ATG WEB COMMERCE

Version 10.0.2

Personalization Programming Guide

Oracle ATG
One Main Street
Cambridge, MA 02142
USA

ATG Personalization Programming Guide

Document Version

Doc10.0.2 PERSPROGGUIDEv1 04/15/2011

Copyright

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/us/corporate/accessibility/index.html.

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/support/contact.html or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

Contents

Part I: Personalization Module Programming

The Personalization module is included with the ATG platform and adds visitor profiling, content management, and targeting functionality to the ATG development framework. These features make it possible for developers and business managers to customize the behavior, content, and presentation of an ATG application to match each visitor's characteristics and preferences.

Most Personalization module features are available through the ATG Control Center (ACC). For more information, see the *ATG Personalization Guide for Business Users*. If your system includes ATG Content Administration, you can access these features through the ATG Business Control Center, which is described in the *ATG Business Control Center User's Guide*. The chapters in this manual (listed below) explain how to access and extend these features programmatically.

Personalization Module Tracking
Explains how to capture information about Personalization module events and use that information to perform actions such as updating visitor profiles or changing component properties.

Personalization Module Logging
Describes how to use Personalization module events (page requests, content viewed, and user events) to record useful information about the operation of your Web application.

1 Setting Up a Profile Repository

This chapter includes the following topics:

Introduction to Profile Repositories

A user profile is a set of attributes that represent the data you want to store for a site user, for example login name, password, e-mail address, registration date, and so on. ATG applications use profile repositories to manage user profiles. A standard profile repository is a SQL repository component of class `atg.adapter.gsa.GSARepository`. Each user profile is represented by an item in the profile repository, and the attributes that make up the profile are stored as properties of the repository item. When you design your sites, you need to determine which profile attributes you want to track for your users and then set up a database to store the profiles.

As a SQL profile repository is an instance of the generic SQL repository, the process of setting up a SQL profile repository is similar to the process described in the *SQL Repositories* chapter of the *ATG Repository Guide*. It requires you to perform the following basic steps:

1. Design your profile repository definition, determining the properties you want to have available in your user profiles. Most profile repositories will use a single profile item type (typically called `user`) for all users within that repository. See the Defining the Profile Repository section in this chapter.
2. Create the SQL database schema on the SQL database server that corresponds to the profile repository definition you designed.
3. Make any needed configuration changes to the appropriate profile repository component. See the Configuring a Profile Repository Component section in this chapter.

Although most profile repositories are SQL repositories, it is also possible to store profile information in an LDAP directory or a composite SQL/LDAP repository. This chapter focuses on using a SQL repository as the data store. For information on using an LDAP directory as the data store, see the Setting Up an LDAP Profile Repository chapter. For information on using a composite SQL/LDAP profile repository, see Setting Up a Composite Profile Repository.

Internal and External User Profiles

You can maintain separate profile repositories for external and internal users, and the ATG platform and most ATG applications are configured by default to do so. External user profiles represent anyone who visits your externally-facing Web sites. For commerce sites, external users are typically customers. Internal user profiles represent people within your organization who use ATG applications such as the ATG Business Control Center or ATG Service to create and manage site content. Maintaining distinct profiles for internal and external users has a number of benefits:

- You can store information about your internal users in a separate data source from the profiles used with your outward-facing Web applications.
- You can authenticate internal and external users separately, which helps eliminate the possibility of an external user gaining access to an internal application.
- You can create different sets of targeters and scenarios for external and internal users. (This feature is typically used by ATG Service applications, for example to display content to customer service representatives.)

The default external user profile repository is `/atg/userprofiling/ProfileAdapterRepository`, which is defined by the `userProfile.xml` file located in `<ATG10dir>\DPS\config\profile.jar`. Each ATG application that adds properties to the external user profile stores its `userProfile.xml` file in an `ExternalUsers` sub-module.

Internal profiles are stored in the `/atg/userprofiling/InternalProfileRepository`, defined by the `internalUserProfile.xml` file in `<ATG10dir>\DPS\InternalUsers\config\config.jar`.

For information on the default internal/external profile repository model, including information on how the repositories are used by the ATG Business Control Center and other ATG applications, refer to the *ATG Business Control Center Administration and Development Guide*.

Internal Profile Repository

This chapter describes how to set up and configure the `ProfileAdapterRepository`. However, the `InternalProfileRepository` is also an instance of `atg.adapter.gsa.GSARepository`, and you can extend and configure it using the methods described in this chapter for the `ProfileAdapterRepository`.

Many of the services provided for the `ProfileAdapterRepository` that are described in this manual also exist for the `InternalProfileRepository`. For example, the ATG Personalization module includes an `/atg/userProfiling/InternalProfileTools` component, which is implementation of class `atg.userprofiling.ProfileTools` configured for the `InternalProfileRepository`.

A parallel set of database tables also exists for internal user profiles. Where the `user` item in the `ProfileAdapterRepository` references the `dps_user` table, the `user` item in the `InternalProfileRepository` points to a `dpi_user` table, and so on.

In most cases, external user profiles are created automatically when a user visits one of your Web sites. However, internal user profiles are typically not created in this way. After you have defined the properties that make up the internal profile repository, you must create individual profile items for each user in your system. The recommended interface for doing so is the ATG Business Control Center. See the next section, Profile Repository Administration Interfaces.

Note that the ATG Business Control Center is configured by default to accept logins from profiles in the internal user repository.

Profiles in a Multisite Environment

A multisite ATG environment is one in which a single instance of ATG products supports more than one Web site, and the sites are configured to share resources such as a shopping cart. As an example, an apparel manufacturer could have two brands, one for clothing and one for shoes. Each brand could have its own Web site, but the shopping cart could be shared between the two sites, allowing cross-selling opportunities such as the ability to give promotions for one site based on items purchased at the other.

In a multisite environment, user profiles are automatically shared across all your Web sites. The same external and internal profile repositories are used in all cases; there is no option to set up separate profile repositories for each site you support.

It is important to be aware of the implications of this behavior in regard to registration and login. When a user registers at any one of your sites, a persistent profile is created for him or her and added to the external profile repository. However, because the repository is not site aware, the registration is not specific to Site A. Users who register at Site A are effectively registered at Site B as well. If you require login credentials to access your sites, the user is automatically permitted to log into both Site A and Site B and can do so using the same username and password. A user who logs into Site A simultaneously logs into Site B.

Mechanisms such as targeting rules and scenarios that manage the display of dynamic, personalized content can be configured to be site aware, which means you can display specific content to individual users according to their current site. For example, you could set up a scenario that waits for users to register on Site A and displays them personalized content. However, as stated above, the profiles themselves are the same for all sites.

For information on configuring your environment for multisite support, refer to the *ATG Multisite Administration Guide*. For more information on the multisite features available in targeting rules, scenarios, and other personalization mechanisms, see the *ATG Personalization Guide for Business Users* and the chapters for those features later in this guide.

Profile Repository Administration Interfaces

The Personalization module includes three interfaces you can use to view or edit profile repository items:

- The ATG Dynamo Server Admin interface
- The Personalization > Users options in the ATG Business Control Center
- The Profile Repository window in the ACC

Profiles in the ATG Dynamo Server Admin Interface

You can view the `ProfileAdapterRepository` component in the ATG Dynamo Server Admin interface at `http://hostname:port//dyn/admin/nucleus/atg/userprofiling/ProfileAdapterRepository`. The default port numbers on JBoss, Oracle WebLogic, and IBM WebSphere are 8080, 7001, and 9080, respectively. You can view the `InternalProfileRepository` at `http://hostname:port/dyn/admin/nucleus/atg/userprofiling/InternalProfileRepository`. For more information, see the *ATG Installation and Configuration Guide*.

A profile repository's interface page, like the other component pages in the Admin interface, displays the properties of the profile repository component. The interface page also includes links for each of the item descriptors in the repository that let you list all repository items belonging to that item descriptor or examine the item descriptor, displaying the name, short description, and property type of each property of the item descriptor.

In addition, the page displays the full definition file for the repository, including the results of all XML file combination operations. The definition file is displayed in the value of the `definitionFiles` property in the ATG Dynamo Server Admin:

```
http://hostname:port/dyn/admin/nucleus/atg/userprofiling/
ProfileAdapterRepository/?propertyName=definitionFiles
```

or

```
http://hostname:port/dyn/admin/nucleus/atg/userprofiling/
InternalProfileRepository/?propertyName=definitionFiles
```

You can use this view of a definition file for debugging XML file combination problems.

Profiles in the ATG Business Control Center

After you have defined the properties that make up your profile repository, you can use the Personalization > Users options in the ATG Business Control Center to manage the profiles. For information, refer to the *ATG Business Control Center Administration and Development Guide*.

The ATG Business Control Center is the recommended interface for managing profiles and is intended to replace the ACC for this task (see the next section).

Profiles in the ACC

You can also create and edit profile repository items through the People and Organizations > Profile Repository window in the ACC. The query editor along the top of the screen lets you search for items in the repository. You construct your queries by clicking through a series of dropdown attribute choices. For example, you can create a query like this:

```
Items of type user whose locale is fr_FR
```

When you click the List button, the repository items (if any) that match your query appear in the panel on the left side of the screen. You can add and delete repository items in this panel as well. When you select an item from the list, a table of attribute tags and values appears on the right side of the screen. You can edit attribute values by clicking in the value cells.

Important: By default, the ACC that is installed with the ATG platform is configured to point to the profile repository that contains external (customer) user profiles. To change this behavior so that the ACC points to the internal user profile repository instead, include the `DPS.InternalUsers.ACC` module in your EAR file.

Defining the Profile Repository

A *profile repository definition* is a list of all the profile properties that you want to track for users. Since a SQL profile repository is just a way of using the SQL repository, the profile repository definition is an instance of a SQL repository definition, and each profile property is a property of a repository item in the profile repository. For example, you can define a profile repository to track the *first name*, *last name*, and *address* for site members. The Personalization module uses the profile repository definition to create profiles for all the users of your sites. The Personalization module also gathers profile information and maintains a single profile for a user across multiple site visits.

You can set values for your users' profile properties explicitly, from information the users provide through a registration form, a preferences page, or another personal information source, or implicitly, from sensors triggered by user behavior on your sites. Before you can track user profiles, you must define your profile repository definition both in your profile database and in the profile repository.

The Personalization module includes a standard profile repository definition, located in the configuration path at `/atg/userprofiling/userProfile.xml`. You can use this standard profile repository definition as is if it meets the user profiling requirements of your ATG application. More likely, you will want to either use it as a model, or extend it, as seems appropriate. See the Standard User Profile Repository Definition section of this chapter for a description of this repository definition. See Extending the Standard User Profile Repository Definition for information about how to modify it, and see Replacing the Standard User Profile Repository Definition for information about how to replace the standard repository definition with an entirely new profile repository definition.

Defining Profile Sub-Types

You can define multiple profile types in a single profile repository definition file. Once you have defined a root profile type, such as user or organization, you can then create sub-types that contain all the properties of their super-type, plus any other properties that you define. Each profile sub-type is represented by an item-descriptor in your profile repository definition XML file. **Note:** If you add item-descriptors to the userProfile.xml file, make sure that the user item-descriptor is always the first item-descriptor defined in the file.

Consider the following example. The Quincy Funds demo defines some users as “user of type investor” or “user of type broker,” where “investor” and “broker” are simply values of the “type” profile property. Instead, you could create multiple profile sub-types to define distinct kinds of user; each sub-type can then have a range of different properties that are specific to it and define its characteristics.

The following code sample creates two simple profile types, investor and broker. The investor profile type contains all the properties of its super-type user, as well as the property mothersMaidenName. Perhaps you want to require that investors provide this information and use it to log them onto your sites for added security. The broker profile type also contains all the characteristics of its super-type user, and adds the property commissionPercentage. A user who is a broker is the only kind of user who needs this property, so a separate profile type holds such broker-specific properties.

```
<item-descriptor name="investor" super-type="user" sub-type-value="investor">
  <table name="dss_qf_investor" type="auxiliary" id-column-name="id">
    <property category-resource="categoryQuincyFundsInvestorProperties"
       name="assetValue" data-type="float" default="0.0" column-name="asset_value"
       description-resource="investorPropertiesDescription"
       display-name-resource="assetValue">
      <attribute name="resourceBundle"
           value="atg.projects.dssj2eedemo.UserProfileTemplateResources"/>
    </property>
  </table>
</item-descriptor>

<item-descriptor name="broker" super-type="user" sub-type-value="broker">
  <table name="dss_qf_broker" type="auxiliary" id-column-name="id">
    <property category-resource="categoryQuincyFundsBrokerProperties"
       name="commissionPercentage" data-type="int" default="5"
       column-name="commission_pct"
       description-resource="brokerPropertiesDescription"
       display-name-resource="commissionPercentage">
      <attribute name="resourceBundle"
           value="atg.projects.dssj2eedemo.UserProfileTemplateResources"/>
    </property>
  </table>
</item-descriptor>
```

Note that the ATG Business Control Center and the ACC do not currently support changing a user’s sub-type after the user has been created. For this reason, it is recommended that you set the uiwritable attribute to false for any sub-types you add to your repository definition files. For information on the appropriate syntax, refer to the gsa_1.0.dtd file in <ATG10dir>\DAS\lib\classes.jar.

Defining multiple profile types allows you to write targeting rules or use queries that sort users by their profile type instead of by a property in the user profile that defines them as an investor or a broker. You can also query a parent item-descriptor for properties that are defined in its children item-descriptors. For example, if you defined the children item-descriptors above, you could search for “all User items whose commissionPercentage is greater than 15%” even though the User item-descriptor does not contains the commissionPercentage property.

For more information about using multiple profile types in targeting rules, see the Creating Rules for Targeting Content chapter in this manual.

Profile Repository Caching

A SQL profile repository, like all SQL repositories, can maintain repository items in item caches. For most ATG environments, it is strongly recommended that you do **not** use locked caching for the user item in the profile repository. By default, the user items in both the ProfileAdapterRepository and the InternalProfileRepository are configured to use simple cache mode. Caching is disabled for the Password property in both repositories. For more information on cache modes, see the *SQL Repository Caching* chapter in the *ATG Repository Guide*.

Note: Changes made to an external profile through a production site may not appear immediately in the profile UI in the ATG Business Control Center. For example, a customer could change his address through a Web site form, and the new address would not appear immediately to an internal user viewing the same profile through the ATG Business Control Center. The reverse is also true. The delay occurs because of profile repository item caching – until the item cache expires on the server where the changes were **not** made, the changes do not appear on that server. To minimize this problem, you can set the item-expire-timeout attribute for the user item descriptor to force the cache to expire after a short time. For more information, refer to the *ATG Repository Guide*.

If necessary, you can also use the ATG Dynamo Admin UI to flush the item cache.

Standard User Profile Repository Definition

The Personalization module includes a standard profile repository definition with the configuration path name /atg/userprofiling/userProfile.xml, located in <ATG10dir>/DPS/config/profile.jar. This is the default repository definition used by the ProfileAdapterRepository. You can use it as is, use it as a model, or extend it, as determined by the user profiling needs of your ATG application. You can use the standard user profile repository definition with no changes if the following conditions are true:

- the standard user profile repository definition includes all the profile properties that you need in your application, and
- the database schema defined in the standard user profile repository definition fits your database requirements.

If the standard user profile repository definition includes profile properties you don’t need, no particular harm is done. Your ATG application can ignore the unneeded properties, so long as those profile properties allow null values.

The standard profile repository definition in the Personalization module defines the following item descriptors:

Item Descriptor Name	Description
`user`	represents a single user, typically an external user (a customer or other visitor to your Web sites)
`contactInfo`	the contact information for a user, including name, postal address, phone number and fax number
`mailing`	properties used for contacting a user by e-mail, including properties used by the Targeted E-mail facility
`role`	properties that define the non-assignable roles assigned to a user
`organizationalRole`	properties that define the global roles assigned to a user
`organization`	represents an organization
`genericFolder`	defines the folders in which you can place organizations
`roleFolder`	defines the folders in which you can place global roles

Other ATG modules, including the Quincy Funds demo and ATG Commerce, extend this profile repository definition, adding new item descriptors and properties. Each ATG module that extends the profile repository definition includes its own definition file, each of which is also located in the configuration path at `/atg/userprofiling/userProfile.xml`. The system combines all of these profile repository definition files into a single composite repository definition, using the XML file combination rules described in the *XML File Combination* section of the *Nucleus: Organizing JavaBean Components* chapter in the *ATG Programming Guide*.

For more information on defining organizations and roles, see the Working with the Dynamo User Directory chapter in this manual.

Modifying Standard Profile Properties

This section describes the properties defined in the standard user profile repository definition. If you extend or replace the standard user profile repository definition, it is important to understand many of these standard properties. A profile repository definition can include whatever profile properties you think are useful for your ATG application. The `userProfile.xml` files included in ATG application modules add certain properties that are used in various ATG features or demo applications.

Some features, such as scenarios and the Targeted E-mail facility, expect certain profile properties to exist with certain specified names. If you eliminate the needed profile property, the ATG features that expect the property will not work. In some cases, you can change the profile name, but you must also configure a Nucleus component to register the new name. Sometimes you may have to change the profile name or configure a component to use a different property in several different components. If you don’t have to rename or remove a property from the standard profile definition, it is probably better not to.

If you add item-descriptors to the userProfile.xml file, make sure that the user item-descriptor is always the first item-descriptor defined in the file. Doing so prevents errors from occurring in the ACC and the ATG Business Control Center.

If you do decide to rename or remove properties from your profile repository definition, the names of a number of these properties are registered in the /atg/userprofiling/PropertyManager component. If you rename a property that is registered in the PropertyManager, you must also register the new property name in the PropertyManager. The following section describes the PropertyManager component and lists the properties it contains.

Configuring the Property Manager Component

The PropertyManager component registers the names of a number of properties defined in the userProfile.xml file. If you change a property name in the userProfile.xml file, you must also change its name in the PropertyManager component. To configure property names in the PropertyManager component, do the following:

1. Access the PropertyManager component through the ACC at Pages and Components > by Path > atg/userprofiling/PropertyManager.
2. Change the configured value of the property name that you have modified by clicking on the configured value field and typing the new name in the text field.

Configuring the Profile Tools Component

The ProfileTools component includes a reference to the current profile repository through the profileRepository property. If you change this repository, you should also change this reference. This service provides methods that deal with the session-scoped Profile object and also contains lower-level, repository-specific helper methods. The ProfileTools class defines methods for locating users by login or user ID, creating users, and updating properties of profiles. If you change or remove user profile properties, you should make sure that the other components that refer to these properties are correctly referenced in the ProfileTools component. For more information, see User Profiling Tools in the Working with User Profiles chapter of this manual.

Personalization Module

Even if a property listed here is not required by the Personalization module, it may be required by another ATG server or demo application. Removing or renaming any of these properties from the profile repository definition may break other ATG features.

Property	Description
securityStatus	Used, if present, to set how users have authenticated themselves. This property name is registered in the securityStatusPropertyName property of the /atg/userprofiling/PropertyManager component.

`login`	Required for authentication. You should avoid changing the name of this property since it is required by many other ATG services. This property name is registered in the `loginPropertyName` property of the `/atg/userprofiling/PropertyManager` component.
`password`	Required for password authentication. You should avoid changing the name of this property since it is required by many other ATG services. This property name is registered in the `passwordPropertyName` property of the`/atg/userprofiling/PropertyManager` component.
`userType`	Used for having sub-types of the base `user` item-descriptor.
`locale`	The Quincy Funds demo uses this property to allow customers to select their preferred locale. Not required, but used if present by the `RequestLocale` object in the internationalization features implemented by those demo applications. See the *Internationalizing a Dynamo Web Site* chapter in the *ATG Programming Guide* for more information. This property name is registered in the `localePropertyName` property of the`/atg/userprofiling/PropertyManager` component. If you use a property with a name other than `locale` to represent a user's locale, you must also change the value of the `localePropertyName` property of the`/atg/userprofiling/PropertyManager` component.
`lastActivity`	Set to the time when the person last logged in. This property is set in the `/DSS/TrackActivity` scenario that ships with the Scenarios module. This property name is registered in the `lastActivityPropertyName` of the `/atg/userprofiling/ProfileFindForm` component.
`registrationDate`	Set to the time when the person registered. This property is set in the `/DSS/TrackActivity` scenario that ships with the Scenarios module.
`email`	The person's e-mail address. Used by the targeted e-mail facility. This property name is registered in the `emailAddressPropertyName` property of the`/atg/userprofiling/PropertyManager` component. If you use a property with a name other than `email`to represent a user's e-mail address, you must also change the value of the `emailAddressPropertyName` property of the`/atg/userprofiling/PropertyManager` component.
`emailStatus`	Boolean flag that indicates whether the e-mail address is valid. This property name is registered in the `emailStatusPropertyName` property of the`/atg/userprofiling/PropertyManager` component. If you use a property with a name other than `emailStatus` to represent whether a user's e-mail address is valid, you must also change the value of the `emailStatusPropertyName` property of the`/atg/userprofiling/PropertyManager` component.

`receiveEmail`	Boolean flag to indicate if a user wants to receive e-mail. This property is used in scenarios to filter which users should receive targeted e-mail messages. This property name is registered in the `receiveEmailPropertyName` property of the`/atg/userprofiling/PropertyManager` component. If you use a property with a name other than `receiveEmail`to represent whether a user should receive e-mail, you must also change the value of the `receiveEmailPropertyName` property of the`/atg/userprofiling/PropertyManager` component.

Scenarios Module

Property	Description
`scenarioInstances`	Points to a set of scenario instances currently in effect for the profile. This property name is registered in the `scenarioInstancesPropertyName` of the `/atg/userprofiling/PropertyManager` component.

ACC Sorting Attributes

The Personalization module's base profile repository definition uses two XML attributes that govern how a property value is viewed in the ACC: `category` and `propertySortPriority`. All properties with the same `category` attribute value are listed together. For example, the Login Name and Password properties are listed together under the heading Login. Each `<property>` tag uses the attribute `category="Login"`.

If you examine the Basics category in the ACC user profile display, you will see the properties First name, Middle name, and Last name listed in that order. The order in which these properties appear in the UI depends on the `propertySortPriority` attribute. Properties with the same category attribute are listed in ascending order according to the integer value of their `propertySortPriority` attributes. Properties that do not have a `propertySortPriority` attribute are listed in alphabetical order.

ACC Display Name Attribute

An item descriptor or a property in a SQL repository can have a `display-name-resource` attribute that determines the label used by the ACC for that item type or property. In the base profile repository definition, the user item descriptor has this attribute:

```
display-name-resource="User"
```

This causes the People and Organizations > Profile Repository window of the ACC to list profiles under the type User. You can change the `display-name-resource` attribute in the `UserProfileTemplateResources.properties` file in the distribution. The strings in the XML file that appear in the ACC are separated by item-descriptor in this file. For example, if you want to change the `display-name-resource` attribute for the `user` item-descriptor, change the word “User” in the following line after the comment:

```
# item descriptor User
itemDescriptorUser=User
```

Extending the Standard User Profile Repository Definition

This section describes how to add custom item-descriptors to a standard profile repository definition.

Note: If you extend the standard profile definition, make sure that the `user` item-descriptor is always the first item-descriptor in the definition file.

Adding Properties to a Database Table

If you want to add properties to the profile definition file, you need to add a column in the appropriate database table for the item you want to add, and you also need to add a reference to that column in the profile repository definition file. For example, if you wanted to add the property “region” to the `dps_contact_info` table, add a column called `region` to the table, then add the following lines to the Personalization module’s `userProfile.xml` file:

```
<table name="dps_contact_info" type="primary">
  <property name="region" data-type="string" column-name="region"
     required="false"/>
```

If you are using the Personalization module, you don’t need to extend any objects to add a property to a database table. However, if you are using another ATG application, such as ATG Commerce, you may need to extend an object if that object is used by the another repository item. For example, the ATG Commerce order processing system needs to be aware of the `region` property so that it can correctly send an order.

For more information see *Working with Purchase Process Objects* in the *ATG Commerce Programming Guide*.

XML File Combination and the User Profile Repository Definition

If you install the complete ATG platform suite, together with the Quincy Funds demo and ATG Commerce, the standard external user profile repository definition is defined as a composite of the `/atg/userprofiling/userProfile.xml` files in the following JAR files:

```
<ATG10dir>/DPS/config/profile.jar
<ATG10dir>/DSS/config/config.jar
<ATG10dir>/DCS/config/config.jar
<ATG10dir>/B2CCommerce/config/config.jar
```

plus the following XML file:

```
<ATG10dir>/DSSDemo/config/atg/userprofiling/userProfile.xml
```

If you install the B2CCommerce suite, the standard user profile repository definition is defined as a composite of the `/atg/userprofiling/userProfile.xml` files in the following JAR files:

```
<ATG10dir>/DPS/config/profile.jar
<ATG10dir>/DSS/config/config.jar
<ATG10dir>/DCS/config/config.jar
<ATG10dir>/B2BCommerce/config/config.jar
<ATG10dir>/B2BStore/config/config.jar
```

plus the following XML file:

```
<ATG10dir>/DSSDemo/config/atg/userprofiling/userProfile.xml
```

If you install any ATG applications that extend the standard external user profile definition, the definition also includes the `/atg/userprofiling/userProfile.xml` file located in the application's `ExternalUsers` sub-module. This configuration allows all external profile properties to be visible to all internal ATG application users. For more information on these sub-modules, including where to run them, refer to the *ATG Multiple Application Integration Guide*.

The external user profile repository definition is specified by the `definitionFiles` property of the `ProfileAdapterRepository` component. This property names a file in the ATG configuration path. Each of the above-listed user profile repository definition files appears in the configuration path at `/atg/userprofiling/userProfile.xml`. The system uses XML file combination to combine these files into a single definition to use for the profile repository. XML file combination is described in detail in the *XML File Combination* section of the *Nucleus: Organizing JavaBean Components* chapter in the *ATG Programming Guide*. This section describes some factors to remember in modifying the standard user profile repository definition with XML file combination.

When you extend a user profile repository definition, it is important to understand how the Dynamo Application Framework combines XML files. An item descriptor cannot have two properties with the same name. The `xml-combine="replace"` directive works only with property definitions that match exactly, from the outermost tag to the property tag.

Moving Properties to a Different Database Table

Here is an example of what can go wrong if you do not understand how the Dynamo Application Framework combines XML files. Suppose you extend the `userProfile.xml` definition file with your own file in the `localconfig` configuration layer. You want to use the `receiveEmail` property that appears in the standard user profile repository definition, but you want this property to be stored in the `my_user` table, rather than the `dps_user` table. The `receiveEmail` property is defined in the standard profile repository definition like this:

```
<!-- DPS userProfile.xml -->

  <item-descriptor name="user" ...>
    <table name="dps_user" ...>
...
      <property category="Email" name="receiveEmail" data-type="enumerated"
```

```
                default="yes" column-name="receive_email"
                display-name="Receive email">
        <attribute name="useCodeForValue" value="false"/>
        <option value="yes" code="1"/>
        <option value="no" code="0"/>
      </property>
...
    </table>
  </item-descriptor>
```

To replace this property with a similar property that is stored in the `my_user` table rather than the `dps_user` table, you might try to define this property in a `/atg/userprofiling/userProfile.xml` file in your application module or your `localconfig` directory, using the `xml-combine="replace"` directive, like this:

```
<!—Bad Example -->
<!-- my userProfile.xml -->

  <item-descriptor name="user" ...>
    <table name="my_user" ...>
      <property category="Email" name="receiveEmail" data-type="enumerated"
                default="yes" column-name="receive_email"
                display-name="Receive email" xml-combine="replace">
        <attribute name="useCodeForValue" value="false"/>
        <option value="yes" code="1"/>
        <option value="no" code="0"/>
      </property>
    </table>
  </item-descriptor>
```

This approach won't work. In the Personalization module's `userProfile.xml`, the `receiveEmail` property is defined under the `dps_user` table. But in the second `userProfile.xml` file, that property is defined under the `my_user` table. XML combination operates recursively starting from the outermost tags and working inwards. In this case, the `<table>` tags are matched by name. Since the table names `dps_user` and `my_user` don't match, these two `<table>` tags are not combined. And since the `<table>` tags don't match, the tags inside the `<table>` tags will not match up against each other. When the two `userProfile.xml` files are combined, you end up with two properties in the `user` item descriptor, each with the same property name: `receiveEmail`. So you end up with:

```
<!—Bad Example -->
<item-descriptor name="user" ...>
  <table name="dps_user" ...>
    <property name="receiveEmail" ... />
  </table>
  <table name="my_user" ...>
    <property name="receiveEmail" ... />
  </table>
</item-descriptor>
```

Since the resulting user item descriptor has two properties named receiveEmail, errors result.

To fix this is easy. Use the xml-combine="remove" directive in the dps_user table, rather than the xml-combine="replace" directive in the my_user table, as in this example:

```
<item-descriptor name="user" ...>

  <table name="dps_user" ...>
    <property name="receiveEmail" xml-combine="remove" />
  </table>

  <table name="my_user" ...>
    <property category="Email" name="receiveEmail" data-type="enumerated"
              default="yes" column-name="receive_email"
              display-name="Receive email" xml-combine="replace">
       <attribute name="useCodeForValue" value="false"/>
       <option value="yes" code="1"/>
       <option value="no" code="0"/>
    </property>
  </table>
</item-descriptor>
```

The xml-combine="remove" directive ensures that the old receiveEmail property is eliminated from the profile repository definition.

Debugging Repository Definition Files

If you use multiple repository definition files, it can be difficult to understand how the Dynamo Application Framework combines all the files into a single repository definition. You can see the composite repository definition in the ATG Dynamo Server Admin interface in the definitionFiles property of the profile repository component. The display lists the composite XML repository definition, as well as the DTD and the locations of all of the source files that make up the repository definition.

