted to print the `age` and `name` properties of some object, but you do not know ahead of time what that object is. Presumably a pointer to that object is passed as a parameter—in this example, `currentPerson`.

The following code prints those parameter properties:

```
<dsp:valueof param="currentPerson.name"></dsp:valueof> is
<dsp:valueof param="currentPerson.age"></dsp:valueof> years old.
```

Notice how the `dsp:param` tag looks like it always has, except that instead of naming a parameter, the tag names a specific property of a parameter.

This form of the `dsp:param` tag can be used be used to set a parameter, using `param=..`, or update a parameter with another parameter value as in:

```
<dsp:setvalue param="currentPerson.name" paramvalue="user1"/>
```

This tag sets the first parameter, `currentPerson.name`, to the value in the second, `user1`. The `currentPerson` page parameter maps to a component: that component's name property takes the string value of `user1`. Earlier, you set `currentPerson` as follows:

```
<dsp:param name="currentPerson" bean="/db/personGetter.person">
```

The parameter can also be set through Java code, as outlined in the previous section.

Processing Servlet Beans

When a JSP executes a servlet bean, the `dsp:droplet` cycles through its code internally several times in order to arrange the servlet bean code in a manner that is cohesive with the expectations of open parameters.

Note: All references to `dsp:droplet` in this section describe the `dsp:droplet` tag or its class. The term servlet bean refers to a specific kind of bean implemented by a `dsp:droplet` tag.

Consider how the ATG platform processes this example:

```
<dsp:droplet name="/atg/dynamo/droplet/ForEach">
  <dsp:param name="array" bean="/samples/Student.subjects"/>
  <dsp:oparam name="output">
    <p><dsp:valueof param="element"/>
  </dsp:oparam>
</dsp:droplet>
```

1. The `dsp:droplet` tag is called.
2. `dsp:droplet` allows its body to be executed once. During that execution, the nested input parameter tags (in this case, just `array`) pass their information back to `dsp:droplet`, which uses it to construct a table of input parameter names (`array`) and values (`Reading`; `Writing`; `Arithmetic`). The open parameter tags are ignored during this phase.
3. `dsp:droplet` finds the servlet bean referenced by the `dsp:droplet` "name=" property (`forEach`) and calls the servlet bean's `service()` method with the input parameter values collected during step #2 (`Reading`; `Writing`; `Arithmetic`).
4. As the servlet bean executes, it halts calls to `setParameter` and `serviceParameter`, and instead records them as a list of `DropletActions`. These methods are organized in a manner that is readable by the open parameters that process them and are made available to open parameters for execution.
5. The `dsp:droplet` parses through each `setParameter` and `serviceParameter` method in `DropletActions` :
   - `setParameter` directs the `dsp:droplet` to set the specified request parameter to the recorded name (`ouput`) and value (`element`).
   - `serviceParameter` instructs the `dsp:droplet` to allow its body to be executed. This causes the related open parameter to run (`element` equals `Reading`; `Writing`; `Arithmetic`).
6. After the `dsp:droplet` finishes the `DropletActions` list, servlet bean execution ends.

Limitations in Custom Servlet Beans

The main limitation that you need to be aware of when you are creating servlet beans is that open parameters are not executed precisely when their `serviceParameter` is called. Instead, open parameters remain dormant until the servlet bean `service` method completes and the `dsp:droplet` tag begins reviewing the `DropletActions` as described in the previous section. Code your servlet beans to expect that the servlet bean's `service` method is not immediately followed by the execution of the open parameters `serviceParameter`.

The effects of this restriction have several side effects that might not be obvious, such as how nested open parameters interact with each other. See the following sections for details on these side effects.

Here are some general operations you should avoid:

- Setting a global or thread-state variable that is accessed by code invoked from an open parameter.
- Opening or closing a socket or JDBC result set that is accessed by code invoked from an open parameter.

- Replacing the output stream/print writer in the response with your own designed to capture the output of an open parameter.

Open Parameter Dependencies

A servlet bean's `service` method cannot depend on the effects of a nested open parameter. Because the JSP executes the open parameter after the `service` method completes, the `service` method cannot act on any results or changes produced by the open parameter.

For example, when a certain output open parameter throws an exception, a `service` method catches it and renders an `error` open parameter. This sequence does not operate successfully because the `service` method completes execution before the open parameter throws the exception.

A servlet bean's `service` method should not rely on values that are themselves determined during the execution of the open parameter. If an open parameter, for example, were to set the value of a profile attribute, you might think the service method can access that new value after the `serviceParameter` method has returned. Because the open parameter changes the value after the `service` method call completes, the `service` method is unaware of the open parameter change.

Similarly, a servlet bean's `service` method cannot manipulate the output rendered from an open parameter. For example, the servlet bean's `service` method might attempt to translate the value produced from an open parameter into another language. Again, the servlet bean's `service` method is processed before the other open parameter delivers the value so the translation does not occur.

Actions that Rely on Timely Open Parameter Processing

A servlet bean cannot perform arbitrary actions around an open parameter and expect the open parameter to be affected by the results of those actions. For example, a servlet bean might:

```
set profile property to "x"
call open parameter "a"
set profile property to "y"
```

This code executes as follows:

1. Set profile property to x.
2. Call open parameter a.
3. Set profile property to y.
4. Servlet bean code ends.
5. Execute open parameter a.

Because the open parameter is actually executed after the profile property is set to y, the open parameter never sees the profile property set to x.

Request parameters are an exception to this rule. When you set a request parameter on the ATG request object, that global parameter is visible to the open parameters within a given page. The record and play back mechanism in `dsp:droplet` permits interdependence between open parameters and request parameters.

Open Parameter as an Object

You cannot manipulate an open parameter as an object.

Using Custom Servlet Beans with the ATG Control Center

In order to use custom ATG servlet beans with the ATG Control Center, the following requirements apply:

- The class must extend `atg.servlet.DynamoServlet`.
- You must create a `BeanInfo` file that describes the servlet bean's parameters.

The `BeanInfo` file describes the servlet bean's parameters to the ATG Control Center. The ATG Control Center uses this information to provide guidelines for the parameters you need to set to make the servlet bean work properly. The ATG Control Center does not guarantee that these parameters are present or that they have values with the correct types. It is still up to the servlet bean to validate its input parameters. The ATG Control Center also uses the `BeanInfo` as a source of the information displayed in the servlet bean's information panel in the ATG Control Center Components editor.

A `BeanInfo` is a standard Java Beans mechanism through which a Java Bean can describe information about the features that it exposes—for example, its properties and events. ATG servlet beans extend this notion to include parameter descriptors that specify information about the parameters that the servlet bean uses in a JSP. If you do not explicitly create a `BeanInfo` class for your Java bean, a `BeanInfo` is generated for you by the Java Beans Introspector. When you do build a `BeanInfo` for a custom ATG servlet bean, it must describe all features of your Java Bean. You need to add descriptors for each property and event you want your bean to expose. For more information about how to construct a `BeanInfo`, see the JSDK documentation for Java Beans at `<`*`JSDK_dir`*`>/jdoc/java/beans/BeanInfo.html`. The next section describes how to augment a `BeanInfo` with `ParameterDescriptors` to describe servlet bean parameters.

Parameter Descriptors

Each `BeanInfo` has a single `BeanDescriptor` that is used to provide information about the Bean. The `BeanDescriptor` supports a list of named attributes that augment the standard Java Bean's information. You set these using the `BeanDescriptor`'s `setValue` method. ATG servlet beans look for a `paramDescriptors` attribute that contains an array of `atg.droplet.ParamDescriptor` objects. Each of these `ParamDescriptor` objects defines one of the parameters of your servlet bean expects to be provided. It defines the following information:

Argument	**Type**
Name of the parameter	String
Description of the parameter's function.	String
The Java class describing the parameter. For `oparam` parameters, you should specify the class to be of type `javax.servlet.Servlet` because that is how `oparam` parameters are compiled and represented.	String

Argument	Type
Whether or not this parameter is optional or required. Set this to `true` if the parameter is optional.	Boolean
Whether this parameter is local. Set this to `true` if the parameter is accessed with `getLocalParameter` or `serviceLocalParameterlocal`.	Boolean
If this `paramDescriptor` describes a parameter of type `javax.servlet.Servlet`—that is, an `oparam` parameter—list the `ParamDescriptors` that define which parameters are set by the servlet bean before it renders the `oparam` parameter. Otherwise, set this to `null`.	`ParamDescriptor []`

ParamDescriptor Example

For example, the following `paramDescriptor` describes a parameter named `numItems`. Its description is `number of times to call output`. It is described by the `Integer` class, is a required parameter, is not local, and is not an `oparam` parameter.

```
paramDescriptors[0] = new ParamDescriptor("numItems",
                      "number of times to call output", Integer.class,
                      false, false, null);
```

Defining the Component Category

The ATG Control Center Components by Module view organizes components according to their component category. You can set the component category of your custom servlet bean in its `BeanInfo`. For example, to set the category to Servlet Bean, use this `beanDescriptor`:

```
beanDescriptor.setValue("componentCategory", "Servlet Beans");
```

BeanInfo Example

To describe the parameters for a class called `YourServlet`, create a `YourServletBeanInfo.java` class like this:

```
import atg.droplet.ParamDescriptor;

public class YourServletBeanInfo extends java.beans.SimpleBeanInfo {
  static java.beans.BeanDescriptor beanDescriptor = null;

  public java.beans.BeanDescriptor getBeanDescriptor() {
    if (beanDescriptor == null) {
      ParamDescriptor [] paramDescriptors = new ParamDescriptor[2];
      ParamDescriptor [] outputDescriptors = new ParamDescriptor[1];

//This parameter is set before we service the output parameter.
```

```
        outputDescriptors[0] = new ParamDescriptor("index", "loop index (0-based)",
                                                Integer.class, false, false, null);

        paramDescriptors[0] = new ParamDescriptor("numItems",
                                                 "number of times to call output",
                                                 Integer.class, false, false, null);
        paramDescriptors[1] = new ParamDescriptor("output",
                                                  "rendered for each iteration",
                                                  DynamoServlet.class,
                                                  false, true, outputDescriptors);

        beanDescriptor = new BeanDescriptor(YourServlet.class);
        beanDescriptor.setShortDescription("A custom servlet bean.");
        beanDescriptor.setValue("paramDescriptors", paramDescriptors);
        beanDescriptor.setValue("componentCategory", "Servlet Beans");

      }
      return beanDescriptor;
    }
  }
```

Resolving Component Names

An ATG application performs name resolution for Nucleus components whenever it encounters component names in a tag. For example, the `<dsp:droplet name=` and `<dsp:valueof bean=...>` tags both specify component names that the ATG server needs to resolve to pointers to actual objects. In performing this name resolution, the server can create objects if they do not exist, and merge session and global namespaces automatically.

Nucleus also provides a way to connect objects to each other by naming those objects in properties files. For example, if the `Person` object needs a pointer to a `Talents` object, the `Person` object defines a property of type `Talents`, and specifies the component name of that `Talents` object as the value of that property in the properties file. Nucleus automatically resolves the name, creating the `Talents` object if necessary.

If you write your own servlet beans in Java, you might also want to resolve component names to Java objects. This functionality is provided by the `resolveName()` method of the `DynamoHttpServletRequest` class. The `resolveName()` method handles both absolute and relative names, and also implements the merged global and session namespaces.

For example, the following code obtains a pointer to the `/services/scheduler/Scheduler` component:

```
import atg.naming.*;
import atg.service.scheduler.*;
```

```
...

public void service (DynamoHttpServletRequest request,
                     DynamoHttpServletResponse response)
     throws ServletException, IOException
{
  Scheduler scheduler = (Scheduler)
    request.resolveName ("/services/scheduler/Scheduler");

...

}
```

Because resolution is a potentially expensive operation, you should consider caching the results of a name lookup of a global scope component, rather than requiring a name lookup be performed on every request.

There might be times when you want to look up an existing component, but you do not want to create an instance of the component if it does not already exist. In that case, use the `resolveName()` method with a `false` Boolean argument.

If you want to create a session-scoped or request-scoped component that looks up another session-scoped or request-scoped component, then add a property of type `DynamoHttpServletRequest` to your component. For example, to look up another request-scoped component, you might set a property called `request` as follows:

```
request=/OriginatingRequest
```

Your component can now call `getRequest().resolveName("`*`target-component`*`")`, in your component's `doStartService()` method, where *`target-component`* is the name of the component you are looking up. For instance, you can display the request locale in Java code with:

```
if (request.getRequestLocale() != null)
  out.print(request.getRequestLocale().getLocale());
```

You can use the URI of the request as the action attribute of a form tag in a JSP like this:

```
<dsp:getvalueof id="form0" bean="/OriginatingRequest.requestURI"
  idtype="java.lang.String">
  <dsp:form action="<%=form0%>"/>
</dsp:getvalueof>
```

If you are using a session-scoped component, the value of the `request` property becomes invalid upon completion of the current request. To work around this problem, add the following line at the end your `doStartService()` method:

```
setRequest(null);
```

Note: It is not standard practice to have a session-scoped component refer to a request-scoped value. This is a special case that you can use in a restricted way to access the request that creates your session-scoped component.

If you want to resolve the name of a Nucleus component from Java code that is not itself a Nucleus service, you must first initialize Nucleus with this construct:

```
Nucleus.getGlobalNucleus().resolveName("target component")
```

where `target component` is the name of the component you are looking up. Note that this construct works only for components with global scope.

You can also resolve names of Nucleus components using the Java Naming and Directory Interface (JNDI). The following example shows how you can use JNDI to access the `Scheduler` component:

```
String jndiName = "dynamo:/atg/dynamo/service/Scheduler";
Context ctx = new javax.naming.InitialContext ();
Scheduler s = (Scheduler) ctx.lookup (jndiName);
```

Before using these methods to resolve names, make sure that the functionality you want is not already provided by configuration files or servlet bean tags.

6 Working with Forms and Form Handlers

The *ATG Page Developer's Guide* describes how to use the form handlers that are provided with the ATG platform. This chapter describes form handler classes, and shows how you can modify and extend these to suit the specific requirements of your application. It also discusses other form processing tools.

In this chapter

This chapter includes the following sections:

- Form Handlers and Handler Methods explains how to create form handlers and handler methods for processing forms.
- Creating Custom Tag Converters explains how to create your own tag converters to parse and display values in a variety of formats.
- File Uploading describes how to create form elements and components that enable users to upload files to a site.

Form Handlers and Handler Methods

Form handlers evaluate the validity of form data before it is submitted, check for errors, and determine what action to take—for example, submit the data, direct the user to a different page, and display an error message. Often when you use a form handler, the form input fields are associated with properties of the form handler rather than the component you ultimately want to modify. For example, your form might include this tag:

```
<dsp:input type="text" bean="MyFormHandler.age"/>
```

When the form is submitted, a method of `MyFormHandler` associated with the age property is invoked. Depending on how this method is written, it might validate the data and then set the value of `Person1.age`, or use the data to fill in a record in a database.

A form handler class must include one or more handler methods. A handler method is typically invoked when the user clicks the submit button, and handles the processing of the form. Depending on the purpose of the form handler, it can have several different handler methods that each perform a different operation. For example, a form handler that works with user profiles might have separate handler methods for creating the profile, modifying the profile, and logging the user in.

Subclassing ATG Form Handlers

ATG form handler classes all implement the interface `atg.droplet.DropletFormHandler`. Three form handler base classes implement this interface:

- `atg.droplet.EmptyFormHandler`
- `atg.droplet.GenericFormHandler`
- `atg.droplet.TransactionalFormHandler`

You can create a form handler by extending one of these classes or any of their subclasses. The *ATG API Reference* lists all form handler classes that implement the `DropletFormHandler` interface.

EmptyFormHandler

`atg.droplet.EmptyFormHandler` implements the `DropletFormHandler` interface and provides empty implementations of the methods in this interface.

GenericFormHandler

`atg.droplet.GenericFormHandler` extends `EmptyFormHandler`. It provides simple implementations of `DropletFormHandler` interface methods and adds basic error handling logic. If errors occur in processing a form that uses `GenericFormHandler`, the errors are saved and exposed as properties of the form handler component. This form handler is included in the public API: see *ATG API Reference* for more information.

TransactionalFormHandler

`atg.droplet.TransactionalFormHandler` extends `atg.droplet.GenericFormHandler`; it treats the form processing operation as a transaction. Although this form handler methods are processed discretely, their results are saved simultaneously. The transaction management occurs in the `beforeGet` and `afterGet` methods. This establishes the transaction before any of properties are set or handler methods are called, rather than in the handler methods themselves. See Handler Methods for an explanation of handler methods.

Handler Methods

Form handlers use `handleX` methods to link form elements with Nucleus components, where `X` represents the name of a form handler property to set. `handleX` methods have the following signature:

```
public boolean handleX (javax.servlet.http.HttpServletRequest request,
                        javax.servlet.http.HttpServletResponse response)
```

A `handleX` method can also declare that it throws `java.io.IOException` or `javax.servlet.ServletException`.

`handleX` methods are called on form submission. If a corresponding `setX` method also exists, the `setX` method is called before the `handleX` method is called.

Request and Response Object Arguments

The `handleX` method is passed the request and response objects that encapsulate the request. (See the Java Servlet specifications on how to use these request and response types.) Often, the handler method does nothing with the request and response objects. For example, the handler method might simply redirect the user to another page.

A `handleX` method can also use the ATG extensions to the `HttpServletRequest` and `HttpServletResponse` interfaces, `DynamoHttpServletRequest` and `DynamoHttpServletResponse`. Thus, your handler method signature can look like this:

```
public boolean handleX (atg.servlet.DynamoHttpServletRequest request,
                        atg.servlet.DynamoHttpServletResponse response)
```

Handler Method Returns

The handler method returns a Boolean value, which indicates whether to continue processing the rest of the page after the handler is finished:

Return value	Action
`false`	No further values are processed process after the handler is called, and the rest of the page is not served. For example, a handler that redirects the user to another page should return `false`.
`true`	Normal processing of the remaining values continues, and the page specified by the form's `action` attribute is served.

As mentioned earlier, ATG form handlers typically implement the interface `DropletFormHandler`. This interface has the following methods, which are called at specific points as the form is processed:

Method	When called:
`beforeSet`	Before any `setX` methods are called
`afterSet`	After all `setX` methods are called
`beforeGet`	Before any input tags that reference this component are rendered
`afterGet`	After page rendering is complete, before the socket is closed
`handleFormException`	If an exception occurs when trying to call the `setX` method of a form

The `beforeGet` and `afterGet` methods are called only when the page is rendered, not when the form is submitted. The `beforeSet` and `afterSet` methods are called only when the form is submitted, not when the page is rendered. It is possible to have all four of these methods called on the same page.

See the Quincy Funds demo for an example of form handling. This demo uses form handler to process registration, login, and profile updates.

Submit Handler Methods

You can create handler methods that are associated with a form's submit button in the same way that form fields are associated with specific properties. Submit handler methods are particularly powerful, because the tags that implement them are processed after tags that use other input types so you can use them to evaluate the form as a whole and take appropriate actions. For example, you might write a handler method called `handleRegister`, which is invoked when the user clicks the Register Now button created by this tag:

```
<dsp:input type="submit" value="Register Now" bean="MyFormHandler.register"/>
```

A form handler can have multiple submit handler methods that each handle form submissions differently. For example, registration, update, and change password forms might use the same form handler, which has three handler methods: `handleRegister`, `handleUpdate`, and `handleChangePassword`. On submission, each form calls the appropriate method. For example, the change password form might contain the following tag for the submit button:

```
<dsp:input type="submit" value="Change Password"
  bean="MyFormHandler.changePassword"/>
```

Extending handleCancel()

The `atg.droplet.GenericFormHandler` class (and any subclass of `GenericFormHandler` you write) includes a `handleCancel` method for implementing form Cancel buttons. This method redirects to the URL specified in the form handler's `cancelURL` property. Typically this property is set to the current page, so clicking Cancel redisplays the form page. If the form handler component is request scoped, this creates a new instance of the form handler object, and reinitializes the form data.

Note: If your form handler component is session scoped (see Form Handler Scope), this reinitialization does not occur. In this case you need to extend the `handleCancel` method of its class so it resets the appropriate form data. In the following example, the `handleCancel` method calls a method you wrote that resets the form data:

```
public boolean handleCancel(DynamoHttpServletRequest pRequest,
               DynamoHttpServletResponse pResponse)
       throws ServletException, IOException
   {
     resetMyFormData();
     return super.handleCancel(pRequest, pResponse);
   }
```

Transactions in Repository Form Handlers

A form handler that can manipulate repository items should ensure that all operations that occur in a handler method call are committed in a single transaction. Committing all operations simultaneously helps ensure data integrity. A repository or database transaction is completed successfully or not at all; partially committed data is rolled back if an error occurs mid-transaction.

The `RepositoryFormHandler` and `TransactionalRepositoryFormHandler` classes ensure atomic transactions in this way. If you subclass either class without overriding the handler methods, your subclass handles transactions properly. If you override any handler methods, or add new handler methods, you must make sure that these methods handle transactions properly.

Note: A form handler starts and ends a transaction only when no other active transactions are in progress. If a transaction is in progress, the form handler returns a status for each operation it attempts and permits the transaction to continue after the form handler itself finishes processing.

To create a form handler that works with repository items while a transaction is in progress:

1. Create a form handler that subclasses `RepositoryFormHandler` or `TransactionalRepositoryFormHandler`, depending on your requirements. The source code for both form handlers is provided in

   `<ATG10dir>/DAS/src/Java/atg/repository/servlet`

   Both form handlers create a transaction if one is not already in progress and provide the status of all operations performed by the form handler while the transaction is in place. The two form handlers mainly differ in the transaction lifespan.

2. Create methods on your new form handler that are transaction-aware. See Transaction-Aware Methods.

Note: `RepositoryFormHandler` and `TransactionalRepositoryFormHandler` are useful for manipulating repository items. The form handler class `atg.droplet.TransactionFormHandler` supports transactions and lets you work with the JDBC directly.

RepositoryFormHandler

`atg.repository.servlet.RepositoryFormHandler` is a base form handler that provides tools for creating, modifying, and deleting items stored in an SQL repository. ATG provides one direct instance of this class, `/atg/demo/QuincyFunds/FormHandlers/EmailRepositoryFormHandler`.

In the `RepositoryFormhandler`, the transaction is governed entirely by the submit handler method, meaning the transaction starts when a submit handler method is invoked and ends when it completes execution. Transaction status is reported for the data validation and the data commit operations.

TransactionalRepositoryFormHandler

The `atg.repository.servlet.TransactionalRepositoryFormHandler`, a subclass of the `RepositoryFormhandler`, provides enhanced transaction support by broadening the scope of the transaction. This class also defines a few additional properties that are useful for transaction monitoring. ATG Adaptive Scenario Engine does not include any instances of this class.

Transactions begin when the `beforeSet` method is invoked and end with the `afterSet` method. Because a transaction status is generated for all operations that occur during its execution, a status is recorded for each the following operations:

- `beforeSet` method execution
- Processing of all other tags in the JSP (tags that implement submit operations have the lowest priority on the page)
- Submit handler method data validation
- Submit handler method data commit
- `afterSet` method execution

Transaction-Aware Methods

When you create a form handler that subclasses `RepositoryFormHandler` or `TransactionalRepositoryFormHandler`, make sure that its methods can correspond with the Transaction Manager. There are two ways to do this:

- Base new handler methods on handleUpdate code so that you can reuse the transaction code in it, then modify the rest accordingly.
- Modify existing handler methods by inserting code before or after their execution in the `preX` or `postX` methods, respectively.

Base New Handler Methods on handleUpdate Source Code

The code provided here implements the `handleUpdate` method. Create your own handler methods by making changes to this code sample and inserting it into your subclassed form handler:

```
public boolean handleUpdate(DynamoHttpServletRequest pRequest,
                            DynamoHttpServletResponse pResponse)
     throws ServletException, IOException
{
  TransactionDemarcation td = getTransactionDemarcation();
  TransactionManager tm = getTransactionManager();
  try {
    if (tm != null) td.begin(tm, td.REQUIRED);

    int status = checkFormError(getUpdateErrorURL(), pRequest, pResponse);
    if (status != STATUS_SUCCESS) return status == STATUS_ERROR_STAY;

    // update the repository item
    preUpdateItem(pRequest, pResponse);

    if (!getFormError())
      updateItem(pRequest, pResponse);

    postUpdateItem(pRequest, pResponse);
```

```
      // try to redirect on errors
      if ((status = checkFormError(getUpdateErrorURL(), pRequest, pResponse))
       != STATUS_SUCCESS)
     return status == STATUS_ERROR_STAY;

      // try to redirect on success
      return checkFormSuccess(getUpdateSuccessURL(), pRequest, pResponse);
    }
    catch (TransactionDemarcationException e) {
      throw new ServletException(e);
    }
    finally {
     try { if (tm != null) td.end(); }
      catch (TransactionDemarcationException e) { }
    }
  }
```

Modify Existing Handler Methods

The three existing submit handler methods (`handleCreate`, `handleUpdate`, and `handleDelete`) each provide a pair of empty `pre` and `post` methods where you can add custom code.

It is likely that you use either the `preX` or the `postX` method for a given existing handler method although you can customize both. For example, consider a subclass of `TransactionalRepositoryFormHandler` where `preUpdate` and `postUpdate` are used:

1. The form is rendered, which causes `getX` method to display current values for properties used in the form.
2. The user fills in the form and submits it.
3. The form handler's `beforeSet` method is invoked. If a transaction is not currently in progress, the `TransactionalRepositoryFormHandler` component creates a one.
4. If tag converters are used, they are applied to the specified content. Any form exceptions that occur now or at any point during the form handler execution are saved to the form handler's `formException` property.
5. The `setX` method is called, followed by the form handler's `handleUpdate` method. Severe form exceptions might cause form processing to stop and the transaction to rollback, before redirecting users to a different page.
6. The `preUpdateItem` method is invoked. The `preX` method for the `handleUpdate` method is `preUpdateItem`. Serious errors generated from this operation might also prompt the transaction to rollback.
7. The `updateItem` method, which is the `handleUpdate` method responsible for processing the content and updating the database, is invoked. Again, this is another operation that can cause a transaction to rollback when serious errors are detected. At this point, the changes made by the actions associated with the transaction are kept private, meaning that they are only visible within the transaction itself.

8. The `postUpdateItem` method is invoked. Again, the transaction is rolled back if serious errors are detected.
9. The `afterSet` method is invoked. If the transaction was started by the `beforeSet` method, the transaction concludes and the content it saved to the database is publicly visible.

Use the `preX` method when you want to expand the constraints for data validation. For example, you might want to check if the user-entered zip code and country correspond to each other. The countries that use zip codes require them to be a certain length. The `preX` method can verify that a zip code uses the appropriate format for the country, before saving the zip code and country values to the database. Any discrepancies produce a form exception and roll back the transaction.

The `postX` method is useful for verifying user entered-data after that data has been converted by tag converters. For example, a form handler that saves credit card information might use a `postX` method to handle authorization. After that credit card number has been formatted correctly and all related information is updated in the database, the `postX` method executes. If the authorization fails, the transaction is rolled back, the original data refreshed, a form error exception is thrown, and the user is redirected to a page where the use can re-enter credit card information.

Handler Methods and dsp:setvalue

When a `dsp:setvalue` tag is rendered, it invokes the same methods as events processed from a form. This includes calling both the appropriate `setX` and `handleX` methods. For example, a JSP might contain the following tag:

```
<dsp:setvalue bean="/test/Person1.name" value="Frank"/>
```

When this tag is encountered during page rendition, the following actions occur:

1. `setName("Frank")` is called on the `/test/Person1` component, if a `setName` method exists for the class that `Person1` instantiates.
2. The `handleName` method is called, if it exists;

This process is the same as when property values are set from a form input tag.

You can use this technique with a form handler to set up aspects of the form before it is displayed and handled. For example, you might want to display the form only to users who are over the age of 13. You can write a `handleAge` method for your form handler that returns `false` if the user is under 13. The page might contain something like this:

```
<dsp:setvalue bean="MyFormHandler.age" paramvalue="currentUser.age"/>

<dsp:form action=...>
 ...
```

In this example, before the form is displayed, the `setvalue` statement sets the value of the form handler's age property to the value of the `currentUser.`age parameter. (Presumably, this parameter is set from information stored when the user registered.) Setting the age property causes the form handler's `handleAge` method to be invoked. If age is less than 13, this method returns `false`, and the rest of the page (including the form) is not rendered. If age is 13 or greater, `handleAge` returns `true`, and the rest of the page is rendered.

Form Handler Scope

A form handler component should be request-scoped or session-scoped. A request-scoped form handler exists for the duration of the request. Consider a form that is held in one page. By clicking the submit button, the user makes a request that, in turn, creates an instance of the form handler. The configured values in the form handler's properties file are used. You can override the values of these properties using a `dsp:setvalue` or `dsp:input` tags. After a user submits the form, the form handler processes the data.

When a form handler spans several pages such as in a multi-page registration process, values entered in each page should persist until final submission. If the form spans only two pages, you can implement the registration process with a request-scoped form handler by designing it to support a redirect: make data from page one available to page two. Only one redirect is available to a given form handler instance , so this approach is valid only for forms that are no longer than two pages.

if a form spans more than two pages, two approaches enable persistence of form values:

- Use a session-scoped form handler
- Use page-specific request-scoped form handlers

Session-scoped form handler

A session-scoped form handler ensures that all form values persist across multiple pages. However, the form remains in memory for the entire user session, so use of multiple form handlers within a single session can incur considerable overhead.

If you use session-scoped form handlers, be sure to reset or clear values between uses of the form handler because values remain in effect throughout the entire session. Also clear error messages so they do not appear in other forms where they are not relevant.

Page-specific request-scoped form handlers

Each page of a multi-page form can use separate, request-scoped form handler instances; all pages share the same session-scoped component. With each form submission, the form handler automatically copies the data to the session-scoped component. That way, if you want page five to hold a list of data entered in pages one through four for validation, you need only reference the relevant properties in the session-scoped component. This technique offers the persistence of a session-scoped form handler and data refresh provided in a request-scoped form handler.

This technique is especially helpful in implementing search form handlers when you want to search results available for future reference. In order to implement this behavior, design your form handler's submit handler method to retrieve the session-scoped component and set specific properties on it.

Tag Converters

ATG provides tag converter classes that let you explicitly control how form data is converted on input and displayed on output, and when exceptions are thrown. Certain DSP tags such as `dsp:input` and `dsp:valueof` can specify these tag converters; details about syntax and usage is provided in the *ATG Page Developer's Guide*.

Creating Custom Tag Converters

You can modify the tag converters provided in the ATG platform; you can also create a custom tag converter in the following steps:

1. Extend an existing tag converter class or create one that implements interface `atg.droplet.TagConverter`. The new class must implement the following TagConverter methods:
   - `getName()`
   - `getTagAttributeDescriptors()`
   - `convertStringToObject()`
   - `convertObjectToString()`
2. Optionally, create attributes for the tag converter through an instance of the `TagAttributeDescriptor` class.
3. Register the new tag converter with the TagConverterManager (`atg.droplet.TagConverterManager`) by calling `TagConverterManager.registerTagConverter()` with an instance of the new tag converter's class. The tag converter must be registered before a JSP can use it. Include the call to `registerTagConverter()` in a class that is initialized during the startup process of your module—that is, is in the `Initial` services list.

getName()

Returns the name of the tag converter that is supplied as an argument to the `converter` attribute. For example, the `getName()` method that is implemented by the tag converter class `atg.droplet.CurrencyTagConverter` returns the string `currency`. So, a `dsp:valueof` tag specifies this tag converter as follows:

```
<dsp:valueof param="myPrice" converter="currency"/>
```

getTagAttributeDescriptors()

Returns an array of TagAttributeDescriptors (`atg.droplet.TagAttributeDescriptor`). The constructor for a TagAttributeDescriptor is defined as follows:

```
TagAttributeDescriptor(String pName,
                       String pDescription,
                       boolean pOptional,
                       boolean pAutomatic)
```

Each TagAttributeDescriptor is defined with an attribute name and description, and two Boolean properties:

- `Optional`: Specifies whether this attribute is optional or required. For example, ATG's `currency` converter takes a `locale` attribute whose `Optional` property is set to `true`. If this attribute is omitted, the locale associated with the current request is used. Conversely, the `Date` converter's `date` attribute is marked as required, so all `Date` converters must supply a date format.
- `Automatic`: Not supported for custom tag converter, this property specifies whether you can supply the attribute without the `converter` attribute. Only one attribute be marked as automatic; all other attributes must set this property to `false`.

For example, the tag converter class `atg.droplet.Warthog` defines the two attributes, `recommendations` and `winddirection`, and sets their Optional and Automatic properties as follows:

```
public class Warthog implements TagConverter
{
...
static final String RECS_ATTRIBUTE = "recommendations";
static final String WINDDIR_ATTRIBUTE = " winddirection";
...
private final static TagAttributeDescriptor[] sTagAttributeDescriptors = {
    new TagAttributeDescriptor(RECS_ATTRIBUTE,
        "A string for recommendations",
        false, false),
    new TagAttributeDescriptor(WINDDIR_ATTRIBUTE,
        "A string for wind direction",
        true, false),
}
```

Note: Two constraints apply to custom tag converter attributes:

- The `Automatic` property is supported only for ATG tag converters; DSP tags can only reference custom tag converters through the `converter` attribute.
- Custom tag converters must reference their attributes through the `converterattributes` attribute. See Using Custom Tag Converters for details.

convertStringToObject()

This method is called when a tag converter is used by one of the following DSP tags:

`dsp:input`	On form submission, `convertStringToObject()` is called before the target property's `setX` method is called. `convertStringToObject()` creates the `Object` value for use by the property's `setX` method. The method can throw a `TagConversionException` if an error occurs during the conversion process. `convertStringToObject()` typically returns null if a form field is left empty. In this case, the target property's `setX` method is not called, and its value remains unchanged. Alternatively, this method can specify setting the target property to a null value if the field is empty, by returning `TagConverterManager.SET_AS_NULL`. This instructs the `setX` method to set the property value to null.
`dsp:param`	`convertStringToObject()` is called when the `dsp:param` tag defines the parameter's value.

convertObjectToString()

This method is called in two situations:

- When you use a converter in a `valueof` tag, this method is used to convert the `Object` value into a `String` value before displaying it.
- When you use this tag in an `input` tag with a bean attribute and no existing `value` attribute, this method is called to fill in the `value` attribute with the current value of the bean property.

Attribute Definition Constraints

Automatic attributes must be unique among all registered tag converters. Multiple tag converters can define attributes with the same name only if a given attribute specification can always be associated unambiguously with the appropriate converter.

For example, the class `atg.droplet.RequiredTagConverter` defines `required` as an automatic attribute. The class `atg.droplet.DateTagConverter` defines `date` as an automatic attribute and `required` as an optional attribute. Given these attribute definitions, DSP tags can use either tag converter with the `required` attribute, as shown in the following examples:

DSP tag	Converter
`<dsp:valueof param="date" date="M/dd/yy" required/>`	`DateTagConverter`

`<dsp:input type="text"`
`  bean="Person.userName" required="true"/>`
	`RequiredTagConverter`

Using Custom Tag Converters

A DSP tag that specifies a custom tag converter must reference its attributes through the `converterattributes` attribute, as follows:

```
<dsp-tag converter="converter-name" converterattributes=attr-list />
```

`attr-list` is a list of semi-colon-delimited attributes that are defined in the tag converter, and their values. The `convertattributes` attribute allows use of attributes that are unknown to the `DSPJSP` tag library (`http://www.atg.com/taglibs/daf/dspjspTaglib1_0`).

For example, given a custom tag converter `Warthog` that defines two attributes, `recommendations` and `winddirection`, a `dsp:input` tag can specify this tag converter as follows:

```
<dsp:input type="text"
           bean="NuclearOrderFormHandler.address1"
           converter="Warthog"
           converterattributes="recommendations=splunge;winddirection=west" />
```

Sample Tag Converter

The following example shows how you might create a tag converter that converts minutes into a formatted string that shows the number of hours and minutes. For example, given installation of this tag converter, a JSP can include the following tag:

```
<dsp:valueof param="numMinutes"
  converter="minutestToHoursMinutes"
  converterattributes="asHoursAndMinutes=true"/>
```

If the page parameter `numMinutes` has a value of 117, this tag converter displays the value as follows:

```
1 hr 57 mins
```

```
import atg.droplet.TagConverter;
import atg.droplet.TagAttributeDescriptor;
import atg.droplet.TagConversionException;
import atg.servlet.DynamoHttpServletRequest;
import java.util.Properties;

//Convert minutes into the format H hr(s) M min(s)

public class MinutesToHoursMinutesConverter implements TagConverter {

  private static final int MINUTES_PER_HOUR = 60;
  static final String ASHOURSANDMINUTES_ATTRIBUTE = "asHoursAndMinutes";
  private static final TagAttributeDescriptor[] S_TAG_ATTRIBUTE_DESCRIPTORS ={
    new TagAttributeDescriptor(
      "ASHOURSANDMINUTES_ATTRIBUTE",
      "If provided, assume input is # of minutes, format as H hr(s) M min(s)",
      false, true)};

  public String getName() {
    return "minutesToHoursMinutes";
```

```
}

public TagAttributeDescriptor[] getTagAttributeDescriptors() {
  return S_TAG_ATTRIBUTE_DESCRIPTORS;
}

public Object convertStringToObject(
  DynamoHttpServletRequest request,String string, Properties properties)
    throws TagConversionException {
      throw new TagConversionException("Unable to convert string to object.");
}

public String convertObjectToString(
  DynamoHttpServletRequest request, Object pValue, Properties properties)
    throws TagConversionException {
      if(pValue == null){
      return null;
      }
      if (pValue instanceof Integer) {
        return buildOutput(((Integer)pValue).intValue());
      } else {
        return pValue.toString();
      }
}

private String buildOutput(int totalMinutes) {
  int hours = (int) Math.floor(totalMinutes / MINUTES_PER_HOUR);
  int minutes = totalMinutes % MINUTES_PER_HOUR;
  return formatOutput(hours, minutes);
}

private String formatOutput(int hours, int minutes) {
  StringBuffer buf = new StringBuffer(100);
  if(hours > 0) {
    buf.append(hours);
    if(hours == 1) {
      buf.append(" hr");
    } else {
      buf.append(" hrs");
    }
  }
  if(minutes > 0) {
    buf.append(" ").append(minutes);
    if(minutes == 1){
      buf.append(" min");
    } else {
      buf.append(" mins");
    }
  }
  return buf.toString();
  }
}
```

File Uploading

You can use a DSP tag library form element to enable users to upload files. The form defines the input as type file and makes sure that there is a place to put the file. Here is an example of a file upload form element:

```
<dsp:form enctype="multipart/form-data" action="a_page.jsp" method="post">
   Pick a file to upload:
   <dsp:input type="file" bean="/FileUploadComponent.uploadProperty" value=""/>
   <dsp:input type="submit" value="Upload now"/>
</dsp:form>
```

This form must exist in a JSP. The `enctype multipart/form-data` and the method `post` are required for file upload. For the purposes of the upload itself, the value of the `action` attribute is not important.

The crucial attribute, the bean= attribute, is in the file input tag. This attribute links to a Nucleus component written by you and containing as a property a member of type `atg.servlet.UploadedFile`. Here, that component is `/FileUploadComponent`, and its property is named `uploadProperty`. The file upload component's property must have the usual `getX` and `setX` methods required of a JavaBean.

File Upload Component Example

The following is an example of a component that handles file uploads. One way to use this component is through a form element such as the one presented in the previous example. The example here includes two alternatives, one that returns the uploaded file as a byte array and one (which is commented out) where the uploaded file is read from the input stream.

```
import atg.servlet.UploadedFile;
import java.io.*;
import atg.droplet.GenericFormHandler;
public class FileUploadComponent extends GenericFormHandler
{
  /**
   * This method is called when the form above is submitted.  This code makes
   * sure that it has an appropriate object and then pass it along for further
   * processing.
   * @param Object either an UploadedFile or an UploadedFile[]
   **/
  public void setUploadProperty(Object fileObject) {
    if(fileObject == null) {
        System.err.println("**** ERROR: FileUploadDroplet received a NULL file.");
        return;
      }

    if (fileObject instanceof UploadedFile[]) {
      System.out.println("Reading in UploadedFile[]");
      readUpFiles((UploadedFile[]) fileObject);
    }else if (fileObject instanceof UploadedFile){
      readUpFiles(new UploadedFile[]{(UploadedFile)fileObject});
    }else{
      System.err.print
        ("**** ERROR: FileUploadDroplet received an Object which is "
         + "neither an UploadedFile or an UploadedFile[].");
    }
  }
/**
   * Returns property UploadProperty
   **/
  public Object getUploadProperty() {
    // return null since we don't need to maintain a
    // reference to the original uploaded file(s)
    return null;
  }
//-------------------------------------
  /**
   * Here you can access the data in the uploaded file(s). You
   * should get the data from the uploaded file before the
   * request is complete.  If the file is large, it is stored as a temporary
   * file on disk, and this file is removed when the request is complete.
   * @param UploadedFile[] the uploaded file(s)
   **/
  void readUpFiles(UploadedFile[] pFiles){
    UploadedFile upFile = null;
    String clientFilePath = null;
    String fileName = null;
    File localFile = null;
    FileOutputStream fos = null;
    byte[] fileData = null;

    for (int i = 0; i < pFiles.length; i++){
      upFile = pFiles[i];
      clientFilePath = upFile.getFilename();

      // Check that file uploaded is not size 0.
      if(upFile.getFileSize() <= 0){
        System.err.println
          (" FileUploadDroplet Cannot upload - file has length 0: "
          + clientFilePath);
        return;
      }
```

```
/**
 * Extract the FilePath, which is the file location provided by the
 * browser client. Convert the file separator character to use the one
 * accepted by the web client's Operating system.
 **/

String otherSeparator = "/";
if ( "/".equals(File.separator))
  otherSeparator = "\\";
  String convertedClientFilePath = atg.core.util.StringUtils.replace
  (clientFilePath,otherSeparator,File.separator);

fileName =
  new String
  (convertedClientFilePath.substring
  (convertedClientFilePath.lastIndexOf
  (File.separator)+1));

// Construct a local file (using the uploaded file directory)
localFile = new File
  (mUploadDirectory
  + File.separator
  + fileName);

// You can either get the file as an array of bytes ...
try {
  fileData = upFile.toByteArray();
  System.out.println
    (" ** client filename: " + clientFilePath);
  System.out.println
    (" ** client file is " + upFile.getFileSize() + " bytes long.");
  fos = new FileOutputStream(localFile);
  fos.write(fileData);
  fos.flush();
}
catch (IOException e) {
  System.err.println("FileUploadDroplet failed");
  e.printStackTrace();
}
finally {
  if (fos != null){
    try {
      fos.close();
    }catch(IOException exc) {
      exc.printStackTrace();
    }
  }//end try/catch
}//end finally

// ... or you can read the data yourself from the input stream.
```

```
      /**
      try{
        InputStream is = upFile.getInputStream();
        ...
      }
      catch (IOException e) {
      }  **/
    }// end for
  }// end readUpFiles

  //-------------------------------------
  // property: UploadDirectory
  // where we will put the uploaded file
  String mUploadDirectory;

  /**
   * Sets property UploadDirectory
   **/
  public void setUploadDirectory(String pUploadDirectory) {
    mUploadDirectory = pUploadDirectory;
  }

  /**
   * Returns property UploadDirectory
   **/
  public String getUploadDirectory() {
    return mUploadDirectory;
  }

}
```

7 Accessing Nucleus in a Web Application

As discussed in the Nucleus-Based Application Structures section of the Developing and Assembling Nucleus-Based Applications chapter, each EAR file assembled by the `runAssembler` command includes a web application named `atg_bootstrap.war`. This module's `web.xml` file includes the tags necessary to configure the application to run an instance of Nucleus. This Nucleus instance is then available to any other web applications in the EAR file.

In addition to `atg_bootstrap.war`, the EAR file typically includes one or more web applications that actually run your site. For example, the `QuincyFunds.ear` file includes a web application named `quincy.war` that runs the demo site. This application uses Nucleus components that implement ATG's personalization features and are accessed in the application's JSPs through the DSP tag libraries (described in the *ATG Page Developer's Guide*).

This chapter describes how you can access Nucleus in web applications by adding the necessary entries to the web application's `web.xml` deployment descriptor. For general information about `web.xml` files, see the J2EE specifications.

In this chapter

This chapter discusses the following topics:

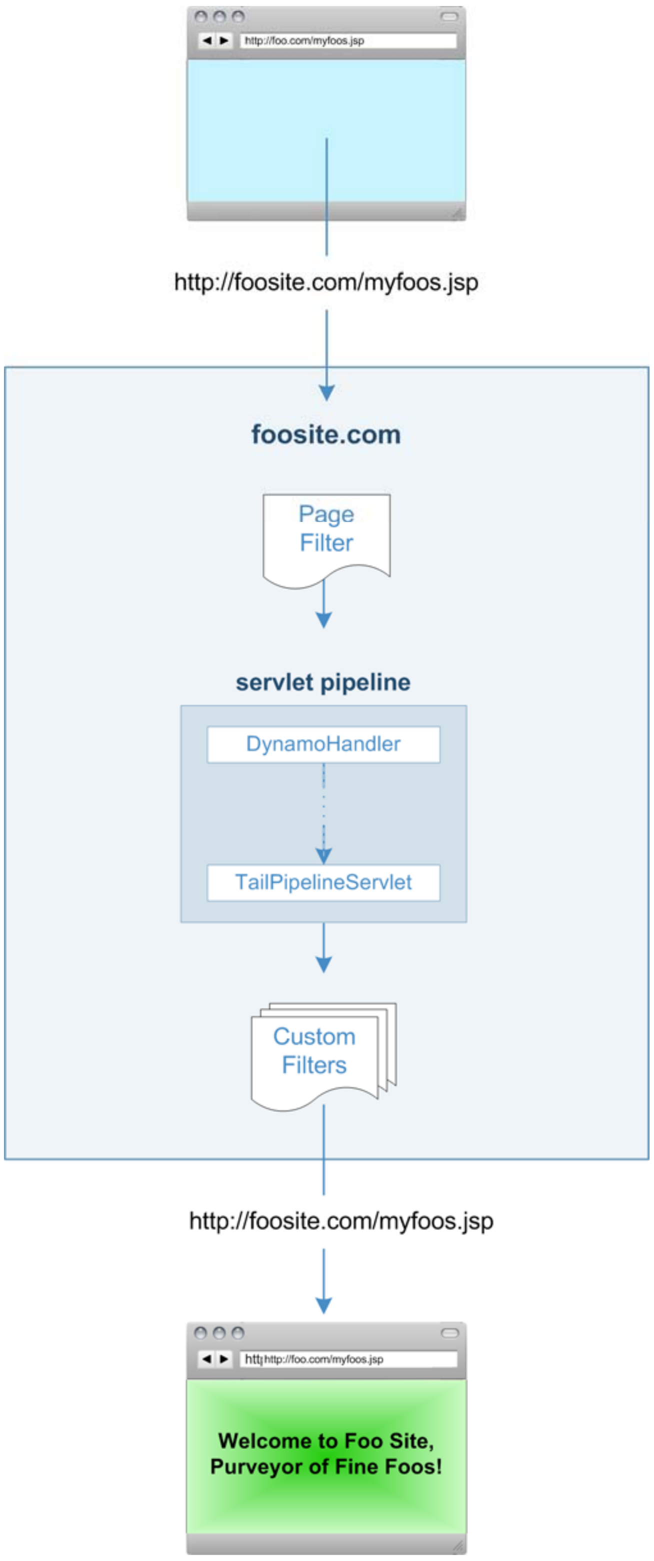
Request Processing in a Nucleus-Based Application

The main entry point to Nucleus is though its request-handling facilities. When a web application running Nucleus receives an HTTP request, it invokes a series of servlets and servlet filters that process the request and construct the response:

1. When a user requests a page, that request is sent to the web server as a stream of information that the web server parses and holds in an `HTTPServletRequest` object.
2. The web server passes the `HTTPServletRequest` to the application server, which it wraps in its own flavor of request around the generic one before passing it to the web application.

3. Any custom J2EE servlets are processed.
4. The web application calls the `PageFilter` resource and starts the servlet pipeline.
5. The request is passed on to a pipeline of servlets. Each servlet available to the pipeline is designed to insert itself into the pipeline when a given request includes the information that it processes. So, a unique pipeline is constructed for each request. If you created any custom servlets and they apply to the current request, they are executed now.
6. If you created custom filters and a JSP is being requested, they execute after the last pipeline servlet, but before the request returns to the application server.
7. After the servlet pipeline reaches the end, it returns the request to the application server for final processing (executing the servlet representation of the page).

The following figure illustrates this process:


http://foo.com/myfoos.jsp
http://foosite.com/myfoos.jsp
foosite.com
Page Filter
servlet pipeline
DynamoHandler
TailPipelineServlet
Custom Filters
http://foosite.com/myfoos.jsp
http://foo.com/myfoos.jsp
Welcome to Foo Site, Purveyor of Fine Foos!

For more information about ATG's request handling pipelines, see the Request Handling with Servlet Pipelines chapter in this guide.

Resources in web.xml

The web application deployment descriptor specifies a series of resources that are instantiated and configured based on the settings you provide. In `web.xml`, you are required to include the following resources:

- The DTD declaration and web application name in your `web.xml` as you would for any other J2EE application.
- `NucleusServlet`, which is the servlet responsible for running Nucleus as a servlet.
- A `PageFilter` that starts the request-handling pipeline.
- A `<distributable/>` tag, in order to enable session failover when running in a cluster. This tag is added automatically when you assemble your EAR using the –`distributable` flag.

In additional to these required resources, you can specify other optional resources (described later in this chapter) that the ATG platform provides. Most applications require site-specific resources.

Running Nucleus

To use ATG platform functionality, a web application needs to start Nucleus by invoking `NucleusServlet`. This servlet does not need to have any paths mapped to it, but must have the `load-on-startup` flag set to 1 so that it runs before any other ATG component. The `web.xml` file in `atg_bootstrap.war` includes the following lines:

```
<servlet>
  <servlet-name>NucleusServlet</servlet-name>
  <servlet-class>atg.nucleus.servlet.NucleusServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

The `NucleusServlet` creates an instance of Nucleus and sets it as an attribute of the web application. The Nucleus instance can be retrieved using the `Nucleus.getGlobalNucleus()` method.

Note: When you declare servlets in `web.xml`, you can use optional `load-on-startup` tags to determine the order the servlets are called. If you do not use these tags, the servlets are called in the order that they appear in `web.xml`. Because `NucleusServlet` must run first, its `load-on-startup` value must be 1, as in the example above.

Starting the Request-Handling Pipeline

After invoking NucleusServlet, the web container calls `PageFilter` filter, which starts execution of the request-handling pipeline for JSP requests by calling the pipeline's first servlet, `/atg/dynamo/servlet/dafpipeline/DynamoHandler`. This servlet generates a `DynamoHTTPServletRequest` that wraps the generic `HTTPServletRequest` so pipeline servlets can read information from the request and also modify it. A matching `DynamoHTTPServletResponse` is also generated. `DynamoHandler` passes the Dynamo request/response pair to the next servlet in the request-handling pipeline.

For more information, see Request Handling with Servlet Pipelines.

To include `PageFilter` in `web.xml`:

- Insert the filter name and class name in enclosing `<filter>` tags.
- Map the filter to either a directory holding JSPs or the `.jsp` extension. This information is included in enclosing `<filter-mapping>` tags.

For example:

```
<filter>
 <filter-name>PageFilter</filter-name>
 <filter-class>atg.filter.dspjsp.PageFilter</filter-class>
</filter>
<filter-mapping>
 <filter-name>PageFilter</filter-name>
 <url-pattern>*.jsp</url-pattern>
</filter-mapping>
```

Optional Resources

The ATG platform installation provides a number of useful optional resources. Some of these resources enable key portions of the ATG platform that are accessible only if `web.xml` includes them.

Context Root Changes on a Live Site

The `context-root` context parameter is a mechanism for making a web application context root visible to the ATG Control Center. ATG Scenarios relies on this parameter to inform the ATG Control Center about changes to the context root that is defined for a live site. For more information, see the *ATG Personalization Programming Guide*.

PageFilter Debugger

If set to true, the `atg.filter.PagefilterDebug` parameter signals error information to be tracked in a log. The default setting is false. Set this parameter to `true` if you encounter unexpected behavior and need to provide error information to Technical Support.

Targeted Email

When your site supports targeted email, you must invoke the `InitSessionServlet` and map it to the appropriate component. See the *ATG Installation and Configuration Guide*.

ATG Dynamo Server Admin

To access ATG Dynamo Server Admin:

- Specify `AdminProxyServlet` and indicate `AdminHandler`, the first servlet in the Admin servlet pipeline as an initialization parameter.
- Map `AdminProxyServlet` to the directory which, when requests for pages in it are called, prompts the `AdminProxyServlet` to execute.

For more information, see Including ATG Dynamo Server Admin.

Prevention of Profile Swapping

Several ATG applications contain preview features that allow users to test content on a sample user profile. The implementation of this feature requires swapping the profile of the logged-in user with the profile selected for preview. If your web application does not include preview features, it is recommended that you disable profile swapping by setting the `atg.preview` context parameter to false in the `web.xml` file. For more information, see the *ATG Personalization Programming Guide*.

Tag Libraries

You can make a tag library available to a web application in two ways:

- Put the tag library class files and TLD in the web application WEB-INF directory. In the TLD, specify the URI value that matches the value in the JSPs that use the tag library. All ATG tag libraries are implemented in this fashion.
- Use `web.xml` to define the tag library URI and TLD location. The URI must match the one used in JSPs.

Both methods are equally effective; and with two methods available, you can support two URIs. You do so by declaring the tag library in `web.xml` with one URI, and keeping the tag library files, including the TLD that defines a second URI, in WEB-INF.

The following example shows how to declare the DSP tag library in `web.xml`:

```
<taglib>
 <taglib-uri>/dspTaglib</taglib-uri>
 <taglib-location>/WEB-INF/taglibs/dspjspTaglib1_0.tld</taglib-location>
</taglib>
```

For more information about the DSP tag libraries, see the *ATG Page Developer's Guide*.

Web Services

You make web services available to your J2EE application by declaring them in `web.xml`. It is common practice to define web services in their own web application so they are the only resource defined in `web.xml`. All ATG web services are implemented in this way. When you create custom web services in the ATG platform, a new web application is created for them where they are specified in `web.xml`.

You can include any of ATG's prepackaged web services in an assembled EAR file by including the module that contains the desired services. For example, to include the ATG Commerce services, specify the `DCS.WebServices` module when you invoke the `runAssembler` command. To include web services you created through the Web Service Creation Wizard, use the `runAssembler` flag `–add-ear-file` to specify the EAR file that contains the service.

For more information about ATG web services, see *ATG Web Services and Integration Framework Guide*.

Adding Request-Handling Resources to web.xml

You might want to include resources in your web application that allow you to pass dynamic information to Nucleus components through the Dynamo request. Such resources include context parameters, filters, servlets, web services, and tag libraries. The resources you create need to be consistent with the J2EE (or W3C for web services) standards described in their respective specifications. After resources are created, you must register them with the web application by adding them to `web.xml` in accordance with the J2EE specifications.

Keep in mind that the J2EE servlets you create are processed after the servlet pipeline. Therefore, custom filters should be called just after `PageFilter`. Thus, `PageFilter` first activates the servlet pipeline; on completion, your custom filters execute.

Creating Filters and Servlets

It is likely that the filters and servlets you create need to access the Dynamo request and response objects, in order to modify the data they retrieve during pipeline-processing. To do so, code your filters and servlets as follows:

1. Import `atg.servlet.DynamoHttpServletRequest`, `atg.servlet.DynamoHttpServletRequest`, and `atg.servletServletUtil`.
2. Call `ServletUtil.getDynamoRequest(request)`. This method returns a reference to the `DynamoHttpServletRequest`.

Add filters and servlets to `web.xml` following the J2EE specifications.

Filter Example

This filter accesses the Dynamo request and response objects and retrieves the Profile object attached to the request. Next, the filter finds the Profile ID on the Profile object and saves as an attribute of the request. Finally, the filter passes control to the filter chain so it determines the next resource to call.

Keep in mind that this code sample might not provide the most efficient means for obtaining the Profile object, but rather it is an easy-to-follow code sample that illustrates how a filter operates in the context of a filter chain.

```
import atg.servlet.ServletUtil;
import atg.servlet.DynamoHttpServletRequest;
import atg.servlet.DynamoHttpServletResponse;
import atg.userprofiling.Profile;

import javax.servlet.*;
import javax.servlet.http.*;

/*
 * An example filter that demonstrates how
 * to get the DynamoHttpServletRequest
 * in a Filter.
 */
public class MyFilter
  implements Filter {

  /*
   * Called when MyFilter is started
   * by the application server.
   */
  public void init(FilterConfig pConfig) {
   // Initialize MyFilter here.
  }

  /*
   * Called when MyFilter is about to
   * be destroyed by the application server.
   */
  public void destroy() {
   // Cleanup MyFilter here
  }
  /*
   * Called by the application server
   * when this filter is involved in a request.
   * Resolves the Profile nucleus component
   * and adds the Profile id as a request
   * attribute.
   */
   public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain)
    throws IOException, ServletException
  {
    // Get the Dynamo Request/Response Pair
   DynamoHttpServletRequest dRequest =
         ServletUtil.getDynamoRequest(request);
   DynamoHttpServletResponse = dRequest.getResponse();

   // Resolve the Profile object
   Profile profile =
     (Profile)dRequest.resolveName("/atg/userprofiling/Profile");

   // Add the Profile id as a request attribute
   request.setAttribute("PROFILE_ID",
     profile.getRepositoryId());

   // Pass control on to the next filter
   chain.doFilter(request,response);
   return;
  }
}
```

The example described here accesses the request and response in this manner. It also resolves a component in Nucleus, which is another common operation that can be handled by a filter. Any resource that makes calls to a Nucleus component must also provide a means for discerning that component's Nucleus address. For instructions on how to do this, see the Basic Nucleus Operation section in the Nucleus: Organizing JavaBean Components chapter.

8 Request Handling with Servlet Pipelines

One of the most important tasks for an ATG server is handling HTTP requests. The ATG server extends the basic web server model with various Nucleus services that implement the `Servlet` interface, and which are linked in order to process HTTP requests. Each servlet performs a specialized function on a request, then relays the request—sometimes in modified form—to the next servlet in the chain. While each servlet performs a unique service, it often relies on changes that previous servlets made to the request. This chain of servlets is called a *request handling pipeline*.

For example, a typical request might be processed as follows:

1. Compare the request URI against a list of restricted directories, to make sure that the user has permission to access the specified directory.
2. Translate the request URI into a real file name, taking index files into account when the file name refers to a directory.
3. Given the file name's extension, determine the MIME type of the file.
4. From the MIME type, dispatch the request to the appropriate handler.

This is one of many request-handling configurations. Other configurations might dispatch based on a beginning path such as `/cgi-bin`. Other configurations might move the session-tracking step to be performed only for files with the MIME type `text/session-tracked`.

Because the request handling pipeline is composed of Nucleus components that are independently configurable, it is easy to modify, giving you the flexibility that enterprise applications often require.

In this chapter

This chapter includes the following sections that describe how to use and customize the DAF servlet pipeline:

This chapter does not address features provided by the ATG Portal module. For more information on these, see the *ATG Portal Development Guide*.

Request Processing

A request processed by an application server follows the path described in this section, which assumes you configure your web application to use `PageFilter` as demonstrated in `atg_bootstrap.war`.

When a user performs an action that prompts a response, the application server creates an instance of the `HttpServletRequest` and `HttpServletResponse`. Based on the directories and file extension of the `requestURI`, the application server uses servlet and filter mappings defined in `web.xml` to determine the next resource to call.

By default, `PageFilter` is mapped to handle JSP requests. When the application server invokes `PageFilter`, it checks the request and response for a reference to a Dynamo request and response pair. The pair does not exist, so `PageFilter` starts the DAF servlet pipeline by calling `DynamoHandler`, the first servlet in the pipeline. The DAF servlet pipeline processes through a series of servlets that modify the request and response by extracting path information and providing session, user, and security information. The last servlet in the pipeline is `TailPipelineServlet`. It is responsible for calling `FilterChain.doFilter()`, which invokes the next filter defined in `web.xml`. The web application, unless it uses ATG Portal, does not include other servlet filters.

By default, no filters are involved in request-handling process. For more information on how to implement J2EE servlets and filters in an ATG web application, see the Accessing Nucleus in a Web Application chapter in this guide.

Servlet Interface

In order to use the servlet pipeline, you should be familiar with the `Servlet` interface and the servlet model for handling requests. This section outlines basic concepts.

The role of the web server can be summarized as parsing HTTP requests into request/response object pairs, `HttpServletRequest` and `HttpServletResponse`, respectively. These object pairs are relayed to servlets that actually handle the requests. A servlet services each request by examining request parameters and producing the appropriate output.

Request Handling

When a web server receives a request, it receives a stream of information from the browser. This information is parsed into different parts, such as a request URI, query arguments, headers, and cookies (a subset of the headers). This information is packaged into a single Java object called a `javax.servlet.http.HttpServletRequest`.

A request might also carry additional information depending on the type of the request. For example, a form submitted through a POST request uses this additional information to pass the form submission arguments. This additional information can be read as a stream from a `javax.servlet.ServletInputStream`, which can be obtained from the `HttpServletRequest`.

Generated Response

After the web server receives the request, it generates output to send back to the browser. The output includes a response code such as 404 or 200, header data, and the response data, which can consist of an HTML page, an image, and so on. Methods for setting the response code and headers are encapsulated in a `javax.servlet.http.HttpServletResponse`. The response data is written directly through a `javax.servlet.ServletOutputStream`, which can be obtained from the `HttpServletResponse`.

Servlet Interface

A servlet must implement the `javax.servlet.Servlet` interface. This interface defines the `service` method that is called to handle a request:

```
void service (ServletRequest, ServletResponse)
  throws ServletException, IOException
```

HttpServletRequest

The `HttpServletRequest` breaks down a request into parsed elements, such as request URI, query arguments and headers. Various `get` methods allow you to access various parts of the request:

- requestURI
- Parameters
- Attributes
- ServletInputStream

requestURI

The `requestURI` deals with the URL sent by the browser. For example:

```
http://server:80/MyWebApplication/personal/info/top.html?info=intro
```

When the server receives this request, the `http://server:80` is stripped from the URL, leaving two parts:

- `requestURI`
- `queryString`: the string that follows a question mark (?), set to null if there are no query arguments.

In the previous example, the `requestURI` and `queryString` are `/MyWebApplication/personal/info/top.html` and `info=intro`, respectively.

The URI `/MyWebApplication/personal/info/top.html` is further split up into `contextPath`, `servletPath` and `pathInfo`. This distinguishes the path to a file or other data from the prefix that indicates who handles the request. In this case, the `/MyWebApplication` might be the `contextPath`, `/personal` might act as the `servletPath`, while the `/info/top.html` represents the `pathInfo`.

The `contextPath` is the name of the J2EE web application accessed by the `requestURI`. One or more `contextPaths` can be defined for a web application in the `application.xml` file.

The `pathInfo` is usually translated to a real file path by appending it to a document root or web application root. This real file path is available through `getPathTranslated`.

Given the earlier request, the following methods are provided to access the request URI and query string:

Method	Returns ...
`GetRequestURI`	`/MyWebApplication/personal/info/top.html`
`GetContextPath`	`/MyWebApplication`
`GetServletPath`	`/personal`
`GetPathInfo`	`/info/top.html`
`GetPathTranslated`	`/www/docs/info/top.html`
`GetQueryString`	`info=intro`
`GetRequestURIWithQueryString`	`/personal/info/top.html?info=intro`

The following equations describe the relationships among these properties:

```
requestURI = contextPath + servletPath + pathInfo
pathTranslated = documentRoot + pathInfo
```

Notice that `contextPath`, `servletPath`, and `pathTranslated` require additional information. For example, to determine the `pathTranslated` from the `pathInfo`, the web server must determine the document root. The web server uses the application's `application.xml` file to recognize the `contextPath`. Or to split the `servletPath` from the `pathInfo`, the web server needs to know what prefixes are to be treated specially, such as `/personal`. Other requests might not have a `contextPath` or `servletPath`, and the `pathInfo` is not split up at all.

The web server is not expected to know all of this information. The web server figures out what it can and leaves the rest blank. For example, the web server might leave the `pathTranslated` and `servletPath` blank. Servlets in the pipeline are given the responsibility of determining `pathTranslated`, and splitting `servletPath` from `pathInfo`.

Parameters

`HttpServletRequest` methods let you access request parameters. The request type determines where the parameters come from. In most implementations, a GET request obtains query string parameters, while a POST request obtains parameters from the posted arguments.

The methods `getParameter()`, `getParameterValues()`, and `getParameterNames()` let you access these arguments. For example, in a GET request with a query string of `info=intro` the call `getParameter("info")` returns `intro`.

Note: If you submit a form with `method="POST"`, the method `ServletUtil.getDynamoRequest.getParameter` does not return parameter values for query parameters. You must call `ServletUtil.getDynamoRequest.getQueryParameter` to get query arguments in pages that might get hit from a POSTed form

Attributes

The request object defines a method called `getAttribute()`. The servlet interface provides this as a way to include extra information about the request that is not covered by any of the other `HttpServletRequest` methods.

A servlet in the pipeline can use attributes as a way to annotate the request with additional computed information. For example, a servlet in the pipeline might be responsible for reading a cookie from the request, finding a session object corresponding to that cookie, and making that session object available to subsequent servlets in the pipeline. The servlet can do this by adding the session object as an attribute using a well-known attribute name. Subsequent servlets can extract the session object using that name.

ServletInputStream

The `ServletInputStream` is an `InputStream` that allows your servlets to read all of the request's input following the headers. For example, the `ServletInputStream` can be used to read the incoming submission from a POST argument.

All servlets in the pipeline share the same `ServletInputStream`, so if one servlet reads from the stream, the data that is read is no longer be available for other servlets. Certain operations also perform an implicit read on the `ServletInputStream`. For example, it was mentioned earlier that in a POST request, the calls to `getParameter` return values taken from the posted data. This implies that the posted data has already been read, and is no longer available through the `ServletInputStream`. It is instead made available through the parameters.

In general, you should expect to read POST data through parameters rather than the `ServletInputStream`. The `ServletInputStream` is more useful for reading other forms of request data.

HttpServletResponse

The `HttpServletResponse` can perform these tasks:

- Set response codes
- Set headers
- Send response codes and headers
- Send redirects
- Set ServletOutputStream

Set Response Codes

The response code for a request is a numeric value that represents the status of the response. For example, 200 represents a successful response, 404 represents a file not found, and so on. The `setStatus()` methods can be used to set the response code. `HttpServletResponse` defines a number of constants for the various codes—`SC_OK` for 200, `SC_NOT_FOUND` for 404, and so on.

By default, a response is automatically set with a response code of `SC_OK`. The response code can be changed.

Set Headers

Headers for the response can be set by calling `setHeader`, specifying the name and value of the header to be set. If you want to set more than one HTTP header with the same name, you can call `addHeader`, `addDateHeader`, or `addIntHeader`. You might want to do this if, for example, you wanted to set more than one cookie in a single response.

Send Response Codes and Headers

The response code and headers might not be sent immediately upon calling `setStatus` or `setHeader`. Typically, the response code and headers are not committed until something is actually written to the `ServletOutputStream`. A call to the `ServletResponse.isCommitted ()` method lets you know whether the response codes and headers were sent. If nothing is ever written to the `ServletOutputStream`, the response code and headers are committed when the request is finished.

You should not call `setHeader` or `setStatus` after you write something to the `ServletOutputStream` as the response might already be committed.

A couple of other methods can cause the response code and headers to be sent immediately. Calling `sendError` instead of `setStatus` sets the status code and immediately writes the response code and any headers set up to that point. Calling `sendRedirect` or `sendLocalRedirect` has the same effect.

Send Redirects

The `sendRedirect` method is used to issue a redirect to the browser, causing the browser to issue a request to the specified URL. The URL passed to `sendRedirect` must be an absolute URL—it must include protocol, machine, full path, and so on.

If you are redirecting to another page on the same site, you should call `sendLocalRedirect` instead of `sendRedirect`. Unlike `sendRedirect`, the `sendLocalRedirect` method lets you specify a relative URL, such as `errors/LoginError.jsp`. The `sendLocalRedirect` method also includes session information in the location URL, which is required to maintain a user's session across a redirect.

After calling `sendRedirect` or `sendLocalRedirect`, no other operations should be performed on the response. If you use `response.sendRedirect()` or `response.sendLocalRedirect()` calls, they must be made before any content has been output to the `response.getOutputStream()`. After you send content to the output stream, the response headers are already sent and it is no longer possible to modify the response headers to perform a redirect.

This means that you cannot have any content, including any white space, in a JSP before the redirect call is performed. White space is treated as content of the page unless it is between <% and %> tags or between `<dsp:droplet>` and `</dsp:droplet>` tags (and not in an `<dsp:oparam>` tag).

Here is an example of a redirect that does not work, because it includes white space in the `<dsp:oparam>` tag before the <% tag:

```
------ top of the page:
<dsp:droplet name="/atg/dynamo/droplet/Switch">
  <dsp:param bean="FormHandler.shouldRedirect" name="value"/>
  <dsp:oparam name="true">
     <% ServletUtil.getDynamoResponse(request,response).sendLocalRedirect
     ("/error.jsp", request); %>
  </dsp:oparam>
</dsp:droplet>
```

Here is the same example coded so that it does work:

```
------ top of the page:
<dsp:droplet name="/atg/dynamo/droplet/Switch">
  <dsp:param bean="FormHandler.shouldRedirect" name="value"/>
  <dsp:oparam name="true"><% ServletUtil.getDynamoResponse(request,response).
sendLocalRedirect
     ("/error.jsp", request); %>
  </dsp:oparam>
</dsp:droplet>
```

Set ServletOutputStream

The `ServletOutputStream` is obtained by calling `getOutputStream` on the `HttpServletResponse`. The `ServletOutputStream` is a subclass of `OutputStream` that contains a number of convenient `print` and `println` methods.

Data written to the `ServletOutputStream` goes straight back to the browser. In addition, the first data written to the stream causes the response code and headers to be sent out, which means that the headers cannot be changed after data has been written to the `ServletOutputStream`. The `ServletOutputStream` cannot be used to print headers, response codes, or redirect commands. These must be performed by using the appropriate `HttpServletResponse` methods.

In the servlet pipeline, all servlets in the pipeline generally share the same `ServletOutputStream`. So if one servlet prints something to the stream, the next servlet in the pipeline prints something to the stream, both outputs appear in the order they were printed.

Note: This is different from the servlet chaining function provided by some web servers. In servlet chaining, the output from one servlet becomes the input of the next servlet, and so on. In the servlet pipeline model, the servlets in the pipeline share the same input and output streams, which lead back to the browser.

DynamoHttpServletRequest and Response

One of the functions of the servlet pipeline is to modify a request as it runs through various processing elements. For example, one pipeline element might find the file associated with a given `pathInfo`, and use that file name to set the `pathTranslated` property of the request.

The `HttpServletRequest` interface is immutable—it only provides methods for reading the various properties, but does not provide methods for setting those properties. ATG provides the class `atg.servlet.DynamoHttpServletRequest`, which implements `HttpServletRequest` and provides methods to change request properties, such as `setPathInfo` and `setPathTranslated`.

Similarly, the class `atg.servlet.DynamoHttpServletResponse`, which implements `HttpServletResponse`, lets you change response properties such as the output stream, and access its values such as `statusCode`.

The very first element of the servlet pipeline converts an incoming `HttpServletRequest/Response` pair into a `DynamoHttpServletRequest/Response` pair. This allows subsequent elements of the servlet pipeline to use the additional functions provided by `DynamoHttpServletRequest/Response`. These functions are outlined below.

DynamoHttpServletRequest

As mentioned previously, `DynamoHttpServletRequest` provides methods for setting the various fields of the request, making these changed values visible to subsequent elements in the pipeline. The request is available as a Nucleus component at `/OriginatingRequest`.

In addition to property setters, `DynamoHttpServletRequest` provides methods for adding attributes to the request, which lets you annotate the request with additional information. The `DynamoHttpServletRequest` also provides a way to attach attribute factories to the request, which allows you delay the computation of attributes until they are first needed.

Finally, `DynamoHttpServletRequest` provides a way to attach permanent attributes to the request. These are attributes that stay around from request to request. These permanent attributes allow you to reuse objects, which is the key to getting high throughput out of the servlet pipeline.

These features are described in the following topics:

- Request Property Setters
- OriginatingRequest Component
- Request Attributes
- Attribute Factories
- Permanent Attributes

Request Property Setters

`DynamoHttpServletRequest` offers the following `setX` methods that allow you to change the properties of a request:

```
setAuthType
setContentLength
setContentType
setInputStream
setMethod
setPathInfo
setPathTranslated
setProtocol
setQueryString
setRemoteAddr
setRemoteHost
setRemoteUser
setRequestURI
setScheme
setServerName
setServerPort
setServletPath
```

These methods are derived from the base class, `atg.servlet.MutableHttpServletRequest`.

In addition, `DynamoHttpServletRequest` offers its own `setX` methods, such as:

```
setBaseDirectory
setRequestLocale
setMimeType
setSession
```

If you set a property with one of these `setX` methods, subsequent calls to the corresponding `getX` method return the value that you set. These new values are also visible to servlets farther down in the pipeline.

OriginatingRequest Component

In addition, the current request is available in Nucleus as `/OriginatingRequest`. The HTTP headers of the request are available as properties of this Nucleus component. This lets you get the value of the HTTP REFERER header like this, for example:

```
<dsp:valueof bean="/OriginatingRequest.referer"/>
```

Request Attributes

You can also add arbitrary keyword/value mappings to the request. These mappings are called attributes. They are added to the request by calling `setAttribute`. After an attribute has been added, it can be retrieved by calling `getAttribute`. These attributes are visible to your own servlets and servlets farther down the pipeline.

Attributes are often used to annotate a request with information derived from the request. For example, an attribute might hold the values of the cookies that came with the request, represented as a `Dictionary` of cookie name/cookie value pairs. The entire `Dictionary` is added as an attribute using a well-known attribute name, and subsequent servlets in the pipeline can access the `Dictionary` of cookies by retrieving that attribute by name.

After a request has been completed, all attributes are cleared from the request before the next request begins.

Attribute Factories

One of the techniques that can be used to improve performance is to avoid calculating values unless they are needed. The previous section described how a `Dictionary` of cookies might be useful as an attribute. But if only 10 percent of the requests actually use that `Dictionary`, 90 percent of the requests waste cycles calculating that `Dictionary`.

The `DynamoHttpServletRequest` lets you register an attribute factory for an attribute. This attribute factory is able to compute the value of the attribute when it is needed. When `getAttribute` is called on a request, the request determines if the value of the attribute has already been set. If not, the request checks to see if an attribute factory has been registered for that attribute. If so, the attribute factory is called to generate the value of the attribute. The generated value is then registered as an attribute and is available for subsequent calls to `getAttribute`.

So for the cookies case, register an attribute factory with the request. The attribute factory can create the `Dictionary` of cookies the first time the cookies attribute is accessed.

The attribute factory must be of type `atg.servlet.AttributeFactory`, which defines a single method `createAttributeValue`. An attribute factory is registered by calling `setAttributeFactory`.

Like attributes, all attribute factories are cleared from the request after a request has been completed.

Permanent Attributes

Perhaps the most important technique for achieving high performance is reuse of objects. In Java, every object creation is expensive, and every object creation also has a delayed cost in garbage collection, so reusing objects is a guaranteed way to improve performance.

The `DynamoHttpServletRequest` provides a way for you to register permanent attributes. Unlike normal attributes, permanent attributes are not cleared between requests, meaning that these permanent attributes are available for reuse by multiple requests.

For the cookie example, the `Dictionary` used to hold the cookies might be stored as a `Hashtable` that is a permanent attribute of the request. Instead of creating a `Hashtable` for each request, the permanent `Hashtable` attribute can be extracted from the request, cleared, and reused.

Adding permanent attributes to the request uses a slightly different process from adding normal attributes. Instead of having separate `get` and `set` methods, permanent attributes are only accessed by a `getPermanentAttribute` method. The `getPermanentAttribute` must be passed an `AttributeFactory`. This `AttributeFactory` serves two purposes: it acts as the key for the attribute, and it is used to create the attribute if it has not already been created.

The following shows how you might use a permanent attribute to store the `Hashtable` for the cookies:

```
// Member variables
class CookiesFactory implements AttributeFactory {
  public Object createAttributeValue ()
  { return new Hashtable (); }
}
AttributeFactory cookiesKey = new CookiesFactory ();
...

public void service (DynamoHttpServletRequest request,
                     DynamoHttpServletResponse response)
  throws ServletException, IOException
{
  Hashtable cookies = (Hashtable)
    request.getPermanentAttribute (cookiesKey);
  cookies.clear ();
  ...
}
```

The first part of the example shows how to define an `AttributeFactory` inner class that both creates the `Hashtable` when needed, and also acts as the key. In the method where you need to extract the `Hashtable`, you call `getPermanentAttribute`, passing it the `AttributeFactory`. The first time you call this on the request, it fails to find a value associated with that key, and calls on the `AttributeFactory` key to create the value. Subsequent calls on that same request find and return the value that was previously registered with that key. This value remains part of the request even after the request is complete and a new request begins.

Each request object gets its own copy of your permanent attribute, so if your server is running 40 request-handling threads, you might see the permanent attribute get created 40 times, once for each request object. But after all request objects have a copy of the attribute, no more object creations are necessary.

DynamoHttpServletResponse

The `DynamoHttpServletResponse` is a wrapper around `HttpServletResponse` that adds a few useful methods:

getHeader

Returns a Dictionary of all the headers that were set so far in the response.

setOutputStream

Sets the ServletOutputStream that subsequent servlets use to write their responses. You can use this to intercept the outgoing data and process it in some way, before sending it to the original output stream. For example, a caching element might set the output stream to send data both to a cache and to the browser.

isRequestComplete

Returns true if a complete response is already sent. This is typically only true if a redirect or error has been sent through this response.

getStatus

Returns the status code sent through this response object. If no status code has been set explicitly, `SC_STATUS_OK` is returned.

Accessing DynamoHttpServletRequest and DynamoHttpServletResponse

To access information contained in the request and response in your page, do so by making direct calls to `HttpServletRequest` and `HttpServletResponse`. When you need application-specific information held only by the Dynamo request and response, you should import the request or response using the `atg.servlet.ServletUtil` class. For example, to access the `state` object parameter, your JSP might use this code:

```
<%=atg.servlet.ServletUtil.getDynamoRequest(request).getObjectParameter("state")%>
```

Any references to the Dynamo request and response are interpreted as calls to the generic `HttpServletRequest` and `HttpServletResponse`.

Filters and PageFilter

Another way to alter a request or response is through the use of filters. A filter, as it is defined in the Java Servlet Specification v2.3, implements the `javax.servlet.Filter` interface. You use a filter to create a wrapper for the request and response in order to modify the data within it. You can also use a filter to examine the headers in the request and to specify the next resource to call.

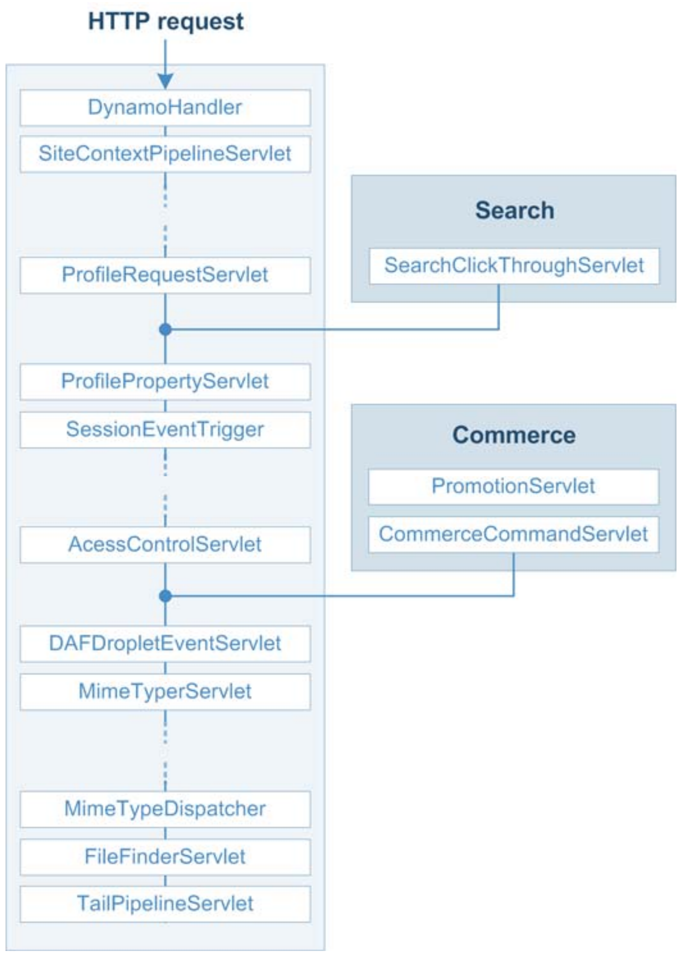
A series of filters are managed by a filter chain. After a filter completes execution, it makes a call to the filter chain. The filter chain is responsible for determining the next operation: invoking another filter, halting the request execution, throwing an exception, or calling the resource that passed the request to the first filter in the chain.

Nucleus-based web applications use one filter, `PageFilter`, by default. For information on how to implement `PageFilter`, see Starting the Request-Handling Pipeline.

Request-Handling Pipeline Servlets

The standard request handling pipeline configuration for an ATG server comprises various servlet pipeline components that perform various operations on each request. The servlets included in the pipeline vary according to the modules that are assembled into the application.

The following graphic provides a truncated view of the servlet pipeline as it might be assembled for an application that includes ATG Commerce, and enabled for multisite. The main pipeline includes standard platform servlets. The ATG Commerce and ATG Search modules insert their servlets at different points on the main pipeline as required:



You can use the ATG Dynamo Server Admin Component Browser to view request handling pipeline servlets and their sequence within the pipeline:

1. In the Component Browser (`http://`*`host`*`:`*`port`*`/dyn/admin/nucleus/`), navigate to the first pipeline servlet:

   `/atg/dynamo/servlet/dafpipeline/DynamoHandler`

2. The Component Browser lists all pipeline servlets in order of execution.

The following table lists servlets according to their likely order in a production server's request handling pipeline. The servlets actually contained in a given request handling pipeline and their order is likely to vary, depending on the application.

For detailed information about these and other available servlets, see Appendix E, Request Handling Pipeline Servlets Reference.

Core Platform Servlets	Module servlets	Module
DynamoHandler		
SiteContextPipelineServlet		
ThreadUserBinderServlet		
DAFPassportServlet		
PathAuthenticationServlet		
URLArgumentServlet		
DynamoServlet		
ProtocolSwitchServlet		
ProfileRequestServlet		
ThreadNamingPipelineServlet		
	SearchClickThroughServlet	ATG Search
ProfilePropertyServlet		
SessionEventTrigger		
PageViewServletTrigger		
SessionSaverServlet		
SiteSessionEventTrigger		
AccessControlServlet		
	PromotionServlet	ATG Commerce
	CommerceCommandServlet	ATG Commerce
DAFDropletEventServlet		
MimeTyperServlet		
ExpiredPasswordServlet		
CookieBufferServlet		
MimeTypeDispatcher		
FileFinderServlet		
TailPipelineServlet		

Customizing a Request-Handling Pipeline

The ATG installation provides a servlet pipeline that is invoked each time an ATG server handles a request. ATG Dynamo Server Admin also has its own servlet pipeline, which starts with the servlet `/atg/dynamo/servlet/adminpipeline/AdminHandler`. You can construct pipelines used by your own applications, or you can customize existing ATG server pipelines.

Inserting Servlets in the Pipeline

The `atg.servlet.pipeline` package provides interfaces for creating request handling pipeline servlets. All pipeline servlet classes directly or indirectly implement interface `atg.servlet.pipeline.PipelineableServlet`. This interface provides a `nextServlet` property that points to the next component in the pipeline. The ATG installation provides the implementation class `atg.servlet.pipeline.PipelineableServletImpl`, which you can subclass to create your own servlets. `PipelineableServletImpl` implements all `Servlet` methods, so you only need to override the `service` method.

To insert into a request handling pipeline a servlet that subclasses `PipelineableServletImpl`:

1. Extend `atg.servlet.pipeline.PipelineableServletImpl`.
2. Define the servlet as a globally scoped Nucleus component.
3. Reset the previous servlet's `nextServlet` property to point to the new servlet.
4. Set the new servlet's `nextServlet` property to point to the next servlet in the pipeline.
5. Add the servlet's path to the `initialServices` property of `/atg/dynamo/servlet/Initial`.

The `PipelineableServlet` interface has two sub-interfaces that provide more flexibility for inserting new servlets into the pipeline:

- `atg.servlet.pipeline.InsertableServlet`
- `atg.servlet.pipeline.DispatcherPipelineableServlet`

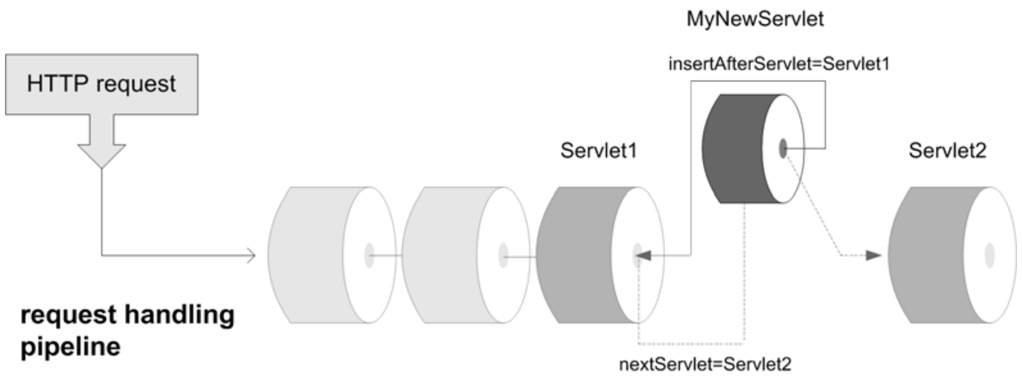
InsertableServlet

The InsertableServlet interface lets a servlet insert itself into the pipeline when the service starts, without requiring changes to servlets already in the pipeline, through the `insertAfterServlet` property, which points back to the preceding servlet. The inserted servlet reads the preceding servlet's `nextServlet` property and points to it as the next servlet to execute after itself.

For example, a servlet pipeline might contain Servlet1, whose `nextServlet` property points to Servlet2. You can insert MyNewServlet, which implements `InsertableServlet`, between the two by setting its `insertAfterServlet` property so it points to Servlet1. This configuration splices ServletNew between Servlet1 and Servlet2 as follows:

- MyNewServlet sets its own `nextServlet` property to the value of Servlet1's `nextServlet` property.

- MyNewServlet reads Servlet1's `nextServlet` property and links to Servlet2 as the next servlet to execute after itself.



If an insertable servlet inserts only itself into a pipeline, it uses the `nextServlet` property of its `insertAfterServlet` servlet to resume pipeline execution. However, if the inserted servlet starts a secondary pipeline, it sets its own `nextServlet` property to the next servlet in that pipeline. After the last secondary pipeline servlet executes, it passes control to the `nextServlet` servlet originally specified by the `insertAfterServlet` servlet, and the original pipeline resumes execution.

The ATG installation provides an implementation of the InsertableServlet interface, `atg.servlet.pipeline.InsertableServletImpl`. This class implements all `Servlet` methods, so you only need to override the `service` method.

To add an `InsertableServlet` to the servlet pipeline:

1. Write your servlet by extending `atg.servlet.pipeline.InsertableServletImpl`.
2. Define the servlet as a globally scoped Nucleus component.
3. Set the `insertAfterServlet` property of your servlet to point to the path of the pipeline servlet you want your servlet to follow. For example, you can insert a servlet after `DynamoServlet` as follows:

   ```
   insertAfterServlet=/atg/dynamo/servlet/dafpipeline/DynamoServlet
   ```

4. Add the servlet's path to the `initialServices` property of `/atg/dynamo/servlet/Initial`:

   ```
   initialServices+=/myServlet
   ```

When the inserted servlet finishes processing, it calls the method `passRequest()` (defined in `InsertableServletImpl`), which automatically passes the request and response objects to the next servlet in the pipeline.

Sample Servlet Code

The following pipeline servlet class `URIPrinter` extends `atg.servlet.pipeline.InsertableServletImpl`. It prints the request URI before passing the request on to the next servlet in the pipeline:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import atg.servlet.*;
import atg.servlet.pipeline.*;

public class URIPrinter extends InsertableServletImpl{
  public URIPrinter () {}
  public void service (DynamoHttpServletRequest request,
                       DynamoHttpServletResponse response)
      throws IOException, ServletException
  {
    System.out.println ("Handling request for " +
                        request.getRequestURI ());
    passRequest (request, response);
  }
}
```

Note: Subclasses of `InsertableServletImpl` that add their own logic to `doStartService` must call `super.doStartService()`.