Replacing the Standard User Profile Repository Definition

Instead of modifying the standard user profile repository definition, as described in the Extending the Standard User Profile Repository Definition section, you can replace it completely with a new repository definition that is fitted to the needs of your ATG application. You can choose between two approaches to replacing the standard user profile repository definition:

- Use a file name other than userProfile.xml for the profile repository definition file. See Using a Different Definition File.

- Use userProfile.xml as the file name of the profile repository definition file, eliminating the standard user profile repository definition with XML file combination. See Replacing userProfile.xml.

Be aware that if you decide to use an XML file other than userProfile.xml, some features in ATG Commerce and the Scenarios module will not work unless you include the item-descriptors on which they depend in your own XML file.

Using a Different Definition File

A profile repository component has a property named definitionFiles that points to the profile repository definition file. By default, the /atg/userprofiling/ProfileAdapterRepository component has the following value for this property:

```
definitionFiles=/atg/userprofiling/userProfile.xml
```

You can configure your profile repository to use a different user profile repository definition by changing the value of this property. For instance, if you define your user profile repository definition with a repository definition file named /myModule/userprofiling/userTemplate.xml, you would set the definitionFiles property as follows:

```
definitionFiles=/myModule/userprofiling/userTemplate.xml
```

The definitionFiles property indicates a location in your configuration path.

Be aware that if you don't use the default definition file , some components and services that depend on the item descriptors and properties in this file will not work.

Replacing userProfile.xml

You can replace the standard external user profile repository definition and create an entirely new profile repository definition file . To do this, use XML file combination to remove the existing item types. It is possible to remove or replace any tag within the definition file. If you remove or replace a top level tag such as the <item-descriptor name="user"> tag, you will remove or replace everything within that tag. This means that all the properties that other ATG modules add to the user item descriptor will be lost.

You must decide if you want to replace the entire definition file, or only parts of it. While you can replace the whole file, a better approach is to remove specific tables or properties and to append your own table and property definitions within the file. Once you have decided what parts of the file you want to change, you can place a file containing your changes in your configuration path at /atg/userprofiling/userProfile.xml. This removes all item-descriptors, tables, or properties of a specified name defined in the profile repository by other ATG modules.

For example, if you wanted to modify the userProfile.xml file by adding a property "position" and removing the property middleName, there is a good way and a bad way to accomplish this. If you remove the entire contactInfo item-descriptor or the dps_contact_info table, you'll erase all the other properties belonging to these tags that you may want to keep. Here's a bad example of how to modify the contactInfo information:

```
<!-- DPS userProfile.xml -->
<!—-Bad Example -->

<gsa-template>
  <item-descriptor name="contactInfo" xml-combine="replace">
   <table name="dps_contact_info">
     <property name="firstName">
     <property name="middleName">
     <property name="lastName">
   </table>
</item-descriptor>
</gsa-template>
```

Instead, do the following:

```
<!-- DPS userProfile.xml -->
<!-—Good Example -->

<gsa-template>
  <item-descriptor name="contactInfo">
   <table name="dps_contact_info">
     <property name="firstName">
     <property name="middleName" xml-combine="remove">
     <property name="lastName">
     <property name="position" xml-combine="append">
   </table>
</item-descriptor>
</gsa-template>
```

For more information on XML file combination, see *XML File Combination* in the *Nucleus: Organizing JavaBean Components* chapter of the *ATG Programming Guide*

Configuring a Profile Repository Component

The default profile repository components, `/atg/userprofiling/ProfileAdapterRepository` and `/atg/userprofiling/InternalProfileRepository`, are standard SQL repository components. For information on how to configure them, refer to the *Configuring the SQL Repository Component* section in the *ATG Repository Guide*.

Migrating Profiles for Use with an Internal Profile Repository

The Profile Migration Manager is a utility provided with the ATG platform that you can use to migrate items (and selected properties) from one profile repository to another. Among other uses, it allows you to migrate an environment in which a single `ProfileAdapterRepository` is used for internal and external users to one in which internal profiles are stored separately.

The utility migrates items from a `ProfileAdapterRepository` to an `InternalProfileRepository`. If you are running your application with the `–layer Preview` switch, you can use the `ExternalProfileRepository` instead of the `ProfileAdapterRepository` as the source of the profiles to migrate. (For more information on the Preview layer and the `ExternalProfileRepository`, refer to the *ATG Business Control Center Administration and Development Guide*.)

To use the Profile Migration Manager, complete the following steps:

1. Navigate to the `/atg/userprofiling/ProfileMigrationManager` in the ATG Dynamo Admin Server at the following URL:

   ```
   http://hostname:port/dyn/admin/nucleus/atg/userprofiling/
   ProfileMigrationManager
   ```

2. Use the Properties listing to configure the migration parameters as required. See the descriptions of the available properties below. To change a property value, click the name of the property, select or type the new entry in the New Value field, and then click Change Value.
3. Click Build Plan to display a list of the items that qualify for migration. The list shows the number of qualifying items of each item descriptor and the repository IDs of the items that will be migrated.

   or

   Click Build and Execute Plan to display the list of qualified items and start the migration.

When the migration is complete, the message “INFO Plan execution finished” appears. If you do not see the message, check the logs for error messages.

Note that the profile migration manager does not delete the profiles from the source repository after copying them to the destination. If you want to remove the profiles from the source, you must delete them manually.

Migrated profile items keep the same repository ID. In other words, a user item with ID 10001 in the source repository has the same ID, 10001, in the destination repository.

If any errors occur during the migration, the entire transaction is rolled back, and the destination repository is returned to its previous state.

Profile Migration Manager Properties

You can configure the following properties of the Profile Migration Manager:

- `sourceRepository`

  Specify the path of the `ProfileAdapterRepository` component from which you want to copy items. It resolves by default to `/atg/userprofiling/ProfileAdapterRepository`. If you are running your application with the `–layer Preview` switch, you can also specify an `ExternalProfileRepository` component as the source.

- `destinationRepository`

  Specify the path of the `InternalProfileRepository` component into which the items will be copied. The property is set to `/atg/userprofiling/InternalProfileRepository` by default.

- `itemDescriptorMapping`

  This property lists the item descriptors in the source repository and shows the item descriptors they will be copied to in the destination repository. By default, all item descriptors in the source are mapped to item descriptors with matching names in the destination. To exclude an item from the migration, remove its item descriptor and mapping from this list.

  If a specified item descriptor does not exist in the destination repository, an error is generated when the migration is started.

  The following values are specified by default:

  ```
  user=user
  contactInfo=contactInfo
  role=role
  organizationalRole=organizationalRole
  organization=organization
  genericFolder=genericFolder
  roleFolder=roleFolder
  ```

  When changing these values, use commas to separate item descriptors, as shown:

  ```
  itemdesc1=itemdesc1,itemdesc2=itemdesc2,itemdesc3=itemdesc3
  ```

- `excludedProperties`

  By default, the utility copies all properties of all items specified in the `itemDescriptorMapping` parameter. Use this property to specify any properties that you do not want to copy.

  The following properties are excluded by default:

  `user` item: `userType` and `scenarioValues` properties
  `role` item: `version` property
  `organizationalRole`: `version` property

  Use single commas to separate item descriptors and double commas to separate properties, as shown:

```
itemdesc1=property,,property,,property,itemdesc2=property,,property
```

- `includedItemReferences`

  Profile items typically have references to items in other repositories, for example the Orders repository. This property indicates the references that should be copied. The default configuration excludes the following references:

  `user` item: references to `homeAddress`, `parentOrganization`, `roles`, `ancestors`
  `organizationalRole` item: `relativeTo` reference
  `organization` item: references to `parentOrganization`, `childOrganizations`, `ancestorOrganizations`, `members`
  `genericFolder` item: references to `parent`, `childFolders`
  `roleFolder` item: references to `childItems`, `childFolders`, `parent`

  Use single commas to separate item descriptors and double commas to separate references, as shown:

  ```
  itemdesc1=ref,,ref,,ref,itemdesc2=ref,,ref,itemdesc3=ref,,ref
  ```

- `overwriteDestinationRepository`

  Use to indicate whether to overwrite any items in the destination repository that match items in the source. Items match if they have the same repository ID. Set to false by default. If an item in the source repository has the same repository ID as a profile in the destination, and `overwriteDestinationRepository` is set to false, no migration is attempted for that item. If `overwriteDestinationRepository` is set to true, the source item replaces the destination item.

The default properties file for the `ProfileMigrationManager` component is shown below.

```
# @version $Id: //product/DPS/main/templates/DPS/InternalUsers/
config/atg/userprofiling/ProfileMigrationManager.properties#2
$$Change: 416942 $
$class=atg.repository.migration.RepositoryMigrationManager

sourceRepository=/atg/userprofiling/ProfileAdapterRepository
destinationRepository=/atg/userprofiling/InternalProfileRepository
transactionManager=/atg/dynamo/transaction/TransactionManager

itemDescriptorMapping=\
  user=user,\
  contactInfo=contactInfo,\
  role=role,\
  organizationalRole=organizationalRole,\
  organization=organization,\
  genericFolder=genericFolder,\
  roleFolder=roleFolder
candidateItemQueryMap=\
  user=ALL,\
  contactInfo=ALL,\
  role=ALL,\
```

```
  organizationalRole=ALL,\
  organization=ALL,\
  genericFolder=ALL,\
  roleFolder=ALL
# use double comma (',,') as delimiter for properties
excludedProperties=\
  user=userType,,scenarioValues,\
  role=version,\
  organizationalRole=version
# use double comma (',,') as delimiter for references
includedItemReferences=\
  user=homeAddress,,parentOrganization,,roles,,ancestors,\
  organizationalRole=relativeTo,\
  organization=parentOrganization,,childOrganizations,,
    ancestorOrganizations,,members,\
  genericFolder=parent,,childFolders,\
  roleFolder=childItems,,childFolders,,parent
overwriteDestinationRepository=false
```

2 Setting Up an LDAP Profile Repository

Lightweight Directory Access Protocol (LDAP) directories are widely used to store personnel information and other kinds of data. ATG's LDAP profile repository is an implementation of the Repository API that enables you to store and access profile data in an LDAP directory. The LDAP repository is similar in functionality to the SQL profile repository, as described in the Setting Up a Profile Repository chapter. While by default the Personalization module is configured to use a SQL profile repository, you can change the configuration to use an LDAP repository instead. Using an LDAP repository enables you to tap into the profile data you already have in an LDAP directory, and to share user information across multiple applications.

Just like the SQL profile repository, the LDAP repository implements the ATG repository API to allow you to store, access, modify, and query user profile information. As in the SQL profile repository, repository items are first created as transient items (RAM profiles); they become persistent after they are added to the database. For complete information about LDAP repository concepts, architecture, and code, see the *LDAP Repositories* chapter in the *ATG Repository Guide*.

It is important to note, however, that the LDAP repository implementation is not specific to user profiles in any way. Since an LDAP directory can be used to store any kind of data (people, groups, mailing lists, documents, printers, etc.), you could use the LDAP repository to expose any of that data in an ATG application. For more information, refer to the *LDAP Repositories* chapter in the *ATG Repository Guide*.

Scenarios module and LDAP Repositories: You cannot use scenarios with an LDAP profile repository, because the LDAP repository is not currently powerful enough to express all the data relationships required by the Scenarios module. If you want to run scenarios, you must use either a SQL repository or a composite repository to store all profile information.

This chapter includes the following sections:

Creating the LDAP Profile Repository Component

Configuring the Personalization Module to use the LDAP Repository
Describes how to configure the Personalization module to use an LDAP repository.

Sample LDAP Profile Repository Definition File
An example of an XML file that defines an LDAP profile repository.

Creating the LDAP Profile Repository Component

The LDAP profile repository is a component of class `atg.adapter.ldap.LDAPRepository`. Create and configure an instance of this component as described in the *LDAP Repositories* chapter of the *ATG Repository Guide*.

Configuring the Personalization Module to use the LDAP Repository

By default, the Personalization module is configured to use a SQL database to store profiles. To use an LDAP directory instead, you need to configure the following Personalization module components to work with the LDAP repository.

Property	**Description**

`/atg/userprofiling/ProfileTools`	The `profileRepository` property of the `ProfileTools` component needs to point to an LDAP repository instance, rather than a SQL profile repository. Set the `profileRepository` property to the Nucleus address of your LDAP profile repository component, for example:
	`profileRepository=`
	`/atg/adapter/ldap/LDAPRepository`
	The `defaultProfileType` property of `ProfileTools` needs to specify the name of the item descriptor to which the Profile repository item belongs. By default, this property is set to `user`. If your LDAP repository definition uses a different name for the Profile item descriptor, set the `defaultProfileType` property accordingly.
	Unless the LDAP profile item descriptor contains a `securityStatus` property, you should set the `enableSecurityStatus` property of `ProfileTools` to `false`.

`/atg/userprofiling/PropertyManager`	You may have to modify the `PropertyManager` from its standard configuration to match the LDAP password encryption scheme. See the LDAP Password Encryption topic in this section.

LDAP Password Encryption

The `passwordHasher` property of the `/atg/userprofiling/PropertyManager` component points to a password hasher component that handles password encryption. By default, this property is set as follows:

```
passwordHasher=/atg/dynamo/security/DigestPasswordHasher
```

Change this property to ensure consistency with the LDAP password encryption method you've chosen. For Netscape Directory Servers, set the `passwordHasher` property like this:

```
passwordHasher=/atg/adapter/ldap/NDSPasswordHasher
```

The `NDSPasswordHasher` component supports SHA or no encryption. Set the `encryption` property of the `/atg/adapter/ldap/NDSPasswordHasher` to the appropriate value:

```
encryption=SHA
```

to use SHA password encryption, or

```
encryption=clearText
```

to disable password encryption.

For LDAP servers other than Netscape Directory Server, you may need to create your own `PasswordHasher` implementation, if none of the `PasswordHasher` implementations included in the ATG platform meet your requirements. See the *Password Hashing* section in the *Customizing Application Security* chapter of the *ATG Programming Guide* for more information about ATG's `PasswordHasher` implementations.

See User Profiling Tools in the Working with User Profiles chapter for more information about configuring the `PropertyManager` component.

Sample LDAP Profile Repository Definition File

The following sample LDAP profile repository definition file defines a base item descriptor and view named `user`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE ldap-adapter-template
        PUBLIC "-//Art Technology Group, Inc.//DTD LDAP Adapter//EN"
        "http://www.atg.com/dtds/ldap/ldap_1.0.dtd">

<ldap-adapter-template>

<header>
  <name>ldapUserProfile.xml</name>
  <author>ATG</author>
  <version>$Id$</version>
</header>

<!-- user view -->
<view name="user" default="true">

  <!-- item descriptor -->
  <item-descriptor name="user" display-name="User" display-property="login">

    <!-- special properties -->
    <id-property name="id" in-ldap="false"/>
    <object-classes-property name="objectClasses" ldap-name="objectclass"/>

    <!-- object classes -->
    <object-class>top</object-class>
    <object-class>person</object-class>
    <object-class>organizationalPerson</object-class>
    <object-class>inetorgPerson</object-class>

    <!-- properties -->
    <property name="login" ldap-name="uid" data-type="string" required="true">
      <attribute name="unique" value="true"/>
    </property>
    <property name="password" ldap-name="userpassword" data-type="string"
     required="true"
              editor-class="atg.beans.PasswordPropertyEditor"/>
    <property name="fullName" ldap-name="cn" data-type="string" required="true"/>
    <property name="lastName" ldap-name="sn" data-type="string" required="true"/>
    <property name="firstName" ldap-name="givenName" data-type="string"/>
    <property name="email" ldap-name="mail" data-type="string"/>

    <!-- item creation -->
    <new-items parent-dn="o=example.com" rdn-property="login"/>

  </item-descriptor>

  <!-- search roots -->
  <search-root dn="o=example.com"/>

</view>

</ldap-adapter-template>
```

3 Setting Up a Composite Profile Repository

The composite repository features included with the Personalization module allow you to set up a profile repository that integrates information from various data sources, for example one or more SQL databases and an LDAP directory. The composite repository consolidates these data sources into a flexible, top-level data layer, allowing you to change or make additions to the data underneath without restructuring the entire profile repository. Web applications that use this data model are easier to maintain than others with a more rigid structure, and they can also be written more quickly – for example, you can create a simple Web site for a company that has a tight deadline for launching its online business, and then you can add data sources later without disturbing the data model.

In addition to its flexibility, the composite repository provides the following benefits over using a single-source profile repository:

- Support for scenarios. The Scenarios module requires access to a SQL repository, so as long as your composite repository includes a SQL database as its primary data source, you can run scenarios against it. For more information, see Configuring the Scenarios Module to Use a Composite Profile Repository.
- It allows you to use LDAP directories and other popular forms of data storage while still giving you access to the power and robustness of ATG's Generic SQL Adapter (GSA). For more information on the GSA, see *Introduction to Repositories* in the *ATG Repository Guide*.
- Queryability. A composite repository allows almost unlimited querying of data regardless of the underlying source. For more information, refer to Performing Queries against a Composite Profile Repository.

Even if the site you are building currently uses only one type of data source, you may want to set up and use a composite repository to manage all profile data so that you can easily add data from other sources in the future.

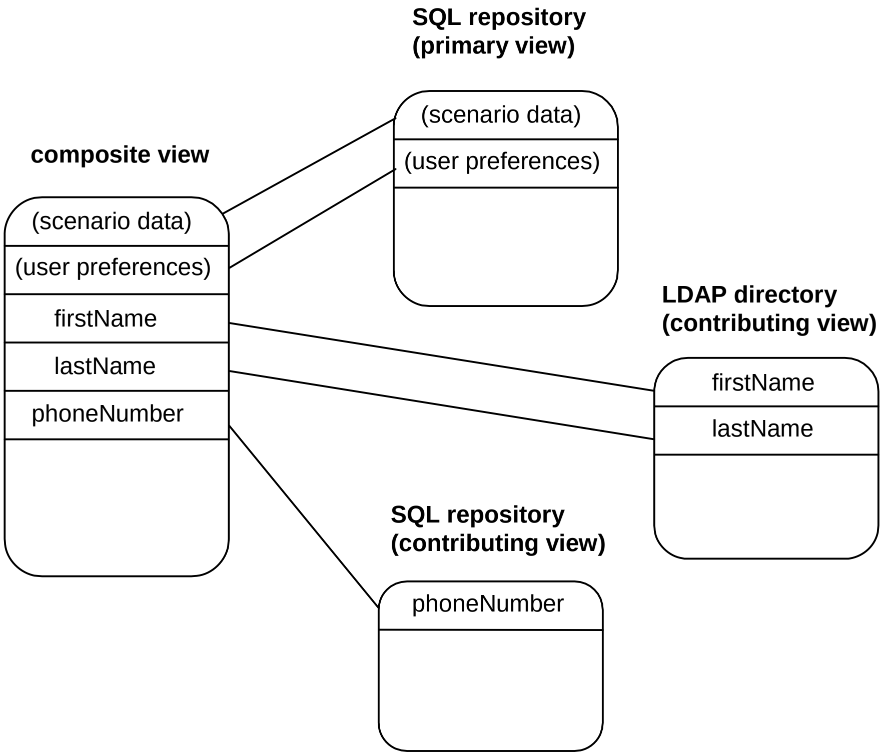
This chapter describes how to set up a composite repository specifically for use as a profile repository. For a more comprehensive discussion of composite repositories, including information on basic concepts and architecture, refer to the *Composite Repositories* chapter in the *ATG Repository Guide*.

Introduction to Composite Profile Repositories

A composite profile repository can have any number of composite views, for example “user,” “broker,” and “investor.” Each composite view is a top layer of data consolidated from any number of contributing views. One contributing view is designated as the “primary view,” which provides the ID space for the item that the views define. (Again, these concepts are explained in more detail in the *Composite Repositories* chapter in the *ATG Repository Guide*.)

The following diagram shows a data model for a composite profile repository that has one composite view and stores profile data as follows:

- Scenario data and user preferences are stored in a SQL repository
- The user’s phone number is stored in a separate SQL repository
- The user’s first name and last name are stored in an LDAP directory

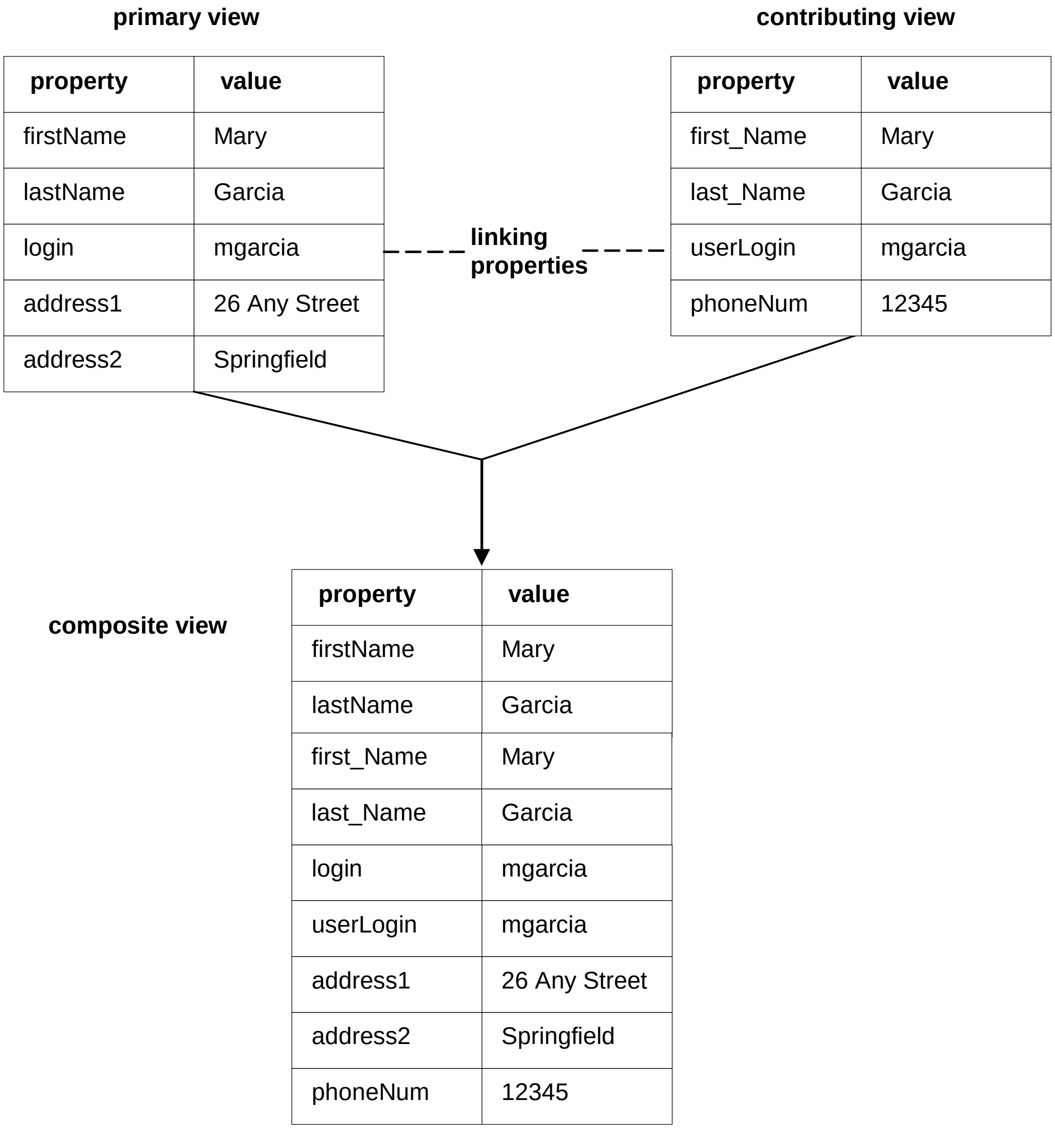


Composite Profile Repository Data Model

Creating Composite Profile Items

As described above, you designate one contributing view as the primary view. Then you specify a method for matching (linking) items in each contributing view to items in the primary view – the two possible methods are linking by repository ID or linking by the value of one or more given properties. (Both techniques are described in the *Composite Repositories* chapter of the *ATG Repository Guide*.) When a `getItem()` or `createItem()` method attempts to retrieve an item from the composite repository, the repository component compares profile items in the contributing and primary views to see if any are linked; if they are, it creates a composite profile item that is an amalgamation of the properties from the contributing and primary views.

Consider the following example: You have a primary view and a contributing view as shown in the diagram below. You choose to link the primary and contributing view by the value of the user's login property (called `login` in the primary view and `userLogin` in the contributing view). If the values in these properties match, a composite profile item is created when an item is retrieved from the composite repository.



primary view

property	value
firstName	Mary
lastName	Garcia
login	mgarcia
address1	26 Any Street
address2	Springfield

contributing view

property	value
first_Name	Mary
last_Name	Garcia
userLogin	mgarcia
phoneNum	12345

composite view

property	value
firstName	Mary
lastName	Garcia
first_Name	Mary
last_Name	Garcia
login	mgarcia
userLogin	mgarcia
address1	26 Any Street
address2	Springfield
phoneNum	12345

The expression you would use in the composite repository definition file to specify how to link the two views is as follows:

```
<primary-item-descriptor-link>
  <link-via-property primary="login" contributing="userLogin"/>
</primary-item-descriptor-link>
```

For more information, refer to the description of the `primary-item-descriptor-link` tag in the *Composite Repositories* chapter of the *ATG Repository Guide*.

Basic Process for Setting Up a Composite Profile Repository

This section provides a basic overview of the steps you follow to set up and configure a composite profile repository. Each step is described in detail in the *Composite Repositories* chapter of the *ATG Repository Guide*.

1. Set up all contributing repositories. Note that the composite repository relies on the data caching mechanism of the underlying repositories; it does not perform any caching itself. As part of this step, therefore, make sure you configure caching for each contributing repository as required.
2. For each composite view, set up contributing views and a primary view.
3. Determine the method for linking the primary and contributing views (by repository ID or by the value of one or more given properties). See Creating Composite Profile Items in this chapter.
4. If you choose the property value method of linking, select one or more uniquely valued properties from each contributing view.
5. Identify any contributing properties that you want to exclude from the composite view. See Resolving Property Names in a Composite Repository in this chapter.
6. Identify any properties you want to rename in the composite view. See Resolving Property Names in a Composite Repository in this chapter.
7. Write the definition file for the composite profile repository. See the example in this chapter and the detailed description in the *ATG Repository Guide*.
8. Configure the Personalization module to use the new repository. See Configuring Your Personalization Module for a Composite Profile Repository in this chapter.
9. If necessary, configure the Scenarios module to use the new repository. See Configuring the Scenarios Module to Use a Composite Profile Repository.

Resolving Property Names in a Composite Repository

The composite repository requires that all property names in a composite view map to only one property from a primary or contributing view. Property names are resolved as follows:

1. If the `all-properties-propagate` attribute in the definition file is set to true for any view, all properties from that view are combined into the composite view, retaining their property names, property types, and any metadata they may have defined.
2. Any properties marked for exclusion are removed from the composite view. See Explicit Property Exclusion.
3. All explicit property mappings are performed. See Explicit Property Mappings.
4. If any remaining property name is the same as any other property name in the composite view, the Personalization module generates an error.

Explicit Property Exclusion

Sometimes you might not want to expose absolutely every property from the primary and contributing views in the composite view. You can use explicit property exclusion to remove from the composite view any properties that you do not require.

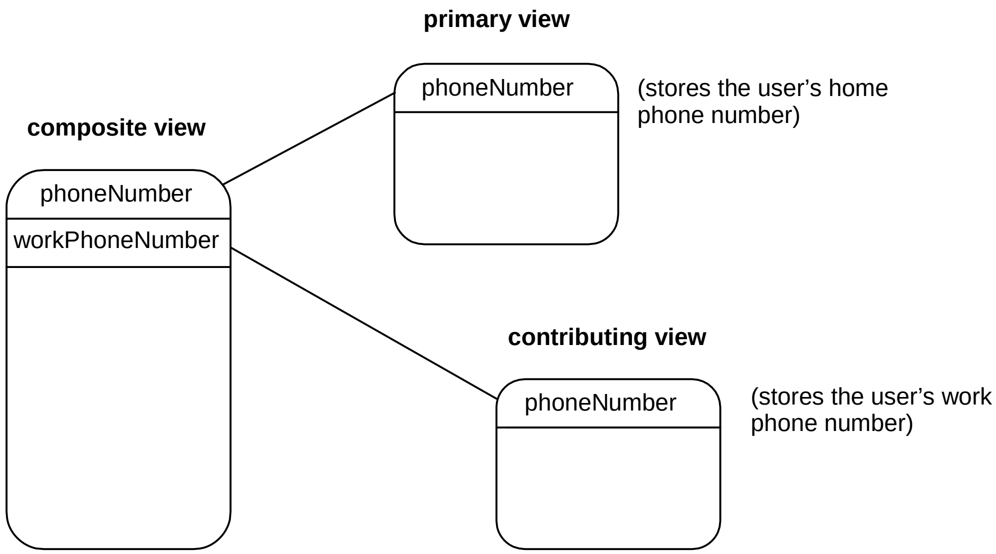
Examples:

```
<primary-item-descriptor name=user>
…
   <property mapped-property-name="fax" exclude="true"/>
…
</primary-item-descriptor>

…

<contributing-item-descriptor name=legacyUser>
…
   <property mapped-property-name="age" exclude="true"/>
…
</contributing-item-descriptor>
```

The default setting for the `exclude` attribute is false. The `mapped-property-name` attribute identifies the property in the underlying repository view that you want to exclude from the composite view.

Explicit Property Mappings

All properties in the composite view must be unique. If two contributing views have properties with the same name, you can avoid collision by mapping a given property in the composite view to one of the contributing property names. The following diagram shows a primary view and a contributing view that both contain a property called `phoneNumber`. One property stores a user's home phone number, and the other holds his or her work phone number. In the example, a `workPhoneNumber` property is created in the composite view and mapped to the work phone property in the contributing view.



Within either the primary or contributing item descriptor definition, use the `mapped-property-name` attribute of the `<property>` tag to create an explicit property mapping. Example:

```
<contributing-item-descriptor name=legacyUser>
…
   <property name="workPhoneNumber" mapped-property-name="phoneNumber"/>
…
</contributing-item-descriptor>
```

The `mapped-property-name` specifies the name of the property in the primary or contributing view (the underlying repository). The `name` value specifies the new name for the property in the composite view.

For more information on `<property>` tags, see Standard User Profile Repository Definition in this manual and also the *SQL Repository Definition Tag Reference* chapter in the *ATG Repository Guide*.

Sample Definition File for a Composite Profile Repository

The following sample shows a simple composite repository definition file for a profile repository that has a SQL primary view and one contributing LDAP view. For detailed information on the tags you can include in this file, including a description of the composite repository DTD, please refer to the *ATG Repository Guide*.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE composite-repository-template
  PUBLIC "-//Art Technology Group, Inc.//DTD Scenario Manager//EN"
  'http://www.atg.com/dtds/composite-repository/composite-repository_1.0.dtd' >
```

```
<!-- composite repository definition -->
<composite-repository-template>

  <header>
    <name>A sample Composite Repository template</name>
    <author>Granny</author>
    <version>$Change: 240431 $$DateTime: 2002/05/17 17:23:12 $$Author: jsmith
       $</version>
  </header>

  <!-- composite item descriptor definition -->
  <item-descriptor name="user"
                   default="true"
                   display-property="login"
                   contributing-item-creation-policy="eager"
                   null-contributing-item-policy="null"
                   link-method="static">

    <!-- primary view definition -->
    <primary-item-descriptor name="gsaUser"
      repository-nucleus-name="/atg/userprofiling/gsa/GSARepository"
      repository-item-descriptor-name="user"
      all-properties-propagate="true"
      all-properties-queryable="true">
    </primary-item-descriptor>

    <!-- contributing view definition -->
    <contributing-item-descriptor name="ldapUser"
      repository-nucleus-name="/atg/userprofiling/ldap/LDAPRepository"
      repository-item-descriptor-name="user"
      all-properties-propagate="true"
      all-properties-queryable="true">
      <primary-item-descriptor-link>
        <link-via-property primary="login" contributing="ldapLogin"/>
      </primary-item-descriptor-link>
      <property name="ldapFirstName" mapped-property-name="firstName"/>
      <property name="ldapLastName" mapped-property-name="lastName"/>
      <property mapped-property-name="password" exclude="true"/>
    </contributing-item-descriptor>

  </item-descriptor>

</composite-repository-template>
```

Configuring Your Personalization Module for a Composite Profile Repository

By default, the Personalization module is configured to use a standard SQL repository to store profiles. To use a composite repository instead, you need to perform the following steps:

1. Override the `ProfileAdapterRepository` component with an instance of `atg.adapter.composite.MutableCompositeRepository`. See Overriding the ProfileAdapterRepository Component for more information.