DispatcherPipelineableServlet

The `DispatcherPipelineableServlet` interface provides a mechanism for conditionally branching the pipeline. This interface includes a `dispatcherServiceMap` property that is a Map of possible servlets to invoke next, depending on some condition. For example, the `MimeTypeDispatcher` servlet determines which servlet to invoke depending on the MIME type of the request. ATG provides the implementation class `DispatcherPipelineableServletImpl`.

Using J2EE Servlets and Filters

The servlets discussed in this chapter are primarily ATG servlets created for use only in Nucleus. ATG servlets are distinct from J2EE servlets, which run in a J2EE web container and follow the standards defined by the J2EE specifications. While J2EE servlets and filters can interact with requests much like ATG servlets, they differ in key respects:

- ATG servlets exist in the servlet pipeline, which executes before the request reaches the J2EE web container. J2EE servlets are executed by the web container.
- ATG servlets themselves determine the order in which they execute. The application deployment descriptor `web.xml` describes the order and conditions in which J2EE servlets execute.

Use the type of resources that best suit your preferences. You might find J2EE servlets and filters a more portable and familiar technology in comparison to ATG servlets.

The J2EE specifications describe how to create J2EE servlets and filters. For information on how to implement J2EE resources in the ATG platform, see Accessing Nucleus in a Web Application.

Exceptions in Pipeline Servlets

If you write a servlet and add it to the servlet pipeline, your servlet's `service` method is called for all requests that reach this stage in the pipeline. If your servlet does not call the `passRequest()` method (perhaps because an application exception is thrown), no content reaches the browser and the likely result is a `Document Contains No` Data message from the browser. Make sure your servlets are coded properly to avoid this problem.

Your servlet (whether it appears in the servlet pipeline or not) should generally not catch `IOExceptions` that occur when writing to the `ServletOutputStream`, as those exceptions indicate that the user has clicked the browser's stop button. If you need to execute some code when the user clicks the stop button, your code should catch the `IOException`, do any needed processing, and then re-throw the `IOException`.

Authentication

The `BasicAuthenticationPipelineServlet` class provides authentication using the Basic HTTP authentication mechanism. A component for this servlet is not included in the standard servlet pipelines, but the class is available for use in servlet pipelines you might create in your own applications.

If a request comes in without an authorization header, this servlet immediately sends back a reply that causes the browser to pop up an authorization window. The user is expected to enter a user name and password. The request is then repeated, this time with an authorization header. The servlet checks that the user name and password in the header are valid. If so, the servlet passes the request to the next servlet in the pipeline. Subsequent requests contain the correct authorization and no longer cause the authorization window to pop up. The request is never passed on if the correct authorization is not received.

Checking the user name and password is performed by a separate component that implements `atg.servlet.pipeline.Authenticator`. This defines a single method called `authenticate`, that takes a user name and password and returns `true` if the combination is valid, `false` if not. ATG provides an implementation of `Authenticator` called `atg.servlet.pipeline.BasicAuthenticator`. This takes a `passwords` property of type `Properties`, that maps user IDs to passwords. If a user ID/password combination is found in the `passwords` property, the authentication is successful. Otherwise, the authentication fails. Other `Authenticator` implementations are possible, such as implementations that check names and passwords in a database.

Example

The following example shows how to configure an authentication servlet and authenticator:

`AuthenticationServlet.properties`:

```
$class=atg.servlet.pipeline.BasicAuthenticationPipelineServlet
realm=Dynamo6.0
authenticator=Authenticator
nextServlet=SomeHandler
```

`Authenticator.properties`:

```
$class=atg.servlet.pipeline.BasicAuthenticator
passwords=\
        admin=jjxr2,\
        hank=angry
```

In this example, the authentication servlet passes a request to `SomeHandler` only if the request is authenticated with a name and password found in the `passwords` property of the authenticator component. The `realm` property specifies what realm is to be shown to the user in the window that asks for name and password.

BrowserTyper

One service made available by `DynamoHttpServletRequest` is the `BrowserTyper`. The `BrowserTyper` service enables an ATG server to identify a visitor's web browser type and group it into one or more categories. This service (a component of class `atg.servlet.BrowserTyper`) identifies browsers by the user-agent header field in the request. The `BrowserTyper` manages a list of browser types, each of which has a name that identifies the browser and a list of patterns that the `BrowserTyper` uses for matching user-agent fields to browser types.

The list of browser types is found in `/atg/dynamo/servlet/pipeline/BrowserTypes`. It includes all commonly used browsers, and groups them according to several different attributes, including, among others:

- the vendor name
- whether the browser can handle frames or cookies
- whether the request is from a non-browser user-agent (a web robot)
- whether the browser supports file uploading

Each browser type is defined as a component in the `/atg/dynamo/servlet/pipeline/BrowserTypes` directory. These components include two properties: the `name` of the browser type and the `patterns` in the user-agent header by which the `BrowserTyper` attempts to recognize the browser type.

You can add to the list of browser types that the `BrowserTyper` can recognize. To do this:

1. Add the names of your browser types to the `browserTypes` property of the `BrowserTyper` component in `/atg/dynamo/servlet/pipeline`, like this:

   ```
   browserTypes+=\
           BrowserTypes\MyFirstBrowserType,\
           BrowserTypes\MySecondBrowserType
   ```

2. Create an `atg.servlet.BrowserType` class component in the `/atg/dynamo/servlet/pipeline/BrowserTypes` directory for each additional browser type.
3. The `.properties` file of the `BrowserType` component should look like this:

   ```
   $class=atg.servlet.BrowserType
   ```

```
name=MyFirstBrowserType
patterns=\
          regular-expression-1,\
          regular-expression-2
```

4. The `patterns` property is a list of simplified regular expressions that are matched against the user-agent header for the request. If any of them match, the `isBrowserType()` method of `atg.servlet.BrowserTyper` returns `true`.

5. The simplified regular expression string has the following form:

```
<regexps> = empty |
     <regexps> <regexp>

<regexp> =
     <base type>
     <type set>
     <regexp>*
     <regexp>+
     <regexp>?
     <regexp>.
     <regexp>|<regexp>
     (<regexps>)

<base type> =
     any character other than:
       * + ? | ( ) [ ] .

<type set> =
     [<base types>]

<base types> = empty |
     <base types> <base type>

'*' 0 or more times
'+' 1 or more times
'?' 0 or 1 time
'.' Matches any character except \n
'|' Separates alternatives, e.g. [a | b | c] is 'a or b or c'
[ ] Join multiple expressions into one expression
( ) Group expressions
```

Browser Caching of Dynamic Pages

Some browsers handle page caching in a way that conflicts with dynamic page requests. ATG's browser typer marks page requests from those browsers as `non-cacheable` to override the aggressive caching behavior of some browsers and proxy servers. Because an ATG server does not set a `Last-modified` header for JSP requests, browsers should not cache results. However, some browsers (such as Microsoft IE 5.0) do cache these pages. Thus, these browsers might display stale content to users on your site. This occurs because of bad caching: instead of re-requesting the JSP, the browser incorrectly displays the cached version. In addition to showing potentially stale content, URL-based session tracking breaks with these browsers.

To prevent browsers from caching dynamic pages, an ATG server sends headers to these browsers with the following:

```
Pragma: no-cache
Expires: date-in-the-past
```

This behavior is controlled with a special ATG browser type called `bad-cacher` defined by the following component:

```
/atg/dynamo/servlet/pipeline/BrowserTypes/BadCacher
```

This component has a `patterns` property that defines a regular expression that matches the `user-agent` header sent by the browser. If the `user-agent` matches, the `Pragma: no-cache` and `Expires:` *`date-in-the-past`* headers are sent with each request. By default, Microsoft IE 5.0 is listed as one of these browsers. You can control the list of `user-agents` where caching is disabled by editing the values of the `BadCacher` component's `patterns` property.

BrowserAttributes Component

ATG includes a request-scoped component at `/atg/dynamo/servlet/pipeline/BrowserAttributes` that exposes all the known `BrowserTyper` characteristics of the current request as boolean properties.

This component enables you to create JSPs that display different features, depending on properties like the browser type, so you can include browser-specific or feature-specific code in your pages without resorting to embedded Java tags to test the browser type.

The following example tests whether the request comes from an Internet Explorer browser:

```
<dsp:droplet name="/atg/dynamo/droplet/Switch">
  <dsp:param bean="BrowserAttributes.MSIE" name="value"/>
  <dsp:oparam name="true">
    Hmmm... you seem to be using Internet Explorer.
  </dsp:oparam>
  <dsp:oparam name="false">
    You aren't using Internet Explorer.
  </dsp:oparam>
</dsp:droplet>
```

PageFilterUtil

The `atg.servlet.pagefilter.PageFilterUtil` class lets you encode URLs in HTML dynamically. When an HTML file is read from the input stream, the data is written to the response and, at the same time, URLs found in the HTML are replaced with encoded versions. For improved performance, an offset table can be generated the first time a given HTML file is parsed. Subsequent requests use the offset table to locate the URLs.

Any of the following operations can be accomplished by incorporating `PageFilterUtil` in a JSP tag or custom servlet bean:

- Session IDs appended to URLs for session tracking are striped from the URLs when the page containing them is rendered.
- Relative URLs are rewritten with a prepending forward slash (/) so they are recognized as such by the browser.
- Query parameters appended to URLs are striped from URLs when the page containing them is rendered.
- URLs are appended to support exit tracking.

You can find `PageFilterUtil` in `<ATG10dir>\DAS\lib\classes.jar` so it is appended to your `CLASSPATH` by default. Design your classes to instantiate and reference `PageFilterUtil` as needed.

The `writeHTMLFile` method determines whether the page requires encoding and when needed, accomplishes this task before writing the page content to the response object. When a page does not require any URL Rewriting, page content is sent directly to the browser.

The `encodeHTMLFile` method calls `response.encodeURL` for every URL within a given page and writes the resultant page content to the response object. Because pages that do not require parsing and encoding follow the same process as those that do, you ought to use `writeHTMLFile` to speed page rendering when you are unsure of the content in your pages.

The remaining methods, `writeFile`, `writeByteFile`, and `writeCharFile`, pass a page to the response object directly. These methods assume encoding has already been applied to the page content, however you can specify encoding using these methods in order to parse the page as characters and write out the result through `PrintWriter`.

For more information see the `atg.servlet.pagefilter.PageFilterUtil` section of the *ATG API Reference*.

Improving Page Compilation Performance

The first time a page is requested, it undergoes several conversions: from JSP to Java code to HTML. The first transformation from JSP to Java code causes a slight delay in performance that is easily avoided by precompiling your JSPs with the Java compiler.

You can precompile individual JSPs at application startup by specifying them in your web application deployment descriptor. Here is an example of what you'd add for a page called `MyPage.jsp` to `web.xml` in enclosing <web-app> tags:

```
<servlet>
  <servlet-name>MyPage.jsp</servlet-name>

  <jsp-file>/path/to/jsp/MyPage.jsp</jsp-file>

  <load-on-startup>1</load-on-startup>
</servlet>
```

An explanation of each tag is as follows:

- `<servlet-name>` identifies the servlet to be compiled
- `<jsp-file>` identifies the path to the servlet
- `<load-on-startup>` identifies the order this servlet should be compiled

Servlet Pipeline Examples

The following examples show how to perform various functions with a pipeline servlet.

- Setting an Attribute
- Setting an Attribute Factory
- Setting a Permanent Attribute

Setting an Attribute

In this example, a pipeline servlet examines a request to see if it starts with `/stocks` or `/bonds`. If so, it sets a `wallStreet` attribute to `true`. Otherwise, it sets a `wallStreet` attribute to `false`.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import atg.servlet.*;
import atg.servlet.pipeline.*;

public class WallStreet extends PipelineableServletImpl {
  public WallStreet () {}
  public void service (DynamoHttpServletRequest request,
                       DynamoHttpServletResponse response)
       throws IOException, ServletException
  {
    String pathInfo = request.getPathInfo ();
    boolean val =
      pathInfo.startsWith ("/stocks") ||
      pathInfo.startsWith ("/bonds");
    request.setAttribute ("wallStreet", new Boolean (val));
    passRequest (request, response);
  }
}
```

This `wallStreet` attribute is now available for use by subsequent servlets. For example, the following servlet might follow the `WallStreet` servlet, printing a message if it finds the `wallStreet` attribute is `true`:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import atg.servlet.*;
import atg.servlet.pipeline.*;

public class Trader extends PipelineableServletImpl {
  public Trader () {}
  public void service (DynamoHttpServletRequest request,
                       DynamoHttpServletResponse response)
       throws IOException, ServletException
  {
    Boolean b = (Boolean) request.getAttribute ("wallStreet");
    if (b != null && b.booleanValue ()) {
      System.out.println ("I'm on Wall Street!");
    }
    passRequest (request, response);
  }
}
```

Setting an Attribute Factory

The sample pipeline servlet element described in the previous section has a problem: it always examines `pathInfo` and creates a new Boolean attribute, whether that attribute is needed or not. This attribute can be expensive to create and wasteful if the attribute is never accessed.

Rather than setting an attribute, this pipeline servlet would be more efficient if it set an attribute factory that creates the attribute value the first time it is needed. The following shows how to do this:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import atg.servlet.*;
import atg.servlet.pipeline.*;

public class WallStreet extends PipelineableServletImpl {
  // The AttributeFactory
  class WallStreetFactory implements AttributeFactory {
    DynamoHttpServletRequest request;
    public void setRequest (DynamoHttpServletRequest request)
    { this.request = request; }

    public Object createAttributeValue ()
    {
      String pathInfo = request.getPathInfo ();
      boolean val =
        pathInfo.startsWith ("/stocks") ||
        pathInfo.startsWith ("/bonds");
      return new Boolean (val);
    }
  }
  public WallStreet () {}
  public void service (DynamoHttpServletRequest request,
                       DynamoHttpServletResponse response)
       throws IOException, ServletException
  {
    WallStreetFactory f = new WallStreetFactory ();
    f.setRequest (request);
    request.setAttributeFactory ("wallStreet", f);
    passRequest (request, response);
  }
}
```

The `AttributeFactory` is defined as an inner class. Every time a request comes through, a new attribute factory is created and registered with the request. This factory is given a pointer to the request, so that when the factory is asked to create the attribute value, it can compute the value from the request.

Setting a Permanent Attribute

The previous example showed how a request can improve performance by delaying computation of the `wallStreet` attribute until it is needed. But there is still the problem that a `WallStreetFactory` is created on every request. This repeated creation can be avoided by using permanent attributes. In this example, the `WallStreetFactory` is stored as a permanent attribute that is accessed during the request.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import atg.servlet.*;
import atg.servlet.pipeline.*;

public class WallStreet extends PipelineableServletImpl {
  // The AttributeFactory
  class WallStreetFactory implements AttributeFactory {
    DynamoHttpServletRequest request;
    public void setRequest (DynamoHttpServletRequest request)
    { this.request = request; }

    public Object createAttributeValue ()
    {
      String pathInfo = request.getPathInfo ();
      boolean val =
        pathInfo.startsWith ("/stocks") ||
        pathInfo.startsWith ("/bonds");
      return new Boolean (val);
    }
  }

  // The permanent attribute
  class KeyFactory implements AttributeFactory {
    public Object createAttributeValue ()
    { return new WallStreetFactory (); }
  }
  KeyFactory key = new KeyFactory ();

  public WallStreet () {}
  public void service (DynamoHttpServletRequest request,
                       DynamoHttpServletResponse response)
       throws IOException, ServletException
  {
    WallStreetFactory f = (WallStreetFactory)
      request.getPermanentAttribute (key);
    f.setRequest (request);
    request.setAttributeFactory ("wallStreet", f);
    passRequest (request, response);
  }
}
```

Now the pipeline servlet performs no object allocations when a request comes through. And when the attribute is accessed, only a single `Boolean` value is created. More permanent attributes can be used to avoid even the creation of that `Boolean` value.

Notice how this implementation is approximately twice as large as the first implementation that just created a new attribute value with every request. Remember that all of this extra code improves performance—it reuses objects and delays computation of attributes until they are needed. So even though the code is longer, the performance should be better.

Dispatching Servlets

The `PipelineableServletImpl` provides a convenient method for specifying a `nextServlet` and for passing calls to that servlet. This provides a nice linear model of processing. There are many instances, however, where you might want to branch to one of many servlets based on the request. For example, if you want to call a servlet that is not a pipeline servlet, you must use an alternative method.

To do this, make sure your servlet implements the `java.servlet.RequestDispatcher` interface defined in the J2EE specifications. This interface creates a wrapper for your custom servlet. That wrapper specifies the servlets that come before and after your servlet. You also need to define your servlet in your `web.xml` deployment descriptor.

9 Multisite Request Processing

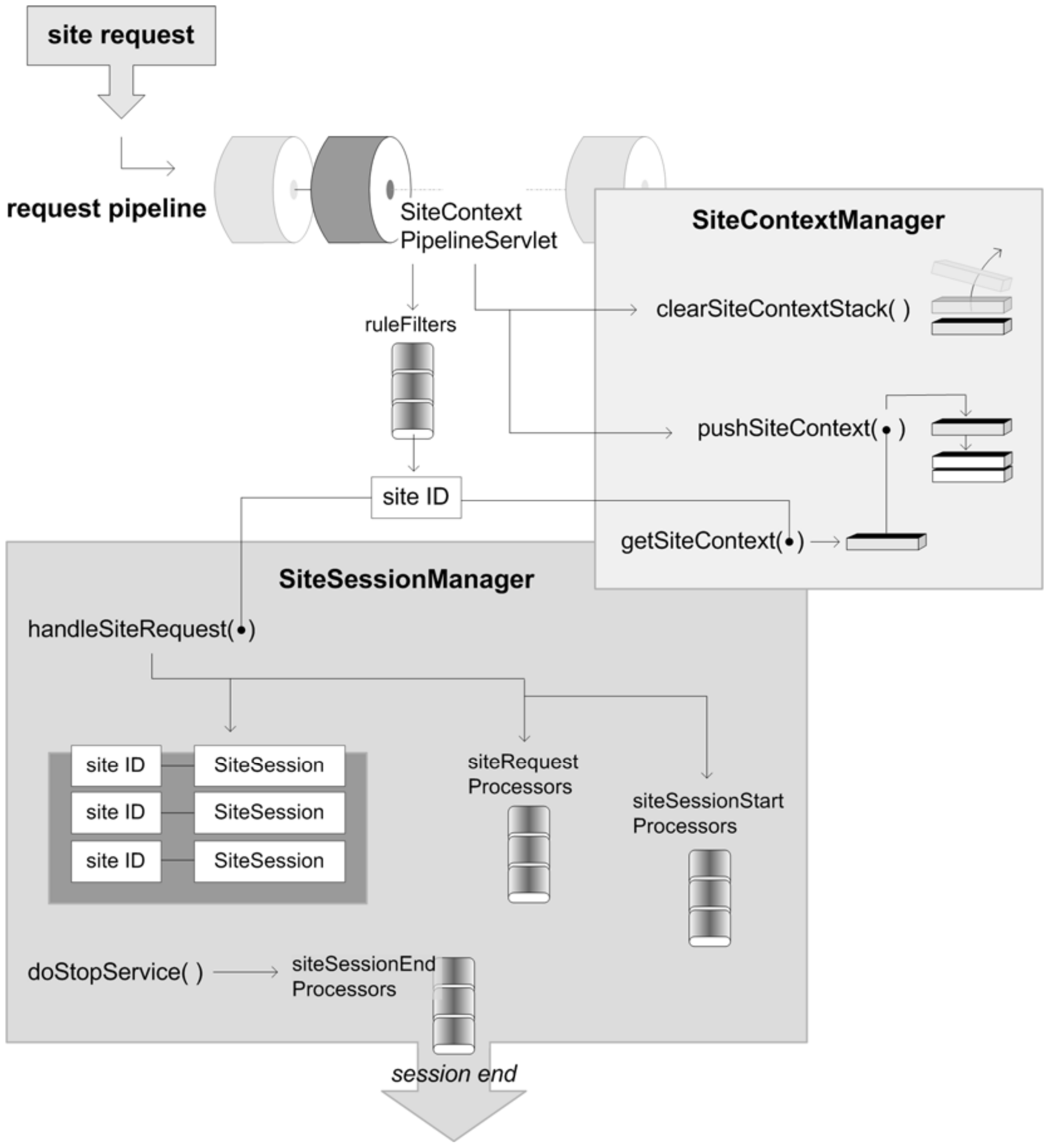
Note: For a general overview of multisite architecture, and detailed information about setting up a multisite environment, see the *ATG Multisite Administration Guide*.

When an ATG web server receives a request in a multisite environment, the request handling pipeline servlet SiteContextPipelineServlet (`atg.multisite.SiteContextPipelineServlet`) evaluates the request to determine the identity of the ATG site associated with it. That identity enables delivery of site-specific information in the server response. It is also important to maintain a site's identity in order to differentiate it from other ATG sites that the user might visit during the same session.

Broadly speaking, SiteContextPipelineServlet performs the following tasks:

1. Derives a site ID from the request URL.
2. Determines whether site information is available for requests.
3. Passes the site ID to the SiteContextManager and SiteSessionManager components:
    - The SiteContextManager (`atg.multisite.SiteContextManager`) creates a request-scoped SiteContext component, which gives the request thread access to site properties.
    - The SiteSessionManager (`atg.multisite.SiteSessionManager`) associates the site with a session-scoped SiteSession component, which maintains information about the site during the current session.

The following graphic illustrates this process and highlights components that play important roles. Later sections in this chapter describe these components in greater detail.



Site Identification

On receiving a request, the SiteContextPipelineServlet examines each request URL in order to determine which site to associate it with. This process comprises the following steps:

1. Iterate over an array of SiteContextRuleFilter components, or *rule filters*, which are set on the SiteContextPipelineServlet property `ruleFilters`.
2. Call each rule filter's `filter()` method until a site ID is returned for that request.

3. If none of these rule filters returns a site ID, call the rule filter that is set on the `DefaultRuleFilter` property, `DefaultSiteRuleFilter`.
4. If no rule filter returns a site ID, the SiteContextPipelineServlet stops processing and passes on the request to the next servlet in the request pipeline.

Errors

If a site ID is determined, but the SiteContextPipelineServlet cannot find a site configuration that corresponds to the site ID, it logs a warning, stops processing, and passes on the request to the next servlet in the request pipeline.

Installed Rule Filters

The ATG installation provides the following SiteContextRuleFilter components for identifying a site:

- RequestParameterRuleFilter: Evaluates query parameters that supply the site ID and specify whether that site should persist for the remainder of the current session.
- URLPatternMatchingRuleFilter: Encapsulates rules for obtaining a site ID from the request URL.
- DefaultSiteRuleFilter: Returns the server's default site ID.

RequestParameterRuleFilter

Based on the class `atg.multisite.PushedSiteParamFilter`, the component `/atg/multisite/RequestParameterRuleFilter` is the first rule filter to execute. This filter processes request query parameters that set the current site, and specify it as a *sticky site* that persists throughout the session of that request. This filter is typically useful for testing and previewing sites that are under development; it should be disabled for production sites.

RequestParameterRuleFilter checks the request URL for two query parameters:

- `pushSite` is set to a site ID, which is returned by the rule filter's `filter()` method.
- `stickySite`, if set to `setSite`, makes the `pushSite`-specified site sticky for the current session. Unless explicitly reset or unset, the sticky site is used for all subsequent requests during that session.

A sticky site remains valid for the current session until another request URL sets one of the following query parameters:

- `pushSite` specifies another site ID, which becomes the current site. If `stickySite` is also set to `setSite`, this SiteContext becomes the new sticky site.
- `stickySite` is set to `unsetSite`. This unsets the sticky site, and the RequestParameterRuleFilter returns null. The SiteContextPipelineServlet executes subsequent rule filters in its `ruleFilters` property until one returns a valid site.

RequestParameterRuleFilter is enabled through two properties:

- `enabled` specifies whether the filter is active within the filter chain. If set to `false`, SiteContextPipelineServlet skips over this filter when processing a request. By default, this property is set to `true`.

- `enableStickySite` disables sticky site functionality if set to `false`. The filter remains active and supports use of the `pushSite` query parameter; it ignores the `stickySite` query parameter. By default, this property is set to `false`.

  Note: Sticky site functionality is always enabled on preview servers through the SiteURLManager property `autoAppendStickySiteParams`. For more about the SiteURLManager, see Multisite URL Management later in this chapter.

If desired, you can change the names of the query parameters that RequestParameterRuleFilter expects by setting these properties:

- `pushSiteParamName`
- `stickySiteParamName`

Note: Changing these properties on an asset management or preview server might disrupt preview functionality.

URLPatternMatchingRuleFilter

Based on the class `atg.multisite.SiteContextRuleFilter`, the filter component `/atg/multisite/URLPatternMatchingRulefilter` encapsulates rules for obtaining a site ID from a request URL. The filter implements two algorithms for determining the site ID:

- Looks up the request URL in a map that is set in the filter's `URLs` property, which pairs URLs with site IDs.

  The `URLs` property setting lets a specific server substitute URL-to-site mappings that are otherwise set and managed by the SiteURLManager. This is generally useful for testing purposes, and is not typically used in a production environment.

- Passes the request URL to the method `SiteURLManager.getSiteIdForURL()`. This method obtains a site ID from the multisite URL management system, described elsewhere in this chapter.

Several Boolean properties determine whether the URLPatternMatchingRuleFilter is enabled and how it executes:

Property	**Description**
`enabled`	The filter is enabled.
`enableSimpleAlgorithm`	Use the `URLs` property to look up the URL request.
`enableSiteURLManagerAlgorithm`	Pass the request URL to the SiteURLManager for processing.

By default, all properties are set to `true`.

URLPatternMatchingRuleFilter also checks the request for the context parameter `atg.multisite.URLPatternMatchingEnabled`, which the application's `web.xml` file can set to `true` or `false`. If the parameter is set to `false`, the filter does not execute and returns null to the

SiteContextPipelineServlet. If `web.xml` omits this context parameter, the URLPatternMatchingRuleFilter behaves as if it were set to `true`.

DefaultSiteRuleFilter

Based on the class `atg.multisite.DefaultSiteContextRuleFilter`, the component `/atg/multisite/DefaultSiteRuleFilter` is configured by a single property, `defaultSiteId`, which identifies the server's default site ID. This filter executes after all filters that are specified in the `ruleFilters` property execute, and only if none of them returns a site ID.

Custom Rule Filters

To add your own rule filter:

1. Write the rule filter class.
2. Add the rule filter to the SiteContextPipelineServlet.

Write the Rule Filter Class

A custom rule filter class implements the SiteContextRuleFilter interface and its `filter()` method, which analyzes the request and returns the site ID. The `filter()` method has this signature:

```
public String filter(DynamoHttpServletRequest pRequest,
                     SiteSessionManager pSiteSessionManager)
```

Add the Rule Filter

Rule filters are managed by the SiteContextPipelineServlet. The servlet's `ruleFilters` property specifies the installed rule filters and their order of execution:

```
$class=atg.multisite.SiteContextPipelineServlet
$scope=global
siteContextManager=SiteContextManager
ruleFilters=\
      RequestParameterRuleFilter,\
      URLPatternMatchingRuleFilter

insertAfterServlet=/atg/dynamo/servlet/dafpipeline/DynamoHandler
```

If you use standard += notation to add your rule filter, it executes only after the installed rule filters. If any of these filters returns a site ID, your custom filter might not execute.

You can use Nucleus configuration to modify this behavior in two ways:

- Disable any installed rule filter that might preempt execution of custom rule filters by setting its `enabled` property to `false`.
- Override the SiteContextPipelineServlet's `ruleFilters` property and set the execution order of installed and custom filters as desired.

For example:

```
ruleFilters=\
      RequestParameterRuleFilter,\
      MyCustomRuleFilter,\
      URLPatternMatchingRuleFilter
```

Site Accessibility

After the SiteContextPipeline identifies a site, it checks the site configuration to determine whether the site is *enabled* and *active*:

- Enabled: The site's enabled property is set to `true`.
- Active: The site is enabled and the current date falls between the site's open and close dates, as configured in the site's `openDate` and `closingDate` properties, respectively. If these properties are not set, the site is regarded as always active.

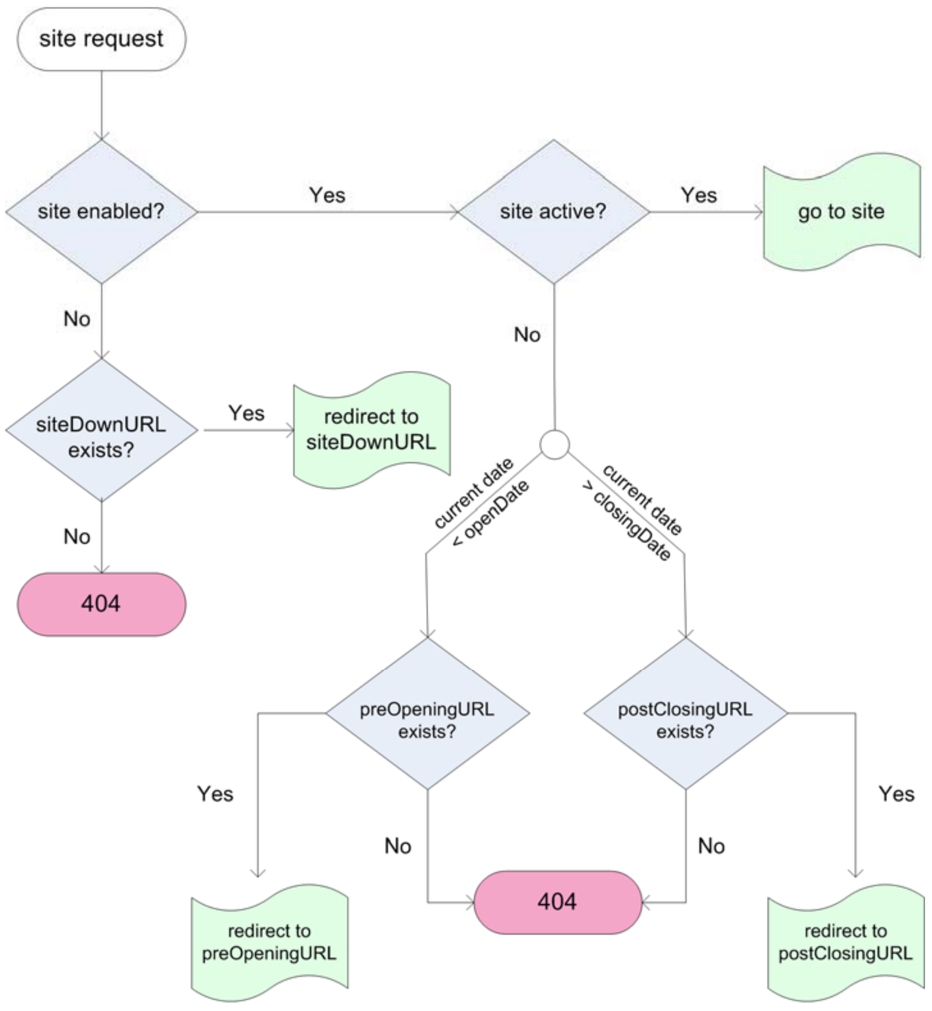
Note: Although an enabled site can be inactive, a disabled sites is always inactive, regardless of its `openDate` and `closingDate` settings.

In the event that a site is disabled or inactive, other site properties can specify where to redirect requests. The following site properties determine whether a site is enabled or active, and how to redirect requests for disabled or inactive sites:

Property	Description
`enabled`	A Boolean property, specifies whether the site is enabled. Default: `false`
`siteDownURL`	Specifies where to redirect requests when the site's `enabled` property is set to `false`.
`openDate`	A Date property, specifies when the site starts accepting requests.
`closingDate`	A Date property, specifies when the site starts refusing requests.
`preOpeningURL`	Specifies where to redirect requests before its `openDate` setting.
`postClosingURL`	Specifies where to redirect requests after its `closingDate` setting.

Redirecting Requests

The SiteContextPipelineServlet directs a request to the appropriate site as shown in the following flow chart:



Redirect Constraints

The SiteContextPipelineServlet prevents an infinite loop of redirects by allowing only one redirect per request. On the first redirect, the pipeline servlet appends a parameter to the redirect URL. A request with this parameter cannot be redirected again; instead, it returns with a 404 error.

For example, the site repository might provide the following redirect settings for sites `Jasper` and `Felix`:

- Jasper sets its `siteDownURL` property to the URL of site Felix.
- Felix sets its `preOpeningURL` and `postClosingURL` properties to the URLs of sites Casper and Astro, respectively.

Jasper is disabled, so the SiteContextPipelineServlet redirects requests to Felix. Felix is inactive, but instead of forwarding the request to sites Casper or Astro, the SiteContextPipelineServlet aborts the request and returns a 404 error.

Site Accessibility Processing

Processing of site accessibility is divided between two components:

- The component `/atg/multisite/SiteManager` determines whether a site is active.
- The request pipeline's SiteContextPipelineServlet redirects requests for disabled and inactive sites.

Active Site Evaluation

The SiteManager determines whether a site is active through its SiteStateProcessor component, which is referenced by the SiteManager's `siteStateProcessor` property—by default, `/atg/multisite/DefaultSiteStateProcessor`. This component implements interface `atg.multisite.SiteStateProcessor`. The ATG installation provides one implementation class, `atg.multisite.DefaultSiteStateProcessorImpl`. This class defines the method `isSiteActive()`, which evaluates the RepositoryItem of a given site and returns `true` (active) or `false` (inactive).

`DefaultSiteStateProcessorImpl` has several properties that let individual servers determine whether a site is enabled and active, regardless of its site repository settings:

Property	Description
`ignoreEnabled`	If set to `true`, nullifies the site's `enabled` property setting and enables the site. Default: `false`
`ignoreOpenDate`	If set to `true`, nullifies the site's `openDate` property setting, and leaves the site's opening date unspecified. Default: `false`
`ignoreClosingDate`	If set to `true`, nullifies the site's `closingDate` property setting and leaves the site's closing date unspecified. Default: `false`

In its default implementation, `isSiteActive()` reads these properties when it determines whether a site is enabled and active. By default, these Boolean properties are set to false; in this case, `isSiteActive()` reports to the SiteManager on a site's availability solely on the basis of its repository settings. For example, if a site's `closingDate` precedes the current date, the SiteManager regards the site as inactive and denies access to it. However, if the siteStateProcessor's `ignoreClosingDate` property is set to `true`, the SiteManager ignores the `closingDate` setting and regards the site as active.

Customizing Active Site Evaluation

You can modify and extend the logic of active site evaluation as follows:

1. Create a class that implements the `atg.multisite.SiteStateProcessor` interface.
2. Create a Nucleus component from the new class.

3. Set the SiteManager's `siteStateProcessor` property to the new Nucleus component.

Customizing Redirection

The SiteContextPipelineServlet redirects requests for disabled and inactive sites through its `DefaultInactiveSiteHandler` property. This property is set to a component that is based, by default, on the class `atg.multisite.InactiveSiteHandlerImpl`. This component redirects requests to the appropriate URL. You can customize redirect handling as follows:

1. Create a class that extends `atg.multisite.InactiveSiteHandlerImpl`. The subclass should override `InactiveSiteHandlerImpl.determineRedirectURL()`. This method should return a String that contains the redirect URL, or null in order to return the default 404 error code.
2. Create a Nucleus component from the new subclass.
3. Set the SiteManager's `siteStateProcessor` property to the new Nucleus component.

Note: If you use your own implementation of the SiteStateProcessor interface, you should also customize the DefaultInactiveSiteHandler to reflect those changes. The SiteStateProcessor only determines whether a site is active; it supplies no information why the site is inactive.

Preview Request Handling

A preview server ignores a site's enabled/active state. The preview configuration layer sets the SiteContextPipelineServlet's `isOnPreviewServer` property to `true`, which prevents all redirection.

Site Context Management

After receiving a site ID from the SiteContextPipelineServlet, the SiteContextManager (`/atg/multisite/SiteContextManager`) uses it to create a SiteContext component for the current request by calling `getSiteContext()`. The SiteContext component gives the current request thread access to site properties, and provides a mechanism for storing and retrieving transient attributes related to that site.

After the SiteContext component is created, the SiteContextPipelineServlet then calls the following methods on the SiteContextManager:

- `clearSiteContextStack()` ensures that the SiteContext stack is cleared out for this request, in case another request used this thread.
- `pushSiteContext()` pushes the new SiteContext object onto the SiteContext stack, making it the SiteContext for the current thread.

Subsequent calls to `SiteContextManager.getCurrentSite()` return this SiteContext. Also, if a request resolves a reference to the component at `/atg/multisite/SiteContext`, it receives an implementation of SiteContext that routes all method calls to the SiteContext returned by `SiteContextManager.getCurrentSite()`.

Site Session Management

The session-scoped SiteSessionManager manages SiteSession components; together, they provide the mechanism that coordinates requests for multiple sites within a single session. Each SiteSession component maintains information about a site during the current session; the SiteSessionManager maintains a map of all SiteSession objects keyed by site IDs.

After receiving a site ID from the SiteContextPipelineServlet, the SiteSessionManager performs these tasks:

1. Checks its Map of site IDs against SiteSession components:
   - If the site ID is mapped to a SiteSession, it uses that SiteSession.
   - If the site ID is not among the Map keys, the SiteSessionManager creates a SiteSession object and adds it to its Map of SiteSession objects.
2. Calls `handleSiteRequest()`, which performs these tasks:
   - Iterates over an array of SiteRequestProcessor components, calling each component's `processSiteRequest()` method.
   - In the case of a new site session, iterates over an array of SiteSessionStartProcessor components, calling each one's `processSiteSessionStart()` method.

SiteRequestProcessor Components

On each request, the SiteSessionManager invokes its `handleSiteRequest()` method to iterate over its array of SiteRequestProcessor components and call each processor's `processSiteRequest()` method. This method typically updates SiteSessionManager or SiteSession attributes, in accordance with current request properties. It has the following signature:

```
void processSiteRequest
   (SiteContext pSiteContext, DynamoHttpServletRequest pRequest)
```

The ATG installation provides one class that implements the SiteRequestProcessor interface, `atg.multisite.LastSiteVisitedProcessor`. Its implementation of `processSiteRequest()` sets the current session's last-visited site in the SiteSessionManager's `lastVisitedSite` attribute.

SiteSessionStartProcessor Components

In the case of a new site session, `SiteSessionManager.handleSiteRequest()` iterates over the array of SiteSessionStartProcessor components, calling each one's `processSiteSessionStart()` method.

The ATG installation provides one class that implements the SiteSessionStartProcessor interface, `atg.multisite.ReferringSiteProcessor`. This processor sets the `referringSite` property on the requested site's SiteSession.

Session Expiration

When the HTTP session expires, the SiteSessionManager's `doStopService()` method is called. This method iterates over the array of SiteSessionEndProcessor components, calling each one's `processSiteSessionEnd()` method on each SiteSession.

You can also end a SiteSession directly by calling its `SiteSession.end()` method. This method runs all SiteSessionEndProcessor components on the SiteSession, and removes the SiteSession from the SiteSessionManager.

Multisite URL Management

ATG's multisite URL management system ensures that an ATG server can associate a request with the correct site. The system can handle a wide variety of URLs for production, staging, and preview servers.

On receiving a request, an ATG server relies on the globally scoped Nucleus component `/atg/multisite/SiteURLManager` to map the request URL to a site. The SiteURLManager manages this task as follows:

1. Collects from the site repository the URLs for all registered sites.
2. Optionally, generates URLs according to the URL transformation rules that are configured for that SiteURLManager.
3. Maps all URLs to site IDs and vice versa.

About SiteURLManager

The component `/atg/multisite/SiteURLManager` is an instance of the class `atg.multisite.SiteURLManager`. A SiteURLManager is registered with the component `/atg/epub/DeploymentAgent`.

Site URL Collection

On application startup and after each site configuration deployment, the SiteURLManager collects the URLs for all registered sites:

- Obtains from the site repository each site's *production site URL* and any additional URLs from repository properties `productionURL` and `additionalProductionURLs`, respectively.
- If necessary, runs transformation rules on production site URLs (see URL Transformation Rules).

The SiteURLManager then processes the list of URLs as follows:

1. Creates a map keyed by the collected URLs, that pairs URLs with site IDs. If it finds any duplicate URLs, it logs an error and removes the duplicates.
2. Creates a list of all URL keys in the map, which organizes URLs in the following order of precedence:

- Full URLs: contain domain and paths.
- Domain names only.

  All domain names have equal precedence: a URL that consists only of a parent domain is equal to a URL that includes subdomains. Thus, `foo.com` and `bar.foo.com` are regarded as equal.
- Paths only. Path-only URLs are sorted according to the number of levels in the path. Thus, `/foo/bar` has precedence over `/foo`.

For example, URLs might be ordered as follows:

```
foobar.com/foo/bar
foobar.com/foo
foobar.com
/foo/bar
/foo
```

3. Creates an empty cache for storing requested URL-site mappings.
4. Creates a map keyed by all site IDs mapped to URLs, for reverse lookups. This map is used primarily by ATG servlet beans in order to generate links from JSPs. The method `SiteURLManager.getProductionSiteBaseURL()` also uses it to get a production site URL from the supplied site ID.

   The SiteURLManager executes an array of SiteBaseURLProcessor components, specified through its `siteBaseURLProcessors` property. These control how URLs are processed. For more information, see Absolute URL Generation later in this chapter.

URL Transformation Rules

The SiteURLManager can use a transformation rules file to transform production site URLs, which is generally useful for generating staging server URLs and associating them with the correct site IDs. In a multisite environment with many servers, this can save much administrative overhead. For example, you can use transformation rules to generate staging site URLs by appending the string `-staging` to all production site URLs, in order to generate staging site URLs. Thus, requests for `wishesArePonies.com` and `wishesArePonies-staging.com` are mapped to the same site ID and use the same site configuration.

The SiteURLManager's `transformRuleFile` property points to an XML file that contains transformation rules—by default, `/atg/multisite/urlTransform.xml`. The SiteURLManager's Boolean property `enableURLTransform` determines whether it executes URL transformations—by default set to `true`.

Note: It is generally good practice to implement transformation rules only for servers that are used for testing purposes, such as staging and preview servers. Production servers should avoid using transformation rules, and rely on production site URLs as configured through Site Administration.

Constraints

The following constraints apply to URL transformation rules:

- Only one rule is allowed per server configuration. All URLs are subject to the same transformation rule.

- A multisite application uses generated URLs only for a given site. URL transformations have no effect on repository settings.
- Transformation operations apply only to portions of the domain name. They have no effect on URL path elements.

Transformation Operations

URL management supports three rule operations:

Operation	Syntax
`replace`	`<rule op="replace">` `<new-string>`*`new-string`*`</new-string>` `<original>`*`old-string`*`</original>` `</rule>`
`prepend`	`<rule op="prepend">` `<new-string>`*`new-string`*`</new-string>` `<level>`*`integer`*`</level>` `</prepend>`
`append`	`<rule op="append">` `<new-string>`*`new-string`*`</new-string>` `<level>`*`integer`*`</level>` `</append>`

Replace Operations

Replace operations can replace any portion of a domain name. For example, given the following rule:

```
<rule op="replace">
  <new-string>foobar</new-string>
  <original>example</original>
</rule>
```

`hockey.example.com` is transformed as follows:

`hockey.foobar.com`

Prepend and Append Levels

`prepend` and `append` transformation rules specify a string to add to the domain name. The `<level>` tag provides an integer value that specifies which label of the domain name to modify:

Level	Specifies...
1	Top-level domain—for example, com, edu, org
2	Parent domain
≥3	Subdomain

For example, given the following rule:

```
<rule op="append">
  <new-string>-staging</new-string>
  <level>3</level>
</append>
```

`accessories.wishesArePonies.com` is transformed as follows:

```
accessories-staging.wishesArePonies.com
```

Production Site URL Conventions

URL site management rules generally assume that site URLs are differentiated by their domain names or context paths:

- Domain names: Each site has a unique domain name.
- URL context paths: All sites share the same domain, and are differentiated by their URL context paths

Note: While it is possible to mix URL context path and domain naming conventions in a single application, it is generally advisable to choose one convention and use it for all sites.

Domain Names

You can differentiate sites through unique domain names; request URLs are mapped to site IDs accordingly. This convention encompasses two approaches:

- Sites differentiate themselves by domain names.
- Sites differentiate themselves by subdomain names that share the same parent domain.

For example, you can differentiate three sites that specialize in different sports through their domain names:

```
www.baseball.com/
www.hockey.com/
www.basketball.com/
```

You can also differentiate these sites through their subdomain names:

```
www.baseball.sports.com/
www.hockey.sports.com/
www.basketball.sports.com/
```

Note: Production site URLs and additional URLs must not include protocols such as `HTTP` or `HTTPS`. Site Administration automatically removes protocols from user-entered data; however, if you directly update the site repository's `siteConfiguration` properties `productionURL` or `additionalProductionURLs`, make sure that the URLs written to these properties excludes protocols.

Domains versus Subdomains

You can rely on subdomains to differentiate sites, where all subdomains use the same parent domain. In this case, you configure the application server to set the host name in JSESSIONID cookies to the parent domain. Sharing the same session among different parent domains requires a different approach. For more information, see Sharing a Session Across Multiple Domains.

URL Context Paths

You can differentiate sites that share the same domain through their URL context paths. For example:

```
www.mysports.com/baseball
www.mysports.com/hockey
www.mysports.com/basketball
```

To handle this case, follows these steps:

1. Configure the production site URLs of different sites with unique paths. For example: `/baseball`, `/hockey`, and so on.
2. Configure each site to specify the context root of the content-serving web application.

Virtual Context Roots

You can configure multiple sites so they access the same web application. To do so, you set their production URLs to *virtual context roots*—that is, URL context paths that do not map directly to the actual content-serving web application. Instead, URL requests that contain the virtual context root are routed to the actual context root as set in the site configuration's `contextRoot` property.

For example, you might set the context root for several sites to `/sportswhere` and set their production site URLs as follows:

```
/sportswhere/baseball
/sportswhere/hockey
/sportswhere/basketball
```

The following requirements for using virtual context roots apply:

- In the properties file for `/atg/dynamo/service/VirtualContextRootService`, set `enabled` to `true`.
- Configure application servers so URL requests that contain virtual context roots are redirected to the actual context root. For more information, see Configuring Virtual Context Root Request Handling later in this chapter.

Sharing a Session Across Multiple Domains

Different sites with unique domains must be able to share sessions. Session sharing generally requires a JSESSIONID cookie whose session identifier is shared among all sites within the same session.

If sites share a parent domain name—for example, `www.baseball.sports.com` and `www.hockey.sports.com`—the session cookie can bind to the parent domain name `sports.com`. The browser can use the same cookie for all hosts that share the parent domain name. However, sites with unique parent domain names—for example, `baseball.com` and `hockey.com`—rely on *session recovery*, where all ATG servers can access a common session ID that is maintained by a *canonical session ID host*, which is one of the ATG instances in the multisite environment.

In general, ATG supports two session recovery approaches:

- Session recovery without JavaScript
- Session recovery with JavaScript

The non-JavaScript approach is simpler. However, it requires several URL changes within the address bar that are visible to end users. It also exposes `jsessionid` in the address bar URL, which might be problematic from a security viewpoint. The JavaScript approach avoids changing the address bar URL and embedding `jsessionid` in the URL itself; however it might require extra requests, depending on the browser.

By default, session recovery with JavaScript is enabled. You configure the desired behavior through the component `/atg/multisite/CanonicalSessionHostnameServlet` by setting its Boolean property `enableJavaScript`.

General Requirements

- All servers that participate in session recovery must be configured to point to the same JVM instance or same cluster of JVM instances.
- If a cluster is used, the load balancer must make sure that a given session ID is always sent to the same JVM, regardless of host name.
- The load balancer must use the JSESSIONID cookie and the URL's `jsessionid` path parameter (`;jsessiond=`*`session-id`*) to identify sessions.

Note: Session recovery tries to set the cookie for each host once per session, unless the cookie is already present. Because session recovery duplicates some application server cookie configuration, the overlapping settings must agree. As installed, session recovery supports standard J2EE settings. If you change application server settings, be sure also to change the corresponding session recovery settings.

WebSphere Application Server Requirements

Session recovery requires additional configuration on each WebSphere application server that is part of a multisite installation, as follows:

1. In the IBM WebSphere console, navigate as follows:

   ```
   Servers
     -> Server Types
       -> WebSphere application servers
         -> server-name
           -> Web Container settings (on right)
             -> Web Container
               -> Session Management
   ```

2. Set the following checkbox fields to true:
   - Enable URL rewriting
   - Enable protocol switch rewriting
3. Navigate to Custom Properties and set the following properties to true:
   - `com.ibm.ws.webcontainer.invokefilterscompatibility`
   - `HttpSessionReuse`

Session Recovery without JavaScript

Session recovery can avoid JavaScript through simple redirects to the canonical session ID host and back. The following example and graphic illustrate this approach:

1. A browser issues a request for `http://foosite.com`, and no session currently exists for this request—either because the request provides no cookie or `jsessionid` path parameter, or because the old session expired.
2. The ATG instance redirects to the canonical session ID host `barsite.com`:

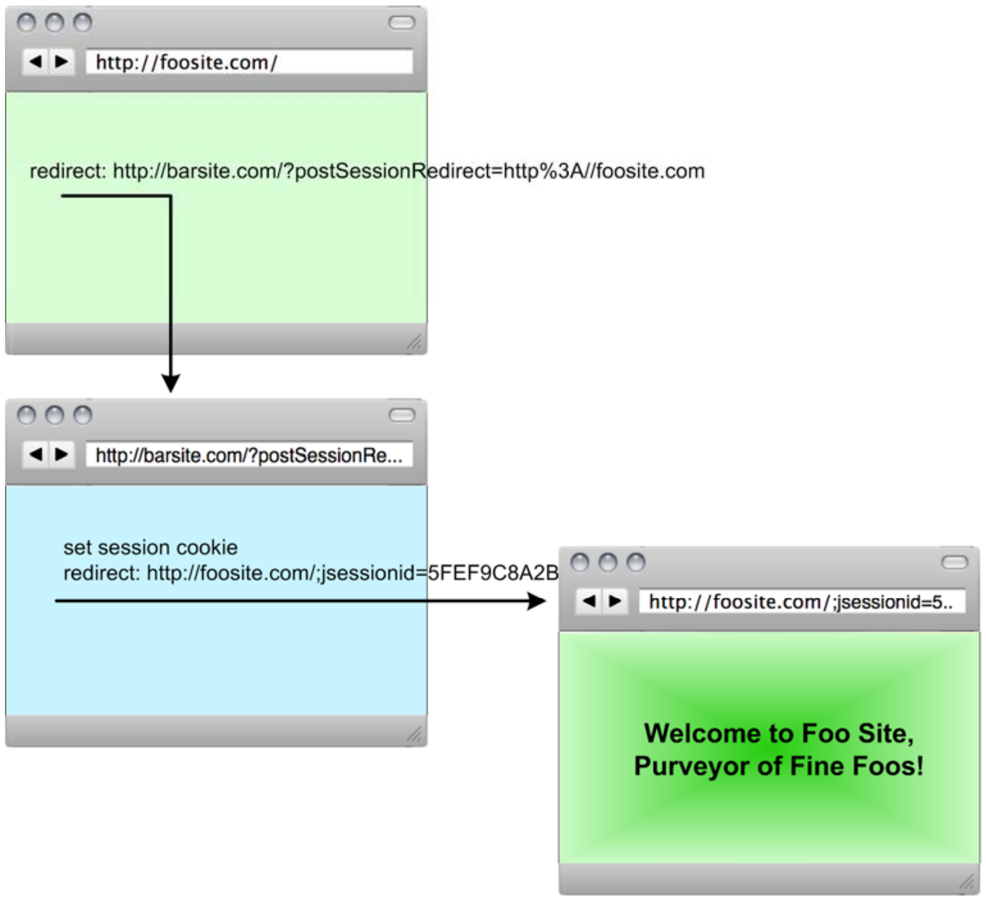
   `http://barsite.com/?postSessionRedirect=http%3A//foosite.com/`

3. The browser provides a cookie for the redirect, and the canonical session ID server redirects back to the original URL, which now contains its stored `jsessionid`:

   `http://foosite.com/;jsessionid=5FEF9C8A2B1AA8F6073DF9A7A352DF35`

   The redirect also includes a session ID cookie for `foosite.com`, if it doesn't already exist on the browser.

4. The browser gets the real content of `foosite.com`.


http://foosite.com/
redirect: http://barsite.com/?postSessionRedirect=http%3A//foosite.com
http://barsite.com/?postSessionRe...
set session cookie
redirect: http://foosite.com/;jsessionid=5FEF9C8A2B
http://foosite.com/;jsessionid=5..
Welcome to Foo Site,
Purveyor of Fine Foos!

Session Recovery with JavaScript

Session recovery can be implemented through JavaScript-enabled HTML pages. This approach encompasses two scenarios that apply to different browser capabilities:

- The browser sets the JSESSIONID cookie in an IFRAME that points to the canonical session ID server, and supplies callback data for the container page in one of two ways:
  - Invoke the `postMessage()` method.
  - Appends callback data to the container page's URL (Internet Explorer 6 and higher).
- The browser cannot set the JSESSIONID cookie in an IFRAME and lacks `postMessage()` support.

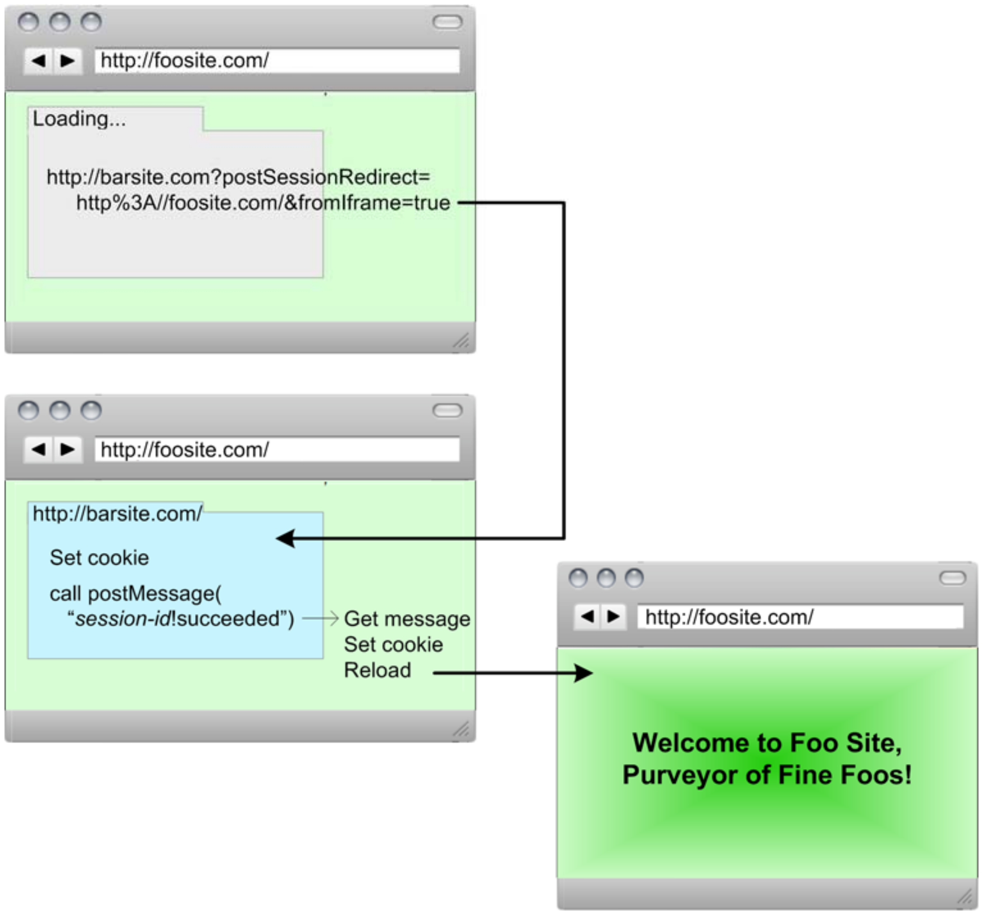
Note: These approaches apply to browsers that do not support the property `XMLHttpRequest.withCredentials`, and therefore cannot issue JavaScript background requests.

The following example and graphic illustrate the first JavaScript scenario—set the JSESSIONID cookie in an IFRAME:

1. A browser issues a request for `http://foosite.com`. No session currently exists for this request—either because the request provides no cookie or `jsessionid` path parameter, or because the old session expired.
2. The ATG instance renders a blank page with an invisible IFRAME that points to the canonical session ID server `barsite.com`.
3. The browser loads the IFRAME URL:

   ```
   http://barsite.com/?postSessionRedirect=
      http%3A/foosite.com/&fromIframe=true
   ```

4. The ATG instance renders a JavaScript-enabled page in the IFRAME as `barsite.com`. This page calls the `postMessage()` method, which invokes a callback on the container HTML page; in the case of Internet Explorer 6 and higher, the callback is specified by modifying the container page's URL with a hash anchor (#). In both cases, the callback contains the session ID of `barsite.com`, and indicates whether the session cookie for `barsite.com` was set.
5. The container page reloads `http://foosite.com`.



The following example and graphic illustrate the second JavaScript scenario: the IFRAME cannot set the session cookie and lacks `postMessage()` support:

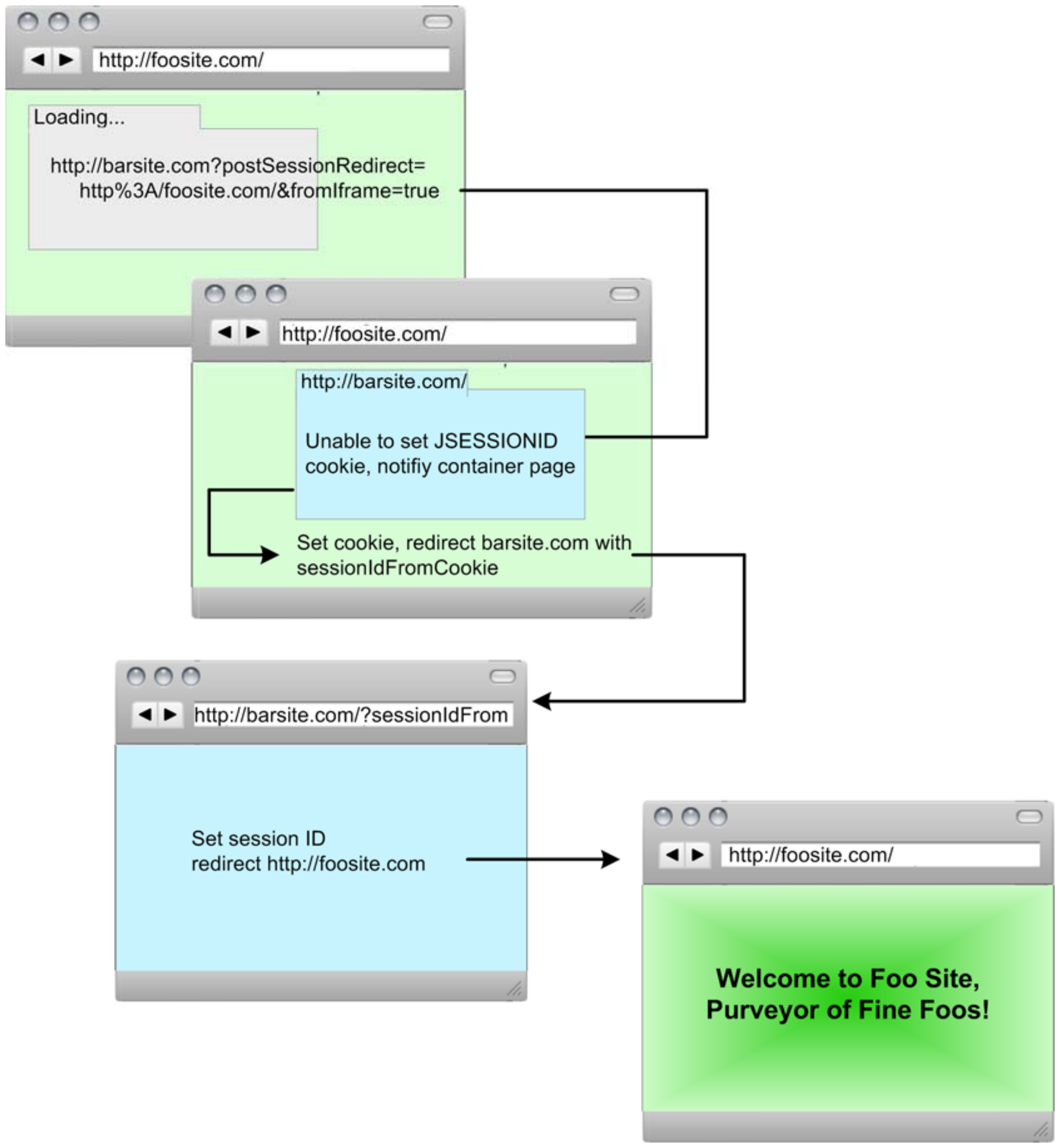
1. A browser issues a request for `http://foosite.com`. No session currently exists for this request—either because the request provides no cookie or `jsessionid` path parameter, or because the old session expired.
2. The ATG instance renders a blank page with an invisible IFRAME that points to the canonical session ID server `barsite.com`.
3. The browser loads the IFRAME URL:

   ```
   http://barsite.com/?postSessionRedirect=
      http%3A/foosite.com/&fromIframe=true
   ```

4. The IFRAME cannot set the JSESSIONID cookie. It notifies the container HTML page `foosite.com` of the failure.
5. The container HTML sets the cookie locally, then redirects to the canonical session ID server `barsite.com` with a URL that contains the JSESSIONID:

   http://barsite.com/?sessionIdFromCookie&postSessionRedirect=http%3A//foosite.co
   m

6. The canonical session ID server sets its JSESSIONID cookie and redirects back to `http://foosite.com`.


http://foosite.com/
Loading...
http://barsite.com?postSessionRedirect=http%3A/foosite.com/&fromIframe=true
http://foosite.com/
http://barsite.com/
Unable to set JSESSIONID cookie, notifiy container page
Set cookie, redirect barsite.com with sessionIdFromCookie
http://barsite.com/?sessionIdFrom
Set session ID redirect http://foosite.com
http://foosite.com/
Welcome to Foo Site, Purveyor of Fine Foos!

Handling POST Requests

Form submissions whose method is set to POST are not redirected, in order to avoid loss of form data in the POST request. Instead, POST requests are processed as usual, and the session is marked for redirection to the canonical server on the next non-POST request.

JavaScript Page Templates and Configuration

The HTML pages that are used in the JavaScript approach are rendered from page templates instead of JSPs, as JSPs require a session and must have a given location in a web application. Page templates are also easy to modify or replace, depending on your application's specific requirements.

The ATG installation provides two templates:

- `<ATG10dir>/DAS/lib/atg/multisite/jsRedirectPage.tmpl`: Renders a page that tries to obtain the session ID from the canonical session ID server.
- `<ATG10dir>/DAS/lib/atg/multisite/jsRedirectBack.tmpl`: Renders a page that acts as follows:
    - Loads from the canonical session ID server in a hidden IFRAME, and tries to set the canonical JSESSIONID. The page messages to the containing page the JSESSIONID or its failure to set one.
    - Serves as the target of a top-level redirect, if the client browser IFRAME fails to set the canonical JSESSIONID. The page determines whether the JSESSIONID cookie is set and responds appropriately.

As installed, the two page templates provide limited functionality:

- Only expressions of the form `${`*`expr`*`}` are replaced, where *`expr`* is an exact match for a predefined replacement symbol—for example `cookieName`.
- `${--` *`comment`* `--}` is replaced with nothing, where *`comment`* is a comment that is omitted from the final page.

Both page templates define a number of JavaScript variables that correspond to properties in the CanonicalSessionHostnameServlet, and are set by those properties. Each template contains comments that describe the variables it uses.

The rendering of HTML pages is managed by two servlet components that implement `atg.servlet.TemplateRendererServlet`, and are referenced by two `/atg/multisite/CanonicalSessionHostnameServlet` properties as follows:

- `/atg/multisite/JavaScriptRedirectPageTemplateRendererServlet`: Referenced by the property `redirectJavaSriptRenderer`, this servlet component renders the redirect page from the page template specified in its `templateResourcePath` property—by default, set to `atg/multisite/jsRedirectPage.tmpl`.
- `/atg/multisite/JavaScriptRedirectBackPageTemplateRendererServlet`: Referenced by the property `redirectBackJavaSriptRenderer`, this servlet component renders the redirect back page from the page template specified in its `templateResourcePath` property—by default, set to `atg/multisite/jsRedirectBack.tmpl`.

Page templates are loaded from the ClassLoader at the path specified in the TemplateRendererServlet property `templateResourcePath`. Use this property to specify your own templates or override those provided by the installation. Only complete replacement is allowed.

Each TemplateRendererServlet relies on a component that implements interface `atg.servlet.TemplateRendererServlet.DynamicSubstitutionTextProvider`. This component maps page template symbols to substitution strings and supplies this map to the appropriate TemplateRendererServlet. Two `/atg/multisite/CanonicalSessionHostnameServlet` properties reference DynamicSubstitutionTextProvider components:

- `redirectDynamicSubstitutionTextProvider`: Specifies the servlet that helps the `JavaScriptRedirectPageTemplateRendererServlet` render redirect pages.
- `redirectBackDynamicSubstitutionTextProvider`: Specifies the servlet that helps the `JavaScriptRedirectBackPageTemplateRendererServlet` render redirect back pages.

At start-up, the CanonicalSessionHostnameServlet registers the appropriate DynamicSubstitutionTextProvider for each TemplateRendererServlet servlet.

The TemplateRendererServlet interface also provides the `echoToStdout` property, which can be helpful during development and debugging.

Session Recovery Configuration

Most session recovery behavior is configured through `/atg/multisite/CanonicalSessionHostnameServlet` properties (class `atg.servlet.pipeline.CanonicalSessionHostnameServlet`).

Property	**Description**
`browserTyper`	The name of the BrowserTyper to determine whether the agent string of a given browser specifies a browser that appears to support HTML 5's postMessage but actually does not.
`canonicalHostname`	Host name of the canonical session ID server. If this property is null, the CanonicalSessionHostnameServlet is disabled. Default: `null`
`canonicalPort`	Port used by the canonical session id server. Default: 80
`canonicalRedirectProtocol`	Protocol used for redirects to the canonical session ID server. Default: `http`

Property	Description
`canonicalSubPath`	Represents any subpath needed when redirecting to the canonicalSubPath to trigger this servlet (typically triggered by an ATG web application using the PageFilter). If this property is null, the request's `requestURI` is used instead. **Note:** On WebSphere application servers, set this property to the full path of the web application's default JSP. For example: `/myapp/index.jsp`
`cookieSetFailedParameterName`	The name of the parameter that indicates failure to set the cookie on the canonical session ID server. This parameter is used when redirecting back from the JavaScript-enabled redirect page in the inner IFRAME. Default: `canonicalSessionCookieSetFailed`
`enableExternalSessionIdWorkarounds`	Boolean, specifies whether to enable workarounds for application servers that use a different external/internal session ID. These workarounds may not function properly in all cases.
`enableJavaScript`	Boolean, specifies whether to enable session recovery with JavaScript. Default: `true`
`enableSettingSessionCookie`	Boolean, specifies whether to enable setting the JSESSIONID cookie. This is done on application servers that assume that setting the cookie failed because we have an existing session, but `jsessionid` is in the URL.
`excludedUrlRegexes`	A list of URLs to exclude from session recovery—for example, URLs that implement REST web services.
`fromIframeParameterName`	The name of the parameter that is set to `true` when an IFRAME is used to fetch the session ID from the canonical host. Default: `fromIframe`
`hasDelayedRedirect`	Boolean, specifies whether to delay the redirect that sends the session ID to the canonical session ID server.
`iframeTimeout`	Number of milliseconds to wait for notification from the hidden IFRAME when using IFRAMES for session recovery. Default: `http://pt-skua:8180/dyn/admin/nucleus/atg/multisite/CanonicalSessionHostnameServlet/?propertyName=iframeTimeout5000`

Property	Description
`jsessionIdName`	The `jsessionid` attribute that is inserted into the URL. Default: `jsessionid`
`jsessionIdPlaceholderUrl`	A string that is used as a placeholder for the session ID parameter in the rendered URL when the JSESSIONID cookie is not set. The installed page template `jsRedirectPage.tmpl` uses this setting to replace the session ID parameter.
`localHostConfiguration`	Set to a `localHostConfiguration` component used to calculate a list of local host names; used if `allowLocalHost` is set to `true`. Default: `http://pt-skua:8180/dyn/admin/nucleus/atg/multisite/CanonicalSessionHostnameServlet/?propertyName=localHostConfiguration/atg/dynamo/service/LocalHostConfiguration`
`noRedirectJavaScriptParameterName`	The name of the parameter to use that specifies not to render JavaScript. This parameter is typically set in a `<noscript>` tag on the JavaScript redirect page template. Default: `noRedirectJavascript`
`noScriptURL`	The redirect URL that is used by a JavaScript page template when JavaScript is disabled. This is the original request URL with an additional parameter to indicate that the JavaScript page should not be rendered. This URL is typically referenced in the template page's `<noscript>` tag.
`noSessionRecoveryBrowserTypes`	An array of BrowserTyper types for which session recovery is not attempted.
`redirectBackDynamicSubstitutionTextProvider`	References the component `/atg/multisite/JavaScriptRedirectBackPageTemplateRendererServlet`, which renders the redirect back page for session recovery with JavaScript.
`redirectBackJavaScriptRenderer`	Sets the TemplateRendererServlet that is used to render the JavaScript-enabled redirect back page, from the page template specified in its `templateResourcePath` property. Default: `http://pt-skua:8180/dyn/admin/nucleus/atg/multisite/CanonicalSessionHostnameServlet/?propertyName=redirectBackJavaScriptRenderer/atg/multisite/JavaScriptRedirectBackPageTemplateRendererServlet`

Property	Description
redirectDynamicSubstitutionTextProvider	References the component /atg/multisite/JavaScriptRedirectPageTemplateRendererServlet, which renders the redirect page for session recovery with JavaScript.
redirectJavaScriptRenderer	Sets the TemplateRendererServlet that is used to render the JavaScript-enabled redirect page, from the page template specified in its templateResourcePath property Default: http://pt-skua:8180/dyn/admin/nucleus/atg/multisite/CanonicalSessionHostnameServlet/?propertyName=redirectJavaScriptRenderer/atg/multisite/JavaScriptRedirectPageTemplateRendererServlet
sessionCookieDomain	The domain of the cookie that is used to carry the session ID. If null, then cookies are returned only to the host that saved them. Default: null
sessionDataPath	The Nucleus path of the CanonicalSessionData component
sessionIdFromCookieParameterName	The name of the query parameter that represents an existing session ID, obtained from a cookie on the non-canonical host. Default: http://pt-skua:8180/dyn/admin/nucleus/atg/multisite/CanonicalSessionHostnameServlet/?propertyName=sessionIdFromCookieParameterNamesessionIdFromCookie
sessionNotificationParameterName	The name of the query parameter that marks a request to the canonical session ID server that a session already exists. Default: http://pt-skua:8180/dyn/admin/nucleus/atg/multisite/CanonicalSessionHostnameServlet/?propertyName=sessionNotificationParameterNamecannonicalSessionIsSessionNotification
URLPatternMatchingRuleFilter	Specifies the URLPatternMatchingRuleFilter that is used to obtain a site if allowAllSiteURLs is true. Default: http://pt-skua:8180/dyn/admin/nucleus/atg/multisite/CanonicalSessionHostnameServlet/?propertyName=URLPatternMatchingRuleFilter/atg/multisite/URLPatternMatchingRuleFilter

Property	Description
`useFoundSessionCookieParameter`	Boolean, specifies whether to use the found session cookie parameter on the redirect back. Default: `false`
`useJsessionIdOnCookieMatch`	Boolean, specifies whether to use a JSESSIONID in the URL after a cookie match; might be required by WebSphere application servers. Default: `false`

The following properties should match those used by the Web application server. These are used to set the session cookie if the application server does not do so:

Property	Description
`sessionCookieName`	Name and attributes for the RFC2109 cookie storing the session, set with the following syntax: `SessionCookieName` *`name attributes`* Default: `JSESSIONID`
`sessionCookiePath`	Default: /
`sessionCookieComment`	The comment of the session cookie.
`sessionCookieSecure`	`False`
`sessionCookieMaxAge`	-1 (never expires)

Several CanonicalSessionHostnameServlet properties restrict which URLs are allowed for session recovery. These properties address a potential security risk where the CanonicalSessionHostnameServlet responds to requests for an unrelated domain. For example, a third party might generate a request and cause redirection back to its own server with a recovered `jsessionid`.

The following table describes properties that can help minimize this security risk:

Property	Description
`allowAllSiteURLs`	Boolean, specifies whether to allow all site URLs that are configured in the site repository, and the URLs generated from them via URL transformation rules. Default: `true` **Caution:** If you set this property to `true`, be sure that all URLs in the site repository are differentiated by unique domain names—that is, each URL contains a host name. If this property is set to `true` and any site URL omits a host name, that URL can be used by unknown hosts to obtain session IDs.
`allowLocalHost`	Boolean, specifies whether to allow known host names from the local host for session recovery, including addresses such as `localhost` and `127.0.0.1`. You specify host names through the `localHostNames` property. Default: `true`
`allowedHostNames`	A list of host names that are allowed to participate in session recovery. This list can be used on its own, or can supplement other ways of allowing session recovery.
`localHostNames`	Set to a list of host names that are explicitly allowed to obtain session IDs during session recovery. In order to make these host names available, `allowLocalHost` must be set to `true`.