   Alternatively, you can create a new instance of the composite profile repository and then change `/atg/userprofiling/ProfileTools` and other components to point to it. This option requires more changes and it is useful only if you must keep both the standard SQL profile repository and a composite profile repository. See Creating a Separate Composite Profile Repository for more information.

2. If necessary, change the `defaultProfileType` property in the `ProfileTools` component to point to the name of a composite view in the composite profile definition. Select the view that you want to use as the default item type if no other types are specified, for example in a query. You must perform this step if the composite repository does not contain a composite view with the same name as the current value of this property (the default value is `user`). Note that this change may be necessary regardless of the option you select in step 1.

3. Change the names of any properties as required in the `PropertyManager` component. See Updating the PropertyManager for a Composite Repository for more information.

Overriding the ProfileAdapterRepository Component

To use a composite profile repository instead of the standard SQL profile repository, override the default `/atg/userprofiling/ProfileAdapterRepository` so that it is an instance of a `MutableCompositeRepository` instead. To do this, create an instance of `atg.adapter.composite.MutableCompositeRepository` as described in Creating a Composite Profile Repository Component, but save it at the Nucleus address `/atg/userprofiling/ProfileAdapterRepository`.

Note that performing this step also allows a writable composite profile repository to appear in the ACC (for example, in the People and Organizations > Users window). The composite repository whose definition file you specify in the `configurationFile` property appears in the ACC interface.

Creating a Composite Profile Repository Component

The composite profile repository itself is a component of class `atg.adapter.composite.MutableCompositeRepository`. Create an instance of this component with contents similar to the settings in the `MutableCompositeRepository.properties` file shown here:

```
$class=atg.adapter.composite.MutableCompositeRepository

repositoryName=UserProfiles
```

```
configurationFile=/atg/userprofiling/composite.xml
transactionManager=/atg/dynamo/transaction/TransactionManager
groupContainer=/atg/registry/RepositoryGroups
#loggingDebug=true
#debugLevel=20
```

In the `configurationFile` property, specify the XML definition file you created for the composite profile repository. For more information, refer to *Configuring the Composite Repository Component* in the *Composite Repositories* chapter of the *ATG Repository Guide*.

Creating a Separate Composite Profile Repository

If you need to keep both the standard SQL profile repository and a separate composite profile repository, you can do so by creating a new instance of `atg.adapter.composite.MutableCompositeRepository` as described in the previous section, and then pointing all profile repository references that may exist in other components to this composite repository. For example, you must change the value of the `profileRepository` property in the `/atg/userprofiling/ProfileTools` component as described below. In addition, you must find all references to the `ProfileAdapterRepository` in all other components and change them so that they refer to the new composite repository. Because there are potentially dozens of such references in an ATG application, this option is not recommended.

Configure the `ProfileTools` component as follows:

Point the `profileRepository` property of the `ProfileTools` component to the composite profile repository. For example:

```
profileRepository=/atg/userprofiling/mutableCompositeRepository
```

For more information, refer to Configuring the ProfileTools Component in the *Setting Up a Profile Repository* chapter of this guide.

Updating the PropertyManager for a Composite Repository

The `/atg/userprofiling/PropertyManager` component keeps track of the property names of commonly used profile properties such as `login`, `password`, and `email`. Update the setting in this component to reflect the names of these properties as they are in the composite view after all explicit mappings and exclusions have been performed. For example, assume you store the user’s email address in a contributing LDAP repository in a property called `emailAddress`. In the `PropertyManager` component, you would set the email address property name as follows:

```
emailAddressPropertyName=emailAddress
```

See the description of the `atg.userprofiling.PropertyManager` class in the *ATG API Reference* for a complete list of `PropertyManager` property names that you may have to update.

Configuring Targeted E-mail for a Composite Repository

If you want to use persistent targeted e-mail (the `TemplateEmailPersister` component) with a composite repository, and the primary view is an LDAP repository, you must change the `user_id` column in the `dps_user_mailing` table from the following:

```
varchar(40) references dps_user(id)
```

to the following:

```
varchar(255)
```

This change is necessary because when you use a composite repository with LDAP as the primary view, the ID that is stored in this column is the LDAP user ID, not the default Dynamo ID. You can make the change to the database or to the SQL scripts that initialize it.

For more information about the `TemplateEmailPersister` component, see Stopped E-mail Campaigns.

Configuring the Scenarios Module to Use a Composite Profile Repository

Scenario-related profile data must be stored in a SQL repository. If you use a composite profile repository, scenario data must be stored in the primary view, which must use a SQL repository as its data source.

The scenario data that must be in the primary view is any scenario property described in the default `userProfile.xml` file included with the Scenarios module.

If you want to use scenarios that perform queries against a user directory (for example, scenarios that filter users according to their roles or organizations), the user directory information must be stored in a SQL repository. You cannot run scenarios against an LDAP-based user directory, even within a composite repository.

Note also that the Scenarios module uses the values of some profile properties, either as criteria for triggering an element or as part of the process of tracking a user’s progress through a scenario. If you change the names of any of the profile properties required by the Scenarios module, you must update the `/atg/userprofiling/PropertyManager` component for the Scenarios module (class `atg.scenario.userprofiling.ScenarioPropertyManager`) to register the new names.

If you choose to keep both the default SQL profile repository and a separate composite repository (see Creating a Separate Composite Profile Repository), you must also update the `subjectRepository` property in the `/atg/scenario/ScenarioManager` component to point to the new repository.

Performing Queries against a Composite Profile Repository

The composite profile repository supports all Repository Query Language (RQL) queries, even those referencing properties that come from different underlying repositories (as long as they are included in the composite view). For more information on RQL, see *Repository Query Language* in the *ATG Repository Guide*.

Note that query performance depends to some extent on the complexity of the query—specifically, it depends on the number of comparisons across contributing views that a given query requires. For example, the following very common type of query generally provides a high level of efficiency:

- Find all users whose `firstName` is John and whose `phoneNumber` is 555-1212

This type of query compares properties in the composite view to constant values, and it joins those comparisons together via `AND` and `OR` statements. In this example, the composite repository stores `firstName` in the primary (GSA) view and `phoneNumber` in a contributing (LDAP) view. It is impossible, however, to return the appropriate set of composite users from a single query. Therefore, the composite view breaks the query down into sub-queries, each of which can be run against a single underlying repository. Then the results of the sub-queries are put together using the `AND` or `OR` rules, and a final result set is returned.

By contrast, the following example shows a type of query that may perform poorly:

- Find all users whose `dayPhone` is equal to their `workPhone`

Here, the `dayPhone` composite property comes from the primary (GSA) view and `workPhone` comes from a contributing (LDAP) view. For this query, the composite view must find the day phone number of every user in the primary view and compare it to the work phone number of its counterpart in the contributing view. If the values match, the user is added to the result set. Obviously, if the number of users is large, this search could take some time. It is recommended that you use this type of query sparingly to avoid an adverse effect on site performance.

4 Working with User Profiles

This chapter discusses how to work with user profiles to control access to your sites. This chapter includes the following sections:

Tracking Users

Whenever a user accesses a site that uses the Personalization module, two different mechanisms are used to track the user's actions:

- A session is created for the user, and is maintained either through a cookie or through URL rewriting.
- The user is associated with a profile.

The *Session Tracking* chapter of the *ATG Programming Guide* discusses session creation and tracking. This section describes methods for tracking users by associating them with profiles. It discusses the following:

- methods for tracking guest users

- methods for tracking permanent users
- how to use cookies to maintain information about users

Tracking Guest Users

Guests are anonymous users who have not registered and have not logged in. There are several different strategies you can use to track anonymous guest users:

- Maintain a session for each guest, using ATG's session tracking, but do not attempt to gather any additional profile information.
- Maintain a profile for each guest in memory, using implicit profile properties, but discard the profile when the guest's session expires.
- Maintain a profile for each guest in the database, using a persistent cookie to identify anonymous users on subsequent visits. Note, however, that most sites have large numbers of casual or infrequent users. If you maintain a profile for each person who visits a site even once, your database resource requirements may be very heavy.

To maintain persistent profiles for guest users, perform the following steps:

- Set the `persistentAnonymousProfiles` property of the `/atg/dynamo/servlet/dafpipeline/ProfileRequestServlet` component to `true`. With this setting, a new profile is created in the database for each anonymous visitor.
- Set the `persistAfterLogout` property of the `/atg/dynamo/servlet/dafpipeline/ProfileRequestServlet` component to `true`. Setting this property ensures that a profile is created in the profile repository immediately after an anonymous user logs out of a Web site.
- Set the `profileRequestTools` property of the `/atg/dynamo/servlet/dafpipeline/ProfileRequestServlet` component to `/atg/userprofiling/ProfileRequestTools`.
- Enable auto-login by setting the `autoLogin` property to true in the `userprofile.xml` file. (For more information, see Auto-Login with Cookies.)
- Configure your profile repository so that no properties are required (check the repository definition file to make sure that the `required` attribute is not set to true for any properties).
- Configure your profile repository so that properties not necessary for auto-login are not marked as required (check the repository definition file to make sure that the `required` attribute is not set to true for any such properties).

  Note that the login and password properties are required for auto-login to work correctly. The properties are set temporarily to the user ID. When the user registers, they are populated with data that the user supplies.

  To change the value that is used for the temporary data, extend the `/atg/userprofiling/ProfileRequestTools` component and use the `setTemporaryRequiredPropertyValue` method to specify a different temporary value for your required properties.

By default, the Personalization module does not send a login event when it creates a persistent profile for an anonymous visitor. To have the system send a login event in these circumstances, set the `sendLoginEventForNewPersistentAnonymousProfiles` property of the `ProfileRequestServlet` to `true`.

For information about controlling access for guests, see the Controlling Anonymous User Access section in this chapter.

Tracking Registered Users

When a registered user accesses one of your sites, that user needs to identify himself or herself to the Personalization module to ensure that the correct profile is associated with him or her. Typically, this is done by requiring the user to log in. Requiring a login helps maintain the security of your sites.

However, logging in may be an annoyance to users, especially if it is not necessary for security reasons. For example, you may use the Personalization module to target personalized content to registered users, but your sites may not contain any material to which access is restricted. If this is the case, you may want to use the Personalization module's Auto-Login feature. If you use Auto-Login, users do not have to login to your sites, but the Personalization module still has sufficient information to determine the profile to use.

When you enable auto-login, the Personalization module attempts to log in a visitor automatically, using the value of the REMOTE-USER HTTP header from the visitor's request. If the Personalization module doesn't find a profile with a visitor login property that corresponds to the REMOTE-USER header, it treats the visitor as anonymous and creates an anonymous profile, until the visitor actively logs in. The Personalization module cannot automatically log in a visitor unless the visitor is registered.

Auto-Login

The Personalization module offers three methods you can use to automatically log in visitors who are returning to a site without requiring them to enter a login:

- Auto-Login with Basic Authentication
- Auto-Login with Cookies
- Auto-Login by Profile

Note that if you use any of these forms of auto-login, the Personalization module sends the login event before it sends the session creation event. This is because the session event needs to be able to refer to the user's profile.

Auto-Login with Basic Authentication

You can set up your sites to log in member visitors automatically either using cookies, as described in the Auto-Login with Cookies section, or using the ATG's Basic Authentication service.

To enable the auto-login feature for Basic Authentication, set the following property in the `/atg/userprofiling/ProfileRequestServlet`:

```
verifyBasicAuthentication=true
```

For more information about using the Basic Authentication service, see the discussion of the `BasicAuthenticationPipelineServlet` in the *Request Handling with Servlet Pipelines* chapter of the *ATG Programming Guide*.

Auto-Login with Cookies

You can configure the Personalization module to use persistent cookies with its auto-login features. To configure the Personalization module to send auto-login cookies, set the following property in the `/atg/userprofiling/CookieManager` component:

```
sendProfileCookies=true
```

Note that if you use auto-login with cookies, the user can access a site without logging in until the cookie expires. However, if the user explicitly logs out, the Personalization module overwrites the persistent cookie with a temporary cookie, so that the user must explicitly log in the next time he or she accesses the site. (This gives users a way to keep unauthorized people from accessing their data.)

If you use auto-login with cookies, you should not also use auto-login with Basic Authentication. Disable the auto-login feature for Basic Authentication by setting the following property in the `/atg/userprofiling/ProfileRequestServlet`:

```
verifyBasicAuthentication=false
```

See Auto-Login with Basic Authentication for more information.

Auto-Login by Profile

You can set up your sites so that registered users can choose whether or not they want the sites to log them in automatically. After a user registers on one of your sites for the first time, his or her login information is stored in a user profile. You can provide a “Log me in automatically” option that users can choose on your Web sites. If they choose this option, set the `autoLogin` property of their user profile to `true`. (Note that `autoLogin` is the default name for this property; you can change it by setting the `autoLoginPropertyName` property of the `/atg/userprofiling/PropertyManager` component.)

Once users choose auto-login, every time they visit one of your Web sites, the Personalization module checks their user profile and, if the auto-login property is set to true, the `PropertyManager` component tells the `ProfileRequestServlet` to send out an auto-login cookie and allows the user to automatically access the sites. To use this form of auto-login, you must therefore also set the `sendProfileCookies` property of the `CookieManager` component to true. See Auto-Login with Cookies for more information.

Note that auto-login applies to all Web sites in a multisite environment. If a user is automatically logged in for one site in your system, he or she is logged into all sites.

Profile Cookie Configuration

You can configure many aspects of whether and how the Personalization module sends profile cookies using the `CookieManager` component (`/atg/userprofiling/CookieManager`). The `CookieManager` has the following properties:

sendProfileCookies
Set to `true` to send a profile cookie including the user ID. See Auto-Login with Cookies. (Default `false`)

profileCookieDomain
If present, this defines the value of the `domain` field that is sent for profile cookies. (Default null)

profileCookieComment
Comment of the cookie used to carry the user ID, if cookies are in use. (Default null)

profileCookieMaxAge
If present, this defines the value of the maximum age of the cookie, in seconds. A value of -1 indicates that there is no maximum age, making cookies non-persistent. (Default -1)

profileCookiePath
If present, this defines the value of the `path` field that will be sent for profile cookies. (Default `/`)

profileCookieSecure
If `true`, cookies will include the `secure` field, which indicates to the browser that cookies should only be sent using a secure protocol, such as HTTPS or SSL. (Default `false`.) Note that, depending on the browser, this setting could prevent visitors from using the auto-login feature to access the site.

cookieHashKey
Sets a secret key that the Personalization module uses to hash the user ID cookie. This behavior makes user cookies more secure and prevents users from using another user's profile by changing their cookie. Invalid profile cookies are ignored. You may want to change this from the default value, so that your site cookies will be hashed with a different key from that used by other sites that run ATG products.

Using Persistent Cookies

By default, the cookies that the Personalization module sends are temporary; they expire when the user exits the browser. To enable auto-login or persistent anonymous profiles, you must configure the `/atg/userprofiling/CookieManager` component to use persistent cookies.

The `profileCookieMaxAge` property of the `CookieManager` component controls cookie persistence. This property sets the number of seconds from the time the profile cookie is sent until it expires. If you set the property to -1 (the default), cookies are not persistent.

For example, suppose you enable auto-login, but you want the user to log in manually after a week. You would set `profileCookieMaxAge` to the number of seconds in a week:

```
profileCookieMaxAge=604800
```

Securing Cookies

To make user cookies more secure and prevent site visitors from changing their cookies (which could allow them to use someone else's profile), the Personalization module includes a feature for checking profile ID cookies that it can use to validate the visitor's cookie.

If you choose to send profile cookies (by setting `sendProfileCookies` to `true`), the Personalization module automatically sends two cookies, DYN_USER_ID and DYN_USER_CONFIRM. The DYN_USER_CONFIRM cookie is a hash of the user ID cookie. If the hashed DYN_USER_CONFIRM cookie does not match the user ID cookie, the Personalization module ignores the cookies and creates a new profile.

To change the secret key that the Personalization module uses to hash the user ID cookie, edit the following property of `/atg/userprofiling/CookieManager`:

```
cookieHashKey=
```

Security Status

Web sites need to balance the user's convenience with the need for security. If security is critical, you will typically want to require the user to enter a user name and password to access your sites. If security is not critical, your site can use auto-login so the user does not have to manually log in.

In some cases, you may want to require users to log in manually only if they access certain pages. To do this, you can use a combination of auto-login and manual log in. A user can access a site without manually logging in, but must explicitly provide a user name and password before seeing restricted pages.

The `securityStatus` property of the `Profile` component is used to indicate the type of verification the user has passed through. When a user logs in (or is automatically logged in), the Personalization module sets the value of this property to an integer that indicates the login method used. You can then use this information in other areas of the site.

For example, if the user tries to access a restricted page, you can check the `securityStatus` value. If `securityStatus` indicates the user has already manually logged in, the restricted page is displayed; if `securityStatus` indicates the user was automatically logged in, the site prompts for a user name and password before displaying the page, to ensure that no unauthorized person can access the restricted content. The system then changes the value of `securityStatus` to indicate that the user has logged in manually, so future attempts to access restricted content during the session do not require re-entering the user name and password.

The following table lists the values of `securityStatus` and the login methods to which they correspond. Note that the default value is 0. If a user accesses the site without logging in at all, the value of `securityStatus` is 0. If the user logs in manually, or is automatically logged in, the system sets `securityStatus` to one of the other values.

Value	Login Method Used
0	Anonymous login
1	Auto-login by URL parameter By default this login method is disabled. You can enable it by setting the `extractProfileFromURLParameter` property of `/atg/userprofiling/ProfileRequest` to `true`.

2	Auto-login by cookie By default this login method is disabled. You can enable it by setting the `extractProfileFromCookieParameter` property of `/atg/userprofiling/ProfileRequest` to `true`.
3	Login by http-(basic-authentication) The user has provided a login and password, but it was provided to and verified by the Web server, not the Personalization module.
4	Explicit login or registration in the Personalization module The user signed in or registered using a login form that invoked `/atg/userprofiling/ProfileFormHandler`.
5	Explicit login or registration in the Personalization module under `https` protocol. Similar to 4, but login was through secure sockets layer (`https` protocol).
6	Certificate (not supported by ATG at this time).

Repository Definition

The security status property is defined in the standard user profile template as follows:

```
<!-- The securityStatus property is transient but a column exists in
the dps_user table.-->
<!-- If you would like to use it, just put this property descriptor within
the table tags.-->
<!-- If this property is made persistent, you may also make it queryable.-->
<property name="securityStatus" data-type="enumerated" default="ANONYMOUS"
    queryable="false" category-resource="categoryInfo"
    display-name-resource="securityStatus"
    property-type="atg.repository.SessionEnumPropertyDescriptor">
       <option value="ANONYMOUS" code="0"/>
       <option value="URL-PARAM" code="1"/>
       <option value="AUTO-SIGNIN" code="2"/>
       <option value="HTTP-BASIC-AUTH" code="3"/>
       <option value="EXPLICIT-SIGNIN" code="4"/>
       <option value="SECURE-SIGNIN" code="5"/>
       <option value="CERTIFICATE" code="6"/>
       <attribute name="resourceBundle"
        value="atg.userprofiling.UserProfileTemplateResources"/>
     </property>
```

`securityStatus` is a transient property by default, which means that it is not stored in the underlying database (for more information, see the *ATG Repository Guide*). It is also a property of type `atg.repository.SessionEnumPropertyDescriptor`, which means that its state is maintained in the user's session rather than in the cache for the profile repository item. This behavior ensures that each visit to the Web site produces a unique securityStatus value, and that the securityStatus always expires when the session ends.

Disabling Security Status

By default, security status is enabled. You can disable it by setting the enableSecurityStatus property of the /atg/userprofile/ProfileTools component to false.

If you do not have the securityStatus property configured in your profile definition file, the Personalization module automatically disables security status to avoid errors.

Security Status and Failover

In some cases you may want the security status to be reset for all users after server failover, requiring them to log in again. In other cases, it may be appropriate for your users' security status to persist after failover. You can define this behavior through the failedOverSecurityStatus property in the /atg/userprofiling/ProfileFailService component.

By default, the property is set to 3 (basic authentication only; users must log in to the Personalization module again). If you set the value to -1, the users' existing security status will be the same after failover.

If you change the value of the failedOverSecurityStatus property, you must also add ProfileFailService.failedOverSecurityStatus to the list of properties to fail over. See *Enabling Session Backup* in the *ATG Installation and Configuration Guide* for more information.

Using Security Status in Content Pages

You can access the value of the securityStatus property in any content page (JSP or .JHTML file). For example, in JSP code you could display the value using a dsp:valueof tag:

```
<dsp:valueof bean="/atg/userprofiling/Profile.securityStatus"/>
```

In JHTML:

```
<valueof bean="/atg/userprofiling/Profile.securityStatus"></valueof>
```

In actual use, you will typically want to compare the value of securityStatus to some minimum security level. The /atg/userprofiling/PropertyManager component has seven properties that correspond to the levels of securityStatus:

- securityStatusAnonymous
- securityStatusUrl
- securityStatusCookie
- securityStatusBasicAuth
- securityStatusLogin
- securityStatusSecureLogin
- securityStatusCertificate

For example, the PropertyManager.securityStatusLogin property has a default value of 4, which is the value that Profile.securityStatus is set to if the user logs in manually under http protocol.

The following JSP example illustrates a typical use of securityStatus. The Compare servlet bean compares the value of securityStatus to the value of the securityStatusLogin property of the PropertyManager component. If the value of securityStatus is less than the value of securityStatusLogin, the user did not log in manually, so a login form is displayed. If the value of securityStatus is greater than or equal to the value of securityStatusLogin, the user has already manually logged in, so the requested content is displayed.

```
<dsp:droplet name="Compare">
  <dsp:param bean="Profile.securityStatus" name="obj1"/>
  <dsp:param bean="PropertyManager.securityStatusLogin" name="obj2"/>
  <dsp:oparam name="lessthan">
     <!-- send the user to the login form -->
     <dsp:include page="login_form.jsp"></dsp:include>
  </dsp:oparam>
  <dsp:oparam name="default">
     <!-- allow the user to proceed to the protected content -->
     <dsp:include page="protected_content.jsp"></dsp:include>
  </dsp:oparam>
</dsp:droplet>
```

Here is the same example in JHTML:

```
<droplet bean="Compare">
  <param name="obj1" value="bean:Profile.securityStatus">
  <param name="obj2" value="bean:PropertyManager.securityStatusLogin">
  <oparam name="lessthan">
     <!-- send the user to the login form -->
     <droplet src="login_form.jhtml"></droplet>
  </oparam>
  <oparam name="default">
     <!-- allow the user to proceed to the protected content -->
     <droplet src="protected_content.jhtml"></droplet>
  </oparam>
</droplet>
```

User Profiling Tools

This section describes services within the Personalization module that can help you handle user profiles in your application. These services within Nucleus are built upon the underlying Repository API.

/atg/userprofiling/Profile

The Personalization module creates a `Profile` session-scoped object for each user when the request is processed by the `ProfileRequestServlet`. This component is a wrapper around the `RepositoryItem`, which represents the user. In the default configuration, the `ProfileRequestServlet` creates an anonymous RAM-based profile and sets the `dataSource` property of the `Profile` object. Later, if the user logs in or goes through some other authentication process, the instance of the `Profile` object does not change in the session, but the `dataSource` value is swapped out with the `RepositoryItem` that represents the user's persistent profile.

You should use the `Profile` component to access the profile properties of the user. A registered property mapper allows the profile properties to be accessible from your site content pages. (See the Setting Up Targeting Services chapter.) The instance of the `Profile` object is what the targeting engine uses to evaluate business rules in context of the person.

The class that defines this component implements the `RepositoryItem` interface, and all calls to those methods pass through to the `RepositoryItem` specified by the `dataSource` property. In addition, this class adds an extra JavaBean property named `transient`. The `transient` property returns `true` when the profile for the user is anonymous and returns `false` when the user is authenticated. You may find this property useful with `Switch` servlet beans within content pages to show content to members of your sites.

You can subclass this component and add your own session-based JavaBean properties. These properties will also be accessible in targeting and visible in the the ATG Business Control Center and the ACC. To register your own subclass to be instantiated for the `/atg/userprofiling/Profile` component, you need to override the `$class` definition from the default configuration.

You can also add new property types to the SQL repository. Since most Profile objects are `RepositoryItems` in the SQL repository, adding new property types to an item may be more effective than creating your own subclass of the Profile object. For more information, see *User-Defined Property Types* in the *SQL Repositories* chapter of the *ATG Repository Guide*.

/atg/userprofiling/ProfileRequest

The `ProfileRequest` component is a request-scoped object that gives you information about the request as related to profile-specific parameters. It tells a developer the source of information used to fetch the profile (e.g. persistent cookie, basic authentication). In addition, it can give you the status of the profile request (e.g. old vs. new profile).

/atg/userprofiling/ProfileTools

The `ProfileTools` service is a very useful component because it implements many different pieces of functionality related to the Repository API. In addition, it has references to the other globally-scoped Open Profile Adapter services. It includes a reference to the current Profile repository through the `profileRepository` property. If you need access to the global profile services within your own components, it usually makes sense to have your own classes include a property reference to the `ProfileTools` object. From this object, you can access all other facilities. Alternatively, you can keep specific properties in your components (for example just a `MutableRepository` property), but you must define the source of your property to link to the `ProfileTools's` property reference.

This service provides methods that deal with the session-scoped `Profile` object and also contains lower-level, repository-specific helper methods. The class defines methods for locating users by login or user ID, creating users, and updating properties of profiles. These methods save the developer time because they all use the same Repository APIs. For example, if you need to find a profile based on a login name, normally you would have to do the following:

- fetch the `RepositoryView` that contains the user
- build a query object to perform a comparison between the supplied user name and the login profile property
- execute the query

That logic sequence is written already, and you can access it with a single method call from the `ProfileTools` component.

Note that a similar implementation of this service exists for the internal profile repository: `/atg/userprofiling/InternalProfileTools`.

/atg/userprofiling/PropertyManager

The `PropertyManager` keeps track of information about important profile properties. This includes system-wide functionality that manages "well known" properties that the personalization engine expects. For example, most personalization systems rely on a login profile property. You may want to change the exact name of that property due to legacy issues with different systems (for instance, a system may need to call it `username` rather than `login`). This service allows you to configure the Personalization module to know the name it should use when dealing with these expected profile properties. This component is used by the `ProfileTools.locateUserFromLogin` method, which builds a `Repository` query looking for people with a specific login property. You can use this service to configure the name that this method uses for the login property.

The `PropertyManager` also provides configuration properties to determine if the Personalization module should be using a one-way hash of a password or store it as clear text in the database, and to specify the hashing algorithm. For more information, see the Password Hashing section of this chapter.

/atg/userprofiling/ProfileEventTrigger

The `ProfileEventTrigger` defines methods that fire off events into the system that correspond to discrete points in a user's site experience. These points include login, logout, and registration events. The `ProfileFormHandler` and `MultiProfileFormHandler` use this object to automatically broadcast these events. If you create custom form handlers, you can use this service to broadcast events as well.

/atg/userprofiling/ProfileUpdateTrigger

The `ProfileUpdateTrigger` component defines methods that send JMS messages when a user or an administrator updates a profile by way of a profile form handler or multiple profile form handler (see Profile Form Handlers).You can then set up scenario events that are triggered when one of these messages is received. For example, you could create a scenario that sends an e-mail when a user changes his or her `maritalStatus` property from "single" to "married". For information on how to configure such a scenario, see *Updates Profile Property* in the *Scenario Events* section of the *ATG Personalization Guide for Business Users*.

The `ProfileUpdateTrigger` component contains four properties that trigger the sending of different types of profile update message. All four properties are set to true by default, and each one has a corresponding event that you can include in scenarios. (For more information, see Using Scenario Events.)

The properties of the `ProfileUpdateTrigger` component are described below.

Property	Description
`messageSource`	Specifies the name of the `messageSource` component to use to send messages Default: `/atg/userprofiling/DPSMessageSource`
`generateProfileUpdateEvents`	Sends a `ProfileUpdate` message when a user changes his or her profile. Triggers an Updates Profile scenario event in the Scenarios module. Default: `true`
`generateAdminProfileUpdateEvents`	Sends an `AdminProfileUpdate` message when an administrator changes a user profile. Triggers a Profile Is Updated event in the Scenarios module. Default: `true`
`generateProfilePropertyUpdateEvents`	Sends a `ProfilePropertyUpdate` message. A message is sent when a users changes any of the properties listed in the `propertiesToSendUpdateEvents` property. Triggers an Updates Profile Property event in the Scenarios module. Default: `true`

`generateAdminProfilePropertyUpdateEvents`	Sends `AdminProfilePropertyUpdate` messages. A message is sent when n administrator changes any of the properties listed in the `propertiesToSendAdminUpdateEvents` property. Triggers a Profile Property is Updated event in the Scenarios module. Default: `true`
`propertiesToSendUpdateEvents`	The list of properties referenced by the `generateProfilePropertyUpdateEvents` property. Default: `null` (a message is sent when any property is changed)
`propertiesToSendAdminUpdateEvents`	The list of properties referenced by the `generateAdminProfilePropertyUpdateEvents` property. Default: `null` (a message is sent when any property is changed)

Profile Form Handlers

The Personalization module provides form handler classes that you can use to create and manage user profiles. A profile form handler connects a registration or login page to a profile in a profile repository. You can use a profile form handler to add new profiles, edit the current profile, and handle user login and logout.

The main profile form handler class is `atg.userprofiling.ProfileFormHandler`. This class provides all of the form handling functionality that many sites will need. The *Using Profiles and Profile Forms* chapter of the *ATG Page Developer's Guide* explains how to use this form handler class in content pages.

Most of the functionality in the `ProfileFormHandler` class is inherited from `atg.userprofiling.ProfileForm`, which it extends. The source code for both `ProfileFormHandler.java` and `ProfileForm.java` can be found in the `<ATG10dir>/DPS/src/Java/atg/userprofiling` directory.

This section discusses how the `ProfileForm` class and the `ProfileFormHandler` class work internally. You can create your own form handler classes by extending either of these classes. The class to subclass depends on what you want to use the new profile form handler for. The `ProfileForm` class is designed to be more generic and can perform operations on any profile, whereas the `ProfileFormHandler` class operates on the user's current session-scoped `Profile` object.

Note that when you use an instance of a profile form handler, you must make sure that certain components are correctly configured:

- All profile form handlers have a `profileTools` property that must reference the globally scoped service `/atg/userprofiling/ProfileTools`.
- If the profile form handler is of class `atg.userprofile.ProfileFormHandler` (or a subclass of this class), it has a `profile` property that must reference the session-scoped `Profile` object, located at `/atg/userprofiling/Profile`.

The ProfileForm Class

The `ProfileForm` class has a set of submit handler methods for working with profiles. Each of these methods performs a specific form handling task, such as logging in a user. When a handler performs its task, it can perform several additional actions, including preprocessing and post-processing steps, and redirecting the user to the appropriate page depending on whether or not it encountered errors.

The following table summarizes the main handler methods of the `ProfileForm` class. Each of these methods is then explained in greater detail.

Method	Function
`handleCreate`	Creates a new permanent profile and sets the profile attributes to the values entered in the form.
`handleUpdate`	Modifies the attributes of the current profile.
`handleLogin`	Uses the `login` and `password` values entered by the user to associate the correct profile with that user.
`handleChangePassword`	Changes the password attribute of the profile to the new value entered by the user.
`handleLogout`	Resets the profile to a new anonymous profile and optionally expires the current session.

handleCreate

The `handleCreate` method calls three other methods that you can override to add to or modify the default functionality. The `preCreateUser` and `postCreateUser` methods are protected methods with no functionality. They exist as stubs for subclasses to insert application logic before and after a persistent user is registered in the profile repository. After the `preCreateUser` method is called, the handler invokes the `checkFormError` method. This checks for errors in the form and stops processing if it finds any. The `createUser` method takes all the form values submitted by the user and attempts to register the user.

The first step in registering a user is verifying that the user has supplied all the required form parameters. By default, the only required form parameter is the confirmation password parameter. The `checkForRequiredParameters` method is invoked if the `ProfileForm` component's `checkForRequiredParameters` property is set to `true`. In addition, if the `ProfileForm` component's confirmPassword property is set to true, the handler requires the user to submit the form with an additional parameter that is not part of the profile definition. This parameter is the confirmation password parameter, which is used to verify the password profile attribute value supplied.

After this first pass for verification, the system checks to see if a user already exists with the given login name, and then uses the createProfileItem method to construct a MutableRepositoryItem that represents the new user. At this point, the user's profile exists as an object, but is not yet stored in the database.

After the MutableRepositoryItem is created, it is updated to contain all the attributes the user supplies. Next, if the ProfileForm component's checkForRequiredProperties property is set to true, the form calls the checkForRequiredProperties method to see if any of the profile attributes are required. If any profile attributes are required, it checks whether the value of any of the required profile attributes is null.

Finally, the createUser method transforms the RAM-based profile into a persistent profile stored in the profile database by invoking the addUser method. After the createUser method is finished, the handleCreate method takes the new persistent MutableRepositoryItem and updates the form handler to use that as the current profile. At this point, the session-scoped Profile object's dataSource property is updated so that it also references the new persistent user data structure. As a final step, the postCreateUser method is called, by which time the user will be a fully registered member. As noted above, the postCreateUser does not do anything; it exists as a stub that you can override if you subclass ProfileForm.

handleLogin

The handleLogin method is similar to the handleCreate method. It, too, provides three significant methods that you can override: preLoginUser, findUser and postLoginUser. For each of the handleX methods, the preprocessing and post-processing methods have no implementation. They are stubs that you can override in subclasses. The findUser method is called after the preLoginUser method. It attempts to locate the correct user based on the login and password information that the user supplies, invoking the /atg/dynamo/security/IdentityManager to perform the verification of the login. If a user is not found with a matching login name and password, the method then attempts to find a user based on only the login name. If this process finds a user, then it is evident that the password supplied was invalid and the form indicates that in an appropriate form exception. If no user is found with a matching login, then the form indicates that the login name supplied was incorrect.

You can specify that certain properties should be copied upon login from the anonymous profile to the registered user's account. This is useful if anonymous site visitors accumulate profile information that you do not want to lose when they log in. The method copyPropertiesOnLogin implements this functionality. Specify which profile attributes you want copied to the persistent profile with the propertiesToCopyOnLogin property of the ProfileForm component.

After the user has been identified by login and password and any desired properties are copied, the data structures for the form handler and session-scoped Profile object are updated to reference the registered user's RepositoryItem.

For more information on login verification, see Managing User Logins.

handleLogout

The handleLogout method allows users to log out of their authenticated sessions. Like the other handleX methods, you can subclass ProfileForm to use the preLogoutUser and postLogoutUser methods. The ProfileForm property expireSessionOnLogout defines the default behavior of the handleLogout operation. The user's session is invalidated when the handleLogout method is called if the expiresSessionOnLogout property is set to true, which is the default value. After logout, a subsequent request from the same user creates a new anonymous RAM profile.

If you do not wish to expire the session when you perform the logout, set the expireSessionOnLogout property to false. With this setting, the form handler and session-scoped Profile objects are reset to use a new anonymous RAM profile, but maintain the same session. When the user is redirected after a successful logout operation, the Personalization module adds an extra query argument labeled DPSLogout to the session ID information The default value of DPSLogout is true. The parameter name is defined by the constant atg.userprofiling.ProfileRequestServlet.LOGOUT_PARAM. The ProfileRequestServlet recognizes this query parameter and disables auto-login for the life of the session by setting a parameter AutoLogin in the user's session to Boolean.FALSE. Otherwise, if you have configured the Personalization module to use auto-login, the persistent cookie or Basic Authentication information automatically attempts to reload the user profile even though the user logged out. See the Tracking Users section of this chapter for more information.

handleDelete

The handleDelete method permanently removes a user from the profile repository. The preDeleteUser and postDeleteUser methods are available for performing actions before and after the process of actually removing the user profile. The removeUser method performs the function of deleting the user profile. This method uses the current value of the repositoryId property of ProfileForm when invoking the MutableRepository.removeItem method.

handleUpdate

The handleUpdate method takes values entered into a profile form and explicitly updates the user's profile with new attribute settings. Processing can occur before and after the update procedure by overriding the preUpdateUser and postUpdateUser methods. The updateUser method modifies the user's profile with the values submitted in the form. This method checks for required parameters in a process similar to the handleCreate method.

The updateUser method calls the updateProfileAttributes method before committing the changes by calling the MutableRepository.updateItem method. All the parameters that are managed by the form handler are stored in a Dictionary through the ProfileForm component's value property. These values are stored in the Dictionary for all form submissions, including operations for handling the registration and login of users. The updateProfileAttributes method iterates through all the submitted parameters and looks for an associated attribute in the user profile.

Once the updateProfileAttributes method finds a profile attribute, the same method checks the value of the profile attribute's RepositoryPropertyDescriptor writable property to determine whether the attribute can be updated. If this value is true, then the value stored in the Dictionary is parsed into the correct data type for the profile attribute. Additional transformations may occur, depending on the property to be updated. For example, for the password property, in the default configuration, the plain-text password value is turned into a hash representation and updated in the profile.

By default, the login and email attributes are trimmed to remove trailing and leading spaces. The list of properties to trim is defined by the trimProperties property. The protected method isTrimProperty is used to test if the specific property should be trimmed. The current implementation iterates over the list of properties specified by the trimProperties property and returns true if a match is found.

The MutableRepositoryItem for the profile is updated with all the form changes. After the update, the same verification for required profile attributes occurs as in the handleCreate process. If there are no errors, all updates are committed in one transaction. If there are any errors, no changes are made.

handleChangePassword

The handleChangePassword method allows users to update their password profile attribute. This specific handler is available because it allows the form to check that the user changing the password knows the original password value. Optionally, you can ask for a confirmation password value to verify that the user did not mistype the new password entry. This handler also contains the standard preprocessing and post-processing methods. The changePassword method compares the old password value with the confirm password value. The generatePassword method of the PropertyManager is used to transform the user's password value (using a one-way hash) into the value that is stored in the database.

The ProfileFormHandler Class

The atg.userprofiling.ProfileForm class can perform its operations on any profile. The property ProfileForm.repositoryId defines the current profile that is being manipulated.

The ProfileFormHandler class is designed to perform its operations on the profile of a particular user navigating your Web site. It adds a profile property that should be set to the session-scoped atg.userprofiling.Profile object. The repositoryId property get and set methods are overridden to reference this Profile object.

The createProfileItem method is subclassed to add optional behavior for registration. Typically, when a visitor registers, you want to transform the current anonymous profile into a registered profile, taking along all of the visitor's current profile attributes. ProfileFormHandler allows you to set the property createNewUser to true, with the result that all registrations start with a clean profile.

The ProfileFormHandler class uses the preprocessing and post-processing methods (such as postCreateUser and preLogoutUser) to extend the functionality in the ProfileForm class. When the user registers, logs in, or logs out, the ProfileFormHandler broadcasts HTTP cookies as needed and fires off Profile Events using the ProfileEventTrigger. You can configure actions that correspond to these events through the ACC. For more information, see the *ATG Personalization Guide for Business Users*.

Ensuring Transactions in Form Handlers

The handler methods in a form handler that manipulates repository items should ensure that all the operations that occur in a method call happen in a single transaction. If all the operations do not occur in the same transaction, there is the risk that incomplete data will be committed to the repository if something goes wrong before the operation is finished. Committing all the operations at once ensures that a repository or database transaction is either completed successfully, or not completed at all (in which case, any partially committed data is rolled back).

The RepositoryFormHandler, ProfileForm, and ProfileFormHandler classes all ensure “atomic” transactions in this way. If you subclass one of these classes without overriding the handler methods, your subclass will handle transactions properly. If you override any handler methods, or add new handler methods, make sure that they handle transactions appropriately as described here.

If you are writing a new form handler that works with repository items, you can avoid the need to add any transaction management code to your handler methods by subclassing the TransactionalFormHandler. In this class, the transaction management occurs in the beforeSet and afterSet methods. This behavior establishes the transaction before any of your properties are set or handler methods are called, rather than in the handler methods themselves. For an example of transaction management within a handler method, see the ProfileFormHandler.java source code in the ATG distribution at <ATG10dir>/DPS/src/Java/atg/userprofiling/.

Multiple Profile Form Handlers

The Personalization module provides multiple profile form handler classes that you can use to create and manage batches of user profiles. A profile form handler connects a registration page to a profile in a profile repository. You can use the multiple profile form handlers to batch add, update, or delete user profiles using a single form.

The multiple profile form handlers extend the atg.repository.servlet.RepositoryFormHandler.MultiProfileForm class. The *Using Profiles and Profile Forms* chapter of the *ATG Page Developer’s Guide* explains how to use these form handler classes in content pages.

You can extend the Multiple Profile Form Handlers, but if you do, you should make sure that they maintain transactional integrity. See the Ensuring Transactions in Form Handlers section for more information.

The MultiProfileForm class

The MultiProfileForm class contains all the code that is shared between the MultiProfileAddForm and MultiProfileUpdateForm classes. This class extends RepositoryFormHandler and is the base class for the MultiProfileAddForm and MultiProfileUpdateForm classes. To learn more about the RepositoryFormHandler class, see *Using Repository Form Handlers* in the *ATG Page Developer’s Guide*.

The MultiProfileAddForm class

Each of the following handle methods calls the corresponding method in the MultiProfileForm class and extends that method. The handleCancel method does not call any method in the MultiProfileForm class. The handle methods in the MultiProfileAddForm class do the following:

handleCreate

The handleCreate method take the current set of user profiles and, if there were no errors submitting the form, creates a new profile for each one by combining the common set of property values with the values specified for each new user. The preprocessing and post-processing methods `preCreateUser`() and `postCreateUser`() are provided to extend the functionality in the `MultiProfileAddForm` class.

handleCancel

The `handleCancel` method clears out the current list of per-user and common property values.

handleClear

The `handleClear` method is identical to the `handleCancel` method: it clear the current list of per-user and common property values. You can use `handleClear()` for greater readability within a form.

The MultiProfileUpdateForm class

Each of the following handle methods calls the corresponding method in the `MultiProfileForm` class and extends that method. The `handleCancel` method does not call any method in the `MultiProfileForm` class. The handle methods in the `MultiProfileUpdateForm` class do the following:

handleUpdate

The `handleUpdate` method loops through a list of `repositoryIds` and modifies the associated profile properties for each update submitted by calling the `RepositoryFormHandler.handleUpdate` method for each of the IDs. The preprocessing and post-processing methods `preUpdateUser`() and `postUpdateUser`() are provided to extend the functionality of the `MultiProfileUpdateForm` class.

handleDelete

The `handleDelete` method loops through a list of `repositoryIds` and modifies the associated profile properties for each update submitted by calling the `RepositoryFormHandler.handleUpdate` method for each of the IDs. The preprocessing and postprocessing methods `preDeleteItem`() and `postDeleteItem`() are provided to extend the functionality of the `MultiProfileUpdateForm` class.

handleCancel

The `handleCancel` method clears out the current contents of the `value` Dictionary for each of the `repositoryIds`.

handleClear

The `handleClear` method is identical to the `handleCancel` method: it clear the current contents of the `value` Dictionary for each of the `repositoryIds` You can use `handleClear()` for greater readability within a form.

Managing User Logins

As mentioned earlier in the description of the `ProfileForm's` handleLogin method, the Personalization module uses the `IdentityManager` interface (specifically, the `/atg/dynamo/security/IdentityManager` component) to handle user login verification and management tasks. The following methods in the `ProfileForm` and `ProfileformHandler` classes call the `IdentityManager`:

- `ProfileForm.findUser()`
- `ProfileFormHandler.preLoginUser()`
- `ProfileFormHandler.postLoginUser()`
- `ProfileFormHandler.preLogoutUser()`

The `IdentityManager` component itself calls the `/atg/userprofiling/ProfileUserAuthority` component to perform the actual process of verifying logins. The `ProfileUserAuthority` is a Personalization module implementation of the `LoginUserAuthority` interface.

For more information on the `IdentityManager` interface, refer to the *ATG API Reference*.

Note: The `UserLoginManager` was used in previous versions of the Personalization module to manage the process of login verification. The `UserLoginManager` functionality was superseded in ATG 6.1 by the `IdentityManager`, and the `UserLoginManager` now points to the `/atg/dynamo/security/IdentityManager` component through an `identityManagerPath` property. For information on updating any existing `UserLoginManager` references to use the new model, refer to the ATG Migration Guide for your version of the product.

Using Case Insensitive Login Names

The Personalization module includes the profile attributes `login` and `password` for each registered user of a Web site. These properties are both case sensitive when performing login queries. If you want to use case-insensitive login names, you need to store a version of the login name with all lowercase characters. Using case-insensitive passwords is not recommended because of security precautions. Using case-insensitive passwords makes it easier for other users or systems to guess a password. To configure the system to use case-insensitive login names, you need to subclass the Profile Form Handler and configure an attribute that converts the login name to lowercase using Java's `toLowerCase` method.

First, add a String attribute to the `Profile` object. You can call this String anything you want; it represents the login name as the user enters it into the form. In this example, this String is called `memberName`. The lower case version of the `memberName` is stored in the `login` attribute.

Then, modify your basic registration form so that the `login` name field references the `memberName` attribute. Next, subclass the Profile Form Handler so that it converts the `memberName` attribute to lowercase and copies it to the login name attribute. You can do this by overriding the `preCreateUser` method of the `ProfileFormHandler` class. Once the super-class performs its operation, you can check the `ProfileFormHandler.value` Dictionary property to see if the `memberName` attribute was submitted. If it is found, then convert the `memberName` attribute to lowercase and place it in the `ProfileFormHandler.value` Dictionary property under the key `login`. The remainder of the registration process in the form then automatically updates the `login` profile attribute with the lowercase value. The following code demonstrates this explanation:

```
protected void preCreateUser(DynamoHttpServletRequest pRequest,
                             DynamoHttpServletResponse pResponse)
```

```
     throws ServletException, IOException
{
   super.preCreateUser(pRequest, pResponse);
   // Look for the submitted member name
   String memberName = getValue().get("memberName")

   if (memberName != null) {
     // Normalize the member name
     String login = memberName.toLowerCase();
     getValue().put("login", login);
   }
   else {
     // If the member name is not available, then make sure we clear out
     // any old login values
     getValue().remove("login");
   }
}
```

If you allow site members to change their login name after registration, you need to make sure that the `memberName` and `login` attributes are still in sync and that the `login` value is always the lowercase version of the `memberName` value. To do this, override the `ProfileFormHandler.preUpdateUser` methods so that it performs the same function as the `preCreateUser` example above. This ensures that the login and authentication process also uses the lowercase version of the login name when performing queries.

If all your authentication is performed through the `ProfileFormHandler`, then you can override the `findUser` method. For example:

```
protected RepositoryItem findUser(String pLogin,
                                  String pPassword,
                                  Repository pProfileRepository,
                                  DynamoHttpServletRequest pRequest,
                                  DynamoHttpServletResponse pResponse)
     throws RepositoryException, ServletException, IOException
{
   if (pLogin != null) {
     return super.findUser(pLogin.toLowerCase(), pPassword,
                           pProfileRepository, pRequest, pResponse);
   }
   else {
     return super.findUser(pLogin, pPassword,
                           pProfileRepository, pRequest, pResponse);
   }
}
```

In this example the `super.findUser` methods uses the `ProfileTools.getItem` method. If you examine the `ProfileForm` class code, you can see that this method does the following:

```
return getProfileTools().getItem(pLogin, pPassword, getLoginProfileType());
```

If you use the `ProfileTools` class to perform authentication in other areas of your Web site, then you should override this `getItem` method so that it also performs the `toLowerCase` operation. For example, you could use the following code to override the `getItem` method:

```
public RepositoryItem getItem(String pLogin, String pPassword,
                              String pProfileType)
{
   if (pLogin != null) {
     return super.getItem(pLogin.toLowerCase(), pPassword, pProfileType);
   }
   else {
     return super.getItem(pLogin, pPassword, pProfileType);
   }
}
```

Password Hashing

For security reasons, you may want to store passwords in hashed form. This guards against the possibility that someone who gains unauthorized access to the database can retrieve the passwords of every user in the system. Hashing performs a **one-way** transformation on a password, turning the password into another String, called the **hashed password**. “One-way” means that it is practically impossible to go the other way - to turn the hashed password back into the original password. There are several mathematically complex hashing algorithms that fulfill these needs. By default, the Personalization module uses the MD5 algorithm to perform a one-way hash of the password value and to store it in hashed form.

The hashed password value is not encrypted before it is stored in the database. When a member attempts to log in, the Personalization module takes the supplied password, performs a similar one-way hash and compares it to the database value. If the passwords match, then login is successful.

If you do not want to use the hashing function, you can disable it by setting the `passwordHasher` property of the `/atg/userprofiling/PropertyManager` component to `/atg/dynamo/security/NullPasswordHasher`. Thereafter all passwords will be stored and compared in clear text. You can change the hashing algorithm used by setting the `passwordHasher` property to point to a `PasswordHasher` component that uses the appropriate hashing algorithm. ATG provides the following `atg.security.PasswordHasher` implementations:

Component	Description
`/atg/dynamo/security/DigestPasswordHasher`	Uses the `java.security.MessageDigest` mechanism for hashing passwords. This hasher digests the password and then encodes using the binary-to-text encoding scheme specified by the `encoding` property (`base16` by default). This hasher does not support one-time hashing; passwords are encoded the same way every time.
`/atg/dynamo/security/MD5PasswordHasher`	MD5-specific version of `DigestPasswordHasher` that supports one-time hashes.
`/atg/adapter/ldap/NDSPasswordHasher`	A password hasher for use with the LDAP repository and the Netscape Directory Server.
`/atg/dynamo/security/NullPasswordHasher`	Stores passwords unhashed. Use this if you want passwords maintained in plain text, rather than hashed.

Password Management Features

This section describes the password management features included as part of the ATG Personalization module. Password management is an important part of administering any site that includes personal information.

Password management features include:

- Setting a regular period at which passwords expire
- Defining rules that users must satisfy when creating passwords
- Handing forgotten passwords
- Notifying a user when a password is about to expire
- Forcing all passwords to expire immediately

Using Password Expiration

This section describes the features that allow you to force passwords to expire periodically or all at the same time.

The password expiration feature allows you to require users to change their passwords after a specified period of time, for example 90 or 120 days.

Enabling Password Expiration

Password expiration is disabled by default. Password expiration must be enabled to either require regular password changes, or force password expiration on all users.

To enable password expiration

1. Override or edit the properties file at `/atg/userprofiling/ExpiredPasswordService` and set `enabled=true`.
2. Add a change password JSP or JHTML page to your sites. This is the form that users are redirected to when it is determined that their password is expired. This can be done using the ACC page template wizard.
3. Configure the `ExpiredPasswordService.redirectPath` property to point to the change password JSP/JHTML page you created.
4. Optionally, configure the `ExpiredPasswordService.passwordValidForNumDays` property to the value of the number of days a password remains valid.

ATG recommends that the change password page be completely static HTML. Once it has been determined that the user's password has expired, all requests passing through the servlet pipeline are redirected to the URL in the `redirectPath` property. Any linked elements in the change password page, such as links to CSS files or images, must be explicitly set in the `/atg/dynamo/servlet/pipeline/ExpiredPasswordServlet.localUrlsToAllow` property in order for the page to render correctly. Note that you do *not* need to list page includes using `dsp:include` and `jsp:include` tags in `localUrlsToAllow`; these bypass the redirect. An example follows:

```
localUrlsToAllow=/templates/style/css/style1.jsp ,
/templates/style/css/style2.jsp
```

Password Expiration Process

Password expiration works as follows:

1. After a user successfully completes the login process, the `ProfileFormHandler` calls the `/atg/userprofiling/ExpiredPasswordService` component to determine if the user's password is expired.

   This component adds the value of the `passwordValidForNumDays` property in the `ExpiredPasswordService` component to the profile's `lastPasswordUpdate`. The result is the date through which the password is valid. If the `lastPasswordUpdate` value is null, it sets the property to 1/1/1970.

   The component compares the result to the current date. If the current date is after the result, it marks the password as expired by setting a the `passwordexpired` session variable to `true`.
2. The `ExpiredPasswordServlet` checks the `passwordexpired` session variable. If true, it redirects the user to the change password form URL defined in the `ExpiredPasswordService.redirectPath` property.
3. When the user submits the change password form successfully, the `passwordexpired` session variable is set to false. The `lastPasswordUpdate` property is set to the current timestamp and persisted.

4. The user can then browse the site as usual.

If the user leaves the site before completing the change password form successfully, the session times out. The password expiration process is repeated the next time the user logs in.

Example 1:

```
passwordValidForNumDays = 90
lastPasswordUpdate in Jim's profile = 01/01/2005
today's date = 03/17/2005
```

`passwordValidForNumDays + lastPasswordUpdate = 01/04/2005`, which is after today's date. The `passwordexpired` session variable is set to `false` for Jim's current session.

Example 2:

```
passwordValidForNumDays = 90
lastPasswordUpdate in Jim's profile = 01/01/2005
today's date = 05/17/2005
```

`passwordValidForNumDays + lastPasswordUpdate = 01/04/2005`, which is before today's date. The `passwordexpired` session variable is set to `true` for Jim's profile.

Forcing All Passwords to Expire

As well as configuring passwords to expire individually according to the date of the last change, you can force all passwords in the profile repository to expire on the same date. To do so, set the `forcePasswordUpdateTimeStamp` property in the `/atg/userprofiling/ExpiredPasswordService` component to the date when you want the passwords to expire. The property is a timestamp that is set to 01/01/2000 by default. All users will be prompted to change their passwords the first time they log in after the specified date.

To expire all passwords immediately and force all users to change their passwords the next time they log in, set the value to the current date.

Setting the `forcePasswordUpdateTimeStamp` value to a date in the future schedules all passwords to expire on that date.

The examples below are all valid formats for specifying the property value:

- 04/23/2007 4:45
- April 23 2007
- April 23 2007 4pm

23 April 2007 16:45 Forced password expiration works as follows:

1. After a user successfully completes the login process, the `ProfileFormHandler` calls the `/atg/userprofiling/ExpiredPasswordService` component, which compares the `forcePasswordUpdateTimestamp` value to the `lastPasswordUpdate` property in the user's profile.

2. If the force update value is after the last password update and before the current date, the password is marked as expired, and the process for having the user change the password is initiated. See the Password Expiration Process section for details.

Example 1:

```
forcePasswordUpdate = 04/04/2005
lastPasswordUpdate in Maria's profile = 02/15/2004
today's date = 05/17/2005
```

The force update value is after the last password update and also before today's date, so the `passwordexpired` session variable is set to true for Maria's current session.

Example 2:

```
forcePasswordUpdate = 04/04/2005
lastPasswordUpdate in Maria's profile = 04/15/2004
today's date = 05/17/2005
```

The force update value is before the last password update, so the `passwordexpired` session variable is not set for Maria's current session.

Notifying Users of Impending Expiration

You can include the `PasswordExpiresSoon` droplet on a page to notify users when their password is about to expire.

This droplet can be found and configured at `/atg/dynamo/droplet/PasswordExpiresSoon`. The `displayCount` setting determines how many times per session the password expiration notification is shown to the customer logging in.

The droplet form is:

```
<dsp:droplet name="/atg/userprofiling/PasswordExpiresSoon">
       <dsp:param name="login" bean="/atg/userprofiling/Profile.login"/>
       <dsp:oparam name="soontoexpiremessage">
       <p>Password will expire in <dsp:valueof param="daysUntilExpired">
           </dsp:valueof> days.
       <p>Change password form here : <dsp:valueof
param="changePwdLocalUrl"></dsp:valueof> <br />
       </dsp:oparam>
</dsp:droplet>
```

Using Strong Password Rule Checks

Strong password checking lets you define criteria for new passwords; for example, you can specify that new passwords cannot be duplicates of old ones or contain the same characters as login names.