Several properties configure the various URL parameter names that the servlet uses during session recovery. These are available in case there is a conflict with a parameter name. Typically, only two parameter names might need to be changed on the canonical session ID server, as they serve to trigger special processing on a request:

Property	Default parameter name
`postSessionRedirectParameterName`	`postSessionRedirect`
`renderSessionIdParameterName`	`canonicalSessionRenderSessionId`

Configuring Virtual Context Root Request Handling

If you rely on URL context paths to differentiate sites, you must configure your environment to forward HTTP requests to the content-serving application that these sites share. Configuration largely depends on whether HTTP requests can be mapped directly to the context root of the shared application, or whether they must first be handled by the default web application. Two scenarios apply:

- Configure the content-serving web application to handle requests
- Configure the default web application to handle requests

Configure the Content-Serving Web Application to Handle Requests

If all production site URLs contain the context root of the content-serving web application, you can configure the application's `web.xml` to handle requests directly. For example, a multisite application might configure its context root as `/sportswhere`. It also configures its production site URLs to include `/sportswhere`:

```
/sportswhere/baseball
/sportswhere/hockey
/sportswhere/basketball
```

Given this configuration, HTTP requests for one of these sites always include the context root `/sportswhere` as in the following example:

```
http://sportswhere.com/sportswhere/baseball/uniforms
```

The non-virtual portion of this URL's path—`/sportswhere` —maps directly to the context root of the content-serving application. You enable the application to handle all URLs of this type by configuring its `web.xml` with `ForwardFilter`, `PageFilter`, and `NucleusServlet` as follows:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                       http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   version="2.5">
...
  <filter>
    <filter-name>ForwardFilter</filter-name>
    <filter-class>atg.servlet.ForwardFilter</filter-class>
  </filter>

  <filter>
    <filter-name>PageFilter</filter-name>
    <filter-class>atg.filter.dspjsp.PageFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>ForwardFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>FORWARD</dispatcher>
  </filter-mapping>

  <filter-mapping>
    <filter-name>PageFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
  </filter-mapping>
  <servlet>
    <servlet-name>Nucleus</servlet-name>
    <servlet-class>atg.nucleus.servlet.NucleusServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
```

Note: The `<dispatcher>` element requires that the application's `web.xml` use `web-app_2_5.xsd` or later.

Configure the Default Web Application to Handle Requests

If production site URLs exclude the multisite context root, you must configure the `web.xml` of the application server's default web application to handle HTTP requests. For example, a multisite application might configure its production site URLs as follows:

```
/baseball
/hockey
/basketball
```

Given a request URL of `http://sportswhere.com/baseball/uniforms`, the non-virtual portion of the URL path is / (forward slash), which must be handled by the default web application. To do so, its `web.xml` must include ATG resources `PageFilter` and `NucleusServlet` as follows:

```
<filter>
  <filter-name>PageFilter</filter-name>
  <filter-class>atg.filter.dspjsp.PageFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>PageFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<servlet>
  <servlet-name>NucleusServlet</servlet-name>
  <servlet-class>atg.nucleus.servlet.NucleusServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

The content-serving web application must also be configured with the `ForwardFilter` servlet filter:

```
...
<filter>
  <filter-name>ForwardFilter</filter-name>
  <filter-class>atg.servlet.ForwardFilter</filter-class>
</filter>
```

```
...

<filter-mapping>
  <filter-name>ForwardFilter</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

Note: The `<dispatcher>` element requires that the application's `web.xml` use `web-app_2_5.xsd` or later.

Configuration of the default web application varies among application servers. The following sections describe different requirements among the various application servers that ATG supports.

JBoss
Set the default web application's `web.xml` as shown earlier, at:

jboss-root-dir/server/*server-name*/deploy/ROOT.war/web.xml

IBM WebSphere
You configure the default web application in IBM WebSphere in the following steps:

1. In the IBM WebSphere console, navigate as follows:

   ```
   Servers
     -> Server Types
       -> WebSphere application servers
         -> server-name
           -> Web Container settings (on right)
             -> Web Container
               -> Custom Properties (on right)
   ```

2. Set this property to true:

   `com.ibm.ws.webcontainer.invokefilterscompatibility`

3. From the IBM WebSphere console, remove or disable `DefaultApplication`.

4. Recreate and deploy a default web application in your ATG application EAR. The new default web application must be in the EAR before it is deployed to the server.

   The default web application requires two files:

   /*default-app*.ear/*default-war*.war/WEB-INF/web.xml
   /*default-app*.ear/META-INF/application.xml

`web.xml` includes ATG resources `PageFilter` and `NucleusServlet`, it also includes the `<display-name>` and `<description>` tags, as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
```

```
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                       http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   version="2.5">

  <display-name>default-war-name</display-name>
  <description>description</description>

  <filter>
    <filter-name>PageFilter</filter-name>
    <filter-class>atg.filter.dspjsp.PageFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>PageFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <servlet>
    <servlet-name>Nucleus</servlet-name>
    <servlet-class>atg.nucleus.servlet.NucleusServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
</web-app>
```

The following example shows how you might set the contents of `application.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
 1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">

<application>
  <display-name>
    default-app-name
  </display-name>
  <module>
    <web>
      <web-uri>
        default.war
      </web-uri>
      <context-root>
        /
      </context-root>
    </web>
  </module>
</application>
```

Note: The `<context-root>` tag in `application.xml` must be set to / (forward slash).

Oracle WebLogic
You configure the default web application in Oracle WebLogic in the following steps:

1. From the Oracle WebLogic Server Administration console, remove or disable the default web application `mainWebApp`.
2. Recreate and deploy a default web application that replaces the previous one. This application requires two files:

   `/`*`default-app`*`.ear/`*`default-war`*`.war/WEB-INF/web.xml`
   `/`*`default-app`*`.ear/META-INF/application.xml`

   For information on configuring these files, see the previous section on IBM WebSphere.

Error Pages

If you configure error pages in the request-handling application's `web.xml`, the following requirements apply:

- The `ErrorFilter`'s `filter-mapping` element must precede the `PageFilter`'s `filter-mapping` element.
- The `PageFilter`'s `filter-mapping` element must handle `ERROR` dispatches.
- The request-handling application's `web.xml` must include all `error-page` elements.

One constraint applies when the default web application handles HTTP requests for multiple content-serving web applications. In this case, an error page that is defined in the request-handling `web.xml` must be in the same location within the various WAR files of the content-serving web applications.

For example, you might modify the request-handling `web.xml` by configuring an error page as follows (changes that pertain to error handling are in bold face):

```
...
  <filter>
    <filter-name>ErrorFilter</filter-name>
    <filter-class>atg.servlet.ErrorFilter</filter-class>
  </filter>

  <filter>
    <filter-name>ForwardFilter</filter-name>
    <filter-class>atg.servlet.ForwardFilter</filter-class>
  </filter>
...
  <filter-mapping>
    <filter-name>ErrorFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>ERROR</dispatcher>
  </filter-mapping>
...
```

```
  <filter-mapping>
    <filter-name>PageFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>ERROR</dispatcher>
  </filter-mapping>

  <error-page>
    <error-code>404</error-code>
    <location>/pageNotFound.jsp</location>
  </error-page>
...
```

Given this configuration, any 404 errors are directed to the same location within the appropriate WAR file: *`context-root`*`/pageNotFound.jsp`.

Welcome Files

In order to use welcome files with virtual context roots, set the list of valid welcome files in the component `/atg/dynamo/service/VirtualContextRootService`, through its `defaultWelcomeFiles` property. This property has the following default setting:

```
defaultWelcomeFiles=/index.jsp,/index.html,/index.htm,/home.jsp,/main.jsp
```

For example, you might configure several sites with the same context root `/sportswhere` and assign production site URLs as follows:

```
/sportswhere/baseball
/sportswhere/hockey
/sportswhere/basketball
```

You might also configure the `VirtualContextRootService.defaultWelcomeFiles` property as follows:

```
defaultWelcomeFiles=/index.jsp,/welcome.jsp
```

Given this configuration and the following request:

```
http://www.mysports.com/baseball/
```

the VirtualContextRootService launches the following search under the context root `/sports`:

1. Looks for `/index.jsp`.
2. Looks for `/welcome.jsp`.
3. If neither welcome file exists, returns a 404 error.

Absolute URL Generation

The SiteURLManager can generate URLs from site IDs, typically on requests from ATG servlet beans such as `SiteLinkDroplet` (see the *ATG Page Developer's Guide*). Given a site ID and path, you can use this servlet bean to write page code that generates links to different sites and paths. To create these links, the SiteURLManager maintains a map keyed by site IDs that are paired with URLs.

URL Construction Rules

The SiteURLManager constructs URLs for a requested site according to the following rules:

- No path supplied: New URL retains old path relative to the new site's root.
- Path contains leading forward slash (/): Path is relative to the new site's root.
- Path omits leading forward slash (/): Path is relative to the current page, on the new site.

URL construction varies depending on whether production site URLs conform to a domain-based or path-based convention (see Production Site URL Conventions). The following examples show how SiteLinkDroplet generates URLs according to these diferent conventions.

Production-Site URLs Use Path-Based Strategy

Target site's production site URL:
`/foo`

Current URL:
`http://domain.com/bar/dir/index.jsp`

Input path	New URL
`""`	`http://domain.com/foo/dir/index.jsp`
`/`	`http://domain.com/foo`
`/path/help.jsp`	`http://domain.com/foo/path/help.jsp`
`path/help.jsp`	`http://domain.com/foo/dir/path/help.jsp`

Production-Site URLs Use Domain-Based Strategy

Target site's production site URL:
`foo.com`

Current URL:
`http://domain.com/dir/index.jsp`

Input path	New URL
`""`	`http://foo.com/dir/index.jsp`
`/`	`http://foo.com`
`/path/help.jsp`	`http://foo.com/path/help.jsp`
`path/help.jsp`	`http://foo.com/dir/path/help.jsp`

Sticky Site Parameters

A sticky site is specified through special URL query parameters `pushSite` and `stickySite` (see RequestParameterRuleFilter earlier in this chapter). Links that are generated on a preview server automatically include these query parameters through the SiteURLManager property `autoAppendStickySiteParams`. The generated link retains the root of the current site; the URL path is modified according to the URL construction rules described in the previous section.

Note: A preview server should always set its SiteURLManager property `autoAppendStickySiteParams` to `true`; changing this property to `false` can yield unpredictable results.

The following examples show how the SiteURLManager generates links, given a site ID of `fbar` and this current URL:

```
http://domain.com/root/path/index.jsp
```

Input path	New URL
`""`	`http://domain.com/root/path/index.jsp? pushSite=fbar&stickySite=setSite`
`/`	`http://domain.com/root/? pushSite=fbar&stickySite=setSite`
`/path/help.jsp`	`http://domain.com/root/path/help.jsp? pushSite=fbar&stickySite=setSite`
`path/help.jsp`	`http://domain.com/root/path/path/help.jsp? pushSite=fbar&stickySite=setSite`

getProductionSiteBaseURL()

You can also obtain site URLs by calling `SiteURLManager.getProductionSiteBaseURL()`, which, given a DynamoHttpServletRequest and a site ID, returns an absolute URL based on the URL construction rules described earlier.

This method has the following signature:

```
public String getProductionSiteBaseURL(DynamoHttpServletRequest pRequest,
                                       String pSiteId,
                                       String pPath,
                                       String pQueryParams,
                                       String pProtocol,
                                       boolean pInInclude)
```

`getProductionSiteBaseURL()` takes the following optional parameters:

Parameter	Description
`pPath`	Path string to include in the returned URL.
`pQueryParams`	Query parameters to include in the returned URL.
`pProtocol`	The protocol to use—`http` or `https`—with domain names. If omitted, the protocol is set from the SiteURLManager property `defaultProtocol`.
`pInInclude`	A Boolean flag, specifies whether relative paths should use the URI of the included page.

Note: If a site ID contains unconventional characters—for example, hash (#) and plus (+)—you can enter them directly for the `pSiteID` parameter and the equivalent input parameter in ATG servlets. However, settings for the `pPath` parameter and query parameters must use URL encoding for non-alphanumeric characters—for example, `%40` and `%7E` for at (@) and tilde (~) characters, respectively.

SiteBaseURLProcessors

generate URLs from site ID " "When required to generate a URL, the SiteURLManager executes an array of SiteBaseURLProcessor components, specified through its `siteBaseURLProcessors` property. The ATG installation provides two processors:

- NullSiteBaseURLProcessor: Returns / (slash) if the production site URL is null.
- PreviewSiteBaseURLProcessor: Removes the domain portion of the base URL to generate URLs on a preview server.

Multisite Data Sharing

Data sharing is central to a multisite environment, and is enabled through site groups. Each site group contains multiple sites that share a common resource—for example, two affiliated sites that share a shopping cart. When you define a site group in Site Administration, you must specify the resources that its member sites share. The shared resources can be Nucleus components, non-Nucleus Java objects, and other resources.

ShareableType Components

A site group can share any resources that are referenced by a ShareableType. A ShareableType is a globally scoped Nucleus component that is created from the class `atg.multisite.ShareableType` or an extension. The ATG installation provides the subclass `atg.multisite.NucleusComponentShareableType`. A component of this class can reference any Nucleus components that support data sharing.

You must register all ShareableType components with the globally scoped component `/atg/multisite/SiteGroupManager`, through its `shareableTypes` property. The SiteGroupManager keeps track of all site groups and their ShareableType components.

Sharing Nucleus Components

You designate the Nucleus components that can be shared by site groups through a NucleusComponentShareableType. This component is created from the class `atg.multisite.NucleusComponentShareableType`, which extends `atg.multisite.ShareableType`.

To designate Nucleus components as available for sharing:

- Create a component that uses or extends `atg.multisite.NucleusComponentShareableType`.
- Set the NucleusComponentShareableType component's `paths` property to the Nucleus components that you wish to be available for sharing.
- Register the NucleusComponentShareableType with the SiteGroupManager through its `shareableTypes` property.

ATG Commerce provides a ShoppingCartShareableType component. By default, this component's paths property is set to two installed components:

```
paths=/atg/commerce/ShoppingCart,\
      /atg/commerce/catalog/comparison/ProductList
```

This setting enables use of a shopping cart and product comparison list as components that are shared by member sites of any given site group. When you configure your multisite environment, you register the ShoppingCartShareableType component with the SiteGroupManager by setting its `shareableTypes` property as follows:

```
shareableTypes+=/atg/multisite/ShoppingCartShareableType
```

After registering a `ShoppingCartShareableType`, you can define site groups whose member sites share a ShoppingCart component and a ProductList component. For detailed information about defining site groups, see the *ATG Multisite Administration Guide*.

Making Nucleus Components Shareable

The ATG installation provides two Nucleus components that can be shared by a site group:

- `/atg/commerce/ShoppingCart`

- `/atg/commerce/catalog/comparison/ProductList`

If desired, you can make other Nucleus components shareable. Keep in mind that the component's state might depend on other components or data that must be configured correctly for sharing; otherwise, the component is liable to exhibit unexpected behavior across different sites.

In general, the following guidelines and constraints apply:

1. If a shared Nucleus component depends on other Nucleus components for state information, these must also be shared.
2. If a shared Nucleus component depends on repository data for state information, it must be coded so that only sites sharing this component have access to the same repository data.

Note: It might be difficult to ascertain and modify all dependencies for an installed Nucleus component in order to make it shareable; doing so might require major changes to ATG code. Repository data dependencies can be especially difficult to determine.

Sharing non-Nucleus Resources

ATG provides the infrastructure for sharing non-Nucleus objects and other resources. In order to enable sharing of non-Nucleus components:

- Create a component of class `atg.multisite.ShareableType` or an extension.
- Register this component with the `/atg/multisite/SiteGroupManager`, through its `shareableTypes` property. Your application code can use the SiteGroupManager to identify existing sharing groups and all registered ShareableType components. You can use this information to set and access shared data as needed.

While the `paths` property of a NucleusComponentShareableType points to one or more Nucleus components such as a shopping cart, a ShareableType component requires you to write your own code in order to associate with it the objects that are shared within a site group. It is also possible to register a ShareableType component that has no objects associated with it; it exists solely in order to create site groups where membership is based on sharing an abstract condition or quality. The following examples outline both use cases.

Sharing Java Objects

You can use a ShareableType component in order to create site groups that share non-Nucleus components, such as Java objects or other resources. In order to do so, you must write your own code in order to associate these objects/resources with the ShareableType component, so they can be shared within a group that is configured with that ShareableType.

The following steps outline a simple implementation:

1. Create the ShareableType component `/atg/test/MyShareableType` with the following configuration:

   ```
   $class=atg.multisite.ShareableType
   id=SType
   ```

2. Register the `MyShareableType` component by setting the SiteGroupManager's `ShareableTypes` property as follows:

   ```
   shareableTypes+=/atg/test/MyShareableType
   ```

3. Use the Site Administration utility to create a site group from several existing sites, and set the site group's Shared Data field to `SType`.

4. Create a Java class for instantiating shared objects:

   ```
   public class SharedObj
   {
     String id;
     String dataProperty;
     // define getX and setX methods
     ...
   }
   ```

5. Create an handler component that provides a SharedObj instance to share among all sites within the SType site group. To implement this handler:

   - Define the custom Java class `atg.test.SharedTypeHandler` as outlined below.
   - Configure the handler component as follows:

     ```
     //component path: /atg/test/SharedTypeHandler
     $class=atg.test.SharedTypeHandler
     shareableTypeId^=/atg/test/MyShareableType.id
     ```

In this implementation, the Java class `SharedTypeHandler` is defined as follows:

```
package atg.test;
public class SharedTypeHandler
{
 ...
 String shareableTypeId;   // set from properties file, identifies ShareableType
 SharedObj globalObj=null; // share with all sites if shareableTypeId unregistered
 Hashmap<String, sharedObj> siteObjectMap; // maps each site to a SharedObj
 ...
 // get a site's SharedObj or globalObj
public SharedObj getSharedObject(){

/****************************************************************
  start pseudo code:
****************************************************************

 get current site-id from SiteContextManager
 if site-id null
   return null
 else if site-id not null
   ask SiteGroupManager whether shareableTypeID is registered
     call SiteGroupManager.getShareableTypeById( shareableTypeId )
```

```
    if getShareableTypeById() returns null
      no site groups for shareableTypeID

      if globalObj property is null
        - create SharedObj instance with unique id
        - store SharedObj in globalObj and return globalObj
      else (globalObj property not null)
        return globalObj

    else if shareableTypeId registered
      get siteGroup ID from SiteGroupManager by calling
         getSiteGroup(siteId, shareableTypeId)

      if siteGroup ID null
        site-id does not belong to any group configured with shareableTypeID
        so give site-id its own copy of SharedObj:
          - create SharedObj instance with id property set to current site-id
          - save SharedObj in siteObjecMap map with the current site-id as key
          - return SharedObj
      else if siteGroup ID not null
        if site-id in siteObjecMap
          - return the associated SharedObj
        else if site-id not in siteObjectMap
          - get all site-ids in siteGroup with this siteGroup ID
          - create SharedObj instance with id property set to siteGroup ID
          - iterate over all sites in site group, map all site-ids to same
             SharedObj instance
          - return new SharedObj instance
****************************************************************
  end pseudo code
***************************************************************/
}
```

Given this configuration, a JSP can obtain data from the `sharedObj` instance that the current site shares. For example, the following code renders the current site group ID:

```
<dsp:valueof bean="/atg/test/SharedTypeHandler.sharedObject.id" />
```

Identifying Sites in a Sharing Group

A ShareableType component is not required to be associated with any objects at all; it might be used solely to create a site group where membership is based on sharing an abstract condition or quality. For example, the ATG reference application Commerce Reference Store groups sites that represent various geographical versions of the same store. Thus, it configures the ShareableType component `RelatedRegionalStores`:

```
$class=atg.multisite.ShareableType
```

```
# The shareable type ID used by application code
id=crs.RelatedRegionalStores

# Information used to find strings appropriate for localized UIs
displayNameResource=relatedRegionsShareableTypeName
resourceBundleName=\
   atg.projects.store.multisite.InternationalStoreSiteRepositoryTemplateResources
```

This ShareableType component serves two goals:

- Enables creation of a site group whose member sites represent different geographical versions of the same store.
- Allows JSPs to use the ATG servlet bean `/atg/dynamo/droplet/multisite/SharingSitesDroplet` to identify and differentiate site group members.

The Commerce Reference Store application uses the ShareableType component `RelatedRegionalStores` to create a site group that includes two sites: ATG Store US and ATG Store Germany. The application's JSP code uses the servlet bean `SharingSitesDroplet` to determine which sites share the sharing group `RelatedRegionalStores` with the current site. When ATG Store US is the current site, the servlet bean returns other sites in the group—in this case, ATG Store Germany—and vice versa. This approach allows addition of sites to the group (via Site Administration) without requiring code changes and application reassembly.

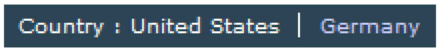
The JSP code renders a widget for each site group member: the current site is represented by an identifying label, while other sites are represented by hyperlinks:

```
<dsp:page>
  <dsp:importbean bean="/atg/multisite/Site"/>
  <dsp:importbean bean="/atg/dynamo/droplet/ComponentExists"/>
  <dsp:importbean bean="/atg/dynamo/droplet/ForEach" />
  <dsp:importbean bean="/atg/dynamo/droplet/multisite/SharingSitesDroplet" />

  <%-- Verify that this is an international storefront. If so, the Country portion
       of the site picker should be rendered. --%>
  <dsp:droplet name="ComponentExists">
    <dsp:param name="path" value="/atg/modules/InternationalStore" />
    <dsp:oparam name="true">

      <%-- Retrieve the sites that are in the Related Regional Stores sharing
           group with the current site. --%>
      <dsp:droplet name="SharingSitesDroplet">
        <dsp:param name="shareableTypeId" value="crs.RelatedRegionalStores"/>
        <dsp:oparam name="output">

          <dsp:getvalueof var="sites" param="sites"/>
          <dsp:getvalueof var="size" value="${fn:length(sites)}" />
```

```
<c:if test="${size > 1}">

  <%-- Get the site ID for the current site. The current site should not
       be rendered as a link in the site picker. --%>
  <dsp:getvalueof var="currentSiteId" bean="Site.id"/>

  <div id="atg_store_regions">
    <h2>
      <fmt:message key="navigation_internationalStores.RegionsTitle" />
      <fmt:message key="common.labelSeparator"/>
    </h2>

    <ul>
      <dsp:droplet name="ForEach">
        <dsp:param name="array" param="sites"/>
        <dsp:setvalue param="site" paramvalue="element"/>
        <dsp:param name="sortProperties" value="-countryDisplayName"/>

        <dsp:oparam name="output">
          <dsp:getvalueof var="size" param="size"/>
          <dsp:getvalueof var="count" param="count"/>
          <dsp:getvalueof var="countryName"
               param="site.countryDisplayName"/>
          <dsp:getvalueof var="siteId" param="site.id"/>

          <li class="<crs:listClass count="${count}" size="${size}"
              selected="${siteId == currentSiteId}" />">
            <c:choose>

              <%-- For the current site, render its name only. --%>
              <c:when test="${siteId == currentSiteId}">
                <dsp:valueof value="${countryName}" />
              </c:when>

              <%-- For other sites, render a cross-site link. --%>
              <c:otherwise>
                <%-- this page uses SiteLinkDroplet to create a link
                from the supplied site ID and customURL value --%>
                <dsp:include page=
                           "/global/gadgets/crossSiteLinkGenerator.jsp">
                  <dsp:param name="siteId" value="${siteId}"/>
                  <dsp:param name="customUrl" value="/"/>
                </dsp:include>
                <dsp:a href="${siteLinkUrl}"
                     title="${countryName}">${countryName}</dsp:a>
              </c:otherwise>

            </c:choose>
          </li>
        </dsp:oparam>
```

```
                </dsp:droplet>
              </ul>
            </div>
          </c:if>
        </dsp:oparam>
      </dsp:droplet>
    </dsp:oparam>
  </dsp:droplet>
</dsp:page>
```

When on the ATG Store US site, the code renders the following output, where the display name for the current site, United States, is rendered as a label, while a hyperlink is generated for the ATG Store Germany site:



Shared Component Proxying

In a multisite environment, site groups and individual non-grouped sites might each require access to the same shared component. For example, the site group `US_StoresGroup` might contain two member sites: `RetailSite` and `OutletSite`, where both sites share the same ShoppingCart component. Two non-grouped sites—`EuroSite` and `ChinaSite`—each have their own ShoppingCart component. If a user in the same session accesses sites `US_StoresGroup.RetailSite` and `EuroSite`, two separate ShoppingCart instances must be created and managed, and each client request on a ShoppingCart must be relayed to the appropriate instance.

To handle this, Nucleus relies on a `cglib2`-based proxying mechanism that creates and maintains unique ShoppingCart instances for each site group and ungrouped site, and maintains a ProxyCallback map for relaying each ShoppingCart request to the appropriate instance.

In general, Nucleus processes all component requests in a multisite environment as follows:

1. Checks whether the component can be shared—that is, the component is referenced by a registered ShareableType.
2. For a shared component, invokes one of the installed proxy factories to create:
   - A `cglib2`-based proxy that intercepts all calls to this component.
   - An `atg.service.proxy.multitarget.ProxyCallback` object, which maps method invocations to the appropriate context-specific proxy target component.

The ProxyCallback creates, as needed, a proxy target component for each site group that uses the shared component; it also creates a proxy target component for each site that does not belong to any group.

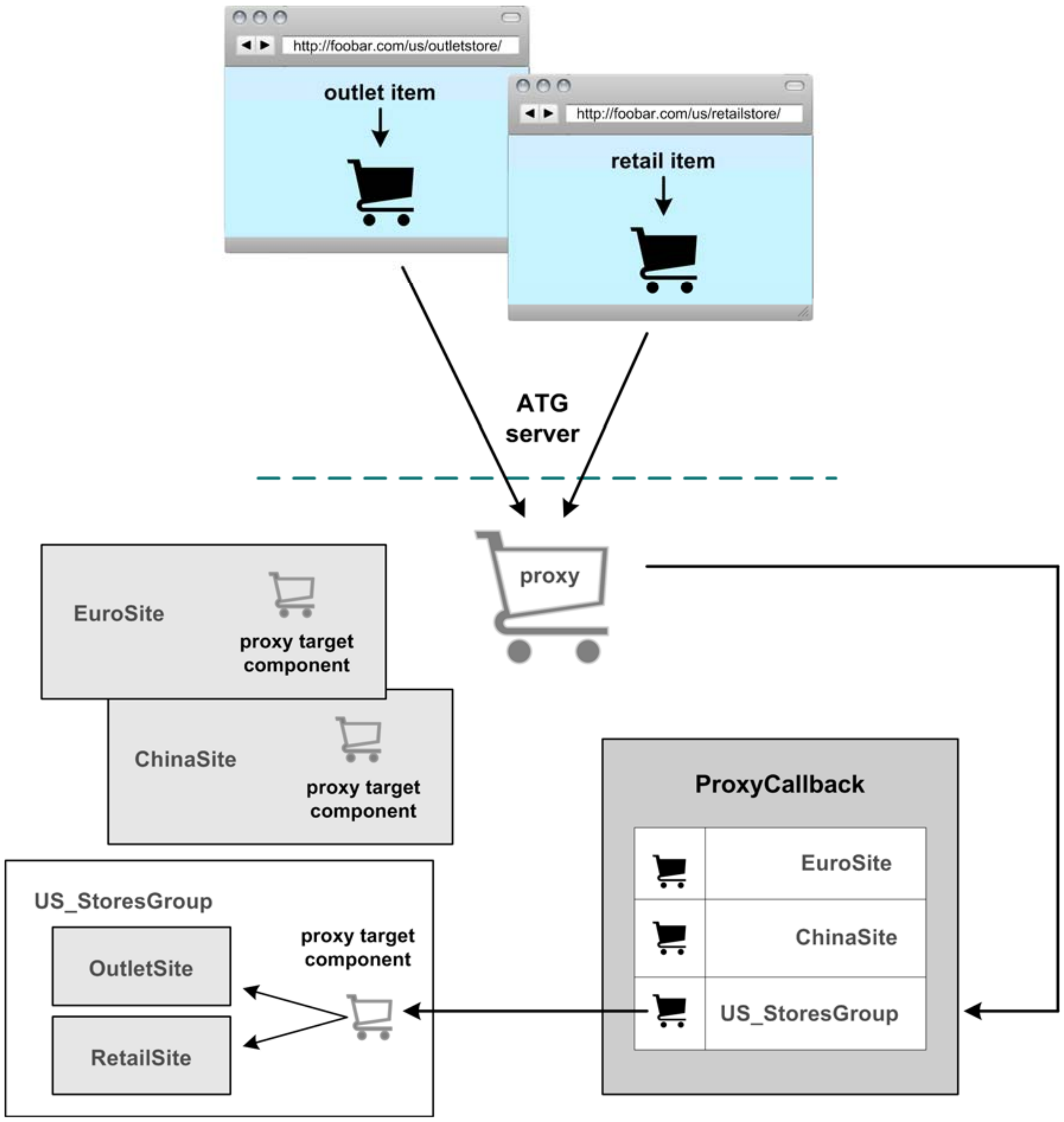
Viewing Shared Components

A shared component's proxy is visible to debugging and administrative tools such as ATG Dynamo Server Admin. The ATG Dynamo Server Admin also lets you view proxy target components for a given site group through its Component Browser Context page.

Restricting Access to Proxy Target Components

A proxy target component such as a ShoppingCart cannot return a direct reference to itself to the calling code. This provides basic protection from invocations that inadvertently circumvent the ShoppingCart proxy, which should intercept and route all calls. However, by following references that eventually lead back to the proxied ShoppingCart, it is possible to circumvent this protection. For example, care should be taken to prevent the proxy target component from returning references that can be used, in turn, to obtain a reference to the parent object.

The following graphic shows how proxying might handle invocations on a shared ShoppingCart component:



Proxy Factories

Nucleus uses two extensions of the `MultiTargetComponentProxyFactory`:

- `atg.multisite.session.MultisiteSessionComponentProxyFactory`: Creates session-scoped proxies for multisite components.
- `atg.userprofiling.preview.PreviewComponentProxyFactory`: Creates proxies that handle the special requirements of components that are accessed in a preview session. For more information about configuring asset preview, see the *ATG Business Control Center Administration and Development Guide*.

10 Core ATG Services

ATG comes with a host of classes that can be used as Nucleus components. Because these classes provide functionality that is generally useful across many applications, these classes are often called services. This chapter outlines the different services that are available in ATG.

Like all Nucleus components, these classes are created and configured through configuration files (properties files, most often) in the Nucleus configuration tree. Some services are meant to be used by multiple applications, and are thus instantiated once in a well-known place. For example, the `Scheduler` service is used by many applications, and is instantiated as part of the standard ATG server configuration. Other services are instantiated and configured differently for different applications, sometimes creating many instances for the same application. And finally, other services are not instantiated at all, but instead are extended by subclasses, which should then be instantiated.

In this chapter

This chapter describes the following ATG services:

TCP Request Server

ATG includes a standard HTTP request-handling mechanism for implementing HTML-based interfaces to applications. Your applications might want to create additional TCP servers to handle other types of requests from outside sources. These servers all run within the same process.

The class `atg.server.tcp.RequestServer` provides a mechanism for building TCP servers. When the `RequestServer` starts, it creates a server socket that waits on a specific port. It also creates a number of `atg.server.tcp.RequestServerHandler` objects that run in their own separate threads. The `RequestServer` then waits for incoming connections, and each incoming connection is passed to a free `RequestServerHandler` to be handled. If no `RequestServerHandler` objects are free, the `RequestServer` waits until one is free. When a `RequestServerHandler` is done handling a request, the handler is returned to the pool of handlers waiting for requests until it is asked to handle another request. A `RequestServer` reuses `RequestServerHandler` objects between requests—it does not create and destroy `RequestServerHandler` objects on each request. This enables high throughput for request-based services that require high performance.

To use the `RequestServer` class, you must extend the `RequestServerHandler` and implement the `handleRequest` method to actually handle the request. You must also extend `RequestServer` and implement the `createHandler` method to create instances of your `RequestServerHandler` subclass. When you configure a `RequestServer` instance, you specify the port and number of handlers that are created.

Defining a RequestServer

The following code example shows how to create a `RequestServer` that lets you telnet into a port, type your name, and have it printed back to you. First, you must define a subclass of `RequestServerHandler`:

```
import java.io.*;
import java.net.*;
import atg.server.tcp.*;
public class MirrorHandler
extends RequestServerHandler
{
  public MirrorHandler (ThreadGroup group,
                        String threadName,
                        RequestServer server,
                        Socket socket)
  {
    super (group, threadName, server, socket);
  }

  protected void handleRequest (Socket socket)
  {
    try {
      BufferedReader br = new BufferedReader(new
         InputStreamReader(socket.getInputStream()));
      OutputStream out = socket.getOutputStream ();
      PrintStream pout = new PrintStream (out);

      String name = br.readLine ();
      pout.println ("Hello " + name + "!");
    } catch (IOException exc) {}
    finally {
      try { if (socket != null) socket.close (); }
      catch (IOException exc) {}
  }
}
```

Ignore the constructor, as your constructor probably always looks like that. Look at the `handleRequest` method, which contains the code that reads a line and sends it back. Note the use of `try...finally`, which ensures that no matter what happens inside the handler, the socket is always closed. This is important because sockets are a limited resource, and leaving them open accidentally because of unanticipated errors can cause you eventually to run out of sockets.

Remember that these objects are reused from request to request. If you find that you are doing a lot of setup in each request, that setup might be hurting your throughput. See if you can move some of the setup to the constructor by creating objects that can be reused from request to request.

Also be aware that your handler need not be thread-safe. You can be assured that only one request is running through your request handler at any one time. The `RequestServer` spins off multiple handler objects to handle multiple simultaneous requests. This should help you make your handler more efficient for reuse.

In addition to extending `RequestServerHandler`, you must also extend `RequestServer` to tell it what subclass of `RequestServerHandler` to create:

```
import java.net.*;
import atg.server.tcp.*;
public class Mirror
extends RequestServer
{
  public Mirror ()
  {
    super ();
  }

  protected RequestServerHandler createHandler (ThreadGroup group,
                                               String threadName,
                                               Socket socket)
  {
    return new MirrorHandler (group, threadName, this, socket);
  }
}
```

That should be all you need to do for `RequestServer`.

Configuring a RequestServer

A `RequestServer` typically needs to be configured with a port and a `handlerCount` property. For example:

```
port=8832
handlerCount=20
```

This declares that the server runs on port 8832, and spins off 20 handler objects, meaning that it can handle up to 20 simultaneous requests.

You can also set the `handlerCount` to 0, which represents a special case. In that case, the server creates no handler objects when it starts. Instead, each incoming connection results in a new handler object (and corresponding thread). The handlers are used once and destroyed after each connection.

In order to start your server, you probably need to include a pointer to your server in the `initialServices` property of some `InitialService` object. See Starting a Nucleus Component in the Nucleus: Organizing JavaBean Components chapter to review how to do this.

A `RequestServer` has a `connectionAcceptor` property. This property specifies a Nucleus component (`atg.server.tcp.SimpleConnectionAcceptor`) that regulates how client connections to the server are accepted. Each `RequestServer` should use a separate connection acceptor component. The default connection acceptor component allows the requesting thread to wait on the server socket, adjusting its priority as configured by the `priorityDelta` property of the connection acceptor.

RequestServer Statistics

The `RequestServer` (and its subclasses) expose a number of runtime statistics as read-only properties. You can examine these statistics through the Component Browser, or by calling the appropriate `get` methods:

runningHandlerCount

The number of handlers running, including both idle handlers and handlers currently handling requests. This should be the same as `handlerCount`, unless `handlerCount` is set to 0.

activeHandlerCount

The number of handlers currently handling requests.

handledRequestCount

The total number of requests that all handlers completed.

totalRequestHandlingTime

The total amount of time taken to complete all requests handled by all handlers. This adds time spent in parallel operations, so if 10 handlers running at the same time take 100msec each, the total handling time is 1000msec.

RMI Services

An ATG server includes a service that can expose certain components to other applications through Java remote method invocation (RMI). If you write a service according to the RMI specifications, you can register your service with the ATG RMI server, and other applications can access it.

Writing an RMI Service

Before using the RMI server, you must write your service according to the RMI specifications. The following example shows how to do that.

First, use an interface to encapsulate the functionality you want to expose through RMI. For example, say that you want to make a bank account component that allows someone to adjust the balance. You might design the `BankBalance` interface to look like this:

```
import java.rmi.*;
public interface BankBalance extends Remote
{
    public void adjustBalance (double amount) throws RemoteException;
    public double getCurrentBalance () throws RemoteException;
}
```

Remember that you do not have to put your service's complete functionality in this interface—just the parts that you want to make accessible remotely. And remember that the interface must extend `java.rmi.Remote`, and every method must declare that it throws `java.rmi.RemoteException`.

After you finish writing the remote interface, write the actual implementation of that interface. For example:

```
import java.rmi.*;
public class BankBalanceImpl
extends atg.nucleus.GenericRMIService
implements BankBalance
{
  double balance;
  public BankBalanceImpl () throws RemoteException {}

  public void adjustBalance (double amount) throws RemoteException
  { balance += amount; }
  public double getCurrentBalance () throws RemoteException
  { return balance; }
}
```

This implementation can have any methods you wish, as long as it implements your remote interface. It can even implement multiple remote interfaces. However, it must include the functionality of `java.rmi.UnicastRemoteObject` and also implement `atg.naming.NameContextElement`. ATG provides a convenient base class that does both, called `atg.nucleus.GenericRMIService`. This class extends `GenericService` and adds the RMI capabilities provided by `UnicastRemoteObject`.

Now compile the `BankBalance` and `BankBalanceImpl` classes using any Java compiler. For example:

```
javac BankBalance.java BankBalanceImpl.java
```

In order for Java to use these classes remotely through RMI, it must have stub and skeleton classes corresponding to your implementation. The JSDK comes with a command line utility called `rmic`, which automatically generates the stub and skeleton classes. When you run `rmic`, you should use as an argument the full class name of the implementation class (not the remote interface class). For example:

```
rmic BankBalanceImpl
```

You should see two new class files appear: `BankBalanceImpl_Skel.class`, and `BankBalanceImpl_Stub.class`.

Your classes are now ready for use with the `RmiServer`. But first you must define an instance of your `BankBalance` object in Nucleus. For example, the following might go into a `BankBalance.properties` file:

```
$class=BankBalanceImpl
```

Exporting an RMI Service

After you create your RMI service, you can use the `atg.server.rmi.RmiServer` class to make that service available to remote clients through the RMI interface.

To export a service, add its Nucleus name to the `exportedServices` property of the `RmiServer`. ATG comes with an `RmiServer` instance already configured at `/atg/dynamo/server/RmiServer`. You might export your `BankBalance` component by adding this property setting to your `RmiServer` component:

```
exportedServices+=/yourcomponents/BankBalance
```

You can export as many services as you wish, separating their names with commas. The names must be full Nucleus names—that is, they must start with a forward slash (/). The next time Nucleus starts after making these changes, your services are available for use through RMI.

Making an RMI Client

After you export your RMI service, you can test it by creating an RMI client. Accessing a remote object from a Java client requires a single RMI call, and the URL to the remote object. The URL for a remote object in an ATG server is formed like this:

```
rmi://{dynamo host}:{rmi port}{object's Nucleus name}
```

The standard RMI port for an ATG server is 8860, so a typical URL might look like this:

```
rmi://myhost:8860/yourcomponents/BankBalance
```

The following program demonstrates you can access an object using this URL:

```
import java.rmi.*;
import java.io.*;
public class BankBalanceClient {
  public static void main (String [] args)
       throws RemoteException, NotBoundException, IOException {
    BankBalance bb = (BankBalance)
      Naming.lookup ("rmi://myhost:8860/yourcomponents/BankBalance");
    System.out.println ("current balance = " + bb.getCurrentBalance ());
    System.out.println ("adding $8.23");
    bb.adjustBalance (8.23);
  }
```

After starting Nucleus, you can run this program a few times (with the URL changed to match your particular configuration) to prove to yourself that you are accessing the Nucleus object remotely.

RMI Socket Factories

RMI is designed to be extensible at runtime, and in particular it supports the notion of socket factories, which are pluggable objects with a responsibility for manufacturing network socket connections. A default socket factory is installed when RMI starts up, which simply uses regular unencrypted TCP connections to communicate RMI method calls and their results. However, other socket factories can be optionally installed. Socket factories control how RMI endpoints communicate at the raw byte-stream level. They have no effect on the higher-level operation of RMI, such as method calls and thread usage.

You can designate a Java class that is instantiated and used as RMI's socket factory. This permits transparent use of third-party vendor socket factory implementations by RMI, and thus by client/server RMI communication. This supports such features as:

- Secure sockets (SSL)
- Tunneling—for example, RMI via firewall proxies

RMI-over-SSL can be useful in cases where a secure server needs to be contacted by authorized entities outside the firewall, or where security behind the firewall is an issue.

Configuring an Alternate Socket Factory

The component `/atg/dynamo/server/RmiInitialization` (of class `atg.server.rmi.RmiInitialization`) is responsible for performing RMI configuration prior to starting the ATG RMI server. It provides two ways to specify the socket factory class to be used:

- You can use the `RMISocketFactory` property to specify a Nucleus component that is an instance of a socket factory class.
- You can use the `RMISocketFactoryClass` property to specify a socket factory class directly.

In general, specifying a Nucleus component is preferable, because you can easily configure the component through a properties file. Specifying a class directly is a useful alternative, if, for example, you are using a preconfigured RMI socket factory implementation obtained from a third-party vendor.

To configure an alternate socket factory:

1. Install the third party vendor package that contains the factory and place the vendor code into the CLASSPATH.
2. On the server, set the `RMISocketFactoryClass` property of the `/atg/dynamo/server/RmiInitialization` component to the name of the socket factory class to be used. Or create a properties file for a Nucleus component that is an instance of the socket factory class, and set the `RMISocketFactory` property to the Nucleus address of this component.
3. Edit the client's startup script to add the following to the JAVA_ARGS line:

   ```
   -Drmi.socket.factory.class=classname
   ```

   where *`classname`* is the name of the socket factory class.

To debug a socket factory at runtime, use vendor-supplied debugging switches if available. You can also use the Java argument `-Djava.rmi.server.logCalls=true` to force all RMI method calls to be logged to the console.

RMI Over SSL

You can secure RMI communications by transmitting them over SSL. The ATG platform includes a class, `atg.net.ssl.SSLRMISocketFactory`, for creating secure sockets for RMI, and a Nucleus component that is an instance of this class, `/atg/dynamo/service/socket/SSLRMISocketFactory`. To enable RMI over SSL, set the `RMISocketFactory` property of the `/atg/dynamo/server/RmiInitialization` component to point to the `SSLRMISocketFactory` component:

```
RMISocketFactory=/atg/dynamo/service/socket/SSLRMISocketFactory
```

Configuring Keys and Certificates

To use RMI over SSL, you configure public and private keys and wrap the public key in a self-signed certificate. In a development environment, you can use the default self-signed ATG certificate in your truststore. In a production environment, however, you must create a keystore, truststore, and certificate, as described in Generating a New Certificate.

To use the default ATG certificate, use the JDK `keytool` utility to import the certificate into the truststore in your development environment:

1. Go to your `<ATG10dir>\home` directory.
2. Use the `keytool` utility to export the ATG certificate:

   ```
   keytool -export -alias atgkey –keystore ..\DAS\keystore\atg-ssl.jks
   -rfc –file ..\DAS\keystore\atg-ssl.cer
   ```

3. Enter `atgkey` when prompted for the keystore password:

   ```
   Enter password name: atgkey
   Certificate stored in file <..\das\keystore\atg-ssl.cer>
   ```

4. Now import the ATG certificate:

```
keytool -import -alias atgcert -file ..\DAS\keystore\atg-ssl.cer -
keystore
..\DAS\keystore\cacerts.jks
```

The keytool utility displays information about the certificate:

5.
```
Owner: CN=Snorthog, OU=Dynamo, O=Art Techonology Group, L=Cambridge
,
ST=MA, C=US
Issuer: CN=Snorthog, OU=Dynamo, O=Art Techonology Group, L=Cambridg
e,
ST=MA, C=US
Serial number: 3eef2fc2
Valid from: Tue Jun 17 11:12:02 EDT 2003 until: Thu May 04 14:50:08
EDT 2006
Certificate fingerprints:
         MD5:  95:0E:9A:3A:D7:C9:A6:CA:73:B5:CA:C0:44:DB:E0:1E
         SHA1: 32:38:3C:AD:57:BB:59:B7:9C:91:A3:79:03:56:9E:96:44:3
7:20:4C
```

6. Answer yes when prompted whether to trust the certificate:

```
Trust this certificate? [no]: yes
Certificate was added to keystore
```

These settings match the default configuration settings of the component `/atg/dynamo/security/BasicSSLConfiguration`, so you do not need to modify the configuration of that component.

Generating a New Certificate

In a production environment, you should not use the default ATG certificate. Instead, you should use the `keytool` utility to generate a new private key and public key, and wrap the public key into a new self-signed certificate.

1. Create a keystore and truststore for each server.
2. Use the JDK `keytool` utility with the `–genkey` flag to generate a new self-signed certificate that wraps the public key.
3. Import the certificate into the truststore of each server.
4. Configure the `/atg/dynamo/security/BasicSSLConfiguration` component on each server. You must set the `keyStore` and `trustStore` properties to point to your new keystore and truststore file locations. You must also set the `keyStorePassword` and `trustStorePassword` properties to the values that you used when creating the keystore and truststore.

For more information about SSL keys and certificates, and for documentation about the Java Secure Socket Extension (JSSE) APIs, see the Oracle Web site.

Alternative RMI Implementations

By default, ATG's RMI Server handles RMI objects. If, however, you want to import an alternative RMI implementation, you can do so:

1. Include the RMI implementation in your CLASSPATH, for both the server and UI clients.
2. Write your own code to export and resolve RMI objects using those implementations.
3. Be sure to either disable the ATG RMI Server (set the `rmiEnabled` property of `/atg/dynamo/Configuration` to `false`) or run it on an alternate port number (set the port number in the `rmiPort` property of `/atg/dynamo/Configuration`).

Port Registry

One of the more confusing aspects of server-side applications is the number of TCP servers that might start up. The standard configuration for an ATG application, for example, starts between four and six separate TCP servers. To help organize these servers, ATG includes a service called `atg.service.portregistry.PortRegistry` that keeps track of which services started servers on which ports. The `PortRegistry` has a Nucleus address of `/atg/dynamo/server/PortRegistry`. This component's page in the Component Browser lists which services are using which ports.

The `PortRegistry` does not automatically find server components. Instead, individual server components are expected to register themselves with the `PortRegistry`.

Every subclass of `RequestServer` already has the ability to automatically register with a `PortRegistry` when it starts up. To do this, the component must be configured with a property that points to the `PortRegistry`. For example, the HTTP server might be configured like this:

```
portRegistry=/atg/dynamo/server/PortRegistry
```

If you create a server component that does not extend `RequestServer`, your component should define a `portRegistry` property to be set in the configuration file. When it starts, it should register itself with the `PortRegistry` like this:

```
if (getPortRegistry () != null) {
  getPortRegistry ().addEntry (getPort (), getAbsoluteName ());
}
```

Note: This assumes that `port` is available as a property. The `getAbsoluteName` method is available only if your server component extends `atg.nucleus.GenericService`. If you extend `GenericService`, this code most likely appears inside your `doStartService` method. Also, your `doStopService` method should include the following code:

```
if (getPortRegistry () != null) {
  getPortRegistry ().removeEntry (getPort (), getAbsoluteName ());
}
```

Scheduler Services

Most server-side applications have routine tasks that must be performed periodically. For example, a component in the application might need to clear a cache every 10 minutes, or send email each morning at 2:00, or rotate a set of log files on the first of every month.

ATG includes a Scheduler service, `atg.service.scheduler.Scheduler`, which keeps track of scheduled tasks and executes the appropriate services at specified times. You can see a list of all scheduled tasks in the Component Browser at `/atg/dynamo/service/Schedule`.

ATG also includes a `SingletonSchedulableService`, `atg.service.scheduler.SingletonSchedulableService`, which enables multiple ATG servers to run the same scheduled service, while guaranteeing that only one instance of the service performs the scheduled task at a time. This provides some protection from server failures, as the loss of one ATG server does not prevent the scheduled service from running on another ATG server.

Scheduling a Task

In order to schedule a task, a component needs a pointer to the Scheduler, which is usually set as a component property. The component schedules a new task by calling `addScheduledJob` on the Scheduler. The Scheduler executes the job as scheduled.

When the Scheduler executes a job, it calls `performScheduledTask` on the object that performs the task, which must implement `atg.service.scheduler.Schedulable`. Typically, the component that schedules the task is also the `Schedulable` component that executes it, but this is not strictly required.

When a component schedules a task, it must provide the Scheduler with the following information:

- A name for the scheduled job; used only for display to the administrator.
- The name of the component scheduling the job; used only for display to the administrator.
- The `Schedulable` object that handles the job; typically, the same as the component that schedules the job.
- A flag that indicates how to run the job:
  - In a separate thread.
  - In the same thread as other scheduled services.
  - In a dedicated, reusable thread.

  If a job runs in the same thread as other services, no other scheduled services can run until the job finishes. If the job is long and expensive, it should run in a separate thread. If the job is short, it should run in the same thread. For more information, see ScheduledJob Thread Methods.
- The `Schedule` that indicates when the job should run. This is specified as an object that implements `atg.service.scheduler.Schedule`. The `scheduler` package provides a set of useful `Schedule` types, including schedules that represent an event at a specific time, schedules that represent periodic events, and schedules that

represent events based on the calendar—for example, on the 1st and 15th of every month. Usually the `Schedule` is passed in as a property. For more information, see Schedule Settings.

All of this information is encapsulated in a `ScheduledJob` object, which is passed to the Scheduler's `addScheduledJob()` method.

When a job is added to the Scheduler, the Scheduler returns an integer job ID, which you can later use to reference that job. For example, to stop a scheduled job, you can call `removeScheduledJob` on the Scheduler, passing in the ID of the job to stop.

When the `Schedulable` object is called to perform a task, it is passed the `ScheduledJob` object that was used to schedule that task. This is useful in the case where a single service is supposed to perform several kinds of scheduled tasks, and needs the properties of the `ScheduledJob` to determine which task it is supposed to perform.

Writing a Schedulable Component

This section describes how to write a component that schedules itself to perform a task according to different schedules. In this case, the task to be performed is to write `Hello` to the console.

Such a component might look like this:

```
import atg.nucleus.*;
import atg.service.scheduler.*;

public class HelloJob extends GenericService implements Schedulable
{
  public HelloJob () {}

  // Scheduler property
  Scheduler scheduler;
  public Scheduler getScheduler () { return scheduler; }
  public void setScheduler (Scheduler scheduler)
  { this.scheduler = scheduler; }

// Schedule property
  Schedule schedule;
  public Schedule getSchedule () { return schedule; }
  public void setSchedule (Schedule schedule)
  { this.schedule = schedule; }

  // Schedulable method
  public void performScheduledTask (Scheduler scheduler,
                                    ScheduledJob job)
  { System.out.println ("Hello"); }

  // Start method
  int jobId;
  public void doStartService () throws ServiceException
  {
    ScheduledJob job = new ScheduledJob ("hello",
                                         "Prints Hello",
                                         getAbsoluteName (),
                                         getSchedule (),
                                         this,
                                         ScheduledJob.SCHEDULER_THREAD);
    jobId = getScheduler ().addScheduledJob (job);
  }

  // Stop method
  public void doStopService () throws ServiceException
  {
    getScheduler ().removeScheduledJob (jobId);
  }
}
```

Notice that this component extends `GenericService`, which allows it to be notified of start and stop conditions through `doStartService` and `doStopService`. The component also implements `Schedulable` by defining the `performScheduledTask` method to print `Hello`. The component also requires the `scheduler` and `schedule` to be set as properties.

When the component is started, it constructs a `ScheduledJob` from the `schedule` property, and also specifies that the job should run in the Scheduler's thread. The component then adds the job to the Scheduler, and stores the job ID returned by the Scheduler. When the component is stopped, it removes the scheduled job by passing the ID of the job.

ScheduledJob Thread Methods

A `ScheduledJob` component has a `threadMethod` property that indicates what threading model should be used to execute the scheduled job. The `threadMethod` property can have one of the following values:

`SCHEDULER_THREAD`	The job is run in the Scheduler's own thread. This is the most efficient thread method, but it blocks the Scheduler from performing any other jobs until the job is complete. Therefore, use this thread method only for jobs that do not take a long time to run. Jobs that use I/O should probably avoid using this option.
`SEPARATE_THREAD`	Each time the job is triggered, a new thread is created to run the job. The thread is destroyed when the job completes. If a job is set to run periodically at a rate that is faster than the job itself can run, new threads are created to handle those jobs.

`REUSED_THREAD`	A separate thread is created to handle the job. Whenever the job is triggered, that thread is directed to run the job. When the job completes, the thread stays around waiting for the next time the job is triggered. If the job is still running when it is scheduled to begin to run again, it does not start again until the next time around. With this method, the Scheduler thread is not blocked by executing jobs, and threads do not multiply out of control if the job is triggering too quickly.

Configuring a Schedulable Component

The previous section defines a Schedulable component that schedules a task to print `Hello` to the console. To schedule this task, the component needs to be configured with two properties:

`scheduler`	Points to a Scheduler such as the standard ATG Scheduler.
`schedule`	Points to the standard ATG Scheduler. The `schedule` property can be set in a wide variety of formats, which ATG interprets through the `PropertyEditor` that is defined for the `Schedule` type. For more information on format options, see the next section, Schedule Settings.

For example:

```
scheduler=/atg/dynamo/service/Scheduler
schedule=every 10 seconds
```

Schedule Settings

The `schedule` property of a Schedulable component can be set in a variety of formats, which ATG interprets through the `PropertyEditor` that is defined for the `Schedule` type.

The different types of Schedules can also be created programmatically by creating instances of `RelativeSchedule`, `PeriodicSchedule`, or `CalendarSchedule`.

PeriodicSchedule/RelativeSchedule

You can set a schedule property to a `PeriodicSchedule` and `RelativeSchedule` alone or in combination:

- `PeriodicSchedule` specifies a task that occurs at regular intervals, in this format:

  `schedule=every` *`integer time-unit[`* `with catch up`*`]`*

- `RelativeSchedule` specifies a time relative to the current time, in this format:

  `schedule=in` *`integer time-unit`*

You set *`time-unit`* to one of the following:

```
msec
sec
seconds
min
minute
minutes
hour
hours
day
days
```

For example, the following `schedule` property is set to a RelativeSchedule that specifies to start a task in 30 seconds:

```
schedule=in 30 seconds
```

The next example shows a `schedule` property set to a PeriodicSchedule that specifies to run a task every 20 minutes:

```
schedule=every 20 minutes
```

Finally, the following example shows a `schedule` property set to a PeriodicSchedule and a RelativeSchedule that specify to wait 20 minutes before running a task, then run the task every 10 seconds thereafter:

```
schedule=every 10 seconds in 20 minutes
```

with catch up

You can qualify a PeriodicSchedule with the string `with catch up`, which determines whether the `PeriodicSchedule` tries to catch up on missed jobs. By default, the schedule executes without catch up. For example, you might schedule two seconds between jobs as follows

```
schedule=every 2 seconds
```

If a job occurs at 14:00, the next job should occur at 14:02. If the Scheduler cannot handle the job until 14:05—for example, the polling interval is greater than the period between jobs—the Scheduler schedules the next job to start at 14:06, so the jobs scheduled for 14:02 and 14:04 are missed.

By specifying `with catch up`, you can force execution of the missed jobs:

```
schedule=every 2 seconds with catch up
```

Given this property setting, at the first opportunity—in this case,14:05—the Scheduler runs two jobs to compensate for the jobs missed at 14:02 and 14:04. It runs the next job as regularly scheduled at 14:06.

CalendarSchedule

A `CalendarSchedule` schedules a task to run at designated points on the calendar. For example, you might schedule a task for 2:30am each Sunday, or a specific date such as January 1. The format for a CalendarSchedule looks like this:

`schedule=calendar` *`mos dates wkdays mo-occurs hrs mins`*

You set these parameters as follows:

Parameter	Values	Description
`mos`	`0..11`	The months when the task occurs, where 0 represents December.
`dates`	`1..31`	The days of the month when the task occurs
`wkdays`	`1..7`	The days of the week when the task occurs, where 1 represents Sunday.
`mo-occurs`	`1..4,` `last`	Occurrences of the specified *`wkdays`* in a month—for example, the first and third Wednesdays of the month.
`hrs`	`0..23`	The hours of the day when the task occurs, where 0 represents midnight.
`mins`	`0..59`	The minutes of the hour when the task occurs

You can specify multiple values for each parameter in two ways:

- Separate discrete values with commas. For example, to indicate May and August in the *`mos`* parameter:

  ```
  4,7
  ```

- Indicate a range of values with a dash (wraparound is allowed).. For example, to indicate each day between Monday and Friday inclusive in the *`wkdays`* parameter:

  ```
  2-6
  ```

You can substitute the following characters for each parameter:

- `*` (asterisk) specifies all values for that parameter
- `.` (period) specifies no values for that parameter.

Using the `CalendarSchedule`, a task is performed if all of the following conditions are true:

- The current month matches one of the month entries
- One of the following is true:
  - The current day of the month matches one of the date entries,
  - The current day of the week matches one of the *`wkdays`* entries and its occurrence in the month matches one of the occurrence in month entries
- The current hour matches one of the hour entries
- The current minute matches one of the minute entries

The following table provides some examples:

CalendarSchedule setting	Task occurrence
`calendar * 1,15 . * 14 5`	1st and 15th of every month, 2:05pm
`calendar * . 1 1,last 14 5`	1st and last Sunday of every month, 2:05pm
`calendar 1 * . * 1,13 0`	Every day in February at 1am and 1pm
`calendar 5 . 2 * * 0`	Every Monday in June, every hour on the hour
`calendar * * * * 9-17 30`	Every day, between 9am-5pm on the half hour

Backwards Compatibility in CalendarSchedules

Early versions of ATG use a 5-field `CalendarSchedule`. This 5-field format is still supported; a `CalendarSchedule` with 5 fields is interpreted as having an *`mo-occurs`* value of *.

Monitoring the Scheduler

Information about all tasks being run by the Scheduler is available through the Component Browser. If you go to the page for the `/nucleus/atg/dynamo/service/Scheduler` component, you see a list of all tasks monitored by the Scheduler.

The tasks are divided into two categories: scheduled and unscheduled. The only tasks that appear in the unscheduled category are those using `CalendarSchedules` that never actually occur, such as a `CalendarSchedule` that specifies only `Feb. 30`.

In addition, by default all scheduled jobs are instrumented with Performance Monitor `startOperation` and `endOperation` methods. These operations appear grouped together under the line Scheduled Jobs in the Performance Monitor page. If you do not want performance monitoring of scheduled jobs, you can set the Scheduler's `performanceMonitorEnabled` property to `false` to disable this behavior.

Running the Same Schedulable Service on Multiple Servers

Schedulable services are useful for a wide variety of tasks in an ATG application, including session expiration, content and component indexing, session backup, JMS message polling, log file management, and reporting. Typically, these are tasks that are performed periodically, and only affect the ATG server where they run. There are certain recurring tasks, however, that should be performed no more than once under any circumstances. There are many examples of at-most-once behavior, such as:

- Recurring or pre-scheduled orders in a commerce application, where it is critical that each scheduled order be placed once and only once.
- Periodic (batch) order fulfillment, where it is critical that each order be shipped once and only once to the customer.
- Mass or pre-scheduled email delivery, where each email message should be delivered only once.
- Periodic report generation, where each report should be generated only once.

The typical approach to implementing such tasks has been to create a scheduled service, then configure it on only one ATG server within the site. This provides at-most-once behavior, but has the disadvantage of introducing a single point of failure into the system. If the server handling order placement goes down, orders are not placed. If the server handling report generation goes down, report generation stops. How easily one can recover from this situation depends largely on the complexity of the scheduled services involved.

The `SingletonSchedulableService`, a subclass of the standard `SchedulableService`, works in conjunction with a client lock manager to guarantee that only one instance of a scheduled service is running at any given time throughout a cluster of cooperating ATG servers. This provides the foundation for an architecture where scheduled services can be configured on multiple ATG servers to provide fault tolerance, but the system can still ensure that each task is performed at most once.

`SingletonSchedulableService` is an abstract base class that implements the Schedulable interface by subclassing `SchedulableService`, and that automatically checks to see if any other instance of the service is running anywhere on the local ATG server cluster before performing its regularly scheduled task. Applications can subclass `SingletonSchedulableService` to provide concrete implementations that perform whatever application-specific work is required.

Note: Singleton behavior of the scheduled service is necessary, but not in itself sufficient, to ensure at-most-once behavior of the overall system. Consider the case of periodic order fulfillment:

Obviously, two scheduled fulfillment services should never wake up at the same time on two different ATG servers and fulfill the same orders at the same time. The result would be to ship two copies of each order to every customer, and to bill for both of them as well. On the other hand, two scheduled fulfillment services should not wake up at completely different times and fulfill the same orders. Even though the services might not overlap at all, another mechanism is necessary to keep the second service instance from shipping orders that the first instance already handled.

`SingletonSchedulableService` is designed to work in situations where the job performed by the system can be broken down into discrete work units, and where each work unit has some status indicator that tells whether or not it currently requires processing. Examples of such systems include:

- A fulfillment system where the work units might be individual orders and the status might be a flag indicating whether or not the order has shipped.
- A recurring order service where the work units might be individual orders and the status might be a timestamp indicating the next date and time at which the order should be placed.
- An email system where the work units might be mailings or individual messages, and the status might be a Boolean field in a database somewhere indicating whether or not each mailing has been successfully delivered.

The assumption behind `SingletonSchedulableService` is that on each run the service wakes up, obtains a lock to ensure that it is the only instance of the service running, fetches the work units that require processing, processes them, and then updates their status before relinquishing the lock.

The following sequence must be performed atomically from the service's point of view:

1. Fetch pending work units.

2. Process pending work units.
3. Update status of work units.

The automatic locking mechanism provided by `SingletonSchedulableService` ensures this atomicity.

Guaranteeing that only one instance of the service runs at a time throughout the ATG server cluster prevents two instances of the service that happen to wake up at the same time from trying to process the same work units. The fact that the service updates the status of the work units before releasing its lock prevents a subsequent instance of the service from trying to process those same work units again. The overall result is that each work unit should be processed once and only once, without the need to introduce a single point of failure into the system.

Configuring a SingletonSchedulableService

`SingletonSchedulableService` is an abstract class that extends `SchedulableService` and implements the `performScheduledTask` method in a way that tests for other instances of the service before proceeding.

To implement a `SingletonSchedulableService`, create a subclass of this class, and then create a component based on that subclass. Configure the component with a reference to a client lock manager, a lock name, and a timeout period. For example:

```
#Thu Aug 09 19:15:14 EDT 2001
$class=myclasses.MySingletonSchedulableService
$scope=global
clientLockManager=/atg/dynamo/service/ClientLockManager
lockName=ReportingLockManager
lockTimeOut=2000
```

The code in `performScheduledTask` contacts the client lock manager and requests a write lock using the specified lock name and timeout. If a lock is obtained successfully, the `SingletonSchedulableService` checks to see whether it is a global write lock or a local write lock.

If a global write lock is returned the service can assume that it is the only instance running anywhere on the cluster, at least if all instances of the service point to client lock managers than in turn point to the same server lock manager. In this case `SingletonSchedulableService` calls the `doScheduledTask` method to do whatever work the service does, then releases the global lock.

If a local write lock is returned it means that the client lock manager was unable to contact the server lock manager. The implementation then makes the pessimistic assumption that another instance of the service might be running, so it logs a debugging message, releases the local lock without doing any work, and goes back to sleep until the scheduler calls it again.

If the request to obtain a lock times out, the implementation assumes that another instance of the service is running and taking a long time about it, so it logs a debugging message and goes back to sleep until the scheduler calls it again.

Finally, if a `DeadlockException` is thrown, the implementation logs an error message and makes the pessimistic assumption that another instance of the service might be running, so it logs an error message and goes back to sleep.

Note: If the client lock manager's `useLockServer` property is set to false, it means that global locking is disabled for that lock manager. In this case, `SingletonSchedulableService` accepts a local lock in place of the global lock, which at least ensures that only one instance of the service runs at a time on the current ATG server.

ShutdownService

When Nucleus shuts down, it recursively shuts down all child services. The sequence in which these services shut down is undefined. You can control this sequence by configuring the Nucleus component `/atg/dynamo/service/ShutdownService`, which implements the class `atg.service.ShutdownService`. The `services` property of this component lists the order in which services are shut down. For example:

```
$class=atg.service.ShutdownService
services+=\
   /atg/reporting/datacollection/search/QueryFileLogger
   /atg/reporting/datacollection/search/UseSearchEnvironmentFileLogger
   /atg/reporting/datacollection/search/ViewContentFileLogger
   /atg/reporting/datacollection/search/EnvironmentFileLogger
   /atg/reporting/datacollection/search/TopicFileLogger
   /atg/reporting/datacollection/userprofiling/SiteVisitFileLogger
   /atg/reporting/datacollection/userprofiling/UserFileLogger
   /atg/reporting/datacollection/userprofiling/SegmentFileLogger
   /atg/reporting/datacollection/search/ProjectFileLogger
   /atg/reporting/datacollection/commerce/OrderFileLogger
```

As a message sink that implements the interface `atg.dms.patchbay.MessageSink`, ShutdownService listens for the JMS message `atg.das.Shutdown`, which is generated just before Nucleus starts to shut down. On receiving this message, ShutdownService iterates over the list of services configured in its `services` property and calls `stopService()` on each one. After this process is complete, Nucleus recursively shuts down all remaining services.

Default Configuration

The ATG installation specifies the ShutdownService as an initial service in the DAS module, in `/atg/dynamo/service/Initial`. The installation also provides three ShutdownService components in the following modules:

- DAS configures no services.
- DPS configures three services:

```
/atg/reporting/datacollection/userprofiling/SiteVisitFileLogger
/atg/reporting/datacollection/userprofiling/UserFileLogger
/atg/reporting/datacollection/userprofiling/SegmentFileLogger
```

- DCS configures a single service:

  `/atg/reporting/datacollection/commerce/OrderFileLogger`

Sampler Services

After an application has been deployed, monitoring the state of that application becomes an important task. In the JavaBeans model, the current state of the application is usually exposed through properties. These properties are often read-only, meaning that they expose only the `getX` method.

For example, the `atg.server.tcp.RequestServer` component exposes its state through properties such as `handledRequestCount` and `totalRequestHandlingTime`, which report the number of requests handled by that component, and how much time it took to handle those requests. By sampling these values periodically, you can follow the throughput and latency of the system over time.

ATG provides an `atg.service.statistics.Sampler` service that you can configure to monitor a set of component properties. The Sampler can be instructed to generate periodic samples of the specified components, or to generate samples of the component on demand. Other components, such as the `MemorySampler`, can use the basic Sampler to keep a history of the samples, or to summarize the samples into a daily email, or perhaps display the samples in a graph.

Sample Class

The `atg.service.statistics.Sample` class represents the value of a single property as observed at a specific point in time. It contains the following information:

- the name of the component and property from which the sample was taken
- the time the sample was taken
- the value of the sample, as an Object

In addition, if the value is a numeric value such as an `Integer`, the `Sample` also contains the difference in value between the current sample and the last sample taken of that property. This is presented as the following information:

- the difference in value, as a double
- the difference in time, in milliseconds
- the rate of change of the value, determined by dividing the difference in value by the difference in time, expressed as change/seconds

The `atg.service.statistics.SampleEvent` class holds an ordered list of Samples, presumably taken at the same point in time. When the `Sampler` runs, it generates `SampleEvents`, from which individual `Sample` objects can be obtained.

Sampler Class

An `atg.service.statistics.Sampler` is configured with a list of service name/property name pairs, naming the properties to be sampled. From this list, the `Sampler` can generate a `SampleEvent` containing the current values of those properties. The method `sample()` generates such a `SampleEvent`.

A `Sampler` can also have one or more `atg.service.statistics.SampleListener` objects added to it. When a `Sample` is taken, the sample can be broadcast to all `SampleListener` objects by calling `acceptSample()` on those listeners. Calling `sample(true)` both generates a new `SampleEvent` and broadcasts that event to all listeners.

A `Sampler` can also be configured with a `Scheduler` and `Schedule`. If so, the `Sampler` automatically calls `sample(true)` according to the specified `Schedule`.

In summary, the `Sampler` can be used in the following ways:

- Call `sample()` to obtain samples manually.
- Attach `SampleListeners` and call `sample(true)` to broadcast samples manually.
- Attach `SampleListeners` and specify a `Scheduler` and `Scheduler` to broadcast samples automatically.

Configuring the Sampler

The properties of a `Sampler` component determine which properties of which services are to be sampled, how often they should be sampled, and to whom the samples should be sent. You can use the following properties to configure a `Sampler`:

sampleSources

The list of services and properties to be sampled. Each element of the list is of the form `<service name>.<property name>`. For example:

```
sampleSources=\
     /atg/dynamo/server/HttpServer.handledRequestCount,\
     /atg/dynamo/server/HttpServer.averageRequestHandlingTime,\
     /atg/dynamo/server/HttpServer.activeHandlerCount
```

The order of the `Samples` in the `SampleEvent` matches the order of the properties declared in `sampleSources`.

scheduler

If you want the `Sampler` to perform sampling automatically, set this property to point to the `Scheduler` that schedules the samples:

```
scheduler=/atg/dynamo/service/Scheduler
```

schedule

If you want the `Sampler` to perform sampling automatically, set this property to specify the schedule used to run the samples:

```
schedule=every 10 seconds
```

sampleListeners

The list of the `SampleListener` objects that receive the `Samples` broadcast by the `Sampler`:

```
sampleListeners=\
        MemorySampler,\
        SampleGrapher
```

ATG comes with a `Sampler` component at `/atg/dynamo/service/Sampler`. It monitors various aspects of the ATG HTTP server, including throughput, latency, and simultaneous connections. It is configured as follows:

```
$class=atg.service.statistics.Sampler
scheduler=Scheduler
schedule=every 60 sec
sampleSources=\
        ../../../VMSystem.freeMemory,\
        ../servlet/sessiontracking/SessionManager.residentSessionCount,\
        ../servlet/sessiontracking/SessionManager.createdSessionCount,\
        ../servlet/pipeline/DynamoHandler.handledRequestCount,\
        ../servlet/pipeline/DynamoHandler.averageRequestHandlingTime,\
        ../servlet/pipeline/DynamoHandler.totalRequestHandlingTime,\
        ../server/DrpServer.activeHandlerCount,\
        ../server/HttpServer.activeHandlerCount

sampleListeners=\
        LogSampler
```

You can modify this component's configuration, or define your own `Sampler` component.

SampleListeners

The `Sampler` is responsible for generating `SampleEvents`, but does not actually do anything with those `Samples`. Functions such as logging, graphing, and watchdog notifications should be performed by `SampleListeners`.

ATG includes an example `SampleListener` called `MemorySampler`. This listener does nothing more than save a small history of the `SampleEvents` sent to it. You can view that history by viewing the `MemorySampler` component in the Component Browser.

The MemorySampler component that comes with ATG has a service name of /atg/dynamo/service/MemorySampler. Its configuration file, MemorySampler.properties, looks like this:

```
$class=atg.service.statistics.MemorySampler
sampleListSize=40
```

The sampleListSize property determines how many SampleEvents are stored before the MemorySampler starts discarding the oldest events.

Secure Random Number Generator

ATG includes a component you can use to generate secure random numbers. This component, with a Nucleus address of /atg/dynamo/service/random/SecureRandom, can generate random numbers more efficiently than the Java class, java.security.SecureRandom, as it provides the random number generator with a random seed, rather than using the slower process of Java's SeedGenerator.

You can configure the SecureRandom service using another component, /atg/dynamo/service/random/SecureRandomConfiguration. The SecureRandomConfiguration component can configure the behavior of the SecureRandom service with these properties:

Property	Description	Default Values
algorithm	The secure random algorithm to use.	SHA1PRNG
provider	The security provider supplying SecureRandom algorithms to use.	SUN
seed	ATG generates a seed value for the random number generator when your application starts up. If the seed property is null, the SecureRandom service uses that seed value. Otherwise, the seed is supplemented by the byte array specified by the seed property.	null

For more information, read the Java security architecture documentation, including the Javadoc for java.security.SecureRandom.

ID Generators

In many circumstances, an ATG application might need to generate unique identifiers. For example, each repository item in a repository needs a unique repository ID, so that the item can be retrieved by its ID. The atg.service.idgen package provides an interface and implementations that you can use to generate unique IDs in a variety of ways.

IdGenerators and IdSpaces

ATG IdGenerator services use ID spaces that can be shared by multiple IDGenerator components. Within an ID space, all IDs generated by these components are guaranteed to be unique. When an IdGenerator starts up, it initializes one or more IdSpaces for use by applications.

The examples in the Using IdGenerators section describe ways to create an IdSpace programmatically. An IdSpace can also be configured with an XML file. The location of the XML file is specified by an IdGenerator component's initialIdSpaces property. For example, the SQLIdGenerator uses:

```
initialIdSpaces=/atg/dynamo/service/idspaces.xml
```

Each IdSpace is defined in the XML file with an <id-space/> tag. The <id-space/> tag has the following attributes:

Attribute Name	Description	Default
name	A string that uniquely identifies an IdSpace within an IdGenerator. An IdGenerator can refer to an IdSpace using this name.	none
seed	The first ID in the space to reserve.	1
batch-size	How many IDs to reserve at a time.	100000
prefix	A string to prepend to the beginning of all string IDs generated from this IdSpace.	null
suffix	A string to append to the end of all string IDs generated from this IdSpace.	null

If you want to define additional IdSpaces, it is best not to modify the XML configuration file at <ATG10dir>/DAS/config/atg/dynamo/service/idspaces.xml. Instead, create your own XML file of the same name in your localconfig directory and let ATG's XML combination facility combine the files. Note, however, that the idspaces.xml file is read only when the das_id_generator database table is empty. If the das_id_generator table is already populated when you add a new IdSpace to the XML file, your changes are not picked up. As a workaround, manually add the IdSpace with the desired prefix to the das_id_generator table before trying to use that IdSpace.

Important: In general, you should not delete IdSpace rows from the das_id_generator table; doing so can cause ID collisions if an application attempts to use the deleted IdSpace. If you are certain that an IdSpace is never used again, it is safe to delete it.

Using IdGenerators

The IdGenerator interface includes methods for generating IDs that are strings or longs:

```
generateLongId(String pIdSpace)
generateStringId(String pIdSpace)
```

When you want to get a new ID, use these `IdGenerator` methods of the interface.

Normally, applications access the standard ID generator service at `/atg/dynamo/service/IdGenerator`, which starts up when your application is started. The following examples demonstrate how to use `IdGenerator` APIs in order to construct and use an ID generator. You can see these examples in context in the sample class located at:

`<ATG10dir>/DAS/src/Java/atg/service/idgen/sample/Example1.java`

First, construct an `IdGenerator` and get some IDs. You do not need to specify a name for the `IdSpace`; the default `IdSpace` is used:

```
TransientIdGenerator gen = new TransientIdGenerator();
gen.initialize();

    for (int i=0; i<3; i++)
      {
       gen.generateLongId();
      }
```

Generating an ID

The next line shows how you might generate a long ID in an `IdSpace` named `foo`. With the `IdGenerator` component's `autoCreate` property set to true, as it is by default, you do not have to create the `foo` `IdSpace`—it is created automatically:

```
gen.generateLongId("foo");
```

Given a seed of 1, this generates the ID 1.

Creating an IdSpace

In most cases, your application uses the `SQLIdGenerator` and configure `IdSpaces` for it in an XML configuration file. The following example shows how to create an `IdSpace` using the Java API:

```
IdSpace barSpace = new IdSpace("bar",  // name of id space
                               100,    // starting id (seed)
                               "Bar",  // prefix
                               null);  // suffix

gen.addIdSpace(barSpace);
```

Generating More IDs

Now, let's generate more IDs in the `bar` and `foo` `IdSpaces`:

```
gen.generateLongId("bar"); //generates ID=100
gen.generateLongId("bar"); //generates ID=101

// see how the "foo" space is independent of the bar space
gen.generateLongId("foo");  //generates ID=2
```

Generating String IDs

Now generate some `String` IDs. String IDs use the `prefix` and `suffix` properties of the `IdSpace`. These properties are not consulted when long IDs are generated. Within an `IdSpace`, the same pool of IDs is used for String and long IDs.

```
gen.generateStringId("bar"); //generates ID=Bar102
gen.generateStringId("bar"); //generates ID=Bar103
gen.generateStringId("bar"); //generates ID=Bar104
gen.generateLongId("bar");   //generates ID=105
gen.generateLongId("bar");   //generates ID=106
gen.generateStringId("bar"); //generates ID=Bar107
```

IdGeneratorException

`IdGenerator` methods throw the checked exception `IdGeneratorException`. This exception indicates an ID cannot be generated. Common causes include database trouble for the `SQLIdGenerator` and asking for an ID in a name space that does not exist when `autoCreate` is `false`. Production applications should catch this exception. The following example forces an exception for demonstration purposes:

```
gen.setAutoCreate(false);
try
  {
    gen.generateStringId("bogus");
  }
catch (IdGeneratorException ige)
  {
    System.out.println("rats, couldn't get an id");
    ige.printStackTrace();
  }
```

SQLIdGenerator

ATG includes a full-featured `IdGenerator` implementation that can be used in a distributed system. This component, located at `/atg/dynamo/service/IdGenerator` and with a class name `atg.service.idgen.SQLIdGenerator`, uses an `IdSpace` to reserve a batch of IDs. Because of the way that the `SQLIdGenerator` stores information about `IdSpaces` in a database table, `SQLIdGenerators` operating in different ATG servers can independently assign IDs while being assured that the IDs are unique across the system.

Using IdSpaces with the SQLIdGenerator

`IdSpaces` used by the `SQLIdGenerator` should define a batch-size attribute. The batch size determines how many IDs the `IdGenerator` should reserve. For example, you might have an `IdSpace` with a seed of 1 and a batch size of 10000, and two `IdGenerators`. The IDGenerators execute as follows:

1. When the first `IdGenerator` starts, it performs the following tasks:
    - Consults the `IdSpace`.
    - Starts with the seed ID and reserves all IDs that start with the seed, up to the batch size, 10000.
    - Sets the value in the ID generator database table to 10001.
2. When the second `IdGenerator` starts, it performs these tasks:
    - Consults the ID generator table and finds that the first available ID is 10001.
    - Reserves all IDs from 10001 to 20000
    - Sets the value in the ID generator database table to 20000.
3. When each `IdGenerator` exhausts its supply of reserved IDs, it returns to the ID generator table and reserves another batch of IDs. It does not need to consult with other `IdGenerators` to guarantee that its reserved IDs are unique.

Because an `IdGenerator` must access the database each time it gets another batch of IDs, it is best for performance purposes to make batch sizes comparatively large. As IDs are Java longs (64 bit ints), you will not run out of IDs for millions of years even with a batch size of 1 million and 1000 application restarts each day.

For example, the following XML tag defines an `IdSpace` named `MessageIds`, with a seed of `1` and a batch size of `10000`:

```
<id-space name="MessageIds" seed="1" batch-size="10000"/>
```

If for some reason it is essential that your IDs be generated sequentially with no gaps for unused IDs, you can create an `IdSpace` like this:

```
<id-space name="costly_id_space" seed="0" batch-size="1"/>
```

However, every time you generate an ID, it requires an extra database request. Therefore, this is not generally recommended unless you use this `IdSpace` very infrequently.

das_id_generator Database Table

The `SQLIdGenerator` uses a database table to store persistently information about what portion of the `IdSpace` has been assigned. This table must be present when the `SQLIdGenerator` starts up. By default, this table is named `das_id_generator`. If you have a different database table with this name, problems occur unless you configure the `SQLIdGenerator` to use a different database table name, using the `tableName` property:

```
tableName=my_id_generator
```

The ID generator table has a row for each IdSpace, and a column for each of the properties defined in the XML file for the IdSpace: name, seed, batch size, prefix, and suffix. You can also specify different values for these column names by setting the following properties of the SQLIdGenerator:

Property Name	Default Value
nameColumn	id_space_name
seedColumn	seed
batchSizeColumn	batch_size
prefixColumn	prefix
suffixColumn	suffix

Each time that an SQLIdGenerator accesses the IdSpace for another batch of IDs, it increments the seed value for that ID space by the number of IDs, as specified by the batch size. So, at any given moment, the seed value of the IdSpace indicates the first ID of the next batch of IDs to be reserved.

In addition to configuring the ID generator table properties, you can configure other properties of the SQLIdGenerator:

Property Name	Description	Type and Default Value
autoCreate	If true, the SQLIdGenerator can automatically create an IdSpace on each attempt to generate an ID, if it does not find one.	Boolean true
defaultIdSpaceName	If no name is specified for an IdSpace, this default IdSpace is used.	String __default__
defaultIdSpace	Defines the properties needed to construct the default IdSpace. The properties in order are: name, seed, lastSeed, batchSize, prefix, and suffix.	IdSpace(__default__,1,1 , 100000,null,null)

TransientIdGenerator

Another IdGenerator implementation is the TransientIdGenerator (class atg.service.idgen.TransientIdGenerator). This component is a sequential ID generator the IdSpaces of which are not persistent. This IdGenerator is suitable for applications that you do not want to be dependent on a database and which do not need IDs whose uniqueness is maintained across JVMs or application restarts. IDs generated by the TransientIdGenerator are guaranteed to be unique within a JVM as long as it runs, but a TransientIdGenerator does not maintain any persistent record of which IDs were generated.

ObfuscatedSQLIdGenerator

In some ATG applications, you might want to generate IDs that, for security purposes, cannot be easily guessed. For example, ATG Commerce generates IDs for a gift certificates. The ObfuscatedSQLIdGenerator obfuscates the generated IDs in two ways. First, for a given batch of reserved IDs, it gives out only a few IDs. Second, IDs can be optionally hex encoded when being used as String IDs. Both the String and long IDs generated use a pseudo-random number generator to get a long ID from the current batch of IDs. In addition to not giving out the same ID twice, this implementation is not given out adjacent long IDs (or String IDs that come from adjacent long IDs).

The IdSpace properties batchSize and idsPerBatch are used in conjunction. The batchSize property works as in the SQLIdGenerator. The idsPerBatch property is the maximum number of IDs that are given out in any given batch.

It is strongly recommended that idsPerBatch be less than 1 percent of the batchSize. This is both for security and performance. For security, a sparse—that is, less dense—ID space makes it harder to guess IDs. Because this implementation does not give out adjacent IDs, it might be forced to do more work to find suitable IDs if the ID space is too dense. This implementation does not allow an ID space to be added that is denser than 10 percent. That is, idsPerBatch divided by batchSize must be less than 0.1. Always set these two properties together to maintain the 1 percent density goal.

The recommended values for batchSize and idsPerBatch are 100000 and 997, respectively. These numbers are not magic: 100000 is the default batch size, while 997 is a prime number that is slightly less than 1 percent of the batch size.

Extending the IdGenerator

The SQLIdGenerator and TransientIdGenerator implement the atg.service.idgen.IdGenerator interface, and extend the AbstractSequentialIdGenerator abstract class. If you want to create your own IdGenerator implementations, it is probably best to extend AbstractSequentialIdGenerator.

The SQLIdGenerator and TransientIdGenerator implementations happen to generate sequential IDs, but that does not have to be true for all IdGenerator implementations. The AbstractSequentialIdGenerator includes two empty hook methods, postGenerateLongId() and postGenerateStringId(), that you can override in a subclass to provide additional ID generation logic.

Resource Pools

Most ATG applications must be able to handle large numbers of simultaneous requests. In these applications, one of the keys to improving throughput is to share and reuse expensive resources.

For example, a single JDBC database connection might require several seconds to establish a connection to a database and verify the password. After it is connected, however, a JDBC connection can be used repeatedly to execute database operations quickly. So one of the tricks to achieving high throughput is to create a pool of JDBC connections ahead of time. When requests come in, they grab connections from the pool and use them, then return them to the pool when they are done. This approach is far more efficient than requiring each request to create its own connection.

This pooling approach is applicable for any resource that is expensive to create, but cheap to share and reuse. ATG includes a class called `atg.service.resourcepool.ResourcePool` that encapsulates the notion of pooling shared resources. Subclasses can be defined which create their own types of resources. For example, the `atg.service.jdbc.MonitoredDataSource` class is a subclass of `ResourcePool` that pools JDBC `Connection` objects.

Subclassing ResourcePool

The most common way to use `ResourcePool` is to subclass it and override the methods `createResource` and `destroyResource`. The `createResource` method creates a new instance of your expensive resource. The `destroyResource` method is called when the resource pool decides that the resource is no longer needed, and is called to give you a chance to perform any cleanup procedures required for that resource.

You also have the option of overriding `verifyResourceValidity`. This is called to make sure that a resource is still available for use—for example, it can detect if a previously opened connection was closed since the last time it was used.

The following example shows how one might subclass `ResourcePool`:

```
import atg.service.resourcepool.*;

public class MyPool extends ResourcePool {
  public MyPool () { }

  public Object createResource () throws ResourcePoolException {
    return new ReallyExpensiveObject ();
  }

  public void destroyResource (Object resource) {
    ((ReallyExpensiveObject) resource).close ();
  }
}
```

Notice that `createResource` throws `ResourcePoolException`. If your object creation procedure results in an error, you must throw a `ResourcePoolException` to report that error.

Configuring a Resource Pool

Like all other components, your resource pool is created and configured through properties files. You can use the following properties to configure a resource pool:

min

The property `min` sets the minimum number of resources the pool should start out with. Because resource creation can be expensive, some applications require a starting minimum number of resources already in the pool before the pool becomes active. This minimum is only a starting minimum and is not maintained throughout the life of the pool. As invalid resources are checked back into the pool, the number of pooled resources can drop below the starting minimum. After startup, resource creation is driven by resource demand.

max

The maximum number of objects that can be kept in the pool. This includes both free objects and objects already in use.

blocking

If someone tries to check out a resource, and all free resources are currently checked out, the resource pool creates a resource. But if the `max` number of resources has been reached, the resource pool can perform one of the two following actions, depending on the value of the `blocking` property:

- If `blocking` is `false`, an exception is thrown, indicating that there are no more resources available.
- If `blocking` is `true`, the default setting, the resource pool can block and wait until someone else returns a resource to the pool. When that happens, the resource is passed to the waiting customer.

checkoutBlockTime

You can use this property to set a limit on how long a resource pool can block. The value of this property is the maximum time in milliseconds to block waiting for a resource on checkout. If this time limit is reached, an exception is thrown. A value of zero (the default) indicates indefinite blocking.

warnOnNestedCheckouts

This setting enables or disables warnings about nested resource checkouts that might cause deadlocks.

maxFree

Certain types of resources can be expensive to keep around if they are not being used. In this case, you might want to limit the number of resources kept around unused. The `maxFree` property indicates how many resources are to be kept around that are not in use by other services. This might be different from the `max` property, which indicates how many total resources are to be kept, both used and unused.

If, when a resource is checked into the pool, the number of resources then in the pool is greater than both the `maxFree` property and the `min` property, the pool destroys the resource being checked.

The default value for `maxFree` is `-1`, indicating that the number of maximum free resources is not limited except by the `max` property. This is usually the case, as there is rarely a need to destroy unused resources.

maxSimultaneousResourcesOut

When you are designing your application, it can be difficult to predict how many resources you need to make available in a pool. The `maxSimultaneousResourcesOut` property keeps track of the largest number of resources that were checked out of the resource pool at one time. You can examine this property during testing and after deployment to get an idea of the maximum number of resources your application requires. If the value of `maxSimultaneousResourcesOut` is significantly less than the value of `max`, you can probably reduce the size of your resource pool.

maxThreadsWithResourcesOut

The maximum number of threads that can have resources checked out of the pool concurrently.

maxResourcesPerThread

The maximum number of resources a thread can check out of the pool concurrently.

Using a Resource Pool

After you define a resource pool component, other Nucleus components can use that resource pool to check out resources and check them back in when they are done. A component that needs to use the resource pool can be passed a pointer to the resource pool through a properties file. It defines a property for that resource pool as follows:

```
ResourcePool resourcePool;
public ResourcePool getResourcePool ()
{ return resourcePool; }
public void setResourcePool (ResourcePool resourcePool)
{ this.resourcePool = resourcePool; }
```

When the component requires a resource from the pool, it calls `checkOut`:

```
try {
  // Get a resource
  ResourceObject resource =
    getResourcePool ().checkOut (getAbsoluteName ());
catch (ResourcePoolException exc) {
  if (isLoggingError ()) logError (exc);
}
```

This line gets the resource pool and checks out a resource from the pool. When it calls `checkOut`, it must pass an identifying string, in this case the name of the service checking out the resource. This is required so that the administrator can look at the resource pool and see which parts of the application are using which resources.

The object returned by `checkOut` is of type `ResourceObject`. This object contains the resource you wanted in the first place. You obtain the resource by calling `getResource`:

```
try {
  // Get a resource
  ResourceObject resource =
    getResourcePool ().checkOut (getAbsoluteName ());
  ReallyExpensiveObject obj = (ReallyExpensiveObject)
    resource.getResource ();
}
catch (ResourcePoolException exc) {
  if (isLoggingError ()) logError (exc);
}
```

After you obtain the resource, it is yours to use. You can work on the assumption that no one else uses the resource at the same time.

When you are done with the resource, you must check it back in:

```
try {
  // Get a resource
  ResourceObject resource =
    getResourcePool ().checkOut (getAbsoluteName ());
  ReallyExpensiveObject obj = (ReallyExpensiveObject)
    resource.getResource ();

  ...

  getResourcePool ().checkIn (resource);
}
catch (ResourcePoolException exc) {
  if (isLoggingError ()) logError (exc);
}
```

After checking in the resource, you are expected to no longer use that resource. If you need the resource again, you must check out another resource from the pool.

Avoiding Resource Leaks

One of the most common mistakes that occurs when using a resource pool is forgetting to check resources back in. This leads to resource leaks, a condition where resources disappear faster than expected, until no resources are left and the application locks up waiting for resources that never appear.

The most obvious way to avoid resource leaks is to make sure that for every `checkOut` you have a corresponding `checkIn`. This should be a fairly easy error to catch, because forgetting to do this causes you to run out of resources fairly quickly.

A subtler problem arises in the case where an exception occurs, terminating the operation and bypassing the `checkIn` call. If exceptions occur infrequently, it takes longer for your application to run out of resources, and it is far harder to debug because your application appears to lock up at random intervals.

The way to avoid this problem is to put the `checkIn` call inside of a `finally` statement, thereby ensuring that no matter what happens, the `checkIn` call is still made.

So the code from the previous example looks like this:

```
ResourceObject resource = null;
try {
  // Get a resource
  resource = getResourcePool ().checkOut (getAbsoluteName ());
  ReallyExpensiveObject obj = (ReallyExpensiveObject)
    resource.getResource ();

  ...
}
catch (ResourcePoolException exc) {
  if (isLoggingError ()) logError (exc);
}
finally {
  if (resource != null) getResourcePool ().checkIn (resource);
}
```

Remember that this is not an optional coding style. Failing to program in this manner is almost guaranteed to cause your application to lock up at random, unexpected intervals.

Checking the Health of a Resource Pool

A common cause of performance problems is when request handling threads get hung up waiting for a resource from a resource pool that has become unresponsive. To limit this problem, you can set the `ResourcePool`'s `creationTimeLimit` and `maxPendingCreations` properties.

creationTimeLimit

When `creationTimeLimit` is set, if a resource creation fails and the attempt exceeded the value of `creationTimeLimit` in milliseconds, the resource pool is disabled. In addition, before an attempt to create a resource occurs, a check is made to see if a resource creation attempt already in progress has exceeded the `creationTimeLimit`. If so, the resource pool is disabled.

maxPendingCreations

If you set the `maxPendingCreations` property, the resource pool has a limit on the maximum number of resource creation attempts that can be pending at one time. This can prevent a situation where all available request handling threads are tied up trying to create resources in an unresponsive resource pool.

The resource pool is disabled if the `maxPendingCreations` property is set to a value other than zero, and the following conditions are also true:

- The resource pool is not in its startup cycle.

- The minimum resources (set by the `min` property of the resource pool) is greater than zero.
- There are no valid resources being managed by the resource pool.
- The number of pending resource creation attempts exceeds the value of the `maxPendingCreations` property.

Disabled ResourcePools

When a resource pool is marked as disabled, it can still attempt to create resources when a thread attempts to check out resources from the pool. However, only one thread at a time can do so. Any other threads are returned a `ResourcePoolException`. This prevents more than one thread at a time from getting hung on a disabled pool. The resource pool is not shut down; it is simply marked disabled so threads seeking resources know that the resource pool is not behaving properly. The pool is marked enabled as soon as there is a successful resource creation.

ResourceClassName

The previous sections demonstrated how to subclass `ResourcePool` to create the type of resource object to be managed by your pool. The `ResourcePool` class provides a shortcut that lets you create resource objects without subclassing `ResourcePool`. You can use this shortcut if your resource object fits the following criteria:

- The resource object has a public constructor that takes no arguments.
- The resource object requires no special initialization beyond the constructor.
- The resource object requires no special cleanup operations to be performed when the resource object is destroyed.

If your resource object fits these criteria, you can use the base `ResourcePool` class without defining a subclass. To do this, specify an extra property called `resourceClassName`. This property should define the full class name of the resource object. For example:

```
resourceClassName=atg.resources.ReallyExpensiveObject
```

Now, whenever the resource pool requires a new object, it calls:

```
new atg.resources.ReallyExpensiveObject()
```

When the resource is no longer needed, the resource pool simply discards the object without calling any special notification methods.

MonitoredDataSource

The type of resource pool most used by an ATG application is the `MonitoredDataSource`. This service is a resource pool that is used to pool JDBC `Connection` objects. These resources represent connections to databases that are established and ready for use.

Events and Event Listeners

JavaBeans provides a way for a JavaBean to declare events that can be fired by that bean. Such a bean that fires events is called an event source. Other objects can register with the event source to receive those events when they are fired. These objects are called the event listeners.

The JavaBeans specification explains how to create event sources and listeners. The following is a short example that demonstrates how Nucleus services can fire and listen for events. In this example, a `StockPricer` service fires `stock` events whenever it receives an update for a stock price. The `StockWatcher` service listens for these events and prints them out.

Event Objects

First, create an object that represents the event. The class name of the event is based on the event name, which in this case is `stock`. Thus, the event is called `StockEvent`:

```
public class StockEvent extends java.util.EventObject {
  String symbol;
  double price;

  public StockEvent (Object source,
                     String symbol,
                     double price) {
    super (source);
    this.symbol = symbol;
    this.price = price;
  }
  public String getSymbol () { return symbol; }
  public double getPrice () { return price; }
}
```

Event Listener and Event Source Requirements

Next, define an interface for objects that listens for `stock` events. In the example, the interface defines a single method that is called when a stock price is updated. The interface must be named based on the event name, and must extend `java.util.EventListener`.

```
public interface StockListener extends java.util.EventListener {
  public void stockPriceUpdated (StockEvent ev);
}
```

The event source Bean must include the following methods with these exact signatures:

```
public void addStockListener (StockListener listener);
public void removeStockListener (StockListener listener);
```

All of the above is taken directly from the JavaBeans specifications.

Event Listener Example

Now, create the definitions for the `StockPricer` and `StockWatcher` services. First, define the listener service:

```
public class StockWatcher implements StockListener {
  public StockWatcher () {
  }
  public void stockPriceUpdated (StockEvent ev) {
    System.out.println ("Stock " + ev.getSymbol () +
                        " is at price " + ev.getPrice ());
  }
}
```

Not much is needed here. Like all services, this requires a public constructor with no arguments. Aside from that, it implements `StockListener` by printing the current stock price.

Event Source Example

The implementation of the event source is not much more complex:

```
public class StockPricer {
  java.util.Vector listeners = new java.util.Vector ();

  public StockPricer () {
  }
  public synchronized void addStockListener (StockListener listener) {
    listeners.addElement (listener);
  }
  public synchronized void removeStockListener (StockListener listener) {
    listeners.removeElement (listener);
  }
  public void broadcastStockEvent(StockEvent ev);{
    java.util.Enumeration e = listeners.elements()
    while( e.hasMoreElements() )
      ((StockListener) e.nextElement()).stockPriceUpdated(ev);
    }
}
```

This implementation uses a `Vector` to store the list of listeners. In order for this to be recognized as an event source, only the `addStockListener` and `removeStockListener` methods must be declared. The `broadcastStockEvent` method is a convenience method created to send the event to all listeners. Another useful method you might use exposes the listeners as a property:

```
public synchronized StockListener [] getStockListeners () {
    StockListener [] ret = new StockListener [listeners.size ()];
    listeners.copyInto (ret);
    return ret;
  }
```

Testing the Event System

Now create the two services. Create a `localconfig/test/services/stockWatcher.properties` file that looks like this:

```
$class=StockWatcher
```

And create a `localconfig/test/services/stockPricer.properties` file that looks like this:

```
$class=StockPricer
stockListeners=stockWatcher
```

The `stockListeners` property is recognized by Nucleus as indicating that the specified services act as listeners for the `stock` event. If your event source has multiple listeners, those listeners should be separated by commas. This means that the Bean should avoid creating a property called `stockListeners`.

Modify `localconfig/Initial.properties` to specify the initial service:

```
initialService+=/services/stockPricer
```

Now restart the application. This creates the `stockPricer` object, then creates the `stockWatcher` to listen to the `stock` events. Because no one is actually sending any events, nothing should actually happen.

In the following example, `stockWatcher` starts a thread that waits for 4 seconds, then fires an event.

Note: This example is for demonstration purposes only, and should not be considered a general programming technique.