You can configure and enable rules to define criteria for new passwords, whether created as part of the forced password expiration process or by a user who has just registered.

Setting Required Criteria for New Passwords

The Nucleus components that represent the default rules are shown below. All password rule components are implementations of the `atg.security.PasswordRule` interface, and they have the Nucleus address `/atg/userprofiling/passwordchecker/<RuleName>`, for example `/atg/userprofiling/passwordchecker/PasswordNotInPreviousNRule.`

Rule	**Description**	**Default Value**
`PasswordMinLengthRule`	The password length must be at least *n* characters.	8
`PasswordMustNotIncludeLoginRule`	The password cannot include the same sequence of characters as the value of the user's `login` property.	none
`PasswordMustIncludeNumberRule`	The password must include at least one numeric character. The rule is an instance of the class `atg.security.PasswordMustIncludeCharacterRule`.	none
`PasswordMustIncludeSymbolRule`	The password must include at least one special character such as a question mark. The rule is an instance of the class `atg.security.PasswordMustIncludeCharacterRule`.	`~!@#$%^&*()_-+={}[]|:;<>,./?`
`PasswordMixedCaseRule`	The password must contain both upper- and lowercase characters.	none
`PasswordNotInPreviousNRule`	The password cannot be equal to the previous *n* passwords.	3

Enabling Password Rule Checking

To enable password rule checking, set the `enabled` property to true in the `/atg/userprofiling/passwordchecker/PasswordRuleChecker` component. The following example shows a properties file for this component, which includes an array that lists the rules you want to use to check passwords:

```
$class.atg.security.PasswordRuleCheckerImpl
enabled=true

rules+=/atg/userprofiling/passwordchecker/PasswordMinLengthRule,\
        /atg/userprofiling/passwordchecker/PasswordMixedCaseRule,\
        /atg/userprofiling/passwordchecker/PasswordMustIncludeNumberRule,\
        /atg/userprofiling/passwordchecker/PasswordMustIncludeSymbolRule,\
        /atg/userprofiling/passwordchecker/PasswordMustNotIncludeLoginRule,\
        /atg/userprofiling/passwordchecker/PasswordNotInPreviousNRule
```

Exclude rules as needed by removing the appropriate line from the array.

Adding New Password Rules

If the preconfigured rules described in the previous section are not sufficient for your sites, you can add additional rules by following the procedures described here.

1. Create a Java class, following the template below. It should extend `atg.nucleus.GenericService` and implement `atg.userprofiling.PasswordRule`.

```
package customPackage;

import atg.nucleus.GenericService;
import atg.servlet.ServletUtil;
import atg.userprofiling.PasswordRule;

public class myPasswordRule extends GenericService implements
     PasswordRule {

     /**
      *
      * Checks the given password against a rule
      *
      * @param password
      * @return true if password passes the rule
      */
     public boolean checkRule(String password, Map map) {
          boolean passed = false;

          if (password==null)
               return false;

          //Do some test
          passed = true;

          return passed;
     }

     /**
      * Returns the rule description as a message for use in
      * a droplet exception for display to user
      */
     public String getRuleResource() {
          return ProfileUserMessage.format("myPasswordRule",
ServletUtil.getUserLocale());
     }
}
```

1. Add code to the `CheckRule()` method to test the given password according to your criteria.
2. Edit the `getRuleResource()` method to replace `myPasswordRule` with a resource. Set up that resource.
3. Compile the class.
4. Create a Nucleus properties file for this rule and save to a Nucleus path similar to `/customPackage/MyPasswordRule.properties`.

   Example:

   ```
   $class=customstuff.myPasswordRule
   ```

5. Edit the Nucleus properties file for the `PasswordChecker` at `/atg/userprofiling/passwordchecker/PasswordRuleChecker.properties`. Include the new rule in the list of rules to check.

   ```
   $class=atg.userprofiling.PasswordRuleCheckerImpl
   rules+=/atg/userprofiling/passwordchecker/PasswordMinLengthRule,\
          /atg/userprofiling/passwordchecker/PasswordMixedCaseRule,\
          /atg/userprofiling/passwordchecker/PasswordMustIncludeNumberRule,\
          /atg/userprofiling/passwordchecker/PasswordMustIncludeSymbolRule,\
          /atg/userprofiling/passwordchecker/
                 PasswordMustNotIncludeLoginRule,\
          /atg/userprofiling/passwordchecker/PasswordNotInPreviousNRule,\
          /mycustomstuff/MyPasswordRule.properties
   ```

Handling Forgotten Passwords

The Personalization module can handle forgotten passwords by directing users to a form where they are prompted to supply a known value for a specific profile property (by default, e-mail address). The Personalization module generates a new password and sends it via e-mail. The e-mail uses a template and can be customized to include the login name and the URL of the login page as well as the new password.

Forgotten Password Process

The forgotten password process works as follows:

1. Users browse to the forgotten password form.
2. Users enter their e-mail address (or username) and submit the form.
3. The `ForgotPasswordHandler` does the following:
   - Locates the user's profile. If the servlet cannot find the user, an error message is displayed stating 'No user information for `email@address.com`."

- Generates a new password by calling the `PasswordGenerator` component.
- Updates the password property in the profile with the new value. The new password is valid for all Web sites in a multisite environment.
- Flags the profile as having a generated password.
- Constructs and sends the e-mail message using `TemplateEmailSender`.

  If the form’s `emptySite` property is set to `true`, a null `siteId` is passed to the e-mail, and any `siteId` specified in the form is ignored. If the `emptySite` property is set to `false` or not provided, the `siteId` that is passed through the form is used in the e-mail. If no `siteId` is provided, the form defaults to the current site context.

4. Users receive an email including their login name and the new password based on the given template. A link is provided to the login form based on the given template.
5. Users either click the e-mail link or browse to the site, and then log in with the new credentials.
6. The server sets the transient `passwordExpired` flag in the user profile to true, thereby requiring the user to change the password immediately.

Generated passwords are not saved to the Previous N passwords list.

Note that the Forgotten Password logic replaces the password value in the user profile, so the old password is no longer valid.

Enabling the Forgotten Password Features

To enable and configure the forgotten password features, complete the following steps:

1. Create a form that allows the user to submit a known value, for example e-mail address.
2. Configure the SMTP server. This can be done in the Configuration Manager at:

   ```
   http://localhost:port/dyn/admin/atg/dynamo/admin/en/
   configure-email-handler.jhtml
   ```

   (where the default port numbers on JBoss, Oracle WebLogic, and IBM WebSphere are 8080, 7001, and 9080, respectively. For more information, see *Connecting to the Dynamo Administration UI* in the *ATG Installation and Configuration Guide*.
3. Set the Email Handler Host to your SMTP server. The Email Handler Port is usually set to port 25.
4. Override or edit the Nucleus component `/atg/userprofiling/ForgotPasswordEmailInfo` and set the following properties:

   ```
   # The URL of the email template jsp/jhtml page
   templateURL=

   # Subject field of the email
   messageSubject=Forgot Password Email
   ```

```
# From field of the email
messageFrom^=/atg/dynamo/service/SMTPEmail.defaultFrom

# MessageContentProcessor responsible for processing the content
contentProcessor=/atg/userprofiling/email/
       HtmlContentProcessor
```

5. Customize the email template.

   Note: if your template JSP/JHTML page contains links to other URLs on your site, you must specify them as absolute URLs in order for the email recipients to be able to access the linked pages. Use the full <code>http://server:port/...</code> form of the URL.

   An example follows:

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

<HTML> <HEAD>
<TITLE>Forgot Your Password</TITLE>
</HEAD>

<dsp:importbean bean="/atg/userprofiling/Profile"/>

<BODY BGCOLOR="#FFFFFF" VLINK="#637DA6" LINK="#E87F02">
<font face="verdana" size=2>Dear
<dsp:droplet name="/atg/dynamo/droplet/Switch">
  <dsp:param bean="Profile.firstname" name="value"/>
  <dsp:oparam name="unset">
   Sir or Madam,
  </dsp:oparam>

  <dsp:oparam name="default">
    <dsp:valueof bean="Profile.firstName"/>
    <dsp:valueof bean="Profile.lastName"/>,
  </dsp:oparam>
</dsp:droplet>

<p>Here is your login information with a new passord.
<p>Login: <dsp:valueof bean="Profile.login"/>
<p><b>New password: <%=request.getParameter("newpassword")%></b>

</font>
</body>
</html>
</dsp:page>
```

Access Control

Some Web applications need to control access to some or all of the pages on the site. For instance, you may want to restrict access to registered members or paid subscribers.

atg/dynamo/servlet/dafpipeline/AccessControlServlet is documented in the ATG Prog. Guide,

The Personalization module adds an Access Control Servlet to the standard ATG servlet pipeline. The Access Control Servlet (`/atg/userprofiling/AccessControlServlet`) can allow or deny access to a page or group of pages based on criteria such as membership in a group or satisfaction of a targeting rule.

Configuring the Access Control Servlet

The Access Control Servlet registers one or more `AccessController` components in its `accessControllers` service map property. This property maps URLs to `AccessController` components. If the URL requested is mapped to an `AccessController` component, the request's Profile object is passed to the `AccessController`, which determines whether or not access should be allowed. If access is allowed, the request is passed on; if access is denied, the servlet redirects the user to a specified `deniedAccessURL`.

`AccessController` is an interface that has a number of implementation classes. (The interface and the implementation classes are found in the `atg.userprofiling` package.) Each of the classes implements a different mechanism for enforcing access control. Some of these implementation classes are discussed in the sections below: AccessRightAccessController, GroupAccessController, and RuleAccessController. For additional information about the `AccessController` interface and the classes that implement it, see the *ATG API Reference*.

In addition to the `accessControllers` property, the Access Control Servlet has `accessAllowedListeners` and `accessDeniedListeners` properties. You can use these properties to specify `atg.userprofiling.AccessAllowedListener` and `atg.userprofiling.AccessDeniedListener` components, which are notified when page access is granted or denied.

The Access Control Servlet is enabled by default. You can disable it by setting the `enabled` property of `/atg/userprofiling/AccessControlServlet` to `false`.

The following is an example of an `AccessControlServlet.properties` file:

```
$class=atg.userprofiling.AccessControlServlet

enabled=true

# Nucleus path of the Profile object
profilePath^=ProfileRequestServlet.profilePath

# List of mappings between paths and AccessController objects.  If a
# path refers to a directory, all the documents in that directory and
# its subdirectories will be protected by the given AccessController.
accessControllers=\
  /docs/members=/your/path/MemberAccessController,\
  /docs/members/preferred=/your/path/PreferredMemberAccessController

# List of "access allowed" event listeners
# accessAllowedListeners=

# List of "access denied" event listeners
# accessDeniedListeners=

# The URL to redirect to if access is denied.  If the AccessController
# supplies its own deniedAccessURL, it will overwrite this value.
deniedAccessURL=http://yourserver/noaccess.html
```

AccessRightAccessController

This implementation of the `AccessController` interface performs access control based on access rights. You specify the access rights through the `allowedAccessRightNames` property, which is a List of access right names. If a page's URL is mapped to an `AccessRightAccessController` component (through the Access Control Servlet's `accessControllers` property) , then that component's access rights are used to control access to the page.

Access rights are associated with users through global and organization roles. If a user's role and the `AccessRightsAccessController` associated with the page have at least one access right in common, the user is allowed to access the page.

GroupAccessController

This implementation of `AccessController` performs group-based access control. Two properties, `allowGroups` and `denyGroups`, specify the names of the groups whose members should be allowed or denied access, respectively. A user is allowed access only if he is a member of one of the `allowGroups`, but not a member of one of the `denyGroups`.

If the `allowGroups` property is not specified, all groups are implicitly considered to be "allow" groups. If the `denyGroups` property is not specified, no groups are considered to be "deny" groups. For example, if `allowGroups` is not specified and `denyGroups=Kids,Teenagers`, then everybody but kids and teenagers is allowed access. If, on the other hand, the `denyGroups` property is not specified and `allowGroups=Kids,Teenagers`, then only kids and teenagers are allowed access.

As an example, here is a configuration for a `PreferredMemberAccessController` component that allows access only to members of the `GoldAccounts` group:

```
$class=atg.userprofiling.GroupAccessController

enabled=true
```

```
allowGroups=GoldAccounts
groupRegistry=/atg/registry/RepositoryGroups

# URL to redirect to if access is denied
deniedAccessURL=http://yourserver/preferredAccessOnly.html
```

RuleAccessController

This implementation of `AccessController` performs access control based on a set of rules, specified via the service's `ruleSetService` property. For example, suppose there is a `RuleSetService` named `FemaleRuleSetService`, configured with the following rule set:

```
<ruleset>
 <accepts>
 <rule op=eq>
    <valueof target="Gender">
    <valueof constant="female">
 </rule>
 </accepts>
</ruleset>
```

Set the `ruleSetService` property of the Access Controller to point to `FemaleMembersRuleSetService`. The user will be allowed access only if she is in the `Female` profile group. Here is the example configuration:

```
$class=atg.userprofiling.RuleAccessController

enabled=true

# Rules used to determine whether access should be allowed
ruleSetService=/your/path/rules/FemaleRuleSetService

# URL to redirect to if access is denied
deniedAccessURL=http://yourserver/femaleAccessOnly.html
```

Note that when the rules are evaluated, the user's `Profile` object is used to resolve the target expressions. Note also that the rules must evaluate to a Boolean. The rules used by a `RuleAccessController` component use the same syntax as those used for content targeting. See the Creating Rules for Targeting Content and the Setting Up Targeting Services chapters for more information.

The optional `sourceMap` property, if provided, is used to resolve any bean expressions in the access control rules. If no such expressions occur in the rules, you can leave this property as null.

Controlling Anonymous User Access

When you create a profile group, you specify a set of rules that allows a group of registered users to access some part of your Web site. The `AccessController` allows access by anonymous users unless your profile group rule explicitly prevents it. To avoid specifying that anonymous users should be excluded in every rule you write, you can set the `denyAnonymousUsers` property of the `GroupAccessController` or `RuleAccessController` class to `true`. If `denyAnonymousUsers` is `false`, all anonymous users are allowed access even if they are in a group specified by `denyGroups`.

If you are persisting anonymous profiles, `denyAnonymousUsers` does not control whether or not persistent anonymous profiles are allowed access, even if you set this property to `true`. This is because the Personalization module treats persistent anonymous profiles as actual profiles and creates a repository item for each persistent anonymous profile. You should expect that persistent anonymous profiles will be allowed access unless you specifically use a rule that denies them access.

Configuring Derived Properties that Calculate Time and Date

To any GSA repository template, including the user profile repository, you can add derived properties that allow you to compute and store values related to time and date. The following derived properties are available for this purpose:

- TimeInterval
- YearMonthDay
- DaysBeforeAnnualEvent

For general information on how derived properties work, see the *ATG Repository Guide*.

TimeInterval

The `TimeInterval` derived property can be used to calculate the interval between two specified times. The feature is useful in situations in which you need to compute an unknown date or amount of time, or to calculate the length of time between two dates. For example, you could add a custom property called `age` to the user profile and use the time interval derived property to calculate its value by comparing the user's date of birth to today's date.

`TimeInterval` derived properties are readable and queryable. Note, however, that the feature does not support comparison queries on a derived property that is calculated using two date properties. See the second example at the end of this section.

The `TimeInterval` derived property is implemented by the property descriptor `atg.repository.dp.TimeIntervalPropertyDescriptor`, which uses the supporting derivation method `atg.repository.dp.TimeInterval`.

The interval is an integer that can be represented as follows:

```
        int timeStampDiff(int interval, Date date1, Date date2)
```

where the integer is the number of `intervals` (which can be years, quarters, months, weeks, hours, days, minutes, seconds, milliseconds) by which `date2` is greater than `date1`.

The derived property has three attributes, which are described below.

Attribute	Type	Required	Description
`timeStampInterval`	String	Yes	Represents the returning interval between the corresponding times. Must be one of `year`, `quarter`, `month`, `week`, `day`, `hour`, `minute`, `second`, or `millisecond`.
`datePropertyName1`	String	No	The name of the property to use as the `date1` argument. Defaults to the current time if not specified.
`datePropertyName2`	String	No	The name of the property to use a the `date2` argument. Defaults to the current time if not specified.

The following example shows how to configure the `TimeInterval` derived property in a GSA repository definition to calculate the age of a user:

```
<property name="age"
          data-type="int"
          writable="false"
          property-type="atg.repository.dp.TimeIntervalPropertyDescriptor">
   <attribute name="timeStampInterval" value="year"/>
   <attribute name="datePropertyName1" value="dateOfBirth"/>
</property>
```

The next example illustrates the use of the time interval derived property for computing the number of days between two date properties.

```
<property name="daysOfEmployment"
          data-type="int"
          writable="false"
          property-type="atg.repository.dp.TimeIntervalPropertyDescriptor">
   <attribute name="timeStampInterval" value="day"/>
   <attribute name="datePropertyName1" value="startDate"/>
   <attribute name="datePropertyName2" value="endDate"/>
</property>
```

Important: In some cases, time interval derived properties cannot be included in comparison queries. This restriction typically applies when the query involves a property derived from two dates, such as the daysOfEmployment example above. Queries that compare properties derived from two dates with another value (for example, "daysOfEmployment > 90")are not supported in RQL.

YearMonthDay

The YearMonthDay derived property represents a date that is derived from the values of year, month, and day fields. It is implemented by the property descriptor atg.repository.dp.YearMonthDayPropertyDescriptor, which uses the supporting derivation method atg.repository.dp.YearMonthDay.

The YearMonthDay property is readable, writable, and queryable. It has the following three attributes:

Attribute	Type	Required	Description
yearPropertyName	string	yes	The name of the property that represents the year field.
monthOfYearPropertyName	string	yes	The name of the property that represents the month field.
dayOfMonthPropertyName	string	yes	The name of the property that represents the day field

All dates use the default time zone of the JVM.

The following example shows how to configure the time interval derived property in a GSA repository definition to calculate a user's date of birth:

```
<property name="dob"
          data-type="date"
          writable="true"
          property-type="atg.repository.dp.YearMonthDayPropertyDescriptor">
   <attribute name="yearPropertyName" value="dobYearField"/>
   <attribute name="monthOfYearPropertyName" value="dobMonthOfYearField"/>
   <attribute name="dayOfMonthPropertyName" value="dobDayOfMonthField"/>
</property>
```

Note that the date computed for this derived property does include the time, which is always 12:00 AM. However, any comparison queries performed with this property ignore the time and use the date only.

DaysBeforeAnnualEvent

The DaysBeforeAnnualEvent derived property represents the number of days before an annual event occurs. Note that the value of this property is cyclical. It starts at a positive integer less than 365 and decreases each day to zero. Once zero is reached, the value is reset to 365 and starts decreasing again the next day.

The DaysBeforeAnnualEvent derived property is implemented by the property descriptor atg.repository.dp.DaysBeforeAnnualEventPropertyDescriptor, which uses the supporting derivation method atg.repository.dp.DaysBeforeAnnualEvent. The property is readable and queryable but not writable.

The property has two attributes, which are described below:

Attribute	**Type**	**Required**	**Description**
monthPropertyName	string	yes	The name of the property that represents the month field.
dayOfMonthPropertyName	string	yes	The name of the property that represents the day field.

The following example shows how to configure the DaysBeforeAnnualEvent derived property in a GSA repository definition to calculate the number of days before a user's birthday.

```
<property name="daysBeforeBirthday"
      data-type="int"
      writable="false"
      property-type="atg.repository.dp.DaysBeforeAnnualEventPropertyDescriptor">
  <attribute name="monthPropertyName" value="dobMonthOfDayField"/>
  <attribute name="dayOfMonthPropertyName" value="dobDayOfMonthField"/>
</property>
```

Managing Preview User Swapping

ATG Content Administration, ATG Merchandising, and ATG Outreach include preview features that allow internal users to test content against a sample user profile before deploying it. Preview profiles are stored in a ProfileAdapterRepository component on the management server, and they reference the dps* tables in the versioned database. For a diagram showing the profile repositories used by the ATG platform, see the *ATG Business Control Center Administration and Development Guide*.

Preview is initiated when an internal user clicks a custom Preview button in the relevant UI. The user selects a preview profile to use, and a page containing the previewed content appears in a separate browser window. (For information on how to add a Preview button and create a Web application containing the preview page or pages, refer to the *ATG Business Control Center Administration and Development Guide*.)

To allow a user to preview content as another user, the ATG preview code must swap the logged-in user's profile for the preview profile for the duration of the request. To determine whether swapping should occur for a given Web application, the preview code checks the current request for the value of the `atg.preview` context parameter. If this parameter is set to false , the profile swap does not occur. Note that the `atg.preview` parameter is set to false for all ATG's management applications.

In some cases it is necessary or desirable to force preview swapping in a Web application that otherwise has the feature turned off. For example, preview swapping is disabled for ATG Outreach, but it is required by the application's e-mail preview feature. For such cases, you can specify a `forcePreviewParam` query parameter within the preview URL that tells the preview code to ignore the `atg.preview` setting.

After determining whether preview swapping should occur, the preview code looks for a specific request query parameter that indicates the repository ID of the profile selected for the preview.

Note: In addition to adding a Preview button and creating the pages to use for the preview, you must run the application on the management server with the `–layer Preview` switch for preview features to be available. See the *ATG Business Control Center Administration and Development Guide* for more information.

PreviewProfileRequestProcessor Component

The `ProfileRequestServlet` contains a `profileRequestProcessors` property, which is an array of a `ProfileRequestProcessor` interface. Preview user swapping is implemented through the `/atg/userprofiling/PreviewProfileRequestProcessor` component, which is an implementation of `atg.userprofiling.ProfileRequestProcessor`.

The table below shows the properties of the `PreviewProfileRequestProcessor` component.

Property	Type	Required	Description
`previewCriteria`	`atg.userprofiling.PreviewCriteria`	No	An implementation of `atg.userprofiling.PreviewCriteria`.
`previewRepository`	Repository	Yes	The repository that contains the preview users.
`previewItemType`	String	No	The item type of the preview users to use. The default item type for the repository is used if this property is not set.
`repositoryItemIdParam`	String	No	The name of the repository item ID query parameter
`forcePreviewParam`	String	No	The name of the force preview query parameter

The default values for these properties are shown below:

```
$class=atg.userprofiling.PreviewProfileRequestProcessor

   previewCriteria=/atg/userprofiling/DefaultPreviewCriteria
   previewRepository=/atg/userprofiling/ProfileAdapterRepository
   previewItemType=user
```

Preventing Profile Swapping in Non-Preview Web Applications

As described above, the preview mechanism uses a context parameter called `atg.preview`, which is set in an application's `web.xml` file, to determine whether to initiate a preview user swap for a given request.

```
<context-param>
  <param-name>atg.preview</param-name>
  <param-value>false</param-value>
</context-param>
```

Preview swapping can occur only for applications on the management server that are run with the –`layer Preview` switch. However, the `atg.preview` parameter must be set to false for any Web applications that do not specifically require preview user swapping. If `atg.preview` is not set to false, the preview system assumes that any request to the given Web application is a preview request. Preview requests do not use transient profiles, so `atg.preview` should be set to false for any non-previewable apps on the management server to prevent 403 errors occuring if the user is not logged in.

Note that the swap will still occur, regardless of the `atg.preview` setting, if the `forcePreviewParam` parameter is specified in the request query. The following example is from `editSendEmail.jsp` in ATG Outreach.