```
public class StockPricer implements Runnable {
  java.util.Vector listeners = new java.util.Vector ();

  public StockPricer () {
    new Thread (this).start ();
  }
  public void run () {
    try { Thread.sleep (4000); }
    catch (InterruptedException exc) {}
    broadcastStockEvent (new StockEvent (this, "ATGC", 20.75));
  }
  public synchronized void addStockListener (StockListener listener) {
    listeners.addElement (listener);
  }
  public synchronized void removeStockListener (StockListener listener) {
    listeners.removeElement (listener);
  }
  public synchronized StockListener [] getStockListeners () {
    StockListener [] ret = new StockListener [listeners.size ()];
    listeners.copyInto (ret);
    return ret;
  }
  public void broadcastStockEvent (StockEvent ev) {
    for (int i = 0; i < listeners.size (); i++) {
      ((StockListener) listeners.elementAt (i)).stockPriceUpdated (ev);
    }
  }
}
```

Now reassemble your application. When you restart it, the `StockPricer` should wait 4 seconds, then fire an event to the `StockWatcher`, which prints the event out.

Queues

Queues are the most important piece of architecture you can put into an application to improve throughput and remove bottlenecks. There are many cases where an application creates contention on certain resources where contention is not necessary. Queues eliminate these points of contention.

For example, a request handler might log every single page view to a log file. Suppose that it takes 50 milliseconds to write a line to a log file. If that is the case, the request handler cannot serve requests any faster than 20 per second, even if the rest of the request handling mechanism is blazingly fast. But writing a line to a log file is not a critical part of the request handling operation, and thus should not be such a limiting factor.

The solution to this problem is to introduce a queue between the request handler and the logging facility. When the request handler wants to log a message, it places the message on the queue then continues handling the rest of the request. A separate thread removes messages from the queue and writes them to the log. This arrangement decouples the request handlers from the loggers, thereby eliminating the bottleneck introduced by the log.

In ATG, the notion of queues is generalized to all types of JavaBean event listeners. ATG comes with a utility that generates the Java code needed to create an `EventQueue` class specialized to a type of `EventListener`. This queue class implements the same interface as the original `EventListener`, so to other components it acts the same as the original component. Calls made to the queue class are handled by being placed on a queue. A separate thread then takes the calls from the queue and passes them to the original component.

After you realize that a particular component is a bottleneck, you should be able to create and insert a queue component for that component, without changing any of your other components. The following topics describe how to use queues:

Candidates for Queuing

Queuing is not always an appropriate way to improve throughput. Queuing is usually appropriate only when an event occurs during a process, and the process does not care about the result of the event. Logging a message is a perfect example, where a process fires off the logging event, but the rest of the process does not depend on the result of writing that message to the log. In that case, any time spent waiting around for the log message to be written is wasted, and is best pushed off to the other side of the queue.

However, a process that makes a call to a database and returns the results to the user is not a good candidate for queuing. In that case, the process depends on the result of the operation, and must wait for the operation to occur.

Here are two good rules of thumb for deciding if an operation is a good candidate for queuing:

- Is the operation best thought of as an event—that is, should that operation be encapsulated in an `EventListener` interface?
- Does the method that invokes the operation return `void`? If so, it indicates that the component does not depend on the result of the event.

Creating a Queue Class

A queue class is generated for a specific interface, usually an `EventListener` interface. For example, consider the `atg.nucleus.logging.LogListener` interface:

```
package atg.nucleus.logging;
import java.util.EventListener;
public interface LogListener extends EventListener {
  public void logEvent (LogEvent logEvent);
}
```

This is a simple `EventListener` interface where the relevant methods return `void`—a good candidate for queuing.

EventQueueGenerator Utility

To create the appropriate queuing class for this interface, ATG includes a utility class called `atg.service.queue.EventQueueGenerator`. The class is run using the `java` command, like this:

```
javan ATG.service.queue.EventQueueGenerator\
  atg.nucleus.logging.LogListener\
  mypackage.queues\
  LogListenerQueue
```

The first argument is the name of the interface for which you wish to generate a queuing class. The second and third arguments are the package and class name of the new queuing class.

The output of the command is written to `stdout`, so you can redirect the contents to a file like this:

```
javan ATG.service.queue.EventQueueGenerator\
  atg.nucleus.logging.LogListener\
  mypackage.queues\
  LogListenerQueue > LogListenerQueue.java
```

You should place the resulting `.java` file into the correct package of your source hierarchy. Like all of your other source files, you must compile this one and add it to source control as if you created this class yourself.

The resulting class looks fairly cryptic if you examine it yourself. But it has the following important characteristics:

- It implements `LogListener`, so anything that used to send events to a `LogListener` can send events to this queue instead.
- It implements `addLogListener` and `removeLogListener`. This means that the class is a source of `LogEvents`, as well as a listener for `LogEvents`.

Using a Queue Component

A Queue class acts as an event filter. It listens for events and places those events on a queue. Another thread pulls events from that queue and rebroadcasts them to the queue's listeners. This means that you can interpose a Queue between two components that originally had an event source/event listener relationship.

For example, say that component A generates LogEvents and broadcasts them to any listeners. Component B listens for LogEvents from A and writes the log events to a log:

```
A -> B -> file
```

Now say that component B is starting to hamper the throughput of component A because of the time required to write to a file. The solution is to interpose a `LogListenerQueue` as component Q:

```
A -> Q -> B -> file
```

This can be done purely through changing configuration files. Neither components A nor B need to know that there is a queue sitting between them.

The original configuration files for A and B might look like this:

```
A.properties:

$class=blah.blah.LoggingSource
logListeners=B

B.properties:

$class=atg.nucleus.logging.FileLogger
logFileName=events.log
```

With the queue component interposed, the configuration files look like this:

```
A.properties:

$class=blah.blah.LoggingSource
logListeners=Q

Q.properties:

$class=atg.nucleus.logging.LogListenerQueue
logListeners=B

B.properties:

$class=atg.nucleus.logging.FileLogger
logFileName=events.log
```

Configuring a Queue Component

In general, you should be able to configure a queue component just by specifying a list of listeners, as shown in the previous example. There are, however, two additional properties you might want to change:

initialCapacity

This property sets the initial size of the queue, specifying how many elements can be queued up before the queue must resize itself. The queue automatically resizes itself, so it is usually not necessary to set this property. Its default value is 16. For example:

```
initialCapacity=32
```

threadCount

This property specifies the number of threads that are to pull events from the queue. By default, this is set to 1. You might wish to increase this number if it makes sense to handle multiple events in parallel, and if you are not concerned with the order events are handled. This value should always be set to at least 1.

Email Senders and Listeners

ATG includes a facility for sending email, and a JavaMail-based implementation for sending Internet email through SMTP. The email interface is called `atg.service.email.EmailListener`, and the SMTP implementation is called `atg.service.email.SMTPEmailSender`. Internally, `SMTPEmailSender` uses JavaMail's SMTP implementation to send the email.

Email is sent using an event listener model. A single piece of email is described by an `atg.service.email.EmailEvent`. The `SMTPEmailSender` implements `EmailListener`, so you can send a piece of mail by calling `sendEmailEvent()` on the `SMTPEmailSender`, passing it the `EmailEvent`.

This event source/event listener model lets you use `EventQueues` (see the Events and Event Listeners and Queues sections) to queue up email messages, thereby preventing email from becoming a bottleneck in high-throughput systems.

EmailEvent

An `EmailEvent` can be defined with the various properties that you expect for a piece of email: `From`, `Recipient`, `Subject`, and `Body.` You can also set additional headers to be sent in the mail, and specify a list of recipients as opposed to a single recipient.

For example:

```
EmailEvent em = new EmailEvent ();
em.setFrom ("dynamotester");
em.setRecipient ("test@example.com");
em.setSubject ("I'm just testing the e-mail sender");
em.setBody ("Sorry to bother you, but I'm testing the e-mail sender");
```

The `EmailEvent` also includes a number of constructors that simplify the construction of typical email events:

```
EmailEvent em =
  new EmailEvent ("dynamotester",
                  "test@example.com",
                  "I'm just testing the e-mail sender"
                  "Sorry to bother you, but I'm testing the e-mail sender");
```

You can also set a list of recipients:

```
String [] recipients = {
  "test@example.com",
  "noone@example.com"
};
em.setRecipientList (recipients);
```

Creating JavaMail Messages

While the examples above demonstrate how to create `EmailEvent` objects describing simple email messages, they do not show you how to create messages with different recipient types, multipart messages, or messages with file attachments. These more sophisticated capabilities can be achieved using JavaMail's `javax.mail.Message` class to describe the email message.

You can create Message objects yourself via method calls to one of the Message child classes, such as javax.mail.internet.MimeMessage. Alternatively, you can use the atg.service.email.MimeMessageUtils helper class to create and fill in MimeMessage objects. For example, here is how one might use MimeMessageUtils to create the simple email message shown in the previous section:

```
Message msg = MimeMessageUtils.createMessage();
MimeMessageUtils.setFrom(msg, "dynamotester");
msg.setSubject("I'm just testing the e-mail sender");
MimeMessageUtils.setRecipient(msg, RecipientType.TO, "test@example.com");
msg.setText("Sorry to bother you, but I'm testing the e-mail sender");
```

or, alternatively,

```
Message msg = MimeMessageUtils.createMessage
  ("dynamotester",
   "I'm just testing the e-mail sender",
   "test@example.com",
   "Sorry to bother you, but I'm testing the e-mail sender");
```

MimeMessageUtils can also be used to create much more complex Message objects. For example, here is how one might create a multi-part message with a text/plain part and a text/html part, a file attachment, and several kinds of recipients:

```
// create a Message with the given From and Subject
Message msg = MimeMessageUtils.createMessage("dynamotester",
                                             "more complex test");
// set the To and Bcc recipients
MimeMessageUtils.setRecipient(msg, Message.RecipientType.TO, "test@example.com");
MimeMessageUtils.setRecipient(msg, Message.RecipientType.BCC, "dynamotester");

// set the Cc recipients
String[] ccAddresses = { "fred@example.com", "jane@example.com" };
MimeMessageUtils.setRecipients(msg, Message.RecipientType.CC, ccAddresses);

// set the message content: multipart message + attachment
ContentPart[] content =
  { new ContentPart("this is plain text", "text/plain"),
    new ContentPart("this is <b>html</b> text", "text/html") };
File attachment = new File("attachment.html");
MimeMessageUtils.setContent(msg, content, attachment, false);
```

After you have a Message object, you can use it to set the email event's message property:

```
EmailEvent em = new EmailEvent();
em.setMessage(msg);
```

Or, more simply,

```
EmailEvent em = new EmailEvent(msg);
```

Registering Content Types

When using JavaMail `Message` objects to send email, you must specify the MIME type of the message content. ATG provides support for sending messages with content type of `text/plain` and `text/html`. If you need to send content that has some other MIME type, you must first register your content type with the JavaBeans Activation Framework (JAF), which JavaMail uses to handle message content.

For example, suppose you'd like to send messages with a MIME type of `application/x-foobar`. To do this, you must provide an implementation of `javax.activation.DataContentHandler` for your MIME type, say `FoobarDataContentHandler`. You must then create a mapping between the MIME type `application/x-foobar` and `FoobarDataContentHandler`, so that JAF can use it.

One general way to register a `DataContentHandler` with JAF is to provide a mailcap file with the appropriate content handler entry in it. Another way, which is ATG-specific but perhaps more convenient, is to configure the `/atg/dynamo/service/DataContentHandlerRegistry` component to know about your content type and the associated handler. The `dataContentHandlerMap` property of the `DataContentHandlerRegistry` contains a list of mappings between MIME types and the associated `DataContentHandler` class names. To register a new MIME type, simply add a mapping as follows:

```
dataContentHandlerMap+=\
       application/x-foobar=package.name.FoobarDataContentHandler
```

See the API documentation for the `javax.activation` package or the Oracle Web site for more information about JAF.

Sending Email

You can send email to an `EmailListener` by calling `sendEmailEvent()` on the listener, passing it a `EmailEvent` containing the email you want to send. Typically, a service broadcasts an `EmailEvent` to all of its attached `EmailListener` objects. For example, the following code defines a service to be a source of `Email` events and shows how to broadcast an `EmailEvent` to the listeners of that event:

```
Vector emailListeners = new Vector ();
public void addEmailListener (EmailListener listener) {
  emailListeners.addElement (listener);
}
public void removeEmailListener (EmailListener listener) {
  emailListeners.removeElement (listener);
}
public EmailListener [] getEmailListeners () {
  EmailListener [] ret = new EmailListener [emailListeners.size ()];
  emailListeners.copyInto (ret);
  return ret;
}
```

```
public void broadcastEmailEvent (EmailEvent event) {
  for (int i = 0; i < emailListeners.size (); i++) {
    try {
      ((EmailListener) (emailListeners.elementAt (i))).sendEmailEvent (event);
    }
    catch (EmailException exc) {}
  }
}
```

The properties file configuring your service can then hook up the listeners of the email events like this:

```
emailListeners=/atg/dynamo/service/SMTPEmail
```

Now, when you call `broadcastEmailEvent`, your email is sent through the `/atg/dynamo/service/SMTPEmail` component, which sends the email.

Configuring SMTPEmail

ATG comes with a standard component of type `SMTPEmailSender`, located at `/atg/dynamo/service/SMTPEmail`. You can send your email messages to the `SMTPEmail` component, or you can create your own emailer component.

You can configure the `SMTPEmail` component to define several default properties—for example, for `From`, `Recipients`, and `Subject`. If one of these default properties is set, the default is used in when the corresponding property is not set in the `EmailEvent`. The `defaultBody` property is an exception—if the `defaultBody` property is set, that `defaultBody` is prepended to all email messages, whether they specify a body or not.

Note: The `defaultFrom` and `charSet` properties must be set to ASCII values. `SMTPEmail` cannot send email if either of these properties has a non-ASCII value.

The `emailHandlerHostName` and the `emailHandlerPort` properties should be set to the name of the host (usually remote) that sends the email, and the port number it uses. The default value for `emailHandlerHostName` is `localhost`, and the default value for `emailHandlerPort` is 25.

Some SMTP servers require authentication. If your SMTP server requires authentication, set the values of the `username` and `password` properties of the `SMTPEmail` component to the username and password for the account it uses to send email.

You might want to increase the `waitForConnectionMillis` property to reduce timeouts; the default is 5 seconds. To increase the setting, modify the `waitForConnectionMillis` property in `<ATG10dir>/home/localconfig/SMTPEmail.properties`. For example:

```
waitForConnectionMillis=30000
```

Using BatchEmailListener

Each time an `SMTPEmailSender` is used to send an `EmailEvent`, an SMTP connection is opened to the mail server, the email is sent, and the connection is closed. A new SMTP connection is opened and closed every time an email is sent, even if you are calling `sendEmailEvent` continuously to send multiple email messages. This can be costly and unnecessary if many messages need to be sent at once.

A `BatchEmailListener` performs batch sending of email over a single connection to the mail server. Like `SMTPEmailSender`, it implements `EmailListener`; but instead of sending email after receiving each `EmailEvent`, it collects the `EmailEvent` objects and periodically sends the messages out in batches. The `emailMessageSender` property points to the component that actually performs the message sending, for example, `SMTPEmail`.

Two properties of `BatchEmailListener` control how often the batch sends are performed, `maxBatchSize` and `sendSchedule`. If `maxBatchSize` is specified, a send is performed whenever the number of batched email events reaches `maxBatchSize`. Also, if `sendSchedule` is specified, sends are performed according to the given schedule.

ATG comes with an instance of `BatchEmailListener` at `/atg/dynamo/service/SMTPBatchEmail`. This batch listener points to `SMTPEmail` as its `emailMessageSender`. The default configuration has no `maxBatchSize`, and a `sendSchedule` which calls for a send to be performed every 3 minutes.

Using EmailListenerQueue

Sending email can be an expensive operation, and you generally do not want your components waiting for email to be sent. To prevent email from being a bottleneck, ATG includes an `EmailListenerQueue` class. This class again implements `EmailListener`, so it can be used in place of the `SMTPEmail` component. Any email messages sent to components of this class are queued up and handled by a separate thread, freeing your component from the potential bottleneck. That separate thread pulls email messages from the queue and sends them to another `EmailListener`, such as the `SMTPEmail` component or the `SMTPBatchEmail` component.

ATG comes with an instance of `EmailListenerQueue` at `/atg/dynamo/service/SMTPEmailQueue`. This queue empties into the `/atg/dynamo/service/SMTPBatchEmail` component. Thus, if you send your email events to the queue, they are first queued and then batched, for maximum performance. If you wish your email queue to empty directly into the `SMTPEmail` component, simply override the `SMTPEmailQueue` configuration such that its `emailListeners` property points to `SMTPEmail` rather than `SMTPBatchEmail`.

You probably want to configure your services to send email to the queue, rather than going directly to the `SMTPEmail` or the `SMTPBatchEmail` component:

```
emailListeners=/atg/dynamo/service/SMTPEmailQueue
```

11 Logging and Data Collection

ATG includes three different systems for sending, receiving, and recording messages generated by components: Logging, Data Collection, and Recorders. ATG Logging provides a convenient way to log system messages. Any component that extends the `GenericService` class or implements the `ApplicationLogging` interface can send `LogEvents` that are received by log listeners and logged to a flat file. The logging system can log only text messages and exceptions.

Like Logging, Data Collection is based on the Java event model. But Data Collection lets you record data contained in any JavaBean (not just subclasses of `LogEvent`). Therefore, your choice of Beans to use as data items is not restricted. In addition, Data Collection provides in-memory summarization, which makes it suitable for handling the data needs of more demanding applications.

Recorders collect data through a combination of scenario events, mappers, and datasets.

Note: If you are running the DSS module, use recorders rather than the Data Collection techniques described in this chapter. See the *ATG Personalization Programming Guide* for more information.

In this chapter

This chapter includes the following sections:

- ATG Logging: ATG's message logging system, based on the `atg.nucleus.logging` API.
- Data Collection Sources and Events: Data collection begins with a data collection source, which can be any JavaBean with a `dataListeners` property. Events are generated by data sources and sent to data listeners.
- Data Listeners: Data collection events are received by data listeners, which then process them.
- Formatting File Loggers: A type of data listener that logs data to a file.
- Database Loggers: A type of data listener that logs data to a database.
- Data Collector Queues: A type of data listener that stores data in a queue, before flushing it to another data listener.
- Summarizers: A type of data listener that accumulates data events and passes a summary of the data to another data listener.

ATG Logging

You can use the ATG logging facility as a method for producing logging events that can be used by any component. Use the message logging facility for error, warning, debug, or informational messages that need to be communicated to a developer or administrator.

Logging is performed through JavaBeans events. `LogEvent` objects contain logging messages, and are broadcast by components that log those messages. Events are received by `LogListener` objects that handle the logging events in various ways. A `LogListener` object might perform these tasks:

- Write events to a log file.
- Send email.
- Dispatch events to multiple listeners.

The separation between log source and log listener allows for a flexible logging configuration that can be tailored to individual applications.

The following topics describe how message logging works in an ATG application:

- LogEvents
- LogListeners
- Logging Levels
- Broadcasting LogEvents
- Using ApplicationLogging
- Improving Log Readability
- Using Terse Logging
- Implementing Logging

LogEvents

In an ATG application, log messages are treated as JavaBeans events. Each log message is encapsulated in an `atg.nucleus.logging.LogEvent` object. Various types of messages are represented by subclasses of `LogEvent`, such as `ErrorLogEvent` and `WarningLogEvent`. When a component wants to send a logging message, it creates a `LogEvent` object of the appropriate class, containing the contents of the message. Those contents can include a String message, a Throwable, or both. The component then broadcasts that event object to all listeners.

Components that implement interface `atg.nucleus.logging.ApplicationLogging` can act as sources of `LogEvents`. Because `GenericService` implements `ApplicationLogging` and Nucleus components extend `GenericService`, Nucleus components all follow the ATG logging conventions and can act as sources of error, warning, info and debug `LogEvents`.

LogListeners

In keeping with the JavaBeans specifications, objects that receive logging messages must implement the `LogListener` interface. This also means that log sources must have `addLogListener` and `removeLogListener` methods.

ATG provides several `LogListener` implementations that perform the following tasks:

- Write log messages to files, the console, and so on. See LogEvent Sinks and the components in `/atg/dynamo/service/logging`.
- Dispatch a log message to one of several destinations, so error events are written to one file, warning events are written to another file, and so on. See DispatchLogger.
- Queue log events from various components before sending them to their final destinations. A component can send a log event without waiting for the event to be written to disk; the event is sent to the queue, which later passes the event on to the listener that eventually writes it to the file. See LogListenerQueue.

A log source does not need to know where its log messages go, whether they are queued, and so on. Because listeners are defined in properties files, all logging decisions are configurable. The log source is only responsible for generating and broadcasting logging messages.

Logging Levels

As installed, ATG defines four standard logging levels:

Level	Description
`Error`	Represents fault conditions that indicate an immediate problem. Default: Log all error messages.
`Warning`	Represents fault conditions that might indicate a future problem. Default: Log all warning messages.
`Info`	Represents events that occur during the normal operation of the component. For instance, server messages indicating handled requests are usually sent as Info messages. Default: Log all info messages.
`Debug`	Represents events specific to the internal workings of the component that should only be needed for debugging situations. Default: Do not log debug messages.

A log source can emit logging events at one or more of these levels. Individual components can enable or disable logging messages at any level through the Boolean properties `loggingError`, `loggingWarning`, `loggingInfo`, and `loggingDebug`. These components must implement the following methods:

```
public void setLoggingError (boolean loggingError);
public boolean isLoggingError ();
public void setLoggingWarning (boolean loggingWarning);
public boolean isLoggingWarning ();
public void setLoggingInfo (boolean loggingInfo);
public boolean isLoggingInfo ();
public void setLoggingDebug (boolean loggingDebug);
public boolean isLoggingDebug ();
```

Before sending a log message, a component should check whether logging is enabled for that log message's level, in order to avoid unnecessary overhead. For example:

```
// Log an error
if (isLoggingError ()) {
  // create and broadcast the logging message
}
```

Broadcasting LogEvents

In order to send a log message, a component must create a `LogEvent` object, then broadcast that object to all `LogListener` objects attached to the component.

This operation is best placed in a method that can be called quickly. For example, the following implementation includes such convenience methods:

```
Vector mLogListeners;
public synchronized void addLogListener (LogListener pListener){
  if (mLogListeners == null) mLogListeners = new Vector ();
  mLogListeners.addElement (pListener);
}
public synchronized void removeLogListener (LogListener pListener){
  if (mLogListeners != null)
  mLogListeners.removeElement (pListener);
}
public int getLogListenerCount (){
  return (mLogListeners == null) ? 0 : mLogListeners.size ();
}
public synchronized void sendLogEvent (LogEvent pLogEvent){
  if (mLogListeners != null) {
    int len = mLogListeners.size ();
    for (int i = 0; i < len; i++) {
      ((LogListener) mLogListeners.elementAt (i)).logEvent (pLogEvent);
    }
  }
}
public void logError (String pMessage){
  logError (pMessage, null);
}
public void logError (Throwable pThrowable){
  logError (null, pThrowable);
}
public void logError (String pMessage, Throwable pThrowable){
  sendLogEvent (new ErrorLogEvent (pMessage, pThrowable));
}
```

With these methods available, the component can now send error events like this:

```
// Log an error
if (isLoggingError ()) {
  logError ("Look out, it's gonna blow!");
}
```

Using ApplicationLogging

ATG includes an interface called `atg.nucleus.logging.ApplicationLogging` that encapsulates the above concepts. It also includes a sample implementation of this interface named `atg.nucleus.logging.ApplicationLoggingImpl`.

For each logging level, `ApplicationLogging` defines the following methods:

```
public void setLoggingError (boolean loggingError);
public boolean isLoggingError ();
public void logError (String str);
public void logError (Throwable t);
public void logError (String str, Throwable t);
```

Similar methods are also defined for warning, info, and debug log levels.

`ApplicationLoggingImpl` also includes the methods that define a component as a source of log events:

```
public void addLogListener (LogListener listener);
public void removeLogListener (LogListener listener);
```

The `ApplicationLogging` interface is meant to serve as a template for components that wish to follow ATG logging conventions. This is useful for developers that wish to subclass an existing component. If you know that the base component already implements `ApplicationLogging`, you can follow the ATG conventions for sending logging messages in the subclass.

Components that are derived from `GenericService` automatically inherit all of these behaviors because `GenericService` implements `ApplicationLogging`. Components that are unable to subclass `GenericService` can also implement `ApplicationLogging`. The source code for the sample implementation, located at `<ATG10dir>/DAS/src/Java/atg/nucleus/logging/ApplicationLoggingImpl.java`, can be used as the template for such implementations.

Improving Log Readability

To improve the readability of logged output, you can configure certain properties in the log listener component. The following table shows the properties that you can set on components of these classes:

```
atg.nucleus.logging.FileLogger
atg.nucleus.logging.RotatingFileLogger
atg.nucleus.logging.PrintStreamLogger
```

Property	Description
`cropStackTrace`	Boolean, determines whether to show the entire stack trace. This option is typically set to `false` only in development environments. Set to `true` for production environments, in order to prevent excessive log file growth. **Note:** If set to `true`, sure to set `maxLinesInStackTrace` to a value that provides enough information to troubleshoot potential problems—in general, 100 or greater. Default: `false`
`maxLinesInStackTrace`	If `cropStrackTrace` is set to true, sets the maximum number of lines to log when a log event occurs that contains a Java exception.
`prefixEachLine`	Boolean, determines whether to prepend the logging prefix (date and component name) to each line of logging output for multi-line log messages.

You can configure the values of these properties in each of the following log listener components:

```
/atg/dynamo/service/logging/{DebugLog, ErrorLog, InfoLog, WarningLog, ScreenLog}
```

The default value for `cropStackTrace` is `true`. The default value for `maxLinesInStackTrace` is 10. The default value for `prefixEachLine` is `true`.

Using Terse Logging

The `atg/dynamo/service/loggingScreenLog` component lets you see shortened versions of logging information, in this form:

```
[type] time ComponentName Message
```

For example:

```
[i] 01:13:00 MBeanServer  MBeanServer, MBeanServer is running.
```

In the preceding example, `[i]` means info. Only the short hh`:`mm`:`ss time format is shown, with no date, and only the component name (`MBeanServer`) is shown.

The first time a component appears in the log, the log prints out a name mapping, identified by a [k], denoting a key message:

```
[k] MBeanServer --> /atg/dynamo/service/management/MBeanServer
[i] 01:13:00 MBeanServer  MBeanServer, MBeanServer is running.
```

If there are multiple components with the same name at different paths (such as is the case with MBeanServer), the terse logger differentiates them like this:

```
[k] MBeanServer(2) --> /atg/management/MBeanServer
[i] 01:13:10 MBeanServer(2)  MBeanServerService started: domain = Dynamo
```

To use this feature, set the `terse` property on the `/atg/dynamo/service/logging/ScreenLog` component to `true`.

Note: You should use terse logging only during development, as fragments of terse logs do not contain complete component path and date information.

Implementing Logging

Logging is performed through JavaBeans events. To log a message, a component creates a `LogEvent` that includes the message, then broadcasts the event. Events are received by `LogListener` objects that handle the logging events in various ways. Some `LogListener` objects write events to a log file, some send email, some dispatch events to multiple listeners. The separation between log source and log listener allows for a flexible logging configuration that can be tailored to individual applications.

A `LogListener` can be either a `LogEvent` sink (performs a final action) or a `LogEvent` filter (sends an event to other `LogListeners`). The following sections describe how to implement log events:

- LogEvent Sinks
- DispatchLogger
- LogListenerQueue
- Logging Configuration
- Designing Logging Systems

LogEvent Sinks

A LogEvent sink is a `LogListener` that performs a final action on a `LogEvent`. This can include writing the `LogEvent` to a file, sending the `LogEvent` as email, or writing the `LogEvent` to a database. ATG defines several different kinds of `LogEvent` sinks:

- PrintStreamLogger
- FileLogger
- RotatingFileLogger
- EmailLogger

PrintStreamLogger

A `PrintStreamLogger` writes logging messages to a `PrintStream`. By default, a `PrintStreamLogger` is configured to write logging messages to `System.out`, which usually leads to the console.

A `PrintStreamLogger` is useful as a debugging tool during development. ATG defines a `PrintStreamLogger` called `/atg/dynamo/service/logging/ScreenLog` of the `atg.nucleus.logging.PrintStreamLogger` class. By default, the `ScreenLog` component is a `logListener` for all Nucleus components that implement `ApplicationLogging`. You can disable the `ScreenLog` component by setting its `loggingEnabled` property to `false`. This is the recommended setting for live ATG sites.

FileLogger

A `FileLogger` writes logging messages to a text file. Two properties define an instance of a `FileLogger`:

Property	Description
`logFilePath`	The path to the directory that holds the log file. The path can be relative to the directory where the ATG server runs. For example, `logFilePath=./logs` points to the `<ATG10dir>/home/logs` directory, while `logFilePath=logs` points to the `<ATG10dir>/home/servers/`*`<server>`*`/logs` directory.
`logFileName`	The actual name of the log file, within the `logFilePath`. So if `logFilePath` is `./logs`, and `logFileName` is `warnings.log`, the logging messages are written to `<ATG10dir>/home/logs/warnings.log`.

You can disable any `FileLogger` component by setting its `loggingEnabled` property to `false`.

RotatingFileLogger

A `RotatingFileLogger` is a subclass of `atg.nucleus.logging.FileLogger` that periodically archives its log file to another directory. This prevents log files from growing without bound, but still lets you keep some log file history around.

The archiving is controlled by the following properties:

Property	Description
`scheduler`	The `Scheduler` to use to perform the archiving. This is usually set to `/atg/dynamo/service/Scheduler`.

Property	Description
`schedule`	The `Schedule` to use to perform the archiving (see Configuring a Schedulable Component). This is often set to a `CalendarSchedule`, allowing it to perform the archiving on a calendar-based schedule such as `every Sunday morning at 1am`.
`logArchivePath`	The directory where the archived log files are to be placed. This is usually different from the `logFilePath`, to make it easier for you to manage your log files and your archive files separately.
`maximumArchiveCount`	This is the maximum number of archive files that are kept for a particular log file. After this maximum has been reached, the oldest file is discarded whenever the log file is archived.
`archiveCompressed`	Specifies whether log files are compressed before being archived. See below.

When the log file is archived, it is moved from the `logFilePath` to the `logArchivePath`, and is renamed `<logFileName>.0`. If there already is a `<logFileName>.0`, it is renamed `<logFileName>.1`. 1 is renamed to 2, 2 is renamed to 3, and so on. This rotation stops at the `maximumArchiveCount`. If the `maximumArchiveCount` is 10, `<logFileName>.9` is not moved to `<logFileName>.10`, but is instead erased.

After the log file is archived, a new log file is opened in the `logFilePath`, and logging continues as normal.

You also have the option of compressing log files before they are archived. If the `archiveCompressed` property is set to `true`, log files are compressed into a ZIP file format. The archived log files also have the extension `.zip`. These compressed log files can be read by a standard ZIP file reader, or by using the `jar` command that comes with the JSDK:

```
jar xvf info.log.0.zip
```

One example instance of `RotatingFileLogger` can be found at `/atg/dynamo/service/logging/InfoLog`. It has the following properties:

```
$class=atg.nucleus.logging.RotatingFileLogger
logFilePath=./logs
logFileName=info.log
logListeners=ErrorLog

scheduler=../Scheduler
schedule=calendar * . 1 1 0
logArchivePath=./logs/archives
maximumArchiveCount=20
archiveCompressed=true
```

EmailLogger

An `EmailLogger` takes log messages and sends them out as email to a list of recipients. This is useful for system administrators who wish to be notified whenever certain parts of the system malfunction. Administrators who use email-to-pager gateways can be paged when certain critical events take place.

The `EmailLogger` batches log messages before sending them as email. This is extremely valuable in situations where the system malfunctions in such a way that it is generating many error messages in a short amount of time. In such a situation, an administrator finds it much more helpful to receive, say, ten pieces of email with 100 error messages in each, than to receive 1000 messages with one error in each. The logger can be triggered to send its batched log messages when a certain number of messages are batched, or after a certain amount of time.

When the logger sends its email message, it generates an `EmailEvent`, which is then sent to an `EmailSender`.

The following properties control the configuration of an `EmailLogger`:

Property	**Description**
`logEventThreshold`	The number of log messages that are batched before being sent as email.
`schedule`	Using the above threshold, messages are not sent until the threshold is reached. So if the threshold is 10, and 9 log events are issued, email is still not sent until the 10th is received. By specifying a schedule, you can tell the `EmailLogger` to send out email according to a time trigger as well as a threshold. So if the `schedule` is set to `every 5 minutes`, email is sent within 5 minutes of receiving a log event, whether or not the log event threshold has been reached.
`scheduler`	If you are going to specify a `schedule`, you must also specify a `scheduler`. This is usually set to `/atg/dynamo/service/Scheduler`.
`emailListeners`	This is a pointer to the `EmailSender` that performs the task of sending email. This is usually set to `/atg/dynamo/service/SMTPEmailQueue`.
`defaultRecipients`	This is a comma-separated list specifying the email addresses of those for whom the email is intended. For example, `sysadmin@example.com,test@example.com`.
`defaultFrom`	This is what you want to appear in the `from` field of the email.
`defaultSubject`	This is what you want to appear in the `subject` field of the email.
`defaultBody`	Anything placed in here appears at the top of the email body. The log messages are placed after the `defaultBody`.

A sample `EmailLogger` can be found at `/atg/dynamo/service/logging/EmailLog`:

```
$class=atg.nucleus.logging.EmailLogger
emailListeners=../SMTPEmail
logEventThreshold=10
scheduler=../Scheduler
schedule=every 5 minutes
defaultRecipients=sysadmin@example.com,test@example.com
defaultFrom=Dynamo_Number_12
defaultSubject=Main Reactor Core Down
defaultBody=Run now!
```

DispatchLogger

A `DispatchLogger` is a `LogListener` that routes `LogEvents` to other `LogListeners` based on the types of those `LogEvents`. For example, you might wish to send `ErrorLogEvents` to an `EmailLogger`, while all other log event types are sent to a file.

A `DispatchLogger` is configured with the following properties:

logEvents

The class names of the different types of log events to be dispatched to various listeners. For example, to dispatch `ErrorLogEvents` and `WarningLogEvents` to different listeners, specify:

```
logEvents=\
        atg.nucleus.logging.ErrorLogEvent,\
        atg.nucleus.logging.WarningLogEvent
```

The next property, `logDestinations`, specifies where those two types of events are to be sent.

logDestinations

The names of the `LogListeners` that receive the log event types specified by the `logEvents` properties. For example:

```
logDestinations=\
        SysadminPager,\
        SysadminEmailer
```

This specifies that `ErrorLogEvents` are to be sent to the `SysadminPager` component, while `WarningLogEvents` are to be sent to the `SysadminEmailer` component. The `LogEvent` is sent to the first destination matching the given class, as either an exact class match, or a subclass. So any `ErrorLogEvent` or subclass of `ErrorLogEvent` is sent to `SysadminPager`.

defaultDestinations

The destinations of any log events that do not match any of the types in `logEvents`. For example:

```
defaultDestinations=\
        FileLogger
```

This specifies that any `LogEvents` that are not errors or warnings are sent to the `FileLogger` component. You can specify multiple destinations; in that case, the event is sent to all specified destinations in order. If you do not specify the `logEvents` or `logDestinations` properties, events are always be distributed to the `defaultDestinations`. This is a useful way for you to send a single `LogEvent` to multiple destinations—for example, email and a file.

However, unlike the `defaultDestinations` property, the `logDestinations` property cannot be used to send one type of `LogEvent` to two different destinations. If you set these properties:

```
logEvents=\
                   InfoLogEvent,\
                   InfoLogEvent
logDestinations=\
                  /logging/infoListener1,\
                  /logging/infoListener2
```

then no `InfoLogEvents` reach `infoListener2`; all are sent to infoListener1. You can send a single `LogEvent` to multiple destinations either by using the `defaultDestinations` property, or by using two `DispatchLoggers` in sequence. The first `DispatchLogger` might have these properties:

```
logEvents=\
                   InfoLogEvent,\
                   FooLogEvent
logDestinations=\
                  /logging/infoDispatchLogger2,\
                  /logging/fooListener
```

while the second, the `/logging/infoDispatchLogger2` named in the logDestinations property, receives only `InfoLogEvents` and can use the `defaultDestinations` property to route the `InfoLogEvents` to both `/logging/infoListener1` and `/logging/infoListener2`:

```
defaultDestinations=\
                  /logging/infoListener1,\
                  /logging/infoListener2
```

LogListenerQueue

A `LogListenerQueue` is a subclass of `EventQueue` that buffers calls to `LogListeners` (see the Queues section). This allows a high-throughput process, such as HTTP request handling, to be decoupled from the slower logging processes such as writing to files or sending email. The `logListeners` property specifies the `LogListener` where log messages are to be sent after being run through the queue.

All log messages are typically sent through a `LogListenerQueue` before being run through the rest of the logging system.

Logging Configuration

In the standard ATG configuration, all components are directed to send their logging events to a single `LogQueue` component. This is specified in the `/GLOBAL.properties` file, which you can view in the Configuration tab of the Component Editor of any Nucleus component:

```
logListeners=\
     atg/dynamo/service/logging/LogQueue,\
     atg/dynamo/service/logging/ScreenLog
```

All components also direct their output to the `ScreenLog` component, causing all messages to appear on the console. This is useful for debugging at development time, and should be removed at production time.

The `LogQueue` component queues log events, preventing the handling of those events from impacting the throughput of the rest of the system. The `LogQueue` feeds its output to a `LogDispatch` component:

```
logListeners=LogDispatch
```

The `LogDispatch` separates the error, warning, info, and debug log events and directs them to separate components. Any events that do not match the above classes are sent to the info logs:

```
logEvents=\
     atg.nucleus.logging.InfoLogEvent,\
     atg.nucleus.logging.WarningLogEvent,\
     atg.nucleus.logging.ErrorLogEvent,\
     atg.nucleus.logging.DebugLogEvent,\
     atg.nucleus.logging.LogEvent
logDestinations=\
     InfoLog,\
     WarningLog,\
     ErrorLog,\
     DebugLog,\
     InfoLog
```

Each of the destination logs (`InfoLog`, `WarningLog`, `ErrorLog`, `DebugLog`) is a `RotatingFileLogger`. Each log is stored in a separate file in the `./logs` directory, and is archived at 1am every Sunday into the `./logs/archives` directory:

```
$class=atg.nucleus.logging.RotatingFileLogger
logFilePath=./logs
logFileName=info.log
```

```
scheduler=../Scheduler
schedule=calendar * . 1 1 0
logArchivePath=./logs/archives
maximumArchiveCount=20
archiveCompressed=true
```

As you can see, the entire ATG logging system is completely defined using standard Nucleus components and configuration files. This means that you can change the logging configurations and procedures by changing configuration files, usually without writing any Java code.

Designing Logging Systems

The logging model used by ATG provides a flexible mechanism for setting up complex application logging rules. With a combination of filters and sinks, you can design a logging configuration that handles all requirements.

The key to designing logging systems is to model your logging rules in terms of the logging filters and sinks provided with ATG (or with new filters and sinks that you write yourself).

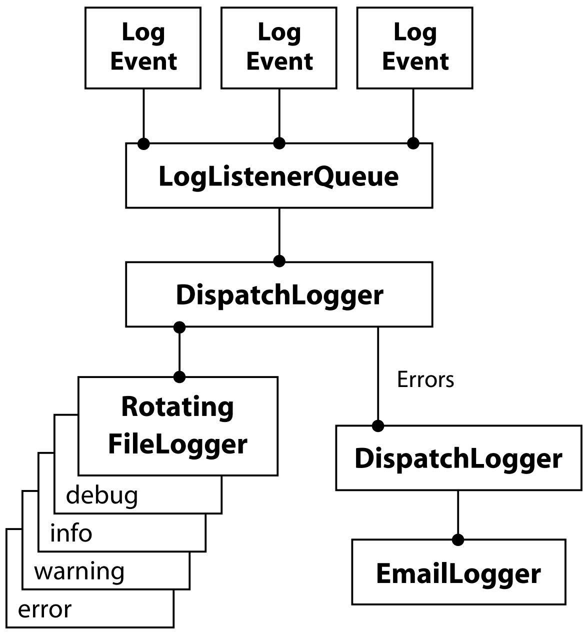
For example, if you want to monitor a particular component so errors are sent as email, but all messages, including errors, are sent to a single file, you need the following:

- `LogListenerQueue`, to ensure the component is not hampered by the logging processes
- `DispatchLogger` that:
    - receives events from the `LogListenerQueue`
    - defines only the `logDestinations` property
    - distributes all events to two listeners
- Another `DispatchLogger` that feeds from the first `DispatchLogger` but only recognizes `ErrorLogEvents`
- `EmailLogger` to receive events from the second `DispatchLogger` and to send those events as email
- `RotatingFileLogger` to receive all events from the first `DispatchLogger` and write those events to a file

Finally, the log source component must specify the `LogListenerQueue` as one of its `logListeners`.

Here is an example of what a logging system might look like:

Logging System Example



Logging for Non-GenericService Components

Using Nucleus logging for non-`GenericService` objects can present two relatively common problems:

- Your Nucleus-instantiated component cannot extend `GenericService`
- You use classes that are not created by Nucleus, but for which you want to do Nucleus-style logging

The `LoggingPropertied` interface and the `ClassLoggingFactory` can be used to solve these problems.

Note: You should use `GenericService` Nucleus components as much as possible, because that interface is simple to use and well supported. If you cannot use `GenericService` components, use a Nucleus-instantiated component that implements the `LoggingPropertied` interface. This retains the power and flexibility of a Nucleus configured component, even if your base class did not implement `ApplicationLogging`. Use the `ClassLoggingFactory` if neither of these approaches is possible.

Logging with Nucleus-instantiated Non-GenericService

If your Nucleus component cannot extend `GenericService`—for example, because it already extends some other class—you can use the `LoggingPropertied` interface.

The `LoggingPropertied` interface consists of a single method:

```
public ApplicationLoggingSender getLogging();
```

This method returns the application logging instance your component uses for its logging. Using this interface means that you do not have to implement the entire `ApplicationLogging` interface on your subclass, or use another component's `ApplicationLogging` instance, which means that your own component's name does not appear in the log files.

A `LoggingPropertied` component typically implements `LoggingPropertied` and `ServiceListener` (to get `startService` calls so that the component name can be set on the logging instance), and includes the following code:

```
ApplicationLoggingImpl mLogging = new ApplicationLoggingImpl(
  this.getClass().getName());

public ApplicationLoggingSender getLogging() {
  return mLogging;
}

public void startService (ServiceEvent pEvent) throws ServiceException {
  mLogging.initializeFromServiceEvent(pEvent);
}

public void stopService() {
}
```

Then, when your component needs to log, it can use code such as the following:

```
if (getLogging().isLoggingDebug()) {
  getLogging().logDebug("Debugging!");
}
```

Nucleus is now aware of the `LoggingPropertied` interface, and displays properties for the `ApplicationLogging` instance. The administrative UI also lets you access the `ApplicationLogging` properties, including changing them at runtime.

Nucleus understands simple dot notation for nested property names, with some limitations. So the Nucleus properties file for your component implementing `ApplicationLogging` can contain the following to turn on logging debug when your application starts up:

```
logging.loggingDebug=true
```

Note: The logging property object must exist before the `startService`, because all property settings are applied before `startService` is invoked.

Logging with Non-Nucleus-instantiated Classes

In some cases, you need logging for classes that are not instantiated by Nucleus, such as a servlet class created by the web container, a static utility class that has no instances, or a light-weight class that has many instances, which do not each need their own `ApplicationLogging` instances. In this case, you can use the `ClassLoggingFactory` to create a logger. An `ApplicationLogging` logger created by `ClassLoggingFactory` is shared by all instances of that class, and appears in Nucleus under the `/atg/dynamo/service/logging/ClassLoggingFactory` named for the class that created it. So, for example, given the following class:

```
package my.lovely;
import atg.nucleus.logging.*;

public class NonNucleus {
  ApplicationLogging mLogging =
    ClassLoggingFactory.getFactory().getLoggerForClass(NonNucleus.class);
}
```

an `ApplicationLogging` instance appears in Nucleus as follows:

`/atg/dynamo/service/logging/ClassLoggingFactory/my.lovely.NonNucleus`

This lets you turn logging on and off at runtime. Note that the component appearing in Nucleus is just the `ApplicationLogging` logger, and not the class (or instance) itself.

You can turn on logging for `ClassLoggingFactory` client classes when your application starts up by creating a properties file at the appropriate location. Given the previous example, you can create a properties file `<DYNAMO_HOME>/atg/dynamo/service/logging/ClassLoggingFactory/my.lovely.NonNucleus.properties` with the following content:

```
$class=atg.nucleus.logging.ApplicationLoggingImpl
loggingDebug=true
```

This turns on `loggingDebug` for the example `NonNucleus` class when the application starts up.

Introduction to Data Collection

ATG's Data Collection facility provides a flexible way to collect information. Like Logging, Data Collection is based on the Java event model. But Data Collection lets you record data contained in any JavaBean (not just subclasses of `LogEvent`). Like the Logging system, the Data Collection system includes components that act as sources, events, and listeners. In addition, Data Collection summarizer components perform in-memory summarization. The following topics describe the components of the ATG Data Collection system:

- Data Collection Sources and Events
- Data Listeners
- Formatting File Loggers
- Database Loggers
- Data Collector Queues
- Summarizers

Data Collection Sources and Events

The most important aspect of data collection is its use of arbitrary JavaBeans as data points. Your components do not have to subclass `LogEvent` (or anything else) to use data collection. You can even use data collection without writing any Java.

ATG includes the source code for a class named `atg.service.datacollection.DataSource`, located at `<ATG10dir>/DAS/src/Java/atg/service/datacollection/DataSource.java`. This class serves as a sample design pattern you can use when creating components that serve as sources of data for the data collection system.

There are three ways you can make a component be a data source:

- Subclass `atg.service.datacollection.DataSource`
- Encapsulate a Data Source (declare an object that extends `DataSource` in your component and pass through the `DataListener` method calls)
- Implement the `DataListener` method calls in your component.

To be a source of data items, a class needs to implement the Java event design pattern for event sources. All data collection listeners implement the `atg.service.datacollection.DataListener` interface. So to create a source of data events, implement the following two methods in your data source component:

```
public void addDataListener(DataListener listener);
public void removeDataListener(DataListener listener);
```

Then to send a data item to your listeners, call the method:

```
sendDataItem(Object dataItem)
```

This method sends the data item as a data collection event to each data listener in your list. The data listeners can then examine and extract information from the data item's properties. This is based on the Java event model, so there is no interface that you need to implement.

Data Listeners

Data collection sources each have one or more data listeners (specified by the data source's `dataListeners` property). Depending on how you design your data collection system, a data listener might log the data event's properties to a file, log them to a data base, summarize a set of data events, or queue data events before passing them to another data listener. ATG data listeners implement the `atg.service.datacollection.DataCollector` interface. The following topics describe different sorts of data listeners:

- Formatting File Loggers
- Database Loggers
- Summarizers
- Data Collector Queues

You can also create your own data listener by writing a class that implements `atg.service.datacollection.DataCollector`.

Compatibility with Logging

For backward compatibility and the convenience of those who do not want to implement the data source design pattern, data listeners implement the `atg.nucleus.logging.LogListener` interface in addition to the `atg.service.datacollection.DataCollector` interface. So you can send data items (that are actually `LogEvents`) to the data collection facility from any `GenericService`, simply by configuring one or more Data Listeners as one of the `logListeners` of your `GenericService`. Also, by extending `LogEvent`, your data item can use all features of data collection.

Formatting File Loggers

You can use a formatting file logger (`atg.service.datacollection.FormattingFileLogger`) to write data from a data collection event to a flat file in a format you specify. A formatting logger lets you specify:

- Which properties of your data item should be logged
- The order the properties should be logged
- Arbitrary constant strings to log
- Format control strings (for date properties)
- Field delimiters (written after each field except the last one)
- Line terminator (written after the last field in the log entry)

`FormattingFileLogger` components are also rotating loggers: you can set a schedule where the log file is closed and a new log is opened with a different name.

In production systems, you should use `DataListenerQueues` to feed data to your formatting file logger. This allows unlogged data to queue up without being lost and without affecting the performance of data sources. See Data Collector Queues.

Configuring Fields

Having control over the order of fields lets you configure a formatting logger to write files suitable for post-processing or bulk data loading into an SQL database. You can implement more advanced formatting, such as changing the delimiter or terminator. You can also create a logger that emits data formatted in XML.

The following properties of the `FormattingFileLogger` component control the contents of log fields and fields:

- `formatFields`
- `fieldDelimiter`
- `lineTerminator`

formatFields

This property is an ordered list of the properties to log, taken from the incoming data item. Each item in the list represents a single field in a line of formatted text in the log file. Separate each item with a comma. For example:

```
formatFields=id,requestId,contentId
```

Remember that Java properties files treat white space as part of the property value. Set the `formatFields` property like this:

```
formatFields=name,address.number,address.streetName
```

and not like this, with white space between the comma separator and the field name:

```
formatFields=name, address.number, address.streetName
```

Note: As shown in the example above, you can log subproperties, such as `address.streetName`.

Formatting Individual Fields (Dates)

By default, each property of a data item is converted to a string by calling the standard `toString()` method. This is usually what is expected and desired. However, sometimes it is not the right thing. For instance, `Date` objects often require special formatting.

To handle this, format fields can have format strings. To use a format string, specify the property name, followed by a colon and the format string. Here is an example that shows how the `RequestLogger` component (`/atg/dynamo/service/logging/RequestLogger`) logs the `currentDate` property:

```
currentDate:d/MMM/yyyy:H:mm:ss
```

If a format string is present, the field is formatted using that string and the JSDK standard `java.text` formatting facility. Currently, this formatting is only supported for `java.util.Date` objects. If you have a property to format in a certain way, you can make that property be a class and override its `toString()` method.

Note, however, that formatting a date can be an expensive operation. If logging performance is an issue, consider storing date or timestamp information as a long primitive.

For `Date` objects, possible formats are those supported by the `java.text.SimpleDateFormat` of the JSDK you are using. See the documentation for your JSDK at, for example, `<JSDK dir>/jdoc/java/text/SimpleDateFormat.html`. The formatting loggers use this date format by default:

```
yyyy-MM-dd HH:mm:ss
```

fieldDelimiter

By default, a formatting logger delimits fields with tabs. You can specify a different separator with the `fieldDelimiter` property. For example, to use the colon (`:`) as a delimiter, you can set the following property:

```
fieldDelimiter=:
```

You might want to have a different delimiter for each field. You can set the `fieldDelimiter` property to null and set the delimiter for each field in the value of the `formatFields` property, using single quotes to add labels to each line of the log, as in this example:

```
formatFields='RemoteAddr='request.remoteAddr,' - - -\
    [Date=',currentDate:d/MMM/yyyy:H:mm:ss,'] '
fieldDelimiter=
```

This produces output that looks like the following:

```
RemoteAddr=remoteAddr1 - - -[Date=12Jul1999:22:04:47]
RemoteAddr=remoteAddr2 - - -[Date=13Jul1999:02:16:31]
```

From the example, you can see that strings enclosed in single quotes are written to the log file `as-is`. This lets you craft almost any kind of flat file format you like without writing a single line of Java.

lineTerminator

By default, a formatting logger terminates lines in the log file with newlines. This behavior is configurable with the `lineTerminator` property:

```
lineTerminator=\n
```

Configuring Log File Names

The following properties of the `FormattingFileLogger` component enable you to control how you name the log files:

Property	Type	Function
`logFileDir`	string	Specifies the directory where log files are written, relative to `<ATG10dir>/home/` Example: `logs`
`logFileName`	string	Specifies the first element of the log's file name. Example: `logFileName=userevents_`
`logFileExtension`	string	Specifies the file extension (such as `data`, `log`) that is attached to each log file. Example: `logFileExtension=data`
`timestampLogFileName`	Boolean	If set to `true`, a timestamp is included in each log file name.
`timestampDateFormat`	string	Specifies the date format to use in file name timestamps. Example: `MM-dd-yyyy_HH-mm-ss-SS`

For example, the following property settings yields log file names like `userevents_02-09-2001_18-36-03-55.data`:

```
logFileName=userevents_
logFileExtension=data
timestampLogFileName=true
timestampDateFormat=MM-dd-yyyy_HH-mm-ss-SS
```

Using timestamps in your log file names ensures that log files have unique names and are preserved on application restarts.

In the `timestampDateFormat`, avoid using separator characters that result in invalid file names in your operating system. For example, if you set:

```
timestampDateFormat=yyyy-MM-dd_HH:mm:ss
```

the resulting log file name is like this:

```
userevents_02-09-2001_18:36:03.data
```

Because the colon (`:`) is not a valid character in Windows file names, this yields errors on a Windows platform.

The `schedule` and `scheduler` properties of the `FormattingFileLogger` determine when a log file is closed and a new log created with a new name.

Formatting Logger Example: the RequestLogger

ATG includes a formatting file logger component that can be set to log page requests from users, `/atg/dynamo/service/logging/RequestLogger`. The RequestLogger logs properties of the request and response objects for each user request, which it obtains from the ATGServlet. To use the RequestLogger, set the `dataListeners` property of `/atg/dynamo/servlet/pipeline/DynamoServlet` as follows:

```
dataListeners=/atg/dynamo/service/logging/RequestLogger
```

You can set the `dataListeners` property using the Event tab in the Component Editor:

1. Select the `data` event set.
2. In the Registered Event Listeners column, click **...** .
3. Click Insert Before or Insert After.
4. Select `/atg/dynamo/service/logging/RequestLogger` as a registered event listener.

The `RequestLogger` has the following properties file:

```
$class=atg.service.datacollection.FormattingFileLogger

# directory and file name
logFileDir=logs
logFileName=request.log

formatFields=request.remoteAddr,' - - [',currentDate:d/MMM/yyyy:H:mm:ss,']
  "',request.method,' ',request.requestURI,' ',request.protocol,'"
  ',response.status,' -'

# the default field delimiter is a tab char ('\t')
# in this example we set it to null as our formatFields
# above include custom field delimiters
fieldDelimiter=
```

The `$class` line loads the `FormattingFileLogger` class, which formats data items and logs them to a file. The `logFileDir` and `logFileName` properties control the file you log to.

The key property to look at here is `formatFields`. This is an ordered list of the properties to log. In this example, the `RequestLogger` is expecting data items that have properties named `request.remoteAddr`, `request.method`, `request.requestURI`, `request.protocol`, and `response.status.` The `RequestLogger` gets this data from the request and response objects. By default, fields are delimited by tabs and terminated by newlines. However, the `formatFields` property in this example provides custom field delimiters. One line in the log is written for each data item that the logger receives. To log just the `requestURI`, change the `formatFields` property to:

```
formatFields=request.requestURI
```

This writes to the `logs/request.log` file, entering the `request.requestURI` of each data item followed by a newline.

Database Loggers

Another type of data listener is an SQL table logger that writes data directly to a relational database. Use a component of class `atg.service.datacollection.JTSQLTableLogger`. SQL table loggers are configured with properties that link them to the JDBC data source, database table name, and database column in which to log the data items.

Each data item an SQL table logger receives is written to the named table using an appropriate INSERT statement. For this to work, the table must exist, and the `dataSource` must have INSERT permission on the table. The SQL table logger attempts to reconnect to the database as needed. It also gives you control over the size of the database transactions it uses when flushing data.

In production systems, you should use `DataListenerQueues` to feed data to your SQL table logger. This allows unlogged data to queue up without being lost and without affecting the performance of data sources. See Data Collector Queues.

The following table describes the properties you use to configure an SQL table logger:

Property	**Description**
`dataSource`	A JTA data source that the SQL table logger uses to connect to the database. See the *ATG Installation and Configuration Guide*.
`tableName`	The name of the SQL table that holds the logged data.

Property	Description

`SQLColumnMappings`	A mapping of the property name of the data collection event to the column name in the database table specified by the `tableName` property, in the form
	
	`propertyName``:`*`columnName`*
	
	where *`propertyName`* is the name of a property to be logged, and *`columnName`* is the name of the column within the database table that holds the value of the property. For example:
	
	`username:user_name,firstname:first,lastname:last`

`dataItemThreshold`	Flush data after receiving this number of data items. See Data Flushing.
`scheduler`	Scheduler component to use with a data flushing schedule. See Data Flushing.
`schedule`	Schedule to use in flushing data. See Data Flushing.
`transactionManager`	The transaction manager used by the SQL table logger. See the Transaction Management chapter.
`transactionSize`	The maximum number of rows to be inserted in each database transaction. See Configuring Transaction Size.
`enableTruncation`	With truncation enabled (the default), the SQL table logger determines the SQL column size when the application starts up. String items that are longer than the available SQL column size are truncated before being logged. If truncation is disabled, an attempt to log a string that is too large for the SQL column results in the insertion failing and the data being lost. Truncation of number and time entries is handled, if at all, by your JDBC driver.
`bufferSize`	The maximum number of entries to accumulate before flushing the data to the database. See Configuring Transaction Size.
`blocking`	Should the data source be blocked if the buffer is full? See Using Blocking with a Data Collector Queue.

The following properties can be helpful in cases where the user does not own the table where log entries are to be made:

Property	Description
`tablePrefix`	If the user does not own the table where log entries are to be made, you can use this property to construct a qualified table name. This property is not used during the initial metadata query, but if present is prepended to the table name when inserts or updates are made.
`metaDataSchemaPattern`	A String representing a schema name pattern. If the user does not own the table where log entries are to be made, this property can be used once during initialization of the logger in a call to determine the column types. Make sure you set this property using the exact case (upper, lower, mixed) that your database uses to store object identifiers. For example, Oracle stores its identifiers in uppercase. In this case, use `metaDataSchemaPattern=DYNAMO` instead of `metaDataSchemaPattern=Dynamo`. See the Javadoc for `java.sql.DatabaseMetaData.getColumns()` for more information.
`metaDataCatalogName`	A String representing a catalog name. If the user does not own the table where log entries are to be made, this property can be used once during initialization of the logger in a call to determine the column types. See the Javadoc for `java.sql.DatabaseMetaData.getColumns()` for more information.

For instance, in a case where a table named `ztab` is owned by `admin` and the user is `dynamo`, here is how these properties can be used with Oracle and Microsoft SQL Server DBMS:

Property	Oracle	Microsoft SQL Server
`tableName`	`ztab`	`ztab`
`metaDataSchemaPattern`	`admin`	If the table owner is not the database owner, use the table owner name (here, `admin`). If the table owner is the database owner, use dbo.
`metaDataCatalogName`	Ignored, leave blank.	If user shares a database with the table owner, you might be able to leave this blank; otherwise use the table owner name (here, `admin`).
`tablePrefix`	Leave blank if user has an Oracle synonym, otherwise use `admin`.	`admin`

Data Flushing

You can configure an SQL table logger to flush data to the database using either a schedule or a data threshold. A schedule flushes data to the database based on a time schedule, while a data threshold flushes data to the database upon receiving a specified number of events. It is strongly recommended that you use a schedule rather than a data threshold.

To enable in-line flushing using a data threshold, set the value of the `dataItemThreshold` property to whatever threshold you want. A threshold of 10 means that the SQL table logger flushes its data after receiving 10 events.

For best performance in production systems, use a schedule to control your flush operations, and not a `dataItemThreshold`. The schedule frequency should be tuned based on the rate at which data is being fed to the SQL table logger. Configure the schedule with the `scheduler` and `schedule` properties. For example:

```
scheduler=/atg/dynamo/service/Scheduler
schedule=every 3 minutes
```

Configuring Transaction Size

The `transactionSize` property controls the way the SQL table logger batches its database operations. The default value, 0 is in effect infinity; it means that each flush occurs in a single database transaction, no matter how many rows of data are inserted. This might be undesirable if the flush has a lot of data to store. By setting this property, you can tell the SQL table logger to batch its INSERT operations into chunks that are as big as the `transactionSize` setting. So, for example if you set the `transactionSize` property to 20, when the SQL table logger flushes its data, it commits after every 20 rows. The SQL table logger always commits after the last row regardless of the `transactionSize` setting.

The best value for this property depends on the size and number of data items you are logging. Because a data item can be any object of any size you might have to experiment with this property to see what works best for your site. A good starting value might be 100.

Configuring the Buffer Size

AN SQL table logger uses an internal buffer to hold data before flushing it to the database. When this buffer is full, the SQL table logger flushes data to the database, whether or not the scheduled time has arrived. You should not need to change this parameter. However, the maximum amount of data that can be flushed at once is equal to the size of the buffer. So, if you have an SQL table logger that is expected to store a lot of data with each flush, you should set the `bufferSize` property accordingly. When the SQL table logger flushes data to the database, the maximum transaction size is the lesser of `bufferSize` and `transactionSize`.

Using Blocking with a Data Collector Queue

The `blocking` property controls the behavior of the SQL table logger when its internal buffer is full. The default value is `true`, which means that the data source feeding the SQL table logger blocks until there is room in the buffer. Use this setting in conjunction with a `DataListenerQueue`. See Data Collector

Queues in this chapter and Queues in the Core ATG Services chapter for more information about using queue components.

If you are not using a `DataCollectorQueue` and do not want your data source to block, set `blocking` to `false`. This is not recommended, however, as it causes new data to be lost when the internal buffer is full.

SQL Data-types

AN SQL table logger uses the `setObject()` method to convert the following Java types to their default SQL data-types as specified by JDBC:

- `String`
- `Number`
- `java.sql.Timestamp`
- `java.sql.Date`

A `java.util.Date` property value is first converted to a `java.sql.Timestamp` and then `setObject()` is called on it. Properties of other data-types are logged as strings, using the `toString()` method on the property value.

Data Collector Queues

Just as in the logging system, a well-designed data collection system usually interposes a data collector queue between the source of the data collection events and the logger or other data listener that acts on the event. A `DataCollectorQueue` (`atg.service.datacollection.DataCollectorQueue`) is a subclass of `EventQueue` that buffers calls to data listeners (see the Queues section of the Core ATG Services chapter). Using a queue allows a high-throughput process, such as HTTP request handling, to be decoupled from the slower logging processes, such as writing to files or database tables. A `DataCollectorQueue` is a data listener that passes data collection events on to other data listeners. The `dataListeners` property of a `DataCollectorQueue` specifies the data listeners where data collection events are to be sent after being run through the queue.

Summarizers

When collecting volumes of data, handling the information sometimes becomes a problem. Applications often fire off events to be logged and analyzed later, such as HTTP requests logged by a web server. Often you summarize the data, then archive or delete the detailed events. In some applications the detailed events are not even pertinent; it is only the summary that is required. Logging huge volumes of data to files or to an SQL database just to summarize it carries an unnecessary performance and administrative overhead. You can handle situations of this sort using a data collection summarizer (`atg.service.datacollection.GenericSummarizer`).

A summarizer is a data listener that listens for data items and summarizes them in memory. The summarizer summarizes and groups beans by one or more properties. Summarized data can then be logged to flat files or SQL tables. At intervals, the summarizer flushes summarized data to its `dataListeners` (typically, a file logger or an SQL logger).

Summarizer Method and Timestamps

The method used by summarizers is a simple counting mechanism coupled with timestamps. The summarizer checks whether each data item it receives matches any item it has seen before. If the data item matches an existing item, the summary count for that item is incremented. If the data item does not match, a new slot for the item is created and its count is set to 1.

Matching and the groupBy Property

The summarizer uses the `groupBy` property to determine whether or not a data item matches any of the data items in its list. The `groupBy` property lists the data item properties that are considered in the matching process. To compare two data items, the summarizer compares each of the values in the `groupBy` property. If each of the values match, the data items are said to match for this summarizer. The net effect is analogous to the use of the SQL `GROUP BY` clause used in many relational database reports.

SummaryItems

Summarized data are grouped together in instances of the Java class `atg.service.datacollection.SummaryItem`. A `SummaryItem` contains a reference to the data item being summarized as well as the following summary information:

- `summaryCount`: count of data items received
- `summaryFromTime`: time first data item was received
- `summaryToTime`: time last data item was received

Thus, the summarizer's summarized data is a list of `SummaryItems`. Each time new data is received, the matching `SummaryItem` is updated or a new `SummaryItem` is added to the list.

Summarizer Example

For example, you might have an object that represents the delivery of an advertisement called `AdEvent`. Let's assume an `AdEvent` has three relevant properties: `accountName`, `campaignName`, and `adName`. In order to summarize by all three properties, set your `groupBy` property as follows:

```
groupBy=accountName,campaignName,adName
```

This causes the summarizer to only consider two `AdEvents` as matching if all three of the properties are the same. To summarize by campaigns instead (regardless of accounts or ads), set the `groupBy` property to:

```
groupBy=campaignName
```

This causes the summarizer to consider two `AdEvents` as matching if their `campaignNames` are equal. You can have more than one summarizer listening for the same data items. So if you want to combine the last two summarization examples, configure two summarizers and have them both listen for the same data items.

Flushing Data from the Summarizer

The summarizer keeps its `SummaryItem` list until it is time to flush it to its `dataListeners`. The summarizer flushes on the earlier of:

- when its scheduled flush time comes
- when it receives `dataItemThreshold` data items

Both the scheduled time and the `dataItemThreshold` are configurable properties of the summarizer. By using these properties to control the flush interval, you can balance the performance of the summarizer and the amount of summarized data that would be lost in a system crash.

When the summarizer flushes its data, it sends `SummaryItems` to the data listeners specified by the summarizer's `dataListeners` property. These data listeners can be queues, and are usually one of the loggers that come with the data collection package. The summarizer's `dataListeners` consists of a list of summary loggers, `FormattingSummaryLogger` or `JTSQLTableSummaryLogger`.

Logging SummaryItems

`SummaryItems` are JavaBeans with properties; thus, they can be logged. When logging a `SummaryItem`, it is useful to log properties of both the `SummaryItem` and the data item being summarized. For this reason, the data collection package contains summary loggers that extend the logging syntax to support this.

The Formatting File Logger has a corresponding formatting summary logger and the SQL table logger has a corresponding SQL table summary logger. The summary loggers are just like the regular loggers, except that they add the ability to refer to summary variables as well as data item properties. See the Formatting File Loggers and Database Loggers sections in this chapter, and the Summary Variables topic in this section.

Continuing the example, you might have a summarizer listening for `AdEvents` with the following `groupBy`:

```
groupBy=accountName,campaignName,adName
```

To log the summarizer's `SummaryItems` to a log file, configure a summary logger as a `dataListener` of the summarizer, with properties as follows:

```
$class=atg.service.datacollection.FormattingSummaryFileLogger
logFileDir=logs
logFileName=foo.txt
# the things that will be logged
formatFields=accountName,campaignName,AdName,%SummaryCount
fieldDelimiter=:
```

The only thing new here is `%SummaryCount` value in the `formatFields` property. This refers to the `SummaryCount` summary variable, while the other properties refer to properties of the data item being summarized. In the example, the logger writes the `accountName`, `campaignName`, `adName`, and the count of how many `AdEvents` were received. The summarizer might receive the following events during a single flush interval:

accountName	**campaignName**	**adName**
OmniCorp	new image	`small_banner`
OmniCorp	new image	`small_banner`
OmniCorp	new image	`small_banner`
OmniCorp	new image	`large_banner`
OmniCorp	new image	`large_banner`
MegaSomething	traditional	`small_banner`
MegaSomething	new image	`small_banner`

The summarizer generates `SummaryItems`, sends them to the summary logger, which in turn writes the following to the log file:

```
OmniCorp:new image:small_banner:3
OmniCorp:new image:large_banner:2
MegaSomething:traditional:small_banner:1
MegaSomething:new image:small_banner:1
```

Summary Variables

The following table outlines the available summary variables.

Summary Variables	**Description**
`%SummaryCount`	Number of data items received
`%SummaryFromTime`	Time (`java.sql.Timestamp`) the first data item was received
`%SummaryFromTimeMillis`	Time the first data item was received (Java `long` value in milliseconds since Jan 1, 1970)
`%SummaryToTime`	Time (`java.sql.Timestamp)` the last data item
`%SummaryToTimeMillis`	Time the last data item was received (Java `long` value in milliseconds since Jan 1, 1970)

Summary Variables	Description
`%CurrentTime`	Time (`java.sql.Timestamp`) the `SummaryItem` was flushed
`%CurrentTimeMillis`	Time the `SummaryItem` was flushed (Java `long` value in milliseconds since Jan 1, 1970)

DBWriteMethod in an SQL Table Summary Logger

The `JTSQLTableSummaryLogger` component extends `JTSQLTableLogger`. It adds a new property, `DBWriteMethod`, which determines how summary log information is written to the SQL database table. The `DBWriteMethod` property is a string whose valid values are `insert` and `update`.

If you use the `update` write method (`DBWriteMethod=update`), you also need to configure the `SQLColumnMappings` property of the `JTSQLTableSummaryLogger`. If a property should be incremented when the table is updated, use the `:add` string in the `SQLColumnMappings` property. If a property should be set to the new value when the table is updated, use the `:set` string in the `SQLColumnMappings` property. For example:

```
SQLColumnMappings=cartItems:itemsPurchased:add
```

You can define more than one value in `SQLColumnMappings`, as in this example:

```
SQLColumnMappings=company:comp_id:set,dept:dept_id:set,hits:num_hits:add
```

Summarizer Flush Methods

The `GenericSummarizer` class extends `DataCollectorService`, which provides the summarizer with several methods in controlling how summary data gets flushed from the summarizer. When the summarizer component or the ATG server is shut down, you want to make sure that any data in the summarizer is flushed before shutdown. To accomplish this, the summarizer's `doStopService()` method calls its `flush()` method.

You might also want the summarizer to instruct its data listeners to flush the data as well. To provide for that case, the summarizer has a `propagateFlush` property. If `propagateFlush` is set to true, the `flush()` method also causes the data listeners to flush.

Special `localFlush()` and `flushListeners()` methods are available to selectively flush only the summarizer or its listeners, respectively.

12 ATG Message System

The Java Message Service (JMS) defines a standard way for different elements of a J2EE application to communicate with each other. With JMS, components do not access each other directly. Instead, a component posts a message to a message broker, which then distributes the message to other components. In general, the posting and delivery actions occur in separate transactions, and might even occur in different processes, machines, or sites. This mechanism decouples the actions of the sending and receiving components to so that the sender can continue with its work without having to wait for the receiver to process the message. This decoupling is often called asynchronous processing.

The JMS API defines the interfaces for sending and receiving messages. It also defines the semantics for message delivery and acknowledgement, and how message delivery should behave in a transactional environment. The API is intended to be flexible enough to allow applications to work with existing enterprise messaging products.

In this chapter

This chapter discusses JMS and the ATG Message System, which is a set of tools that ATG provides for working with JMS. The chapter includes the following sections:

Overview of JMS

As discussed above, a critical architectural feature of JMS is that it decouples the objects that send messages from those that receive messages. This architecture contrasts, for example, with the JavaBean event model, where an event listener must register with the object that fires the event. In the JMS messaging model, objects that send messages (message producers) and objects that receive messages (message consumers) do not need to be aware of each other, because a producer does not send messages directly to a consumer.

Instead, a JMS message producer sends a message to a destination, where it is retrieved by one or more message consumers. JMS defines two types of destinations, corresponding to two basic forms of messaging:

- **topic**: A destination used in publish/subscribe messaging. If a topic has several subscribed listeners, each message published to that topic is delivered to all listeners.
- **Queue**: A destination used for point-to-point messaging. If a queue has several subscribed receivers, each message is delivered to only one of the receivers. A different receiver might be chosen for each message, possibly depending on some load balancing mechanism.

JMS Message Producers and Consumers

The JMS API defines a set of interfaces for creating message producers and consumers. There are separate interfaces for producers and consumers, and for objects that communicate with topics and queues. These interfaces are all part of the `javax.jms` package:

- `QueueSender`
- `QueueReceiver`
- `TopicPublisher`
- `TopicSubscriber`

In addition to implementing one of these interfaces, the producer or consumer must do a considerable amount of setup in order to send or receive messages: obtain a ConnectionFactory, find destinations, obtain a JMS Connection, create a JMS Session, and so on. One of the main advantages of using ATG's Patch Bay system is that it handles the bulk of these setup tasks, so your code does not have to. See the Overview of Patch Bay section for more information.

JMS Destinations

As mentioned above, JMS defines two types of destinations, topics and queues. Most JMS providers support both topics and queues, and an application can make use of both. ATG applications typically use topics, as they offer the most flexibility for expansion. However, a messaging application might use queues for certain purposes, such as load balancing.

The use of destinations provides much of the flexibility in JMS. If a new application needs to send messages to or receive messages from an existing application, it can publish or subscribe to the destinations used by that application. The new application does not need to be aware of the message producers and consumers in the original application, just the destinations. This means that message producers and consumers can be added to or removed from one application without affecting other applications, as long as the destinations remain the same.

Each destination is maintained by a single JMS provider, which typically maintains many destinations. The creation and management of destinations within a JMS provider is usually an administrative or configuration operation. If a message is sent to a destination, that destination's JMS provider is responsible for receiving the message and passing it on to subscribers waiting for messages from that destination. Different providers might use different mechanisms to accomplish this. For example, ATG's SQL JMS uses an SQL database to store and deliver messages, for applications that require the messaging system to be highly reliable. Other JMS providers might use file- or memory-based storage.

Message Persistence

Queue destinations typically are persistent. If a message is sent to a queue but no receiver is online, the message is kept in the queue, waiting for a receiver to connect and start reading from the queue. After a message is delivered to a single receiver, it is removed from the queue.

Topics, however, are non-persistent by default. If a message is sent to a topic, it is delivered to all subscribers to that topic that are currently online, and then removed. Any subscriber that is offline does not receive the message. If no subscribers are currently online, the message is simply removed from the topic without being delivered anywhere.

Some applications require the flexibility of a topic, but also the persistence offered by a queue. For example, suppose an application requires a message to be delivered to several subscribers, but it is not acceptable for a subscriber to miss any of the messages if it goes offline. Sending email to a mailing list demonstrates this paradigm (in a non-JMS environment), where a single message is distributed to many readers, and queued up for each reader to be delivered when the reader comes online.

JMS addresses this need through the use of durable subscriptions. A message consumer that has a durable subscription to a topic can go offline, then reconnect later and pick up any messages that were sent to the topic in its absence. Durable versus non-durable is a property of each individual subscriber, not of the topic as a whole. A topic can have a mix of subscribers, some durable and some non-durable.

Durable and non-durable subscribers are created through the JMS API. Creating a durable subscriber requires specifying a name that the topic uses to identify the subscriber. Each durable subscriber to a topic must have a name that is unique for that topic. If a subscriber disconnects, the JMS provider holds any subsequent messages under that name. If the subscriber then reconnects using the same durable subscription name, the messages held under that name are delivered to the subscriber.

JMS Message Formats

The JMS API defines the standard form of a JMS message, which should be portable across all JMS providers. Because the JMS API was designed to accommodate many existing providers, the resulting message form encompasses a wide variety of features. ATG supports all of these features, but internally adheres to conventions that greatly narrow the set of features developers must master.

A JMS message consists of two parts:

- Message header
- Message body

Message header

The header contains system-level information common to all messages, such as the destination and the time it was sent, while the body contains only application-specific data. The header can also contain some application-specific information, stored as keyword/value properties. However, not all providers allow an arbitrary amount of data to be stored in the header, it is a good idea to keep most application-specific data in the message body.

The most important header value is the `JMSType`. This is a String that is used to identify what kind of message is being sent. Handlers often examine the `JMSType` to see how they should handle an incoming message.

The header is useful for specifying message selectors. When a receiver subscribes to a destination, it can specify a message selector, which acts as a filter for weeding out messages the receiver does not want to see. The message selector must be specified in terms of the message's header. For example, a receiver can specify a message selector saying that it wants to see only messages whose `JMSType` is `atg.das.Startup`. The message selector can refer to both system-level and application-specific header properties.

Message body

To accommodate the various data formats of existing providers, JMS defines five distinct message body types. In the JMS API, these translate into five Java interfaces, each subclassing `javax.jms.Message`:

Interface	**Message body type**
`javax.jms.TextMessage`	A block of text, represented in Java as a String. For example, this type of message can be used to represent a message as an XML file.
`javax.jms.ObjectMessage`	A Java object (which must be serializable). For example, the message can contain a Java Bean whose properties represent the different data elements of the message.
`javax.jms.MapMessage`	A set of keyword/value pairs.
`javax.jms.BytesMessage`	A block of binary data, represented in Java as a byte array. This format is often used to interface with an external messaging system that defines its own binary protocol for message formats.
`javax.jms.StreamMessage`	A list of Java primitive values. This type can be used to represent certain data types used by existing messaging systems.

JMS systems can support all, or only a subset, of these message formats. ATG's JMS providers support the subset described in the next section.

ATG and JMS

ATG includes a number of JMS-related tools, which are known collectively as the Dynamo Messaging System (DMS). The main parts of DMS are:

- Two JMS providers, Local JMS and SQL JMS. Local JMS is built for high-speed low-latency synchronous messaging within a single process. SQL JMS is more robust, and uses an SQL database to handle communication between components within the same ATG application, or components running in different processes.
- Patch Bay is an API and configuration system layered on top of JMS. Patch Bay is designed to ease the development of messaging applications in ATG. The Patch Bay API allows Nucleus components to send and receive messages. The configuration system uses an XML file to specify how these components should be connected. This file allows developers to change or add connections between components without changing code. Patch Bay also maintains a Message Registry that the ATG user interfaces use to present lists of possible notifications to users. ATG registers the messages that it sends with the Message Registry. Applications can also register their own messages, which then appear in the ATG user interfaces.

The different DMS pieces can be used independently. For example, you can use Local JMS, SQL JMS, or both, with or without Patch Bay. You can use a third-party JMS provider, or use the JMS implementation provided with your application server, also with or without Patch Bay. For more information about other JMS providers you can use, see the documentation for your application server.

ATG Message Conventions

ATG's JMS providers use the following message format conventions, based on a subset of the JMS message options:

- Messages are of type `javax.jms.ObjectMessage`. The objects stored in the `ObjectMessage` are serializable Java Beans whose properties contain the message's data. These Java Beans are called message beans.
- The class names for the Message Beans all end with `Message`—for example, `atg.nucleus.dms.DASMessage`.
- The `JMSType` header is used to identify the type of message being fired. `JMSType` names follow package name conventions—for example, `atg.das.Startup`. The `JMSType` name does not need to be an actual Java class name; it follows the package naming conventions to avoid collisions with other JMS applications.
- Each `JMSType` corresponds to exactly one Message Bean class. For example, a message of `JMSType` `atg.das.Startup` is always an `ObjectMessage` containing a bean of type `atg.nucleus.dms.DASMessage`. Multiple `JMSTypes` can correspond to the same Message Bean class. For example, `JMSType` `atg.das.Shutdown` also corresponds to `atg.nucleus.dms.DASMessage`.
- Messages avoid the use of application-specific header values. All such values are instead represented as properties of the contained message bean.

Using Local JMS

Local JMS is a JMS provider supplied with ATG. Messages sent through Local JMS can travel only between components in the same ATG process. Local JMS delivers messages synchronously. This means that when a component sends a message, the sending component blocks until the receiving components receive and process the message. In fact, the entire message sending and receiving process occurs within a single thread. As a result, both the sending and receiving of the message occurs in the same transaction. Also as a result, Local JMS has extremely high performance, adding very little overhead to each message delivery.

Local JMS does no queuing. When a message is sent, Local JMS immediately finds out who the receivers are and calls the appropriate methods on the receivers to deliver the message, waiting for each receiver to process the message before delivering the message to the next receiver. Only when the message has been delivered to all receivers does control return to the sender. In this way, Local JMS works more like Java Bean events than like typical JMS implementations; when a Java Bean fires an event, it actually calls a method on several registered listeners.

Local JMS is also non-durable; all messages are non-persistent. If a message is sent to a queue destination that has no listeners, the message disappears. Also, durable subscriptions to topic destinations act exactly like non-durable subscriptions—if a subscriber is not listening to a topic, it misses any messages sent to that topic whether it is subscribed durably or not.

Local JMS is most often used to pass data around to various components within a single request. For example, a user might view content on a certain page, thereby causing a message to be sent. A listener might be configured to listen for that message and update a value in the user's profile as a result. The profile must be updated in the same request, or the updated value might not take effect in time for the next request. To make sure the sender and receiver both carry out their actions in the same request, the message should be carried over Local JMS.

Of course, the same effect can be achieved by using a single component to watch for the user to view content then update the database. But by decoupling the two actions into separate components joined by JMS, the system allows new senders or receivers to be added to the system without changing any existing code.

Creating Local JMS Destinations

In Local JMS, you create destinations by setting the `localJMSQueueNames` and `localJMSTopicNames` properties of the `/atg/dynamo/messaging/MessagingManager` component. For example:

```
localJMSQueueNames+=/Orders
localJMSTopicNames+=/RegistrationEvents,/FinancialEvents
```

When a Nucleus-based application starts up, it creates these destinations. To access a Local JMS destination in your code, you use JNDI references of the form:

```
localdms:/local{queue-or-topic-name}
```

For example, `localdms:/local/Orders`.

You can also use the DMS configuration file (discussed in the Configuring Patch Bay section) to create Local JMS destinations. These destinations are specified by name, separated into topics and queues:

```
<dynamo-message-system>
  <patchbay>
    ...
  </patchbay>

  <local-jms>

    <topic-name>/MyApp/RegistrationEvents</topic-name>
    <topic-name>/MyApp/FinancialEvents</topic-name>
    ...

    <queue-name>/MyApp/Orders</queue-name>
    ...
  </local-jms>
</dynamo-message-system>
```

When a Nucleus-based application starts up, it create these destinations with the JNDI names `localdms:/local/MyApp/RegistrationEvents`, `localdms:/local/MyApp/FinancialEvents`, and `localdms:/local/MyApp/Orders`.

Remember that Local JMS keeps no state, so adding these topics and queues simply creates named locations for messages to be sent locally. Nothing is actually added to a back-end storage system.

Using SQL JMS

Local JMS implements synchronous, extremely high-performance messaging. However, many messaging applications require messaging to be asynchronous. When a sender sends a message asynchronously, the message is handed off to the JMS provider, and the sender continues on with its work. After the sender passes the message to the JMS provider, the sender does not need to be informed if or when the message has been delivered to its final recipients.

Asynchronous messaging is useful for processes that can be broken down into separate stages, where each stage might take an unknown amount of time. For example, ATG Commerce uses asynchronous messaging to process an order. Each stage in the order (calculating tax, checking inventory, sending orders to shipping houses, sending confirmation email to the user) is a single action that is activated by an incoming message from the previous stage, and ends by sending a message to the next stage in the process. When the user submits an order, a message is sent to the first stage in the process. The user is told that the ordering process has started, but does not know about the completion of the process until a later email is sent.

Another key difference between Local JMS and SQL JMS is message persistence. Local JMS stores no state, so if the system fails, all messages are lost. SQL JMS uses an SQL database for persistence of messages. This ensures that messages are not lost in the event of system failure, and enables support for persistent queues and durable subscriptions, as described in Message Persistence.

To deliver messages, SQL JMS polls the database periodically, checking the appropriate tables to see if any new messages were written. If so, those messages are delivered to the appropriate message receivers and then removed from the database. This all occurs transactionally, so if a failure occurs or the transaction rolls back, the messages are all returned to the database, again guaranteeing that messages do not get lost.

Note: In SQL JMS, the sending of a message and the receiving of a message occur in separate transactions. A sender might send a message in a transaction that later commits successfully. This does not mean that the receiver has successfully received the message. It just means that SQL JMS has successfully delivered the message to its destination. At some point in the future, receipt of the message is placed in another transaction. The message is then removed from the database when that second transaction successfully commits.

SQL JMS uses standard JDBC drivers to communicate with the database. This allows SQL JMS to operate in a distributed environment, where an ATG server and the database are located on different machines. SQL JMS can also run on multiple ATG servers at once, all utilizing the same database. This enables multiple ATG servers to use SQL JMS to communicate with each other. Finally, if the JDBC driver supports the XA protocol, SQL JMS also supports XA, so it can participate in transactions involving multiple resources.

By default, the connection factory for all SQL JMS topic and queue connections (including XA connections) is the Nucleus component `/atg/dynamo/messaging/SqlJmsProvider`. If you are using SQL JMS with Patch Bay, you can specify a different connection factory when you configure Patch Bay (though there is generally no reason to do so). If you are using SQL JMS without Patch Bay, you cannot specify a different connection factory.

From the developer's perspective, very little changes when using SQL JMS instead of Local JMS. The message source and receiver components are still coded in essentially the same way whether they are using Local JMS or SQL JMS. The main difference is that the components are configured by pointing them at SQL JMS destinations rather than Local JMS destinations.

Creating and Accessing SQL JMS Destinations

In SQL JMS, destinations are represented by entries in the `dms_queue` and `dms_topic` tables, so adding new destinations is a matter of inserting new rows into these tables. However, this should not be done directly, as is difficult to coordinate this with the mechanism that generates new IDs for the destinations.

Instead, you can create destinations using the `requiredQueueNames` and `requiredTopicNames` properties of the `/atg/dynamo/messaging/SqlJmsProvider` component. For example:

```
requiredQueueNames+=MyApp/Orders
requiredTopicNames+=MyApp/RegistrationEvents,MyApp/FinancialEvents
```

When SQL JMS starts, it looks at these lists of queue and topic names. It then looks into the `dms_queue` and `dms_topic` tables and add any topic or queue names that are not already in those tables.

To access an SQL JMS destination in your code, you use JNDI references of the form:

```
sqldms:/{queue-or-topic-name}
```

For example, the first topic above is:

```
sqldms:/MyApp/RegistrationEvents
```

Administering SQL JMS

When SQL JMS is used, the database keeps track of all the topics and queues that were added to the system. The database also keeps track of any subscribers that are currently in the system so that it can know who should receive a message sent to a particular destination. The database stores messages that were sent through a particular destination to various subscribers, so that the next time a subscriber polls to see if there are any messages, those messages can be delivered at that time. After a message has been delivered to all of its recipients, that message is automatically removed from the database by the last polling recipient.

Because SQL JMS stores state in a database, it requires occasional administration and maintenance. This section explains how the database administrator can perform various tasks to manage SQL JMS:

- Configuring Databases and Data Sources
- Adjusting the SQL JMS Polling Interval
- Removing SQL JMS Destinations and Subscriptions
- Monitoring Message Buildup
- Using the SQL-JMS Administration Interface

Configuring Databases and Data Sources

ATG comes with a SOLID database preconfigured for evaluation purposes. This database already has the necessary tables created for use with SQL JMS. But if SQL JMS is going to be used on another database, the appropriate tables need to be created in that database. These tables are created by the script that creates the DAS schema:

```
<ATG10dir>/DAS/sql/install/{db-type}/das_ddl.sql
```

This script should be run on the appropriate database to initialize the DAS schema, including the SQL JMS tables. To drop the tables in the DAS schema, use the script:

```
<ATG10dir>/DAS/sql/install/{db-type}/drop_das_ddl.sql
```

By default, the SQL JMS system uses the `JTDataSource` component (located in Nucleus at `/atg/dynamo/service/jdbc/JTDataSource`) to obtain its JDBC connections. This means that SQL JMS uses the same database as other ATG application components. If SQL JMS is to be used with a different database, a `DataSource` component must be configured for that database, and the SQL JMS system must be configured to use this new data source. The SQL JMS system is controlled by the Nucleus component at `/atg/dynamo/messaging/SqlJmsProvider`; you can set the `dataSource` property of this component to specify a different data source, like this:

```
dataSource=/atg/dynamo/service/jdbc/MyDataSource
```

Configuring the SQLJmsProvider for Informix

If your ATG installation uses an Informix database system, add the following setting to the properties of `/atg/dynamo/messaging/SqlJmsProvider`:

```
parameterizedSelect=false
```

Configuring the SQLJmsProvider for DB2

If your ATG installation uses a DB2 database system, add the following settings to the properties of `/atg/dynamo/messaging/SqlJmsProvider`:

```
parameterizedSelect=false
useSetBinaryStream=false
```

Adjusting the SQL JMS Polling Interval

SQL JMS works through polling. At periodic intervals it performs a query on the database to see if there are any messages waiting to be delivered to its local clients. By default, this polling occurs at 20 second intervals. This means that messages have an average latency of 10 seconds (average latency is half of the polling interval, as long as messages are sent at random times).

Decreasing the polling interval decreases the average latency. However, decreasing the polling interval also increases the frequency of queries from each client, thereby increasing the database load. For example, if the polling interval is halved to 10 seconds, ATG doubles the number of queries that it makes in the same period of time. If there are many ATG servers running SQL JMS against the same database, this can add up to a significant load on the database.

So if the database load is too high, the administrator should increase the polling interval on the ATG servers to decrease the number of queries each server is making. This increases latency, making some messages take longer to get from sender to receiver, but decreases the load on the database.

The polling interval is set in the `messagePollSchedule` property of the `/atg/dynamo/messaging/SqlJmsProvider` component:

```
messagePollSchedule=every 20 sec in 10 sec
```

Note: While latency is affected by the polling interval, overall system throughput should remain unchanged. Each time a query performs a poll, it reads in all messages waiting for that client. For example, if a message is being sent every second, and the polling interval is set to 20 seconds, every poll reads 20 messages, yielding an effective throughput of 1 message/second, with an average latency of 10 seconds. But if the polling interval is set to 30 seconds, the average latency increases to 15 seconds, but every poll reads 30 messages, again yielding a throughput of 1 message/second.

So, administrators should be aware that they are trading low latency for high database load, or high latency for low database load. But overall throughput is not affected by the polling interval.

Removing SQL JMS Destinations and Subscriptions

The `requiredQueueNames` and `requiredTopicNames` properties can be used to add new destinations to the system, but they cannot be used to remove destinations. Removing a destination from one of those properties just means that the system does not make sure that the destination exists when it starts up; it does not actually remove the destination from the system.

ATG includes a browser-based interface that you can use to administer and remove queues and topics. See Using the SQL-JMS Administration Interface below.

Removing a queue or topic involves more than just removing rows from the `dms_queue` and `dms_topic` tables. Any messages in those queues or topics also have to be removed, as well as any subscriptions associated with those queues or topics.

The process of removing a queue or topic should preferably be done when the ATG application is shut down. If you want to perform this task on a running ATG application, you must first make sure that all message producers and consumers are closed and unsubscribed.

Removing a Queue

You can remove a queue using the SQL-JMS Administration Interface, or by issuing SQL statements. For example, the following SQL statements remove a queue named `fooQueue`:

```
DELETE FROM dms_msg_properties
  WHERE msg_id IN (SELECT msg_id
                   FROM dms_queue_entry
                   WHERE queue_id IN (SELECT queue_id
                                      FROM dms_queue
                                      WHERE queue_name = 'fooQueue'))
DELETE FROM dms_msg
  WHERE msg_id IN (SELECT msg_id
                   FROM dms_queue_entry
                   WHERE queue_id IN (SELECT queue_id
                                      FROM dms_queue
                                      WHERE queue_name = 'fooQueue'))
DELETE FROM dms_queue_entry
  WHERE queue_id IN (SELECT queue_id
                     FROM dms_queue
                     WHERE queue_name = 'fooQueue')
DELETE FROM dms_queue
  WHERE queue_name = 'fooQueue'
```

Removing a Topic

You can remove a topic using the SQL-JMS Administration Interface, or by issuing SQL statements. Before you remove a topic, however, make sure that no message producer is still publishing to the topic, and that all durable subscribers to the topic were deleted. See Removing Durable Subscribers below for more information.

The following SQL statements delete a topic named `fooTopic`, along with any remaining subscribers to that topic:

```
DELETE FROM dms_msg_properties
  WHERE msg_id IN (SELECT msg_id
                   FROM dms_topic_entry
                   WHERE subscriber_id IN (SELECT subscriber_id
                                           FROM dms_topic_sub
                                           WHERE topic_id IN (SELECT topic_id
                                                              FROM dms_topic
                                                              WHERE
                                                    topic_name = 'fooTopic')))
DELETE FROM dms_msg
  WHERE msg_id IN (SELECT msg_id
                   FROM dms_topic_entry
                   WHERE subscriber_id IN (SELECT subscriber_id
                                           FROM dms_topic_sub
                                           WHERE topic_id IN (SELECT topic_id
                                                              FROM dms_topic
                                                              WHERE
                                                    topic_name = 'fooTopic')))
DELETE FROM dms_topic_entry
  WHERE subscriber_id IN (SELECT subscriber_id
                          FROM dms_topic_sub
                          WHERE topic_id IN (SELECT topic_id
                                             FROM dms_topic
                                             WHERE topic_name = 'fooTopic'))
DELETE FROM dms_topic_sub
  WHERE topic_id IN (SELECT topic_id
                     FROM dms_topic
                     WHERE topic_name = 'fooTopic')
DELETE FROM dms_topic
  WHERE topic_name = 'fooTopic'
```

Removing Durable Subscribers

Durable subscriptions hold messages for topic subscribers even when those subscribers are not online. If a message is sent to a destination that has a durable subscriber, the message is stored in the database until that subscriber comes online and reads its message.

However, if a client never comes online to read its messages, perhaps because the application is no longer active or has been changed to use another durable subscription name, those messages build up in the database. If durable subscribers disappear from the system, the appropriate entries in the database should also be removed to prevent messages from building up without bound.

You can remove SQL JMS subscribers using the SQL-JMS Administration Interface, or you can remove them programmatically. There is a standard JMS method for removing durable subscribers, `TopicSession.unsubscribe()`. This method deletes the state being maintained on behalf of the subscriber by its provider. You should not delete a durable subscription while it has an active

TopicSubscriber for it, or while a message received by it is part of a transaction or has not been acknowledged in the session.

The following code removes a durable subscriber:

```
SqlJmsManager manager = (SqlJmsManager) service;
XATopicConnection xac = manager.createXATopicConnection();
xac.start();
XATopicSession xas = xac.createXATopicSession();
TopicSession ts = xas.getTopicSession();
tx.unsubscribe("fooTopic");
xac.close();
```

Monitoring Message Buildup

Both queues and durable topic subscriptions can build up messages in the database. If no client is reading from the queue, or no client connects to read from a durable subscription, those lists of messages continue to grow without bound.

The administrator should periodically check the JMS system to see if there are any queues or durable subscriptions that are growing in this manner. If so, the administrator should contact the application developers to see if their applications are behaving correctly. If necessary, the administrator might wish to remove the messages in those queues and durable subscriptions.

You can check the number of entries in a queue or a durable subscription using the SQL-JMS Administration Interface. You can also check these statistics using SQL, as described below.

Measuring a Queue

The following SQL statements select all entries in the queue named fooQueue. Counting these entries gives the current size of the queue:

```
SELECT msg_id
  FROM dms_queue_entry
  WHERE queue_id IN (SELECT queue_id
                     FROM dms_queue
                     WHERE queue_name = 'fooQueue')
```

Measuring a Durable Subscription

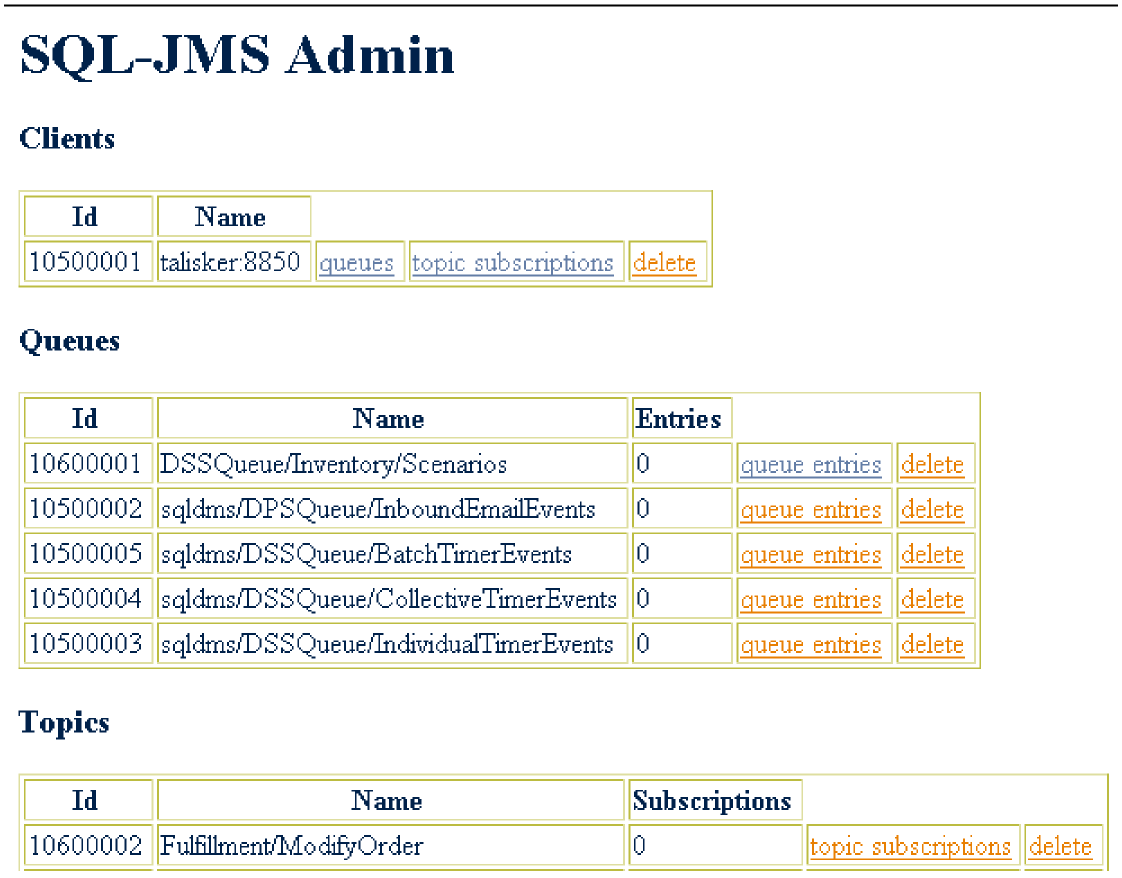
The following SQL statements select all entries in the durable subscription named fooSubscriber. Counting these entries gives the current size of the durable subscription.

```
SELECT msg_id
  FROM dms_topic_entry
  WHERE subscriber_id IN (SELECT subscriber_id
FROM dms_topic_sub
WHERE subscriber_name = 'fooSubscriber')
```

Using the SQL-JMS Administration Interface

ATG includes a browser-based administration interface for its SQL JMS message system. This interface makes it easy to view, add, and delete SQL JMS clients, queues, and topics. For information about starting up and accessing the interface, see the *ATG Installation and Configuration Guide*.

The main page of the SQL-JMS Administration Interface displays lists of all clients, queues, and topics in the SQL JMS system:



SQL-JMS Admin

Clients

Id	Name			
10500001	talisker:8850	queues	topic subscriptions	delete

Queues

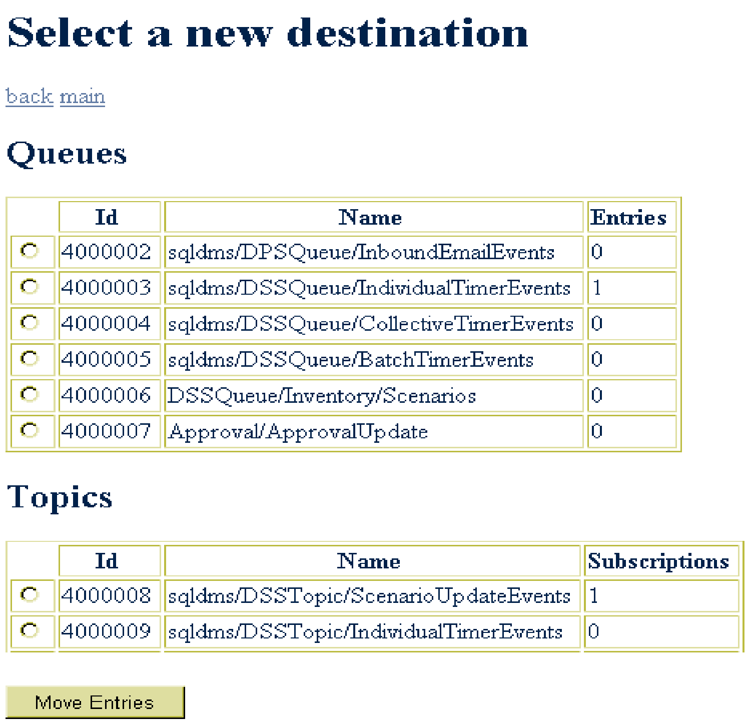
Id	Name	Entries		
10600001	DSSQueue/Inventory/Scenarios	0	queue entries	delete
10500002	sqldms/DPSQueue/InboundEmailEvents	0	queue entries	delete
10500005	sqldms/DSSQueue/BatchTimerEvents	0	queue entries	delete
10500004	sqldms/DSSQueue/CollectiveTimerEvents	0	queue entries	delete
10500003	sqldms/DSSQueue/IndividualTimerEvents	0	queue entries	delete

Topics

Id	Name	Subscriptions		
10600002	Fulfillment/ModifyOrder	0	topic subscriptions	delete

You can click on any of the links to view more details about each client, queue, and topic. You can click on the delete links to delete a client, queue, or topic.

The Queue Entries page displays all pending and unhandled queue entries for the queue you selected. The move and delete links for each entry let you move an item to a different queue or topic, or delete the entry altogether. The radio buttons let you delete or move more than one queue entry.



Select a new destination

back main

Queues

	Id	Name	Entries
○	4000002	sqldms/DPSQueue/InboundEmailEvents	0
○	4000003	sqldms/DSSQueue/IndividualTimerEvents	1
○	4000004	sqldms/DSSQueue/CollectiveTimerEvents	0
○	4000005	sqldms/DSSQueue/BatchTimerEvents	0
○	4000006	DSSQueue/Inventory/Scenarios	0
○	4000007	Approval/ApprovalUpdate	0

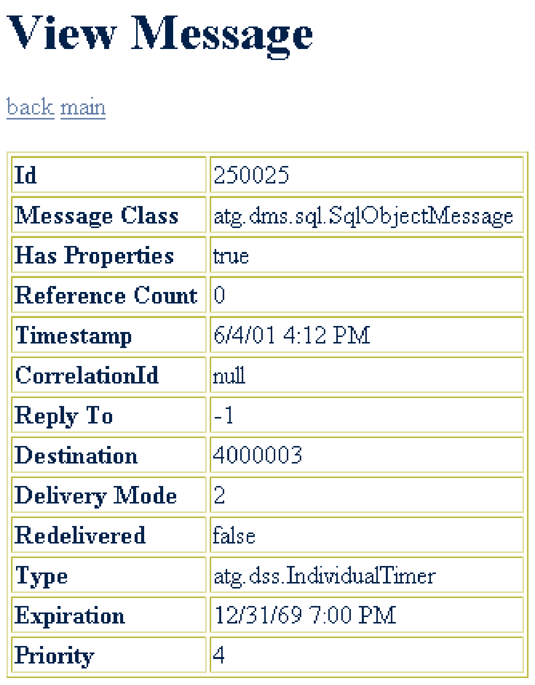
Topics

	Id	Name	Subscriptions
○	4000008	sqldms/DSSTopic/ScenarioUpdateEvents	1
○	4000009	sqldms/DSSTopic/IndividualTimerEvents	0

The Topic Subscriptions page lists information for each topic. You can delete a topic subscription using the delete link, or view the entries for the topic by clicking the topic entries link. The Topic Entries page, just like the Queue Entries page, displays all pending and unhandled topic entries for the topic you selected and lets you move or delete topic entries.

In general, you should avoid manipulating an SQL JMS system while it is running. When you delete SQL JMS components from a system that is running, you only delete entries from the database. Some information can be maintained in memory at that point. If you delete a client while it is not running, you need also to delete any associated queues. Also, remember that it is better to shut down an ATG application normally, using the Stop Dynamo button in ATG Dynamo Server Admin or the ATG Control Center, rather than abruptly killing the process.

Be careful when moving messages. If a message's class is not compatible with the destination where you move it, errors result. You can check the message class in the View Message table in the SQL-JMS Administration Interface:


View Message

back main

Id	250025
Message Class	atg.dms.sql.SqlObjectMessage
Has Properties	true
Reference Count	0
Timestamp	6/4/01 4:12 PM
CorrelationId	null
Reply To	-1
Destination	4000003
Delivery Mode	2
Redelivered	false
Type	atg.dss.IndividualTimer
Expiration	12/31/69 7:00 PM
Priority	4

Overview of Patch Bay

Patch Bay is designed to simplify the process of creating JMS applications. Patch Bay includes a simplified API for creating Nucleus components that send and receive messages, and a configuration file where you declare these components and your JMS destinations. When a Nucleus-based application starts up, it examines this file and automatically creates the destinations and initializes the messaging components. This means your code does not need to handle most of the JMS initialization tasks, such as obtaining a ConnectionFactory, obtaining a JMS Connection, and creating a JMS Session

Patch Bay Manager

Patch Bay is represented in Nucleus as the component `/atg/dynamo/messaging/MessagingManager`, which is of class `atg.dms.patchbay.PatchBayManager`. As with all Nucleus components, Patch Bay is configured with a properties file. The properties file controls the general behavior of the Patch Bay system; it configures such things as the transaction manager used by Patch Bay and logging behavior.

In addition to the properties file, the `MessagingManager` uses an XML file called the DMS configuration file to configure the individual parts of the Patch Pay system, such as JMS providers, message sources and sinks, and destinations. The `definitionFile` property of the `MessagingManager` component names the DMS configuration file. (In some places, the DMS configuration file is also referred to as the Patch Bay definition file.) See Configuring Patch Bay for more information.

Messaging Components

As with standard JMS, the Patch Bay API includes Java interfaces that messaging components must implement in order to send and receive messages. However, these interfaces differ from the standard JMS interfaces, and the terminology is somewhat different:

- Message source: A component that can send messages. A message source must implement the `atg.dms.patchbay.MessageSource` interface.
- Message sink: a component that can receive messages. A message sink must implement the `atg.dms.patchbay.MessageSink` interface.
- Message filter: a component that implements both interfaces, and can send and receive messages.

All message sources, sinks, and filters must have global scope.

Note: Unlike standard JMS, Patch Bay does not have separate interfaces for objects that communicate with topics and those that communicate with queues. A message source can send messages to both topics and queues, and a message sink can receive messages from topics and queues.

In addition to your sources and sinks, you must also define standard JMS destinations; for example, if your JMS provider is SQL JMS, you create destinations as described in Creating and Accessing SQL JMS Destinations. Patch Bay cannot connect a message source directly to a message sink. Instead, the two must be connected through a JMS destination.

Configuration

One of the key DMS design principles is to separate the design of the messaging components from the plumbing. Message sources should be written without regard for where their messages are going. The same code should be used if the messages are to be delivered to multiple subscribers, no subscribers, or subscribers in different processes. Directing where messages go is part of the Patch Bay's configuration, not the message source's code. In the same way, message sinks should be written regardless of where messages are coming from. The same code should be used if messages are coming in from multiple publishers simultaneously, or if messages are arriving from remote processes, or if no messages are arriving at all. Determining how messages are delivered to message sinks is determined by the Patch Bay's configuration, not the code in the message sinks.

For more information about configuring Patch Bay, see the Configuring Patch Bay section.

Patch Bay Initialization

Patch Bay defines a simple life cycle for message sources, sinks, and filters. When Patch Bay is started, it resolves each of the Nucleus names. If the referenced components are not yet created, they are created at this time according to the standard Nucleus name resolution procedure (including a call to `doStartService` if the component extends `GenericService`). For information about Nucleus name resolution procedure, see the Basic Nucleus Operation section of the Nucleus: Organizing JavaBean Components chapter.

At this point, message sinks should be prepared to receive messages, which can start arriving at any time, possibly from multiple simultaneous threads.

Message sources follow a slightly more complicated protocol. After a message source is resolved in Nucleus, Patch Bay calls `MessageSource.setMessageSourceContext()` on the component. This provides the component with a `context` object that it can use to create and send messages. However, the component should not begin to send messages yet.

At this point, Patch Bay initializes the various JMS providers and makes sure that the messaging infrastructure is up and running. It then walks through each of the message sources and calls `MessageSource.startMessageSource()` on each one. After this call, the message sources can start sending messages. Depending on the message source, this method is where a message source registers itself with the scheduler, or start a server to listen for incoming messages, or just set a flag that gates the sending of messages.

Message filters are combinations of message sources and message sinks. They implement both interfaces, and must follow the protocols for both. This means that a message filter must be able to receive messages as soon as it has been initialized, but should not initiate the sending of messages before `setMessageSourceContext()` and `startMessageSource()` are called.

There is one situation where message filters behave differently. A typical operation for a message filter is to receive a message, then to send another message in response. In this case, it is acceptable for the message to send a message in response to a received message, even if `startMessageSource()` has not yet been called (although `setMessageSourceContext()` must be called first in all cases). It is still not acceptable for a message filter to initiate a message before `startMessageSource()` has been called, but it is fine for the message filter to send a message in response to a received message.

Patch Bay API

One of Patch Bay's main design goals is to ease the burden of coding messaging applications. To do this, Patch Bay presents a highly distilled API for messaging components to use to send and receive messages.

This section discusses:

- Creating Message Sources
- Creating Message Sinks
- Creating Message Filters

Creating Message Sources

A message source must implement the `atg.dms.patchbay.MessageSource` interface. Through this interface, the message source is assigned a `MessageSourceContext` that it can use to create and send messages. The following example demonstrates how to do this:

```
import atg.dms.patchbay.*;
import javax.jms.*;

...
```

```
MessageSourceContext mContext;
boolean mStarted = false;

// These methods implement the MessageSource interface
public void setMessageSourceContext (MessageSourceContext pContext)
{ mContext = pContext; }
public void startMessageSource ()
{ mStarted = true; }
public void stopMessageSource ()
{ mStarted = false; }

// This method will send a message
public void sendOneMessage ()
  throws JMSException
{
  if (mStarted && mContext != null) {
    TextMessage msg = mContext.createTextMessage ();
    msg.setJMSType ("atg.test.Test1");
    msg.setText ("Test text string");
    mContext.sendMessage (msg);
  }
}
```

The `setMessageSourceContext`, `startMessageSource`, and `stopMessageSource` methods implement the `MessageSource` interface. Messages can be sent from any method in the message source, such as the `sendOneMessage` method in the example.

The `sendOneMessage` method makes sure that `startMessageSource` has been called. It then creates, populates, and sends a `TextMessage`. Typically the only data that needs to be set on a message is the `JMSType` and the data in the message's body. A `TextMessage`'s body is set by calling `setText`, an `ObjectMessage`'s body is set by calling `setObject`, and so on. The `sendMessage` method then delivers the message to Patch Bay. Depending on how Patch Bay is configured, that message is delivered to a JMS destination or group of destinations.

If any of those destinations are managed by Local JMS, the `sendMessage` call does not return until the message is delivered to all message sinks attached to the Local JMS destinations. Destinations that are not managed by Local JMS (such as those managed by SQL JMS) deliver messages asynchronously. In other words, the `sendMessage` call returns immediately, even if the messages are not yet delivered to their final recipients.

If the destinations are managed by a transactional JMS provider (such as SQL JMS), any messages sent through `sendMessage` are not actually sent until the overall transaction is committed. If the transaction rolls back, none of the messages are sent. This does not apply to Local JMS; because Local JMS is synchronous, sending a message happens instantly, without waiting for the current transaction to complete.

Creating Message Sinks

Message sinks are somewhat simpler than message sources. A message sink must implement the `atg.dms.patchbay.MessageSink` interface. This interface defines a single method, `receiveMessage`, which is called to notify the message sink that a message is being delivered. This method might be called simultaneously by many threads, so the message sink should be coded accordingly. The following is a simple example of how a message sink might handle a message:

```
import atg.dms.patchbay.*;
import javax.jms.*;

...

public void receiveMessage (String pPortName, Message pMessage)
  throws JMSException
{
  System.out.println ("Received message from port " +
                      pPortName +
                      " of JMSType " +
                      pMessage.getJMSType ());

  if (pMessage instanceof TextMessage) {
    System.out.println ("  TextMessage, value = \"" +
                        ((TextMessage) pMessage).getText () +
                        "\"");
  }
  else if (pMessage instanceof ObjectMessage) {
    System.out.println ("  ObjectMessage, value = \"" +
                        ((ObjectMessage) pMessage).getObject () +
                        "\"");
  }
  else if (pMessage instanceof MapMessage) {
    System.out.println ("  MapMessage");
  }
  else if (pMessage instanceof StreamMessage) {
    System.out.println ("  StreamMessage");
  }
  else if (pMessage instanceof BytesMessage) {
    System.out.println ("  BytesMessage");
  }
}
```

This example just prints out a text string whenever it receives a message, including the port name (described in the Using Messaging Ports section of this chapter), the `JMSType`, and some additional information depending on the actual subclass of the message.

Creating Message Filters

Message filters must implement both the `MessageSource` and the `MessageSink` interface. A message filter typically implements `receiveMessage` by manipulating the message in some way, then sending a new message, like this:

```
import atg.dms.patchbay.*;
import javax.jms.*;

...

MessageSourceContext mContext;
boolean mStarted = false;

// These methods implement the MessageSource interface
public void setMessageSourceContext (MessageSourceContext pContext)
{ mContext = pContext; }
public void startMessageSource ()
{ mStarted = true; }
public void stopMessageSource ()
{ mStarted = false; }

public void receiveMessage (String pPortName, Message pMessage)
  throws JMSException
{
  if (pMessage instanceof TextMessage) {
    String text = ((TextMessage).getText ());
    String newText = text.replace ('.', '/');
    TextMessage msg = mContext.createTextMessage ();
    msg.setJMSType (pMessage.getJMSType ());
    msg.setText (newText);
    mContext.sendMessage (msg);
  }
}
```

This filter takes in `TextMessages`, then sends them out again with period (`.`) replaced by forward slash (`/`) in the text. Notice that the `mStarted` flag is not consulted, because a message filter is allowed to send out messages in response to incoming messages regardless of whether it has been started or stopped.

Configuring Patch Bay

Patch Bay is represented in Nucleus as the component `/atg/dynamo/messaging/MessagingManager`. The `definitionFile` property of the component `MessagingManager` names the XML file that configures Patch Bay. The value of this property is:

```
/atg/dynamo/messaging/dynamoMessagingSystem.xml
```

The name refers to a file within the configuration path, and should not be changed. For example, if the configuration path includes `/work/ATG10.0.1/home/localconfig`, the XML file might be found at:

```
/work/ATG10.0.1/home/localconfig/atg/dynamo/messaging/dynamoMessagingSystem.xml
```

For more information about the configuration path, see the Managing Properties Files section in the Nucleus: Organizing JavaBean Components chapter.

As with properties files found in the configuration path, the DMS configuration file might appear at several points in the configuration path. In this case, the configuration files are automatically combined at runtime into a single virtual file, using ATG's file combination feature (see XML File Combination in the Nucleus: Organizing JavaBean Components chapter). The resulting file is then used by the messaging system. This allows multiple applications to layer on top of each other, forming a single configuration file out of multiple configuration files. The overall file used by the messaging system is a combination of all those files, in the order they are found in the configuration path.

Depending on how many ATG products are installed, the configuration file can be compiled from the files with the pathname `/atg/dynamo/messaging/dynamoMessagingSystem.xml` within the various ATG configuration JAR files, using ATG's XML file combination rules. To modify the DMS configuration file, you should not edit any of the files in these JAR files. Instead, create a file with the pathname `/atg/dynamo/messaging/dynamoMessagingSystem.xml` and place it in your own application module or in the `<ATG10dir>/home/localconfig` directory.

To view the full (combined) DMS configuration file on your system, use the Component Browser in ATG Dynamo Server Admin. Navigate to the `/atg/dynamo/messaging/MessagingManager` component, and in the Properties table, click on the `definitionFiles` property. The resulting page displays the configuration path file name, the URL of the DTD, the pathnames of the source files that were combined to make up the configured value, and the full combined text of the XML file. Appendix C: DMS Configuration File Tag Reference shows the DTD for the Patch Bay configuration file, which provides a description of all tags used in the file.

This section discusses:

- Declaring JMS Providers
- Declaring Message Sources, Sinks, and Filters
- Connecting to Destinations
- Using Messaging Ports
- Using the Message Registry
- Delaying the Delivery of Messages
- Configuring Failed Message Redelivery

Declaring JMS Providers

By default, Patch Bay is configured to use Local JMS and SQL JMS. These providers and the connection factories they use are specified through Nucleus components of class `atg.dms.patchbay.JMSProviderConfiguration`. For example, to configure SQL JMS, ATG includes a component of this class named /atg/dynamo/messaging/DynamoSQLJMSProvider. The properties file for this component includes these lines:

```
providerName=sqldms
topicConnectionFactoryName=dynamo:/atg/dynamo/messaging/SqlJmsProvider
queueConnectionFactoryName=dynamo:/atg/dynamo/messaging/SqlJmsProvider
XATopicConnectionFactoryName=dynamo:/atg/dynamo/messaging/SqlJmsProvider
XAQueueConnectionFactoryName=dynamo:/atg/dynamo/messaging/SqlJmsProvider
supportsTransactions=true
supportsXATransactions=true
```

You generally should not need to modify any of these settings for Local JMS or SQL JMS. However, you can configure Patch Bay to work with additional JMS providers. See Using Patch Bay with Other JMS Providers for more information.

You can designate a JMS provider as the default provider in Patch Bay by setting the defaultJMSProvider property of the /atg/dynamo/messaging/MessagingManager component to point to the component that configures the provider. By default, this property points to the DynamoSQLJMSProvider component, which configures SQL JMS. Designating a provider as the default simplifies configuration of the destinations for that provider, and makes it easier to switch between providers. See Connecting to Destinations for more information.

You can also declare JMS providers at the top of the Patch Bay configuration file, using tags that correspond to the properties of JMSProviderConfiguration. Note, however, that a JMS provider declared this way cannot be designated as the default provider.

These tags are equivalent to the DynamoSQLJMSProvider properties shown above:

```
<!-- SQL JMS provider -->
<provider>
  <provider-name>
    sqldms
  </provider-name>
  <topic-connection-factory-name>
    dynamo:/atg/dynamo/messaging/SqlJmsProvider
  </topic-connection-factory-name>
  <queue-connection-factory-name>
    dynamo:/atg/dynamo/messaging/SqlJmsProvider
  </queue-connection-factory-name>
  <xa-topic-connection-factory-name>
    dynamo:/atg/dynamo/messaging/SqlJmsProvider
  </xa-topic-connection-factory-name>
  <xa-queue-connection-factory-name>
    dynamo:/atg/dynamo/messaging/SqlJmsProvider
  </xa-queue-connection-factory-name>
  <supports-transactions>
    true
  </supports-transactions>
  <supports-xa-transactions>
    true
  </supports-xa-transactions>
</provider>
```

Declaring Message Sources, Sinks, and Filters

One of the functions of the DMS configuration file is to name all message sources, sinks, and filters existing in the system. As described earlier, these elements are globally scoped Nucleus services that implement the appropriate interfaces. Each element should be declared with its Nucleus name. For example:

```
<?xml version="1.0" ?>

<dynamo-message-system>
  <patchbay>

    <message-source>
      <nucleus-name>
        /atg/dynamo/messaging/TestSource1
      </nucleus-name>
    </message-source>

    <message-sink>
      <nucleus-name>
        /atg/dynamo/messaging/TestSink1
      </nucleus-name>
    </message-sink>

    <message-filter>
      <nucleus-name>
        /atg/dynamo/messaging/TestFilter1
      </nucleus-name>
    </message-filter>

  </patchbay>
</dynamo-message-system>
```

Note: The Nucleus names are examples only, and might not correspond to actual Nucleus components.

Any number of sources, sinks, and filters can be specified, in any order. Also, as mentioned above, if there are multiple `dynamoMessagingSystem.xml` files spread across configuration path entries, the sources, sinks, and filters from all of those files are registered.

Connecting to Destinations

After a message source, sink, or filter has been declared in the configuration file, it must be hooked up to JMS in order for its messages to go anywhere, or for it to receive messages. As discussed earlier, a messaging component is never connected directly to another component. Instead, a messaging component is hooked up to a JMS destination, maintained by one of the JMS providers registered with Patch Bay. Messaging components communicate with each other by hooking up to the same destination—if message source A sends messages to destination D, and message sink B receives messages from destination D, messages flow from A to B.

Whenever a destination is specified in the DMS configuration file, it must specify which provider owns that destination. The destination must also be named by its JNDI name, using the prefix appropriate to that destination's provider. As discussed earlier, ATG includes two providers: Local JMS and SQL JMS. The following table specifies the required information for each provider:

Provider	**provider-name**	**Destination JNDI prefix**
Local JMS	`local`	`localdms:/local`
SQL JMS	`sqldms`	`sqldms:/`

The following illustrates how a message source is connected to a destination in the DMS configuration file. In this case, the destination is managed by Local JMS, and is called `localdms:/local/TestMessages`:

```
<message-source>
  <nucleus-name>
    /atg/dynamo/j2ee/examples/TestMessageSource1
  </nucleus-name>

  <output-port>
    <port-name>
      DEFAULT
    </port-name>

    <output-destination>
      <provider-name>
        local
      </provider-name>
      <destination-name>
        localdms:/local/TestMessages
      </destination-name>
      <destination-type>
        Topic
      </destination-type>
    </output-destination>
  </output-port>

</message-source>
```

The `output-port` definition is described in the Using Messaging Ports section of this chapter. The important part of this example is the `output-destination` definition. This definition says that messages coming out of this Nucleus component should be directed to the topic called `localdms:/local/TestMessages`, managed by JMS provider `local`. Multiple destinations can be specified for a component. For example:

```
<message-source>
  <nucleus-name>
    /atg/dynamo/j2ee/examples/TestMessageSource1
  </nucleus-name>

  <output-port>
    <port-name>
      DEFAULT
    </port-name>

    <output-destination>
      <provider-name>
        local
      </provider-name>
    <destination-name>
      localdms:/local/TestMessages
    </destination-name>
    <destination-type>
      Topic
    </destination-type>
    </output-destination>

    <output-destination>
      <provider-name>
        sqldms
      </provider-name>
      <destination-name>
         sqldms:/PersistentTopic1
      </destination-name>
      <destination-type>
        Topic
      </destination-type>
    </output-destination>

  </output-port>

</message-source>
```

This says that each message coming out of the component is sent to a destination in Local JMS, and a destination in SQL JMS. The messages are sent in the order specified.

Message sinks are configured in much the same way. For example:

```
<message-sink>
  <nucleus-name>
    /atg/dynamo/j2ee/examples/TestMessageSink1
  </nucleus-name>

  <input-port>
    <port-name>
      DEFAULT
    </port-name>

    <input-destination>
      <provider-name>
        local
      </provider-name>
      <destination-name>
        localdms:/local/TestMessages
      </destination-name>
      <destination-type>
         Topic
      </destination-type>
    </input-destination>

    <input-destination>
      <provider-name>
        sqldms
      </provider-name>
      <destination-name>
        sqldms:/PersistentTopic1
      </destination-name>
      <destination-type>
         Topic
      </destination-type>
      <durable-subscriber-name>
        testMessageSink1
      </durable-subscriber-name>
    </input-destination>

  </input-port>

</message-sink>
```

This configuration says that messages sent to either topic in either provider are passed to the `TestMessageSink1` component, using the `MessageSink.receiveMessage()` call.

Notice that the `sqldms` `input-destination` specifies a `durable-subscriber-name`. This means that the connection to the topic should be made using a durable subscription, with the given durable subscriber name. If messages are sent to this topic while the subscriber is off-line, those messages are held under this name. When the subscriber starts up, the messages held under that name are passed to the message sink.

The `durable-subscriber-name` is optional. If it is not supplied, the subscription is non-durable, meaning that the message sink misses any messages sent to the topic while the message sink server is off-line. Durable subscriptions are probably used whenever SQL JMS is used, as most applications that require the robust persistence of SQL JMS also probably want the functionality of durable subscriptions.

Specifying Destinations for the Default Provider

A potential problem with specifying destinations as described above is that the names are provider-specific, because each provider can use different naming conventions for destinations. This means that if you change providers, you might need to rename all of your destinations in the Patch Bay configuration file. This is especially likely if your application server is IBM WebSphere Application Server or Oracle WebLogic Server, because you might want to switch at some point from SQL JMS to the application server's own provider.

To simplify this process, Patch Bay provides a generic naming scheme for destinations, and automatically maps these names to the actual names used by SQL JMS, IBM WebSphere Application Server, or Oracle WebLogic Server, depending on the provider designated as the default provider in Patch Bay. (See Declaring JMS Providers for information about the default JMS provider.) In this naming scheme, destinations for the default provider begin with the prefix `patchbay:/`. For example, suppose you specify a destination name as `patchbay:/myQueues/alertsQueue`. The following table shows the actual destination name that Patch Bay maps this name to, depending on whether the default JMS provider is SQL JMS, WebSphere, or WebLogic:

Provider	**Destination Name**
SQL JMS	`sqldms:/myQueues/alertsQueue`
IBM WebSphere Application Server	`jms/myQueues/alertsQueue`
Oracle WebLogic Server	`myQueues.alertsQueue`

Using Messaging Ports

In the Patch Bay configuration, a component can be configured to send its messages to a destination (or group of destinations), or to receive its messages from a destination (or group of destinations). Sometimes, however, you might want a component to have more control over where its messages are going. For example, a message filter might read in a message and then resend that message to one of several outputs based on some aspect of the message, such as its `JMSType`. Each of those outputs are then configured in Patch Bay to go to a separate set of destinations.

In Patch Bay, those outputs are called ports. The author of a messaging component chooses the names of the ports that are used by that component. Whenever a message source (or filter) sends a message, it must specify the name of the port through which the message is sent. This means that the port names used by the component are hard-coded into the component.

In Patch Bay, each of a component's output ports can be attached to a different set of destinations. For example:

```
<message-source>
  <nucleus-name>
    /atg/dynamo/j2ee/examples/TestMessageSource1
  </nucleus-name>

  <output-port>
    <port-name>
      Normal
    </port-name>

    <output-destination>
      <provider-name>
        local
      </provider-name>
      <destination-name>
        localdms:/local/NormalMessages
      </destination-name>
      <destination-type>
         Topic
      </destination-type>
    </output-destination>

  </output-port>

  <output-port>
    <port-name>
      Emergency
    </port-name>

    <output-destination>
      <provider-name>
        local
      </provider-name>
      <destination-name>
        localdms:/local/EmergencyMessages
      </destination-name>
      <destination-type>
        Topic
      </destination-type>
    </output-destination>
  </output-port>
</message-source>
```

In this example, it is assumed that `TestMessageSource1` is sending messages through at least two ports: Normal and Emergency. Patch Bay then directs messages coming out of those two ports to different destinations:

- Normal messages go to `localdms:/local/NormalMessages`.
- Emergency messages go to `localdms:/local/EmergencyMessages`.

If `TestMessageSource1` sends a message through some other port name, that message goes nowhere.

A `MessageSource` must be coded to specify which port it wants a message to use. The port is specified in both the `createMessage` and `sendMessage` methods. For example, this sends a `TextMessage` through the Normal port.

```
public void sendOneMessage ()
  throws JMSException
{
  if (mStarted && mContext != null) {
    TextMessage msg = mContext.createTextMessage ("Normal");
    msg.setJMSType ("atg.test.Test1");
    msg.setText ("Test text string");
    mContext.sendMessage ("Normal", msg);
  }
}
```

Notice that the message source does not need to declare what ports it uses. It just sends a message out using a name, and if Patch Bay has destinations hooked up to that name, the message is sent to those destinations. It is the responsibility of the message source developer to provide documentation as to what output ports it uses and in what situations.

Message sinks can also make use of ports. Whenever a message is received, the `receiveMessage` method passes in the name of the port through which the message arrived. For example, the DMS configuration might look something like this:

```
<message-sink>
  <nucleus-name>
    /atg/dynamo/j2ee/examples/TestMessageSink1
  </nucleus-name>

  <input-port>
    <port-name>
      LowPriority
    </port-name>

    <input-destination>
      <provider-name>
        local
      </provider-name>
      <destination-name>
        localdms:/local/TestMessages
      </destination-name>
      <destination-type>
        Topic
      </destination-type>
    </input-destination>

  </input-port>

  <input-port>
    <port-name>
      HighPriority
    </port-name>

    <input-destination>
      <provider-name>
        sqldms
      </provider-name>
      <destination-name>
        sqldms:/PersistentTopic1
      </destination-name>
      <destination-type>
        Topic
      </destination-type>
      <durable-subscriber-name>
        testMessageSink1
      </durable-subscriber-name>
    </input-destination>

  </input-port>

</message-sink>
```

If a message arrives from `localdms:/local/TestMessages`, the `receiveMessage` method is passed `LowPriority` as the name of the port. But if a message arrives from `sqldms:/PersistentTopic1`, the `receiveMessage` methods are passed `HighPriority`. An input port can have many input destinations. If a message arrives from any of those destinations, it is passed in with the name of its associated input port. Again, the message sink need not declare what ports it uses. However, the message sink developer should document what port names the message sink expects to see.

Ports provide another level of flexibility available through Patch Bay, but they should be used with care because they push some of the hookup responsibility into the messaging component code. Many of the functions provided by ports can be provided by other means, such as using different `JMSTypes`. The vast majority of message sources and sinks use only one output or input port. Use of multiple ports should be kept to a minimum, perhaps restricted to special general-purpose components such as multiplexers/demultiplexers or other message distribution components that really require them.

Using the Default Port

If a message source uses only one output port, that port should be called `DEFAULT`. The same is true for message sinks that use one input port. This is illustrated in the examples in the Connecting to Destinations section.

Note: If you use the `DEFAULT` port name, you can omit the `port-name` tag from the Patch Bay configuration file, because the default value for this tag is `DEFAULT`.

The `createMessage` and `sendMessage` methods also default to using `DEFAULT` for the port name. For example:

```
MessageSourceContext.createTextMessage()
```

is equivalent to

```
MessageSourceContext.createTextMessage("DEFAULT")
```

and

```
MessageSourceContext.sendMessage(pMessage)
```

is equivalent to

```
MessageSourceContext.sendMessage("DEFAULT",pMessage)
```

Using the Message Registry

DMS and Patch Bay make no assumptions about what kinds of messages flow through the various destinations. However, it is often useful to document what kinds of messages are used in the system, and what data is associated with those messages.

Patch Bay provides a Message Registry, which is a facility that maps message types to the data carried by those message types. The data in the Message Registry can then be accessed at runtime through a set of APIs. Application construction tools, such as the ATG Control Center, make use of this data to present lists of available message types or types of data associated with each message type.

The Message Registry works only for ATG messages—that is, messages that adhere to the conditions specified in the ATG Message Conventions section of this chapter. The important points are:

- The messages are identified by `JMSType`.
- They are `ObjectMessages`.
- The object in the message is a bean whose class is always the same for a given `JMSType`.

The Message Registry maps the `JMSType` string to the class of bean held by messages of that type. For example, messages with `JMSType` `atg.dcs.Purchase` are `ObjectMessages` containing objects of type `atg.dcs.messages.PurchaseMessage`.

Because there can be many message types in a large system, the Message Registry allows these message types to be grouped into message families. A message family is simply a group of message types that is given a name. For example, each application probably defines its own message family. A message family can itself contain message families, further subdividing the list of message types used by the application.

All of this is declared in the DMS configuration file:

```
<dynamo-message-system>
  <patchbay>
    ...
  </patchbay>
  <local-jms>
    ...
  </local-jms>

  <message-registry>

    <message-family>
      <message-family-name>
        Commerce
      </message-family-name>

      <message-type>
        <jms-type>
          atg.dcs.Purchase
        </jms-type>
        <message-class>
          atg.dcs.messages.PurchaseMessage
        </message-class>
      </message-type>

    </message-family>

  </message-registry>
</dynamo-message-system>
```

This declares a message family named Commerce, which contains a single declared message type. The message is identified by `JMSType` `atg.dcs.Purchase`, and contains objects of type `atg.dcs.messages.PurchaseMessage`. The Commerce family might have subfamilies:

```
<dynamo-message-system>
  <patchbay>
    ...
  </patchbay>
  <local-jms>
    ...
  </local-jms>
  <message-registry>

    <message-family>
      <message-family-name>
        Commerce
      </message-family-name>

      <message-family>
        <message-family-name>
          Purchasing
        </message-family-name>

        <message-type>
          ...
        </message-type>
        ...
      </message-family>

      <message-family>
        <message-family-name>
          CustomerService
        </message-family-name>

        <message-type>
          ...
        </message-type>
        ...
      </message-family>

      <message-family>
        <message-family-name>
          CatalogManagement
        </message-family-name>

        <message-type>
          ...
        </message-type>
        ...
      </message-family>

    </message-family>

  </message-registry>
</dynamo-message-system>
```

These declarations and subdivisions have no effect on how these messages are handled by the messaging system. They only affect the way that tools see these lists. Tools access these lists through the interfaces in the `atg.dms.registry` package: `MessageRegistry`, `MessageFamily`, and `MessageType`. The

MessagingManager component implements the MessageRegistry interface, which exposes the list of MessageFamily objects and searches for a MessageType by a particular MessageType name. Each MessageFamily then exposes its name, the list of MessageFamilies that it holds in turn, and the list of MessageTypes it holds.

Dynamic Message Types

One purpose of the Message Registry is to provide metadata about the expected dynamic beans properties of messages, in the form of a DynamicBeanInfo associated with each MessageType. In most cases, the properties of an object message can be determined purely by analyzing its object's Java class (that is, the class specified by the <message-class> element in the Patch Bay configuration file). The Message Registry does this automatically, by default.

However, in some cases properties might need to be determined dynamically from the application environment. A typical case of this is a message with a property of type atg.repository.RepositoryItem; in advance of actually receiving a message, this item's subproperties can only be determined by locating the appropriate repository within the application and examining its atg.repository.RepositoryItemDescriptor.

To handle this case, the Message Registry includes a facility for dynamic message typing. The optional <message-typer> element can be included immediately following a <message-class> element. It must specify a Nucleus component by use of a child <nucleus-name> element. The component, in turn, must implement the interface atg.dms.registry.MessageTyper; for each message that references the message typer, the typer's getBeanInfo() method is called with the message's name and class to determine that message's DynamicBeanInfo.

Here is an imaginary example:

```
<message-type>
  <jms-type>
    myproject.auction.BidMessage
  </jms-type>
  <message-class>
    myproject.jms.auction.BidMessage
  </message-class>
  <message-typer>
    <nucleus-name>
      /myproject/messaging/MessageTyper
    </nucleus-name>
  </message-typer>
  <message-context>
    session
  </message-context>
  <display-name>
    Bid on an item
  </display-name>
  <description>
     Message sent when someone bids on a repository item.
  </description>
</message-type>
```

The `MessageTyper` interface includes a single method:

```
public interface MessageTyper
{
  //-------------------------------------
  /**
   * Returns the DynamicBeanInfo associated with a JMS message type
   * and optional message object class.  If a class is provided, the
   * MessageTyper can expect that it is the class to which an object
   * message of this type will belong, and can introspect it to
   * determine the non-dynamic portion of the message metadata.
   *
   * @param pJMSType the JMS message type, which is required
   * @param pMessageClass an optional class which will be used at
   * runtime for an object message.
   **/
  public DynamicBeanInfo getBeanInfo (String pJMSType, Class pMessageClass);
}
```

A typical implementation of this interface might analyze the class to determine a basic `DynamicBeanInfo` by calling `DynamicBeans.getBeanInfoFromType(pMessageClass)`, and then return a `DynamicBeanInfo` that overlays the class-based metadata with dynamically determined metadata.

Delaying the Delivery of Messages

Patch Bay includes a feature that lets you delay the delivery of a message until a specific time. To support this behavior, Patch Bay uses a class called `atg.dms.patchbay.MessageLimbo` that receives messages that are marked for delayed delivery, stores them in database tables until the specified delivery time, and then sends them to their intended destinations. The delivery time for a message can be specified by inserting a property in the header of the message. The name of this property is stored in the `MessageLimbo` `DELIVERY_DATE` field, and its value should be a `Long` `datetime`, specified as UTC milliseconds from the epoch start (1 January 1970 0:00 UTC).

For example, the following code creates an SQL JMS message and specifies that it should not be delivered until one hour has passed:

```
Message m = (Message) qs.createMessage();
long hourInMillis = 1000 * 60 * 60;
long now = System.currentTimeMillis();
Long deliveryDate = new Long(now + hourInMillis);
m.setObjectProperty(atg.dms.patchbay.MessageLimbo.DELIVERY_DATE, deliveryDate);
```

If you require delivery on a specific date and not just a time offset, you can use the Java `Date` or `Calendar` class to produce the UTC milliseconds for the specific delivery date.

Note: The message cannot be delivered any sooner than the specified time, but there is no guarantee how much later the delivery actually takes place.

Configuring Delayed Delivery

The following are key properties of the `MessagingManager` component that configure the delayed delivery feature:

- `allowMessageDelays`: If `true` (the default), delayed delivery is enabled. If `false`, delayed delivery is disabled, and the message's delivery time is ignored.
- `limboSchedule`: Controls how often the `MessageLimbo` component polls the database for messages that are ready to be delivered. Default is once per minute.
- `limboDeliveryRetry`: Number of times the `MessageLimbo` can attempt to deliver the message to its destination. For example, if this value is 2 and the first delivery attempt fails, the `MessageLimbo` attempts one more delivery. Default is 1, which means that only one attempt is made, and if it fails, the message is discarded.

Configuring Failed Message Redelivery

JMS can work with the Java Transaction API (JTA) to provide transaction management for messaging. When a transaction manager creates a transaction, resources such as JMS destinations can be enlisted with the transaction. When the application is done processing the data within the transaction, it can ask the transaction manager to commit the transaction. When this occurs, the transaction manager asks each of the resources if it can commit the changes made. If all resources claim they can commit the changes, the transaction manager asks all resources to commit their changes. If a resource claims it cannot commit its changes, the transaction manager directs the resources to undo any changes made. The application can also set the transaction to rollback only mode, which forces the transaction manager to roll back the transaction.

For each JMS input destination, Patch Bay creates a thread that continuously loops through a cycle of beginning a transaction, receiving a message from the destination, and calling the configured message sink, and ending the transaction. If Patch Bay attempts to deliver a message to a message sink and an error condition arises (such as violation of a database constraint), the transaction is rolled back. The message remains in the destination, as if it were never delivered. Patch Bay tries to redeliver the message.

If the failed delivery is the result of some temporary condition, Patch Bay successfully delivers the message in a subsequent attempt. However, in some cases, the exception is caused by a problem with the message itself. This can result in an infinite loop, where the message delivery fails and the transaction is rolled back, and then Patch Bay continually tries to redeliver the message, and each time the delivery fails and the transaction is rolled back.

To avoid this situation, you can configure a message sink (or filter) so that only a certain number of attempts can be made to deliver a message to it. For example:

```
<message-sink>
  <nucleus-name>/fulfillment/OrderFulfiller</nucleus-name>
  <input-port>
    <input-destination>
      <destination-name>patchbay:/Fulfillment/SubmitOrder</destination-name>
      <destination-type>Topic</destination-type>
      <durable-subscriber-name>
        OrderFulfiller-SubmitOrder
      </durable-subscriber-name>
      <redelivery>
       <max-attempts>3</max-attempts>
       <delay>60000</delay>
       <failure-output-port>FulfillmentError</failure-output-port>
      </redelivery>
    </input-destination>
  </input-port>
  <redelivery-port>
    <port-name>FulfillmentError</port-name>
    <output-destination>
      <destination-name>patchbay:/Fulfillment/ErrorNotification</destination-name>
      <destination-type>Queue</destination-type>
    </output-destination>
  </redelivery-port>
</message-sink>
```

In this example, the message sink is configured so that Patch Bay makes a maximum of 3 attempts to deliver a message to it. The delay between each attempt is set to 60,000 milliseconds (10 minutes), which allows time for any transient errors responsible for a failed delivery to resolve themselves. This example also configures a destination to direct the message to if the delivery fails 3 times.

The redelivery port can define multiple destinations. The following example defines a second destination that has no components listening to it to act as a Dead Message queue. This allows the message to be kept in a JMS delivery engine waiting for eventual future delivery. After the source for the error is resolved, an administrator can use tools provided by the JMS provider to move the message back to the correct destination so that the message can be properly processed.

```
<redelivery-port>
  <port-name>FulfillmentError</port-name>
  <output-destination>
    <destination-name>patchbay:/Fulfillment/ErrorNotification</destination-name>
    <destination-type>Queue</destination-type>
  </output-destination>
  <output-destination>
    <destination-name>patchbay:/Fulfillment/DeadMessageQueue</destination-name>
    <destination-type>Queue</destination-type>
  </output-destination>
</redelivery-port>
```

Failed Message Redelivery and the MessageLimbo Service

The failed message redelivery system uses the `MessageLimbo` service discussed in the section Delaying the Delivery of Messages. There are thus two different types of messages handled by the `MessageLimbo` service:

- Delayed messages that are not yet published to a JMS destination
- Messages that were published to a destination, but were not successfully delivered to a message sink

These two types of messages are stored in the same set of tables, except that messages stored for redelivery have entries in one additional table named `dms_limbo_delay`. See Appendix B: DAF Database Schema for more information about these tables.

The maximum number of attempts to redeliver a failed message is set in the Patch Bay configuration file, using the `max-attempts` tag, as shown above. After this number of attempts, if the message still has not been successfully delivered to a message sink, Patch Bay creates a new message object and copies the message properties and message body into the new message. Patch Bay then attempts to publish the new message to the failure destination configured in the redelivery port.

If, due to some error condition, the new message cannot be published to the failure destination, the `MessageLimbo` service makes further attempts to publish the message. The maximum number of attempts is set by the `limboDeliveryRetry` property of the `MessagingManager` component. The default value of this property is 5, so after 5 attempts to publish the message, no further attempts are made, and the message remains in the database tables used by `MessageLimbo`. If the error condition is subsequently resolved, a database administrator can issue SQL statement to reset the counter on the message so the message is published. For example, the following SQL statement resets the counter for all messages being handled by the `MessageLimbo` service:

```
UPDATE dms_limbo_msg SET delivery_count=1
```

Using Patch Bay with Other JMS Providers

Patch Bay comes preconfigured to use Local JMS and SQL JMS. This ensures that ATG products and demos work without any further configuration. If desired, you can configure Patch Bay to use your application server's JMS provider or a third-party JMS provider, through standard Patch Bay tags. Consult the provider's documentation to determine the values to use.

In addition, you must add the client libraries for the JMS provider to ATG's `CLASSPATH`. See the *ATG Installation and Configuration Guide* for information about modifying the ATG `CLASSPATH`.

The DMS configuration file must declare all providers before it declares message sources and sinks. For example:

```
<?xml version="1.0" ?>

<dynamo-message-system>
```

```
  <patchbay>

    <provider>

      <provider-name>
        companyMessaging
      </provider-name>

      <topic-connection-factory-name>
        /newProvider/TopicConnectionFactory
      </topic-connection-factory-name>
      <queue-connection-factory-name>
        /newProvider/QueueConnectionFactory
      </queue-connection-factory-name>
      <xa-topic-connection-factory-name>
        /newProvider/XATopicConnectionFactory
      </xa-topic-connection-factory-name>
      <xa-queue-connection-factory-name>
        /newProvider/XAQueueConnectionFactory
      </xa-queue-connection-factory-name>
      <supports-transactions>
        true
      </supports-transactions>
      <supports-xa-transactions>
        true
      </supports-xa-transactions>
      <username>
        someUser
      </username>
      <password>
        somePassword
      </password>
      <client-id>
        local
      </client-id>
      <initial-context-factory>
        /myApp/jms/InitialContextFactory
      </initial-context-factory>

    </provider>

    <message-source>
      ...
    </message-source>
    ...

  </patchbay>
</dynamo-message-system>
```

If you specify multiple providers, each must have a unique provider-name.

Each `<provider>` tag can supply the following fields:

provider-name

Required, identifies the provider. The field can have any value that is unique among other providers in the system.

When message sources and sinks define input and output destinations, those destinations are associated with a provider name. This provider name must match the provider name declared for the provider that is handling a particular destination.

topic-connection-factory-name
queue-connection-factory-name
xa-topic-connection-factory-name
xa-queue-connection-factory-name

A JMS provider is accessed through `ConnectionFactories` that are identified by JNDI names., as specified by the JMS provider documentation.

Some of these fields might be optional. For example, JMS providers that do not support XA do not require `xa-topic-connection-factory-name` and `xa-queue-connection-factory-name`.

supports-transactions

Set to `true` or `false`, to specify whether the JMS provider supports `commit()` and `rollback()`.

supports-xa-transactions

Set to `true` or `false`, to specify whether the JMS provider supports XA transactions.

Note: If this field is set to `true`, you must also set `xa-topic-connection-factory-name` and `xa-queue-connection-factory-name`.

username
password

Many JMS providers require that clients log in to the JMS system using a username and password. If these fields are defined, their values are used to log in when creating JMS connections.

client-id

Many JMS providers have a notion of a client identifier, which allows the provider to remember who a client is even if that client is disconnected and later reconnects. This allows the JMS provider to queue up messages for the client while the client is disconnected. When the client reconnects, it uses the same client identifier it had previously, and the JMS provider knows to deliver the messages queued up for that client.

This field is optional, but it should be filled in if there are multiple clients running against the same JMS provider. In this case, each client should be assigned a unique `client-id`.

initial-context-factory

JNDI names are used to identify the connection factories and the topics and queues managed by a provider. These JNDI names are resolved against an `InitialContext`. Each provider obtains the `InitialContext` in its own way, as described by its documentation. Typically, a Dictionary is created with several properties, and is passed to the `InitialContext`'s constructor.

For example, a JMS provider might say that the `InitialContext` must be created using this code:

```
Hashtable h = new Hashtable ();
h.put (Context.INITIAL_CONTEXT_FACTORY, "...");
h.put (Context.PROVIDER_URL, "...");
...

Context ctx = new InitialContext (h);
```

In order for Patch Bay to create the `InitialContext` as required by the provider, this code must be packaged into a Nucleus component, and the name of the Nucleus component must be supplied as the `initial-context-factory`.

The Nucleus component must implement the interface `atg.dms.patchbay.JMSInitialContextFactory`, which defines a single method `createInitialContext()`. Patch Bay calls this method to get the Context that it uses to resolve a JNDI name.

The code for that Nucleus component might look like this:

```
import javax.naming.*;
import atg.dms.patchbay.*;

public class MyInitialContextFactory
  implements JMSInitialContextFactory
{
  public Context createInitialContext (String pProviderName,
                                String pUsername,
                                String pPassword,
                                String pClientId)
    throws NamingException
  {
      Hashtable h = new Hashtable ();
      h.put (Context.INITIAL_CONTEXT_FACTORY, "...");
      h.put (Context.PROVIDER_URL, "...");
      ...

    return new InitialContext (h);
  }
}
```

The arguments passed to `createInitialContext` are taken from the provider's other configuration values. Some JMS providers might need this information when creating the `InitialContext`.

This Nucleus component must be placed somewhere in the Nucleus hierarchy and its full Nucleus name must be supplied to the `initial-context-factory`.

13 Transaction Management

Transaction management is one of the most important infrastructure services in an application server. Because nearly all Internet applications access some sort of transactional database through JDBC, ATG developers need to understand how transaction management is handled by a J2EE application server, how transactions affect the behavior of applications, and how applications should be written to cooperate with the transaction system.

In this chapter

This chapter includes the following sections:

Transaction Overview

Most developers are familiar with the concept of a transaction. In its simplest definition, a transaction is a set of actions that is treated as an atomic unit; either all actions take place (the transaction commits), or none of them take place (the transaction rolls back).

A classic example is a transfer from one bank account to another. The transfer requires two separate actions. An amount is debited from one account, then credited to another account. It is unacceptable for one of these actions to take place without the other. If the system fails, both actions must be rolled back, even if the system failed in between the two actions. This means that both actions must take place within the same transaction.

Within an application server, transaction management is a complex task, because a single request might require several actions to be completed within the same transaction. A typical J2EE request can pass through many components—for example, servlets, JSPs, and EJBs. If the application is responsible for managing the transaction, it must ensure that the same transactional resource (typically a JDBC connection) is passed to all of those components. If more than one transactional resource is involved in the transaction, the problem becomes even more complex.

Fortunately, managing transactions is one of the primary tasks of an application server. The application server keeps track of transactions, remembering which transaction is associated with which request, and what transactional resources (such as JDBC or JMS connection) are involved. The application server takes care of committing those resources when the transaction ends.

As a result, transactional programming is much simpler for applications. Ideally, the application components do not need to be aware that transactions are used at all. When an application component needs to access a database, it just asks the application server for a JDBC connection, performs its work, then closes the connection. It is the application server's responsibility to make sure that all components involved in a request get the same connection, even though each component is coded to open and close its own separate connection. The application server does this behind the scenes, by mapping threads to transactions and transactions to connections.

When the application has completed the set of operations, it can commit the transaction that it created. It is the application server's responsibility to know which JDBC connections were used while that transaction was in place, and to commit those connections as a result.

Transactions are often associated with requests, but they can be associated with other sequences of actions performed in a single thread. For example, the ATG scheduler is used to notify components to perform some kind of action at specified times. When a component receives a notification, it can start a transaction, thus ensuring that its operations are treated as an a unit. When the component has completed its work, it can commit the transaction, thereby committing all of these operations.

Transaction Manager

Each active transaction is represented by a transaction object, which implements the interface `javax.transaction.Transaction`. This object keeps track of its own status, indicating if it is active, if it has been committed or rolled back, and so on. The transaction also keeps track of the resources that were enlisted with it, such as JDBC connections. A transaction object lasts for the space of exactly one transaction—when the transaction begins, a new transaction object is created. After the transaction ends, the transaction object is discarded.

A transaction is usually associated with a thread, which is how a transaction appears to be carried along throughout the duration of a request or other sequence of actions. Only one transaction can be associated with a thread at any one time. This leads to the notion of the current transaction, which is the transaction that is currently associated with a thread.

An application server can have many active transactions at once, each associated with a different thread running in the server. A central service, called the Transaction Manager, is responsible for keeping track of all these transactions, and for remembering which transaction is associated with which thread. When a transaction is started, the Transaction Manager associates it with the appropriate thread. When a transaction ends, the Transaction Manager dissociates it from its thread.

The Transaction Manager is implemented through the Java Transaction API (JTA). The JTA includes two main interfaces for managing transactions, `javax.transaction.TransactionManager` and `javax.transaction.UserTransaction`.

The `TransactionManager` interface is intended to be used by the application server, and it provides a full range of methods for managing transactions. It allows transactions to be created, suspended, resumed, committed, and rolled back. It also provides direct access to the javax.transaction.Transaction object, through which synchronizations can be registered and resources can be enlisted.

In ATG applications, the TransactionManager object is represented by a Nucleus component, /atg/dynamo/transaction/TransactionManager. Depending on what application server you are running, this component is configured in various ways to point to the appropriate TransactionManager implementation. See the *ATG Installation and Configuration Guide* for information about how the Nucleus TransactionManager component is configured on your application server.

The TransactionManager object keeps track of the transactions running in ATG applications, and which threads are associated with which transactions. Nucleus components can get a pointer directly to the /atg/dynamo/transaction/TransactionManager component. ATG also exposes this component to standard J2EE components, such as servlets and EJBs, through the JNDI name dynamo:/atg/dynamo/transaction/TransactionManager. However, it is not a standard practice in J2EE for an application to access the TransactionManager interface directly.

J2EE applications instead use the UserTransaction interface, which provides a subset of the methods in the TransactionManager interface. A UserTransaction object can begin, commit, and rollback transactions, set the rollback-only flag, and examine the status of the current transaction, but it cannot suspend or resume transactions, or directly access a Transaction object. The UserTransaction object is represented in ATG by the Nucleus component /atg/dynamo/transaction/UserTransaction. This component actually delegates all of its calls to the /atg/dynamo/transaction/TransactionManager component, but limits those calls to the methods that are part of the UserTransaction interface.

The methods of both interfaces always operate in the context of the calling thread. For example, the begin method creates a new Transaction and associates it with the calling thread, or throws an exception if there is already a Transaction associated with the calling thread.

Accessing the UserTransaction Interface

The UserTransaction interface is typically accessed by EJBs that use bean-managed transactions (BMTs). An EJB generally uses BMTs if it needs to start and end several transactions within the space of a single method. Other J2EE components, such as servlets and JSP tags, can also access the UserTransaction object. However, there are some restrictions; for example, EJBs using container-managed transactions cannot access the UserTransaction object. The full set of restrictions is found in the EJB specification.

The standard way for a J2EE application to get a pointer to the UserTransaction object is by resolving the JNDI name java:comp/UserTransaction. To ensure that this reference works properly, you should configure the Nucleus component /atg/dynamo/transaction/TransactionManager to point to the appropriate class. For information about how to do this, see the *ATG Installation and Configuration Guide*.

Working with Transactions

Although the application server handles most of the transaction management for your applications, there are times when an application needs to control or influence how transactions are handled. This section discusses how to configure applications (either in deployment descriptors or code) to work with J2EE transaction management, including:

- Resource Access and Enlistment
- Transaction Completion
- Transaction Synchronization
- Marking Rollback Only
- Transaction Suspension

Resource Access and Enlistment

The transaction objects maintained by the Transaction Manager do not do any actual transactional work. For example, a transaction object does not know how to commit or rollback changes. Instead, the transaction object is responsible for coordinating these actions in the data storage devices that do know how to commit and rollback. When a transaction object is committed or rolled back, the transaction object passes that request on to the data storage devices responsible for carrying out that actual work.

In order for a transaction to keep track of all the resources used during the transaction's lifetime, those resources must be enlisted with the transaction. At the API level, resource enlistment is somewhat complicated. The resource connection to be enlisted must be able to produce an `XAResource` object, which is then enlisted into the Transaction object associated with the current thread.

Fortunately, resource enlistment is the job of the application server, not the application. When the application asks for a resource connection, the application server takes care of enlisting the connection with the current transaction before returning the connection to the application.

However, this means that applications must obtain resource connections in a manner that cooperates with this process. The model established by J2EE uses a combination of JNDI and resource connection factories. A connection factory is an object supplied by the application server that produces connections of the appropriate type. The interfaces for these connection factories are defined by Java standards—for example, JDBC connections are produced by `javax.sql.DataSource` objects, while JMS connections are produced by `javax.jms.TopicConnectionFactory` or `javax.jms.QueueConnectionFactory` objects.

These factory objects are available in ATG as Nucleus services. For example, the standard ATG `DataSource` object is found at `/atg/dynamo/service/jdbc/JTDataSource`. New resource factories can be added as needed by creating them like any other new Nucleus service.

Nucleus components should acquire resources through the proper connection factory services, rather than accessing drivers directly from their managers. This allows ATG applications containing both Nucleus components and standard J2EE components to interoperate.

An application can enlist multiple resources over the course of a single transaction. For example, an application might read a JMS message, then write a resulting database row through a JDBC connection. Both resources are enlisted into the same transaction, even if the resources were enlisted by the same or different components. At the end of the transaction, both resources are committed, as described below.

An application might use the same resource several times over the course of a transaction, perhaps through multiple disparate components. For example, a request might call an EJB that uses JDBC to perform a database operation, then call a second EJB that also uses JDBC. Each usage of the resource should go through the entire sequence outlined previously: use JNDI to get a pointer to the resource factory, acquire a connection from the factory, then close the connection when finished. The application should not attempt to acquire the resource once and pass it around from component to component in the interest of avoiding the code for acquiring or closing the connection.

The application server does what is necessary to make sure that the connection returned to each component refers to the same transaction. For JDBC drivers, this means that the same `Connection` object must be returned each time a connection is requested throughout a single transaction. (JDBC 2.0 drivers that support XA are not bound by this limitation.) The application server does this by maintaining an internal table mapping transactions to JDBC connections. When a component requests a JDBC connection, the server consults this table to see if a connection is already associated with the current transaction and if so, returns that connection. Otherwise, a new connection is checked out of the connection pool, and remains associated with the current transaction so that further requests for connections return the same `Connection` object.

Application components are required to close JDBC connections when they finish doing their individual portion of work. This might seem odd, especially if other components use the same connection later in the request. However, rather than actually closing the connection to the database, the application server intercepts these close requests and interprets them as signals from the application that it is done with the connection for the time being. The application server then responds to that signal by returning the connection to a pool, or by maintaining the connection's transactional association.

This means that each individual component should be written as if it were the only component in the request that needs to access the database. The component should also be written without regard for how the connection is being managed. The same code should be used regardless of whether connections are being pooled or not, or whether XA connections are supported or not. These are all permutations that the application server supports—the application does not need to consider any of this in its compiled code.

Transaction Completion

All transactions eventually end, either in a commit or a rollback. If a transaction commits, all work done through the resources enlisted over the course of that transaction is made permanent and visible to other transactions. If a transaction rolls back, none of the work done through any enlisted resources is made permanent.

If a single resource has been enlisted with the transaction, the commit or rollback result is passed directly to the resource. This is the most common case, because most applications make use of a single database and communicate with no other transactional resources.

If multiple resources were enlisted with the transaction, such as two database connections or a database connection and a JMS connection, a two-phase commit must be used to end the transaction. A two-phase commit is comprised of two stages, prepare and commit:

- **prepare**: The transaction instructs each resource to prepare itself for a commit. Each resource prepares by evaluating whether a commit succeeds or not, and responds with a vote to commit or roll back. If any resource responds with a rollback during the prepare phase, all resources are immediately rolled back and the transaction ends with a rollback. If a resource votes to commit, that resource must ensure that it can commit its work, even if a system failure occurs before the commit occurs.

- **commit** : If all resources vote to commit, the transaction instructs each resource to commit. Resources cannot roll back at this point.

After a transaction commits or rolls back, it ends and is dissociated from its thread, leaving the thread without a transaction.

Simulating Two-Phase Commit

A two-phase commit is much more complex than a commit involving a single resource. Not only is it more complex for the application server, but the resources themselves must be fairly advanced to be able to ensure that they can commit their work even if the system fails. As it turns out, few databases support this ability, and even fewer JDBC drivers include this support (sometimes called *XA support*). As a result, very few applications make use of multiple resources at once.

Resources can simulate two-phase behavior, even if they do not inherently support two-phase commits. This allows JDBC drivers that do not support the two-phase commit protocol to work with the application server’s two-phase commit mechanism. A resource can simulate the two-phase protocol by committing in the prepare phase, and ignoring the commit phase. If the commit succeeds, the resource votes to commit, otherwise the resource votes to rollback. The transaction can proceed as normal, using both resources that understand the two-phase commit protocol, and those that simulate it.

This works most of the time. In the majority of applications where only a single resource is involved, this technique works flawlessly. However, if a transaction involves multiple resources then there are instances where a resource might commit while the others roll back. If, during the prepare phase, the resource commits but then a subsequent resource votes to rollback, it is too late for the first resource to rollback, so there is an inconsistency.

Fortunately, these situations arise very rarely. Because of this, and because two-phase commits can cause performance problems, resources and drivers that support true two-phase commits are still fairly uncommon. In fact, the default configuration for an ATG application uses a JDBC driver configured to simulate two-phase commits. This driver should be sufficiently robust to handle the majority of applications.

Transaction Synchronization

The Java Transaction API includes a `javax.transaction.Synchronization` interface, which issues notifications before and after a transaction is completed. Objects implementing this interface can be registered with a Transaction object. Just before the transaction’s completion process begins, the `TransactionManager` calls the `Synchronization` object’s `beforeCompletion()` method. After the transaction is committed or rolled back, the `TransactionManager` calls the `Synchronization` object’s `afterCompletion()` method.

The `beforeCompletion()` method is usually used to perform any last-minute work. For example, an application might use this callback to write some built-up state to the database.

The `afterCompletion()` method is called after the commit or rollback, and passes in a status code indicating which of those outcomes occurred. Applications can use this callback to clean up any state or resources that were used during the transaction.

To register synchronizations directly, your code must use the `TransactionManager` to get a hold of the Transaction object. J2EE components do not have explicit access to the `TransactionManager` interface, so J2EE provides other ways for its components to receive synchronization callbacks. Specifically, stateful session EJBs can implement the `javax.ejb.SessionSynchronization` interface, which includes methods for receiving synchronization notifications.

Marking Rollback Only

As a result of an error condition or exception, an application can determine that the current transaction should be rolled back. However, the application should not attempt to rollback the transaction directly. Instead, it should mark the transaction for rollback only, which sets a flag on the transaction indicating that the transaction cannot be committed.

When the time comes to end the transaction, the application server checks to see if the transaction is marked for rollback only, and if so, rolls back the transaction. If the rollback-only flag is not set, the application server attempts to commit the transaction, which can result in a successful commit or in a rollback.

An application can also check whether a transaction has already been marked for rollback only. If so, the application should not attempt to enlist any further resources with the transaction. If a transaction has been marked for rollback only, each subsequent attempt to obtain resources results in an error. Checking for rollback only can eliminate some of these errors and make debugging easier.

Setting and getting the rollback-only flag can be performed using the `setRollbackOnly()` and `getStatus()` methods of the `UserTransaction` interface. J2EE provides other interfaces for implementing these capabilities in specific component types. For example, the `javax.ejb.EJBContext` interface provides `getRollbackOnly()` and `setRollbackOnly()` methods to EJBs.

Transaction Suspension

When a transaction is created, it is associated with the thread that created it. As long as the transaction is associated with the thread, no other transaction can be created for that thread.

Sometimes, however, it is helpful to use multiple transactions in a single set of actions. For example, suppose a request performs some database operations, and in the middle of those operations, it needs to generate an ID for a database row that it is about to insert. It generates the ID by incrementing a persistent value that it stores in a separate database table. The request continues to do some more database operations, then ends.

All of this can be done in a single transaction. However, there is a potential problem with placing the ID generation within that transaction. After the transaction accesses the row used to generate the ID, all other transactions are locked out of that row until the original transaction ends. If generating IDs is a central activity, the ID generation can end up being a bottleneck. If the transaction takes a long time to complete, the bottleneck can become a serious performance problem.

This problem can be avoided by placing the ID generation in its own transaction, so that the row is locked for as short a time as possible. But if the operations before and after the ID generation must all be in the same transaction, breaking up the operations into three separate transactions (before ID generation, ID generation, and after ID generation) is not an option.

The solution is to use the JTA's mechanism for suspending and resuming transactions. Suspending a transaction dissociates the transaction from its thread, leaving the thread without a current transaction. The transaction still exists and keeps track of the resources it has used so far, but any further work done by the thread does not use that transaction.

After the transaction is suspended, the thread can create a transaction. Any further work done by the thread, such as the generation of an ID, occurs in that new transaction.

The new transaction can end after the ID has been generated, thereby committing the changes made to the ID counter. After ending this transaction, the thread again has no current transaction. The previously suspended transaction can now be resumed, which means that the transaction is re-associated with the original thread. The request can then continue using the same transaction and resources that it was using before the ID generator was used.

The steps are as follows:

1. Suspend the current transaction before the ID generation.
2. Create a transaction to handle the ID generation.
3. End that transaction immediately after the ID generation
4. Resume the suspended transaction.

An application server can suspend and resume transactions through calls to the `TransactionManager` object; individual applications should not perform these operations directly. Instead, applications should use J2EE transaction demarcation facilities (described in the next section), and let the application server manage the underlying mechanics.

Transaction Demarcation

When using the J2EE transaction model, developers should not think in terms of starting and stopping transactions. Instead, developers should think about sections of sequential actions that should be enclosed in some sort of transactional behavior. This enclosing of transactional behavior is called transaction demarcation.

Transaction demarcation always wraps a sequence of actions, such as a single request, a single method, or a section of code within a method. The demarcation initializes some transactional behavior before the demarcated area begins, then ends that transactional behavior when the demarcated area ends. The application server uses these demarcations to determine the appropriate calls to the `TransactionManager` object.

Transaction Modes

The simplest form of transaction demarcation is to create a transaction at the beginning of the demarcated area, then end that transaction at the end of the demarcated area. However, there are several transaction demarcation modes, which are defined as follows:

Mode	Description
`Required`	Indicates that a transaction must be in place in the demarcated area. If a transaction is already in place in the area, nothing further is done. If no transaction is in place, one is created when the demarcated area is entered and ended when the demarcated area ends.
`RequiresNew`	Indicates that all activity within the demarcated area must occur in its own separate transaction. If no transaction is in place in the area, a transaction is created at the beginning of the demarcated area and ended at the end of the demarcated area. If a transaction is in place when the demarcated area is entered, that transaction is suspended, and a new transaction is begun; at the end of the demarcated area, the new transaction is ended, and the original transaction is resumed.
`NotSupported`	Indicates that a transaction must not be in place in the demarcated area. If no transaction is in place in the area, nothing further is done. If there is a transaction in place when the demarcated area is entered, that transaction is suspended, then resumed at the end of the demarcated area.
`Supports`	This mode does nothing. If a transaction is in place when the demarcated area is entered then that transaction remains in place. Otherwise, the area is executed without a transaction in place.
`Mandatory`	Throws an exception if a transaction is not in place when the demarcated area is entered. This mode does not create a transaction; it is used to verify that a transaction is in place where the developer expects.
`Never`	Throws an exception if there is a transaction in place when demarcated area is entered. This mode does not end or suspend any existing transactions; it is used to verify that a transaction is not in place where the developer does not expect one.

Declarative Demarcation

When using declarative demarcation, you specify what transaction demarcation modes should be used around certain areas of code. Rather than implementing these demarcations directly in your code, you declare the demarcations in a configuration file or deployment descriptor. The application server is then responsible for making sure that the correct transactional behavior is used around the specified area.

At present, declarative demarcations are used only for EJBs that use container-managed transactions (CMT). In the `ejb-jar.xml` deployment descriptor, you declare the `transaction-type` for the EJB as `container`, and for each method of the EJB, declare what transaction demarcation mode should be used (using the `container-transaction` and `trans-attribute` tags). The application server then makes sure that the declared transaction mode is enacted around the method call.

For example, if an EJB method is declared to have transaction demarcation mode `RequiresNew`, the application server suspends the current transaction and creates a new one before entering the method, then ends the new transaction and resumes the suspended transaction after exiting the method.

Demarcation in Pages

ATG's DSP tag libraries include several tags that you can use to demarcate transactions in JSPs:

- `dsp:beginTransaction` initiates a transaction and tracks its status.
- `dsp:commitTransaction` commits the current transaction.
- `dsp:demarcateTransaction` begins a transaction, executes one or more operations within the transaction, and then commits the transaction.
- `dsp:rollbackTransaction` rolls back the current transaction.
- `dsp:setTransactionRollbackOnly` marks the current transaction for rollback only.
- `dsp:transactionStatus` returns the status of the current transaction.

See the *ATG Page Developer's Guide* for more information about these tags.

Transaction Servlet Bean

In addition to the transaction handling tags in the DSP tag libraries, ATG has a servlet bean class, `atg.dtm.TransactionDroplet`, for demarcating transactions, and includes a Nucleus component of this class at `/atg/dynamo/transaction/droplet/Transaction`. For example:

```
<dsp:droplet name="/atg/dynamo/transaction/droplet/Transaction">
  <dsp:param name="transAttribute" value="requiresNew"/>
  <dsp:oparam name="output">

  ... portion of page executed in demarcated area ...