```
<c:url context="${campaignConfiguration.campaignsUIRoot}"
  value="/preview.jsp" var="previewURL">
    <c:param name="targetAssetType" value="${previewAssetTypeParam}"/>
    <c:param name="targetVirtualPath" value="${previewPathParam}"/>
    <c:param name="${previewParams.repositoryItemIdParam}"
      value="${previewUserIdParam}"/>
    <c:param name="${previewParams.forcePreviewParam}" value="true"/>
</c:url>
```

5 Working with the Dynamo User Directory

ATG repositories are collections of repository items, which are JavaBean components that correspond to an object in an underlying data store. A user directory is a way of organizing the information in a repository by representing repository items as objects in a graph or tree. Objects such as users, organizations, and the roles that those users and organizations possess are displayed as `Principal` objects in the Dynamo User Directory. Displaying these repository items in a user directory allows you to manipulate them in flexible ways within the security system. You can use all `Principal` objects obtained from a user directory directly in Access Control Lists and in entries managed by security domains compatible with that directory.

The Personalization module uses one or more profile repositories to store information about Web site visitors. Each profile in the repository contains information about a user such as the user's name and address. The Dynamo User Directory allows you to manage user profiles by capturing relationships between users and the organizations to which they belong. Web site visitors may have characteristics that come from multiple sources such as the company they work for, the Web browser they use, their age, gender, or other sources. User directories support multiple types of individuals by placing user profiles in an organizational graph based on rules that you specify.

The Dynamo User Directory allows you to assign access rights to repository items.

The Personalization module also organizes information about ACC users in a user directory. You can use the Admin SQL implementation of the Dynamo User Directory to store security information and Access Control Lists for ACC users. For more information about how the security system interacts with the Admin SQL repository, see *Secured Repositories* in the *ATG Repository Guide*.

This chapter contains the following sections:

User Directory Architecture
Describes the basic concepts of a user directory.

User Directory Security
Describes how you can use the Dynamo User Directory to authenticate users and assign access rights to repository items.

User Directory API
An overview of the user directory API.

Dynamo User Directory Implementations
Describes the Dynamo User Directory that is configured out-of-the-box with the Personalization module.

User Directory Architecture

In the Dynamo User Directory, users may belong to **organizations**, and organizations may belong to parent organizations. For example, you could have an organization that represents your company and children organizations that represent the various departments within the company. An organization can be any object you'd like to represent in the user directory. You don't have to assign users to an organization, although you can use organizations to group users. In the user directory UI, users who are assigned to an organization are called **members** of that organization.

Users, organizations, and roles are all considered examples of **principals**. Principals are logical identities that may be granted or denied access rights in various ATG security domains. In this way, user directories provide a uniform way for ATG security models to look at a user and to understand the user from a security point of view.

You can assign **roles** to users and to organizations. Roles define actions that users can take or positions that they hold. For example, you can assign someone the role of "buyer" within a specific organization. Buyers may have access to certain repository items, and you can easily group together all users who have the role "buyer." You could also assign someone the role "VP of Sales." This role is a little different because you probably would assign this role to only one user. You can specify access rights for a role. For example, perhaps the role "VP of Sales" has the ability to view and to edit the profiles of all the users who have the role "buyer."

In addition, you can assign roles to organizations. For example, you could assign an organization in the user directory the role of "Partner," and you can specify that partners have access to certain repository items. Users who belong to an organization can inherit that organization's role. Roles can be one of two types:

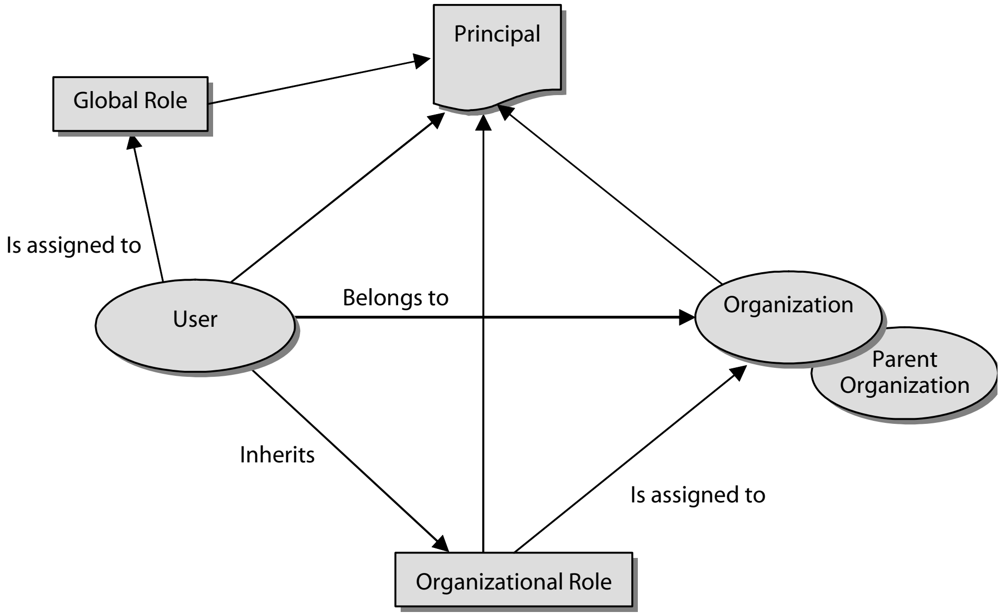
- **Global roles** are roles that you can assign to any user or organization. If you assign a global role to an organization, all users who belong to that organization and any child organizations inherit that role. Global roles are the only kind of roles that you can organize in role folders.

  A **role folder** is a collection of child roles and role folders that serves as a organizing element for the space of global roles. You cannot assign any security permissions to a role folder because it is not a `Principal` object. Some directory implementations may not support the creation of any folders but the root folder. For example, if you do not have the Personalization module installed, a user directory manages the organization of your internal ATG product users, but it does not have the capacity to manage profiles. In this case, a single role folder encapsulates the roles assigned to internal users.

- **Organizational roles** are roles that a user plays in the context of a specific organization. Organizational roles are also called **relative roles**.

  An organizational role has a property called **function** that allows you to establish a connection among similar organizational roles. For example, you could have several different roles, Marketing Director, Human Resources Director, and Customer Service Director, that have the same basic function, "director." By specifying the same value for the `function` property of each role, you can track the connection among the roles and write custom code that makes use of it.

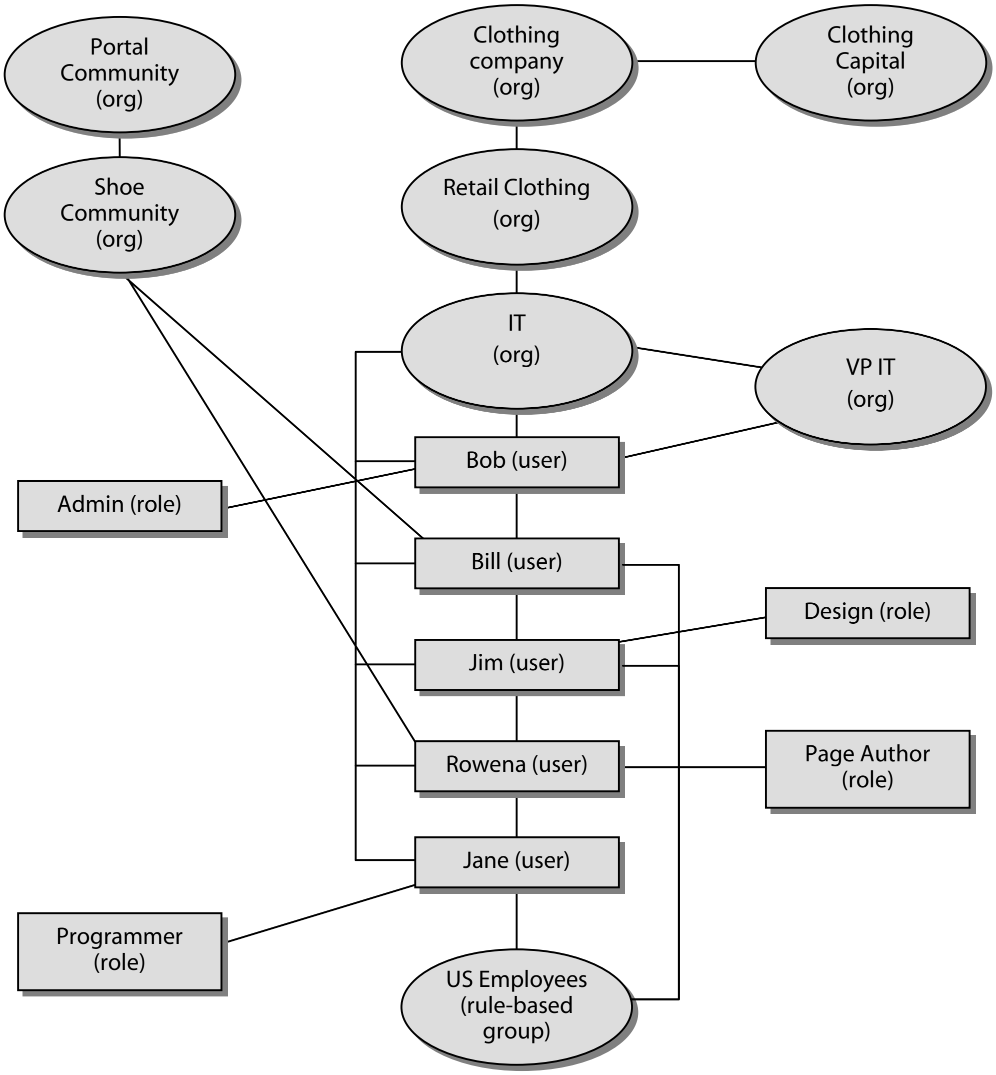
The following figure illustrates the relationships within a user directory:



In this figure, the `User`, `Role`, and `Organization` objects are all `Principal` objects. In this example, the user is assigned a global role and inherits the `OrganizationalRole`, which is a role that pertains to a specific organization. The organization may inherit characteristics from its parent organization. A user directory doesn't need to support all the possibilities detailed in this diagram. It can contain any subset that you decide upon. For example, the concept of relative roles might be absent from a user directory.

While users and organizations are usually `RepositoryItem` objects, groups and roles are usually objects of other types. You can determine the `PrincipalType` of each `Principal` object. Available types are `user`, `organization`, and `role`. The determination of a `PrincipalType` of a principal object is internal to the user directory. You can, however, examine a `Principal` object within the user directory and determine that it is a `RepositoryItem` or another dynamic bean and modify that object accordingly.

The following figure is an example of an organizational scheme:



Creating Organizations and Roles

After you define the profile properties that make up organizations and roles, you can use the ATG Business Control Center to create the organizations and roles required for your Web application. For instructions, refer to the *ATG Business Control Center Administration and Development Guide*.

Important: You can also use the ACC to create organizations and roles. Note however that the ACC that is installed with the ATG platform is set up by default to point to the external profile repository, so any organizations or roles you create through this application are available only to external user profiles. Note: If you do use the ACC to add or edit user directory items, it is highly recommended that you use the Organizations, Roles, and Users screens from the People and Organizations menu. If you perform the same tasks through the profile repository editor (People and Organizations > Profile Repository), the changes you make will not appear in other areas of the ACC unless you restart the server. For example, you could add a role to someone's profile through the profile repository editor, but the ACC would not pick up the change until you restarted the server. The same behavior does not occur if you use the Organizations, Roles, or Users screens.

User Directory Security

The Dynamo User Directory functions both as an organizational tool and as a security tool. As an administrator, you can allow or deny access to specific repository items and to specific properties of those repository items. The Dynamo User Directory accesses two repositories to store information and to implement the security system that controls repository items. These repositories are the profile repository and the Admin SQL repository.

ATG products work with two types of repositories: "concrete" repositories, which actually hold data, and "secure" repositories, which perform security checking. Rather than adding security checking directly to a concrete repository, a secure repository wraps a concrete repository. This secure repository has the same API, behavior, and data as the underlying concrete repository. In general, invoking a method on a secure repository works the same as invoking the same method on a concrete repository, but with one essential difference: the secure wrapper applies security checks as needed, before and after calling the concrete repository to do the work of retrieving items from the database. This security system has two major benefits:

- Nucleus components that require security checking access the secure repository. Components that don't require security checking skip the secure repository and directly access the concrete repository.
- All repository implementations, including any that you add in the future, can use this secure repository wrapper. This is much easier than having to add security as a feature in each new repository implementation.

Configuring a User Authority

When a user logs into a Web site or into an ATG product, a `UserAuthority` object authenticates the user's login name and password. A user authority determines identities throughout the ATG security system. The Personalization module provides several user authorities that are configured out-of-the-box. The `/atg/userprofiling/ProfileUserAuthority` component controls security for the profile repository user directory.

For more information about the `UserAuthority` object, see *Security Services Classes and Interfaces* in the *Managing Access Control* chapter of the *ATG Programming Guide*.

Setting ACC and Object Access Rights through Access Control Lists

You can assign ACL-based access rights to internal and external users. Assigning access rights adds or removes the user's ID from the Access Control List (ACL) that belongs to every object and every property in the repository. In most cases you can set object access rights through the ACC and the ATG Business Control Center. You can also set access rights for specific properties of objects, but the UIs do not support this option. To set access rights for a specific property, you must change the access rights defined for that property in the repository definition XML file.

For information on using the ACC to set access rights for objects such as scenarios and workflows, refer to the documentation for those objects (for example, Setting Up Security Access for Scenarios). For information on using ACLs to control access for ACC users, refer to *Configuring Access Privileges* in the *ATG Programming Guide*.

For information on using access rights to secure access to assets within the ATG Business Control Center, refer to the *ATG Business Control Center Administration and Development Guide*.

The basic types of access rights are as follows:

`create`	Controls the ability to create a new repository item with an item-descriptor. To add the new item to the repository, you must also have WRITE access to the item-descriptor.
`read`	Controls read only access to a repository item.
`write`	Controls the ability to add a repository item to a repository item-descriptor, or to change the contents of a repository item or a property in a repository item. If the WRITE access right is granted for a repository item-descriptor, it does not affect the ability to update a repository item, only the ability to add new items.
`list`	Controls the ability to query the repository for a specific repository item. If a user does not have LIST rights on a repository item, a query of the repository will not return that item. The item may still be available by asking for it specifically. Use the READ access right to control general access to the repository item.
`delete`	Controls the ability to remove a repository item from a repository item descriptor. In order to delete an item you must also have DESTROY access for that item.
`destroy`	Controls the ability to remove a repository item from the repository, destroying its contents. Note that most Secured Repositories also require DELETE access on the repository item-descriptor.
`read-owner`	Controls the ability to read who owns an item
`write-owner`	Controls the ability to change the owner of an item.
`read-ACL`	Controls the ability to read the access control list for an item. This access right is automatically granted to the owner of a repository item.
`write-ACL`	Controls the ability to change the access control list of a repository item. This access right is automatically granted to the owner of a repository item.

Using Roles for Access Control

As described in the previous section, you can use principals such as roles to secure access to the ACC, to the ATG Business Control Center, and to repository objects. You can also use global roles in combination with access rights to secure access to various parts of your Web application UI. For example, you might have several pages on your sites that can be used to edit a customer’s profile, and you want to make these available only to customer service representatives. You can set up a global role called Customer Service

Rep and add an access right to that role that allows entry to the specified site pages. Any user who is assigned that role, either directly or though membership of an organization, can access the appropriate pages.

As well as assigning access rights to global roles to secure the pages of a Web UI, you can also use access rights with roles to perform access control of repository objects (for example, workflows). For example, you could assign access rights to an organizational role, and configure a security check that compared the organization associated with an object against the organization associated with the organizational role. As described earlier, you can control access to objects by assigning an access control list (ACL) that can include a role as a principal. However, doing so requires you to set the ACL individually for each object. Assigning access rights to roles as described in this section allows you to design a security system in which access for multiple objects can be set at once. ATG Service uses an implementation similar to this to secure access to solutions.

The rest of this section contains the following information:

- Using Global Roles to Control Access to UI Pages
- Adding Access Rights to a Role
- Using Roles as Templates for Adding Access Rights

Using Global Roles to Control Access to UI Pages

UI access rights that you assign through global roles work as follows:

1. In the internal user profile repository, you create an `accessRight` repository item that corresponds to some functional aspect of your sites that you want to secure. For example, you could set up an access right item called Edit Customer Profiles to be applied to any page that allows users to change a customer's profile information.
2. You configure an `AccessRightAccessController` component (class `atg.userprofiling.AccessRightAccessController`) that specifies the Edit Customer Profiles repository item you created in step 1.
3. You configure an `AccessControlServlet` component (class `atg.userprofiling.AccessControlServlet`) whose `accessControllers` property specifies the pages to which the Edit Customer Profiles access controller should apply.
4. In the ATG Control Center, you add the Edit Customer Profiles access right to one or more global roles, and you assign those roles to individual users. When a user attempts to display any of the pages specified in the `AccessControlServlet` as requiring the Edit Customer Profile access right, a security check is performed to verify that the user's profile contains the appropriate role.

The following examples show typical contents of the property files for the two components you set up:

`EditCustomerProfilesAccessRightController.properties`:

```
$class=atg.userprofiling.AccessRightAccessController
$scope=global
```

```
accessRightsPropertyName=accessRights
accessRightNamePropertyName=name
allowedAccessRightNames=edit_customer_profiles

deniedAccessURL=/MY-APP/en/accessdenied.jsp
```

Note: The deniedAccessURL specifies the page to display if the user fails the security check. This property can also be specified in the AccessControlServlet (see below), but defining it in the AccessController helps to eliminate conflicts in cases where more than one application is running.

AccessControlServlet.properties:

```
$class=atg.userprofiling.AccessControlServlet

profilePath^=ProfileRequestServlet.profilePath

enabled=true

# List of mappings between paths and AccessController objects.  If a
# path refers to a directory, all the documents in that directory and
# its subdirectories will be protected by the given AccessController.

accessControllers+=\
  /MY-WEB-APP/en/edit_address.jsp=/atg/mymodule/
   EditCustomerProfileAccessRightController,\
  /MY-WEB-APP/en/edit_username.jsp=/atg/mymodule/
   EditCustomerProfileAccessRightController

# Default deniedAccessURL to use if an AccessController
# doesn't supply one
deniedAccessURL=

#loggingDebug=true
```

The item descriptor for the accessRights repository item is shown below:

```
<item-descriptor name="accessRight" sub-type-property="type"
id-space-name="internalAccessRight"
display-name-resource="itemDescriptorAccessRight"
  display-property="name" item-cache-size="1000" query-cache-size="1000"
version-property="version">
  <attribute name="resourceBundle"
   value="atg.userprofiling.InternalUserProfileTemplateResources"/>

    <table name="dpi_access_right" type="primary"
      id-column-name="access_right_id">
      <property name="name" column-name="name" data-type="string"
```

```
            required="true"
            display-name-resource="rightName">
            <attribute name="propertySortPriority" value="10"/>
        </property>
        <property name="version" column-name="version" data-type="int"
            writable="false" expert="true" display-name-resource="version">
            <attribute name="propertySortPriority" value="60"/>
        </property>
        <property name="description" column-name="description" data-type="string"
            display-name-resource="rightDescription">
            <attribute name="propertySortPriority" value="30"/>
        </property>
        <property name="scope" column-name="right_scope" required="true"
            data-type="enumerated"
            display-name-resource="scope">
            <attribute name="propertySortPriority" value="40"/>
            <option value="global" code="1"/>
            <option value="organization" code="2"/>
        </property>
        <property name="type" column-name="right_type" required="true"
            data-type="enumerated" default="generic"
            display-name-resource="rightType">
            <attribute name="uiwritable" value="false"/>
            <attribute name="propertySortPriority" value="80"/>
            <option value="generic" code="1"/>
        </property>

      </table>
    </item-descriptor>
```

Adding Access Rights to a Role

As indicated elsewhere in this chapter, inheritance plays an important part in the way roles and organizations work. Access rights that you grant to a role that is assigned to a parent organization are automatically inherited by any dependent organizations. Before you assign access rights to roles, make sure you have a clear understanding of the parent/child relationships among the entities in your user directory.

To use the ACC to add access rights to a role, complete the following steps:

1. Display the Roles window (select People and Organizations > Roles).
2. In the list in the left pane, select the role (either global or organizational) to which you want to add access rights. For information on the difference between the two types of role, see User Directory Architecture.
3. Display the Profile tab for the role.
4. Click the button in the Direct Access Rights field. The Direct Access Rights dialog box appears. **Note:** If the field does not appear, check that you have started ATG 10.0.2 with a module that supports adding access rights through the ATG Control Center.

5. Click Add. The New Item dialog box appears.
6. Specify Items of Type Access Right as the query, and click List. All existing access rights appear.
7. Select the access rights you want to add for this role.
8. Click OK to return to the Direct Access Rights dialog box.
9. Click OK again.

Note: To see a brief description of an access right, complete the steps in the procedure as far as step 8. If you double-click the name of an access right in the Direct Access Rights dialog box, its properties, including the Description field, are displayed.

Using Roles as Templates for Adding Access Rights

In cases where you need to define access rights for a large number of roles, you can save time by adding the rights to one role and then using that role as a template. Example: You have 20 organizations called Product 1, Product 2, Product 3, and so on. You have forty corresponding organizational roles for Author and Reviewer (for example, Product 1 Author, Product 1 Reviewer, Product 2 Author, Product 2 Reviewer). You want to assign the same access rights to all the Product *n* Reviewer roles. To do so quickly, you can set up a role called Reviewer, define the access rights for it, and then use it as a template for the appropriate organizational roles.

Rights that you add through template roles are cumulative with rights added through the Direct Access Rights field described in the previous section. This behavior allows you to use a combination technique where you can assign shared rights through role templates and define rights that are unique to a role through the Direct Access Rights field.

You can use both global and organizational roles as templates for any type of role. Note that only the role's access rights are inherited by the non-template role; no other properties of the template role are acquired.

To use a global role as a template for adding access rights:

1. Add the required access rights to the role you want to use as a template. Follow the procedure in the previous section, Adding Access Rights to a Role.
2. Select or create the first role to which you want to apply the template, and display the Profile tab for that role.
3. In the `templateRoles` field, click the button. The Template Roles dialog box appears.
4. Click Add. The New Item dialog box appears.
5. Specify Items of Type Role as the query, and click List. All roles in your system appear.
6. Select the role you want to use as the template. (Note that you can select multiple roles if you want to add access rights from more than one template role.)
7. Click OK to return to the Template Roles dialog box.
8. Click OK.
9. Repeat steps 2 to 8 for other non-template roles.

10. Select File > Save when you have finished.

User Directory API

The Dynamo User Directory is an implementation of the User Directory API. This section lists the specific packages, interfaces, and classes that you can use to extend the core User Directory API with any custom code that your application requires. For example, you can extend any of the `Principal` objects to add properties that fit your business model.

The User Directory API includes the following packages:

- `atg.userdirectory`
- `atg.userdirectory.droplet`
- `atg.userdirectory.repository`

atg.userdirectory

The `atg.userdirectory` package contains the following interfaces, which represent each of the different types of objects that can exist in a user directory:

- `atg.userdirectory.User`
- `atg.userdirectory.Organization`
- `atg.userdirectory.Role`
- `atg.userdirectory.RelativeRole` (represents organizational roles)
- `atg.userdirectory.DirectoryPrincipal`
- `atg.userdirectory.OrganizationalEntity`
- `atg.userdirectory.RoleFolder`

Note that each of these interfaces contains methods that you can use to search for items in a user directory. These methods provide alternative and in some cases more flexible techniques for sorting user directory items than the implementations in the `atg.userdirectory.droplet` package described in the next section. For example, the `atg.userdirectory.organizations` interface contains methods for finding all users associated with a directory and for sorting them by first name, last name, login ID, or e-mail address.

In addition to the interfaces described above, the `atg.userdirectory` package contains the interface `atg.userdirectory.UserDirectory`, which manages the organizational tree, and the following additional classes:

- `atg.userdirectory.RoleNotAssignableException`
- `atg.userdirectory.DirectoryModificationException`
- `atg.userdirectory.UserDirectoryUserAuthority`

For information about the `atg.userdirectory` package, refer to the *ATG API Reference*.

atg.userdirectory.UserDirectoryUserAuthority

A user authority (an implementation of the `atg.security.UserAuthority` interface) produces Persona objects that can be used as part of a security model to identify users and associate them with any roles that they may have. The `atg.userdirectory.UserDirectoryUserAuthority` class is a user authority that is designed for creating Persona objects specific to a user directory.

The `UserDirectoryUserAuthority` class supports the following items for identity lookup:

- `user`
- `org`
- `role`
- `login`
- `orgpath`
- `rolepath`

These identities can be included as PRINCIPAL_TYPE access control entries in Access Control Lists and then extracted, for example by an ACL parser. Access Control Entries use the following format:

```
UD_NAME '$' PRINCIPAL_TYPE '$' UD_PRINCIPAL_KEY
```

where UD_NAME is the name of the user directory (for example, Profile), and UD_PRINCIPAL_KEY is the primary key used for looking up the principal in the given user directory. The following table gives example access control entries for the identities that the `UserDirectoryUserAuthority` class supports:

PRINCIPAL_TYPE	UD_PRINCIPAL_KEY	Example
`user`	Profile ID	`Profile$user$9462`
`org`	Profile ID	`Profile$org$341`
`role`	Profile ID	`Profile$role$732168`
`login`	Login name	`Profile$login$Mary`
`orgpath`	The path to the organization	`Profile$orgpath$/MyCorp/Sales`
`orgrole`	The organizational role, by organizational path and function name	`Profile$orgrole$/MyCorp/Sales/Manager`
`rolepath`	The path to the role	`Profile$rolepath$/designer`

For more information on access control entries, refer to *ACL Syntax* in the *ATG Repository Guide*.

The `/atg/dynamo/security/UserAuthority` component is the default implementation of the `UserDirectoryUserAuthority` class. Use the `PrincipalResolver` interface and the addPrincipalResolver() method in the UserDirectoryUserAuthority API to extend the UserDirectoryUserAuthority.

atg.userdirectory.droplet

This package contains the API for the following form handlers and ATG Servlet Beans:

- atg.userdirectory.droplet.CreateOrganizationFormHandler: Use this form handler to create new organizations in the user directory. The Motorprise demo for ATG Business Commerce contains an implementation of this form handler, which it uses to create business units. For more information, refer to *Creating Business Units and Roles* in the *ATG Business Commerce Reference Application Guide*. If you have ATG Business Commerce, you can also look at the JSP code in the page business_unit_new.jsp, located by default in <ATG10dir>\MotorpriseJSP\j2ee-apps\motorprise\web-app\en\admin.
- atg.userdirectory.droplet.HasFunction: Use to query whether or not a user is associated with a particular organizational role function value. If the user has the function, the true oparam is rendered. For an example, refer to *Verifying Admin Access* in the *ATG Business Commerce Reference Application Guide*.
- atg.userdirectory.droplet.TargetPrincipalsDroplet: Use to retrieve a list of organizations that are associated with roles containing a specific function value.
- atg.userdirectory.droplet.UserListDroplet: Retrieves a list of users associated with a specific organization.
- atg.userdirectory.droplet.ViewPrincipalsDroplet: Retrieves a list of Principals (either organizations or roles) for a specific user.

For more information about the atg.userdirectory.droplet package, refer to the *ATG API Reference*.

For information on additional ways to sort the results of queries performed on a user directory, see the previous section, atg.userdirectory.

atg.userdirectory.repository

This package contains the interface atg.userdirectory.repository.RepositoryUserDirectory, which handles conversion between repository items and the Principals that they represent. It also associates each Principal type with a corresponding Repository View in the repository, allowing you to perform queries against these views. The Repository Views can also be used for adding and removing items that correspond to principals of different types.

For more information, refer to the *ATG API Reference*.

Dynamo User Directory Implementation

The Dynamo User Directory is the standard user directory implementation that is configured out-of-the-box with your ATG server. This section describes how you can configure the Dynamo User Directory through the profile repository definition file. It contains the following information:

- User Directory Repository Definition
- Standard User Directory Definition File
- Configuring User Directory Components
- Caching and the User Directory

User Directory Repository Definition

The profile repository definition file, `userProfile.xml`, defines item descriptors and properties that belong to organizations, both types of roles, and other aspects of the user directory. You can extend these template definitions as required, adding the properties that you want to be able to specify for organizations and roles in the ATG Business Control Center and the ACC.

The `userProfile.xml` file is a combination of several `userProfile.xml` files, one defined for each ATG module you install. For more information about which `userProfile.xml` files make up your user directory repository definition, see XML File Combination and the User Profile Repository Definition in the Extending the Standard User Profile Repository Definition section of this guide.

The Personalization module's base profile repository definition file defines the following item descriptors for the user directory:

Item Descriptor Name	Description
`role`	Defines global roles
`organizationalRole`	Defines organizational roles
`organization`	Represents a Personalization module organization
`genericFolder`	Defines the folders in which you can place organizations
`roleFolder`	Defines the folders in which you can place global roles

Some of these item descriptors use **inheritance** to define a hierarchical relationship with other user directory entities; for example, the item descriptor `organizationalRole` inherits attributes from the item descriptor `role`, as defined in these lines from the Personalization module's default `userProfile.xml` file:

```
<item-descriptor name="organizationalRole" super-type="role"
       sub-type-value="organizationalRole"
```

For more information on item descriptor inheritance, refer to the *SQL Repositories* chapter in the *ATG Repository Guide*.

Note that, in addition to setting up inheritance among item descriptors, or as an alternative to it, you can define **derived properties**, which allow one repository item to derive property values from another repository item or from another property in the same repository item. This feature is used extensively for the User item descriptor in the ATG demos; for example, in the Motorprise demo site for B2B Commerce, the User item descriptor includes properties that it derives from the user's parent organization as shown in the following lines from the Motorprise version of the `userProfile.xml` file:

```
<property name="defaultPaymentType" item-type="credit-card"
 display-name-resource="defaultPaymentType"
           category-resource="categoryBillingShipping">
   <derivation override-property="myDefaultPaymentType">
     <expression>parentOrganization.defaultPaymentType</expression>
   </derivation>
   <attribute name="resourceBundle"
    value="atg.b2bcommerce.UserProfileTemplateResources"/>
   <attribute name="propertySortPriority" value="30"/>
 </property>
```

For more examples and for detailed information on derived properties, refer to the *SQL Repositories* chapter in the *ATG Repository Guide*.

Standard User Directory Definition File

The following code sample shows the section of the base `userProfile.xml` file, included in the Personalization module, that defines the item-descriptors used by the Dynamo User Directory. (Note that the sample shows only an excerpt and not the complete `userProfile.xml` file.)

```
<!--  Roles are the base class that organizational roles inherit from. -->

<item-descriptor name="role" sub-type-property="type" version-property="version"
       display-property="name" display-name-resource="itemDescriptorRole"
       sub-type-value="role" default="false" content="false" folder="false"
       use-id-for-path="false" hidden="false" expert="false" cache-mode="simple">
  <attribute name="resourceBundle"
   value="atg.userprofiling.UserProfileTemplateResources" />
  <table name="dps_role" type="primary" id-column-name="role_id">
    <property name="type" data-type="enumerated" expert="true"
     display-name-resource="type" default="role" required="false"
     readable="true" writable="true" queryable="true" hidden="false"
     cache-mode="inherit">
      <attribute name="useCodeForValue" value="false" />
      <option value="role" code="2000" />
      <option value="organizationalRole" code="2001" />
      <attribute name="propertySortPriority" value="50" />
    </property>
    <property name="version" column-name="version" data-type="int"
     writable="false" expert="true" display-name-resource="version"
     required="false" readable="true" queryable="true" hidden="false"
     cache-mode="inherit">
      <attribute name="propertySortPriority" value="60" />
    </property>
    <property name="name" column-name="name" data-type="string" required="true"
     display-name-resource="name" readable="true" writable="true"
     queryable="true" hidden="false" expert="false" cache-mode="inherit">
      <attribute name="propertySortPriority" value="10" />
    </property>
    <property name="description" column-name="description" data-type="string"
     display-name-resource="description" required="false" readable="true"
     writable="true" queryable="true" hidden="false" expert="false"
     cache-mode="inherit">
      <attribute name="propertySortPriority" value="20" />
    </property>
  </table>
  <property name="relativeTo" hidden="true" required="false" readable="true"
   writable="true" queryable="true" expert="false" cache-mode="inherit" />
</item-descriptor>

<!--  The organizational role, which is really an implementation
 of relativeTo of item-type organization. -->

<item-descriptor name="organizationalRole" super-type="role"
 sub-type-value="organizationalRole"
 display-name-resource="itemDescriptorOrganizationalRole" default="false"
 content="false" folder="false" use-id-for-path="false" hidden="false"
 expert="false" cache-mode="simple">
  <attribute name="resourceBundle"
   value="atg.userprofiling.UserProfileTemplateResources" />
  <table name="dps_relativerole" id-column-name="role_id" type="auxiliary">
    <property name="function" column-name="dps_function" data-type="string"
     required="true" display-name-resource="function" readable="true"
     writable="true" queryable="true" hidden="false" expert="false"
     cache-mode="inherit">
      <attribute name="propertySortPriority" value="30" />
    </property>
    <property name="relativeTo" hidden="false" item-type="organization"
     column-name="relative_to" required="true" cascade="update"
     display-name-resource="relativeTo" readable="true" writable="true"
     queryable="true" expert="false" cache-mode="inherit">
      <attribute name="propertySortPriority" value="40" />
    </property>
  </table>
</item-descriptor>

<!--  The Organization definition -->
```

```
<item-descriptor name="organization"
 display-name-resource="itemDescriptorOrganization" display-property="name"
 default="false" content="false" folder="false" use-id-for-path="false"
 hidden="false" expert="false" cache-mode="simple">
  <attribute name="resourceBundle"
   value="atg.userprofiling.UserProfileTemplateResources" />
  <attribute name="categoryBasicsPriority" value="10" />
  <attribute name="categoryOrganizationRolesPriority" value="20" />
  <table name="dps_organization" type="primary" id-column-name="org_id">
    <property category-resource="categoryBasics" name="name" data-type="string"
     required="true" column-name="name" display-name-resource="name"
     readable="true" writable="true" queryable="true" hidden="false"
     expert="false" cache-mode="inherit">
      <attribute name="propertySortPriority" value="10" />
    </property>
    <property category-resource="categoryBasics" name="description"
     data-type="string" column-name="description"
     display-name-resource="description" required="false" readable="true"
     writable="true" queryable="true" hidden="false" expert="false"
     cache-mode="inherit">
      <attribute name="propertySortPriority" value="20" />
    </property>
    <property category-resource="categoryOrganizationRoles"
     name="parentOrganization" item-type="organization"
     display-name-resource="parentOrganization" column-name="parent_org"
     required="false" readable="true" writable="true" queryable="true"
     hidden="false" expert="false" cache-mode="inherit">
      <attribute name="propertySortPriority" value="30" />
    </property>
  </table>
  <table name="dps_org_chldorg" type="multi" id-column-name="org_id">
    <property category-resource="categoryOrganizationRoles"
     name="childOrganizations" data-type="set"
     component-item-type="organization" column-name="child_org_id"
     display-name-resource="childOrganizations" required="false"
     readable="true" writable="true" queryable="true" hidden="false"
     expert="false" cache-mode="inherit">
      <attribute name="propertySortPriority" value="40" />
    </property>
  </table>
  <table name="dps_org_ancestors" type="multi" id-column-name="org_id"
   multi-column-name="sequence_num">
    <property category-resource="categoryOrganizationRoles"
     name="ancestorOrganizations" data-type="list"
     component-item-type="organization" column-name="anc_org"
     display-name-resource="ancestorOrganizations" required="false"
     readable="true" writable="true" queryable="true" hidden="false"
     expert="false" cache-mode="inherit">
      <attribute name="propertySortPriority" value="50" />
    </property>
  </table>