  </dsp:oparam>
</dsp:droplet>
```

In this particular example, the demarcated portion of the page executes in its own separate transaction, as specified by the `requiresNew` directive. The valid values for the `transAttribute` input parameter are `required`, `requiresNew`, `supports`, `notSupported`, `mandatory`, and `never`.

Ending Transactions Early

The transaction demarcation mechanisms, such as the `Transaction` servlet bean, take care of both creating and ending transactions. The application itself does not need to commit or rollback the transaction.

Sometimes, however, you might want to force the transaction to complete. This is usually done if the application needs to determine the outcome of the transaction before reaching the end of the demarcated area. For example, an entire page might be demarcated in a single transaction, meaning that the transaction ends after the page has been served to the user. This is a problem if the user needs to know that there was a problem ending the transaction, because by the time the transaction fails, it is too late to tell the user.

The solution is for the application to end the transaction before the end of the demarcated area. ATG has a servlet bean class, `atg.dtm.EndTransactionDroplet`, for ending transactions, and includes a Nucleus component of this class at `/atg/dynamo/transaction/droplet/EndTransaction`. For example:

```
<dsp:droplet name="/atg/dynamo/transaction/droplet/EndTransaction">
  <dsp:param name="op" value="commit"/>
  <dsp:oparam name="successOutput">
    The transaction ended successfully!
  </dsp:oparam>
  <dsp:oparam name="errorOutput">
    The transaction failed with reason:
    <dsp:valueof param="errorMessage"/>
  </dsp:oparam>
</dsp:droplet>
```

This causes the transaction to commit or rollback (according to the op parameter), and displays one of the two open parameters, depending on the outcome. The remainder of the page executes without any transaction context, so the page must not attempt to access any resources after ending the transaction (unless it demarcates that resource use with a new transaction demarcation).

For more information about including servlet beans and other Nucleus components in pages, and for more information about the `Transaction` and `EndTransaction` servlet beans, see the *ATG Page Developer's Guide*.

Programmatic Demarcation

At times, you might need to demarcate transactions in your code. Generally, you should use programmatic demarcation as little as possible, as it is error-prone and can interfere with the application server's own transaction demarcation mechanisms. If you find it necessary to use programmatic demarcation, you must be very careful to ensure that your code handles any unexpected errors and conditions.

The ATG platform includes two classes that you can use to demarcate transactions in code:

- `atg.dtm.UserTransactionDemarcation` can be used by J2EE components and Nucleus components to perform basic transaction demarcation. This class accesses the `UserTransaction` object to perform its operations.
- `atg.dtm.TransactionDemarcation` can be used by Nucleus components to demarcate areas of code at a fine granularity. J2EE components cannot use this class, because it accesses the `TransactionManager` object directly.

Using the UserTransactionDemarcation Class

The following example illustrates how to use the `UserTransactionDemarcation` class:

```
UserTransactionDemarcation td = new UserTransactionDemarcation ();
try {
  try {
    td.begin ();

    ... do transactional work ...
  }
  finally {
    td.end ();
  }
}
catch (TransactionDemarcationException exc) {
  ... handle the exception ...
}
```

There are a few things to note about using the `UserTransactionDemarcation` class:

- The `begin()` method implements the `REQUIRED` transaction mode only. If there is no transaction in place, it creates a new one; but if a transaction is already in place, that transaction is used.
- If `begin()` creates a new transaction, the `end()` method commits that transaction, unless it is marked for rollback only. In that case, `end()` rolls it back. However, if `begin()` does not create a transaction (because there is already a transaction in place), `end()` does nothing.
- The code must ensure that `end()` is always called, typically by using a `finally` block.
- `begin()` and `end()` can throw exceptions of class `atg.dtm.TransactionDemarcationException`. The calling code should log or handle these exceptions.

Using the TransactionDemarcation Class

The following example illustrates using the `TransactionDemarcation` class:

```
TransactionManager tm = ...
TransactionDemarcation td = new TransactionDemarcation ();
try {
  try {
    td.begin (tm, td.REQUIRED);

    ... do transactional work ...
  }
  finally {
    td.end ();
  }
}
catch (TransactionDemarcationException exc) {
  ... handle the exception ...
}
```

There are a few things to note about using the `TransactionDemarcation` class:

- The `begin()` method takes two arguments. The first argument is the `TransactionManager` object. The second argument specifies one of the 6 transaction modes: `REQUIRED`, `REQUIRES_NEW`, `SUPPORTS`, `NOT_SUPPORTED`, `MANDATORY`, or `NEVER`. If the second argument is not supplied, it defaults to `REQUIRED`.
- The code must ensure that the `end()` method is always called, typically by using a `finally` block.
- The `begin()` and `end()` methods can throw exceptions of class `atg.dtm.TransactionDemarcationException`. The calling code should log or handle these exceptions.

The `TransactionDemarcation` class takes care of both creating and ending transactions. For example, if the `TransactionDemarcation` object is used with a `RequiresNew` transaction mode, the `end()` call commits or rolls back the transaction created by the `begin()` call. The application is not expected to commit or rollback the transaction itself.

If for some reason the application needs to force the transaction to end, this can be done by calling the `TransactionManager.commit()` method:

```
TransactionManager tm = ...
TransactionDemarcation td = new TransactionDemarcation ();
try {
  try {
    td.begin (tm);

    ... do transactional work ...

    tm.commit ();
  }
  catch (RollbackException exc) { ... }
  catch (HeuristicMixedException exc) { ... }
  catch (HeuristicRollbackException exc) { ... }
  catch (SystemException exc) { ... }
  catch (SecurityException exc) { ... }
  catch (IllegalStateException exc) { ... }
  finally {
    td.end ();
  }
}
catch (TransactionDemarcationException exc) {
  ... handle the exception ...
}
```

Ending a transaction in this way should be avoided wherever possible, because handling all exceptions introduces a lot of complexity in the code. The same result can usually be accomplished by more standard means.

14 Managing Access Control

User account security is managed through the `atg.security` API. Using this API, you can manage persistent user accounts, look up user identities and associate them with roles, manage access control lists, and tie together multiple security systems running against the same user account database and/or authentication mechanisms.

The Security Services Interface is a set of fast, flexible APIs that you can use in an application to provide security for the application's features. The Security Management Interface enables programmers to configure account and privilege information with minimal programming.

In this chapter

This chapter covers the following topics:

- Security Services Classes and Interfaces: Outlines the main interfaces, objects and classes of the Security Services.
- Extending the Security Model: Provides examples of extending the default security model and authenticating a user.
- Configuring Access Privileges: Describes how to configure and restore ATG's default login accounts, and how to create accounts, groups, and privileges using the ATG Control Center.
- Configuring LDAP Repository Security: Describes how to configure an ATG application to use an LDAP repository to authenticate users and groups.

Security Services Classes and Interfaces

The main interfaces, objects, and classes for Security Services are as follows:

User Authority	This interface is used for authenticating a user. The interface produces Persona objects that are used to identify a user and any roles that the user might have.

Persona	Identity of a user, a user's role (for example, a user group such as Designers or Developers), or an application privilege. Persona objects can have multiple embedded identities. For example, a user can have several roles, such as manager and developer, and a role can have multiple privileges. The Persona interface is a superset of the standard J2EE Principal interface, and implements the Principal interface for interoperability.
User	The User object holds a collection of Personae that were collected by one or more user authorities. This object is like a wallet where identities are placed. A User object can hold several identities if a user has been authenticated by several means.
Security Policy	A security policy is used to determine whether a user has access to an object by checking an access control list composed of access privileges and/or deny privileges.
Secured Object	The `SecuredObject` interface provides a standard way to look up and change security information related to an object. The `atg.security.StandardSecurityPolicy` class uses this interface to determine the ACL for an object and any related container objects that might affect the ACL.
Secured Container	Like `SecuredObject`, `SecuredContainer` provides a standard interface for determining a list of security-related parents of an object, to support ACL inheritance or other cross-object semantics, for example.
Security Configuration	A security configuration is a security policy grouped together with the user authority that determines the identity information for a user. The security configuration is used primarily for reconstituting persisted ACL information using the `parse()` method of `atg.security.AccessControlList`.
Security Context	Every `SecuredObject` has a related `Security Context`, which is a Security Configuration plus a reference back to the object. This allows the access checker in the security policy to use the object itself to determine access control rules.

User Authority Object

The first contact that a user has with the security system is usually a user authority object, which determines who the user is. At its most basic, the user authority object simply provides a persona object for a user with a particular name.

ATG's central user authority object is in Nucleus at `/atg/dynamo/security/UserAuthority` and is an instance of the `UserDirectoryUserAuthority` class. This class takes the account information from one or more user directories and exposes it through the `UserAuthority` interface. In the standard configuration, both the ATG Control Center and Profile account information are exposed.

The user authority object also can be responsible for authenticating a user. How it does so depends on the implementation. Typically, a user authority authenticates users through name/password verification, but any sort of identification system is possible, including smart cards, certificates, biometrics, or even profiling—for example, a user can be granted or denied access based on responses to a questionnaire.

There are three user authorities that use the name/password verification approach:

- **XmlAccountManager:** This read-only implementation derives user information from an XML file. The implementation is intended for prototyping, although it can be useful in a production environment if the set of accounts and identities is not expected to change often or is expected to remain static. ATG uses an instance of the `XmlAccountManager` to provide a template for the ATG Control Center account information.

- **RepositoryAccountManager:** This implementation derives user information from an ATG repository. The repository can be any type of repository, including XML, SQL, and Profile Repositories. This implementation is for production applications, which typically use a repository-based user authority in conjunction with the Generic SQL Adapter (GSA) connector, which interfaces the Repository API to an SQL database. ATG uses an instance of the `RepositoryAccountManager` to manage the ATG Control Center accounts.

- **UserDirectoryLoginUserAuthority:** Because `UserDirectoryUserAuthority` can merge multiple account databases, the `UserDirectoryLoginUserAuthority` is used to expose the login functionality for only a single database (and, thus, account namespace). There are two such authorities: `/atg/dynamo/security/AdminUserAuthority` (for ATG Control Center account information) and `/atg/userprofiling/ProfileUserAuthority` (for profile accounts). ATG does not yet implement authentication mechanisms other than name/password verification, although it is easy to extend the `UserAuthority` interface as necessary to provide new authentication mechanisms.

All other security objects refer to the user authority to provide namespace separation between different authentication schemes. Two users with the same name (such as `peterk`) have two different identities to an ATG application if they are authenticated by two different user authorities. A single user authority often is shared by multiple security objects to obtain single-log-on functionality.

For more information about configuring the ATG User Directory, see the *ATG Personalization Programming Guide*.

User Object

The system passes around user identity information in a user object. This object is similar to a wallet and can contain more than one identity, just as a wallet can contain a driver's license, credit card, and ATM card. Identities are accumulated over the course of a session as a user becomes identified with various security systems.

A management interface, `atg.security.ThreadSecurityManager`, ties a user object to a particular thread and temporarily assigns user objects to a thread. In this way, identity is associated with an execution context. ATG's request handling pipeline automatically associates the session's User object with the request thread, so calling the `ThreadSecurityManager.currentUser()` returns the user for the current session.

Persona Object

A discrete user identity is called a persona. A persona is more than just the identification of a particular user; it can also be the identity of a group or role or even an identity associated with a system privilege. Persona objects can be compound identities; a user often is a member of various groups or should have access to resources according to the roles the user holds in an organization. Typically, the user authority adds these identities as subpersonae.

Access Privileges

An access privilege is access control for a resource. For example, a file object might have read, write, and delete access privileges. The access privilege object implements the `atg.security.AccessRight` interface, which extends the `java.security.acl.Permission` interface.

Access Control Lists

Access to individual resources is controlled by an Access Control List (ACL). An ACL consists of identities and their access privileges. In the standard implementation of the security system, an ACL is a collection of access control entries, each of which associates a single persona with a set of access privileges. This object extends the `java.security.acl.Acl` interface. An access control entry object extends the `java.security.acl.AclEntry` interface.

For information the `AccessControlList` methods, see `atg.security.AccessControlList` in the *ATG API Reference*.

Security Policy Object

A security policy determines whether a user has access to a particular object. In an ATG application, the standard security policy is in Nucleus at `/atg/dynamo/security/SecurityPolicy`. This instance of the `atg.security.StandardSecurityPolicy` object provides the following policy:

- If no ACL is defined for an object, access is allowed.
- If the accessor is the owner of an object, access is allowed if the desired access privilege is `LIST`, `READ_ACL`, or `WRITE_ACL`. This approach makes the object's security information modifiable if the ACL become corrupted.
- If the ACL for the object has a `deny` (or negative) access privilege that applies to the user, access is denied even if other permissions are positive.
- If the ACL for the object has an `allow` (or positive) access privilege that applies to the user, access is allowed as long as there is not a corresponding deny.
- If no ACL entries apply to the user, access is denied.

Note: This policy differs slightly from the `java.security.acl` policy, where a combination of positive and negative ACL entries with the same Principal negate each other, providing no change to the access control for that Principal. This differentiation is deliberate; ATG believes that in no case should an explicit deny access control entry be ignored.

Extending the Security Model

This section provides two examples of extending the default security model and an example of authenticating a user:

- Extending the Standard Security Policy shows how to deny access if the access control list is null. The second example shows how to deny access except during specified hours.
- Authenticating a User defines a bean and associated form that presents a login form to a user until their login succeeds, then lists some details about the account they logged in with after the login is successful.

Extending the Standard Security Policy

You can extend the `StandardSecurityPolicy` to make the policy more flexible or tighter, depending on the needs of your application.

In the following example, access is denied if the access control list is null (unspecified):

```
public class DefaultDenySecurityPolicy
     extends StandardSecurityPolicy
{
  public int getAccess(AccessControlList pAcl,
                       Object pObject,
                       Persona pPersona,
                       AccessRight pRight,
                       boolean pExactPersona)
  {
    if (pAcl == null)
      return DENIED;
    else
      return super.getAccess(pAcl, pObject, pPersona, pRight, pExactPersona);
  }
}
```

In the following example, access is denied except during the hours of 9:00 to 5:00 in the default (local) time zone:

```
public class DenyOutsideBusinessHoursSecurityPolicy
     extends StandardSecurityPolicy
{
  public int getAccess(AccessControlList pAcl,
                       Object pObject,
                       Persona pPersona,
                       AccessRight pRight,
                       boolean pExactPersona)
  {
    Calendar calender = new GregorianCalendar(new Date());
    int hourOfDay = calendar.get(Calendar.HOUR_OF_DAY);
    if ((hourOfDay < 9) || (hourOfDay > 5))
      return DENIED;
    else
      return super.getAccess(pAcl, pObject, pPersona, pRight, pExactPersona);
  }
}
```

Authenticating a User

The following example defines a Bean and associated form that presents a login form to a user until the user's login succeeds, then lists some details about the account the user logged in with after the login is successful. This example illustrates the use of the `LoginUserAuthority` interface and some features of the Persona interface.

Authenticate Bean (Authenticate.java)

```
import java.io.*;
import javax.servlet.http.*;
import atg.security.*;
import atg.servlet.*;

/**
 * A bean that authenticates and identifies a user.
 */
public class Authenticate extends DynamoServlet
{
  private LoginUserAuthority mAuthority;
  private String mLoginFailedPage;
  private User mUser = new User();
  private String mLogin;
  private String mPassword;

  /////////////////////
  // Bean properties //
  /////////////////////

  /**
   * Returns true if the user has been authenticated.
   */
  public boolean isAuthenticated()
```

```
{
  return mUser.getPersonae(mAuthority) != null;
}

/**
 * Returns the page that the browser will be redirected to when a login
 * fails.
 */
public String getLoginFailedPage()
{
  return mLoginFailedPage;
}

/**
 * Changes the page that the browser will be redirected to when a login
 * fails.
 */
public void setLoginFailedPage(String pPage)
{
  mLoginFailedPage = pPage;
}

/**
 * Returns the persona for the currently logged-in user, if any.
 */
private Persona getLoginPersona()
{
  Persona[] loginPersonae = mUser.getPersonae(mAuthority);
  if ((loginPersonae == null) || (loginPersonae.length == 0))
    return null;
  else
    return loginPersonae[0];
}

/**
 * Returns the account name that the user logged in with.
 */
public String getUserAccount()
{
  Persona loginPersona = getLoginPersona();
  if (loginPersona == null)
    return "<not logged in>";
  else
    return loginPersona.getName();
}

/**
 * Returns the list of groups that the logged-in user is a member of.
 */
public String[] getUserGroups()
```

```
{
  Persona loginPersona = getLoginPersona();
  if (loginPersona == null)
    return new String[] { "<not logged in>" };

  // convert set of personae to a set of account names
  Persona[] groups = loginPersona.getSubPersonae();
  if ((groups == null) || (groups.length == 0))
    return new String[] { "<no groups>" };
  String[] groupNames = new String[groups.length];
  for (int i = 0; i < groups.length; i++)
    groupNames[i] = groups[i].getName();
  return groupNames;
}

/**
 * Returns the currently configured user authority.
 */
public LoginUserAuthority getUserAuthority()
{
  return mAuthority;
}

/**
 * Changes the user authority used for authentication.
 */
public void setUserAuthority(LoginUserAuthority pAuthority)
{
  mAuthority = pAuthority;
}

/////////////////////
// Form properties //
/////////////////////

public String getLogin()
{
  return mLogin;
}

public void setLogin(String pLogin)
{
  mLogin = pLogin;
}

public String getPassword()
{
  return mPassword;
}
```

```
  public void setPassword(String pPassword)
  {
    mPassword = pPassword;
  }

  //////////////////
  // Form handler //
  //////////////////

  /**
   * Handles a form submission to perform a login.
   */
  public void handleAuthenticate(DynamoHttpServletRequest pRequest,
    DynamoHttpServletResponse pResponse)
  {
    try {
      // Check validity of form properties
      if ((mLogin == null) || (mLogin.length() == 0) ||
          (mPassword == null)) {
        pResponse.sendLocalRedirect(mLoginFailedPage, pRequest);
        return;
      }

      // Hash the password as required by the user authority. In a more
      // tightly coupled client/server arrangement the client would obtain
      // the password hasher from the user authority via RMI and pass the
      // hash and the hash key back to the server, but we can't do that
      // in the browser interface so instead we do it all in the server.
      PasswordHasher hasher = mAuthority.getPasswordHasher();
      String hashedPassword;
      if (hasher == null)
        hashedPassword = mPassword; // not hashed
      else
        hashedPassword = hasher.hashPasswordForLogin(mPassword);

      // Perform the login
      if (!mAuthority.login(mUser, mLogin, hashedPassword,
                            hasher.getPasswordHashKey()))
        pResponse.sendLocalRedirect(mLoginFailedPage, pRequest);
    }
    catch (IOException e) {}
    finally { // clear out password
      mPassword = null;
    }
  }
}
```

Authenticate Bean Configuration file (Authenticate.properties)

```
# /atg/dynamo/security/examples/Authenticate
$class=Authenticate
$scope=session
userAuthority=/atg/dynamo/security/AdminUserAuthority
loginFailedPage=loginfailed.html
```

Authenticate JSP (authenticate.jsp)

The `authenticate.jsp` file is as follows:

```
<%@ taglib uri="/dspTaglib" prefix="dsp" %>
<%@ page import="atg.servlet.*"%>
<dsp:page>

<html>
<dsp:importbean bean="/atg/dynamo/droplet/ForEach"/>
<dsp:importbean bean="/atg/dynamo/droplet/Switch"/>
<dsp:importbean bean="/atg/dynamo/security/examples/Authenticate"/>

<body>
<!-- Display a login form if they have not logged in yet,
  -- or their login attributes if they have.
  -->

<dsp:droplet name="/atg/dynamo/droplet/Switch">
  <dsp:param bean="Authenticate.authenticated" name="value"/>
  <dsp:oparam name="false">
    Please log in:<p>

    <dsp:form action="<%=ServletUtil.getDynamoRequest(request).getRequestURI()%>"
      method="post">
      <table>
        <tr><td><b>Name:</b></td><td><dsp:input bean="Authenticate.login"
          type="text"/></td></tr>
        <tr><td><b>Password:</b></td><td><dsp:input bean="Authenticate.password"
          type="password"/></td></tr>
      </table><p>
      <dsp:input bean="Authenticate.authenticate" type="submit" value="Login"/>
    </dsp:form>
  </dsp:oparam>

  <dsp:oparam name="true">
    <b>You are logged in as</b> '<dsp:valueof
      bean="Authenticate.userAccount">?</dsp:valueof>'<p>
    <b>You are a member of the following groups:</b><p>
    <dsp:droplet name="/atg/dynamo/droplet/ForEach">
```

```
      <dsp:param bean="Authenticate.userGroups" name="array"/>
      <dsp:oparam name="output">
         <dsp:valueof param="element">?</dsp:valueof><br>
      </dsp:oparam>
      </dsp:droplet><!-- ForEach -->
    </table>
  </dsp:oparam>
</dsp:droplet><!-- Switch -->
</body>

</html>

</dsp:page>
```

Configuring Access Privileges

The ATG Control Center supports role-based access control and security. This enables administrators to assign different ATG Control Center privileges to users according to their role on the web development team. The process involves several basic steps:

1. Create a user group for each role (for example, `application developers` or `page designers`).
2. Define access privileges for each group.
3. Creating a login account for each user.
4. Assign everyone to the appropriate groups.

An ATG application includes a set of default user accounts and groups with predefined access privileges. You can use the ATG Control Center to modify standard login accounts. You can also create user accounts and groups that suit your specific requirements.

This section covers the following topics:

- Configuring the Default Login Accounts
- Managing User Accounts
- Managing User Groups and Privileges

Configuring the Default Login Accounts

By default, ATG automatically creates a set of standard user accounts, groups, and privileges each time you start your application (except when the `liveconfig` layer is enabled). Doing so ensures that the necessary accounts are initialized correctly.

On each application startup, the `/atg/dynamo/security/AdminAccountManager` runs an account initializer specified by the component’s `accountInitializer` property. By default, this property points to the `/atg/dynamo/security/AdminAccountInitializer` component.

The `AdminAccountInitializer` object obtains its information from another account manager (usually `/atg/dynamo/security/SimpleXmlUserAuthority`), which reads account information from the XML files included in each ATG product module:

```
<ATG10dir>/module root/src/config/atg/dynamo/security
```

ATG combines these files, resulting in an account database that contains the appropriate login accounts, groups, and privileges for each ATG module in your application. The account initializer copies this information from the `SimpleXmlUserAuthority` into the `AdminAccountManager` each time you start your application.

Note: ATG preserves new accounts and groups that you create, and any changes you make to the default login accounts. Any default accounts or groups that you delete, however, are recreated each time you start your application, unless you disable the automatic account creation feature.

Automatic account creation is disabled by default in the `liveconfig` configuration layer. If you want to prevent ATG from recreating the default accounts in development mode as well, set the `forceCreation` property of the `/atg/dynamo/security/AdminAccountInitializer` component to `false`.

Default User Accounts

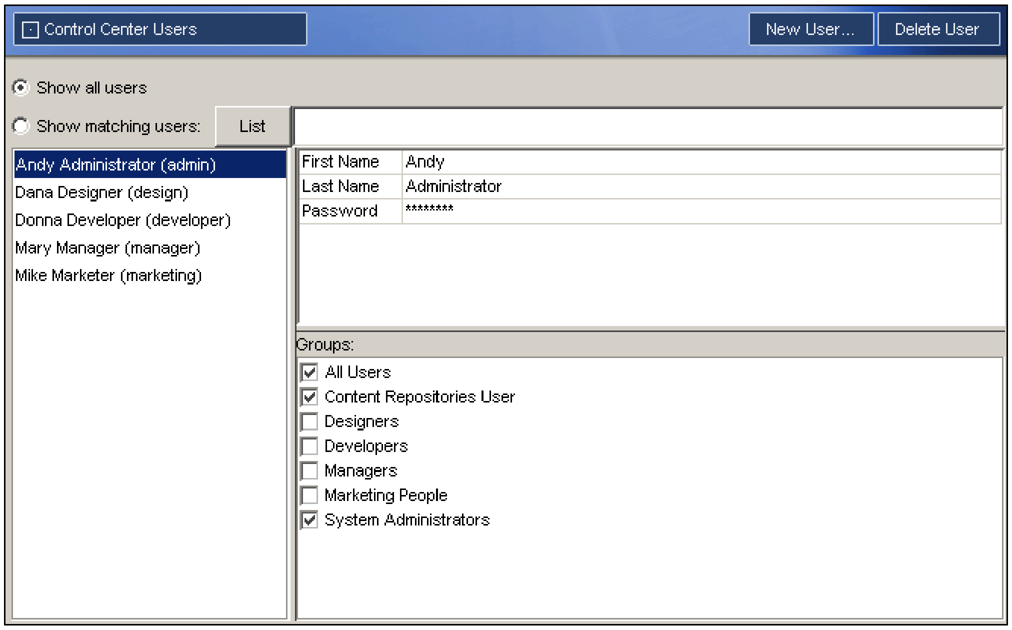
The following table lists the default login accounts for the ATG Adaptive Scenario Engine and ATG Commerce. You can use the ATG Control Center to change the names, passwords and group assignments for any of these accounts. To learn more about these accounts and the access privileges associated with them, see the Managing User Accounts section.

User Name	Login Name / Password	User Group	Module
Andy Administrator *(see note below)*	admin/admin	All Users System Administrators Content Repositories User	DSS
		Commerce Repositories User	DCS
Dana Designer	design/ design	All Users Designers	DSS
Donna Developer	developer/ developer	All Users Developers	DSS
Mary Manager	manager/ manager	All Users Managers	DSS
Mike Marketer	marketing/ marketing	All Users Marketing People Content Repositories User	DSS

User Name	Login Name / Password	User Group	Module
Mark Merchant	merchant/ merchant	All Users Commerce Repositories User	DCS

Managing User Accounts

Select People and Organizations > Control Center Users from the ATG Control Center navigation menu to see the account details and group affiliations of authorized ATG Control Center users.



Andy Administrator (admin)
Dana Designer (design)
Donna Developer (developer)
Mary Manager (manager)
Mike Marketer (marketing)

First Name	Andy
Last Name	Administrator
Password	********

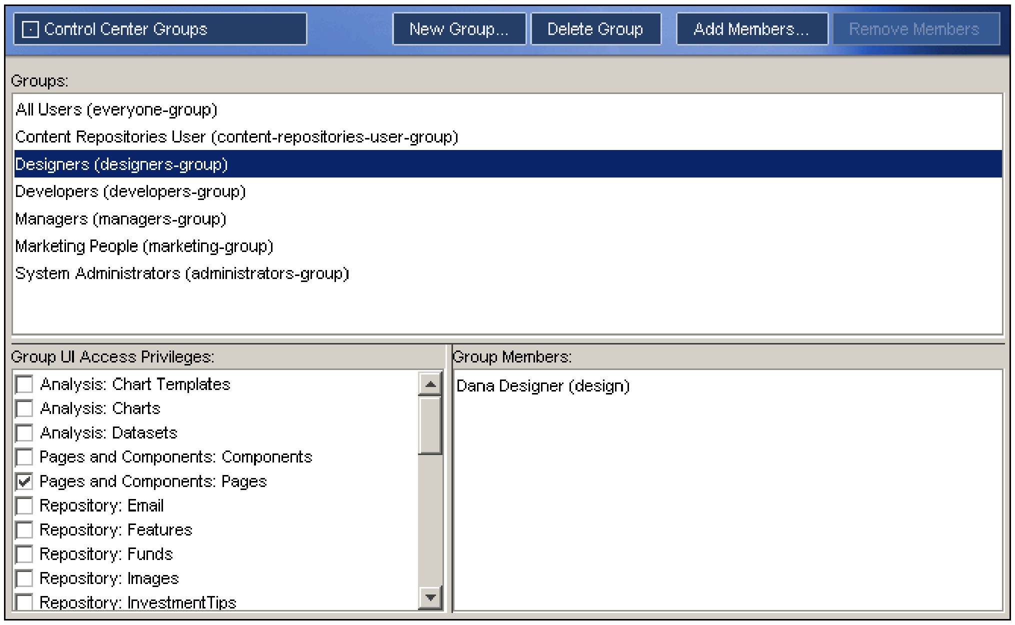
- [✓] All Users
- [✓] Content Repositories User
- [ ] Designers
- [ ] Developers
- [ ] Managers
- [ ] Marketing People
- [✓] System Administrators

People and Organizations > Control Center Users screen

- The accounts list on the left displays individual users by name and login. You can add and delete users from this list by clicking on New User and Delete User in the toolbar. If you need to find a specific account, select the Show Matching Users option (top left), type the account name you are looking for, and click on List.
- The property/value table (top right) displays the account information of the person you select from the accounts list.
- The Groups checklist (bottom right) displays the user groups that were created so far, and to which of these groups, if any, the selected user has been assigned. You can specify group assignments here (by checking the group checkboxes), or in the People and Organizations > Control Center Groups screen described in the next section.

Managing User Groups and Privileges

Select People and Organizations > Control Center Groups from the ATG Control Center navigation menu to see the list of user groups, their members, and their group-based access privileges.



Groups:

- All Users (everyone-group)
- Content Repositories User (content-repositories-user-group)
- Designers (designers-group)
- Developers (developers-group)
- Managers (managers-group)
- Marketing People (marketing-group)
- System Administrators (administrators-group)

Group UI Access Privileges:

- ☐ Analysis: Chart Templates
- ☐ Analysis: Charts
- ☐ Analysis: Datasets
- ☐ Pages and Components: Components
- ☑ Pages and Components: Pages
- ☐ Repository: Email
- ☐ Repository: Features
- ☐ Repository: Funds
- ☐ Repository: Images
- ☐ Repository: InvestmentTips

Group Members:

Dana Designer (design)

People and Organizations > Control Center Groups

- The Groups list (top) displays all user groups that were created so far. You can add and delete groups from this list by clicking New Group and Delete Group in the toolbar.

  Warning: Do not delete the System Administrators group; if you delete this group, the ATG application might not work properly.

- The Group Members list (bottom right) shows you the users currently assigned to the selected group. You can add and delete users from the group by clicking Add Members and Remove Members in the toolbar.

- The Group UI Access Privileges panel (bottom left) lists the individual screens in the ATG Control Center. Everyone in the selected user group has access to all areas that are checked.

If a user belongs to several groups that have different privileges, the user has all the privileges of those groups. For example, the Andy Administrator user account is a member of both the System Administrators group and the Content Repositories User group. The System Administrators group does not have privileges to access repositories, but the Content Repositories User group does. Thus, as a member of both groups, Andy Administrator can access repositories.

Configuring LDAP Repository Security

By default, an ATG application uses an SQL repository to authenticate users and groups and authorize access. You can configure the security mechanism to use the LDAP repository of an Oracle Directory Server (formerly Sun ONE or iPlanet Directory Server) or Microsoft Active Directory Server instead. An LDAP repository can be used to authenticate users and to authorize access by retrieving users' privileges from the LDAP directory. For more information about LDAP repositories, see the *ATG Repository Guide*.

This section describes how to configure an ATG application to use an LDAP repository to authenticate users and roles. The configuration process consists of the following steps:

1. Configure users and groups on an LDAP server.
2. Configure base common names. (Microsoft Active Directory only)
3. Configure a password hasher. (Sun ONE Directory Server 5.0 only)
4. Configure the InitialContextEnvironment component.
5. Create an XML definition file.
6. Test the LDAP Server Connection.
7. Configure the DYNAMO_MODULES variable.
8. Enable security information caching.

The following sections describe these steps in detail.

Configure Users and Groups on an LDAP Server

This section describes how to configure users and groups on ActiveDirectory and Oracle Directory Server. (For information about ATG's default users, groups, and privileges, see the Configuring Access Privileges section, earlier in this chapter.)

Configuring an ActiveDirectory Server

To configure users and groups on an Active Directory server, do the following:

1. Select Start > Program Files > Active Directory Users and Computers.
2. Select Action > New > Organizational Unit. Under the relevant domain, create an organizational unit called `dynamo-users`.
3. From any location in the domain, select Action > New > Users and create the users listed in Creating Users, later in this chapter.
4. In the `dynamo-users` organizational unit, select Action > New > Group and create the groups listed in Creating Groups, later in this chapter. Set the groups' scope to `Universal` and the type to `Distribution`.

Configuring an Oracle Directory Server

To configure users and groups on an Oracle (formerly Sun ONE) Directory Server, do the following:

1. Start the Directory Server Console.

2. In the navigation tree in the left pane, select the Directory Server that you want to use; for example, `"Directory Server" (server_name)`.
3. In the panel on the right side, click Open.
4. Click the Directory tab and locate the organization folder you wish to use (such as `yourcompany.com`).
5. Click the plus sign (+) next to the organization folder to expand the view.
6. To create an Organizational Unit, select Object > New > Organization Unit. Name the new unit `dynamo-users`.
7. Select Object > New > User and create the users listed in Creating Users, later in this chapter.
8. In the right pane, select `dynamo-users`.
9. Select Object > New > Group and create the static groups listed in Creating Groups, later in this chapter.
10. (Optional) If you have other existing users that you want to add to a group, add them to the one of the groups you created in Step 9.

Creating Users

The set of user and group accounts that ATG creates during account initialization depends on the application modules included in your application. If you want your LDAP configuration to support ATG's default set of users, create the following users:

User	Login name	Password	Module
Andy Administrator	admin	admin	DSS
Dana Designer	design	design	DSS
Donna Developer	developer	developer	DSS
Mary Manager	manager	manager	DSS
Mike Marketer	marketing	marketing	DSS
Mark Merchant	merchant	merchant	DCS

Creating Groups

Create the following groups for the ATG Adaptive Scenario Engine:

Group	Description	Members
everyone-group	All Users	admin design developer manager marketing ATG Commerce: merchant
administrators-group	System Administrators	admin
designers-group	Designers	design
developers-group	Developers	developer
managers-group	Managers	manager
marketing-group	Marketing People	marketing
server-restart-privilege	Server Restart	administrators-group developers-group
server-shutdown-privilege	Server Shutdown	administrators-group
support-cases-privilege	Tools: Submit a Support Request	administrators-group designers-group developers-group managers-group
support-knowledge-base-privilege	Support: Knowledge Base	administrators-group managers-group developers-group designers-group
components-module privilege	Pages and Components: Components By Module	administrators-group developers-group
components-path privilege	Pages and Components: Components By Path	administrators-group developers-group
pages-privilege	Pages and Components: Pages	administrators-group designers-group
admin-users-privilege	User Admin: Users	administrators-group managers-group
admin-roles-privilege	User Admin: Groups	administrators-group managers-group
tools-pipeline-editor-privilege	Tools: Pipeline Editor	administrators-group developers-group

Group	Description	Members
tools-integrations-privilege	Tools: Integrations	N/A
content-repositories-user-group	Content Repositories User	administrators-group marketing-group
targeting-profile-groups-privilege	Targeting: Profile Groups	administrators-group content-repositories-user-group marketing-group
targeting-content-groups-privilege	Targeting: Content Groups	administrators-group content-repositories-user-group marketing-group
targeting-targeted-content-privilege	Targeting: Content Targeters	administrators-group content-repositories-user-group marketing-group
targeting-preview-privilege	Targeting: Preview	administrators-group content-repositories-user-group marketing-group
scenarios-privilege	Scenarios: Scenarios	administrators-group marketing-group
scenarios-templates-privilege	Scenarios: Scenario Templates	administrators-group marketing-group
people-organization admin-privilege	Repository: Organizations	administrators-group marketing-group
people-roleadmin-privilege	Repository: Roles	administrators-group marketing-group
people-profiles-privilege	Repository: Profile Repository	administrators-group marketing-group
people-profiles-indiv-privilege	Repository: Profile Repository	administrators-group marketing-group

If you are running ATG Content Administration, create these additional static groups:

Group	Description	Members
publishing-workflow-privilege	Publishing: Workflow	administrators-group
publishing-repository-privilege	Publishing: Epublishing Repository	administrators-group

If you are running ATG Commerce, create this additional static group:

Group	Description	Members
commerce-repositories-user-group	Commerce Repositories User	admin merchant

Configuring Dynamically Generated Privileges

Any ATG Control Center privileges that are associated with a repository are generated dynamically by ATG as needed. If there are any ATG Control Center features with undefined privileges, you might see the following error message when your application starts up:

```
Allowing access for unknown privilege privilege_name
```

For example:

```
Allowing access for unknown privilege commerce-customproductcatalog-privilege
```

If you see an `unknown privilege` error message, create the privilege in your LDAP repository, then add it as a member of the appropriate group, as follows:

Type of Privilege	Member of Group
commerce	commerce-repositories-user-group
repository	content-repositories-user-group

If you want to automatically deny access to ATG Control Center features with undefined privileges (and disable `unknown privilege` error messages), set `/atg/devtools/DevSecurityDomain.allowUnknownPrivileges` to `false`.

Configure Base Common Names

Note: This section applies only to Microsoft Active Directory users.

In the `/atg/dynamo/security/AdminAccountManager` component, set the `baseCNs` property to the locations in the Active Directory server where users might be located. For example:

```
CN=Users,DC=adtest,DC=atg,DC=com,\DC=adtest,DC=atg,DC=com
```

When a user logs into the ATG Control Center or ATG Dynamo Server Admin UI as `admin`, for example, ATG converts that login name to `CN=admin` and appends it to the first entry in the `baseCNs` property. If the first combination fails, ATG tries to combine the login with the next `baseCN` in the list. In the previous example, this yields the following:

```
CN=admin,CN=Users,DC=adtest,DC=atg,DC=com
```

Note: The `baseCNs` values are case-sensitive.

Configure a Password Hasher

Note: This section applies only to Sun ONE Directory Server 5.0 users.

Sun ONE Directory Server 5.0 uses the Salted Secure Hash Algorithm (SSHA) to encrypt passwords. (Version 4.13 uses SHA encryption.) If you are using Sun ONE Directory Server 5.0, you must configure the `/atg/dynamo/security/AdminAccountManager` and `/atg/dynamo/security/AdminUserAuthority` components to use the `SSHAPasswordHasher` instead of the default `SHAPasswordDigestHasher`.

To do this, edit the passwordHasher property of the `AdminAccountManager` and `AdminUserAuthority` as shown below:

```
passwordHasher=/atg/dynamo/security/SSHAPasswordHasher
```

Note: You must edit the `AdminAccountManager.properties` and `AdminUserAuthority.properties` files manually in the following directory:

```
<ATG10dir>/DAS/LDAP/iPlanetDirectory/config/atg/dynamo/security/
```

Configure the InitialContextEnvironment Component

You must set up your `InitialContextEnvironment` component so that it specifies the JNDI environment properties used to connect to the LDAP repository.

Note: You must edit the `InitialContextEnvironment.properties` file manually instead of through the ATG Control Center.

Set the following values in your `InitialContextEnvironment.properties` file, which is in the following directory:

Active Directory

```
<ATG10dir>/DAS/LDAP/MicrosoftActiveDirectory/config/atg/dynamo/security
```

Oracle Directory Server

`<ATG10dir>/DAS/LDAP/iPlanetDirectory/config/atg/dynamo/security`

Property	Description
`providerURL`	URL of your LDAP server. Default value: `ldap://localhost:389`
`securityAuthentication`	Authentication mechanism for the provider to use. Choose one of the following mechanisms: `simple`: weak authentication (cleartext password) `CRAM-MD5`: CRAM-MD5 (RFC-2195) SASL mechanism `none:` no authentication (anonymous) Default value: `simple`

`securityPrincipal`	Identity of the principal to be authenticated, in the form of a distinguished name (DN). This identity is the Root DN's full common name (CN). For information about determining the DN, see your server's documentation.
	
	For example:
	
	For Active Directory:
	`CN=Administrator,CN=Users,DC=atg,DC=com`
	
	For Oracle Directory Server:
	`CN=Directory Manager`
	
	Tip: For Active Directory, you can use Active Directory Service Interfaces (ADSI) to determine the full common name. For more information about ADSI, see the Microsoft web site.
	
	Default value: `RootDN`

`securityCredentials`	Credentials of principal to be authenticated; this is the Root DN's password. Default value: `password`

Create an XML Definition File

To create an XML definition file for the LDAP directory, do the following:

1. Open the `ldapAdminUsers.xml` file in the following directory:

   `<ATG10dir>/DAS/LDAP/`*`<server vendor>`*`/config/atg/dynamo/security`

2. Modify all `yourdomain` references so they refer to the DN of the `dynamo-users` folder. For example:

   ```
   parent-dn="CN=dynamo-users,DC=atg,DC=com"
   search-root dn="CN=dynamo-users,DC=atg,DC=com"
   ```

Test the LDAP Server Connection

You can use ATG's LDAP server connection tool to test whether the JNDI environment properties in the `InitialContextEnvironment` component are configured appropriately for your server.

To test the connection to the LDAP server, do the following:

1. From the command line, switch to the following directory:

   ```
   <ATG10dir>/DAS/LDAP/lib
   ```

2. Issue the following command:

   For Active Directory:
   ```
   java -classpath ./ldap.jar LDAPConnection MicrosoftActiveDirectory
   ```

   For Oracle Directory Server:
   ```
   java -classpath ./ldap.jar LDAPConnection iPlanetDirectory
   ```

If ATG connects successfully to your LDAP server, it displays this message:

```
Successfully Created Context:
javax.naming.directory.InitialDirContextcontext_number
```

Troubleshooting the Server Connection

If ATG does not connect to your LDAP server, it displays one of the following error messages:

Error Message	InitialContextEnvironment Property to Modify
The following Error Occurred: `javax.naming.CommunicationException: <host:port>`. Root exception is `java.net.NoRouteToHostException`: Operation timed out: no further information	`providerURL`
The following Error Occurred: `javax.naming.AuthenticationNotSupportedException:` `SASL support not available:<value>`	`securityAuthentication`
Bad Username and/or Password: `javax.naming.AuthenticationException`: [LDAP: error code 49 - Invalid Credentials]	`securityPrincipal` and/or `securityCredentials`

Configure the DYNAMO_MODULES Variable

The `environment.sh/.bat` file in your `<ATG10dir>/home/localconfig` directory contains a `DYNAMO_MODULES` line that specifies application modules to include when you assemble your application. To include the LDAP Access Control Module when you start your application, you must append the module's name to the `DYNAMO_MODULES` line, as follows (enter the `DYNAMO_MODULES` setting all on one line, with no line breaks):

LDAP Server	Platform	DYNAMO_MODULES Setting
Active Directory	Windows	`set DYNAMO_MODULES=%DYNAMO_MODULES%;DSS;DAS.LDAP. MicrosoftActiveDirectory`
	UNIX	`DYNAMO_MODULES=$DYNAMO_MODULES:DSS:DAS.LDAP. MicrosoftActiveDirectory; export DYNAMO_MODULES`
Oracle Directory Server	Windows	`set DYNAMO_MODULES=%DYNAMO_MODULES%;DSS;DAS.LDAP. iPlanetDirectory`
	UNIX	`DYNAMO_MODULES=$DYNAMO_MODULES:DSS:DAS.LDAP. iPlanetDirectory; export DYNAMO_MODULES`

Note: Do not specify the LDAP Access Control module when you assemble your application; ATG does not set the configuration path properly.

Enable Security Information Caching

The LDAP security mechanism includes an option to enable caching of security information.

By default, caching is disabled to minimize potential security breaches. When caching is enabled, if you make changes on the LDAP server, there is a delay in propagating those changes to an ATG server because the view is not reloaded until the cache expires or is reloaded. You can manually reload the cache as described in the following section, Refreshing the Cache. To enable caching, set the `memberOfCacheEnabled` property of the following component to `true`:

Active Directory

```
<ATG10dir>/DAS/LDAP/MicrosoftActiveDirectory/config/atg/dynamo/security/AdminAccou
ntManager
```

Oracle Directory Server

```
<ATG10dir>/DAS/LDAP/iPlanetDirectory/config/atg/dynamo/security/AdminAccountManage
r
```

Refreshing the Cache

The `AdminAccountManager` gets its information from `/atg/dynamo/security/LDAPRepository`. By default, caching is enabled for this LDAP repository. If you make any changes to the LDAP directory, be sure to refresh the LDAP repository cache before propagating the changes to the `AdminAccountManager`.

To refresh the cache, do the following:

1. Open the ATG Dynamo Server Admin page with your web browser.
2. When ATG Dynamo Server Admin opens, click on the Admin ACC (ATG Control Center) link.
3. Click Reload Cache.

   Note: This button appears only if an LDAP repository is used to authenticate administrative users.

Scheduling Cache Updates

The cache is a schedulable service. You can configure the `AdminAccountManager` to never look in the cache, or you can configure it to reload itself periodically.

To configure the frequency of cache updates, specify the frequency (in minutes) in the `cacheReloadFrequency` property of the following component:

Active Directory
`<ATG10dir>/DAS/LDAP/MicrosoftActiveDirectory/config/atg/dynamo/security/AdminAccountManager`

Oracle Directory Server
`<ATG10dir>/DAS/LDAP/iPlanetDirectory/config/atg/dynamo/security/AdminAccountManager`

For example, to specify that the cache should be updated every 60 minutes, set `cacheReloadFrequency` as follows:

```
cacheReloadFrequency=60
```

For more information about configuring LDAP caching behavior, see the *ATG Repository Guide*.

15 Search Engine Optimization

Search Engine Optimization (SEO) is a term used to describe a variety of techniques for making pages more accessible to web spiders (also known as web crawlers or robots), the scripts used by Internet search engines to crawl the Web to gather pages for indexing. The goal of SEO is to increase the ranking of the indexed pages in search results.

This chapter describes several SEO techniques and the tools that the ATG platform provides for implementing them:

URL Recoding

The URLs generated by web applications can create problems for web spiders. These URLs typically include query parameters that the spider may not know how to interpret. In some cases, a spider will simply ignore a page whose URL includes query parameters.

Even if the spider does index the page, it may give the page lower ranking than desired, because the URL may not contain search terms that could increase the ranking. For example, consider a typical URL for an ATG Commerce site:

```
/mystore/product/product.jsp?prodId=prod1002&catId=cat234
```

This type of URL is sometimes referred to as “dynamic,” because the content of the page is dynamically generated based on the values of the query parameters.

Now consider a static URL for the same page:

```
/mystore/product/Q33+UltraMountain/Mountain+Bikes
```

A spider is more likely to index the page with the static URL, and when it does, it is likely to mark “Mountain Bikes” and “Q33 UltraMountain” as key search terms and weight the page heavily for them. As a result, when a user searches for one of these terms, this page appears near the top of the search results. The dynamic URL may return the same page and content when it’s clicked, but it is less likely to be ranked highly for these searches, and in some cases may not be indexed at all.

To address this concern, the ATG platform includes a URL recoding feature that enables you to optimize your pages for indexing by web spiders, without compromising the human usability of the site. The key to this feature is the ability to render URLs in different formats, depending on whether a page is accessed by a human visitor or a web spider. This is handled through the `atg.repository.seo.ItemLink` servlet bean, which uses the `User-Agent` property of the HTTP request to determine the type of visitor. If the visitor is a spider, the servlet bean renders a static URL that the spider can use for indexing; otherwise, it renders a standard ATG dynamic URL.

Of course, the ATG request-handling components cannot actually interpret these static URLs. Therefore, URL recoding also requires a servlet (`atg.repository.seo.JumpServlet`) that reads incoming static URLs (for example, if a user clicks a link returned by a Google search), and translates these URLs into their dynamic equivalents.

This section describes:

- Using URL Templates
- Configuring the ItemLink Servlet Bean
- Configuring the SEO Jump Servlet

Using URL Templates

To translate URLs from dynamic to static (or vice versa) requires some complex parsing logic and pattern matching. Both the `ItemLink` servlet bean and the SEO jump servlet construct URLs using properties that specify the format of the URL and the type of visitor viewing the page.

An important aspect of URL recoding is the use of URL templates. These templates are Nucleus components that the `ItemLink` servlet bean and the jump servlet use when they construct URLs. URL templates include properties that specify the format of the URLs, the browser types supported, and how to parse requests.

The URL template classes consist of `atg.repository.seo.UrlTemplate`, which is an abstract base class, and its two subclasses:

- `atg.repository.seo.DirectUrlTemplate` defines the format of the direct (dynamic) URLs created by the `ItemLink` servlet bean for human site visitors.
- `atg.repository.seo.IndirectUrlTemplate` defines the format of the indirect (static) URLs created by `ItemLink` servlet bean for web spiders. It is also used by the SEO jump servlet to determine how to translate these static URLs back to dynamic URLs.

In addition, the `atg.repository.seo` package has a `UrlTemplateMapper` interface that is used by `ItemLink` to map repository item descriptors to URL templates. The package also includes a `UrlTemplateMapperImpl` implementation class for this interface.

Configuring URL Templates

The `UrlTemplate` base class has several key properties that are inherited by the `DirectUrlTemplate` and `IndirectUrlTemplate` subclasses. The following list summarizes these properties. Some of the properties are described in more detail in subsequent sections.

urlTemplateFormat
The URL format used by the `ItemLink` servlet bean to generate page links. The format is expressed in `java.text.MessageFormat` syntax, but uses parameter names instead of numbers as placeholders. See Specifying URL Formats.

maxUrlLength
The maximum number of characters in a generated URL.

supportedBrowserTypes
List of browser types supported by this template. Each entry must match the name of an `atg.servlet.BrowserType` component. See Specifying Supported and Excluded Browser Types.

excludedBrowserTypes
List of browser types that are explicitly not supported by this template. Each entry must match the name of an `atg.servlet.BrowserType` instance. See Specifying Supported and Excluded Browser Types.

webAppRegistry
The web application registry that contains the context paths for registered web applications.

The `IndirectUrlTemplate` class has additional properties not found in the `DirectUrlTemplate` class. These properties are summarized in the following list. Note that these properties are used only by the SEO jump servlet, and not by the `ItemLink` servlet bean.

indirectRegex
The regular expression pattern the jump servlet uses to extract parameter values from static request URLs. See Using Regular Expression Groups.

regexElementList
An ordered list where each list element specifies the parameter type of the corresponding regular expression element in `indirectRegex`. See Using Regular Expression Groups.

forwardUrlTemplate
The URL format used by the jump servlet to generate a dynamic URL for forwarding a static request URL. Like the `urlTemplate` property, this is expressed using the same syntax as `java.text.MessageFormat`, but uses parameter names instead of parameter numbers as placeholders.

useUrlRedirect
If `true`, the jump servlet redirects the request to a dynamic URL rather than forwarding it. Default is `false`, which means that forwarding is used.

Specifying URL Formats

The `urlTemplateFormat` property of the `DirectUrlTemplate` and `IndirectUrlTemplate` classes is used to specify the format of the URLs generated by the `ItemLink` servlet bean. In addition, the `urlTemplateFormat` property of the `IndirectUrlTemplate` class is used by the jump servlet to determine how to interpret static request URLs created by the servlet bean.

The value of `urlTemplateFormat` should include placeholders that represent properties of repository items. `ItemLink` fills in these placeholders when it generates a URL. The jump servlet uses them to extract the property values from a static request URL.

The placeholder format is a parameter name (which typically represents a property of a repository item) inside curly braces. For example, a dynamic URL for displaying a product on an ATG Commerce site might be specified in a direct URL template like this:

```
urlTemplateFormat=\
  /catalog/product.jsp?prodId\={item.id}&catId\={item.parentCategory.id}
```

A dynamic URL generated using this format might look like this:

```
/catalog/product.jsp?prodId=prod1002&catId=cat234
```

The static URL equivalent in an indirect URL template might look like this:

```
urlTemplateFormat=/jump/product/{item.id}/{item.parentCategory.id}\
  /{item.displayName}/{item.parentCategory.displayName}
```

Note that this URL format includes the `displayName` properties of the repository item and its parent category, and also the repository IDs of these items. The `displayName` properties provide the text that a web spider can use for indexing. The repository IDs are included so that if an incoming request has this URL, the SEO jump servlet can extract the repository IDs and use them to fill in placeholders in the dynamic URL it generates. In addition, the URL begins with `/jump` to enable the jump servlet to detect it as a static URL (as described in Specifying Context Paths).

A static URL generated using this format might look like this:

```
/jump/product/prod1002/cat234/Q33+UltraMountain/Mountain+Bikes
```

Encoding Parameter Values

By default, the SEO components use URL encoding when they insert parameter values in placeholders. This ensures that special characters in repository item property values do not make the URL invalid. For example, the value of a `displayName` property will typically include spaces, which are not legal characters in URLs. Therefore, each space is encoded as a plus sign (+), which is a legal character.

In some cases, it is necessary to insert a parameter value unencoded. For example, some repository properties represent partial URL strings, and therefore need to be interpreted literally. To support this, the placeholder syntax allows you to explicitly specify whether to encode a parameter. For example:

```
{item.template.url,encode=false}
```

For parameters that should be encoded, you can explicitly specify `encode=true`; however, this is not necessary, because `encode` defaults to `true`.

Another way to specify that a parameter should not be encoded is to use square brackets rather that curly braces. For example:

```
[item.template.url]
```

Specifying Context Paths

When developing a site that uses URL recoding for SEO, you must be careful about whether the generated URLs should include the application's context path. Dynamic URLs must include the context path (so that these URLs are properly interpreted by ATG's request-handling pipeline). Static URLs do not need the context path (the URL is never actually interpreted by the pipeline), and it is better to omit it because it may interfere with the jump servlet's ability to detect static URLs in requests. This is because the `urlTemplateFormat` property of an indirect URL template will typically start with a special string (such as `/jump`) that enables the jump servlet to detect these URLs. The jump servlet should then be configured to use URI-mapping to detect these URLs, as described in Configuring the SEO Jump Servlet.

Therefore, when `ItemLink` generates an indirect URL, it is undesirable for the application's context path to be prepended to the URL. To avoid including the context path, links created with the `<dsp:a>` tag should use the `href` attribute, not the page attribute. The page attribute prepends the application's context path to the generated URL, but the `href` attribute does not.

However, using the `href` attribute means that the context path will not automatically be prepended to the dynamic URLs generated by `ItemLink`. Also, since static URLs won't have the context path, the jump servlet will not be able to include this information in the dynamic URLs it forwards inbound requests to. Therefore, you should include the context path when you configure the following:

- The `urlTemplateFormat` property of each direct URL template
- The `forwardUrlTemplateFormat` property of each indirect URL template

There are two ways you can specify the context path:

- Explicitly include the context path when you set the property.
- Specify the name of a registered web application. There must be a single colon character after the web application name to denote that it is a web application name.

Specifying the name of a registered web application rather than including the context path itself has the advantage that you don't need to know what the context path actually is. Also, if the context path changes, you don't need to update each URL template component. The main disadvantage is that you need to know what web application registry the web application is registered with, and set the `webAppRegistry` property of each URL template component to this value.

Note that for a multisite application that uses a path-based URL strategy, you should not configure URL templates to include the context path in generated URLs. See URL Recoding for Multisite Applications.

When generating a direct URL (either with `ItemLink` using a direct template, or the jump servlet using the forward URL in an indirect template), the following logic is used to determine the context path:

1. If a web application name occurs in the first part of the URL with the format *`webAppName`*`:`*`restOfUrl`*, the web application is resolved using the web application registry specified in the `webAppRegistry` property of the template. The web application's context path is then used to replace the *`webAppName`* placeholder.
2. If there is a colon in the first part of the URL but no web application name before the colon, the context path of the default web application is used. The default web application is specified in the `defaultWebApp` property of the `ItemLink` servlet bean

or of the jump servlet (the former if generating a direct URL for a page link, the latter if generating a forwarding URL for an inbound request).

3. Otherwise, the context path is assumed to already be present.

Specifying Supported and Excluded Browser Types

Both the `ItemLink` servlet bean and SEO jump servlet can be configured to use multiple URL templates. The actual template used for any given request is partly determined by examining the `User-Agent` property of the HTTP request and finding a template that supports this browser type.

The `supportedBrowserTypes` and `excludedBrowserTypes` properties of a URL template are mutually exclusive. You can configure an individual template to support a specific set of browser types, or to exclude a specific set of browser types, but not both. A typical configuration is to set `excludedBrowserTypes` to `robot` in direct URL templates, and set `supportedBrowserTypes` to `robot` in indirect URL templates. This will ensure that web spiders will see indirect URLs, and human visitors will see direct URLs.

The `supportedBrowserTypes` or `excludedBrowserTypes` property is a list of components of class `atg.servlet.BrowserType`. (Note that to add a component to the list, you specify the `name` property of the component, rather than the Nucleus name of the component.) The ATG platform includes a number of `BrowserType` components, which are found in Nucleus at `/atg/dynamo/servlet/pipeline/BrowserTypes`. You can also create additional `BrowserType` components. For more information, see Customizing a Request-Handling Pipeline.

Using Regular Expression Groups

When a static URL is part of an incoming request, the SEO jump servlet parses the URL to extract parameter values, which it then uses to fill in placeholders in the dynamic URL it generates. To extract the parameter values, the servlet uses regular expression groups, which you specify using the `indirectRegex` property of the indirect URL component.

For example, suppose you have a URL format that looks like this:

```
urlTemplateFormat=/jump/product/{item.id}/{item.parentCategory.id}\
  /{item.displayName}/{item.parentCategory.displayName}
```

The regular expression pattern for this format might be specified like this:

```
indirectRegex=/jump/product/([^/].*?)/([^/].*?)/([^/].*?)/([^/].*?)$
```

This pattern tells the jump servlet how to extract the parameter values from a static URL. In addition, the servlet needs information about how to interpret the parameters. Some parameters may be simple String values, while others may represent the ID of a repository item. If the parameter is a repository item ID, the servlet needs to determine the item type and the repository that contains the item.

Therefore the indirect URL template also includes a `regexElementList` property for specifying each parameter type. This property is an ordered list where the first element specifies the parameter type of the first regular expression, the second element specifies the parameter type of the second regular expression, and so on.

The syntax for each parameter type entry in the list is:

```
paramName | paramType [| additionalInfo]
```

The *paramName* is used to match the parameter with placeholders in the direct URL that the servlet forwards the request to.

Valid values for *paramType* are:

- `string`, which denotes a simple string
- `id`, which denotes the ID of a repository item

The optional *additionalInfo* field can be used to specify additional details if *paramType* is `id`. (This field should be omitted if *paramType* is `string`.) The syntax of *additionalInfo* takes one of the following forms:

```
repositoryName:itemDescriptorName
```

```
itemDescriptorName
```

The parameter type list for the regular expression pattern shown above would look similar to this:

```
item | id | /atg/commerce/catalog/ProductCatalog:product
parentCategory | id | /atg/commerce/catalog/ProductCatalog:category
displayName | string
parentCategoryDisplayName | string
```

Configuring URL Template Mappers

URL template mappers are used by the `ItemLink` servlet bean to map repository item descriptors to URL templates. The servlet bean has an `itemDescriptorNameToMapperMap` property that maps item descriptors to URL template mappers. For example:

```
itemDescriptorNameToMapperMap=\
    product=/atg/repository/seo/ProductTemplateMapper,\
    category=/atg/repository/seo/CategoryTemplateMapper
```

Each template mapper component has a `templates` property that specifies one or more templates to use for rendering static URLs, and a `defaultTemplate` property that specifies the template to use for rendering dynamic URLs. So, in this example, the `product` item descriptor is associated with the templates listed by the `ProductTemplateMapper` component, and the `category` item descriptor is associated with the templates listed by the `CategoryTemplateMapper` component. When `ItemLink` generates a link to a specific repository item, it uses this mapping to determine the URL template to use.

Configuring the ItemLink Servlet Bean

The `ItemLink` servlet bean takes a repository item as input and uses a URL template to construct a static or dynamic link to that item, depending on the value of the HTTP request's `User-Agent` property. It uses an indirect URL template if the visitor is a web spider, and a direct URL template otherwise.

For a given item descriptor, there can be multiple URL templates. The one selected is based on matching the value of the HTTP request's `User-Agent` property to the browser types in the template's configuration.

Most of the information needed by the servlet bean is provided using its input parameters. However, there are a few properties that you can set as well:

Property	**Description**
`itemDescriptorNameToMapperMap`	Map of item descriptor names to `UrlTemplateMapper` instances. This property must be set. See Configuring URL Template Mappers.
`defaultItemDescriptorName`	Specifies the value to use for the `itemDescriptorName` input parameter, if the input parameter is not supplied.
`defaultRepository`	Specifies the value to use for the `repository` input parameter, if the input parameter is not supplied.
`defaultWebApp`	Specifies the default web application to use when determining the context path for a URL.

Note: If you embed the `ItemLink` servlet bean in the `Cache` servlet bean, you must be sure to create separate cache keys for human visitors and web spiders (which can be differentiated based on the `User-Agent` value of the request). Otherwise the page may end up containing the wrong type of URL for the visitor.

For additional information about the `ItemLink` servlet bean, see the *ATG Page Developer's Guide*.

Configuring the SEO Jump Servlet

The `atg.repository.seo.JumpServlet` class is responsible for translating static request URLs to their dynamic equivalents. This class extends the `atg.servlet.pipeline.InsertableServletImpl` class, so it can be inserted in the DAS or DAF servlet pipeline. However, because this servlet is intended to process only static URLs, and incoming URLs are typically dynamic, including the servlet in a pipeline may be very inefficient. Therefore, it is generally preferable to configure it as a URI-mapped servlet in the `web.xml` file of your application, to ensure that it processes only static URLs.

To configure the jump servlet in a `web.xml` file, you actually declare another class, `atg.repository.seo.MappedJumpServlet`. This is a helper class that invokes the `JumpServlet` component. In addition, you declare a servlet mapping for the pattern that the servlet uses to detect static request URLs.

For example, if you have configured your static URLs to include /jump/ immediately after the context root, the entry in the web.xml file would be similar to this:

```
<servlet>
  <servlet-name>MappedJumpServlet</servlet-name>
  <servlet-class>atg.repository.seo.MappedJumpServlet</servlet-class>
  <init-param>
    <param-name>jumpServlet</param-name>
    <param-value>ctx:dynamo:/atg/repository/seo/JumpServlet</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>MappedJumpServlet</servlet-name>
  <url-pattern>/jump/*</url-pattern>
</servlet-mapping>
```

There also are several properties you can configure for the Nucleus component:

Property	**Description**
`templates`	An array of `IndirectUrlTemplate` components that the servlet examines in the order specified until it finds one that matches the static request URL.
`defaultRepository`	Specifies the repository to associate with repository items for which a repository is not otherwise specified.
`defaultWebApp`	Specifies the default web application to use when determining the context path for a URL.

In addition, the servlet has `nextServlet` and `insertAfterServlet` properties for including the component in a servlet pipeline. If the servlet is configured through the `web.xml` file, you should not set these properties.

URL Recoding for Multisite Applications

For a multisite application, the URL for a link from one site to another must include information that identifies the target site. Depending on how the application is configured, the site information is part of the domain name or the context root. (Links within a site typically use relative URLs, so site information does not need to be included.)

You do not need to configure any of your URL recoding components to include the site information (such as the context path) in cross-site links. Instead, you pass the URL generated by the `ItemLink` servlet bean to the `atg.droplet.multisite.SiteLinkDroplet` servlet bean, which adds the site information to the URL.

For information about the `SiteLinkDroplet` servlet bean, see *ATG Page Developer's Guide*.

Canonical URLs

A number of Web search engines enable you to specify the canonical form of the URL for an indexed page. For example, suppose your site has a page that can be accessed by several different URLs (either because the query parameters can vary, or because there are multiple paths to the same page). Rather than indexing the page separately by each different URL (and diluting the page ranking as a result), you can instruct search engines to index the page by a single URL in its standard (canonical) form.

You specify the canonical URL for a page using a `link` tag with its `rel` attribute set to `"canonical"`. For example:

```
<link rel="canonical" href="www.example.com/product/Blue+Suede+Shoes" />
```

When a web spider crawls a page, it records the page's URL as the value specified in the `href` attribute, rather than the actual URL that was used to access the page.

Creating Canonical URLs

To code your JSPs to render canonical URLs, you use the URL recoding feature described in the URL Recoding section. The canonical URL generated for a page is similar to the static URL rendered for web spiders by an indirect URL template. The canonical URL should always be static, regardless of whether the page is accessed by a spider or a human user. That way, if a spider happens to access a page using a dynamic URL (e.g., by following a link from another page), it will still see (and record) the static URL it finds in the `link` tag. As with the URL recoding feature, when a user accesses a page via a static URL, the SEO jump servlet translates it back to its dynamic equivalent for processing.

To render canonical URLs, you use the `atg.repository.seo.CanonicalItemLink` servlet bean. This class is similar to the `ItemLink` servlet bean, except that it does not use template mappers, because the URL template used does not depend on the browser type of the request. So rather than configuring the `ItemLink` servlet bean's `itemDescriptorNameToMapperMap` property to map item descriptors to `UrlTemplateMapper` components, you configure the `CanonicalItemLink` servlet bean's `itemDescriptorNameToUrlTemplateMap` property to map item descriptors directly to `UrlTemplate` components. For example:

```
itemDescriptorNameToUrlTemplateMap=\
        product=/atg/repository/seo/ProductIndirectTemplate,\
        category=/atg/repository/seo/CategoryIndirectTemplate
```

The following example illustrates using the `CanonicalItemLink` servlet bean on a product detail page to render a `link` tag specifying the page's canonical URL:

```
<dsp:droplet name="/atg/repository/seo/CanonicalItemLink">
  <dsp:param name="id" param="productId"/>
  <dsp:param name="itemDescriptorName" value="product"/>
  <dsp:param name="repositoryName"
     value="/atg/commerce/catalog/ProductCatalog"/>
  <dsp:oparam name="output">
    <dsp:getvalueof var="pageUrl" param="url" vartype="java.lang.String"/>
    <link rel="canonical" href="${pageUrl}"/>
 </dsp:oparam>
</dsp:droplet>
```

For additional information about the `CanonicalItemLink` servlet bean, see the *ATG Page Developer's Guide*.

Sitemaps

HTML-only pages are generally easy for a web spider to parse, but on pages that use Flash or JavaScript, a spider may have difficulty finding links to other pages. As a result, search engines may give those pages low rankings.

You can often improve the ranking of your site pages by using sitemaps to help spiders find the pages. Sitemaps are files stored on a web server that list the URLs of the site pages, so web spiders are able to identify site content without relying exclusively on their ability to crawl and parse the pages. Sitemaps are not an official standard, but they are supported by many search engines, including Google, Yahoo!, and MSN.

This section includes the following:

- Overview of Sitemaps
- Sitemap Generation Tools
- Configuring Sitemap Generation
- Additional Configuration for Multisite Applications
- Configuring Sitemap Writing
- Invoking Sitemap Generation and Writing

Overview of Sitemaps

Sitemap files are XML documents that contain URLs for the pages of your site. A simple sitemap file would look similar to this:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
   <url>
      <loc>http://www.example.com/</loc>
   </url>
   <url>
      <loc>http://www.example.com/contact/</loc>
   </url>
</urlset>
```

Each `<url>` tag is used to specify the URL of a single page. This tag has several child tags:

- `<loc>` is a required tag that specifies the actual URL. Note that the value of a `<loc>` tag must begin with the protocol (such as `http`) and end with a trailing slash, if your web server requires it. This value must be less than 2,048 characters long.
- `<lastmod>` is an optional tag for specifying the date the page was last modified.
- `<changefreq>` is an optional tag that indicates how often the page is likely to change.
- `<priority>` is an optional tag that assigns a priority value to the page, relative to other pages on the site.

For more information about these tags, see:

```
http://www.sitemaps.org/protocol.php
```

Sitemap Indexes

A single site can have more than one sitemap. Using multiple sitemaps can help make your sitemaps more manageable; for example, you can have a separate sitemap for each area of a site. On very large sites, having multiple sitemaps may be necessary to ensure that no individual sitemap exceeds the maximum file size (10 Mb or 50,000 URLs).

To use multiple sitemaps, you list them all in an XML file called a sitemap index. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<sitemapindex xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
   <sitemap>
      <loc>http://www.example.com/sitemap.xml</loc>
   </sitemap>
   <sitemap>
      <loc>http://www.example.com/sitemap2.xml</loc>
   </sitemap>
</sitemapindex>
```

The `<loc>` tag is a required child tag of the `<sitemap>` tag; it specifies the URL of a sitemap file. The `<sitemap>` tag also has an optional `<lastmod>` child tag that specifies the date the sitemap file was last modified.

Sitemap Generation Tools

An ATG site typically consists of both static pages (such as contact information pages) and dynamic pages (such as product detail pages), as discussed in URL Recoding. The logic for creating sitemaps for these two types of pages differs, so the ATG sitemap generation tools create separate sitemaps for static and dynamic pages, and then include the sitemaps for both types of pages in a single sitemap index.

The process of generating sitemaps and sitemap indexes is managed by the `atg.sitemap.SitemapGeneratorService` class. This service can invoke all of the following:

- One or more components of class `atg.sitemap.StaticSitemapGenerator`, for generating sitemaps of static pages
- One or more components of class `atg.sitemap.DynamicSitemapGenerator`, for generating sitemaps of dynamic pages
- One component of class `atg.sitemap.SitemapIndexGenerator`, for creating a sitemap index containing references to the sitemaps generated by the `SitemapGenerator` components

Creating sitemaps is a two-step process. First, you run the `SitemapGeneratorService`, which creates the sitemaps and the sitemap index as repository items in the `SitemapRepository`. Then, to write out the actual XML files, you run the `atg.sitemap.SitemapWriterService` on each page-serving ATG instance in your production environment. This repository-based approach makes it possible to distribute the sitemaps and index to all of your servers without running the generators multiple times.

There are three ways to invoke the `SitemapGeneratorService` and `SitemapWriterService`:

- Scheduling them to run automatically at specified times or intervals
- Configuring a deployment listener so they are run automatically after a CA deployment
- Invoking them manually through ATG Dynamo Server Admin

Configuring the sitemap generation and writing components is described in Configuring Sitemap Generation and Configuring Sitemap Writing. The different ways of invoking the generation process are discussed in Invoking Sitemap Generation and Writing.

Note that the ATG sitemap generation tools assume that you are using the techniques described in the URL Recoding section, and they invoke some of the same classes and components described there.

Configuring Sitemap Generation

To set up the sitemap generation process, you must create and configure:

- One `SitemapGeneratorService` component
- One or more `StaticSitemapGenerator` components
- One or more `DynamicSitemapGenerator` components
- One `SitemapIndexGenerator` component
- One `SitemapWriterService` component on each page-serving ATG instance

Configuring the SitemapGeneratorService

The `atg.sitemap.SitemapGeneratorService` class manages the process of generating sitemaps and sitemap indexes. The ATG platform includes a component of this class, `/atg/sitemap/SitemapGeneratorService`. To configure a `SitemapGeneratorService` component, set the following properties:

Property	Description
`sitemapGenerators`	An array of components of classes that implement the `atg.sitemap.SitemapGenerator` interface. Typically this is a mix of components of class `atg.sitemap.StaticSitemapGenerator` and components of class `atg.sitemap.DynamicSitemapGenerator`.
`sitemapIndexGenerator`	A component of class `atg.sitemap.SitemapIndexGenerator`.
`sitemapRepository`	The repository that stores the sitemaps and the sitemap index. This should be set to `/atg/sitemap/SitemapRepository`.
`sitemapPropertiesManager`	A component that maps properties in the `SitemapRepository` to the names used in Java code. This should be set to `/atg/sitemap/SitemapPropertiesManager`.
`sitemapTools`	A component with utility methods for looking up and modifying items in the `SitemapRepository`. This should be set to `/atg/sitemap/SitemapTools`.
`maxUrlsPerSitemap`	The maximum number of URLs to be stored in a single sitemap file. If this property is not set explicitly, it defaults to 50000, the maximum allowed by `sitemap.org`.
`maxSitemapSize`	Maximum size of a single sitemap file, in bytes. If this property is not set explicitly, it defaults to 10485760 (10 Mb), the maximum allowed by `sitemap.org`.
`urlPrefix`	String to prepend to the URL entries produced by the generator components. This property is not actually used by the `SitemapGeneratorService` itself, but you can set it here and then set the corresponding property of the generator components by linking to this value.
`webApp`	The Nucleus pathname for the component of class `atg.service.webappregistry.WebApp` that represents the web application that the sitemap is generated for. This property is not actually used by the `SitemapGeneratorService` itself, but you can set it here and then set the corresponding property of the generator components by linking to this value.
`warDir`	The operating-system pathname of the deployed WAR file that the sitemap is generated for. This property is not actually used by the `SitemapGeneratorService` itself, but you can set it here and then set the corresponding property of the generator and writer components by linking to this value.

There are additional properties that need to be configured for a multisite application. See Additional Configuration for Multisite Applications.

In addition to these sitemap-related properties, `SitemapGeneratorService` also has several properties it inherits from `atg.service.scheduler.SingletonSchedulableService`. See Invoking Sitemap Generation and Writing for more information.

A properties file for a `SitemapGeneratorService` component might look like this:

```
$class=atg.sitemap.SitemapGeneratorService
$scope=global

schedule=calendar * * . . 1 .
scheduler=/atg/dynamo/service/Scheduler
clientLockManager=/atg/dynamo/service/ClientLockManager
lockName=SitemapGeneratorService

sitemapGenerators=\
     /atg/sitemap/ProductSitemapGenerator,\
     /atg/sitemap/CategorySitemapGenerator,\
     /atg/sitemap/StaticSitemapGenerator
sitemapIndexGenerator=/atg/sitemap/SitemapIndexGenerator

sitemapRepository=/atg/sitemap/SitemapRepository
sitemapPropertiesManager=/atg/sitemap/SitemapPropertiesManager
sitemapTools=/atg/sitemap/SitemapTools

maxUrlsPerSitemap=10000
maxSitemapSize=5000000
```

Configuring the StaticSitemapGenerator

The `atg.sitemap.StaticSitemapGenerator` class generates sitemaps for static pages. This class has a `staticPages` property that you use to specify a list of static pages to be included in the sitemap. For example:

```
staticPages=/index.jsp,\
            /support/contact.jsp,\
            /company/news.jsp,\
            /company/aboutUs.jsp
```

The entries in the list can use wildcards in the following ways:

- A single asterisk (`*`) matches a filename of any length in the specified directory, but does not include files in subdirectories. For example, `/company/*.jsp` matches any JSP file in the `/company/` directory, but not in the `/company/about/` subdirectory..
- Two asterisks (`**`) match subdirectories to any depth. For example, `/company/**/*.jsp` matches any JSP file in the `/company/` directory or any subdirectory of it.

- A question mark (?) matches any single character in a filename. For example, `/company/news?.jsp` matches `news1.jsp`, `news2.jsp`, etc., in the `/company/` directory.

The `StaticSitemapGenerator` class has `changeFrequency` and `priority` properties for setting the default values of the `<changefreq>` and `<priority>` tags for each URL in the static pages sitemap. You can override these values for an individual page or group of pages by explicitly setting the values in the entry for the page or pages, as in this example:

```
staticPages=/index.jsp,\
           /support/contact.jsp:monthly:0.8,\
           /company/*.jsp:weekly
```

To configure a `StaticSitemapGenerator` component, set the following properties:

Property	Description
`changeFrequency`	The default value to use for the `<changefreq>` tag for each URL. This value can be overridden for specific pages in the `staticPages` property (see above).
`priority`	The default value to use for the `<priority>` tag for each URL. This value can be overridden for specific pages in the `staticPages` property (see above).
`staticPages`	A list of static pages to be included in the sitemap (see above).
`sitemapFilePrefix`	A String used to form the names of the static sitemap files. If a single file is generated, `.xml` is appended to this String to form the filename (e.g., if `sitemapFilePrefix=staticSitemap`, the resulting filename is `staticSitemap.xml`). If multiple files are generated (because the maximum number of URLs or maximum file size is exceeded), `2.xml`, `3.xml`, and so on are appended to the second and subsequent files (e.g., `staticSitemap2.xml`, `staticSitemap3.xml`, etc.). Note that the value of `sitemapFilePrefix` must be unique for each sitemap generator component, to prevent overwriting of files.
`urlPrefix`	String to prepend to the filenames found using `staticPages` to form the URL entries included in the sitemap. This should include the protocol, domain, and port (if needed). If the webApp property is null, `urlPrefix` should also include the context root; for example: `http://www.example.com/mywebapp/`

`webApp`	The Nucleus pathname for the component of class `atg.service.webappregistry.WebApp` that represents the web application that the sitemap is generated for; for example: `/atg/registry/webappregistry/MyWebApp` The `StaticSitemapGenerator` examines the web application to find the context root to append to `urlPrefix`. If you include the context root in `urlPrefix`, leave `webApp` null.
`warDir`	The operating-system pathname of the deployed WAR file that the sitemap is generated for; for example: `C:\jboss-eap-4.2\jboss-as\server\atg\deploy\ATG.ear\mywebapp.war` The `StaticSitemapGenerator` looks in this directory for files that match the patterns specified in the `staticPages` property.

Configuring the DynamicSitemapGenerator

The `atg.sitemap.DynamicSitemapGenerator` class generates sitemaps for dynamic pages. This class uses a URL template to translate dynamic URLs to static URLs for inclusion in the sitemaps. For example, suppose the URL for a product detail page looks like this:

```
http://mywebsite.com/mywebapp/productDetail.jsp?productId=id
```

`DynamicSitemapGenerator` iterates through all of the `product` repository items in the `ProductCatalog` repository and for each item generates a static URL, such as:

```
http://mywebsite.com/mywebapp/jump/product/12345/Oxford+Shirt/
```

See the Using URL Templates section for more information about URL templates.

To configure a `DynamicSitemapGenerator` component, set the following properties:

Property	**Description**
`changeFrequency`	The value to use for the `<changefreq>` tag for each URL.
`priority`	The value to use for the `<priority>` tag for each URL.
`sourceRepository`	The repository whose items are used to construct the dynamic sitemap URLs. For example, for an ATG Commerce site, this is typically `/atg/commerce/catalog/ProductCatalog`.

Property	Description
`itemDescriptorName`	The name of the type of item to retrieve from the source repository to use for constructing URLs. For example, for a product detail page on an ATG Commerce site, this would typically be `product`. Note that an individual `DynamicSitemapGenerator` component can use only a single item type, so if you want your sitemap to include pages based on different item types (e.g., product pages and category pages), you need to configure a separate `DynamicSitemapGenerator` for each item type.
`transactionManager`	The transaction manager to use. Typically `/atg/dynamo/transaction/TransactionManager`.
`numberOfItemsPerTransaction`	The number of repository items to process in each transaction.
`template`	A URL template component that translates URLs for inclusion in sitemaps. Typically this is a component of class `atg.repository.seo.IndirectUrlTemplate`, which translates dynamic URLs to their static equivalents. See Using URL Templates for more information.
`sitemapFilePrefix`	A String used to form the names of the dynamic sitemap files. If a single file is generated, `.xml` is appended to this String to form the filename (e.g., if `sitemapFilePrefix=dynamicSitemap`, the resulting filename is `dynamicSitemap.xml`). If multiple files are generated (because the maximum number of URLs or maximum file size is exceeded), `2.xml`, `3.xml`, and so on are appended to the second and subsequent files (e.g., `dynamicSitemap2.xml`, `dynamicSitemap3.xml`, etc.). Note that the value of `sitemapFilePrefix` must be unique for each sitemap generator component, to prevent overwriting of files.
`urlPrefix`	String to prepend to the URLs created by the URL template. This should include the protocol, domain, and port (if needed). If the `webApp` property is null, `urlPrefix` should also include the context root; for example: `http://www.example.com/mywebapp/`
`webApp`	The Nucleus pathname for the component of class `atg.service.webappregistry.WebApp` that represents the web application that the sitemap is generated for; for example: `/atg/registry/webappregistry/MyWebApp` The `DynamicSitemapGenerator` examines the web application to find the context root to append to `urlPrefix`. If you include the context root in `urlPrefix`, leave `webApp` null.

Configuring the SitemapIndexGenerator

The `atg.sitemap.SitemapIndexGenerator` class generates sitemap indexes. This class creates a sitemap index containing a list of all of the sitemap files generated by the corresponding `SitemapGenerator` components.

To configure a `SitemapIndexGenerator` component, set the following properties:

Property	**Description**
`siteIndexFilename`	The name of the generated sitemap index file; for example, `sitemap.xml`.
`urlPrefix`	String to prepend to the sitemap filenames to form the URL entries included in the sitemap index. This should include the protocol, domain, and port (if needed). If the webApp property is null, `urlPrefix` should also include the context root; for example: `http://www.example.com/mywebapp/`
`webApp`	The Nucleus pathname for the component of class `atg.service.webappregistry.WebApp` that represents the web application that the sitemap is generated for; for example: `/atg/registry/webappregistry/MyWebApp` The `SitemapIndexGenerator` examines the web application to find the context root to append to `urlPrefix`. If you include the context root in `urlPrefix`, leave webApp null.

Additional Configuration for Multisite Applications

For a multisite application, each URL in a sitemap must identify the site it is associated with. The same product can have multiple URLs if it is available on multiple sites.

`StaticSitemapGenerator` and `DynamicSitemapGenerator` components invoke the `getSites()` method of the `SitemapGeneratorService` to determine the sites to generate URLs for. To configure the `SitemapGeneratorService` so it can obtain this information, set the following properties:

Property	**Description**
`siteUrlManager`	The site URL manager component. This should be set to `/atg/multisite/SiteURLManager`.
`siteContextManager`	The site context manager component. This should be set to `/atg/multisite/SiteContextManager`.

`activeSitesOnly`	If `true`, indicates that URLs should be generated only for active sites. Default is `true`.
`enabledSitesOnly`	If `true`, indicates that URLs should be generated only for enabled sites. Default is `true`.

Note that if `activeSitesOnly` is set to `true`, `enabledSitesOnly` is ignored, because an active site is always enabled, but an enabled site may not be active. If both `activeSitesOnly` and `enabledSitesOnly` are set to `false`, then URLs are generated for all sites.

Configuring Sitemap Writing

The `atg.sitemap.SitemapWriterService` class writes sitemaps and sitemap indexes out to XML files. The ATG platform includes a component of this class, `/atg/sitemap/SitemapWriterService`. Typically you need to run a component of this class on each page-serving ATG instance in your production environment.

To configure a `SitemapWriterService` component, set the following properties:

Property	**Description**
`sitemapRepository`	The repository that stores the sitemaps and the sitemap index. This should be set to `/atg/sitemap/SitemapRepository`.
`sitemapPropertiesManager`	A component that maps properties in the `SitemapRepository` to the names used in Java code. This should be set to `/atg/sitemap/SitemapPropertiesManager`.
`sitemapTools`	A component with a utility methods for looking up and modifying items in `SitemapRepository`. This should be set to `/atg/sitemap/SitemapTools`.
`warDir`	The operating-system pathname of the deployed WAR file that the sitemap is generated for; for example: `C:\jboss-eap-4.2\jboss-as\server\atg\deploy\ATG.ear\mywebapp.war` The `SitemapWriterService` writes sitemaps files to the top-level directory of the web application, as recommended by `sitemaps.org`.

In addition to these sitemap-related properties, `SitemapWriterService` also has several properties it inherits from `atg.service.scheduler.SingletonSchedulableService`. See Invoking Sitemap Generation and Writing for more information.

Invoking Sitemap Generation and Writing

There are three ways to invoke the `SitemapGeneratorService` and `SitemapWriterService`:

- Scheduling them to run automatically at specified times or intervals
- Configuring a deployment listener so they are run automatically after a CA deployment
- Invoking them manually through ATG Dynamo Server Admin

After you generate and write out your sitemaps, you must submit them to search engines for indexing. For more information, see `sitemaps.org`.

Scheduling

The `SitemapGeneratorService` and the `SitemapWriterService` classes both extend `atg.service.scheduler.SingletonSchedulableService`, so you can schedule components of these classes to run automatically at specified times or intervals. Because these are singleton services, only one instance of each class can run on your ATG environment at any given time.

In general, you need to run the `SitemapGeneratorService` only after your site is updated. If your site is updated frequently, you might want to configure the service to run once a day at a time when site activity is low. If your site is updated infrequently, rather than scheduling the service, you can invoke it manually through ATG Dynamo Server Admin or configure a deployment listener to run the service after a CA deployment.

The `SitemapWriterService`, on the other hand, can be configured to run frequently. When this service starts up, it checks when the `SitemapRepository` was last modified. If the repository has been modified since the `SitemapWriterService` last ran, the service runs and writes out the updated sitemaps. If the repository has not been modified since the service last ran, it immediately shuts down.

This mechanism ensures that the `SitemapWriterService` runs only if the site has actually changed, and that (assuming the `SitemapWriterService` is scheduled to run frequently) there is never a long delay between running the `SitemapGeneratorService` and running the `SitemapWriterService`. Also, running the `SitemapWriterService` on a schedule (rather than invoking it manually) is desirable because you need to run a separate instance of this service on each page-serving ATG instance in your production environment (unlike the `SitemapGeneratorService`, which can run on a single ATG instance).

To configure the `SitemapGeneratorService` or the `SitemapWriterService` to run automatically, you set various scheduling and locking properties. See the *ATG Programming Guide* for information about configuring schedulable services.

Configuring a Deployment Listener

The `PublishingAgent.base` module includes a deployment listener that can trigger the `SitemapGeneratorService` to run after a CA deployment. This component is `/atg/epub/SitemapGeneratorPolicy`, and by default it is configured like this:

```
$class=atg.deployment.agent.DeploymentMethodInvocationPolicyImpl

object=/atg/sitemap/SitemapGeneratorService
methodName=generateSitemaps
deploymentState=DEPLOYMENT_COMPLETE
failDeploymentOnInvocationError=false
active=false
```

To enable this component, set the `active` property to `true`.

Manual Invocation through ATG Dynamo Server Admin

You can invoke the `SitemapGeneratorService` or the `SitemapWriterService` manually though ATG Dynamo Server Admin. The top-level page has a Sitemap Administration link that takes you to a page with Generate Sitemaps and Write Sitemaps links.

By default, the Generate Sitemaps link invokes `/atg/sitemap/SitemapGeneratorService`, and the Write Sitemaps link invokes `/atg/sitemap/SitemapWriterService`. To configure these links to invoke different components, change the values of the `sitemapGeneratorService` and `sitemapWriterService` properties of `/atg/sitemap/SitemapGeneratorFormHandler`.

SEO Tagging

Web search engines partly base their rankings of pages on the words that appear in certain HTML tags, particularly `<meta>` tags and the `<title>` tag. A common SEO technique is to list key search terms in those tags, to raise the ranking of the pages for those terms.

The ATG platform includes an SEO tag repository for storing the content of these tags. This repository has a single item type, `SEOTags`. An `SEOTags` item has the following properties, whose values are used to create HTML tags:

- `title` -- String used to set the body of the `<title>` tag; for example:

  ```
  Welcome to example.com, home of bargain clothing and shoes!
  ```

  This value can be up to 254 characters long. However, Google and MSN will consider only the first 66 characters, while Yahoo! will consider the first 115.

- `description` -- Used to set the `content` attribute of a `<meta name="description" ...>` tag; for example:

  ```
  example.com offers the finest women's clothing and shoes at low prices.
  ```

  This value can be up to 254 characters long.

- `keywords` -- Used to set the `content` attribute of a `<meta name="keywords" ...>` tag; for example:

  ```
  shoes, women's shoes, dresses, skirts, pants, shorts, jackets, accessories
  ```

This value can be up to 254 characters long.

The property values shown in these examples would result in the following tags:

```
<title>Welcome to example.com, home of bargain clothing and shoes!</title>
<meta name="description" content="example.com offers the finest women's clothing
and shoes at low prices." />
<meta name="keywords" content="shoes, women's shoes, dresses, skirts,
pants, shorts, jackets, accessories" />
```

In addition to `title`, `description`, and `keywords`, `SEOTags` items have three other properties:

- `displayName` -- The display name for the item, used in ATG Merchandising and the ATG Control Center.
- `key` -- An arbitrary identifier used to look up the item. You typically give each `SEOTags` item a unique `key` to ensure the correct content is rendered.
- `sites` -- For multisite applications, a comma-separated list of the sites the tag applies to. You can write page code to use this property to determine the tags to display for a given site.

Creating SEO Tags

If your ATG installation includes ATG Commerce and ATG Content Administration, you can create SEO tags by editing the versioned SEO tag repository in ATG Merchandising. This repository is deployed to the staging or production environment when the catalog is deployed. See the *ATG Merchandising User Guide* for more information.

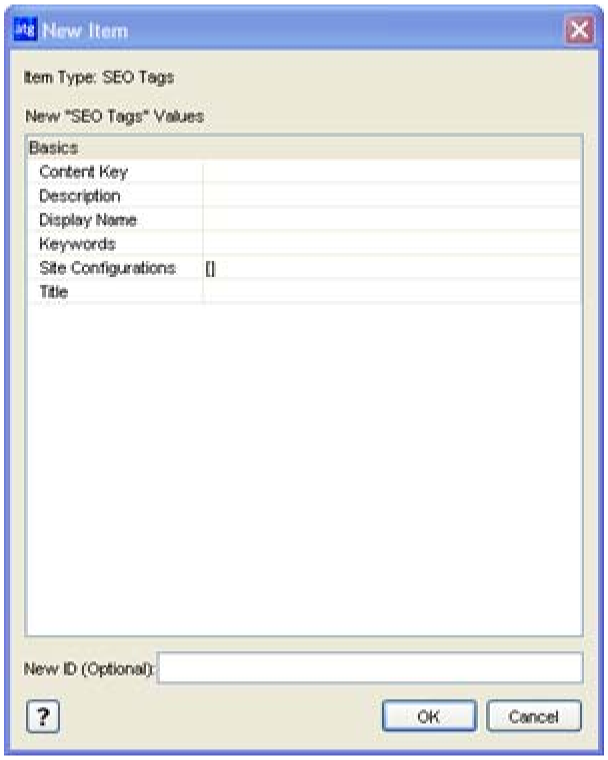
If your ATG installation does not include ATG Commerce or ATG Content Administration, you can create SEO tags by editing the non-versioned SEO tag repository directly on the staging or production environment, using the ATG Control Center:

1. Register the SEO tag repository by adding it to the list of repositories in the `initialRepositories` property of the `/atg/registry/ContentRepositories` component. To do this, in `<ATG10dir>/home/localconfig/atg/registry` create a `ContentRepositories.properties` file containing the following:

   ```
   initialRepositories+=/atg/seo/SEORepository
   ```

2. Start up your ATG application and the ATG Control Center.
3. From the navigation menu, select **Content** > **SEORepository**.

This takes you to an editor where you can view, create, and modify `SEOTags` items. For example, if you click **New Item**, the following window opens:



Item Type: SEO Tags

New "SEO Tags" Values

Basics	
Content Key	
Description	
Display Name	
Keywords	
Site Configurations	[]
Title	

New ID (Optional):

Fill in the values and click **OK** to create the item.

Rendering SEO Tags on Pages

To render SEO tags on a page, you pass the value of the key property of an SEOTags item to the RQLQueryRange servlet bean. This servlet bean finds an SEOTags item with that key value and uses the other properties of the item to supply the content for the tags.

The following example queries the repository for an item whose key property has the value "featured". The other properties of the returned item are then used to render the <title> tag and <meta> tags.

```
<dsp:droplet name="/atg/dynamo/droplet/RQLQueryRange">
   <dsp:param name="repository" value="/atg/seo/SEORepository" />
   <dsp:param name="itemDescriptor" value="SEOTags" />
   <dsp:param name="howMany" value="1" />
   <dsp:param name="mykey" value="featured" />
   <dsp:param name="queryRQL" value="key = :mykey" />

   <dsp:oparam name="output">

     <title><dsp:valueof param="element.title"/></title>

     <dsp:getvalueof var="description" param="element.description"/>
```

```
      <dsp:getvalueof var="keywords" param="element.keywords"/>

      <meta name="description" content="${description}" />
      <meta name="keywords" content="${keywords}"/>

   </dsp:output>
</dsp:droplet>
```

Note that the howMany parameter is set to 1 to ensure that only one set of tags is rendered. In general, you should make sure that the key property of each SEOTags item is unique.

The approach shown in the example above is useful if you have multiple pages using the same tag. You can include this servlet bean call in each of these pages, and they will all create tags use the SEOTags item whose key is "featured".

If you have pages that each require a unique set of tags (and therefore a unique SEOTags item), a better approach is to set each SEOTags item's key to a page-specific portion of the page URL, such as the servlet path. The servlet path does not include the protocol, domain, port, context root, or query arguments, and typically looks similar to this:

```
        /browse/category.jsp
```

In the servlet bean call, you dynamically evaluate the servlet path and pass that value to the servlet bean as the key. This approach allows you to use the same call in multiple pages, rather than having to hard-code the key in each one individually.

The following example illustrates this approach. A parameter named pageURL is set to the servlet path of the originating request. The pageURL parameter is then used to construct RQL to query the repository for an SEOTags item whose key value is that servlet path.

```
<dsp:droplet name="/atg/dynamo/droplet/RQLQueryRange">
   <dsp:param name="repository" value="/atg/seo/SEORepository" />
   <dsp:param name="itemDescriptor" value="SEOTags" />
   <dsp:param name="howMany" value="1" />
   <dsp:param name="pageURL" bean="/OriginatingRequest.servletPath" />
   <dsp:param name="queryRQL" value="key = :pageURL" />
   ...
</dsp:droplet>
```

Note that the key should be based on the actual (dynamic) page URL, not the static URL created through the URL recoding feature. When a spider accesses a page through a recoded (static) URL, the static URL is translated by the SEO jump servlet back to its dynamic equivalent. So the page is actually served using the dynamic URL.

16 DAF Deployment

ATG includes a deployment system you can use to deploy repository and file-based assets from one server cluster to another—typically, from a development environment to a cluster that represents a staging or production site. DAF deployments are designed to optimize performance and scalability by using multi-threading in a multi-server configuration. Error recovery is supported; you can resume a deployment after an error occurs, or after a deployment is halted manually, without restarting the entire operation. If your deployment setup includes ATG Content Administration, you can also roll the target site back to a previous state.

ATG Content Administration is a content and deployment management system that uses DAF deployment. For more information, see the *ATG Content Administration Programming Guide*.

Non-Versioned and Versioned Deployments

DAF deployment can be used to deploy content from non-versioned and versioned repositories:

- You can deploy from multiple non-versioned GSA repositories, where each source repository has a corresponding target repository. DAF deployment can also deploy file items from one or more non-versioned virtual file systems to corresponding virtual file systems on the target site.
- ATG Content Administration uses DAF deployment to deploy data in versioned repositories and files to non-versioned repositories and virtual file systems, respectively.

In this chapter

This chapter contains the following sections:

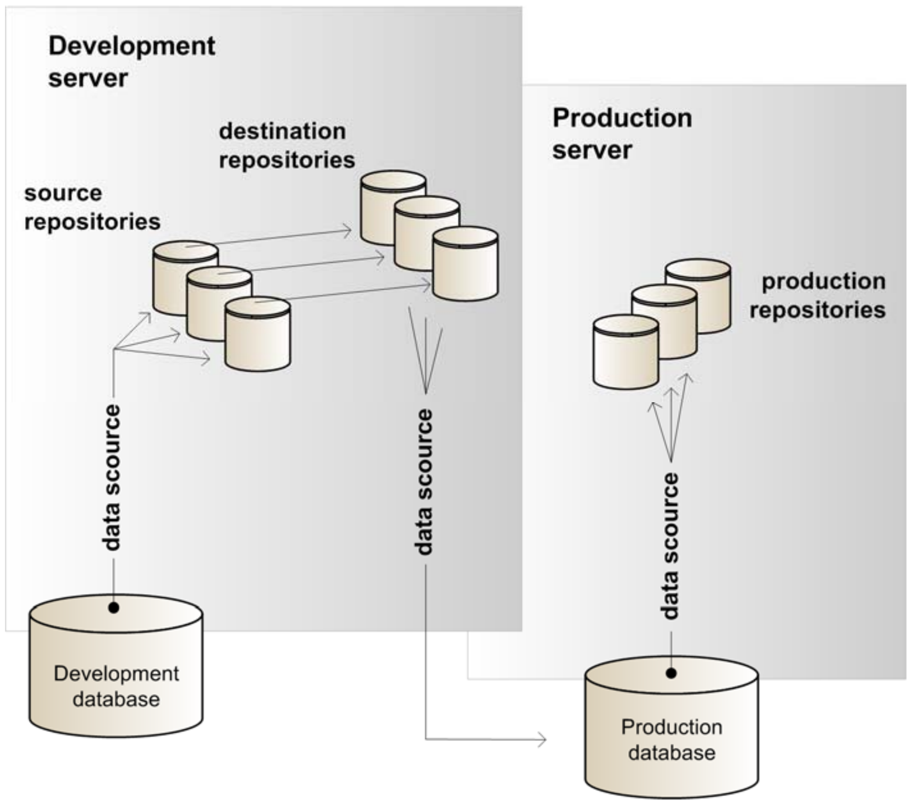
DAF Deployment Architecture

DAF deployment uses multi-threading to move data from a source to a target server. The number of threads and the number of assets per thread are configurable, which lets you tailor deployment performance to available hardware.

In addition to using multiple threads, the work of deploying data can be split among many servers. This clustering capability allows the deployment mechanism to scale as the number of assets to deploy increases.

For repository assets, DAF deployment uses the JDBC driver to connect directly from the source server—for example, ATG Content Administration—to the target database, allowing it to read and write repository assets without converting the data into an intermediate format. File system assets are transferred across the network using a TCP connection.

The following diagram shows the database, data source, and repository setup for instances of ATG servers using DAF deployment.



Each development instance can have multiple source repositories. For example, an ATG Commerce environment might have a product catalog repository and a price list repository. Each development instance also must have a destination repository that is configured for the target environment. The previous example shows just one production target. For examples of configurations with multiple targets, see the *ATG Content Administration Programming Guide*.

Each deployment is initiated by the Nucleus component `/atg/deployment/DeploymentManager`, which spawns a thread to start the deployment process, logs information about the deployment, and returns control to the caller. The deployment process continues asynchronously, as follows:

1. Persist asset data.
2. Send a message to DeploymentManager instances on the asset management servers to start the deployment.
3. For each asset management server, spawn two types of threads:
   - `RepositoryWorkerThreads` process repository assets
   - `FileWorkerThreads` process file assets.

   The number of threads of each type is determined by the ratio of repository and file assets to deploy.

Deployment transactions are performed in batches to avoid memory problems. After a given number of operations (specified by the DeploymentManager's `transactionBatchSize` property), a thread tries to commit its current set of operations as a transaction batch. If the commit succeeds, the thread requests another batch and continues until all batches are processed.

Note: Transactions cannot span threads.

DeploymentManager

The deployment process is initiated by the `/atg/deployment/DeploymentManager` component, which is also responsible for sending JMS messages that signal the start of each subsequent deployment phase. The following code sample shows the properties file for the default DeploymentManager component:

```
# @version $Id: //product/DAF/main/Deployment/config/atg/deployment/
DeploymentManager.properties
$class=atg.deployment.DeploymentManager

deploymentRepository=DeploymentRepository
transactionManager=/atg/dynamo/transaction/TransactionManager
lockManager=/atg/dynamo/service/ClientLockManager
messagingSource=/atg/deployment/messaging/DeploymentMessagingSource
serverNameService=/atg/dynamo/service/ServerName
clusterNameService=/atg/dynamo/service/ClusterName

transactionBatchSize=250
threadBatchSize=1000
maxThreads=10
maxFailureCount=0

loggingDebug=false
```

```
loggingThreadDebug=false
loggingItemDebug=false
loggingPropertyDebug=false
loggingFileDebug=false

phaseCompletePollingInterval=15000
threadSpawnInterval=1000

repositoryMappings=
```

The following table describes key DeploymentManager properties:

Property	Description
`deploymentRepository`	The location of the repository used to store deployment status and management data.
`messagingSource`	The Nucleus component that sends the JMS messages used to indicate the status of each deployment phase. Default: `/atg/deployment/messaging/DeploymentMessagingSource`
`maxFailureCount`	The number of errors that are allowed before a deployment is declared a failure. By default, the deployment fails on the first error. In some cases, particularly during development or testing, you might want to increase this value so that a deployment can continue even if errors are found. Default: 0
`phaseCompletePollingInterval`	The frequency in milliseconds of DeploymentManager queries, which determine whether a deployment phase is complete and the next phase is ready to be launched. Default: 15000 For local deployment, the DeploymentManager uses the `localDeploymentPhaseCompletePollingInterval` value, whose default setting is 750. To expedite large deployments, decrease polling frequency by increasing the value of these properties.
`purgeDeploymentData`	Specifies whether to delete from the deployment repository `marker` and `deploymentData` items associated with this deployment when deployment is complete. Default: True

`maxThreads`	The maximum number of threads that can be spawned for each deployment. Increase the value as the number of assets in the system increases. The value of this property must be less than the number of connections specified in your datasource. For more information, see Configuring DAF Deployment for Performance. Default: 10
`transactionBatchSize`	The number of items that each transaction can process. See Configuring DAF Deployment for Performance for more information. The `transactionBatchSize` is ignored if it is greater than the `threadBatchSize`. Default: 250
`threadBatchSize`	The maximum number of items that can be assigned to each thread. By default this number is 1000, which means that for a deployment containing 2000 items, 2 threads are created. For 5000 items 5 threads are created, and so on until the `maxThreads` limit is reached. Default: 1000
`useDistributedDeployment`	Specifies whether to perform distributed or local deployments. See Enabling Distributed Deployments. Default: False

Enabling Distributed Deployments

By default, deployment events are sent locally only to the server that initiated the deployment. In order to use distributed deployment, where deployment events are sent as JMS messages to configured listeners, set the `useDistributedDeployment` property to true. If you enable distributed deployment but the number of assets to deploy is less than or equal to the number of assets assigned to each thread, deployment is local only, because only one thread is required. By default, `threadBatchSize` is set to1000, so all deployments of 1000 assets or less are always local. This behavior can significantly improve performance for small deployments. The default setting of `useDistributedDeployment` is false.

Configuring Error Logging and Debugging

The DeploymentManager includes several properties that determine the amount and type of error and debugging information that is displayed in the ATG console, and saved to the `debug.log` file in the `<ATG10dir>\home\logs` directory:

Property	Logs this information:
`loggingDebug`	Component-level debugging Default: False
`loggingThreadDebug`	Thread-level debugging Default: False
`loggingItemDebug`	Debugging for repository items Default: False
`loggingPropertyDebug`	Debugging at repository item property level Default: False
`loggingFileDebug`	Debugging for file assets Default: False

Enabling the properties `loggingItemDebug`, `loggingPropertyDebug`, and `loggingFileDebug` can result in very large amounts of output, depending on the number of assets in your system. For performance reasons, set these flags to true only during development and testing.

You can also use the DAF Deployment API to obtain status and error information as it is generated.

Deployment Phases

Deployment has the following phases:

1. Deployment start
2. Pre-deployment
3. Add/Update
4. Reference resolution
5. Delete
6. Destination synchronization
7. Deployment completion

The following sections describe these phases, and the JMS messages that the DeploymentManager sends to coordinate each phase across multiple deployment servers.

Deployment Start

When a call to the `deploy()` method of the DeploymentManager is received, the following actions occur:

1. The DeploymentManager sends a `START_DEPLOYMENT` message. No deployment engine processing occurs at this point; the message exists principally to allow custom integration with the deployment system.

   Note: if another deployment is in progress when the call to the `deploy()` method is received, the second deployment is queued.

2. A thread is spawned that manages the remaining deployment process. This thread is the main or management thread; there is only one instance, and it resides on the calling machine.

3. Control returns to the caller. The `DeploymentData` objects that are passed into the `deploy()` method are retained by the management thread. They should not be altered by any client code.

4. In the management thread, the DeploymentManager writes `DeploymentData` and `Markers` to the deployment repository. This process is controlled by the DeploymentManager where the `deploy()` call was made.

   To write the data efficiently to the deployment repository, the DeploymentManager spawns no more than `maxThreadCount` number of worker threads, assigning each thread `DeploymentData` and `Marker` objects up to a maximum of `threadBatchSize`.

   Before starting the worker threads, the DeploymentManager sends a `MARKER_PERSISTENCE` message.

Pre-deployment

After the `threadBatchSize` number of `Marker` objects are written, deployment enters the pre-deployment phase. With a full deployment, all repository items are deleted during this phase from the target repositories. Depending on the configuration (specifically, when the `/atg/deployment/file/DeploymentConfiguration.noVerificationByChecksum` option is enabled), all files can also be deleted from the target virtual file systems.

When this phase is complete, the DeploymentManager sends an `ADD_UPDATE_PHASE` message that triggers the start of the Add/Update phase.

Add/Update

This phase begins when an `ADD_UPDATE_PHASE` message is sent to a deployment-specific JMS topic. The message contains the ID of the deployment associated with the message. Instances of the DeploymentManager on each server are subscribed to that topic and begin processing on receipt of this message; this includes the DeploymentManager component on the server that manages the deployment process.

During this phase, the following actions occur:

- All repository items whose action property value is add or `update` are created.
- All properties with primitive types are set on items whose action is add or `update`.
- Properties of repository items that are required references are set to a temporary dummy item. Only one dummy item of each referenced item descriptor is created; these are deleted at the end of the deployment.

- Properties marked as `deployable=false` are not deployed.
- Files whose action is add or `update` are copied to the responsible deployment agents.

When these actions are complete, the `Marker` item status is set to `ADD_UPDATE_COMMITTED`.

If an error occurs, the current transaction is rolled back. In a separate transaction, the `Marker` status for the item that failed in the current batch is set to `FAILURE`. The deployment's `failureCount` property is incremented, and the `Marker` is removed from the worker thread's list of items to operate on. The transaction is restarted from the beginning, skipping the item that failed. When the `failureCount` is greater than the `maxFailureCount` property configured in the DeploymentManager, the entire deployment fails.

After the management thread determines that the status of all `Markers` is `ADD_UPDATE_COMMITTED`, it starts the next deployment phase.

Reference Resolution

During this phase, all repository items with an action of add or `update` that contain references to other items, either a single reference or a collection of references, are resolved. To start this phase, the management thread sends a `REFERENCE_UPDATE_PHASE` message to the deployment-specific JMS topic. As in the first phase, each DeploymentManager starts all its threads and begins querying the deployment repository for work.

At the end of this phase, each `Marker` that has been processed has its status changed to `REFERENCES_COMMITTED`. Any `Marker` whose action is `delete` also has its status changed to `REFERENCES_COMMITTED`.

For file assets, this deployment phase is ignored.

Delete

This phase starts when the management thread sends a `DELETE_PHASE` message. All worker threads are started, and each one requests a series of `Markers` to process. In this phase, only `Markers` whose action is `delete` are processed. For those, the specified items are deleted from their repository. If an error occurs, the transaction is rolled back and all `Marker` statuses are set to FAILURE.

When all deletions are complete, control returns to the management thread, which sets the deployment's status field to `DATA_DEPLOYMENT_COMPLETE`.

Destination Synchronization

During this phase, all destination repositories are synchronized by invalidating the caches for all the repositories on the target instances.

Note: For large file deployments in certain configurations (when the `/atg/deployment/file/DeploymentConfiguration.noVerificationByChecksum` option is disabled, or on the second apply phase of a switched deployment when ATG Content Administration is being used), operations that occur on the target can be time consuming.

Deployment Completion

The management thread sets the status of the deployment to `DEPLOYMENT_COMPLETE` and sends a `COMPLETE_DEPLOYMENT` message to all the DeploymentManager components. If the `purgeDeploymentData` flag is true in the deployment repository item, the `Marker` and `deploymentData` items are removed from the repository before the status is set.

DAF Deployment API

The following classes and interfaces comprise the DAF Deployment API:

- `atg.deployment.DeploymentManager`
- `atg.deployment.DeploymentData`
- `atg.deployment.DeploymentOptions`
- `atg.deployment.DeploymentProgress`
- `atg.deployment.DeploymentReporter`
- `atg.deployment.DeploymentFailure`

For detailed information, see the online *ATG API Reference*.

atg.deployment.DeploymentManager

As described earlier, implementations of the `atg.deployment.DeploymentManager` class are used to initiate a DAF deployment and manage the deployment process. The following methods are available:

- `deploy` starts a new deployment.
- `cancel` cancels a running deployment.
- `resume` resumes a deployment that failed or was cancelled.
- `restart` restarts a deployment that failed or was cancelled.
- `isDeploymentRunning` determines whether a given deployment is running.
- `deploymentExists` determines whether the specified deployment exists in the deployment repository.
- `getDeploymentProgress` returns a `DeploymentProgress` object, which contains information about the number of items that were deployed and the total number of items in the deployment.
- `getDeploymentReporter` returns a `DeploymentReporter` object, which contains information about specific items that were successfully deployed or failed deployment.
- `purgeDeploymentData` removes the `DeploymentData` and `Markers` from the `DeploymentRepository` for the specified deployment.

atg.deployment.DeploymentData

A `DeploymentData` object is passed into the `DeploymentManager deploy()` call and defines the source and destination for a single set of data as well as identifying the actual data to be deployed. `atg.deployment.DeploymentData` is a marker interface that has two subclasses that provide the API for their respective types of deployment data object:

- `RepositoryDeploymentData` generates a list of repository items for deployment.
- `FileDeploymentData` generates a list of files for deployment.

Whenever add methods of either subclass are called, the `DeploymentData` object creates a `Marker` object, which is internal to the deployment system and should not be used by the caller. Each `Marker` object represents a single item to be deployed.

The constructors of both subclasses create a `DeploymentData` object. The subclasses also contain these methods:

- `addNewItem` and `addNewFile` methods deploy a new item and file to the target.
- `addItemForUpdate` and `addFileForUpdate` methods deploy a changed item and file (one that exists already on the target).
- `addItemForDelete` and `addFileForDelete` methods delete an item from the target.

Note: These classes are not thread-safe and should be used only by a single thread.

atg.deployment.DeploymentOptions

The `atg.deployment.DeploymentOptions.addOption` method supplies various deployment-wide settings to the DeploymentManager's `deploy()` method. You specify options as one of the following constants:

Constant	**Effect**
`FULL_DEPLOYMENT`	Causes a full deployment, where only add operations are included in the `DeploymentData`, and the deployment deletes any information in the target repositories or virtual file systems that is not included in the `DeploymentData`.
`CODE_STRICT_REPOSITORY_OPERATIONS`	Constrains repository operations so that an add fails if the item exists, and update or delete fails if the item does not exist. If you omit this option, these situations trigger warnings rather than failures. You can also require this behavior through the DeploymentManager property `strictRepositoryOperations`.

Constant	Effect
`CODE_STRICT_FILE_OPERATIONS`	Constrains file operations so that an add fails if the file exists and an update or delete fails if the file does not exist. If you omit this option, these situations trigger warnings rather than failures. You can also require this behavior through the DeploymentManager property `strictFileOperations`.
`CODE_PRESERVE_FILE_TIMESTAMPS`	File operations set the timestamp of changed files to the timestamp recorded in the source virtual file system. Use this setting with care, as it can confuse production systems such as JSP handling that require reliable timestamps. Specifying this option also sets the `noVerificationByChecksum` option for full deployments.
`CODE_NO_VERIFICATION_BY_CHECKSUM`	Files are always pushed to the target server. Without this option, a file is pushed during an add or update operation only if the file size or 64-bit file checksum does not match.

atg.deployment.DeploymentProgress

Methods of this class retrieve the following information about a deployment's status:

- `getWorkCompleted` obtains the number of items that were deployed so far (the current value of the `workCompleted` property of the `deploymentProgress` repository item).
- `getTotalWork` obtains the total number of items in the deployment (the current value of the `totalWork` property of the `deploymentProgress` repository item).
- `getTotalWork` obtains a human-readable message indicating the deployment phase currently in progress, for example `Updating item references`.

atg.deployment.DeploymentReporter

This class can be used to get information about any running or completed deployment. The `DeploymentReporter` returns a list of the items that were committed to the database or failed deployment. It can also show the number of committed and failed items.

The class has the following methods:

- `getCommittedItems` gets a list all the items successfully deployed as part of the deployment with the given ID.
- `getModifiedItems` gets a list of items modified as part of the deployment with the specified ID.

- `getFailures` gets a list of all failed deployment items for the deployment with the specified ID.
- `getCommittedCount` gets the number of items that were committed to the database as part of the deployment with the specified ID.
- `getModifiedCount` gets the number of items that were modified as part of the deployment with the specified ID.
- `getFailedCount` gets the number of items that failed the deployment process as part of this deployment.

atg.deployment.DeploymentFailure

This class lets you retrieve detailed information about a specific deployment failure, including the type of operation, the time the failure occurred, and the item that was the subject of the operation. You can call the method `getSubject` to obtain the repository item that was the subject of the failed operation: either a `repositoryMarker` or a `fileMarker`. These items have the following properties:

`deploymentData`	A reference to the `DeploymentData` repository item, which has source and destination string properties that are the paths to the Nucleus component, either a repository or a virtual file system, where the failed item exists.

`status`	Enumeration set to one of these values:
	
	`Pending`
	`Initiated`
	`AddUpdateCommitted`
	`ReferencesCommitted`
	`Committed`
	`Failure`

`action`	Enumeration set to one of these values:
	
	`Add`
	`Update`
	`Delete`

`deploymentId`	The ID of this marker's deployment.
`itemDescriptorName`	Valid for `repositoryMarkers` only, the name of the specific item type that is represented by this `DeploymentFailure` object.
`itemId`	Valid for `repositoryMarkers` only, the item's repository ID.
`filePath`	Valid for `fileMarkers` only, the path of the failed file.

Deployment Repository

The deployment repository stores runtime and management data for deployments. The deployment repository is defined in `deployment.xml` in `<ATG10dir>\DAF\Deployment\config.jar`. This repository contains the following items:

- deployment
- deploymentProgress
- deploymentData
- marker
- repositoryMarker
- fileMarker
- failureInfo

deployment

Contains management information for a current or past deployment.

This item has the following properties:

Property	Description
`id`	The deployment ID.
`startTime`	The time the deployment was started.
`endTime`	The time the deployment ended.
`failureTime`	The time the deployment stopped because of an error.
`status`	The deployment status, one of the following: 0: `WAITING` 1: `RUNNING` 2: `MARKER_PERSISTENCE_COMPLETE` 3: `DATA_DEPLOYMENT_COMPLETE` 4: `DEPLOYMENT_COMPLETE` 5: `FAILURE` 6: `CANCEL`
`statusDetail`	Contains a user-readable message about the current status.
`currentPhase`	The phase that the deployment is passing through. Can be any of Start Deployment (0), Pre-Deployment (1), Add-Update Phase (2), Reference Update Phase (3), Delete Phase (4), Destination Synchronization (5), Complete Deployment (6)

Property	Description
`repositoryHighWaterMark`	A runtime value used by each repository thread to get the next set of markers to operate on.
`repositoryMarkersAvailable`	A runtime value representing the total number of repository markers for deployment.
`fileHighWaterMark`	A runtime value used by each file thread to get the next set of markers to operate on.
`fileMarkersAvailable`	A runtime value representing the total number of file markers for deployment.
`threadBatchSize`	The size by which to increase the `highWaterMark` when getting a new set of items to operate on.
`failureCount`	The number of items that failed in the deployment.
`purgeDeploymentData`	A boolean that signifies whether the `marker` and `deploymentData` items associated with this deployment should be purged from the deployment repository. The value comes from the `purgeDeploymentData` configuration property in the DeploymentManager. The default is true.
`deploymentData`	A list of `deploymentData` items that define a part of a deployment.
`deploymentOptions`	A map of the option flags set for this deployment through `/atg/deployment/DeploymentOptions`.

deploymentProgress

Represents the same information as the `DeploymentProgress` object.

This item has the following properties:

Property	Description
`id`	The deployment ID.
`workCompleted`	The number of items that successfully deployed for this deployment.
`totalWork`	The total number of items in the deployment.

See `atg.deployment.DeploymentProgress` for more information.

deploymentData

Represents the same information as the `DeploymentData` object.

This item contains the following properties:

Property	Description
`id`	The `deploymentData` ID.
`type`	Specifies whether the item handles file or repository items.
`source`	For `RepositoryDeploymentData` objects, the name of the source repository. For `FileDeploymentData` objects, the Nucleus path of the source component.
`destination`	For `RepositoryDeploymentData` objects, the name of the destination repository. For `FileDeploymentData` objects, the IP and port information for the target where the data should be deployed.
`markers`	A list of `marker` objects that contain information about each item to be deployed.

marker

Stores information about each individual item in the deployment.

There are two types of `marker` item descriptors, `repositoryMarker` and `fileMarker`, where each inherits properties from `marker` and contains information specific to the type of asset being deployed.

The `marker` item contains the following properties:

Property	Description
`id`	The `marker` ID.
`type`	Specifies whether this `marker` handles a repository (0) or file (1) item.
`status`	One of the following: 0: `PENDING` 1: `INITIATED` 2: `ADD_UPDATE_COMMITTED` 3: `REFERENCES_COMMITTED` 4: `COMMITTED` 5: `FAILURE`
`index`	The index of this `marker`. All indexes are unique within a deployment. The value is derived from the `highWaterMark` in the deployment object.
`deploymentData`	A reference to the `marker`'s `deploymentData` object

Property	Description
`action`	One of the following: 0: `ADD` 1: `UPDATE` 2: `DELETE`
`deploymentId`	The ID of the deployment for which this `marker` was created.
`deploymentData`	A reference to the `marker`'s `deploymentData` object.

repositoryMarker

Extends `marker` with the following properties:

Property	Description
`id`	The `repositoryMarker` ID.
`itemDescriptorName`	The item descriptor name of the repository item.
`itemId`	The ID of the item. For composite IDs, the property is a string separated by the configured ID separator character.

fileMarker

Extends `marker` with the following properties:

Property	Description
`id`	The `fileMarker` ID.
`filePath`	The path and name of the file in the virtual file system.

failureInfo

Stores data about deployment errors and has the following properties:

Property	Description
`id`	The `failureInfo` ID.

Property	Description
`deployment`	The ID of the associated deployment item.
`marker`	The ID of the associated `marker` item.
`severity`	One of the following: 0: `WARNING` 1: `ERROR`
`message`	The human-readable message displayed with the warning or error.
`time`	A timestamp showing the time when the warning or error occurred.
`errorCode`	The code associated with the warning or error.
`cause`	The exception that was generated when the failure occurred.

Setting Up DAF Deployment

This section describes how to configure DAF deployment for repository and file items.

Note: If you use ATG Content Administration, see the *ATG Content Administration Programming Guide* for information on configuring deployments.

Setting Up DAF Deployment for Repository Items

The procedure for configuring DAF deployment to deploy repository items requires you to create a `GSARepository` that matches your source repository, and then create a data source for the new repository that points to the database used by the repository that is the target for the deployment. See the diagram in DAF Deployment Architecture earlier for an illustration. It might also be helpful to refer to the deployment setup procedure *ATG Content Administration Programming Guide*, which contains a detailed example.

1. Configure a data source to point to the database used by the target repository. To do so, you can copy and rename the ATG server's `FakeXADataSource` and `JTDataSource` properties files, pointing the new `FakeXADataSource.properties` file to the target site's database. Put the files in the same location as the ATG server's `FakeXADataSource` and `JTDataSource`.

   For more information on the `FakeXADataSource` and `JTDataSource`, see the *ATG Installation and Configuration Guide*.

2. Create a destination `GSARepository` for each source repository/target repository combination. For example, if you have a source repository called `MyContent`, and a target repository, `TargetContent`, create an additional `GSARepository`, DestinationContent. To configure it, copy the source repository's properties file and rename it as appropriate. Change the value of the `repositoryName` property, and

change the `dataSource` property to point to the data source you created in step 1, for example:

```
repositoryName=DestinationContent
dataSource=/atg/dynamo/service/jdbc/TargetJTDataSource
```

Put these files in the `localconfig` directory on the ATG server.

3. Add the repository you just created to the `initialRepositories` property of the `localconfig/atg/registry/ContentRepositories.properties` file on the ATG server.

   Make sure the value you enter matches the `RepositoryName` property in the repository's properties file (rather than the name specified in the properties file itself).

4. Repeat steps 2 and 3 for each additional source/target repository pair.

Setting Up DAF Deployment for Files

Files are deployed directly from the appropriate source component on the ATG server to the component with the same path on the target. Thus, in the case of file deployment, there is no need to create additional destination components on the source server as there is for repository item deployments. Other than setting up the source and target VFS components, the only other step required for file deployments is as follows: by default, the `FileDeploymentServer` uses port 8810 to communicate with the target. Depending on your environment, you might need to open this port in your target server's firewall. If you need to change the port that is used, you can do so by adding an `/atg/deployment/file/FileDeploymentServer.properties` file to the target's `localconfig` directory, and setting the port as follows:

```
port=new-port-number
```

Setting Up DAF Deployment for Multiple ATG Servers

This section describes the additional steps you need to complete to set up DAF deployment for environments where you want to increase performance by using multiple servers to perform the deployment.

Note: Unless you use ATG Content Administration, you cannot deploy different data from multiple ATG servers clusters to a single target. The multiple server setup described here applies only for situations where multiple servers are used as a single cluster to deploy the same data. For information about configuring deployment from multiple server clusters, see the *ATG Content Administration Programming Guide*.

1. Set up a server lock manager and a client lock manager for the ATG server cluster (as for any other multi-server ATG configuration). For information, refer to the *ATG Installation and Configuration Guide*.

2. On each instance in the ATG server cluster, set the `/atg/dynamo/service/ServerName.properties` file to have `serverName` and `drpPort` properties that are unique for each instance in the cluster. For example, server A might be set as follows:

```
serverName=serverA:8850
drpPort=8850
```

Server B might be set with distinct values, as follows:

```
serverName=serverB:18850
drpPort=18850
```

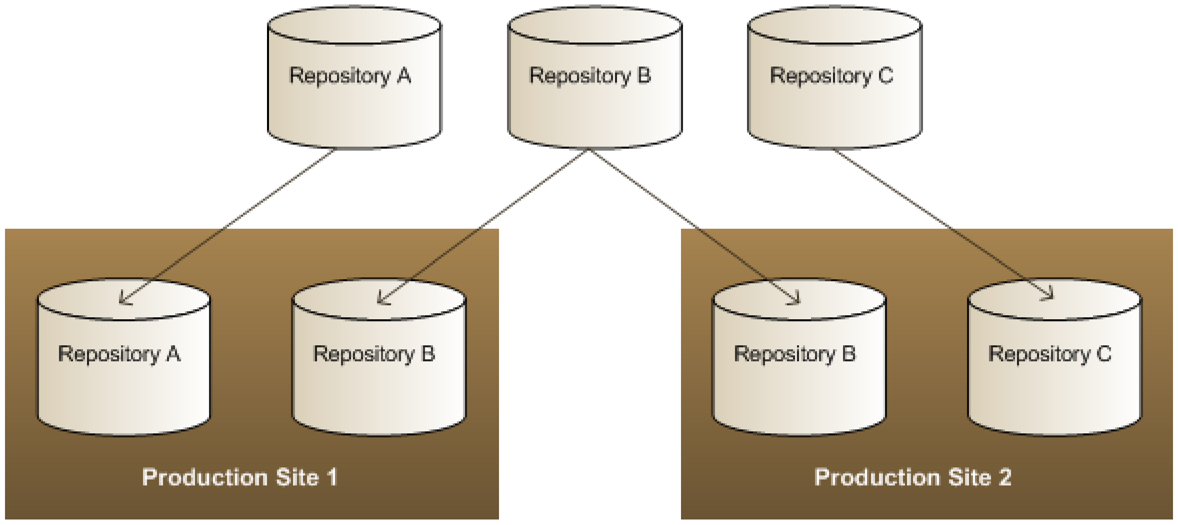
Caution: Do not run the `DAF.Deployment` module on any server that is used as a standalone lock manager. Doing so causes the lock manager to attempt to deploy items, and the deployment deadlocks as a result.

Using DAF Deployment to Deploy to Multiple Sites

The DAF deployment architecture lets you deploy from a single ATG server to multiple sites, including multiple sites on a single target. To do so, follow the steps in Setting Up DAF deployment for a Single ATG Server, creating corresponding repositories on the target server or servers for each repository that you want to deploy from the ATG server. For data that you do not want to deploy to a particular site or target, do not set up a matching repository on the target. The data is not deployed if no matching repository exists.

Any repositories that have data dependencies—for example, items that have links to other assets—must be deployed together.

The following diagram shows two target sites that have different data supplied by a single ATG server instance: Production Site 1 and Production Site 2. On Production Site 1, there is no matching repository for the Repository C that is configured on the source, so this target site does not receive Repository C data. Similarly, Repository A is not configured on Production Site 2, so this target does not receive data for Repository A.



Performing a Deployment

Note: For information on deployments with ATG Content Administration, refer to the *ATG Content Administration Programming Guide*.

The `DAF.Deployment` module is included by default on the ATG source server. The appropriate modules are also included by default on both the source and target servers if your environment includes ATG Content Administration. However, if you do not use ATG Content Administration, and you are deploying files, you must specify `DAF.DeploymentAgent` in the list of modules when you assemble your application for the target server. For more information, see Assembling Applications

To trigger a deployment for repository assets, create an object of type `RepositoryDeploymentData`, specifying a source and destination repository. For deploying assets from multiple source repositories, create a `DeploymentData` object for each pair of source and destination repositories. For each item you want to add, invoke `addNewItem()`. To update an item, use `addItemForUpdate()`. To delete an item, invoke `addItemForDelete()`.

For example:

```
RepositoryDeploymentData dd = new RepositoryDeploymentDataImpl(sourceRepository,
  targetRepository);
RepositoryDeploymentData dd2 = new RepositoryDeploymentDataImpl(sourceRepository2,
  targetRepository2);

dd.addItemForUpdate(myRepositoryItem);
dd.addItemForDelete(myOtherRepositoryItem);
dd2.addItemForUpdate(myRepositoryItem2);
dd2.addItemForDelete(myOtherRepositoryItem2);
```

```
DeploymentData[] dataArray = {dd, dd2};

DeploymentManager manager = (DeploymentManager)
  Nucleus.getGlobalNucleus().resolveName("/atg/deployment/DeploymentManager");

String deploymentId = manager.deploy(dataArray, new DeploymentOptions());
```

To deploy file assets, construct a `FileDeploymentData` object, which represents a particular agent or target process. It holds only the IP address of that machine, and the port number of the `FileDeploymentServer` on the running agent.

Create a separate `FileDeploymentData` object for each agent or target server that needs data from any particular virtual file system. Also, a separate `FileDeploymentData` object should be created for each distinct VirtualFileSystem component targeted on a particular agent process.

Use the `**byURI` functions when deploying from a versioned `ContentRepositoryFileSystem`. The URI passed in is the String representation of the version manager URI. Use the other functions when deploying non-versioned data.

The following example deploys the same data from a single non-versioned virtual file system to two target agents simultaneously:

```
VirtualFileSystem sourceSystem = (VirtualFileSystem)
  Nucleus.getGlobalNucleus().resolveName("/com/path/SourceVFS");

FileDeploymentDestination dest1 =
  new FileDeploymentDestination("server1.company.com:8810");
FileDeploymentDestination dest2 =
  new FileDeploymentDestination("server2.company.com:8810");

FileDeploymentData fdd1 =
  new FileDeploymentData(sourceSystem, dest1); FileDeploymentData fdd2 =
  new FileDeploymentData(sourceSystem, dest2);

fdd1.addNewFile(myVirtualFile1); fdd2.addNewFile(myVirtualFile1);
fdd1.addFileForUpdate(myVirtualFile2);
fdd2.addFileForUpdate(myVirtualFile2);
fdd1.addFileForDelete(myVirtualFile3);
fdd2.addFileForDelete(myVirtualFile3);

DeploymentData[] dataArray = {fdd1, fdd2};

DeploymentManager manager = (DeploymentManager)
  Nucleus.getGlobalNucleus().resolveName("/atg/deployment/DeploymentManager");

String deploymentId = manager.deploy(dataArray, new DeploymentOptions());
```

See `atg.deployment.DeploymentData` for more information.

Performing Switch Deployments

Switch deployments, where a target server has both an active and an offline database and data is sent to an offline database, are not currently supported through the DAF deployment API. For information on configuring switch deployments for an ATG Content Administration environment, refer to the *ATG Content Administration Programming Guide*.

Configuring DAF Deployment for Performance

This section suggests some configuration settings that can help ensure optimal performance for the DAF deployment system.

- Increase the JVM heap size as the number of assets in the system increases.
- Increase the value of the `maxThreads` property in the `/atg/deployment/DeploymentManager` component as the number of assets in the system increases. The default value is 10. However, increasing the value beyond 20 for UNIX systems and 10 for Windows is not recommended. Use additional ATG Content Administration server instances instead.

  Note: The value of `maxThreads` must be less than the number of connections specified in your datasource.
- Make sure that the number of connections to the source and destination repositories is larger than the value of the `maxThreads` property.
- Do not increase the `transactionBatchSize` property in the DeploymentManager to a large value because doing so requires transaction management to use more database memory, and performance of the database can be degraded as a result.
- If your environment uses Oracle databases, perform statistics gathering after the initial data load, on both the source and target databases.

For information on using checksum caching to optimize deployment for file assets, refer to the *ATG Content Administration Programming Guide*.

17 Content Distribution

To achieve faster performance in a large ATG application, it is best to keep as much content cached on the HTTP server as possible. This shortens the request handling process. ATG's content distributor feature manages content across multiple HTTP and ATG servers, pushing content from the ATG document root to the HTTP servers' document roots.

The ATG content distributor system consists mainly of three main components of:

- **DistributorSender** components are installed on ATG servers open TCP/IP connections to a list of DistributorServer components and send put file and file check commands.
- **DistributorServer** components are installed with HTTP servers. They accept TCP/IP connections from the DistributorSender components and create DistributorReader components to handle each such TCP/IP connection.
- **DistributorReader** components handle the input from the DistributorSender, writing the files they receive to the HTTP server's local file system or the document cache.

In this chapter

This chapter includes the following sections:

Content Distribution Operation

In a content distributor system, a `DistributorServer` component is installed with each HTTP server. Each ATG server includes one or more `DistributorSender` components or `DistributorPool` components (a resource pool of `DistributorSender` components) that are configured to connect to the `DistributorServer` component on each HTTP server.

When the `putFile` methods of a `DistributorSender` are invoked, the `DistributorSender`:

1. Generates a file name and directory to use on the remote servers.
2. Checks whether the file was sent to the remote servers:
   - Checks locally in its document cache (`RemoteServerCache`)

- Checks remotely, issuing a `fileCheck` command to the remote `DistributorServer` components.

The check is performed by comparing the file's size and last modified time to that of the version of the file in the local document cache and then in the remote document cache.

3. If the file is not found either in the local cache or the remote cache, the `DistributorSender` sends the file to the remote server. On the remote server, a `DistributorReader` writes the file in the specified directory.
4. The file is cached locally.
5. The `putFile` operation returns a URL that can be used to access the sent file.

Distributor Commands

When a `DistributorServer` receives a connection from a `DistributorSender`, the `DistributorServer` creates a `DistributorReader`, which actually handles the command from the `DistributorSender`. A `DistributorSender` can send two types of commands: `fileCheck` and `put`.

fileCheck Command

The `fileCheck` command sends to the `DistributorReader` the following information:

- Directory name
- File name
- File size
- Last modified time

The `DistributorReader` checks the file in its local document root that corresponds to the directory name and file name. It tests whether the file size and last modified time of this file match those sent in the `fileCheck` command by the `DistributorSender`. The `DistributorReader` sends back a success or failure code, based on whether the files match or not.

put Command

The `put` command sends to the `DistributorReader` the following information:

- Directory name
- File name
- File size
- Last modified time
- Data (a content repository item, file, or byte array)

The `DistributorReader` writes the data in the file in its local document root that corresponds to the directory name and file name in the `put` command.

Using Content Distribution with an SQL Content Repository

The ATG content distributor system can be used with an SQL Content Repository that stores both its content and its metadata in an SQL database. Note, however, that the content distributor system cannot be used with a repository that stores any information on a file system, such as an SQL/File System Connector.

The `putFile` operation of a `DistributorSender` returns a URL that can be used to access content repository items. AN SQL Content Repository can define user-defined properties of type `atg.distributor.DistributorPropertyDescriptor`. You can then use this property to get the URL of a repository item and use the URL to fetch the content of the item.

So, for example, you might have a repository item type named `product`. You can fetch and display its URL in a link like this:

```
<a href="param:product.template.url">link text here</a>
```

Here is an example of how you might set up a property of type `DistributorPropertyDescriptor`. You might define a property like this in an SQL Content Repository's repository Definition file:

```
<property name="url" data-type="string"
      property-type="atg.distributor.DistributorPropertyDescriptor"
      queryable="false">
  <attribute name="poolPath"
             value="/atg/commerce/catalog/ContentDistributorPool"/>
</property>
```

This tag defines a property named `url`. Its type is defined by the `property-type` attribute. The `<attribute>` tag gives the Nucleus address of a `DistributorPool` component to use in fetching the content item. If you want to use a single `DistributorSender` instance, rather than a `DistributorPool`, you use an `<attribute>` tag like this:

```
<attribute name="senderPath" value="/nucleus/path/to/DistributorSender"/>
```

Setting Up a Content Distributor System

The ATG content distributor system includes the following types of components:

- `DistributorSender`
- `DistributorPool`
- `RemoteServerCache`
- `DistributorServer`
- `DistributorReader`

To set up a content distributor system, you need to install and configure a number of these components on your ATG servers and your HTTP servers:

1. Create and configure one `DistributorSender` or `DistributorPool` component on each of your ATG servers that handles user sessions. See DistributorSender and DistributorPool.
2. If you choose to use local caching, create and configure one `RemoteServerCache` component on each of your ATG servers that has a `DistributorSender` or `DistributorPool` component. See RemoteServerCache.
3. Install, configure, and start up a `DistributorServer` component on each HTTP server machine. See DistributorServer.

Note: You do not need to create or configure any `DistributorReader` components. When a `DistributorServer` receives a connection from a `DistributorSender`, the `DistributorServer` automatically creates a `DistributorReader`.

DistributorSender

The `DistributorSender` is an instance of `atg.distributor.DistributorSender`. It opens connections to each host in its `serverList` property. The `DistributorSender` has the following properties that you might want to configure:

Property Name	Description
`serverList`	A comma-separated list of hosts of `DistributorServer` components. Include an entry for each HTTP server in your site.
`cacheLocally`	When a file is sent, should it also be cached locally? Default is `true`
`documentRoot`	The local document root.
`documentRootCachePath`	The directory in the remote HTTP server's document root that should be used to store all the cached content. For example if the remote HTTP server's document root is at `/work/www/doc` and `documentRootCachePath` is set to `DIST_CONTENT` then the `DistributerReceiver` creates a directory named `/work/www/doc/DIST_CONTENT` to hold cached content sent by the `DistributorSender`.
`createCacheDirectories`	Create any directories that do not already exist in the remote cache. Default is `true`
`contentItemNameProperty`	The repository item property to use in generating a file name for a content item. See Generating a File Name for more details.
`remoteServerCache`	A local cache of items that the `DistributorSender` has sent to the remote servers. See RemoteServerCache.

Property Name	Description
`minReconnectInterval`	If the `DistributorSender` fails to connect, it tries again after this interval. Default is 120000 milliseconds, or 2 minutes.

Running the DistributorSender from a Command Line

You can also run the `DistributorSender` from a command line. This can be helpful in testing and development. The `DistributorSender` command uses this syntax:

```
javan ATG.distributor.DistributorSender -<arguments>
```

The `DistributorSender` command takes the following arguments:

Argument	Description
`-put` *`filepath`*, *`filepath`*	Send the named files, using the `put` command.
`-docroot` *`path`*	The local document root from which to get the files.
`-doccache` *`path`*	The local document cache.
`-hosts` *`host:port`*, *`host:port`*	The host names and port numbers of the `DistributorServer` components to connect to.
`-test`	Sends test files to verify that connections can be made. See Test Argument.

Test Argument

When you start the `DistributorSender` with the `-test` option, the sender should output something similar to this:

```
sending file: name="distributortest_0.tst";size="890"
sending file: name="distributortest_1.tst";size="890"
sending file: name="distributortest_2.tst";size="890"
sending file: name="distributortest_3.tst";size="890"
sending file: name="distributortest_4.tst";size="890"
Sending files took 139 msec
```

The `DistributorSender` sends five `distributortest_#.tst` files to the remote servers. Each file is 890 bytes long, and contains 100 lines of text consisting of the word Line and the line number, like this:

```
Line: 0
Line: 1
Line: 2
 ...
```

DistributorPool

If you have a single instance of `DistributorSender`, it can form a performance bottleneck. To avoid this problem, you can configure a resource pool of `DistributorSender` components. ATG provides a class for this purpose: `atg.distributor.SenderResourcePool`. It is a resource pool that pools TCP/IP connections to the `DistributorServer` components. Like other resource pools, you can configure the minimum and maximum size of a `DistributorPool`. (See Resource Pools in the Core ATG Services chapter.) The default maximum size is 10, but you might need to increase that, depending on the load served by your ATG application. An instance of a `DistributorPool` exists in ATG Commerce at `/atg/commerce/Catalog/ContentDistributorPool`.

The `DistributorPool` also has the following properties that you might want to configure:

Property Name	**Description**
`cacheLocally`	When a file is sent, should it also be cached locally? See RemoteServerCache. Default is `true`
`contentItemNameProperty`	The repository item property to use in generating a file name for a content item. See Generating a File Name for more details.
`createCacheDirectories`	Create any directories that do not already exist in the remote cache. Default is `true`
`documentRoot`	The local document root.
`documentRootCachePath`	If you want to store documents in a subdirectory of the document root. For example: `documentRootCachePath=doc/MEDIA`

Generating a File Name

The `DistributorSender` needs to generate a unique file name for each file it sends. The usual way to do this is to concatenate the repository ID with the value of a repository item property. The name of this repository item property is specified by the `contentItemNameProperty` property of the `DistributorSender`.

For example, you might have a content item type defined like this:

```
<item-descriptor name="articles">
     <table name="articles" type="primary" id-column-name="article_id">
     <property name="id" column-name="article_id"/>
     <property name="description"/>
     </table>
</item-descriptor>
```

You can set `contentItemNameProperty=id`, and the `DistributorSender` uses the `id` property in creating a unique file name for a repository item of type `articles`.

RemoteServerCache

If you use local caching (by setting the `cacheLocally` property to `true`), the `DistributorSender` checks the file size and last modified time of items against the entries in a local cache of items that were sent to the remote servers. This cache component is called a `RemoteServerCache`.

To use this feature, create a `RemoteServerCache` component (of class `atg.distributor.RemoteServerCache`) for each `DistributorSender` or `DistributorPool` instance and set the `remoteServerCache` property of the `DistributorSender` or `DistributorPool` to point to it. The `RemoteServerCache` component might be configured like this:

```
$class=atg.distributor.RemoteServerCache
sizeLimit=10000
```

The `sizeLimit` property sets the maximum number of entries in the cache.

DistributorServer

A `DistributorServer` runs as a Nucleus component on each HTTP server machine. When a `DistributorServer` receives a connection from a `DistributorSender`, the `DistributorServer` creates a `DistributorReader`. The `DistributorReader` handles the processing of the `put` or `fileCheck` command from the `DistributorSender`.

Installing the DistributorServer

To install a `DistributorServer` on Windows:

1. Make sure the HTTP server machine has a Java Virtual Machine installed.
2. Download the ATG Web Server Extensions distribution file, `ATGWebServerExtensions10.0.1.exe`, from `www.atg.com`.
3. Run the ATG Web Server Extensions file.
4. The installer displays the Welcome dialog box. Click Next to continue.
5. Select the installation directory, and then click Next to continue. The default directory is `C:\ATG\ATGWeb10.0.1`.
6. The installer displays the list of web server extensions you can configure during the installation process. Make sure the `DistributorServer` is selected, and click Next to

continue. (If you want to install the Publishing web agent as well, see the *ATG Content Administration Programming Guide*.)

7. Specify the port that the `DistributorServer` should use to listen for connections from `DistributorSender` components, and click Next to continue. The default is 8810.

8. Specify the directory that the `DistributorServer` should use to cache files, and click Next to continue. The directory can be the HTTP server's document root, or any subdirectory within it. The default is the `home\doc` subdirectory of the installation directory you previously specified.

9. Enter a name for the Program Folder, and click Next to continue. The default is `<ATG 9dir>\ATG Web Server Extensions`.

10. The installer displays the settings you selected. If you need to make any changes, click Back. Otherwise, click Next to proceed with the installation.

To install a `DistributorServer` on Unix:

1. Make sure the HTTP server machine has a Java Virtual Machine installed.

2. Download the ATG Web Server Extensions distribution file, `ATGWebServerExtensions10.0.1.jar`, from `www.atg.com`.

3. Unpack the `ATGWebServerExtensions10.0.1.jar` file:

   ```
   jar xvf ATGWebServerExtensions10.0.1.jar
   ```

   The installer creates an `ATGWeb10.0.1` subdirectory in the current directory. This subdirectory includes the files and directories needed to install the web server extensions.

4. Change to the `ATGWeb10.0.1/home` directory and enter the following command (or its equivalent) to set read and write permissions for the `Install` script:

   ```
   chmod 755 bin/Install
   ```

5. Run the `Install` script:

   ```
   bin/Install
   ```

6. The installer displays the list of web server extensions to install. Type A to install the `DistributorServer`. (If you want to install the Publishing web agent as well, see the *ATG Content Administration Programming Guide*.)

7. Specify the port that the `DistributorServer` should use to listen for connections from `DistributorSender` components. The default is 8810.

8. Specify the directory that the `DistributorServer` should use to cache files. The directory can be the HTTP server's document root, or any subdirectory within it. The default is the `home/doc` subdirectory of the installation directory.

Configuring the DistributorServer

The `DistributorServer` component is an instance of `atg.server.distributor.DistributorServer`. You can configure it by editing the `DistributorServer.properties` file in the `home\localconfig\atg\dynamo\server` subdirectory of the ATG Web Extensions installation directory. The `DistributorServer` has the following properties:

Property Name	Description
`enabled`	If `true`, the `DistributorServer` service is enabled. Default is `true`.
`port`	The port where the `DistributorServer` should listen for connections from `DistributorSender` components. Default is 8810.
`cacheDirectory`	Directory on the HTTP server where the `DistributorServer` stores files. Default is the `home\doc` subdirectory of the ATG Web Extensions installation directory.
`allowedSenders`	A comma-separated list of `<host>:<port>` entries. If this property is set, the `DistributorServer` accepts connections only from these hosts. By default the property is not set, which means the `DistributorServer` accepts connections from any host running a `DistributorSender`.

Starting the DistributorServer

To start up Nucleus and run the `DistributorServer` component, use the following command:

```
startNucleus -m Distributor
```

If you also have a configured ATG Publishing web agent on the web server, you can start up the `Distributor` and `PublishingWebAgent` modules at the same time. In this case, use the following command:

```
startNucleus –m PublishingWebAgent:Distributor
```

For information about the ATG Publishing web agent, see the *ATG Content Administration Programming Guide*.

18 Internationalizing an ATG Web Site

Internationalizing a web site is the process of creating a site that is capable of displaying content in different languages. ATG internationalization is based on Java internationalization standards. You can design an ATG web site for a single locale or for multiple locales. In your translated web pages, you can vary the display of data such as dates and currencies according to locale-specific formatting rules.

Internationalizing an application and localizing it are different activities:

- Internationalization is the process of preparing a site for delivery in different languages.
- Localization is the process of translating its contents for a specific locale or locales.

For example, an internationalized site is one that has its text messages separated into easily accessible resource files rather than included in the source code; that same site can then be localized easily by translating the text messages into French, for example. You do not have to localize your ATG application to create a web site in another language.

In this chapter

This chapter includes the following topics:

- Overview: Briefly describes the basics of creating an internationalized web site.
- Setting Up a Multi-Locale ATG Web Site: Introduces the steps involved in preparing an ATG web site to serve content to different locales.
- Using ResourceBundles for Internationalization: Explains how to internationalize ResourceBundle files containing Strings that appear in your web site.
- Setting Character Encoding in JSPs: Describes how to specify a JSP's character encoding by setting the content type within the page.
- Using the EncodingTyper to Set the Character Encoding: Explains how to use the EncodingTyper component to determine the character encoding for posted data in forms.
- Configuring the Request Locale: Explains how to use the `RequestLocale` component to associate a character encoding with a request.
- Character Encoding and Locale Configuration Examples: Provides examples for setting the `EncodingTyper` and `RequestLocale` components for different server and locale configurations.
- Setting the Java Virtual Machine Locale: Describes how to set the ATG server locale by changing the locale of the Java Virtual Machine.

- Configuring the Database Encoding: Explains how to set the character encoding for the database server.
- Setting the Email Encoding: Describes how to determine the character encoding for targeted emails.
- Internationalizing Content Repositories: Describes how to configure content repositories to serve content for several different locales.
- Creating Locale-Specific Content Pages: Explains how to set up and work with content pages for a localized site. Includes information on translating JSP tags.
- Designing a Multi-Locale Entry Page: Describes how to create a top-level index page that acts as an entry point for a multi-locale site.
- Converting Properties Files to Escaped Unicode: Describes how to convert properties files containing non-Latin or non-Unicode fonts to Unicode so that they can be processed by tools such as the Java compiler.
- Localizing the Profile Repository Definition: Explains how to localize the entries in the profile repository definition file.
- Localizing Profile Group Names, Scenario Names, and Similar Items: Describes how to localize some of the items that appear in the ATG Control Center interface.
- Changing Date and Currency Formats: Introduces some options for displaying dates and currency information in localized web pages.
- Using Third-Party Software on an Internationalized Site: Briefly describes the third-party software requirements for an internationalized ATG web site.

Before reading this chapter, you should be familiar with Java internationalization standards. For more information, refer to the JavaSoft Internationalization Specification at http://java.sun.com/j2se/1.3/docs/guide/intl/index.html.

Overview

This section provides an overview of some basic internationalization concepts, as well as the ways ATG implements these concepts. It contains the following topics:

- ResourceBundles
- Character Encodings
- EncodingTyper Component
- RequestLocale Component
- Java Internationalization Objects

ResourceBundles

Internationalizing a web site is easier when text messages are not stored directly in code. Java provides the `ResourceBundle` mechanism for the separation of messages from Java code. A `ResourceBundle` is a Dictionary of keys that map to specific text messages.

Most web sites have two types of text messages:

- User messages that are displayed to site visitors
- Developer messages that are visible to developers only—for example, error logs.

To internationalize your web site, create separate `ResourceBundles` for user and developer messages. Often, localization teams do not translate developer messages, so it is helpful for people who are localizing your site if you keep the two types of message separate.

The ATG Control Center also uses `ResourceBundles` to store text for user interface display.

For more information, refer to Using ResourceBundles for Internationalization.

Locales

A locale represents a geographic or cultural region and is used to distinguish between the language variants used by different groups of people. For example, English has several language variants such as British English and American English; each of these is a locale. Locales are usually represented by `language` and `country` parameters. For example, en_`GB` represents British English, en_`US` represents American English, and `fr_FR` represents French used in France.

There are two types of ATG locales: the request locale and the server locale.

- ATG uses the request locale to generate locale-based user messages. For more information, refer to Configuring the Request Locale.
- ATG uses the server locale to generate developer messages. For information on how to change the server locale, refer to Setting the Java Virtual Machine Locale.

The profile repository can also include a `locale` property for each user; when this property is set, for example by means of a language preference specified in a registration form, it can be used with targeting rules or scenarios to display content that is appropriate for the user's locale.

Character Encodings

A character encoding is a technique for translating a sequence of bytes into a sequence of characters (text). For example, content from a web page is stored on the server as a sequence of bytes and, when it is sent to a web browser, it is converted to human-readable text using an appropriate character encoding. Different character encodings are available for handling the requirements of different languages; for example, languages such as English have a relatively small number of characters and can use a single-byte character set such as ISO-8859-1, which allows up to 256 symbols, including punctuation and accented characters. Other languages such as Chinese, however, use thousands of characters and require a double-byte character set such as Unicode, which allows up to 65536 symbols.

You can create internationalized web sites with ATG in any character encodings supported by the Java Development Kit (JDK). Java bases all character data on Unicode. All Strings in Java are considered to be Unicode characters. Likewise, I/O classes support the conversion of character data to and from native encodings and Unicode. You can find a list of character encodings that ATG has tested and supports on www.atg.com.

Developers and web designers generally use a native encoding method for their content. ATG handles native encoded content the same way Java does. When ATG reads in character data, it is converted to Unicode by the `GenericConverter` that is included with ATG. The `GenericConverter` handles any character encodings supported by Java and by your version of the JDK. Whenever data is written out and sent to a web browser, the `GenericConverter` converts the data back to a native encoding. Typically, the encoding written out to a browser is the same as the encoding of the document that is read in by ATG. The Java `InputStreamReader` and `OutputStreamWriter` classes are used to convert text from locale-specific encoding to Unicode and then convert the text back to the locale-specific encoding for display to the user. For more information, see Using the EncodingTyper to Set the Character Encoding in this chapter.

EncodingTyper Component

To properly parse a document, the server must know the character encoding of the document before reading it. Character encoding is determined by specific tags that you add to the JSPs. For more information, refer to Setting Character Encoding in JSPs.

Note: The `EncodingTyper` component is used for determining the encoding of posted form data. See Converting Posted Data with the EncodingTyper.

RequestLocale Component

An internationalized ATG web site can serve content that is in a different language from the one in which the server is running. For example, a server that is configured to use Japanese can serve content in Korean. An ATG server serves content in various languages by identifying the language or locale associated with the request and delivering content that is appropriate.

The `RequestLocale` component is a session-scoped component that attaches locale information to the requests of the session. You can configure the `DynamoHandler` servlet to add a `RequestLocale` to the request object. When a `RequestLocale` component is first created, it runs through a hierarchy of sources to determine which locale to use for the session. When the ATG server finds a source providing the necessary information, a `Locale` object is created and stored in the `RequestLocale` for use by all requests within the user's session. For more information, see the Configuring the Request Locale section in this chapter.

When designing your site, keep in mind that ATG does not automatically ensure that the `RequestLocale` of the current visitor matches the language of the content in the pages that the visitor requests. In order to ensure that a visitor with a particular locale sees content suited for that locale, you must design your site with the appropriate directory and navigational structure. For more information, see Creating Locale-Specific Content Pages and Designing a Multi-locale Entry Page in this chapter. If you want to enforce a correspondence between the `RequestLocale` and document language, you can build this logic into the servlet pipeline.

Java Internationalization Objects

In an internationalized ATG application, specific Java internationalization classes must be used for the following purposes:

- formatting of numbers, percentages, currencies, dates, and times
- formatting of compound messages and plurals
- character checking
- String comparison
- character, word, and sentence text boundaries

In order for these objects to be locale-sensitive, ATG requires these objects to call either a `RequestLocale` object or a `DynamoHttpServletRequest` object.

For information on the Java internationalization classes, refer to the JavaSoft Internationalization Specification at http://java.sun.com/j2se/1.3/docs/guide/intl/index.html. For information on the `RequestLocale` and `DynamoHttpServletRequest` objects, see Configuring the Request Locale in this chapter.

Setting Up a Multi-Locale ATG Web Site

To set up a multi-locale ATG web site, complete the steps outlined below. Each one is described in more detail in the section shown. There might be additional steps to perform depending on your requirements. For example, if you send targeted emails to web site users in different locales, you might have to configure email encoding. For information on these additional steps, refer to the rest of this chapter.

1. Set up your document directory structure with parallel directories for each locale. For more information on this step, see Content Page Directories.
2. For each locale, copy, rename, and translate the `ResourceBundle.properties` files. See Using ResourceBundles for Internationalization.
3. For each locale, copy, rename, and translate the pages that contain content for the web site. Refer to Creating Locale-Specific Content Pages for more information.
4. Design an entry page for the site. See Designing a Multi-locale Entry Page for more information.
5. Configure the character encoding for the site’s content pages. See Using the EncodingTyper to Set the Character Encoding or Setting Character Encoding in JSPs.
6. Configure the request locale. For more information, see Configuring the Request Locale.
7. Change the Java Virtual Machine locale if necessary. See Setting the Java Virtual Machine Locale.
8. Set the encoding for the database server. See Configuring the Database Encoding.
9. Set up your repositories to store multi-locale content. See Internationalizing Content Repositories in this chapter for more information.

Using ResourceBundles for Internationalization

When designing an internationalized ATG web site, you can use the Java `ResourceBundle` class of the `java.util.*` package and store messages as `.properties` files in `ResourceBundles`. Each message `ResourceBundle` is a Dictionary that maps an identifying key to a text message. `ResourceBundle` objects can be used to store the following types of information: globally scoped user messages, server-side exception messages, log messages, and session/request-scoped user messages.

Note: If you store a session-scoped or request-scoped message in a `ResourceBundle`, you should be careful not to store this `ResourceBundle` in static member variables.

You can also store session-scoped or request-scoped user messages in JSPs rather than in `ResourceBundles`. This behavior is useful because content pages are easily accessible to web designers. For information on storing user messages in content pages, see Creating Locale-Specific Content Pages in this chapter.

This section provides an overview of `ResourceBundle` objects in the following topics:

- Introduction to ResourceBundles
- ResourceBundle Objects
- ResourceBundle Inheritance
- Internationalizing ResourceBundles

Introduction to ResourceBundles

There are three types of ATG `ResourceBundles`:

- The `ResourceBundles` that affect the web site visitor's content, such as user messages
- `ResourceBundles` for the ATG Control Center user interface
- Server-side `ResourceBundles` that store developer messages and logs

The only `ResourceBundles` that you need to translate to create an internationalized web site are the `ResourceBundles` that affect the web site visitor's content.

The visitor's request locale determines the locale of the `ResourceBundles` that store user content for an internationalized web site. For example, if the user visits an internationalized ATG web site, and the user's visitor locale is Japanese, the `ResourceBundle_ja.properties` file is used to generate user messages in Japanese.

In order to internationalize an ATG web site, all visitor locale `ResourceBundle.properties` files must be copied and renamed for each locale, according to Java naming guidelines. These naming conventions are necessary in order for the Java locale inheritance system to work correctly. The renamed `.properties` files must then be translated according to the ATG translation guidelines.

For information on `ResourceBundle` inheritance, see the ResourceBundle Inheritance topic. For information on `ResourceBundle` naming, see the Internationalizing ResourceBundles section. For detailed information on ResourceBundles, see the Java documentation for java.util.ResourceBundle.

By default, ATG does not log the names or prefixes of resource bundles if the server's locale is set to en by the JVM (see Setting the Java Virtual Machine Locale). To change this behavior, set the logResourceNames property in the /localconfig/Nucleus.properties file to true.

ResourceBundle Objects

To fetch a ResourceBundle object, use the Java ResourceBundle.getBundle method. This method instantiates a ResourceBundle object for the specified ResourceBundle basename and locale. After using the getBundle method to instantiate the object, use the getObject or getString method to retrieve the value from the specified property in the given ResourceBundle object.

The ResourceBundle that is instantiated depends on the ResourceBundle.properties files that exist for the given basename. If a ResourceBundle.properties file does not exist for the specified basename and locale arguments, the locale of the returned ResourceBundle depends on the ResourceBundle inheritance rules. See the ResourceBundle Inheritance topic in this chapter for more information.

Example

For example, you can create a ResourceBundle with a base name of DynamoBundle and the current request locale as follows:

```
resourceA = ResourceBundle.getBundle ("atg.dynamo.DynamoBundle",
  currentLocale);
```

If the current request locale is de_DE, a DynamoBundle object is created from the DynamoBundle_de_DE.properties file. If this file does not exist, the next best .properties file is used, according to the ResourceBundle inheritance rules.

To retrieve the price label string from the DynamoBundle ResourceBundle, you specify the appropriate key from the DynamoBundle when invoking the getString method:

```
String PriceLabel = resourceA.getString ("PriceKey");
```

This method retrieves the PriceKey value from the instantiated DynamoBundle object.

ResourceBundle Inheritance

ATG uses the Java ResourceBundle inheritance rules. According to Java specifications, all internationalized ResourceBundles belong to a family of ResourceBundle subclasses that share the same basename. The following steps are followed to determine which ResourceBundle.properties file to instantiate.

1. The getBundle method first looks for a class name that matches the request locale's basename, language, country, and variant. For example, if the desired class is DynamoBundle_fr_FR_Unix, it first looks for this class.

If no class is found with the specified `language`, `country`, and `variant` arguments, it proceeds to step 2.

2. The `getBundle` method looks for a class that matches the request locale's `basename`, `language`, and `country`, such as `DynamoBundle_fr_FR`.

   If no class is found with the specified `language` and `country` arguments, it proceeds to step 3.

3. The `getBundle` method looks for a class with a name that matches the request locale's `basename` and `language`, such as `DynamoBundle_fr`.

   If no class is found with the specified `language` argument, it proceeds to step 4.

4. The `getBundle` method then goes through steps 1-3 for the default locale, instead of the request locale. For example, if the default locale is `en_US_UNIX`, it looks for a class name in the following order:

   ```
   DynamoBundle_en_US_UNIX
   DynamoBundle_en_US
   DynamoBundle_en
   ```

   If no class is found for the default locale, it proceeds to step 5.

5. The `getBundle` method looks for a class with a name of the following format: `basename`, such as `DynamoBundle`. This class is used as a default `ResourceBundle` that can be used by any locale that is not supported by the web site.

   If no class is found for the specified `basename`, a Java `MissingResourceException` is thrown.

Internationalizing ResourceBundles

In order to internationalize a web site's `ResourceBundle` logs and user messages, you must copy, rename, and translate the `ResourceBundle.properties` files. These files must be renamed according to the Java naming conventions to enable `ResourceBundle` inheritance.

For translation instructions, see ResourceBundle Translation Instructions. For information on `ResourceBundle` inheritance see ResourceBundle Inheritance.

Preparing ResourceBundles for Internationalization

Each `ResourceBundle.properties` file should contain comments, which are formed by lines with either a # or `!` as the first character. Each property's comments should describe whether the property affects user messages, developer messages, or log messages. These comments should also mark certain properties that should not be translated.

An internationalized ATG web site uses the Java `ResourceBundle` class to dynamically display the appropriate `ResourceBundle` object. The Java `ResourceBundle` naming guidelines stipulate that each `ResourceBundle.properties` file must designate a locale in the file name, which must be in the following format: `basename_language_country_variant`. The `language` suffix is required, and the `variant` and `country` suffixes are optional.

In order to fully internationalize a web site, each user message `ResourceBundle.properties` file must be copied and renamed with the appropriate `language`, `country`, and `variant` suffixes. For example you can copy the `ResourceBundle` `DynamoBundle.properties` and rename it `DynamoBundle_fr_FR.properties` for the `fr_FR` locale.

The following displays an example of `ResourceBundles` with the basename `DynamoBundle` for five locales:

```
DynamoBundle_en_US_UNIX.properties
DynamoBundle_en_US.properties
DynamoBundle_fr_FR.properties
DynamoBundle_de_DE.properties
DynamoBundle_ja.properties
```

Each user message `ResourceBundle.properties` file should be translated according to the guidelines in the ResourceBundle Translation Instructions section in this chapter. In addition, any references to other `.properties` files, JSP files, HTML files, GIF files, and other media files should be changed to reflect the names of new files.

Note the following:

- You must perform an ASCII conversion to Escaped Unicode for all translated `ResourceBundle` files that contain non-Latin 1 fonts or non-Unicode characters (including single-byte character sets). For more information, see Converting Properties Files to Escaped Unicode.
- Make sure that the internationalized `ResourceBundles` are referenced in the ATG CLASSPATH.

ResourceBundle Translation Instructions

The following topics describe the various types of text messages, escape sequences, and media elements in `ResourceBundle.properties` files that might have to be translated:

- Translating ResourceBundles
- Compound Messages
- Escape Sequences
- Line Continuation
- Media Elements

Translating ResourceBundles

ATG uses `ResourceBundles` to generate user messages that display in an ATG web site. Each property in a `ResourceBundle.properties` file is a key-value pair, in the format `key=value`. Make sure that only the values are translated. The keys should remain in English.

Both the keys and values of `ResourceBundle` properties that are commented as properties that should not be converted should remain in English. Any ATG code, such as `DROPLET BEAN=\"/atg/dynamo/droplet/Switch\"`, should not be translated.

Compound Messages

Compound messages are formatted with locale-sensitive formatting objects. A compound message is a user message that contains one or more variables, including dates, times, currencies, and `Strings`.

The message pattern property contains the message variables and the static text portion of the message, in the format xxx {0} xxx. Each variable, such as {0}, represents a dynamic value. A text string might contain multiple variables, such as {0} and {1}. Only the text, not the variables, should be translated. The translator can move the variables as necessary.

Plurals

Plurals are formatted with standard Java internationalization formatting objects. A plural is a plural noun variable, such as `errors` or `files`. In Java, plurals are formatted as a special type of variable within a compound message. The plural compound message is stored in a `ResourceBundle`. The message pattern property contains choices for the message, which vary based on the number of the noun variable, either 0, 1, or greater than 1. The static text portion of the message choices must be translated. The translator can move the variables as necessary.

Possessive Strings

Possessive strings represent the phrases that relate to possession and should be translated accordingly. An example of a possessive string is `Susan's Coat`. The two parameters in this example are `Susan` and `Coat`. The 's is derived from the message format of `{0}''s {1}.` This phrase structure should be translated in accordance with the language of internationalization.

Escape Sequences

If the value contains an `ASCII` escape sequence like one of the following, it is converted to a single character:

```
\t
 \n
 \r
 \\
 \"
 \'
\ (space)
\uxxxx
```

Line Continuation

If the last character on the line is a backslash character, \, the next line is treated as a continuation of the current line. When translating continuation values, each line but the last should end in a space and a `\` character.

Media Elements

The internationalization process can include translation of a subset or all media elements. If changes are made to media files, the content pages must be edited to reflect the names of the internationalized media files.

Setting Character Encoding in JSPs

You use the contentType page directive to specify the character encoding for a JSP. For example, you might place the following line at the top of a page:

```
<% page contentType="text/html; charset=ISO-8859-9"%>
```

You specify the content type before you retrieve the Java PrintWriter or JspWriter. The charset tag is parsed to select the content type the first time the PrintWriter or JspWriter on the response is created. As data goes through the PrintWriter or JspWriter, it is converted from Unicode to the encoding specified in the contentType directive. Make sure you use the IANA name for the encoding; this is the standard required by the HTTP specification. (See www.iana.org for a list of encoding names.)

With this technique, there is one encoding for each response, so the encoding applies to the entire page.

Alternatively, you can set the content type through a method call as follows:

```
<% response.setContentType( "text/html; charset=utf-8" ); %>
```

All code that comes after the method call uses the specified charset. Any code that comes before the method call uses the previous charset setting.

Embedded pages inherit the charset from their parent page.

Converting Posted Data with a Hidden Dyncharset Tag

If you set a page's character encoding using the JSP contentType page directive, the data submitted through a form on the page might not have the same encoding. For example, it might be determined by the user's specified locale. Typically, you use the EncodingTyper component to specify the converter for data that a user enters through a form, as described in Converting Posted Data with the EncodingTyper. However, assuming you use the ATG request wrapper to retrieve your form data, you can specify the encoding for the form data in the page itself by including a hidden "_dyncharset" input tag. For example:

```
<input type="hidden" name="_dyncharset"
    value="<%=response.getCharacterEncoding() %>">
```

When the form is submitted, the specified charset is used to convert the data back into Unicode.

Note: If you use dsp:input tags, you do not need to specify the "_dyncharset" tag; the command is generated automatically. You need to specify the tag only if you use non-DSP <input> tags.

Using the EncodingTyper to Set the Character Encoding

The EncodingTyper component provides the converter that handles the conversion of posted data—for example, from a registration form.

This section describes the `EncodingTyper` in the following topics:

- Introduction to the EncodingTyper
- DefaultEncoding Property
- EncodingMappings Property
- PathPatternPrefixes Property
- Converting Posted Data with the EncodingTyper
- Customizing the EncodingTyper

For information on specifying character encoding for JSPs, see Setting Character Encoding in JSPs.

Introduction to the EncodingTyper

The `EncodingTyper`, a Nucleus component located at `/atg/dynamo/servlet/pagecompile/EncodingTyper`, defines the encoding that an ATG server uses to deliver locale-specific content to the user. If the site is configured to serve content to one locale, the `EncodingTyper` simply uses the `defaultEncoding`. If the site is configured to deliver content to multiple locales, the `EncodingTyper` recognizes the document's encoding through a portion of the content's directory path. It checks the content path for a set of patterns that denote the content's encoding. After an ATG server has determined the encoding, the page compiler compiles the page. The compiled page is stored in a directory that includes the encoding as a suffix in the directory file name.

In a multi-locale site, an ATG server can determine which encoding to use only if the documents are separated in the document root by directories that identify the language (or encoding) of the documents. The directory names must follow a definable pattern. See Content Page Directories for more information. An ATG server recognizes the pattern of content directories and which encodings the directories map to based on the configuration of the `encodingMappings` and `pathPatternPrefixes` properties in the `EncodingTyper`. If none of the site's locale-specific encodings is found for the file, the encoding defined by the `defaultEncoding` property is used.

The `EncodingTyper` also provides character data converters for a given encoding. When a browser posts form data, there is no identifying field from the browser that indicates which encoding the data is in. The `EncodingTyper` determines the encoding of the data that is posted and provides a Converter to convert the data from its native encoding to Unicode for use by an ATG application.

In addition to configuring the `EncodingTyper`, you might have to configure the `DynamoHandler` and `RequestLocale` components. For more information see the Configuring the Request Locale section in this chapter.

Note 1: The encoding that is defined by the `EncodingTyper` determines the charset that is specified in the HTTP header.

Note 2: The `EncodingTyper` discerns the document's encoding exclusively through the directory path. The `EncodingTyper` ignores all HTML meta-tag encoding information.

DefaultEncoding Property

You must set the defaultEncoding property for any internationalized web site that is serving non-ASCII content. If the site is a single-locale site that is using only the ASCII encoding, it is best to leave the defaultEncoding property set to null. This setting allows for faster processing because conversion to UTF-8 is not performed.

If you are setting up an internationalized web site where one non-ASCII encoded content is served, which might or might not be different from the encoding of the server locale, the EncodingTyper simply requires configuration of the defaultEncoding property. Set this property to the encoding that you want your site to display by default. In this case, you do not have to define values for the encodingMappings and pathPatternPrefixes properties (see below).

If you are designing a site with multiple supported request locales, the defaultEncoding, encodingMappings, and pathPatternPrefixes properties must all be configured.

EncodingMappings Property

In a site that is serving content to multiple locales, the property encodingMappings maps the locale-based Page directory names to the actual page encoding of documents in those directories. This mapping uses a set of identifiers found in the relative document path.

The mappings are defined as a list of colon-delimited strings that use the following pattern:

```
Java Encoding:Identifier1:Identifier2:Identifier3:Identifier4
```

You must use the IANA (Internet Assigned Numbers Authority) names to specify the encodingMappings property, instead of the canonical Java encoding names. If you do not use the IANA encoding names the encodingMappings incorrectly sets the character set in the header passed to the web server. You can find a complete list of the IANA names at http://www.iana.org/assignments/character-sets.

Example A

In this example, each directory path uses one identifier, en, fr, or ja. In each case, the identifier is used to identify the associated page encoding.

```
<docroot>/en/...
<docroot>/fr/...
<docroot>/ja/...
```

The following displays the configuration of the encodingMappings property that corresponds with the directory structure in Example A:

```
encodingMappings=\
        US-ASCII:en:::,\
        ISO-8859-1:fr:::,\
        Shift_JIS:ja:::
```

Example B

The keywords used in the directory path do not have to be full directory names, as long as they follow a pattern. In the following example, the same identifiers that were used in Example A are referenced with different directory names:

```
<docroot>/en-content/...
<docroot>/fr-content/...
<docroot>/ja-content/...
```

The following displays the configuration of the `encodingMappings` property that corresponds with the directory structure in Example B.

```
encodingMappings=\
        US-ASCII:en:::,\
        ISO-8859-1:fr:::,\
        Shift_JIS:ja:::
```

In Examples A and B, the values for the `encodingMappings` property are the same. In both cases, only the first identifier is used to identify the encoding.

Example C

The `encodingMappings` property allows for up to four identifiers to indicate the page encoding of a directory path. Content directories can be as simple as those used in Example A, or more complicated as in the following example:

```
<docroot>/en_US/...
<docroot>/en_GB/...
<docroot>/en_CA/...
<docroot>/fr_FR/...
<docroot>/fr_CA/...
<docroot>/ja_JP/EUC/...
<docroot>/ja_JP/Shift_JIS/...
<docroot>/ja_JP/JIS/...
```

The following displays the configuration of the `encodingMappings` property that corresponds with Example C:

```
encodingMappings=\
        ASCII:en:US::,\
        ASCII:en:GB::,\
        ASCII:en:CA::,\
        ISO-8859-1:fr:FR::,\
        ISO-8859-1:fr:CA::,\
        EUC_JP:ja:JP:EUC:,\
        Shift_JIS:ja:JP:Shift_JIS:,\
        JIS:ja:JP:JIS:
```

PathPatternPrefixes Property

The pathPatternPrefixes property specifies the directory patterns in which the identifiers are used. The pathPatternPrefixes property is a list of patterns, each representing a prefix that is looked for at the beginning of a URL document path. The pattern strings are in Java MessageFormat pattern string format. The pattern string argument fields map directly to the mapping identifiers used in the encodingMappings property as follows:

```
{0} = Identifier1
{1} = Identifier2
{2} = Identifier3
{3} = Identifier4
```

The following displays three possible configurations of the pathPatternPrefixes property. These examples correspond with Examples A, B, and C in the EncodingMappings Property topic:

```
A
pathPatternPrefixes=\
        /{0}/

B
pathPatternPrefixes=\
        /{0}-content/

C
pathPatternPrefixes=\
        /{0}_{1}/{2}/,\
        /{0}_{1}/
```

The third example uses two patterns. This is because there are two sets of patterns in the respective mapping. One set of patterns uses two identifiers and one set of patterns uses three identifiers. In order for a pattern to match a relative document path, all identifiers in the mapping must be found in the pattern. In Example C, the relative document path /ja_JP/... does not map to the Shift-JIS encoding (SJIS) because the path contains only two of the identifiers, and this example requires three identifiers in order to specify the correct Japanese encoding for the user's platform.

Converting Posted Data with the EncodingTyper

When a browser posts form data, there is no identifying field from the browser that indicates which encoding the data is in. The EncodingTyper determines the encoding of the data that is posted and provides a converter to convert the data from its native encoding to Unicode for use by an ATG application. Three EncodingTyper properties relate to the conversion of character data from a native encoding to Unicode: encodings, converters, and nullEncodings.

The encodings and converters properties map converter classes to specific encodings. ATG comes with two default converters, GenericConverter and JapaneseConverter. The GenericConverter simply converts the supplied native-encoded character data into Unicode. The JapaneseConverter is described in the next section.

JapaneseConverter

Some languages, for example Japanese, can use more than one encoding. In this case you need a language-specific converter that detects which encoding the characters are using by inspecting a sample of the characters. ATG includes a `JapaneseConverter` that you can use out-of-the-box. The `JapaneseConverter` gathers a sample of the incoming characters and determines whether their encoding is SJIS or EUC. In some situations, however, the converter lacks access to enough characters to be able to detect the encoding. For this reason, when you are using a converter such as the `JapaneseConverter`, make sure to change the `fallbackEncoding` attribute of the `JapaneseConverter` component from its default setting of NULL to a specific encoding that you want the converter to use as the default. If you do not change this attribute, the converter uses ASCII as the default encoding and your web browser cannot correctly display the page.

The `nullEncodings` property of the `EncodingTyper` component defines a list of encodings that do not require conversion. For example, ASCII maps directly into Unicode character for character. In other words, no conversion is necessary to make an ASCII string into a Unicode string. ASCII is therefore listed as a null encoding.

Customizing the EncodingTyper

You can customize the means by which an ATG server determines the encoding type of a specific document by sub-classing the `EncodingTyper` and overriding the `getEncodingType()` method. In the following example, this method is passed document paths relative to the document root.

```
atg.servlet.pagecompile.PageEncodingTyper.java:
-----------------------------------------------
/**
 * Get the encoding to use for the specified path.
 *
 * @return the encoding string, or null if there is no encoding
 * corresponding to the specified path
 */
public String getEncodingType (String pPath)
{
  // Your code here.
}
```

Configuring the Request Locale

If an ATG server is serving content to a locale other than the server locale, the request must have a locale associated with the content being served. The service `OriginatingRequest.requestLocale` is an on-the-fly, session-scoped component that provides locale-related information for the duration of a session. When an `OriginatingRequest.requestLocale` component is first created, it looks through a hierarchy of sources to determine which locale to use for the session. After a Locale object is created, it is stored in the `OriginatingRequest.requestLocale` component for use by all requests within a visitor session. Because this is an on-the-fly component, you cannot access it within the ATG Control Center. It only exists for the duration of a web site user's session. You can configure the `/atg/dynamo/servlet/RequestLocale` component, however, as the `OriginatingRequest.requestLocale` component searches this component for information about a user's locale.

When a request reaches the servlet pipeline, the `DynamoHandler` pipeline servlet adds a `RequestLocale` component in the `requestLocale` property of the `DynamoHttpServletRequest` object. This behavior is controlled by the `DynamoHandler`'s `generateRequestLocales` property; the `RequestLocale` is added to the request object only if `generateRequestLocales=true`. In an ATG servlet bean, the `DynamoHttpServletRequest` object is used to access the `RequestLocale` object. If a Nucleus component does not have access to the `DynamoHttpServletRequest` object, the request locale can be accessed by including `requestLocale` as a property in the component.

When configuring a site's supported request locales, you must configure components related to the `RequestLocale` component. For more information on configuring your request locales, see Using the EncodingTyper to Set the Character Encoding in this chapter.

This section describes the `OriginatingRequest.requestLocale` and `RequestLocale` services in the following topics:

- RequestLocale Hierarchy
- Configuring RequestLocale Properties
- Additional RequestLocale Properties
- RequestLocale in Personalization Module Web Sites
- Allowing Users to Choose a Locale
- Using RequestLocale in an ATG Servlet Bean
- Adding the RequestLocale Property
- HttpServletRequest Component

RequestLocale Hierarchy

When a `OriginatingRequest.requestLocale` component is created for a session, the component must determine which locale to use for the session. In order to determine which locale to use, a hierarchy of sources is checked for information on setting the locale. By default, this hierarchy is as follows:

1. The request's `ACCEPT-LANGUAGE HTTP` header field
2. The default locale set via the `RequestLocale.defaultRequestLocaleName` property
3. The default locale of the server JVM

This hierarchy is important only to ATG web sites that are configured to serve content to multiple visitor locales. Single language sites that are serving content for the same locale as the server locale do not follow this hierarchy. Sites that are delivering content to one locale that is different from the server locale set a locale for use by all sessions and requests.

Note: The `DynamoHandler.generateRequestLocales` property must be set to `true` to generate a `RequestLocale` for each session. By default, the property is set to `false`.

Customizing the Hierarchy

The hierarchy of locale sources can easily be expanded to include custom sources. You can sub-class `atg.servlet.RequestLocale` and override the method:

```
public Locale discernRequestLocale (DynamoHttpServletRequest pRequest,
                                     RequestLocale pReqLocal)
    {
    }
```

This method returns the locale to use in the `RequestLocale`. Add `super.discernRequestLocale()` at the end of your overridden method so that the default request locale is set if no other source provides a locale. Finally, change the class used by the `/atg/dynamo/servlet/RequestLocale` component to that of your new `RequestLocale` sub-class.

Configuring RequestLocale Properties

You must configure the following `RequestLocale` properties when you are configuring your site's supported request locales. See Using the Encoding Typer to Set the Character Encoding for information on configuring components related to the `RequestLocale` component. For information on additional `RequestLocale` properties that do not require configuration, see the Additional RequestLocale Properties section.

defaultRequestLocaleName

If a locale cannot be determined for the session from any other source, this is the locale that is used. This property must be configured.

overrideRequestLocaleName

This property should be configured only if the ATG server is serving visitor content for a single locale that is different from the server locale. In this case, all sessions can use the same request locale. Setting this property to the single locale of the site can improve performance because it avoids the processing time of discerning the request locale. All request locales are set to this value.

detectHTTPHeaderChanges

When this property is set to `false`, `HTTP` headers are not checked to see if the `ACCEPT-LANGUAGE` and `ACCEPT_CHARSET` fields changed. A user does not commonly change these browser options in mid-session. Therefore, for performance reasons, the default value for this property is `false`.

validLocaleNames

The language in a request's `ACCEPT-LANGUAGE HTTP` header field might not be one of the site's supported request locales. Setting the `validLocaleNames` property to the list of locales associated with the site's content languages prevents a request locale from being created for a locale that is not supported by the site.

Additional RequestLocale Properties

In addition to the `RequestLocale` properties that require configuration, the `RequestLocale` component contains the following optional properties.

locale

The `java.util.Locale` objects used in locale-sensitive operations. These `Locale` objects are cached for reuse.

localeString

A String representation of the locale. Calling `toString()` on a `java.util.Locale` results in a new String allocation for each call. The `localeString` property uses the same String representation repeatedly for a given `Locale`.

previousLocale

Sometimes a locale is changed during a session. The `previousLocale` is the `Locale` object used in the previous request.

previousLocaleString

A String representation of the `previousLocale`. The `previousLocaleString` property uses the same String representation repeatedly for a given `Locale`.

acceptLanguage

The `ACCEPT-LANGUAGE HTTP` header field from the request that defined the `RequestLocale` component for the session.

acceptLanguageList

The `acceptLanguage` property parsed into an array of Strings.

acceptCharset

The `ACCEPT-CHARSET HTTP` header field from the request that defined the `RequestLocale` component for the session.

acceptCharsetList

The `acceptCharset` property parsed into an array of Strings.

Request Locale in Personalization Module Web Sites

This section describes the use of the `RequestLocale` and `OrginatingRequest.requestLocale` components that is specific to the Personalization module. It contains the following topics:

- Personalization RequestLocale Hierarchy
- Personalization RequestLocale Properties

Personalization RequestLocale Hierarchy

The Personalization module request locale hierarchy is set by default to the following. This hierarchy differs from the ATG request locale hierarchy in that the Personalization module hierarchy includes the profile attribute.

1. Profile `locale` property: The `locale` value in the profile can be set, for example, by having the user choose a locale from a list of choices on a Preferences page that is provided at the web site.
2. The `ACCEPT-LANGUAGE HTTP` header: This is set in the `/atg/dynamo/servlet/RequestLocale.validLocales` component.
3. Default request locale: This is defined in the `/atg/dynamo/servlet/RequestLocale.defaultRequestLocaleName` component.
4. The default locale specified in the server's JVM.

See also Allowing Users to Choose a Locale. For more information on setting up a site with language options, see the Quincy Funds Demo and the *ATG Quincy Funds Demo Documentation*.

Personalization RequestLocale Properties

The Personalization module sub-classes the ATG `RequestLocale` object to obtain a locale from the visitor's profile. The Personalization module adds the following properties to the `RequestLocale` component.

- `profilePath`: The Nucleus path to the Profile object in each session. This is required so that the `RequestLocale` component can access the visitor's profile.
- `profileAttributeName`: This attribute specifies the locale choice of the visitor. It is best to let this value default to null in the profile template. If this value defaults to null and the user has not made a language choice, no locale is assigned from the profile. The other locale determinants such as the request's `ACCEPT-LANGUAGE HTTP` header are used for the session until the user makes a language choice and thereby sets the profile attribute value.

Allowing Users to Choose a Locale

When designing your web site, you might want to add options to your web pages that allow users to change the initial request locale assigned by the ATG server to one of the other request locales that the site supports. For example, you can set up a Preferences page where users can define their language choice. This language preference is carried across site visits. To do this, design a page with a profile form that presents a list of supported locales, and use the form to update the `locale` property in the profile or the `/atg/dynamo/servlet/RequestLocale.localeString` property. Make sure that `/atg/dynamo/servlet/pipeline/DynamoHandler.generateRequestLocales` is set to true.

The locale that the user specifies does not take effect automatically. For information on the code you must include to update the locale, see also Updating the RequestLocale below.

Using RequestLocale in an ATG Servlet Bean

The `RequestLocale` component is stored for easy access in the `requestLocale` property of the `DynamoHttpServletRequest` object that is passed in each request. This section describes how to reference the `RequestLocale`, and it contains the following topics:

- Getting the RequestLocale
- Updating the RequestLocale

Getting the RequestLocale

The following is an example of getting and using the `RequestLocale` inside an ATG servlet bean or other servlet:

```
public void service (DynamoHttpServletRequest pRequest,
                     DynamoHttpServletResponse pResponse)
        throws IOException, ServletException
{
  RequestLocale reqLocale = null;
  Locale        locale    = null;

  // Make sure the RequestLocale isn't null...
  if ((reqLocale = pRequest.getRequestLocale()) != null) {
   // ...then get the Locale
   locale = reqLocale.getLocale();
  }
  else {
   // Otherwise, just use the JVM's default Locale
   locale = Locale.getDefault();
  }

  // Then you can use the locale to set up formatters; for instance...
  NumberFormat format = NumberFormat.getCurrencyInstance(locale);
  // ...etc....
}
```

Updating the RequestLocale

For performance reasons, the `RequestLocale` is created the first time it is needed in a session. The locale to use for a session is decided at the time of the `RequestLocale` creation. The `Locale` object stored in the `RequestLocale` component is not updated from request to request; it is updated only when the `handleRefresh` method is called to refresh the component. For example, the `handleRefresh` method is required when a visitor has changed the language of his or her session (via one of the sources in the `RequestLocale` hierarchy).

To make the `RequestLocale` component re-evaluate the `Locale` object being used, call the `RequestLocale.handleRefresh()` method. This method can be called from a JSP as follows:

```
<dsp:setvalue bean="/atg/dynamo/servlet/RequestLocale.refresh" value=" "/>
```

Note: Changes to a browser's ACCEPT-LANGUAGE value are not detected unless the property RequestLocale.detectHTTPHeaderChanges is set to true.

After a session has ended, any information that the RequestLocale component provided for the session is stored in the locale property of the user's profile.

If you want to update the RequestLocale immediately after a user has changed the locale on a profile form, you can design the updating form to set the ProfileFormHandler.updateSuccessURL to a success page as follows. For example:

```
<dsp:form action="<%=ServletUtil.getRequestURI(request)%>" method="POST">
<dsp:input bean="ProfileFormHandler.updateSuccessURL" type="HIDDEN"
  value="../index.jsp"/>
```

The success page must then update the request URL by calling the RequestLocale.handleRefresh() method as shown above.

Adding the RequestLocale Property

If a Nucleus component does not have access to the DynamoHttpServletRequest object, another means of accessing the request locale is by including requestLocale as a property in the component. The requestLocale object is session-scoped, so requestLocale cannot be used as a property in a globally scoped component.

HTTPServletRequest Component

The following property that is related to the RequestLocale is contained in the ATG HttpServletRequest component.

requestLocalePath

This property is the Nucleus path to the RequestLocale component. If the RequestLocale component is moved to a different location in the component hierarchy, this property must be changed to point to the new location.

Character Encoding and Locale Configuration Examples

This section summarizes how you set the character encoding and request locale, described in the previous sections, for different types of web site configuration.

- The server runs in the same locale as the content being served. See One Locale for Server and Content.
- The server runs in one locale, and content is served in a different language. See Server Locale and One Different Content Locale.

- The server runs in one locale, and content is served in several different locales. See Server Locale and Multiple Content Locales.

One Locale for Server and Content

If you design a site where the server locale and the web content locale are the same, the following configuration steps are necessary:

1. If necessary, set the server locale by changing the JVM locale. See Setting the Java Virtual Machine Locale for more information.
2. Configure the character encoding for site content. See Using the EncodingTyper to Set the Character Encoding or Setting Character Encoding in JSPs.
3. Set the `generateRequestLocales` property to `false` in the`/atg/dynamo/servlet/pipeline/DynamoHandler` component.
4. If necessary, configure the JDBC driver and database for the appropriate encoding. Generally, the encoding of the database should be the same as the encoding of the site content, including JSP files and repository content. See the Configuring the Database Encoding section in this chapter for more information.

Server Locale and One Different Content Locale

If you design a site with two different locales—one server locale and a different web content locale—the following configuration steps are necessary:

1. If necessary, set the server locale by changing the JVM locale. See Setting the Java Virtual Machine Locale for more information.
2. Configure the character encoding for site content. See Using the EncodingTyper to Set the Character Encoding or Setting Character Encoding in JSPs.
3. Set the `generateRequestLocales` property to `true` in `/atg/dynamo/servlet/pipeline/DynamoHandler` component.
4. In the `/atg/dynamo/servlet/RequestLocale` component, set the `overrideRequestLocale` property to the request locale.

   For more information, see Configuring the Request Locale.
5. If necessary, configure the JDBC driver and database for the appropriate encoding Generally, the encoding of the database should be the same as the encoding of the site content, including JSP files and repository content. See the Configuring the Database Encoding section in this chapter for more information.

Server Locale and Multiple Content Locales

If you design a site with a server locale and multiple visitor locales, the following configuration steps are necessary:

1. Separate the content pages into language-specific directories.

   See the Locale-specific Content Pages section in this chapter for more information.
2. Decide which repository design best fits the needs of your site. Edit targeting rules and repository meta-tags as necessary.

For more information, see the Internationalizing Content Repositories section in this chapter.

3. If necessary, set the server locale by changing the JVM locale. See Setting the Java Virtual Machine Locale for more information.

4. Configure the character encoding for site content. See Using the EncodingTyper to Set the Character Encoding or Setting Character Encoding in JSPs.

5. Set the `generateRequestLocales` property to `true` in the `/atg/dynamo/servlet/pipeline/DynamoHandler` component.

6. Set the `validLocaleNames` and `defaultRequestLocaleName` properties in the `/atg/dynamo/servlet/RequestLocale` component.

   For more information, see Configuring the Request Locale.

7. Design the entry point to your site so that it matches the user's request locale to the appropriate content directory. For more information, see the Designing a Multi-locale Entry Page section in this chapter.

8. If necessary, configure the JDBC driver and database for the appropriate encoding. In this situation, you are likely to want to configure your database to use Unicode. See the Configuring the Database Encoding section in this chapter for more information.

9. Add an attribute for the user's locale to the profile template.

Setting the Java Virtual Machine Locale

In order to run an internationalized ATG web site, you might have to set the Java Virtual Machine (JVM) locale. By default, the JVM locale is the locale of the platform where the JVM is installed. To override the default JVM locale, you must set the appropriate language and region (country) arguments in the server environment. You can do this by adding these arguments to the `environment.sh` file (UNIX) or `environment.bat` file (Windows).

For example, you can add the following line to an `environment.bat` file to change the JVM locale to French:

```
set JAVA_ARGS=-Duser.language=fr -Duser.region=FR %JAVA_ARGS%
```

Note: The JVM locale determines the ATG server locale, which is used for the `resourceBundles` that generate ATG server messages and log file messages.

Configuring the Database Encoding

You should set the character encoding of your JDBC driver and database with the encoding that is suitable for the locales that your site is supporting. The encoding of the database server must be the same as the encoding of the site content, including JSP files and repository content. This encoding should match the `EncodingTyper` encoding. For example, if you are setting up a database for one or more

Western-European languages, the encoding of the database server should be ISO8859_1. If you are setting up a Japanese-locale site that serves content in SJIS, the encoding of the database server should be SJIS. If you are setting up a web site to support multiple locales, including Western-European languages and non-Latin character languages, the encoding of the database server should be Unicode. You should also make sure that the database you use has the appropriate character set installed and selected in order to support multi-byte character sets.

The following are three example configurations:

- If the web content is in one or more Western European languages, set the encoding of the database server to `ISO8859_1`.
- If the web site is serving Japanese content that is in `SJIS`, set the encoding of the database server to `SJIS`.
- If the web site is serving Japanese content and Western-European content, set the encoding of the database server to Unicode.

Evaluate the needs of your web site and choose the appropriate encoding.

Setting the Email Encoding

When you send targeted email, you must make sure the character set used is supported by the most popular email clients. For example, Japanese JSP templates are stored by default in the `SJIS` character set (on Windows) or the `EUC` character set (on UNIX). These are 8-bit encodings, while the default encoding for most Japanese email clients is the 7-bit ISO-2022-JP character set. You can configure an ATG application to send email in the ISO-2022-JP encoding.

The `TemplateEmailSender` component can translate email messages to different encodings for transmission. For example, you can specify that `SJIS` and `EUC` should be translated to `JIS`.

To specify the mapping between the template encoding and the message encoding, set the `emailEncodingMap` property of the `TemplateEmailSender` component used to send out the email. This property is a Hashtable that can list any number of mappings. For example, the default setting of this property is:

```
emailEncodingMap=SJIS=iso-2022-jp,EUC=iso-2022-jp
```

This setting specifies that if the template uses either the `SJIS` or `EUC` character set, the resulting email messages should use the `ISO-2022-JP` character set. (`ISO-2022-JP` is the IANA/MIME equivalent name for the `JIS` Java charset.) You can change these mappings, or append additional mappings of the form `template-encoding=message-encoding` (separated by commas).

You can use either the Java charset or the IANA/MIME names for the character sets. Typically, the template encoding is specified by its Java charset name and the message encoding is specified by its IANA/MIME name. (The default setting shown above uses this convention.) If you specify the Java charset name for the message encoding, `TemplateEmailSender` uses the equivalent IANA/MIME name in the message header.

For more information about targeted email, see the *ATG Personalization Programming Guide*.

Internationalizing Content Repositories

There are various ways to set up SQL-based content repositories to store content for multiple site locales. For example, you can set up a separate repository for each locale, or you can store content for all locales in a single repository and add locale-specific attributes to each repository item.

Multiple Repositories

With this method, you create a separate content repository for each locale. Then you write separate targeting rules for each locale, with each rule referencing the appropriate repository.

One advantage of this method is that it lets you target different pieces of content to different locales. For example, site visitors from Germany might not be interested in the same news articles as site visitors from the US; with separate targeting rules or scenarios for each locale, you can display only those articles that are relevant to each visitor.

Disadvantages include the need to maintain multiple copies of each item (each item is duplicated in each locale-specific repository). In addition, storing separate values for each locale-sensitive property can take more space in the database than storing a single value.

Single Repository

With this method, you configure a single repository that holds content for all locales, and you include a `locale` attribute for each repository item. Then you write a targeting rule or scenario that matches the `RequestLocale`’s `localeString` property to the `locale` attribute of the content repository item, thereby allowing you to display content that is appropriate for each user’s language preference.

Locale-specific properties of each item are stored in multi-valued tables that hold many different language versions of the same information. This method therefore has the advantage of requiring you to maintain only one copy of each repository item, avoiding the duplication of the multiple repository method.

Disadvantages include the need to use a single character encoding that is appropriate for all content locales in the repository. In addition, a large repository that contains items for multiple locales might be less convenient to work with than a set of smaller repositories, each containing items for only one locale.

For more advantages and disadvantages of each method, and for a detailed description of how to set up SQL content repositories for an internationalized site, see the *ATG Commerce Programming Guide*.

The Motorprise demo application (ATG Business Commerce) uses the single repository method to store content in two languages, English and German. For information, see the *ATG Business Commerce Reference Application Guide*.

The Quincy Funds demo application stores content for four locales in a single repository, which is a combination SQL/file system repository. All content is encoded in UTF-8. For more information, refer to the *ATG Quincy Funds Demo Documentation*.

Using the EncodingTyper Component with Content Repositories

If you use the `EncodingTyper` component to determine character encodings, you must configure it to match the locale-based repository directories to encodings; the mapping works in the same way as the `EncodingTyper` mapping of page directories to encodings. See Using the EncodingTyper to Set the Character Encoding for more information.

Localizing an SQL Content Repository Definition File

The configuration file that defines an SQL content repository contains various values that are used in the content repository editor in the ATG Control Center. For example, each item has a `display-name` property and a `description` property whose values are labels that can be used to identify them in the editor. You can localize the definition file so that these values appear in a different language. For detailed information, see the *ATG Repository Guide*.

Localizing User Messages

When designing an ATG web site for internationalization, you should move all text messages from your Java code to `ResourceBundle.properties` files and content pages. You can use `ResourceBundle` objects for any user messages that display in the web site. It is often recommended that you use content pages instead of `ResourceBundles` to store session/request-scoped user messages. This is because content pages are more accessible to web designers than `ResourceBundles`.

You can use the `Switch` servlet bean to generate user messages dynamically according to the event that has occurred. When an error event or any other event that generates a user message occurs, a parameter is set indicating that a message should be displayed. An additional parameter, which is the key, is set that indicates which message should be displayed. The `Switch` servlet bean generates the text contained within the `<oparam> </oparam>` tags for a given key, and this is the text you translate for each locale in a multi-locale web site.

You can then use any one of the methods described earlier in this chapter to determine the page that is appropriate for each user's locale. For example, you can use the user profile's `locale` property.

In the following JSP example, you translate the text that is shown in italics:

```
<dsp:oparam name="keyA">
Text to translate
</dsp:oparam>
```

For information on translating text within content pages, see Creating Locale-Specific Content Pages. For information on the `Switch` servlet bean, refer to the *ATG Page Developer's Guide*.

Creating Locale-Specific Content Pages

In an internationalized ATG web site, JSPs display locale-specific web site content.

Content Page Translation

Content pages for use in an ATG application contain standard HTML tags and JSP tags. When translating text within or between JSP tags, translate only the text that appears to the user; all other code should remain in English.

The following tags require translation:

- `HTML`
- `oparam`
- `param`
- `input`

You must also localize content page directories, as described at the end of this section. For information about localizing user messages, see the previous section.

HTML

Standard HTML text, such as paragraph text, anchor tag values, text in lists, and submit button values must be translated.

oparam

In general, translate the text between the`<oparam>` `</oparam` > tags. Do not translate the text within the `<oparam` > tag. In the following JSP example, you translate the text that is shown in italics:

```
<dsp:oparam name="keyA">
Text to translate
</dsp:oparam>
```

param

Take great care when translating text within `<param` > tags. In some cases, text within `<param>` tags should be translated; in other cases, it should not.

value

In the following JSP example, the `value` text should be translated.