<!--  Adding roles to the list of organizations -->

  <table name="dps_org_role" type="multi" id-column-name="org_id">
    <property category-resource="categoryOrganizationRoles" name="roles"
       data-type="set" component-item-type="role" column-name="atg_role"
       display-name-resource="roles" required="false" readable="true"
       writable="true" queryable="true" hidden="false" expert="false"

       cache-mode="inherit">
      <attribute name="propertySortPriority" value="60" />
    </property>
  </table>

<!--  This Organization has these RelativeRoles -->

  <table name="dps_role_rel_org" type="multi" id-column-name="organization"
   multi-column-name="sequence_num">
    <property category-resource="categoryOrganizationRoles" name="relativeRoles"
     data-type="list" component-item-type="organizationalRole"
     column-name="role_id" display-name-resource="relativeRoles"
     required="false" readable="true" writable="true" queryable="true"
     hidden="false" expert="false" cache-mode="inherit">
      <attribute name="propertySortPriority" value="70" />
    </property>
  </table>
  <table name="dps_user_org" type="multi" id-column-name="organization">
    <property category-resource="categoryOrganizationRoles" name="members"
     data-type="set" component-item-type="user" column-name="user_id"
     writable="true" display-name-resource="members" required="false"
     readable="true" queryable="true" hidden="false" expert="false"
     cache-mode="inherit">
      <attribute name="resourceBundle"
       value="atg.userprofiling.UserProfileTemplateResources" />
      <attribute name="propertySortPriority" value="-2" />
    </property>
  </table>
</item-descriptor>

<!--  The folder structure that is used by organizations -->

<item-descriptor name="genericFolder" sub-type-property="type"
       display-name-resource="itemDescriptorGenericFolder"
       sub-type-value="genericFolder" expert="true" display-property="name"
       default="false" content="false" folder="false" use-id-for-path="false"
       hidden="false" cache-mode="simple">
  <attribute name="resourceBundle"
   value="atg.userprofiling.UserProfileTemplateResources" />
  <table name="dps_folder" type="primary" id-column-name="folder_id">
    <property name="type" data-type="enumerated" expert="true"
     display-name-resource="type" required="false" readable="true"
     writable="true" queryable="true" hidden="false" cache-mode="inherit">
      <attribute name="useCodeForValue" value="false" />
      <option value="genericFolder" code="2000" />
      <option value="roleFolder" code="2001" />
      <option value="orgFolder" code="2002" />
      <attribute name="propertySortPriority" value="60" />
    </property>
    <property name="name" data-type="string" column-name="name" required="true"
     display-name-resource="name" readable="true" writable="true"
     queryable="true" hidden="false" expert="false" cache-mode="inherit">
      <attribute name="propertySortPriority" value="10" />
    </property>
    <property name="description" data-type="string" column-name="description"
     display-name-resource="description" required="false" readable="true"
     writable="true" queryable="true" hidden="false" expert="false"
     cache-mode="inherit">
      <attribute name="propertySortPriority" value="20" />
    </property>
    <property name="parent" item-type="genericFolder" column-name="parent"
     display-name-resource="parent" required="false" readable="true"
     writable="true" queryable="true" hidden="false" expert="false"
     cache-mode="inherit">
      <attribute name="propertySortPriority" value="30" />
    </property>
  </table>
  <table name="dps_child_folder" type="multi" id-column-name="folder_id">
    <property name="childFolders" data-type="set"
       component-item-type="genericFolder" column-name="child_folder_id"
       queryable="true" display-name-resource="childFolders" required="false"
       readable="true" writable="true" hidden="false" expert="false"

       cache-mode="inherit">
      <attribute name="propertySortPriority" value="40" />
    </property>
  </table>
  <property name="childItems" required="false" readable="true" writable="true"
       queryable="true" hidden="false" expert="false" cache-mode="inherit" />
</item-descriptor>

<item-descriptor name="roleFolder" super-type="genericFolder" sub-type-
       value="roleFolder" display-name-resource="itemDescriptorRoleFolder"
       default="false" content="false" folder="false" use-id-for-path="false"
       hidden="false" expert="false" cache-mode="simple">
  <attribute name="resourceBundle"
       value="atg.userprofiling.UserProfileTemplateResources" />
  <table name="dps_rolefold_chld" type="multi" id-column-name="rolefold_id">
    <property name="childItems" data-type="set" component-item-type="role"
```

```
       column-name="role_id" display-name-resource="childItems" required="false"
       readable="true" writable="true" queryable="true" hidden="false"
       expert="false" cache-mode="inherit">
      <attribute name="propertySortPriority" value="50" />
    </property>
  </table>
</item-descriptor>
```

Configuring User Directory Components

Most configuration properties that you may have to edit for the user directory are contained in the following two components:

- ProfileUserDirectory Component
- ProfileUserDirectoryProperties Component

ProfileUserDirectory Component

The following example shows the default values in the .properties file for the atg/userprofiling/ProfileUserDirectory component (class atg.userdirectory.repository.RepositoryUserDirectoryImpl).

```
$class=atg.userdirectory.repository.RepositoryUserDirectoryImpl
#basics

repositoryItemGroupRegistry=/atg/registry/RepositoryGroups
repositoryUserDirectoryProperties=ProfileUserDirectoryProperties
repository=/atg/userprofiling/ProfileAdapterRepository
transactionManager=/atg/dynamo/transaction/TransactionManager

# caches

organizationCache=OrganizationCache
organizationPathCache=OrganizationPathCache
repositoryItemGroupRoleCache=RepositoryItemGroupRoleCache
roleCache=RoleCache
rolePathCache=RolePathCache
folderCache=FolderCache
folderPathCache=FolderPathCache
userCache=UserCache

#view names

userViewName=user
relativeRoleViewName=organizationalRole
organizationViewName=organization
roleViewName=role
folderViewName=roleFolder
```

```
#random

rootRoleFolderPrimaryKey=root
rootOrganizationPrimaryKey=root
```

The properties in the Basics section of the file are used as follows:

Property	Use
`repositoryItemGroupRegistry`	Points to the ATG service that registers any repository item groups (content or profile groups) you have created.
`repositoryUserDirectoryProperties`	Points to the component that keeps track of the names of properties in the user directory. See ProfileUserDirectoryProperties Component below.
`repository`	Points to the profile repository that is accessed by the user directory.
`transactionManager`	Points to the Transaction Manager service. For more information, see *Transaction Management* in the *ATG Programming Guide.*

The properties in the Caches section of the file identify the cache component to use for caching repository items of each item descriptor type. For more information, see Caching and the User Directory.

The properties in the View Names section of the file are the names of the Repository Views that correspond to the user directory item descriptors in the `userProfile.xml` file.

Property	Use
`userViewName`	Corresponds to the user item descriptor
`relativeRoleViewName`	Corresponds to the organizational role item descriptor
`organizationViewName`	Corresponds to the organization item descriptor
`roleViewName`	Corresponds to the global role item descriptor
`folderViewName`	Corresponds to the role folder item descriptor

The properties in the Random section of the file are used as follows:

Property	Use
rootRoleFolderPrimaryKey	Identifies the root folder that appears for global roles in the Roles window of the ATG Business Control Center and the ACC. The value is the root folder's repositoryId.
rootOrganizationPrimaryKey	Identifies the repository ID of the root organization in the Organizations window of the ATG Business Control Center and the ACC. The value is the root organization's repositoryId. **Important:** If you add organizations to your user directory programmatically, none of them will appear in the the ATG Business Control Center or the ACC unless you change this property to point to the repositoryId of the organization that you added as your root.

ProfileUserDirectoryProperties Component

The atg/userprofiling/ProfileUserDirectoryProperties component (class atg.userdirectory.repository.RepositoryUserDirectoryProperties) performs a function for the user directory definition that is similar to the /atg/userprofiling/PropertyManager component's function for the user profile definition (see Modifying Standard Profile Properties). It maintains a list of the names of various user directory properties as defined in the userProfile.xml file. If you rename any of these properties in the XML file, you must also edit the corresponding configured value in the ProfileUserDirectoryProperties component.

The following sample shows the properties file for this component:

```
$class=atg.userdirectory.repository.RepositoryUserDirectoryProperties

userLoginPropertyName^=/atg/userprofiling/PropertyManager.loginPropertyName
userPasswordPropertyName^=/atg/userprofiling/PropertyManager.passwordPropertyName
userFirstNamePropertyName^=/atg/userprofiling/
 PropertyManager.firstNamePropertyName
userLastNamePropertyName^=/atg/userprofiling/PropertyManager.lastNamePropertyName
userEmailAddressPropertyName^=/atg/userprofiling/
 PropertyManager.emailAddressPropertyName
userRolesPropertyName^=/atg/userprofiling/PropertyManager.rolesPropertyName
userParentOrganizationPropertyName^=/atg/userprofiling/
 PropertyManager.organizationPropertyName
userAncestorOrganizationsPropertyName^=/atg/userprofiling/
 PropertyManager.ancestorOrganizationsPropertyName

organizationNamePropertyName=name
organizationDescriptionPropertyName=description
organizationAncestorOrganizationsPropertyName=ancestorOrganizations
organizationParentOrganizationPropertyName=parentOrganization
organizationRolesPropertyName=roles
```

```
organizationRelativeRolesPropertyName=relativeRoles
organizationMembersPropertyName=members
organizationChildOrganizationsPropertyName=childOrganizations

roleTypePropertyName=type
roleNamePropertyName=name
roleVersionPropertyName=version
roleDescriptionPropertyName=description

organizationalRoleFunctionPropertyName=function
organizationalRoleRelativeToPropertyName=relativeTo

folderTypePropertyName=type
folderNamePropertyName=name
folderDescriptionPropertyName=description
folderParentPropertyName=parent
folderChildFoldersPropertyName=childFolders
folderChildItemsPropertyName=childItems
```

Caching and the User Directory

The user directory employs standard ATG SQL repository caching techniques as described in *SQL Repository Caching* in the *ATG Repository Guide*. By default, caching is turned off for all user directory item descriptor types.

Each item descriptor type in the user directory has a corresponding cache component that maps repository IDs to persistent repository items. The cache component is identified in the Caching section of the `ProfileUserDirectory.properties` file. For example, in the default file, the property `userCache` points to the `atg/userprofiling/userCache` component (class `atg.service.cache.Cache`).

The default properties file for the `userCache` component is shown in the following example:

```
$class=atg.service.cache.Cache

# caching is off by default
cacheAdapter=/atg/userprofiling/userCacheAdapter
```

The properties you can set for the `userCache` component are shown in the following table:

Property	Default Value/Description
`cacheAdapter`	The Nucleus address of the adapter that retrieves items not found in the cache. Default: `/atg/userprofiling/userCacheAdapter`

`maximumCacheEntries`	The maximum number of entries in the cache. 0 = cache nothing. Always get objects from the `cacheAdapter`. -1 = unlimited (Default)
`maximumCacheSize`	The maximum number of bytes in the cache. 0 = Cache nothing. Always get objects from the `cacheAdapter`. -1 = Unlimited (Default)
`maximumEntryLifetime`	The maximum time, in milliseconds, that an entry can exist in the cache. 0 = cache nothing. Always get objects from the `cacheAdapter`. -1 = cache entries never expire (Default)
`maximumEntrySize`	The maximum number of bytes in a single cache entry. 0 = cache nothing. Always get objects from the `cacheAdapter`. -1 = cache entries never expire (Default)

Each cache component points to a corresponding cache adapter component, which retrieves items from the repository that are not in the cache. For example, as shown above, the `userCache` component points to `atg/userprofiling/userCacheAdapter` (class `atg.userdirectory.repository.UserCacheAdapter`). The following sample shows the default properties file for the `userCacheAdapter` component:

```
$class=atg.userdirectory.repository.UserCacheAdapter

userDirectory=/atg/userprofiling/ProfileUserDirectory
```

6 Setting Up an LDAP User Directory

This section describes how to modify the standard SQL user directory to make an LDAP system the primary source of organization information. (Note: In this configuration, only users and organizations come from LDAP. All role-related information is still stored only in the SQL repository.)

The process involves two main steps:

1. Setting up a linked repository.
2. Configuring LDAP user directory components.

These steps are described below.

Setting Up a Linked Repository

This section describes how to configure and link the SQL repository definition file, `userProfile.xml`, and the LDAP repository's `ldapUserProfile.xml` file for the purpose of creating an LDAP-based user directory.

1. Set up implicit repository linking for the two repositories. Implicit linking is a technique in which linked profile items share a unique property in both repositories, and linking is performed dynamically through code. In early versions of the ATG Personalization module, implicit linking was the recommended technique for splitting profile data among repositories of different types; in ATG 6 and later, this technique was superseded by the composite repository configuration described in Setting Up a Composite Profile Repository. Using implicit linking is still required, however, if you want to set up an LDAP-based user directory, and information about it is included in this manual for that purpose.

   Follow the directions in Linking SQL and LDAP Repositories. In particular, make sure you perform the steps in the subsection Configuring Personalization Module Components for Linked Repositories.

2. Follow the directions exactly to set up the `user` view.

3. Determine the attributes you will use as the `entryId` and `parentId` LDAP attributes.

   These must be attributes that exist in one of the object classes given as the object classes of a `user`. For example, the default LDAP repository implementation shows that a user has the object classes `top`, `person`, `organizationalPerson`, and `inetorgPerson`. Pick or create an attribute in one of these object classes to act as an `entryId`. Do the same thing for `parentId`.

The parentId attribute holds the entryId of an object's parent object. For example, assume that the organizational unit People (ou=People, dc=atg.com) has the entryId 4. Also assume there is a user in the People organization whose userid is johnq. Johnq will have an entryId of 5, for example, and a parentId of 4. Make LDAP properties out of these attributes (see example).

Note: In some directory servers, this relationship is already set up. However, you may not be able to find entryId and parentId as attributes of any object class. The process described here should work successfully regardless of whether you can find the attributes.

If this relationship is not already defined in your brand of directory server, follow the instructions above to add the necessary attributes to your schema. Then set the values of those attributes for each organization and user that you want to expose in your ATG environment. Make sure that the values set up the relationship pattern outlined above: the root organizational unit has a particular entryId and an empty parentId. Then, all child organizational units and users of the root organization have unique entryIds and a parentId that is the same as the root organization's entryId.

4. Turn the default organization SQL item descriptor into a linked item descriptor. Do this by using XML combination to add a new property, ldapOrganization, to the organization item descriptor. This property looks very much like the sample ldapUser property described in Linking SQL and LDAP Repositories.

   Also, you must add a new view called organizationalUnit to the ldapUserProfile.xml file. See the sample userProfile.xml below for details. This configuration is produced by using the example in Linking SQL and LDAP Repositories as a model and substituting the organization item descriptor for user in the instructions. Pick a particular item descriptor in the LDAP repository which represents an LDAP organization.

   Note: There is sometimes more than one object class that represents an organization in an LDAP system. For example, some people consider a domain to be a type of organization (dc=atg.com). In addition, a typical LDAP installation contains the object classes organization and organizationalUnit. As an ATG installation uses only one item descriptor for all organizations, there can be only one LDAP object class which represents implicitly linked organizations. The default is organizationalUnit, as this is the most commonly used LDAP organizational structure. **Note that the root organization must also be an organizational unit.** OrganizationalUnit is in the default installation—you can select any one object class to represent organizations in LDAP. Unfortunately, you cannot use your domain as your root organization for the reasons listed above. The key point is that there can be only one object class which corresponds to an organization in ATG.

5. Make sure there is a root organization in ATG that is linked to your chosen LDAP root organization. This step needs to be performed only if useGSARepositoryIdAsPrimaryKey is true (see the description of the ProfileUserDirectory component, and the important notes that follow it, for more information). If useGSARepositoryIdAsPrimaryKey is false, the SQL repository root organization will be created for you the first time it is accessed.

   If no root organization exists in your SQL repository, create an organization item whose uniqueIdPropertyLocal property value matches the uniqueIdPropertyRemote property value of the LDAP repository item that corresponds to your chosen LDAP root organization. See Linking SQL and LDAP Repositories for explanations of these terms.

Example: Assume you pick the organization with the DN "ou=People,dc=atg.com" in LDAP as your root LDAP organization. You've set up your LDAP repository's organization item descriptor to have a property, name, that corresponds to the LDAP attribute ou. If you used the instructions here as a guide, you would have an item in the organizationalUnit item descriptor of your LDAP repository whose name is People. You would also set up your SQL repository organization item descriptor to have a property named ldapOrganization that is a RepositoryLinkPropertyDescriptor. In the example, the uniqueIdPropertyLocal is name, and the uniqueIdPropertyRemote is also name. In order to link a SQL repository item with the previously mentioned LDAP item, all you would have to do would be to create a SQL repository organization item whose name is People. The RepositoryLinkPropertyDescriptor does the rest.

If there is a pre-existing root organization in your SQL repository, modify the default root organization to point to the LDAP root organization.

The following steps show how to modify the default root organization:

- In the ACC, select People and Organizations > Profile Repository.
- Perform a query for items of type Organization.
- Edit the organization with the ID root, changing its name property to the name of your selected LDAP root organization, for example People.

Alternatively, use a SQL editor to change the entry in the dps_organization table whose org_id is root. Change the name property to People.

Make sure your LDAP database is using a password encryption scheme supported by ATG's NDSPasswordHasher component.

In addition, make sure that the passwordHasher property of the ATG installation's PropertyManager component points to the NDSPasswordHasher component as follows:

```
passwordHasher=/atg/adapter/ldap/NDSPasswordHasher
```

And then set the encryption property of this component to the appropriate value (clearText, SHA, or SSHA), for example:

```
encryption=SHA
```

Notes:

- If you change your password encryption scheme, you must then regenerate the passwords for all existing users in your LDAP database. This is because all existing users already have their passwords stored in the database and encrypted with the old scheme.
- If you use an LDAP server other than Oracle Directory Server, you must create and configure a custom password hasher component rather than using NDSPasswordHasher. For more information, see *LDAP Password Encryption* in the *LDAP Repositories* chapter of the *ATG Repository Guide*.

Removing Information from an LDAP User Directory

In an LDAP-based user directory, information flows from LDAP into ATG. For this reason, if you remove information from LDAP using LDAP administration tools, you must also remove the corresponding information from the ATG environment by hand. This step is not performed automatically. For example: if you remove the organizational unit with DN “ou=People,dc=atg.com” from LDAP, you must also remove the organization whose name is People from ATG.

Sample XML Files for an LDAP User Directory

This section includes examples of the userProfile.xml file and ldapUserProfile.xml file, set up to show the implicit linking configuration described in the previous section.

Sample userProfile.xml file

The following code sample shows the SQL profile repository definition file, userProfile.xml, set up to support an LDAP-based user directory. (Note that the example includes only the relevant section of this file.)

```
<gsa-template>
  <item-descriptor name="user">

    <table name="dps_user">

      <!-- Remove properties which are in LDAP -->
      <property name="password" xml-combine="remove"/>
      <property name="email" xml-combine="remove"/>
      <property name="firstName" xml-combine="remove"/>
      <property name="lastName" xml-combine="remove"/>

      <!-- Replicate unique id property into LDAP -->
      <property name="login"
          property-type="atg.adapter.gsa.ReplicatePropertyDescriptor">
        <attribute name="replicateProperty" value="ldapUser.login"/>
      </property>

    </table>

    <!-- Add property which points to the LDAP item -->
    <property name="ldapUser"
        property-type="atg.repository.linked.RepositoryLinkPropertyDescriptor"
        repository="/atg/adapter/ldap/LDAPRepository"
        item-type="user">
<!--
        cascade="insert,update,delete">
-->
      <attribute name="uniqueIdPropertyLocal" value="login"/>
      <attribute name="uniqueIdPropertyRemote" value="login"/>
    </property>
  </item-descriptor>

  <item-descriptor name="organization">
    <!-- add a link property -->
    <property name="ldapOrganization"
        property-type="atg.repository.linked.RepositoryLinkPropertyDescriptor"
        repository="/atg/adapter/ldap/LDAPRepository"
        item-type="organizationalUnit">
      <attribute name="uniqueIdPropertyLocal" value="name"/>
      <attribute name="uniqueIdPropertyRemote" value="name"/>
    </property>

  </item-descriptor>

<import-items>
  <add-item item-descriptor="organization"
   repository="/atg/userprofiling/ProfileAdapterRepository" id="root">
    <set-property name="name">People</set-property>
  </add-item>
</import-items>

</gsa-template>
```

Sample ldapUserProfile.xml file

The following code sample shows the LDAP profile repository definition file, `ldapUserProfile.xml`, set up to support an LDAP-based user directory.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE ldap-adapter-template
        PUBLIC "-//Art Technology Group, Inc.//DTD LDAP Adapter//EN"
        "http://www.atg.com/dtds/ldap/ldap_1.0.dtd">

<ldap-adapter-template xml-combine="replace">

<header>
  <name>ldapUserProfile.xml</name>
  <author>ATG</author>
</header>

<!-- organization view -->

<view name="organizationalUnit">

  <item-descriptor name="organizationalUnit" display-name="Organizational Unit"
   display-property="name">
  <!-- special properties -->
  <id-property name="id" in-ldap="false"/>
  <object-classes-property name="objectClasses" ldap-name="objectclass"/>

  <!-- object classes -->
  <object-class>top</object-class>
  <object-class>organizationalUnit</object-class>

  <!-- properties -->
  <property name="name" ldap-name="ou" data-type="string" required="true"/>
  <property name="entryId" ldap-name="entryid" data-type="int" required="true"/>
  <property name="parentId" ldap-name="parentid" data-type="int" required="true"/>

  <!-- item creation -->
  <new-items allowed="false"/>
  </item-descriptor>

  <!-- search roots -->
  <search-root dn="dc=atg.com"/>

</view>

<!-- user view -->
<view name="user" default="true">

  <!-- item descriptor -->
  <item-descriptor name="user" display-name="User" display-property="login">

    <!-- special properties -->
    <id-property name="id" in-ldap="false"/>
    <object-classes-property name="objectClasses" ldap-name="objectclass"/>

    <!-- object classes -->
    <object-class>top</object-class>
    <object-class>person</object-class>
    <object-class>organizationalPerson</object-class>
    <object-class>inetorgPerson</object-class>

    <!-- properties -->
    <property name="names" ldap-name="cn" data-type="string" multi="true"
     required="true"/>
    <property name="login" ldap-name="uid" data-type="string" required="true">
      <attribute name="unique" value="true"/>
    </property>
    <property name="password" ldap-name="userpassword" data-type="string"
     required="false"
              editor-class="atg.beans.PasswordPropertyEditor">
      <attribute name="passwordHasher"
       bean="/atg/adapter/ldap/NDSPasswordHasher"/>
    </property>
    <property name="fullName" ldap-name="cn" data-type="string" required="true"/>
    <property name="lastName" ldap-name="sn" data-type="string" required="true"/>
    <property name="firstName" ldap-name="givenName" data-type="string"/>
    <property name="email" ldap-name="mail" data-type="string"/>

    <property name="parentId" ldap-name="parentid" data-type="int"
     required="true"/>
    <property name="entryId" ldap-name="entryid" data-type="int" required="true"/>

    <!-- item creation -->

    <new-items parent-dn="dc=atg.com" rdn-property="login"/>
<!--
    <new-items parent-dn="DC=atg,DC=com" rdn-property="login"/>
-->
  </item-descriptor>

  <!-- search roots -->
  <search-root dn="dc=atg.com"/>
<!--
  <search-root dn="DC=atg,DC=com"/>
-->

</view>
</ldap-adapter-template>
```

Configuring LDAP User Directory Components

This section describes the changes you need to make to the following user directory components to support an LDAP-based user directory.

- ProfileUserDirectory
- ProfileUserDirectorySpider
- LDAPOrganizationItemFinder
- ProfileItemFinder
- PropertyManager
- ProfileUserDirectoryProperties
- LDAPUserCache
- LDAPUserCacheAdapter
- LDAPOrganizationCache
- LDAPOrganizationCacheAdapter

ProfileUserDirectory

The following section describes how to set the properties of the `atg/userprofiling/ProfileUserDirectory` component for an LDAP user directory configuration.

Property	Description
`profileItemFinder`	The profile item finder which looks users up by their attributes or IDs. Set this property to the Personalization module's profile item finder. The default is `/atg/userprofiling/ProfileItemFinder`.
`repositoryItemGroupRegistry`	The registry of repository item groups (also known as profile groups). No change.
`repositoryUserDirectoryProperties`	The component which tracks repository item property names, much as does the `/atg/userprofiling/PropertyManager`. No change.
`repository`	The backing repository. This property should point to the implicitly linked SQL/LDAP repository. If you follow the procedure earlier in this section, you should not have to change this property.
`transactionManager`	The component which helps the user directory make use of Dynamo's transactional database interaction. For more information, see *Transaction Management* in the *ATG Programming Guide*. **Note**: Since LDAP systems are not transaction aware, the LDAP repository is also not transactional. This behavior means that it is possible for LDAP data to be written permanently even if the transaction within which the operation occurred is rolled back.
`userDirectorySpider`	A spider that walks the user directory's organization and user hierarchy on a specified schedule, thereby creating ATG-side objects corresponding to each user and organization in LDAP, beginning with the root organization. This property is not required. Set this property to null if you don't require this behavior. See also `ProfileUserDirectorySpider`.
`organizationCache`	The cache that translates organization primary keys into organization objects. Set this property to `LDAPOrganizationCache`.
`organizationPathCache`	Translates organization paths within the organization hierarchy into organization objects. No change.

repositoryItemGroupRoleCache	Translates profile groups into role objects. No change.
roleCache	Translates role primary keys into role objects. No change.
rolePathCache	Translates role paths within the role folder structure into role objects. No change.
folderCache	Translates role folder primary keys into role folder objects. No change.
folderPathCache	Translates role folder paths within the role folder structure into role folder objects. No change.
userCache	Translates user primary keys into user objects. Set this property to LDAPUserCache.
userViewName	The name of the user view in the repository backing this directory.
organizationViewName	The name of the organization view in the repository backing this directory.
relativeRoleViewName	The name of the relative role view in the repository backing this directory.
roleViewName	The name of the role view in the repository backing this directory.
folderViewName	The name of the folder view in the repository backing this directory.
rootRoleFolderPrimaryKey	The primary key of the root role folder. No change.
rootOrganizationPrimaryKey	The primary key of the root organization. See Setting the rootOrganizationPrimaryKey Property below.
organizationItemFinder	Similar to the ProfileItemFinder (see ProfileItemFinder Component). Its purpose is to locate organization repository items. This property points to an instance of RepositoryLinkProfileItemFinder that is configured to look for organization items. Set this property to /atg/userprofiling/LDAPOrganizationItemFinder (the default).

useGSARepositoryIdAsPrimaryKey	Determines whether user directory objects use their SQL repository item’s repository ID as their primary key (the default, and the way that the default user directory is configured) or whether user directory objects use the repository ID of the linked LDAP item as their primary key. Your choice here has significant effects on the portability of your code. See Important Notes About the useGSARepositoryIdAsPrimaryKey Property for information.

Setting the rootOrganizationPrimaryKey Property

The value you set for the `rootOrganizationPrimaryKey` property depends on the value of `useGSARepositoryIdAsPrimaryKey`. If `useGSARepositoryIdAsPrimaryKey` is true (the default), this value should be the repository ID of the SQL repository item that is implicitly linked to the LDAP item that you want to be your root organization.

For example, assume you want the organization `People` in LDAP to be your root organization (remember that you must pick an LDAP object of object class `organizationalUnit`, or of whatever type the `organization` LDAP repository view specifies). First make sure that your organization shows up as an item of type `organization` in your LDAP repository. Next, if `useGSARepositoryIdAsPrimaryKey` is false, you set `rootOrganizationPrimaryKey` to be the `People` organization item’s `repositoryId` (`ou=People,dc=atg.com`, perhaps). You don’t need to create a root SQL repository item, because the `organizationItemFinder` will automatically create a SQL repository counterpart item when the root organization is requested.

If `useGSARepositoryIdAsPrimaryKey` is true, you must first create a SQL repository item of type `organization` and configure it so that it is implicitly linked with the LDAP organization repository item for `people` (the LDAP item with ID `ou=People,dc=atg.com`). Then, set `rootOrganizationPrimaryKey` to be the repository ID of your newly created root SQL repository organization (it could be anything, for example `1111`).

See the next section for some important information about the effects of setting the `useGSARepositoryIdAsPrimaryKey` property.

Important Notes About the useGSARepositoryIdAsPrimaryKey Property

Do not write any application code that determines a user directory object’s primary key from a source outside the user directory and then uses that key to fetch a user directory object.

Example: the default user directory uses a profile ID (user repository item `repositoryId`) as a user directory `user` object’s primary key. However, applications should not use this knowledge to take a profile ID from a session and then call `findUserByPrimaryKey` on the user directory, passing in the session’s profile ID. Primary keys are implementation specific. **Therefore, any application code written to rely on aspects of the default user directory implementation’s primary key system is not portable**.

The recommended way to work with user directory primary keys is as follows:

1. Get the root organization.
2. List the child organizations of the root organization.
3. List user members of one of those organizations.
4. Get the primary key of the member and save it for later use.
5. Later, call `findUserByPrimaryKey`, passing in the saved primary key.

Obviously, these steps apply only to an application that wants to get a child organization of the root organization by primary key. The important point is that user directory primary keys must come from the user directory objects themselves, not from a source outside the user directory.

Nonetheless, in some exceptional circumstances it is necessary to write non-portable application code that does exactly what this section recommends against doing. For example, some parts of the ACC interface employ the knowledge that user directory `user` objects have a primary key that is the same as their backing item's `repositoryId` in order to reuse some `RepositoryItem` picker code. This design makes the ACC interface non-portable but more functional. For purposes of maximum compatibility with the user interface, therefore, the default setting for `useGSARepositoryIdAsPrimaryKey` is true. If you require use of the ACC for managing organization, role, and user data, you must leave this property set to true.