```
<dsp:include page="header.jsp">
   <dsp:param name="storename" value="Text to translate"/>
</dsp:include>
```

key

The values associated with each key value should be translated.

bean

No text should be translated, because the <param> value is defined as a bean property, as shown in the following example:

```
<dsp:param bean="SurveyBean.formError" name="value"/>
```

Embedding within anchor Tags

In the following example, the <param> tag is nested within an anchor tag. In this case, the <param> value should be translated. In addition, the text between the <a href></a> tags should be translated.

```
<dsp:a href="rainbow.jsp">
  <dsp:param name="position" value="Text to translate"/>
    Text to translate
</dsp:a>
```

For more information on <param> tags, refer to the *ATG Page Developer's Guide*

input

There are certain cases when text within <input> tags should be translated and certain cases when it should not.

Default Values in Text Bean Input Tags

A default value specified for the value attribute. This value is visible to the user. In this case, the default value text for this attribute should be translated.

Default Values in Hidden Bean Input Tags

No text should be translated. The default value of the value attribute is a hidden value that is not visible to the user.

Checkbox Input Tags

If input tags are used to display a checkbox, no text should be translated.

If multiple options are displayed with checkboxes, the text following the input tag should be translated.

Radio Button Input Tags

If input tags are used to display radio buttons, no text should be translated.

Image Input Tags

If an image input is represented as an image (taken from the src attribute), no text should be translated.

Submit Input Tags

In the case of submit input tags, the value associated with the submit button should be translated. For example, if the `value` attribute specifies the button value the text should be translated.

Content Page Directories

If your site hosts multiple visitor locales, parallel sets of content pages must be contained in locale-specific directories in the document root. If you design your site to serve content to multiple visitor locales, set up a separate page directory for each locale. Each locale-specific directory should be at the same level directly under the context root for JSPs.

All web site content pages should be mirrored for each locale, even if only a portion of the content in each directory is translated.

The following example shows the directory structure for a JSP-based web application—in this case, the Quincy Funds demo:

```
DSSJ2EEDemo/j2ee-apps/QuincyFunds/web-app/en/
DSSJ2EEDemo/j2ee-apps/QuincyFunds/web-app/fr/
DSSJ2EEDemo/j2ee-apps/QuincyFunds/web-app/de/
DSSJ2EEDemo/j2ee-apps/QuincyFunds/web-app/ja/
```

If you use the `EncodingTyper` component to determine the character encoding for the content pages, you must then configure the `EncodingTyper` to map each of the directories to the correct encoding. For more information, see Using the EncodingTyper to Set the Character Encoding in this chapter.

Designing a Multi-Locale Entry Page

For a multi-locale ATG web site, include a single top-level `index.jsp` page as an entry page for all users. Use a `Switch` servlet in this page to check the `language` setting of the user's request locale object (`RequestLocale.locale.language`) and then redirect the request to the index page of the matching language directory. For example, if a user chooses French on the registration page, thereby setting the request's `language` property to `fr`, he or she is redirected to the index page in the French directory.

This example shows the `Switch` servlet from the top-level `index.jsp` file in the Quincy Funds demo:

```
<dsp:setvalue bean="/atg/dynamo/servlet/RequestLocale.refresh" value=" "/>
<dsp:droplet name="/atg/dynamo/droplet/Switch">
  <dsp:param bean="/atg/dynamo/servlet/RequestLocale.locale.language"
    name="value"/>
  <dsp:oparam name="fr">
      <dsp:droplet name="/atg/dynamo/droplet/Redirect">
        <dsp:param name="url" value="fr/index.jsp"/>
      </dsp:droplet>
 </dsp:oparam>
  <dsp:oparam name="de">
      <dsp:droplet name="/atg/dynamo/droplet/Redirect">
        <dsp:param name="url" value="de/index.jsp"/>
      </dsp:droplet>
 </dsp:oparam>
  <dsp:oparam name="ja">
      <dsp:droplet name="/atg/dynamo/droplet/Redirect">
        <dsp:param name="url" value="ja/index.jsp"/>
      </dsp:droplet>
 </dsp:oparam>
  <dsp:oparam name="en">
      <dsp:droplet name="/atg/dynamo/droplet/Redirect">
        <dsp:param name="url" value="en/index.jsp"/>
      </dsp:droplet>
 </dsp:oparam>
   <dsp:oparam name="default">
      <dsp:droplet name="/atg/dynamo/droplet/Redirect">
        <dsp:param name="url" value="en/index.jsp"/>
      </dsp:droplet>
</dsp:oparam>
</dsp:droplet>
```

Converting Properties Files to Escaped Unicode

The Java compiler and other Java tools can process only files that contain Latin-1 and/or Unicode-encoded (\uddd notation) characters. To view a `.properties` file that contains non-Latin-1 font characters, you must convert the `.properties` file into a format that the ATG Control Center can read and process. In order to convert the `.properties` file to the appropriate format, you must run the file through the Java Native-to-ASCII Converter (`native2ascii`). This utility is supplied with your version of the JDK, and it converts the non-Latin font characters into escaped Unicode in the format `\uxxxx`.

For example, the file `ManagerAppResources_ja_SJIS.properties` contains Japanese characters. To convert the file into a `ManagerAppResources_ja.properties` file in escaped Unicode, run the following command:

```
native2ascii ManagerAppResources_ja_sjis.properties
ManagerAppResources_ja.properties
```

If the JDK is properly installed, you should be able to run this command from any directory.

Note: All `.properties` files, including `ResourceBundle.properties` files, must be run through the Native-to-ASCII Converter if they contain non-Latin 1 fonts or non-Unicode characters. You should run the Native-to-ASCII Converter on each file or group of files as you test the internationalization of your site. If you do not run the Converter, your internationalization changes do not appear in the ATG Control Center.

Localizing the Profile Repository Definition

If your ATG product suite includes the Personalization module, and you are setting up a web site for a non-Latin locale, you can perform some localization of the site by translating the Strings in the profile repository definition file, `userprofile.xml`. The Strings in this file are actually keys that link to a `ResourceBundle` file named `UserProfileTemplateResources.properties`. The Strings in this ResourceBundle appear in the People and Organizations section of the ATG Control Center, and they might also appear in any web site pages whose content you generate directly from these values in the profile repository. If you translate these Strings, the translated values appear in the ATG Control Center as well as in your web site.

The following example is taken from the `UserProfileTemplateResources.properties` file. The keys appear on the left, and the text to translate is in italic font on the right side of the equals sign:

```
# item descriptor User
itemDescriptorUser=User
securityStatus=security-status
id=Id
login=login-name
password=password
member=member
firstName=first-name
middleName=middle-name
lastName=last-name
```

For more information on profile repositories, see the *ATG Personalization Programming Guide*.

Localizing Profile Group Names, Scenario Names, and Similar Items

This section describes how to configure the machine that you use to create items such as profile groups, targeters, and scenarios so that their names are saved in an appropriate character set. This step is optional for delivering internationalized web content to end users; it is a localization issue that applies only to items in the ATG Control Center interface. (The names of these items as they appear in the ATG Control Center are also their file names.) These items are used by application developers only, not by site visitors.

The names of items such as profile groups, targeters, and scenarios are saved in the encoding specified in the Java system's `file.encoding` property. For example, if you want to create an EUC profile group name, set the `file.encoding` property to EUC. If you want to create a Greek group name, set the `file.encoding` property to ISO8859-7. To create group names in more than one language, choose an encoding that supports all the languages you require; for example, to create group names in both in Russian and French, set the `file.encoding` property to UTF-8.

To set the `file.encoding` property, set the `JAVA_ARGS` variable as follows:

`JAVA_ARGS=-Dfile.encoding=UTF-8` (or the encoding you require).

On Solaris, you can set the machine's default locale as an alternative to setting the `file.encoding` property. To set the locale to `eucJP`, for example, do the following:

```
setenv LANG ja
setenv LC_ALL ja
```

To set it to UTF-8, do the following:

```
setenv LANG ja_JP.UTF-8
setenv LC_ALL ja_JP.UTF-8
```

Consider the following example: you use the ATG Control Center to create a scenario on a machine that is using a Japanese version of an ATG product. The appropriate character sets are installed on the machine. When you create a scenario, the ATG Control Center appears to allow you to enter the scenario's name using Japanese characters; however, when you save the scenario, its name displays as a series of question marks. To correct the problem, check that the Java `file.encoding` property is set to an appropriate value as shown above.

Changing Date and Currency Formats

The ATG `date` tag converter lets you display Java dates in a variety of formats. For more information about the `date` tag converter, see Creating Custom Tag Converters in the Working with Forms and Form Handlers chapter.

Some date formats, including Japanese and Chinese dates, include date characters after the month value, but not after the day and year values. You need to supply the date characters in non-Western format dates yourself, using the ' (single quote) escape character. For example:

```
<VALUEOF BEAN="myTest.someDate" date="G yyyy'"X'MMMMd'"X'"></VALUEOF>
```

where `X` is the non-Latin character signifying the year and day characters.

Changing Currency Formats

ATG includes several currency conversion utilities, including the `currencyConversion` tag converter (class `atg.droplet.CurrencyTagConverter`), that you can add to JSPs to display currencies according to locale-specific rules. For more information on how tag converters work and how to create your own tag converters, see Creating Custom Tag Converters in the Working with Forms and Form Handlers chapter of this manual.

Note also that the ATG Business Commerce demo application, Motorprise, contains examples of localized currency formatting. For more information, refer to the Motorprise documentation.

Using Third-Party Software on an Internationalized Site

In order to run an internationalized ATG web site, all third-party software that is used in the web site must be compliant with Java internationalization standards. In other words, the JDBC drivers, servers, databases, operating systems, browsers, search engines, and ATG Connectors must be compliant with Java internationalization standards.

Appendix A: Disposable Class Loader

In most cases, when you modify a Java class definition during development, you must not only recompile the class, but also reassemble your application to load the new class definition. Nucleus can also use a disposable Class Loader to instantiate components and to resolve class names in general, which under some circumstances you might be able to use to reduce the number of times you need to reassemble your application during development. This disposable Class Loader loads classes from one or more directories defined by the `atg.nucleus.class.path` system variable. The value of this variable is a comma-delimited list of directories or URLs. These classes must not exist in the regular CLASSPATH; if they do, they are loaded by the regular class loader and are not reloadable. Nucleus then uses a special disposable Class Loader to load the classes specified by `atg.nucleus.class.path`, if it cannot find those classes in the regular CLASSPATH.

To use the Class Loader:

1. Be careful to segregate the classes you are changing in their own build directory tree, outside the CLASSPATH.
2. Specify that build tree in the `atg.nucleus.class.path` system variable. You can do this by adding the following Java argument to your `environment.sh` or `environment.bat` file:

   ```
   set JAVA_ARGS=
   %JAVA_ARGS% -Datg.nucleus.class.path=file:/path/to/build/tree
   ```

   The value of the `atg.nucleus.class.path` system variable is a comma-delimited list of URLs. For example, it might look like this in Windows:

   ```
   set JAVA_ARGS=%JAVA_ARGS% -Datg.nucleus.class.path=
   file:///C:\ATG\Dynamo\MyFiles,file:///C:\Zapf\Dynamo\YourFiles
   ```

   or like this in UNIX:

   ```
   JAVA_ARGS="-Datg.nucleus.class.path=
   file:///work/ATG/Dynamo/MyFiles,file:///relax/Zapf/Dynamo/YourFiles
   ${JAVA_ARGS}"
   ```

3. When you change a class, remove from Nucleus any instances of that class. You can do this by selecting each such component in the ATG Control Center Components window by selecting File > Stop Component or by right-clicking on the component and selecting Stop Component from the pop-up menu.
4. Select Tools > Make New Class Loader in the ATG Control Center Components window.

The disposable class loader does not work with any code that uses `java.lang.Class.forName` to resolve a class,.

Use Caution in Making New Class Loaders

Every Java `Class` has a reference to the class loader that loaded it. Classes of the same name loaded by two different class loaders are considered to be completely different. This causes `instanceof` checks to fail, even if the class is identical in both loaders. As a result, you need to be very careful when using the disposable Class Loader feature. It is very easy to find yourself in a situation where you have two components that appear to be identical, but are in fact inconsistent.

For example, you might perform these tasks:

1. Load a class named `Arbitrary`
2. Instantiate it as object `arbitrary1`
3. Switch class loaders
4. Reload class `Arbitrary`
5. Instantiate it as `arbitrary2`

If `arbitrary2` resolves `arbitrary1` as an Object and casts it to the type `Arbitrary`, a `ClassCastException` is thrown. The disposable Class Loader feature, therefore, is really most helpful when you have a set of classes to be reloaded whose instances are easy to completely expunge from the server prior to creating a Class Loader.

Appendix B: DAF Database Schema

ATG's database schema includes the following types of tables:

Security Tables

DMS Tables

Security Tables

ATG uses the following tables to store security information:

- das_gsa_subscriber
- das_id_generator
- das_secure_id_gen
- das_account
- das_group_assoc
- das_sds

das_gsa_subscriber

This table contains information used by the SQL Repository caching feature when you use `cache-mode="distributed"` on one or more item descriptors. This table is automatically populated by the SQL Repository at application startup and used by each server to determine which other servers need to receive cache invalidation events when items are modified.

Column	Data Type	Constraint
`id`	INTEGER	NOT NULL
	The unique identifier associated with the server	
`address`	VARCHAR(15)	NOT NULL
	The IP address of the `GSAEventServer`	
`port`	INTEGER	NOT NULL

	The port number of the `GSAEventServer`	
`itemdescriptor`	VARCHAR(256)	NOT NULL
	The name of the `itemdescriptor` where you set the cache-mode	

das_id_generator

The `SQLIdGenerator` service uses this table to generate IDs.

Column	Data Type	Constraint
`id_space_name`	VARCHAR(60)	NOT NULL
	A string that uniquely identifies an `IdSpace` within an `IdGenerator`. An `IdGenerator` can refer to an `IdSpace` using this name.	
`seed`	NUMERIC(19)	NOT NULL
	The first ID in the space to reserve.	
`batch_size`	INTEGER	NOT NULL
	How many IDs to reserve at a time.	
`prefix`	VARCHAR(10)	NULL
	A string to prepend to the beginning of all string IDs generated from this `IdSpace`.	
`suffix`	VARCHAR(10)	NULL
	A string to append to the end of all string IDs generated from this `IdSpace`.	

das_secure_id_gen

The `ObfuscatedSQLIdGenerator` service uses this table to generate Ids that are difficult to guess.

Column	Date Type	Constraint
`id_space_name`	VARCHAR(60)	NOT NULL
	A string that uniquely identifies an `IdSpace` within an `IdGenerator`. An `IdGenerator` can refer to an `IdSpace` using this name.	

seed	NUMERIC(19)	NOT NULL
	The first ID in the space to reserve.	
batch_size	INTEGER	NOT NULL
	How many IDs to reserve at a time.	
ids_per_batch	INTEGER	NULL
	The number of IDs to reserve per batch.	
prefix	VARCHAR(10)	NULL
	A string to prepend to the beginning of all string IDs generated from this IdSpace.	
suffix	VARCHAR(10)	NULL
	A string to append to the end of all string IDs generated from this IdSpace.	

das_account

This table contains a list of accounts, groups, and privileges to be used by ATG and the ATG Control Center for administration purposes.

Column	Data Type	Constraint
account_name	WVARCHAR(254)	NOT NULL
	The name of the account a user types to log in.	
type	INTEGER	NOT NULL
	The type of account: (1) a login account that a user can use to log in, (2) a group account used for organization, or (3) a privilege account to control access to an ATG application and/or ATG Control Center features.	
first_name	WVARCHAR(254)	NULL
	For a login (type 1) account, the first name of the user.	
last_name	WVARCHAR(254)	NULL
	For a login (type 1) account, the last name of the user.	
password	VARCHAR(254)	NULL
	For a login (type 1) account, the encrypted password that verifies the user's identity.	

`description`	WVARCHAR(254)	NULL
	For a group (type 2) account, this is the name of the account that displays in the ATG Control Center.	

das_group_assoc

This table associates accounts with the groups and privileges of which they are members.

Column	Data Type	Constraint
`account_name`	WVARCHAR(254)	NOT NULL
	The name of an account that has a group or privilege association.	
`sequence_num`	INTEGER	NOT NULL
	An index number used to define the order of groups. This is required by the SQL Repository for array properties.	
`group_name`	WVARCHAR(254)	NOT NULL
	The name of the group of which the account is a member.	

das_sds

This table contains information about data source switching. Each row in the table corresponds to the state of a single switching data source service.

Column	Data Type	Constraint
`sds_name`	VARCHAR(50)	NOT NULL
	The name of the switching data source.	
`curr_ds_name`	VARCHAR(50)	NULL
	The name of the data source that the switching data source is currently using.	
`dynamo_server`	VARCHAR(80)	NULL
	A pseudo-ID for the ATG server where the switching data source is running. This can be, but does not have to be, a unique ID.	
`last_modified`	TIMESTAMP	NULL

	The time of the last switch operation or the time the switching data source was first started.	

DMS Tables

ATG's DMS messaging system uses the following tables to store messaging data:

- dms_client
- dms_queue
- dms_queue_recv
- dms_queue_entry
- dms_topic
- dms_topic_sub
- dms_topic_entry
- dms_msg
- dms_msg_properties
- dms_limbo
- dms_limbo_msg
- dms_limbo_replyto
- dms_limbo_body
- dms_limbo_props
- dms_limbo_ptypes
- dms_limbo_delay

dms_client

The list of ATG instances that started an SQL-JMS instance pointing to this database. Clients listed in this table might or might not still be active.

Column	Data Type	Constraint
`client_name`	VARCHAR(250)	NOT NULL
	The unique name the client uses to identify itself. By default this name is a combination of the ATG server's DRP IP address and port.	
`client_id`	NUMERIC(19)	NULL

	The unique numeric representation of the client used internally by the SQL-JMS system.	

dms_queue

The list of queues available for messaging.

Column	Data Type	Constraint
`queue_name`	VARCHAR(250)	NULL
	The unique name of the queue used by clients to send messages to and receive messages from a specific queue.	
`queue_id`	NUMERIC(19)	NOT NULL
	The unique numeric representation of the queue used internally by the SQL-JMS system.	
`temp_id`	NUMERIC(19)	NULL
	Denotes whether or not the queue is a temporary queue. If the queue is a temporary queue, the column contains the client ID of the client that created the temporary queue. If the queue is not a temporary queue, the column contains the value zero.	

dms_queue_recv

The list of queue receivers that are registered with a queue. Each row represents a single receiver listening to a queue.

Column	Data Type	Constraint
`client_id`	NUMERIC(19)	NULL
	The unique numeric representation of the client used internally by the SQL-JMS system.	
`receiver_id`	NUMERIC(19)	NOT NULL
	The numeric ID of the receiver listening to the queue.	
`queue_id`	NUMERIC(19)	NULL
	The numeric ID of the queue the receiver is listening to.	

dms_queue_entry

The list of messages currently in any queue. Each row in this table represents a single message in a queue.

Column	Data Type	Constraint
`queue_id`	NUMERIC(19)	NOT NULL
	The queue ID of the queue this message is currently in.	
`msg_id`	NUMERIC(19)	NOT NULL
	The unique numeric representation of the message used internally by the SQL-JMS system.	
`delivery_date`	NUMERIC(19)	NULL
	A Java long date value that specifies when the message should be delivered. The value is a date/time in the form of UTC milliseconds from the epoch start (1 January 1970 0:00 UTC). If there is to be no delayed delivery of the message, this column effectively holds a timestamp of when the message was put into the queue allowing it to be delivered as soon as possible.	
`handling_client_id`	NUMERIC(19)	NULL
	The client ID of the client that is attempting to handle this message. If no client is attempting to handle this message yet, this column contains the value –1.	
`read_state`	NUMERIC(19)	NULL
	The current state of the message. A message that is not currently being handled by a client has a value of zero. A message that is being handled has a non-zero value. Messages that are handled successfully are deleted from this table.	

dms_topic

The list of topics available for messaging.

Column	Data Type	Constraint
`topic_name`	VARCHAR(250)	NULL
	The unique name of the topic used by clients to send messages to and receive messages from a specific topic.	

Column	Data Type	Constraint
`topic_id`	NUMERIC(19)	NOT NULL
	The unique numeric representation of the topic used internally by the SQL-JMS system.	
`temp_id`	NUMERIC(19)	NULL
	Denotes whether or not the topic is a temporary topic. If the topic is a temporary topic, the column contains the client ID of the client that created the temporary topic. If the topic is not a temporary topic, the column contains the value zero.	

dms_topic_sub

The list of topic subscribers that are currently registered with a topic. Each row represents a single subscriber listening to a topic.

Column	Data Type	Constraint
`client_id`	NUMERIC(19)	NULL
	The client ID of the client that created the subscriber.	
`subscriber_name`	VARCHAR(250)	NULL
	The unique name used by the client to identify the subscriber.	
`subscriber_id`	NUMERIC(19)	NOT NULL
	The subscriber ID of the subscriber receiving the message.	
`topic_id`	NUMERIC(19)	NULL
	The topic ID of the topic the subscriber is registered to listen to.	
`durable`	NUMERIC(1)	NULL
	Denotes whether or not the subscriber is durable. Durable subscribers have a value of 1. Non-durable subscribers have the value zero.	
`active`	NUMERIC(1)	NULL
	Denotes whether or not the subscriber is flagged as active by the client.	

dms_topic_entry

The list of messages waiting to be handled by a subscriber listening to a topic. Each row in this table represents a single message for a subscriber.

Column	Data Type	Constraint
`subscriber_id`	NUMERIC(19)	NOT NULL
	The subscriber ID of the subscriber receiving the message.	
`msg_id`	numeric(19)	NOT NULL
	The unique identifier of the message used internally by the SQL-JMS system.	
`delivery_date`	NUMERIC(19)	NULL
	A Java long date value that specifies when the message should be delivered. The value is a date/time in the form of UTC milliseconds from the epoch start (1 January 1970 0:00 UTC). If there is to be no delayed delivery of the message, this column effectively holds a timestamp of when the message was put into the queue allowing it to be delivered as soon as possible.	
`read_state`	NUMERIC(19)	NULL
	The current state of the message. A message that is not currently being handled by a client has a value of zero. A message that is being handled has a non-zero value. Messages that are handled successfully are deleted from this table.	

dms_msg

The list of actual messages currently in the SQL-JMS system. Each row represents a single message that might be in a single queue or waiting to be received by multiple topic subscribers.

Column	Data Type	Constraint
`msg_class`	VARCHAR(250)	NULL
	The Java class of the message.	
`has_properties`	NUMERIC(1)	NULL
	Whether or not the message has properties beyond the standard JMS header properties, such as implementation or application specific properties.	

Column	Data Type	Constraint
`reference_count`	NUMERIC(10)	NULL
	The number of topic subscribers still waiting to receive the message.	
`msg_id`	NUMERIC(19)	NOT NULL
	The unique identifier of the message used internally by the SQL-JMS system.	
`timestamp`	NUMERIC(19)	NULL
	JMS header property: the time the message was handed off to the provider to be sent.	
`correlation_id`	VARCHAR(250)	NULL
	JMS header property: the correlation ID. Currently, this property is unsupported and this column is always null.	
`reply_to`	NUMERIC(19)	NULL
	JMS header property: the destination where a reply to the message should be sent. This column uses the topic or queue ID to represent the reply destination. If no reply-to was specified in the JMS message the column has the value zero.	
`destination`	NUMERIC(19)	NULL
	JMS header property: the destination where the message is being sent. This column uses the topic or queue ID to represent the destination.	
`delivery_mode`	NUMERIC(1)	NULL
	JMS header property: the delivery mode of the message.	
`redelivered`	NUMERIC(1)	NULL
	JMS header property: an indication of whether the message is being redelivered.	
`type`	VARCHAR(250)	NULL
	JMS header property: the message type.	
`expiration`	NUMERIC(19)	NULL
	JMS header property: the message’s expiration value.	
`priority`	NUMERIC(1)	NULL
	JMS header property: the message priority.	

Column	Data Type	Constraint
`small_body`	VARBINARY(250)	NULL
	The body of the message if the body is within the preset size for small bodies.	
`large_body`	LONG VARBINARY	NULL
	The body of the message if the body is larger than the preset size for small bodies.	

dms_msg_properties

This table contains the non-standard properties for messages currently in the SQL-JMS system. Each row represents one property for a single message. A single message with multiple non-standard properties has multiple rows in the table.

Column	Data Type	Constraint
`msg_id`	NUMERIC(19)	NOT NULL
	The message ID of the message with which the property is associated.	
`data_type`	NUMERIC(1)	NULL
	The data type of the property represented as a number.	
`name`	VARCHAR(250)	NOT NULL
	The name of the property used by the JMS client to identify it within the JMS message.	
`value`	VARCHAR(250)	NULL
	The value of the property represented as a String.	

dms_limbo

This table identifies the Patch Bay instances that store delayed messages.

Column	Data Type	Constraint
`limbo_name`	VARCHAR(250)	NOT NULL
	The name of the ATG server Patch Bay is on, which comes from `/atg/dynamo/service/ServerName`.	

`limbo_id`	NUMERIC(19)	NOT NULL
	The generated ID for internal identification of the Patch Bay instance.	

dms_limbo_msg

The main table for delayed messages. Each row corresponds to a single message.

Column	Data Type	Constraint
`msg_id`	NUMERIC(19)	NOT NULL
	A generated ID identifying the message	
`limbo_id`	NUMERIC(19)	NOT NULL
	The generated ID for internal identification of the instance	
`delivery_date`	NUMERIC(19)	NOT NULL
	When the message should be sent, in system milliseconds	
`delivery_count`	NUMERIC(2)	NOT NULL
	The counter for failures to send a delayed message	
`msg_src_name`	VARCHAR(250)	NOT NULL
	The name of the message source that produced this message	
`port_name`	VARCHAR(250)	NOT NULL
	The output port where this message is going	
`msg_class`	VARCHAR(250)	NOT NULL
	The actual class string of the message class—that is, `getClass().getName()`	
`msg_class_type`	NUMERIC(1)	NOT NULL
	The actual class of the message object	
`jms_type`	VARCHAR(250)	NULL
	The JMS header type of the message	
`jms_expiration`	NUMERIC(19)	NULL
	The JMS header expiration of the message	
`jms_correlationid`	VARCHAR(250)	NULL

	The JMS header correlation ID of the message	

dms_limbo_replyto

This table stores the `reply to` headers for delayed messages.

Column	Data Type	Constraint
`msg_id`	NUMERIC(19)	NOT NULL
	A generated ID identifying the message.	
`jms_replyto`	VARBINARY(500)	NULL
	The JMS header `reply to` of the message.	

dms_limbo_body

This table stores the message bodies for delayed messages.

Column	Data Type	Constraint
`msg_id`	NUMERIC(19)	NOT NULL
	A generated ID identifying the message.	
`msg_body`	LONG VARBINARY	NULL
	The body of the specified message type—for example, object, stream, and so on.	

dms_limbo_props

This table stores the message properties for delayed messages.

Column	Data Type	Constraint
`msg_id`	NUMERIC(19)	NOT NULL
	A generated ID identifying the message.	
`prop_name`	VARCHAR(250)	NOT NULL

	The message property name.	
`prop_value`	VARCHAR(250)	NOT NULL
	The message property value.	

dms_limbo_ptypes

A sub-table of properties identifying the property types.

Column	Data Type	Constraint
`msg_id`	NUMERIC(19)	NOT NULL
	A generated ID identifying the message.	
`prop_name`	VARCHAR(250)	NOT NULL
	The message property name.	
`prop_type`	NUMERIC(1)	NOT NULL
	The property type.	

dms_limbo_delay

The table for messages in the redelivery process.

Column	Data Type	Constraint
`msg_id`	NUMERIC(19)	NOT NULL
	A generated ID identifying the message.	
`delay`	NUMERIC(19)	NOT NULL
	The delay between attempts to deliver the message (in milliseconds).	
`max_attempts`	NUMERIC(2)	NOT NULL
	Maximum number of times to attempt to deliver the message.	
`failure_port`	VARCHAR(250)	NOT NULL
	Port through which to send the message to failure destinations if all attempts to deliver the message are unsuccessful.	

Column	Data Type	Constraint
`jms_timestamp`	NUMERIC(19)	NULL
	JMS attribute used to create the new message for redelivery.	
`jms_deliverymode`	NUMERIC(10)	NULL
	JMS attribute used to create the new message for redelivery.	
`jms_priority`	NUMERIC(10)	NULL
	JMS attribute used to create the new message for redelivery.	
`jms_messageid`	VARCHAR(250)	NULL
	JMS attribute used to create the new message for redelivery.	
`jms_redelivered`	NUMERIC(1)	NULL
	JMS attribute used to create the new message for redelivery.	
`jms_destination`	VARBINARY(500)	NULL
	JMS attribute used to create the new message for redelivery.	

Appendix C: DMS Configuration File Tags

This appendix contains the Document Type Definition (DTD) for DMS configuration files. The DTD describes all XML tags that can be used in a DMS configuration file.

```
<!--
This is the XML DTD for the PatchBay 1.0 configuration file.
-->

<!--
The dynamo-message-system element describes the configuration of all
the elements of the dynamo messaging system.  It describes the patch
bay, the local JMS configuration, and the message registry.
-->
<!ELEMENT dynamo-message-system (patchbay, local-jms,
message-registry)>


<!--
The patchbay element defines the configuration of the PatchBay
component of the dynamo messaging system.  It begins with a
declaration of the JMS providers used in the system, then declares
each message-source, message-sink, and message-filter managed by the
PatchBay.

Used in: dynamo-message-system
-->
<!ELEMENT patchbay (provider*, message-source*, message-sink*,
message-filter*)>


<!--
The provider element describes one JMS provider that will be used in
the Patch Bay.  It assigns a name to the provider, describes where the
various ConnectionFactory interfaces can be found, and includes flags
describing the provider's transaction capabilities.

Used in: patchbay
-->
```

```
<!ELEMENT provider (provider-name, topic-connection-factory-name?,
queue-connection-factory-name?, xa-topic-connection-factory-name?,
xa-queue-connection-factory-name?, supports-transactions?,
supports-xa-transactions?, username?, password?, client-id?,
initial-context-factory?)>


<!--
The provider-name assigns a name to a provider for use by destination
references in the file.

Used in: provider, input-destination, output-destination

Example:
<provider-name>MQSeries</provider-name>
-->
<!ELEMENT provider-name (#PCDATA)>


<!--
The topic-connection-factory-name describes the JNDI location of the
provider's TopicConnectionFactory interface.

Used in: provider

Example:
<topic-connection-factory-name>
  dynamo:/dms/local/LocalDMSManager
</topic-connection-factory-name>
-->
<!ELEMENT topic-connection-factory-name (#PCDATA)>


<!--
The queue-connection-factory-name describes the JNDI location of the
provider's QueueConnectionFactory interface.

Used in: provider

Example:
<queue-connection-factory-name>
  dynamo:/dms/local/LocalDMSManager
</queue-connection-factory-name>
-->
<!ELEMENT queue-connection-factory-name (#PCDATA)>


<!--
The xa-topic-connection-factory-name describes the JNDI location of
the provider's XATopicConnectionFactory interface.
```

```
Used in: provider

Example:
<xa-topic-connection-factory-name>
  dynamo:/dms/local/LocalDMSManager
</xa-topic-connection-factory-name>
-->
<!ELEMENT xa-topic-connection-factory-name (#PCDATA)>


<!--
The xa-queue-connection-factory-name describes the JNDI location of
the provider's XAQueueConnectionFactory interface.

Used in: provider

Example:
<xa-queue-connection-factory-name>
  dynamo:/dms/local/LocalDMSManager
</xa-queue-connection-factory-name>
-->
<!ELEMENT xa-queue-connection-factory-name (#PCDATA)>


<!--
The supports-transactions element indicates if the provider supports
transactions through the Session.commit()/rollback() methods.

Used in: provider

Must be one of:
<supports-transactions>true</supports-transactions>
<supports-transactions>false</supports-transactions>
-->
<!ELEMENT supports-transactions (#PCDATA)>


<!--
The supports-xa-transactions element indicates if the provider supports
transactions through the XA interface.

Used in: provider

Must be one of:
<supports-xa-transactions>true</supports-xa-transactions>
<supports-xa-transactions>false</supports-xa-transactions>
-->
<!ELEMENT supports-xa-transactions (#PCDATA)>
```

```
<!--
The username element specifies the username that should be
provided when creating a new connection.

Used in: provider

Example:
<username>
  charles
</username>
-->
<!ELEMENT username (#PCDATA)>


<!--
The password element specifies the password that should be
provided when creating a new connection.

Used in: provider

Example:
<password>
  charles
</password>
-->
<!ELEMENT password (#PCDATA)>


<!--
The client-id element specifies the client identifier that will be
assigned to the connection.  This is primarily used to reconnect to
durable subscription state.

Used in: provider

Example:
<client-id>
  OrderProcessor
</client-id>
-->
<!ELEMENT client-id (#PCDATA)>


<!--

The initial-context-factory element specifies the nucleus name of a
component that implements the
atg.dms.patchbay.JMSInitialContextFactory interface.  This nucleus
component will be called on to create an InitialContext whenever a
```

```
JNDI name needs to be resolved for the provider (i.e., when resolving
the JNDI name of a Topic/QueueConnectionFactory, or a Topic or a
Queue).  If no initial-context-factory is supplied, then the JNDI
names will be resolved against a "vanilla" InitialContext (i.e., one
created by calling "new InitialContext()").

Used in: provider

Example:
<initial-context-factory>
  /atg/jmsproviders/providerx/InitialContextFactory
</initial-context-factory>
-->
<!ELEMENT initial-context-factory (#PCDATA)>


<!--

The message-source element describes one MessageSource.  It specifies
its Nucleus name, and also describes each of the MessageSource's
output ports.

Used in: patchbay
-->
<!ELEMENT message-source (nucleus-name, output-port*)>


<!--
The nucleus-name element specifies the absolute name of a global
Nucleus component.

Used in: message-source, message-sink, message-filter

Example:
<nucleus-name>
  /atg/commerce/sources/EmailSource
</nucleus-name>
-->
<!ELEMENT nucleus-name (#PCDATA)>


<!--
The output-port element specifies how one of the output ports is
connected to possibly many destinations.

Used in: message-source, message-filter
-->
<!ELEMENT output-port (port-name?, output-destination*)>

<!--
```

```
The redelivery-port element specifies how one of the redelivery ports is
connected to possibly many destinations.

Used in: message-sink, message-filter
-->
<!ELEMENT redelivery-port (port-name?, output-destination*)>

<!--
The port-name element specifies the name of an input or output port.

Used in: output-port, input-port

Example:
<port-name>
  DEFAULT
</port-name>
-->
<!ELEMENT port-name (#PCDATA)>


<!--
The output-destination describes one Destination to which Messages
through an output port should be sent.  Each destination describes the
JMS provider through which the Message should be sent, the JNDI name
of the Destination, whether the Destination is a Topic or Queue, and
what options should be set on Messages on their way out.

Used in: output-port
-->
<!ELEMENT output-destination (provider-name?, destination-name,
destination-type, priority?, delivery-mode?)>


<!--
The destination-name element specifies the JNDI name of the
Destination

Used in: output-destination, input-destination

Example:
<destination-name>
  localjms:/local/dcs/PurchaseEvents
</destination-name>
-->
<!ELEMENT destination-name (#PCDATA)>


<!--
The destination-type element specifies the type of the Destination
```

```
Used in: output-destination, input-destination

Must be one of:
<destination-type>Topic</destination-type>
<destination-type>Queue</destination-type>
-->
<!ELEMENT destination-type (#PCDATA)>


<!--
The priority element specifies the JMSPriority that should be assigned
to all Messages going to this Destination through this output-port.
The priority should be between 0 and 9 (inclusive).

Used in: output-destination

Example:
<priority>8</priority>
-->
<!ELEMENT priority (#PCDATA)>


<!--
The delivery-mode element specifies the JMSDeliveryMode that should be
assigned to all Messages going to this Destination through this
output-port.

Used in: output-destination

Must be one of:
<delivery-mode>PERSISTENT</delivery-mode>
<delivery-mode>NON_PERSISTENT</delivery-mode>
-->
<!ELEMENT delivery-mode (#PCDATA)>


<!--
The message-sink element describes one MessageSink.  It specifies its
Nucleus name, and also describes each of the MessageSink's input
ports.

Used in: patchbay
-->
<!ELEMENT message-sink (nucleus-name, input-port*, redelivery-port*)>


<!--
The input-port element specifies how one of the input ports receives
Messages from possibly many destinations.
```

```
Used in: message-sink, message-filter
-->
<!ELEMENT input-port (port-name?, input-destination*)>


<!--
The input-destination element describes one Destination from which
Messages are received and attributed to this input-port.  Each
Destination describes the JMS provider from which the Message should
be received, the JNDI name of the Destination, whether the Destination
is a Topic or Queue, the message selector to be used, and whether
local messages should be received.

Used in: input-port
-->
<!ELEMENT input-destination (provider-name?, destination-name,
destination-type, durable-subscriber-name?, message-selector?,
no-local?, redelivery?)>

<!--
The redelivery element describes the configuration parameters used
for message redelivery during failure conditions. max-attempts defines
the maximum number of delivery attempts by Patch Bay to the input
destination. The delay interval (specified in msec) defines how long a
message should be delayed before a redelivery is attempted. Finally
if the maximum number of delivery attempts has been reached then
the message will be redirected to the output port named through
the failure-output-port element.

Used in: input-destination
-->

<!ELEMENT redelivery (max-attempts, delay, failure-output-port)>
<!ELEMENT max-attempts (#PCDATA)>
<!ELEMENT delay (#PCDATA)>
<!ELEMENT failure-output-port (#PCDATA)>

<!--
The message-selector element describes the filter that will restrict
the flow of Messages from this Destination.

Used in: input-destination

Example:
<message-selector>
  JMSType = 'atg.dcs.Purchase'
</message-selector>
-->
<!ELEMENT message-selector (#PCDATA)>
```

```
<!--
The durable-subscriber-name element specifies the name of the durable
subscription to which this should subscribe.  This may only be
specified for Topic Destinations.  If this is not specified, a durable
subscription will not be used.

Used in: input-destination

Example:
<durable-subscriber-name>
  orders
</durable-subscriber-name>
-->
<!ELEMENT durable-subscriber-name (#PCDATA)>


<!--
The no-local indicates whether Messages sent to this Topic by the same
Session should not be received.  If true, then such messages are
blocked, otherwise such messages are received.  This may only be
specified for Topic destinations.  Defaults to false if not specified.

Used in: input-destination

Must be one of:
<no-local>true</no-local>
<no-local>false</no-local>
-->
<!ELEMENT no-local (#PCDATA)>


<!--
The message-filter element describes one MessageFilter.

Used in: patchbay
-->
<!ELEMENT message-filter (nucleus-name, input-port*, output-port*,
redelivery-port*)>


<!--
The local-jms element configures the Local JMS system that will be
used with the patch bay in the dynamo messaging system.  It configures
the JNDI prefix that will be used for the destination names, and also
names all of the queues and topics in the Local JMS system.

Used in: dynamo-message-system
-->
<!ELEMENT local-jms (jndi-prefix, topic-name*, queue-name*)>
```

```
<!--
The jndi-prefix element specifies what JNDI prefix should be prepended
to each topic or queue name to form the destination's JNDI name.  The
prefix should start with "/" and should not include the "localdms:".
The destination's JNDI name will be
"localdms:{jndi-prefix}{topic/queue-name}".

Used in: local-jms

Example:
<jndi-prefix>
  /local
</jndi-prefix>
-->
<!ELEMENT jndi-prefix (#PCDATA)>


<!--
The topic-name element specifies the name of a Topic in the Local JMS
system.  The name should begin with a "/", and must be unique among
both topic-name and queue-name elements.

Used in: local-jms

Example:
<topic-name>
  /ProfileEvents
</topic-name>
-->
<!ELEMENT topic-name (#PCDATA)>


<!--
The queue-name element specifies the name of a Queue in the Local JMS
system.  The name should begin with a "/", and must be unique among
both queue-name and queue-name elements.

Used in: local-jms

Example:
<queue-name>
  /ProfileEvents
</queue-name>
-->
<!ELEMENT queue-name (#PCDATA)>


<!--
```

```
The message-registry element is the root element of the
MessageRegistry configuration file.  It defines several message-family
elements.

Used in: dynamo-message-system
-->
<!ELEMENT message-registry (message-family*)>


<!--
The message-family element describes a group of message-type elements,
and may also recursively contain a set of message-family elements.

Used in: message-registry, message-family
-->
<!ELEMENT message-family (message-family-name, message-family*,
message-type*)>


<!--
The message-family-name element specifies the name of a
message-family.

Used in: message-registry, message-family

Example:
<message-family-name>atg.dcs</message-family-name>
-->
<!ELEMENT message-family-name (#PCDATA)>


<!--
The message-typer element describes one MessageTyper.

Used in: message-type
-->
<!ELEMENT message-typer (nucleus-name)>

<!--
The message-type element describes one mapping from JMSType to Object
class.

Used in: message-family
-->
<!ELEMENT message-type (jms-type, message-class, message-typer?, message-context?,
                        display-name?, display-name-resource?, expert?, hidden?,
                        description?, description-resource?, resource-bundle?)>


<!--
```

```
The jms-type element specifies the JMSType for this message type.  The
jms-type must be unique across all message types in the message
registry.

Used in: message-type

Example:
<jms-type>
  atg.dcs.Purchase
</jms-type>
-->
<!ELEMENT jms-type (#PCDATA)>


<!--
The message-class element specifies the fully-qualified class name of
the Java Bean that contains the message's data.

Used in: message-type

Example:
<message-class>
  atg.dcs.PurchaseMessage
</message-class>
-->
<!ELEMENT message-class (#PCDATA)>


<!--
The message-context element specifies the nature of the message's
originating context.  If omitted, then no assumptions are made
concerning the message's context.  The following values are recognized:

   request: the message originates in a request thread, and
            request- or session-specific values may be resolved
           via JNDI.

   session: the message originates in a session-specific context, and
            session-specific values may be resolved via JNDI.
Used in: message-type

Example:
<message-context>
  request
</message-context>
-->
<!ELEMENT message-context (#PCDATA)>

<!--
The display-name element specifies a GUI display name for an element
```

```
described in the patch bay definition file.

Example:
<display-name>
  Buys Product
</display-name>
-->
<!ELEMENT display-name (#PCDATA)>

<!--
The display-name-resource element specifies a GUI display name for an element
described in the patch bay definition file, which can be loaded from a resource
bundle.

Example:
<display-name-resource>
  buysProduct
</display-name-resource>
-->
<!ELEMENT display-name-resource (#PCDATA)>

<!--
The description element specifies a GUI description for an element
described in the patch bay definition file.

Example:
<description>
  Generated when user purchases a product
</description>
-->
<!ELEMENT description (#PCDATA)>

<!--
The description-resource element specifies a GUI description for an element
described in the patch bay definition file, which can be loaded from a resource
bundle.

Example:
<description-resource>
  buysProductDescription
</description-resource>
-->
<!ELEMENT description-resource (#PCDATA)>

<!--
The resource-bundle element specifies a resource bundle from which resources
for an element described in the patch bay definition file can be loaded.

Example:
<resource-bundle>
```

```
  atg.dms.Resources
</resource-bundle>
-->
<!ELEMENT resource-bundle (#PCDATA)>

<!--
The hidden element specifies a flag indicating that the given message type
should be hidden in a GUI.

Example:
<hidden>
  true
</hidden>
-->
<!ELEMENT hidden (#PCDATA)>

<!--
The expert element specifies a flag indicating that the given message type
should be hidden in a GUI from non-expert users.

Example:
<expert>
  true
</expert>
-->
<!ELEMENT expert (#PCDATA)>
```

Appendix D: ATG Modules

The following tables list the module names for the main ATG applications, demos, and reference applications. This is not an exhaustive list of all ATG modules.

Module	Description
`Admin.Init`	Adds missing administrative accounts for the ATG Control Center. For more information, see the Managing Access Control chapter.
`Admin.Reset`	Resets the default login accounts for the ATG Control Center. For more information, see the Managing Access Control chapter.
`DAF.Search`	Enables the ATG platform to use ATG Search to index and search content from product catalogs and other repositories. See the *ATG Search Administration Guide.*
`DAS-UI`	Enables an ATG server to accept connections from the ATG Control Center. **Note:** This module must be running if you want to use the ATG Control Center.
`DCC`	Runs the ATG Control Center in the same JVM used by the application server that the Nucleus-based application is running on. For more information, see the *ATG Installation and Configuration Guide*.
`DPS`	ATG Personalization
`DSS`	ATG Scenarios
`DSSJ2EEDemo`	ATG Adaptive Scenario Engine demo (Quincy Funds). **Note:** This demo is supported on Solid and Oracle databases only.
`RL`	Repository Loader. Takes files that are stored in a file system, converts them into repository items, and loads the items into the repository. To learn more, see the *ATG Repository Guide*.
`SQLJMSAdmin`	Browser-based administration interface for ATG's SQL JMS message system. For more information, see Using the SQL-JMS Administration Interface.

Content Administration Modules

Module	Description
`AssetUI`	Supports the building of browser-based user interfaces for an ATG Content Administration (versioned) environment. The module includes the Asset Picker and functionality related to it. Requires the `WebUI` module (see below).
`BizUI`	The ATG Business Control Center. Includes the Home page functionality and the View Mapping system.
`PublishingAgent`	Publishing Agent. Runs on production and staging servers and performs content deployment operations by communicating with the ATG Content Administration server.
`PublishingWebAgent`	Publishing web agent. Runs on the production and staging web servers and performs web content deployment operations by communicating with the ATG Content Administration server.
`Publishing.base`	ATG Content Administration. See the *ATG Content Administration Programming Guide* for more information.
`Publishing.WebAppRef`	The source module for the Web Application Reference Implementation provided with ATG Content Administration.
`Publishing.WebAppRefVer`	The versioning module for the Web Application Reference Implementation provided with ATG Content Administration.
`PubPortlet`	Supplies the portlets that make up the ATG Business Control Center interface. Including this module also causes the `Publishing.base`, `AssetUI`, and `BizUI` modules to be included. Include this module to perform most basic tasks in ATG Content Administration, such as product evaluation.
`WebUI`	Contains support for browser-based user interfaces. Examples are the tree-based asset browsing feature and a calendar widget.

Portal Modules

Module	Description
`Portal.gears`	Includes the Portal Application Framework and baseline gears.
`Portal.paf`	Portal Application Framework (PAF). At a minimum, you must include this module to use ATG Portal. To learn more about the PAF, see the *ATG Portal Administration Guide*.

`Portal.`*`gear-name`*
`Portal.`*`portlet-name`*
	Includes the specified gear or portlet. For example, if you create a gear called `productprices`, include the module `Portal.productprices`.

ATG Commerce Modules

Module	Description
`B2BCommerce`	ATG Business Commerce **Note:** To run ATG Commerce, you must use one and only one of the following modules: `B2BCommerce`, `B2BCommerce.Versioned`, `B2CCommerce`, or `B2CCommerce.Versioned`.
`B2BCommerce.Search`	Enables Business Commerce extensions to the ATG Commerce Search. See the *ATG Search Administration Guide* for more information.
`B2BCommerce.Versioned`	Use instead of `B2BCommerce` module if running ATG Merchandising. (Also requires modules `DCS-UI.Versioned` and `PubPortlet`.) **Note:** To run ATG Commerce, you must use one and only one of the following modules: `B2BCommerce`, `B2BCommerce.Versioned`, `B2CCommerce`, or `B2CCommerce.Versioned`. Including this module also includes `B2BCommerce`, `DCS.DynamicCustomCatalogs.Versioned`, and their modules.
`B2CCommerce`	ATG Consumer Commerce **Note:** To run ATG Commerce, you must use one and only one of the following modules: `B2BCommerce`, `B2BCommerce.Versioned`, `B2CCommerce`, or `B2CCommerce.Versioned`. Including this module also includes `DCS` and its modules.
`B2CCommerce.Versioned`	Use instead of `B2CCommerce` module if running ATG Merchandising. (Requires the `DCS-UI.management` and `PubPortlet` modules also.) **Note:** To run ATG Commerce, you must use one and only one of the following modules: `B2BCommerce`, `B2BCommerce.Versioned`, `B2CCommerce`, or `B2CCommerce.Versioned`. Including this module also includes `B2Commerce`, `DCS.Versioned`, and their modules.

Module	Description
`CommerceGears.orderapproval`	Order Approval Portal Gear (Requires ATG Portal also.) Including this module also includes `Portal.paf`, `Portal.authentication`, `Portal.templates`, `Portal.communities`, `B2BCommerce`, and their modules.
`CommerceGears.orderstatus`	Order Status Portal Gear (Requires ATG Portal also.) Including this module also includes `Portal.paf`, `Portal.authentication`, `Portal.templates`, `Portal.communities`, `DCS`, and their modules.
`Cybersource`	Third-party commerce module (from CyberSource Corp.) for authorizing credit cards, crediting and settling accounts, calculating taxes, and verifying addresses. Including this module also includes `DCS` and its modules.
`DCS`	Base ATG Commerce module Including this module also includes `DSS`, `DPS`, and their modules.
`DCS.AbandonedOrderServices`	Provides tools for dealing with abandoned orders and shopping carts. Including this module also includes `DCS` and its modules.
`DCS.CustomCatalogs`	Runs custom catalogs in an ATG Commerce production environment, required to support pre-ATG 10.0.1 Commerce applications; otherwise unused.
`DCS.DynamicCustomCatalogs`	Runs custom catalogs in an ATG Commerce development environment. Including this module also includes `DCS.CustomCatalogs` and its modules.
`DCS.DynamicCustomCatalogs.Versioned`	Runs custom catalogs in an environment running ATG Commerce and ATG Merchandising. (Requires the `DCS-UI.management` and `PubPortlet` modules also.) Including this module also includes `DCS.DynamicCustomCatalogs`, `DCS.CustomCatalogs.Versioned`, and their modules.
`DCS.PublishingAgent`	Use instead of the `PublishingAgent` module on the target server if ATG Commerce repository items are deployed to that server. Including this module also includes `PublishingAgent`, `DCS`, and their modules.

Module	Description
`DCS.Search`	Enables ATG Commerce to use ATG Search to index and search content from product catalogs and other repositories. See the *ATG Search Administration Guide*.
`DCS.Versioned`	Use instead of `DCS` module if running ATG Commerce with ATG Merchandising. Including this module also includes `Publishing.base`, `DCS`, and their modules.
`DCSSampleCatalog`	ATG Commerce Sample Catalog Including this module also includes `DCS` and its modules.
`Fulfillment`	ATG Commerce order fulfillment Including this module also includes `DCS` and its modules.
`MotorpriseJSP`	ATG Business Commerce reference application (Motorprise) Including this module also includes `B2BCommerce`, `DCS.AbandonedOrderServices`, and their modules.
`PayFlowPro`	Third-party commerce module (from VeriSign) for handling credit card authorization, settlement, and crediting. Including this module also includes `DCS` and its modules.
`Taxware`	Third-party commerce module (from ADP Taxware) for calculating taxes, verifying addresses and determining tax jurisdictions. Including this module also includes `DCS` and its modules.

Appendix E: Request Handling Pipeline Servlets Reference

This section documents the servlets that are provided by the ATG platform modules. Each component description includes its complete component name and class.

AccessControlServlet

Class	`atg.userprofiling.AccessControlServlet`
Component	`/atg/dynamo/servlet/dafpipeline/AccessControlServlet`

`AccessControlServlet` checks the `requestURI` to see if it matches any of the restricted URLs identified in its `accessController` map. The `accessController` map is made up of URLs matched to an `AccessController` instance that governs the rules that determine, when that URL is requested, whether the active `Profile` is permitted to view the page. When access is denied by an `AccessController`, `AccessController` calls `AccessControlServlet`, which redirects the user to the URL in `deniedAccessURL`.

When access is permitted or denied by an `AccessController`, `AccessControlServlet` alerts the registered listeners held in the appropriate property: `accessAllowedListeners` or `accessDeniedListeners`. These properties are populated with the components that register themselves as listeners with `AccessControlServlet`.

You can disable `AccessControlServlet` by setting its `enabled` property to `false`.

For more information on configuring `AccessControlServlet`, see the *ATG Personalization Programming Guide*.

CachePreventionServlet

Class	`atg.servlet.pipeline.CachePreventionServlet`
Component	`/atg/dynamo/servlet/dafpipeline/CachePreventionServlet`

CachePreventionServlet modifies the response headers for certain requests to indicate that the returned content should not be cached.

CheckSessionExpiration

Class	`atg.projects.b2bstore.servlet.WACheckSessionExpiration`
Component	`/atg/dynamo/servlet/dafpipeline/CheckSessionExpiration`

`CheckSessionExpiration` checks the session associated with the request to see whether the it has expired and when it has, what caused the expiration. By examining the `sessionRestored` parameter of the ATG request, `CheckSessionExpiration` can determine if the session ended because of a server failure. When an ATG server fails during a request, a second ATG server creates a new session, retrieves information about the first session from the backup server, and changes the `sessionRestored` parameter from null to the original session ID to indicate that the session `ID` is invalid. `CheckSessionExpiration` reads `sessionRestored` and halts the request, if `sessionRestored` indicates it is been superseded by another request. Otherwise, `CheckSessionExpiration` assumes the session was ended due to an expired cookie; in that case it redirects the user to the URL provided in its `URLExpiration` property.

This servlet is used only by the Motorprise Reference Application. For information on how `CheckSessionExpiration` its use, see the *ATG Business Commerce Reference Application Guide*.

CommerceCommandServlet

Class	`atg.commerce.order.CommerceCommandServlet`
Component	`/atg/dynamo/servlet/dafpipeine/CommerceCommandeServlet`

`CommerceCommandServlet` has an `actionMap` property that matches actions to the servlets that process those actions. When a request includes a `dcs_action` parameter, `CommerceCommandServlet` checks the value of the `dcs_action`, locates the action's corresponding servlet using `actionMap`, and calls that servlet. For example, if a request attempts to add an item to a user's cart by URL, the `dcs_action` is `addItemToCart` and the `AddItemToCartServlet` is called.

For more information, see on this servlet, see the *ATG Commerce Programming Guide*.

CookieBufferServlet

Class	`atg.servlet.http.CookieBuffer`
Component	`/atg/dynamo/servlet/dafpipeline/CookieBufferServlet`

CookieBufferServlet maintains a FIFO queue of cookies in a cookie buffer. When a cookie object is added to the CookieBuffer, it tries to add the cookie to the current HTTP response header. In the next HTTP request, the cookies in the buffer are verified against the cookies returned to the server from the browser. Cookies in the buffer that were properly added to the browser are removed from the buffer. Cookies that were not added to the browser are added back into the HTTP response header.

The API to add cookies to the cookie buffer is found in:

```
atg.servlet.DynamoHttpServletResponse.addCookieToBuffer(Cookie pCookie)
```

You can configure this servlet with the following properties:

Property	**Description**
`maxQueueAttempts`	The number of times the servlet tries to add a queued cookie to the HTTP response header before discarding the cookie.
`requeueCookiePostFailure`	Determines whether a cookie should be re-queued if it was successfully added to the HTTP response header, but not returned in the subsequent HTTP request—that is, the browser disables cookies.
`cookieBufferListeners`	Registers event listeners that implement the interface `atg.servlet.http.CookieBufferListener`.

DAFDropletEventServlet

Class	`atg.droplet.DropletEventServlet`
Component	`/atg/dynamo/servlet/pagecompile/DAFDropletEventServlet`

The `DAFDropletEventServlet` calls the setX/handleX methods of a bean when a form is submitted from a dynamic page or when serving a request from an anchor tag with bean attributes. You can configure how this servlet handles errors encountered in processing a page. By default, errors are set as an element of the `DropletExceptions` set in the request attribute `DropletConstants.DROPLET_EXCEPTIONS_ATTRIBUTE`. Setting the following property returns errors to the requesting page:

```
reportDropletExceptions=true
```

Preventing Cross-Site Scripting Attacks

Cross-site scripting attacks take advantage of a vulnerability that makes it possible for a malicious site you access to use your browser to submit form requests to another site (such as an ATG-based site). To prevent processing of these requests, the ATG platform can use a request parameter `_dynSessConf`, containing a session-confirmation number, to verify that a request is legitimate. This randomly generated long number is associated with the session of the submitted form. On submission of a form or activation of a property-setting `dsp:a` tag, `DAFDropletEventServlet` checks the value of `_dynSessConf` against the current session's confirmation number. If it detects a mismatch or missing number, it can block form processing and return an error.

You can configure this behavior through two properties in the component `/atg/dynamo/Configuration`:

- `enforceSessionConfirmation` specifies whether the request-handling pipeline requires session confirmation in order to process the request; the default value is `true`.
- `warnOnSessionConfirmationFailure` specifies whether to issue a warning on a confirmation number mismatch; the default value is `true`.

You can also control session confirmation for individual requests by setting the attribute `requiresSessionConfirmation` to `true` or `false` on the applicable `dsp:form` or `dsp:a` tag. If this attribute is set to `false`, the `_dynSessConf` parameter is not included in the HTTP request, and the `DAFDropletEventServlet` skips validation of this request's session-confirmation number.

DAFPassportServlet

Class	`atg.userprofiling.sso.DAFPassportServlet`
Component	`/atg/userprofiling/sso/DAFPassportServlet`

`DAFPassportServlet` checks the status of a user's ATG Passport. There are two pieces of a passport, a cookie and a session scoped Passport component, which might or may not be in sync. A Passport can be issued, reissued or a user can be auto authenticated depending on configuration and state.

DynamoHandler

Class	`atg.servlet.pipeline.HeadPipelineServlet`
Component	`/atg/dynamo/servlet/dafpipeline/DynamoHandler`

`DynamoHandler` is always the first servlet in a pipeline. This pipeline servlet takes in an `HttpServletRequest/Response` pair and passes on a `DynamoHttpServletRequest/Response` pair. Putting this servlet at the head of the pipeline ensures that all subsequent pipeline servlets are passed all the functionality of `DynamoHttpServletRequest` and `DynamoHttpServletResponse`.

RequestLocale Object

The DynamoHandler servlet also creates a RequestLocale object in the request. This servlet identifies the locale of the request and sets the locale property of the request's RequestLocale accordingly. This enables you to deliver different content based on the visitor's locale. You can disable the creation of RequestLocale objects by setting the DynamoHandler's generateRequestLocales property to false.

See the Internationalizing an ATG Web Site chapter of this guide for more information.

DynamoServlet

Class	atg.servlet.pipeline.DynamoPipelineServlet
Component	/atg/dynamo/servlet/dafpipeline/DynamoServlet

DynamoServlet sets various properties of the DynamoHttpServletRequest to point to other services in an ATG application. This includes the request scope manager and the MIME typer. These are all services necessary to provide the functionality offered by the DynamoHttpServletRequest.

This servlet is also responsible for determining if a resolved URL should include a context path, using the encodeContextPathModeProperty. This property accepts the following numeric values

- 0 (ENCODE_NONE): The resultant URL does not have a context path.
- 1 (ENCODE_CONTEXT_PATH): The context path defined for the web application should be inserted in the resultant URL. This value is used when none is explicitly provided.
- 2 (ENCODE_IF_NOT_THERE): Causes the ATG platform to check the URL for a context path. If the first entry in the URL is not a context path defined for that web application, the context path is inserted in the final URL.

ExpiredPasswordServlet

Class	atg.userprofiling.ExpiredPasswordServlet
Component	/atg/dynamo/servlet/dafpipeline/ExpiredPasswordServlet

ExpiredPasswordServlet checks a user session for the passwordexpired attribute. If set to true, it redirects the user to a specified URL. This servlet must follow the MimeTyperServlet.

FileFinderServlet

Class	`atg.servlet.pipeline.FileFinderPipelineServlet`
Component	`/atg/dynamo/servlet/dafpipeline/FileFinderServlet`

`FileFinderServlet` finds the disk file associated with a request. More specifically, this servlet sets the `pathTranslated` property of a request by appending the `pathInfo` property to a document root. The document root is specified as the `documentRoot` property of the servlet. This document root is relative to the directory where you run the ATG server, or it can be an absolute pathname. For example:

```
documentRoot=/www/docs
```

If a `pathInfo` specifies a directory rather than a file, this servlet searches for an index file in that directory. If an index file is found, the `pathInfo` and `requestURI` properties are rewritten to use that index file, and the `pathTranslated` property is set to point to that index file. The list of possible index files is specified in the `indexFiles` property.

If no such index file is found, this servlet can then handle the request by listing the files in that directory. This only happens if the `shouldListDirectory` property is `true`; otherwise a file not found error is returned.

If the specified file or index file is found and the `pathTranslated` property is set, the request is passed to the next servlet. Otherwise a file not found error is returned.

One special case comes up when a directory request comes in without a trailing slash character. In this case, the browser must change its request to have a trailing slash in order to handle relative requests properly. The `FileFinder` servlet accomplishes this by issuing a redirect to the browser using the same request URI but including a trailing slash.

alwaysTranslate Property

By default, when you run an ATG server with a commercial web server, such as Apache or Microsoft IIS, the web server translates the `PathInfo` information setting `PathTranslated`. This allows the ATG server to use virtual directories and other special path translation that is normally able to be performed by the web server. In this situation, the `FileFinderServlet` just passes the request on to the next servlet in the pipeline when it sees this request. This requires the web server and the ATG server to have the same file system path structure for documents, even if they are on separate machines.

In this case, the `FileFinderServlet` still translates `pathInfos` that are requested from an ATG server. This includes any calls to the method `DynamoHttpServletRequest.getRealPath("/pathinfo")`.

Using the web server's path translation requires that the ATG server and web server can see documents served with the same absolute pathname. For example, if the web server serves files from `/sun/webserver6.1/docs`, the ATG server must also see files in `/sun/webserver6.1/docs`.

You can change this behavior by setting the `alwaysTranslate` property to `true` in the `FileFinderServlet`. In this case, the web server is still responsible for determining whether requests are sent to the ATG server or not, but the application always performs path translation, overriding the translation performed by the web server. This allows the document roots to be located at different absolute paths. Setting `alwaysTranslate="true"` can also improve security by preventing the ATG server from serving any file outside of the document root. This can have a security benefit, as it can block attempts to have ATG server serve files that are not meant to be served to users.

Translation of Index or Default Files

When the web server sees a request for a directory, it typically translates this `pathInfo` into a request for one of a list of files such as `index.html`, `index.jsp` and so on. When the web server performs path translation, it must be able to see the index file in its document root. By default, the `FileFinderServlet` only performs index files translation if it is performing `pathTranslation` as well. You can set the `processIndexFiles` property to `true` to have it try to expand index files even if `PathTranslated` is already set. This typically is only necessary for web servers that turn `/foo/` into `/docs/foo/` in cases where the path translation should be `/foo/` to `/docs/foo/index.html`.

Virtual File Translation

When the `FileFinderServlet` is performing path translation, it can also translate a list of virtual directories. A virtual directory lets you map all `pathInfos` that start with a particular path prefix to a separate directory on your file system. The virtual directories are specified in `FileFinderServlet` by the `virtualDirectoryMap` property. This property contains a list of mappings of the form: `/pathInfoPrefix=/fileSystemPath`. For example:

```
virtualDirectoryMap=/myVirtURL1=c:/myVirtRoot1,/myVirtURL2=c:/myVirtRoot2
```

You should set the virtual directories if you set `always Translate="true"` or if you translate paths via the `request.getRealPath()` method.

LocaleServlet

Class	`atg.epub.servlet.LocaleServlet`
Component	`/atg/dynamo/servlet/dafpipeline/LocaleServlet`

`LocaleServlet` provides a value of `en_US` to the response locale property when none other is provided.

MimeTypeDispatcher

Class	`atg.servlet.pipeline.MimetypeDispatcherPipelineServlet`
Component	`/atg/dynamo/servlet/dafpipeline/MimeTypeDispatcher`

`MimeTypeDispatcher` is an instance of a subclass of `DispatcherPipelineServlet` that sends a request to one of several servlets depending on the MIME type of the request. The `dispatcherServiceMap` property maps each MIME type to the appropriate servlet. As installed, this property has one mapping:

```
dispatcherServiceMap=\
        dynamo-internal/html=/atg/dynamo/servlet/dafpipeline/FileFinderServlet
```

The MIME type is extracted from `MimeTyperPipelineServlet.ATTRIBUTE_NAME`, so `MimeTyperServlet` must precede `MimeTypeDispatcher` in the pipeline. The MIME types specified in the `dispatcherServiceMap` property must already be mapped to filename extensions (see Adding MIME Types). The servlets specified by `dispatcherServiceMap` must have global scope.

MimeTyperServlet

Class	`atg.servlet.pipeline.MimeTyperPipelineServlet`
Component	`/atg/dynamo/servlet/dafpipeline/MimeTyperServlet`

The `MimeTyperServlet` examines a request's `pathTranslated` property to determine its MIME type. The MIME type is then added as an attribute of the request called `MimeTyperPipelineServlet.ATTRIBUTE_NAME`.

The servlet property `mimeTyper` must be configured. This property points to another service, usually an `atg.servlet.ExtensionMimeTyper`. This MIME typer service contains a list of extensions and corresponding MIME types.

The servlet adds the MIME type using an attribute factory, so that the MIME type is not actually calculated until the first time it is needed.

Forbidden Mime Type

All file types unknown to an ATG server default to the `forbidden` MIME type. The forbidden MIME type designates file types that might compromise a web site's security if the ATG server served them back to a client, such as `.log`, `.properties`, or `.ini` files. Dynamo rejects all requests for files of these MIME types and passes them to `SendErrorServlet`, which returns a 404 error. Therefore, you must configure any MIME types that you want served in addition to those already configured in the `MimeTyperServlet`.

Adding MIME Types

In order to specify handling of additional MIME types, reconfigure the component `/atg/dynamo/servlet/pipeline/MimeTyper` (class `atg.servlet.ExtensionMimeTyper`), which is referenced by the MimeTyperServlet's `mimeTyper` property. The MimeTyper's `extensionToMimeType` property is a string array that pairs filename extensions to MIME types. For example:

```
shtml,magnus-internal/parsed-html,\
cgi,magnus-internal/cgi,\
jsp,dynamo-internal/html,\
```

After you add the desired MIME types, configure the `MimeTypeDispatcher` to specify how the servlet pipeline handles requests of those MIME types

PageViewServletTrigger

Class	atg.userprofiling.PageEventTriggerPipelineServlet
Component	/atg/dynamo/servlet/dafpipeline/PageViewServletTrigger

When a page is requested, `PageViewServletTrigger` fires a `PageEventTrigger`, by passing `PageViewedEvent` in the Dynamo request. In addition, `PageEventTrigger`checks the request for a `dsource` parameter and, when found, `PageEventTrigger` fires a `ClickThroughEvent`.

For more information, see the *ATG Personalization Programming Guide*.

PathAuthenticationServlet

Class	`atg.servlet.pipeline.PathAuthenticationPipelineServlet`
Component	`/atg/dynamo/servlet/dafpipeline/PathAuthenticationServlet`

`PathAuthenticationServlet` provides username and password authentication. You can associate one or more usernames and passwords with any URL request prefix. The servlet requires authentication in the form of a valid username/password pair before it allows service of a URL that begins with that prefix.

The `PathAuthenticationServlet` has the following properties:

Property	**Description**
`Realm`	The realm to use in authentication. Defaults to `Dynamo`.
`Enabled`	Is authentication enabled? Defaults to `false`.
`authenticators`	A `ServiceMap` that maps path prefixes to components that implement the `Authenticator` interface, which checks whether a username/password pair is valid.

By default, `PathAuthenticationServlet` appears in the DAF servlet pipeline between the `ThreadUserBinderServlet` and the `DynamoServlet`, but `PathAuthenticationServlet` is not enabled. You can enable `PathAuthenticationServlet` by setting the `enabled` property to `true`.

`PathAuthenticationServlet` (if enabled) searches all the keys in the `authenticators` map to see if the requested URL starts with any of the path prefixes listed there. The servlet uses the longest path prefix that matches and the corresponding authenticator object is used to authenticate the request.

Example

The following example assumes your HTTP server has a document root of `/docs`. You can enable password authentication for directories called `docs/truth` and `docs/truth/inside_truth` with the following properties settings in the `PathAuthenticationServlet`:

```
enabled=true
authenticators=\
     /truth=/application/auth/TruthPassword,\
     /truth/inside_truth=/application/auth/Inside_TruthPassword
```

Note: The paths exclude the `/docs` prefix; these paths are relative to the `docroot` of the HTTP server.

An authenticator component includes a `passwords` property. The value of the `passwords` property is a list of valid username/password pairs. Thus, the `TruthPassword.properties` file might read:

```
$class=atg.servlet.pipeline.BasicAuthenticator
passwords=\
     satchmo=cornet
```

In this example, if a user requests any document in the `/docs/truth` area, the user is required to provide the username `satchmo` and the password `cornet`. You can create a separate authenticator component at `/application/auth/Inside_TruthPassword` to require a different username/password pair in order to request documents from the `/docs/truth/inside_truth` area.

ProfilePropertyServlet

Class	`atg.userprofiling.ProfilePropertyServlet`
Component	`/atg/dynamo/servlet/dafpipeline/ProfilePropertyServlet`

`ProfilePropertyServlet` sets properties on the profile by calling out to setters that implement the ProfilePropertySetter interface.

ProfileRequestServlet

Class	`atg.userprofiling.ProfileRequestServlet`
Component	`/atg/dynamo/servlet/dafpipeline/ProfileRequestServlet`

`ProfileRequestServlet` manages `Profile` information for the current session. If a `Profile` does not already exist for the active session, `ProfileRequestServlet` creates an instance of the atg/userprofiling/Profile component. While creating the Profile, ProfileRequestServlet creates a transient RepositoryItem and sets it to the Profile component dataSource property.

When a session begins, ProfieRequestServlet prompts the CookieManager to create a cookie containing the Profile ID of the current guest user. When a user logs in or registers, a second cookie created by the ProfileFormHandler with a different Profile ID representing the logged in user overwrites the first. If you'd prefer not to user cookies on your site, you can maintain user session data though an authentication process. To do so, enable Basic Authentication (verifyBasicAuthentication property). For more information on Basic Authentication, see Authentication.

You can set ProfileRequestServlet to maintain persistent information about guest visitors by setting persistentAnonymousProfiles to true. By doing so, you instruct the ATG server to create a persistent RepositoryItem for each new session. There might be circumstances where you do not want this to occur. For example, the session ends when a user logs out, and a new session begins where the user is recognized as a guest. Creating a persistent RepositoryItem might be unnecessary in this case. By default, a temporary RepositoryItem is created in this scenario, because persistAfterLogout is set to false.

ProfileRequestServlet can fire a login event when a user auto-logs in. It can also fire a login event when a persistent anonymous profile is created.

For a complete discussion on Profiles, cookies, the ProfileFormHandler and more detail on the properties mentioned here, see the *ATG Personalization Programming Guide*.

ProjectServlet

Class	atg.epub.servlet.ProjectServlet
Component	/atg/dynamo/servlet/dafpipeline/ProjectServlet

ProjectServlet invokes VersioningLayerServlet, which looks for a project parameter and pushes the project's workspace.

PromotionServlet

Class	atg.commerce.promotion.PromotionServlet
Component	/atg/dynamo/servlet/dafpipeline/PromotionServlet

When PromotionServlet is enabled (enabled property set to true), PromotionServlet scans the requestURI for the PROMO parameter, and when it is present, matches the promotion ID associated to it against the promotion IDs in the promotionItemDescriptorNames property to ensure that the promotion is active. When a match is found, PromotionServlet checks to see if the user qualifies for the promotion by examining the Profile RepositoryItem for persistency and the Promotion giveToAnonmousProfiles property for a value of true. If either condition is met, PromotionServlet adds the promotion ID to the Profile activePromotions property.

ProtocolSwitchServlet

Class	atg.projects.store.servlet.pipeline.ProtocolSwitchServlet
Component	/atg/dynamo/servlet/dafpipeline/ProtocolSwitchServlet/

ProtocolSwitchServlet performs switching between a secure sever and a non-secure server. A list of secure paths and the enable property controls the switching. The servlet is configured with a list of URL mappings; if the URL to access is in the URL mapping, the request is passed off to the secure server. By default, the nonSecureHostName and secureHostname are taken from /atg/dynamo/Configuration. These can be overridden at the component level.

PublishingActionServlet

Class	atg.epub.servlet.PublishingActionServlet
Component	/atg/dynamo/servlet/dafpipeline/PublishingActionServlet

PublishingActionServlet examines the requestURI for the action parameter and when located, PublishingActionServlet passes the located actionId value to the framework component, which executes the action.

PublishingSecurityServlet

Class	atg.epub.servlet.PublishingSecurityServlet
Component	/atg/dynamo/servlet/dafpipeline/PublishingSecurityServlet

PublishingSecurityServlet enables security checking for request threads. It enables it only for requests from logged-in users.

SessionEventTrigger

Class	`atg.userprofiling.SessionEventTrigger`
Component	`/atg/dynamo/servlet/dafpipeline/SessionEventTrigger`

When `SessionEventTrigger` receives a request, it determines whether the session is new. For new sessions, `SessionEventTrigger` fires a `StartSession` event.

`SessionEventTrigger` registers itself as a listener in the `SessionManager` `nameContextBindingListeners` property so it is among the list of listeners alerted when the session expires. After `SessionEventTrigger` detects an expired session, it fires a `EndSession` event.

`SessionEventTrigger` is also responsible for firing `ReferrerEvents`. `SessionEventTrigger` checks the request for the `referer` parameter that is set by the browser when a user clicks a link. The `referer` is set to a URL for the page where the request is initiated: it might be set to a relative path, a portion of the URL or the URL in its entirety. When `referer` is populated with a non-null value, `SessionEventTrigger` fires an event.

For information on `StartSession`, `EndSession`, and `ReferrerEvents`, see the *ATG Personalization Programming Guide*.

SessionSaverServlet

Class	`atg.servlet.sessionsaver.SessionSaverServlet`
Component	`/atg/dynamo/servlet/dafpipeline/SessionSaverServlet`

`SessionSaverServlet` is a part of ATG's session failover architecture. This servlet identifies whether a session has a non-local origin. If the session is non-local, `SessionSaverServlet` restores the session from the backup server. `SessionSaverServlet` also backs up the session's properties to the backup server.

SiteSessionEventTrigger

Class	`atg.multisite.SiteSessionEventTriggerPipelineServlet`
Component	`/atg/dynamo/servlet/dafpipeline/SiteSessionEventTrigger`

`SiteSessionEventTriggerPipelineServlet` checks the current SiteContext to determine whether the current request is the start of a new site session. If it is, it sends an event

SetCurrentLocation

Class	`atg.projects.b2bstore.servlet.WASetCurrentLocation`
Component	`/atg/dynamo/servlet/dafpipeline/SetCurrentLocation`

A web site is divided into sections that share subject matter or function, such as My Profile, Product Catalog, and Administration. `SetCurrentLocation` has a `locationMap` property that matches a directory of pages to the section name that identifies them. For example, one entry might be:

```
/MotorpriseJSP/en/user/=my_account
```

`SetCurrentLocation` examines the `requestURI` and uses `locationMap` to find the matching section name and section root directory. Then, `SetCurrentlocation` saves the section name to its `location` property and the section root directory to the `Profile` component `currentLocation` property.

This servlet is used only by the Motorprise Reference Application. For more information on how `SetCurrentLocation` is used in Motorprise, see the *ATG Business Commerce Reference Application Guide*.

SiteContextPipelineServlet

Class	`atg.multsite.SiteContextPipelineServlet`
Component	`/atg/multisite/SiteContextPipelineServlet/`

When an ATG server receives a request from a given site in a multisite environment, SiteContextPipelineServlet evaluates the request to determine the site's identity. That identity enables delivery of site-specific information in the ATG server response.

For detailed information, see the Multisite Request Processing chapter.

TailPipelineServlet

Class	`atg.servlet.pipeline.TailPipelineServlet`
Component	`/atg/dynamo/servlet/dafpipeline/TailPipelineServlet`

The request is passed to `TailPipelineServlet` when the DAF servlet pipeline has completed processing. `TailPipelineServlet` calls `FilterChain.doFilter()` on `PageFilter` to create `FilterChain` object, which invokes the servlet filter identified in `web.xml`. When no other filters are found, as is with the default DAF servlet pipeline provided with ATG Adaptive Scenario Engine, `PageFilter` passes the ATG request and response back to the application server.

ThreadNamingPipelineServlet

Class	`atg.servlet.pipeline.ThreadNamingPipelineServlet`
Component	`/atg/dynamo/servlet/dafpipeline/ThreadNamingPipelineServlet`

`ThreadNamingPipelineServlet` modifies a request's thread name by appending session- and user-specific information. This servlet can be useful in troubleshooting hanging threads for a given site, as the appended data can help identify the source of the problem—for example, hanging threads that all share the same remote user IP address.

As installed, the servlet class appends the following data to the original thread name:

- Request URI
- Session ID
- Remote user IP address
- User profile ID

For example, given the following JBoss HTTP handler thread name:

```
http-0.0.0.0-8180-2
```

`ThreadNamingPipelineServlet` might modify the thread name as follows:

```
http-0.0.0.0-8180-2 requestURI=/PioneerCycling/example.jsp
jsessionid=4CDA4BC58E0F38F52AA7F87E06446888.drp1 remoteAddr=127.0.0.1
userid=1240001
```

Servlet Properties

The servlet component should be configured as follows:

```
$class=atg.servlet.pipeline.ThreadNameServlet
insertAfterServlet=ProfileRequestServlet

profilePath=/atg/userprofiling/Profile
```

Inserting in the Request Handling Pipeline

To insert this servlet in the pipeline at server startup, set `/atg/dynamo/servlet/Initial.properties` as follows:

```
$class=atg.nucleus.InitialService

initialServices+=dafpipeline/ThreadNameServlet
```

ThreadUserBinderServlet

Class	`atg.servlet.security.ThreadUserBinderServlet`
Component	`/atg/dynamo/servlet/dafpipeline/ThreadUserBinderServlet`

`ThreadUserBinderServlet` takes the `atg/dynamo/security/User` component that was previously associated to the request by `SessionServlet` and associates it to the request thread itself. You can find which `User` component the `ThreadUserBinderServlet` component uses in the `userComponentName` property. If a `User` component does not exist in the request, `ThreadUserBinder` creates one.

This servlet makes user information available to other security-related portions of the ATG platform so be sure to insert `ThreadUserBinderServlet` before any security components that expect user information.

TransactionServlet

Class	`atg.dtm.TransactionPipelineServlet`
Component	`/atg/dynamo/servlet/dafpipeline/TransactionServlet`

The `TransactionServlet` can be configured to cause each request to be wrapped in a transaction. By default, the `TransactionServlet`'s `transAttribute` property is set to `supports`, which means a new transaction is not created for each request. In most circumstances, the `transAttribute` property should remain set to `supports`. In all cases, the `TransactionServlet` checks to see if the request created a Transaction without ending it. If so, the servlet rolls back that Transaction and reports an error.

URLArgumentServlet

Class	`atg.servlet.pipeline.URLArgumentPipelineServlet`
Component	`/atg/dynamo/servlet/dafpipeline/URLArgumentServlet`

A URL can contain one or more arguments that are appended to the URL path and precede any query arguments. Each argument is paired with a value and starts with a semicolon:

`.../mypage;`*`arg1`*`=`*`val1`*`;`*`arg2`*`=`*`val2`*`...`

URLArgumentServlet extracts these arguments from the URL and places them in a `java.util.Dictionary` that maps argument names to values. The URL is rewritten without the arguments, and `requestURI` is modified as follows:

```
requestURI=servletPath+pathInfo+'?'+queryString
```

ValidateURLServlet

Class	`atg.epub.servlet.ValidateURLServlet`
Component	`/atg/dynamo/servlet/dafpipeline/ValidateURLServlet`

`ValidateURLServlet` is a security precaution that prevents spoofing of URLs. When a user initiates an action, an `action` parameter holding an ID for the action is added to the URL. An encryption key based on the `action` parameter value is also added to the URL. Likewise, when a user selects a view, a `view` parameter and corresponding encryption key are appended to the URL. When both `action` and `view` parameters are added to the URL, the encryption key represents the combination of the parameter values.

`ValidateURLServlet` recalculates the encryption key in the URL based on the `action` or `view` parameter values and compares it to the encryption key already in the URL. For URLs with the appropriate key, `ValidateURLServlet` adds an attribute to the request, which permits ATG Content Administration to display the request URL. URLs that lack the expected key do not include the request attribute and as a result, cause errors when rendering the request URL.

The best way to disable `ValidateURLServlet` is to configure ATG Content Administration to display the request URL regardless of whether the request includes the attribute. To do this, set the `validateActions` and `validateViews` properties of `<ATG10dir>\Publishing\base\config\atg\epub\pws\framework\Framework.properties` to `false`.

VersioningLayerServlet

Class	`atg.pub.servlet.VersioningLayerServlet`
Component	`/atg/dynamo/servlet/dafpipeline/VersioningLayerServlet`

`VersioningLayerServlet` checks the `requestURI` to see if it contains a project ID in the `projectId` parameter and if it does, `VersioningLayerServlet` retrieves that project's workspace and makes that workspace available to the user.

Index