When `useGSARepositoryIdAsPrimaryKey` is true, only user directory objects whose SQL repository item has already been created will be available for fetching by primary key. If your application code follows the 5-step coding guidelines shown above, the problem is avoided, because by the time your code asks a user directory object for its primary key, it has by definition got a SQL repository item associated with it. However, if a primary key is fabricated outside the user directory, only items that have been accessed through the user directory in some other way will be available for fetching by primary key.

Consider the following example. Some application code searches for a user whose primary key is `4444`. In the default user directory, the profile repository is not implicitly linked. Therefore, all available users exist in the SQL repository and can be fetched at any time by their primary key (in this case, `4444`). Assume, however, that you change the SQL repository to be implicitly linked to an LDAP repository. With this configuration, users can be in the LDAP repository but not have a SQL repository counterpart yet. When `useGSARepositoryIdAsPrimaryKey` is true, the user with primary key `4444` cannot be found by primary key until he or she is found as a member of an organization or by some similar operation.

To summarize, a `useGSARepositoryIdAsPrimaryKey` setting of true makes all existing user directory application code compatible with the LDAP user directory, but it has some unexpected and potentially undesirable results: namely that LDAP organizations and users cannot be fetched by primary key until a SQL repository link item has been created that is implicitly linked to its LDAP manifestation. If you must set this property to true, for example because you want to use the ACC for managing user directory data, make sure that any code you write follows the 5-step coding guidelines in this section.

When `useGSARepositoryIdAsPrimaryKey` is false, user directory objects use their implicitly linked LDAP repository item's ID as their primary key. This behavior means that application code that must fetch a user directory `user` object directly by primary key can do so, and success can be guaranteed; if the primary key corresponds to a user repository item in LDAP, the user directory user object will always be found.

The false setting is most useful for new application code, as existing code will not be determining the correct primary key for a user. Existing code uses SQL repository IDs rather than LDAP repository IDs as primary keys.

Sample ProfileUserDirectory.properties File

The code below is an example of a `ProfileUserDirectory.properties` file for an LDAP user directory.

```
$class=atg.userdirectory.repository.ldap.LDAPRepositoryUserDirectory

#basics

profileItemFinder=/atg/userprofiling/ProfileItemFinder
repositoryItemGroupRegistry=/atg/registry/RepositoryGroups
repositoryUserDirectoryProperties=ProfileUserDirectoryProperties
repository=/atg/userprofiling/ProfileAdapterRepository
transactionManager=/atg/dynamo/transaction/TransactionManager
userDirectorySpider=ProfileUserDirectorySpider

# caches

organizationCache=LDAPOrganizationCache
organizationPathCache=OrganizationPathCache
repositoryItemGroupRoleCache=RepositoryItemGroupRoleCache
roleCache=RoleCache
rolePathCache=RolePathCache
folderCache=FolderCache
folderPathCache=FolderPathCache
userCache=LDAPUserCache

#view names
userViewName=user
relativeRoleViewName=organizationalRole
organizationViewName=organization
roleViewName=role
folderViewName=roleFolder

#random
rootRoleFolderPrimaryKey=root
# rootOrganizationPrimaryKey=ou=People,dc=atg.com
rootOrganizationPrimaryKey=1111

# ldap
organizationItemFinder=/atg/userprofiling/LDAPOrganizationItemFinder
useGSARepositoryIdAsPrimaryKey=true
```

ProfileUserDirectorySpider

The /atg/userprofiling/ProfileUserDirectorySpider component is a service that periodically walks the LDAP user and organization hierarchy on a specified schedule, thereby pulling user and organization information from LDAP into the SQL repository.

Using this component is not required; if you do not need this behavior, set the userDirectorySpider property in the ProfileUserDirectoryProperties component to null.

The code below is an example of a ProfileUserDirectorySpider.properties file.

```
# Version: $Change: 217855 $$DateTime: 2001/10/31 14:12:45 $
$class=atg.userdirectory.UserDirectorySpider

userDirectory=ProfileUserDirectory
scheduler=/atg/dynamo/service/Scheduler
schedule=every 1 hour in 1 hour
```

Set the properties of this component as follows.

Property	Description
userDirectory	The name of the LDAP user directory to walk.
scheduler	The Nucleus address of the scheduler service that manages the schedule on which the spider walks the user directory.
schedule	The schedule on which the spider walks the user directory.

Note that a default instance of this component is not provided with the Personalization module. If you want to use this component, create an instance of class atg.userdirectory.UserDirectorySpider and configure it as shown here.

LDAPOrganizationItemFinder

The /atg/userprofiling/LDAPOrganizationItemFinder component is an instance of RepositoryLinkProfileItemFinder. Its purpose is to locate organization items in an LDAP repository. It is similar in function and behavior to the standard ProfileItemFinder component.

The code below is an example of an LDAPOrganizationItemFinder.properties file.

```
$class=atg.userprofiling.RepositoryLinkProfileItemFinder
profileTools=ProfileTools
propertyManager=PropertyManager
profileRepository=ProfileAdapterRepository
```

```
linkedRepository=/atg/adapter/ldap/LDAPRepository
linkedProfileTypeMap=organization=organizationalUnit
linkPropertyLocal=name
linkPropertyRemote=name
createLocalProfiles=true
```

Set the properties of this component as follows.

Property	Description
`profileTools`	The profile tools component that this finder uses to find repository items. Default: `/atg/userprofiling/ProfileTools`
`propertyManager`	The Personalization module's standard property manager. Default: `/atg/userprofiling/PropertyManager`
`profileRepository`	The repository in which this finder finds repository items. Default: `/atg/userprofiling/ProfileAdapterRepository`
`linkedRepository`	The repository in which the items implicitly linked to SQL repository items reside. Default: `/atg/adapter/ldap/LDAPRepository`
`linkedProfileTypeMap`	The map of local SQL repository item types to remote LDAP item types. The default assumes that your SQL organization item type is `organization` and your LDAP organization item type is `organizationalUnit`. If you change either your SQL or LDAP organization view names, you must also change these map entries. Default: `organization=organizationalUnit`
`linkPropertyLocal`	The name of the property of SQL repository `organization` items that implicitly links them to LDAP `organizationalUnit` items. The `linkPropertyLocal` property must be set to the same value as `linkPropertyLocal` for the link property in the `organization` item descriptor in the `userProfile.xml` file. Default: name
`linkPropertyRemote`	The name of the property of LDAP `organizationalUnit` items that implicitly links them to SQL repository `organization` items. The `linkPropertyRemote` property must be set to the same value as `linkPropertyRemote` for the link property in the `organization` item descriptor in the `userProfile.xml` file. Default: name

`createLocalProfiles`	Tells this profile finder whether to create local SQL repository organization items when a remote LDAP `organizationalUnit` item is found. Default: `false`

Note that a default instance of this component is not provided with the Personalization module. If you want to use this component, create an instance of class `atg.userprofiling.RepositoryLinkProfileItemFinder` and configure it as shown here.

ProfileItemFinder

The `/atg/userprofiling/ProfileItemFinder` component is used by the Personalization module to find users, either by attribute (for example, a login name or email address) or by ID. For example, when a user logs in using a profile form handler, the supplied username and password are passed on to `ProfileItemFinder` in order to locate the user in the profile repository. The component represents the standard way for any user directory to find a user.

The code below is an example of a `ProfileItemFinder.properties` file configured for an LDAP-based user directory:

```
$class=atg.userprofiling.RepositoryLinkProfileItemFinder

profileTools=ProfileTools
propertyManager=PropertyManager
profileRepository=ProfileAdapterRepository
linkedRepository=/atg/adapter/ldap/LDAPRepository
linkedProfileTypeMap=user=user
linkPropertyLocal=login
linkPropertyRemote=login
loginLinkedPropertyName=login
passwordLinkedPropertyName=password
firstNameLinkedPropertyName=firstName
lastNameLinkedPropertyName=lastName
emailLinkedPropertyName=email
createLocalProfiles=true
```

The following table describes how to set the properties in this component:

Property	Description
`profileTools`	The profile tools component that this finder uses to find repository items. Default: `/atg/userprofiling/ProfileTools`

`propertyManager`	The Personalization module's standard property manager. Default: `/atg/userprofiling/PropertyManager`
`profileRepository`	The repository in which this finder finds repository items. Default: `/atg/userprofiling/ProfileAdapterRepository`
`linkedRepository`	The repository in which the items implicitly linked to SQL repository items reside. Default: `/atg/adapter/ldap/LDAPRepository`
`linkedProfileTypeMap`	The map of local SQL repository item types to remote LDAP item types. The default assumes that your SQL user item type is `user` and your LDAP user item type is also `user`. If you change either your SQL or LDAP user view names, you must also change these map entries. Default: `user=user`
`linkPropertyLocal`	The name of the property of SQL repository `user` items that implicitly links them to LDAP `user` items. The `linkPropertyLocal` property must be set to the same value as `linkPropertyLocal` for the link property in the `user` item descriptor in the `userProfile.xml` file. Default: `login`
`linkPropertyRemote`	The name of the property of LDAP `user` items that implicitly links them to SQL repository `user` items. The `linkPropertyRemote` property must be set to the same value as `linkPropertyRemote` for the link property in the `user` item descriptor in the `userProfile.xml` file. Default: `login`
`createLocalProfiles`	Tells this profile finder whether to create local SQL repository user items when a remote LDAP user item is found. Default: `true`

PropertyManager

To support an LDAP-based user directory, modify the `PropertyManager` component (`/atg/userprofiling/PropertyManager`) so that the `ProfileItemFinder` uses LDAP as its master record of users, looking in the LDAP repository instead of in the `ProfileAdapterRepository`.

The code below is an example of a `PropertyManager.properties` file for this type of configuration:

```
passwordPropertyName=ldapUser.password
emailAddressPropertyName=ldapUser.email
firstNamePropertyName=ldapUser.firstName
lastNamePropertyName=ldapUser.lastName
```

```
loginPropertyName=login
passwordHasher=/atg/adapter/ldap/NDSPasswordHasher
```

Set the properties of this component as follows.

Property	Description
`passwordPropertyName`	The name of a user’s `password` property.
`emailAddressPropertyName`	The name of a user’s `email` property.
`firstNamePropertyName`	The name of a user’s `firstName` property.
`lastNamePropertyName`	The name of a user’s `lastName` property.
`passwordHasher`	The encoder for LDAP passwords. For Oracle Directory Server, set this property to `/atg/adapter/ldap/NDSPasswordHasher`. For more information, see *LDAP Password Encryption* in the *ATG Repository Guide*.

ProfileUserDirectoryProperties

The `/atg/userprofiling/ProfileUserDirectoryProperties` component maps user directory property names to repository property names. Configuring this component for use with LDAP involves adding to the default definitions, as shown in the example below:

```
$class=atg.userdirectory.repository.ldap.LDAPRepositoryUserDirectoryProperties

organizationLinkPropertyName=ldapOrganization
LDAPOrganizationEntryIdPropertyName=entryId
LDAPOrganizationParentIdPropertyName=parentId

userLinkPropertyName=ldapUser
LDAPUserEntryIdPropertyName=entryId
LDAPUserParentIdPropertyName=parentId
```

Set the properties for this component as follows.

Property	Description
`organizationLinkPropertyName`	The name of the link property in the SQL repository `organization` item descriptor.

Property	Description
LDAPOrganizationEntryIdPropertyName	The name of the property in the LDAP organization item descriptor that serves as an item's entryid.
LDAPOrganizationParentIdPropertyName	The name of the property in the LDAP organization item descriptor that serves as an item's parentid.
userLinkPropertyName	The name of the link property in the user item descriptor.
LDAPUserEntryIdPropertyName	The name of the property in the LDAP user item descriptor that serves as a user's entryid.
LDAPUserParentIdPropertyName	The name of the property in the user item descriptor that serves as a user's parentid.

Caching an LDAP User Directory

Each item descriptor type in the user directory has a corresponding cache component that maps repository IDs to persistent repository items. For an LDAP user directory, user and organization are the only item types stored in the LDAP repository, so these items are the only ones whose caching you must change from the default caching methods for a SQL-based user directory (see Caching and the User Directory).

For the user and organization item types, you identify the cache component to use in the Caching section of the ProfileUserDirectory.properties file. Set the userCache property to point to the LDAPUserCache component. Set the organizationCache property to point to the LDAPOrganizationCache component.

Each cache component points to a corresponding cache adapter component, which retrieves items from the repository that are not in the cache.

For more information about setting up caching for a user directory that accesses an LDAP directory rather than a SQL repository, see *Configuring the LDAP Repository Components* in the *ATG Repository Guide*.

LDAPUserCache

The /atg/userprofiling/LDAPUserCache component handles caching for the user item descriptor type in an LDAP user directory.

Set the cacheAdapter property of this component to the /atg/userprofiling/LDAPUserCacheAdapter component as shown in the following properties file:

```
$class=atg.service.cache.Cache

cacheAdapter=LDAPUserCacheAdapter
```

Note that a default instance of this component is not provided with the Personalization module. If you want to use this component, create an instance of class `atg.service.cache.Cache` and configure it as shown here. For information on how to set the other properties of this component, refer to the description of the `UserCache` component, which manages caching for user items in a SQL-based user directory. See Caching and the User Directory.

LDAPUserCacheAdapter

The `/atg/userprofiling/LDAPUserCacheAdapter` component handles the creation of the user objects that populate the `LDAPUserCache`.

Set the `userDirectory` property of this component to the LDAP repository holding the user directory items, as shown in the following properties file:

```
$class=atg.userdirectory.repository.ldap.LDAPUserCacheAdapter

userDirectory=ProfileUserDirectory
```

LDAPOrganizationCache

The `/atg/userprofiling/LDAPOrganizationCache` component handles caching for the `organization` item descriptor type in an LDAP user directory.

Set the `cacheAdapter` property of this component to the `/atg/userprofiling/LDAPOrganizationCacheAdapter` component, as shown in the following properties file:

```
$class=atg.service.cache.Cache

cacheAdapter=LDAPOrganizationCacheAdapter
```

Note that a default instance of this component is not provided with the Personalization module. If you want to use this component, create an instance of class `atg.service.cache.Cache` and configure it as shown here. For information on how to set the other properties of this component, refer to the description of the `UserCache` component, which manages caching for user items in a SQL-based user directory. See Caching and the User Directory.

LDAPOrganizationCacheAdapter

The `/atg/userprofiling/LDAPOrganizationCacheAdapter` component handles the creation of the `organization` objects that populate the `LDAPOrganizationCache`.

Set the userDirectory property of this component to the LDAP repository holding the user directory items, as shown in the following properties file:

```
$class=atg.userdirectory.repository.ldap.LDAPOrganizationCacheAdapter

userDirectory=ProfileUserDirectory
```

7 Linking SQL and LDAP Repositories

Important: In previous versions of the Personalization module, it was possible to use the implicit linking technique described in this chapter to split your profile information between a SQL and an LDAP repository. Although you can still use this type of profile configuration, it is no longer recommended, and it has been superseded by the composite repository configuration described in Setting Up a Composite Profile Repository. The information in this chapter does still apply, however, if you want to configure an LDAP-based user directory as described earlier in this manual (see Setting Up an LDAP User Directory.)

This chapter contains the following sections:

Using Implicit Repository Linking

In some situations, it may not be convenient to store all profile information in a SQL repository. For example, you may want to use already existing profile data from another application that is stored in an LDAP directory. If your profiles also contain data that is not LDAP specific, one solution is to split the profile information between a SQL and an LDAP repository, rather than using one or the other.

This chapter describes how to link a SQL repository and an LDAP repository so that you can use them both to store profile data. This configuration allows you to choose which data to store in which repository. It also allows you to choose the repository through which you authenticate users. You can split up your profile data and authenticate users through LDAP or through SQL.

Repository linking is accomplished by creating a property in the SQL profile template to refer to the profile in the LDAP repository. You can then access LDAP properties via the SQL profile repository item. For example, you can add an `ldapUser` property to the SQL repository’s `user` item descriptor. Then, if your LDAP profile has an `email` property, you can refer to it in a JHTML page as follows:

```
<valueof bean="/atg/userprofiling/Profile.ldapUser.email">
```

Furthermore, the SQL repository can be set up so that when a new SQL profile item is created, updated, or deleted, the corresponding LDAP item is created, updated, or deleted automatically. This technique essentially allows you to treat the profile data stored in both the SQL and LDAP repositories as a single profile object.

The *Linking Repositories* section of the *SQL Repositories* chapter in the *ATG Repository Guide* describes how repository linking can be accomplished by storing the repository ID of the linked repository item as the value of the linked property in the database. This kind of explicit linking relies on the fact that the ID of the linked item does not change over time. However, this may not always be the case in an LDAP repository, which uses a directory entry's DN (Distinguished Name) as its repository ID. Furthermore, explicit linking does not provide for an easy way to authenticate users through the linked LDAP repository.

With implicit repository linking, the SQL repository item does not store the ID of the linked item in the database; instead, the linking is performed dynamically, through code. This is accomplished by requiring that the linked profile items share a property that uniquely identifies them in both repositories. Typically, you would use a property such as `login` or `email`. For example, if both the SQL and LDAP profiles have a `login` property, a SQL profile is linked to an LDAP profile whose `login` property value is the same as the SQL profile's `login` value.

Implicit linking is managed by the package `atg.repository.linked`. The `ldapUser` property, which links the two repositories in the example above, is defined in the SQL profile template as a user-defined property type `atg.repository.linked.RepositoryLinkPropertyDescriptor` (see the *User-Defined Property Types* section of the *SQL Repositories* chapter in the *ATG Repository Guide* for an explanation of user-defined property types). The `RepositoryLinkPropertyDescriptor` class contains the logic that links the two items together; it performs a repository query to obtain the LDAP item corresponding to the given SQL item.

The main limitation of the repository linking approach to combining SQL and LDAP profiles is that the `ldapUser` property is not queryable. Thus, the queries you perform against the SQL repository cannot include any of the LDAP properties. For example, if the `lastName` property is stored in LDAP, you cannot execute a query against the SQL repository to find all users with last name Smith. If you are using the Scenarios module, one consequence is that you cannot have scenarios that perform queries against LDAP properties. For example, you cannot have a scenario that finds all users with the last name Smith and then sends email to them.

In addition, the HTML profile administration interface is not sophisticated enough to handle these kinds of composite profiles, so you cannot use it to create users or to search for or edit their LDAP properties. However, you can still use the ATG Business Control Center or the ACC to accomplish these tasks.

Defining the SQL/LDAP Linked Repositories

This section describes how to configure your SQL and LDAP profile templates to implement repository linking. The example used in this section assumes that you have some basic profile data, such as name, email, login, and password, stored in an existing LDAP directory. Thus, the LDAP repository is used to access this data and to authenticate users. The SQL repository is used to store the remaining profile data, such as address and locale information, and all the ATG-specific data (such as scenario-related data, if you are running the Scenarios module). The SQL repository also replicates the `login` property, which is used to implicitly link the SQL and LDAP profiles.

You can split your data any other way, and modify the profile templates accordingly. The only requirement is that one of the two repositories must contain both the `login` and `password` properties, so that it can be used to authenticate the users. (Note, however, that you must set up the SQL profile repository as the main repository and refer to LDAP profile properties through the SQL repository; you cannot have LDAP as your main repository using this type of configuration.)

To configure your SQL and LDAP repositories, do the following:

1. Decide which profile properties will be stored in SQL and which will be stored in LDAP.

2. Set up your SQL profile repository as the main profile repository. See the *Setting Up a Profile Repository* chapter for more information. Make sure that the `user` item descriptor is configured through the repository definition file to have all the properties you decided to store in SQL, but none of the LDAP properties.

   For example, if you are starting with the default SQL profile template, the `user` item descriptor includes the properties `firstName`,`lastName`, `email`, and `password`. If you decided to store these properties in the LDAP repository, remove them from the SQL definition file.

   Note: Removing these properties will probably require some XML combination techniques. See *XML File Combination* in the *Nucleus: Organizing JavaBean Components* chapter of the *ATG Programming Guide*.

3. Set up your LDAP repository and configure all the associated components. See *Configuring the LDAP Repository Components* in the *LDAP Repositories* chapter of the *ATG Repository Guide*.

   The LDAP repository should **not** be configured as the main profile repository. In other words, the `profileRepository` property of the `/atg/userprofiling/ProfileTools` component should point to `/atg/userprofiling/ProfileAdapterRepository`, as in the default configuration.

   Remove any unneeded properties from the LDAP definition file. If you decide to remove the `login` property, you will have to choose another property to be used when constructing RDNs (Relative Distinguished Names), since the `login` property is used as the value of the `rdn-property` attribute in the default template. This property must have a unique value for each LDAP repository item in order for the corresponding DNs to be unique. See *New Item Creation* in the *LDAP Repository Architecture* section of the *LDAP Repositories* chapter in the *ATG Repository Guide* for more information on constructing RDNs.

4. In the SQL repository definition file, add a property of type `atg.repository.linked.RepositoryLinkPropertyDescriptor` to the `user` item descriptor. This property links the two profiles. For example, you could call this new property `ldapUser` and define it as follows:

   ```
   <property name="ldapUser"
       property-type="atg.repository.linked.RepositoryLinkPropertyDescriptor"
       repository="/atg/adapter/ldap/LDAPRepository"
       item-type="user"
       category="Info"
       display-name="Linked LDAP User"
   ```

```
    cascade="insert,update,delete">
   <attribute name="uniqueIdPropertyLocal" value="login"/>
   <attribute name="uniqueIdPropertyRemote" value="login"/>
</property>
```

The property definition must specify the repository and item type of the linked item. In addition, the attributes `uniqueIdPropertyLocal` and `uniqueIdPropertyRemote` specify which properties should be used as the unique IDs in SQL and LDAP, respectively. The values of these properties must be the same for the two profiles to be linked together.

The `category` and `display-name` properties are optional; they provide better display for the item in the ATG Business Control Center or ACC interface.

If your property definition includes the `cascade="insert,update,delete"` attribute as shown above, the linked item will be automatically created, updated, or deleted whenever the SQL item is created, updated, or deleted. For example, when a user registers through a profile form handler, both a SQL and an LDAP profile will be created for him, and the relevant profile information will be stored in both items.

5. Because the `login` property is replicated in both repositories, the two property values must always be kept in sync. For example, when the user registers, both his `login` and his `ldapUser.login` properties must be set to the same login value. To avoid having to do this by hand, modify the `login` property's definition in the SQL profile template as follows:

```
<property name="login"
    property-type="atg.adapter.gsa.ReplicatePropertyDescriptor">
    <attribute name="replicateProperty" value="ldapUser.login"/>
</property>
```

This code declares the `login` property to be of type `ReplicatePropertyDescriptor`, which overrides `GSAPropertyDescriptor` and takes care of automatically replicating the property value to the property specified via the `replicateProperty` attribute.

With the `login` property configured in this way, you need worry only about setting the `login` property in the SQL profile item; the corresponding LDAP item's `login` property is set automatically.

Sample SQL/LDAP Linked Repository Definitions

The following sample SQL repository definition file overrides the default SQL definition file. It removes several properties from the default configuration, and it adds an `ldapUser` property, as described in step 4 above. It also modifies the default `login` property definition as specified in step 5.

```
<gsa-template>
  <item-descriptor name="user">
```

```
    <table name="dps_user">

      <!-- Remove properties which are in LDAP -->
      <property name="password" xml-combine="remove"/>
      <property name="email" xml-combine="remove"/>
      <property name="firstName" xml-combine="remove"/>
      <property name="lastName" xml-combine="remove"/>

      <!-- Replicate unique id property into LDAP -->
      <property name="login"
          property-type="atg.adapter.gsa.ReplicatePropertyDescriptor">
        <attribute name="replicateProperty" value="ldapUser.login"/>
      </property>

    </table>

    <!-- Add property which points to the LDAP item -->
    <property name="ldapUser"
        property-type="atg.repository.linked.RepositoryLinkPropertyDescriptor"
        repository="/atg/adapter/ldap/LDAPRepository"
        item-type="user"
        category="Info"
        display-name="Linked LDAP User"
        cascade="insert,update,delete"
        queryable="false">
      <attribute name="uniqueIdPropertyLocal" value="login"/>
      <attribute name="uniqueIdPropertyRemote" value="login"/>
    </property>

  </item-descriptor>
</gsa-template>
```

See the *Sample LDAP Profile Repository Definition File* section of the *Setting Up an LDAP Profile Repository* chapter for a sample LDAP repository definition file. No changes are needed to the LDAP definition file to set up repository linking.

Configuring Personalization Module Components for Linked Repositories

Depending on how you split the data between the SQL and LDAP profile repositories, you may have to configure the following Personalization module components to work with your linked repository setup.

PropertyManager Component

The `/atg/userprofiling/PropertyManager` component keeps track of the property names of "well known" profile properties, such as `login`, `password`, and `email`. If any of these properties now reside in

LDAP, you should update their names to reflect this change. For example, if the password property is stored in LDAP, its property path becomes ldapUser.password, rather than password. Thus, you should configure the passwordPropertyName property of the PropertyManager component as follows:

```
passwordPropertyName=ldapUser.password
```

See the atg.userprofiling.PropertyManager class in the *ATG API Reference* for a complete list of PropertyManager property names that must be updated if the corresponding properties are stored in LDAP rather than SQL.

If a property is stored in both LDAP and SQL, such as the login property in our example, you do not need to modify the corresponding PropertyManager property. The PropertyManager simply uses the SQL property name.

In addition, if users are being authenticated through the LDAP repository, you must modify the PropertyManager from its standard configuration to match the LDAP password encryption scheme. See the *Configuring Personalization Server to Use the LDAP Repository* section of the *Setting Up an LDAP Profile Repository* chapter for more information.

ProfileItemFinder Component

The /atg/userprofiling/ProfileItemFinder component is used by the Personalization module to locate users given the values of their login, password, first name, last name, or email attributes. For example, when a user logs in using a profile form handler, the supplied username and password are passed on to ProfileItemFinder in order to locate the user in the profile repository.

By default, ProfileItemFinder is configured as a component of type atg.userprofiling.RepositoryProfileItemFinder. The RepositoryProfileItemFinder class finds a profile by performing a query against the profile repository. However, with the profile data split between the SQL and LDAP repositories, ProfileItemFinder may need to query the SQL repository in some cases and LDAP repository in others. For example, if the login and password properties are stored in LDAP, but the email property is stored in SQL, then the ProfileItemFinder must query the LDAP repository to find users by login/password, but it must query the SQL repository to find users by email.

When your profile data is split between two linked repositories, you should override the ProfileItemFinder configuration so that it uses the class atg.userprofiling.RepositoryLinkProfileItemFinder. This class extends RepositoryProfileItemFinder and allows queries to be performed against both repositories.

With this configuration, the ProfileItemFinder component has the following properties:

Property	Description
profileTools	A pointer to the /atg/userprofiling/ProfileTools component.
propertyManager	A pointer to the /atg/userprofiling/PropertyManager component.

`profileRepository`	The main profile repository. This is set to the SQL repository component, `/atg/userprofiling/ProfileAdapterRepository`, by default.
`linkedRepository`	The repository that is linked to the main profile repository. Set this to your LDAP repository component, for example `/atg/adapter/ldap/LDAPRepository`.
`linkedItemProperty`	The name of the property in the SQL profile repository that points to the item in the linked repository. For example, if your SQL repository has an `ldapUser` property that points to the LDAP item, set `ldapUser` as the value here.
`linkedProfileTypeMap`	The mapping between the main profile repository's profile types and the corresponding linked repository's profile types. For example, if both your SQL and LDAP repositories have a single `user` type, you should set `linkedPropertyTypeMap=user=user`.
`linkPropertyLocal`	The name of the property used to link against in the SQL repository. This name is the same as the value of the `uniqueIdPropertyLocal` attribute in the `ldapUser` property definition.
`linkPropertyRemote`	The name of the property used to link against in the LDAP repository. This name is the same as the value of the `uniqueIdPropertyRemote` attribute in the `ldapUser` property definition.
`loginLinkedPropertyName`	The name of the `login` property in the linked repository, null if the `login` property is stored in the main profile repository. For example, if the `login` property is stored in LDAP, this name should be `loginLinkedPropertyName=login`. If the `login` property exists in both repositories (as in our example), only set this property if the `password` property is also stored in LDAP. That way, the `ProfileItemFinder` will be able to locate users in the linked repository given their login/password.
`passwordLinkedPropertyName`	The name of the `password` property in the linked repository, null if the `password` property is stored in the main profile repository.
`firstNameLinkedPropertyName`	The name of the `firstName` property in the linked repository, null if the `firstName` property is stored in the main profile repository.
`lastNameLinkedPropertyName`	The name of the `lastName` property in the linked repository, null if the `lastName` property is stored in the main profile repository.

`emailLinkedPropertyName`	The name of the `email`property in the linked repository, null if the `email` property is stored in the main profile repository.
`createLocalProfiles`	If true, and a profile is found in the linked repository but not in the main profile repository, the local profile is created, with its link property (specified via `linkPropertyLocal`) set accordingly. Otherwise, the local profile is not created, and the item will not be returned by the search. The default is false. This property is useful in a situation where you have an existing directory of LDAP users, and you would like the corresponding SQL profiles to be created automatically when the users log in. Since the `ProfileItemFinder` is used to find the user during login, setting this property to true forces the SQL profile to be created the first time a user logs in.

8 Creating Rules for Targeting Content

Once you have set up your content repository, you can create **rules** that match repository items with user profiles. The recommended way to create content targeting rules is to use the Targeting and Segmentation interface in the ATG Business Control Center. For more information, refer to the *ATG Business Control Center User's Guide*. If your installation does not include the ATG Business Control Center, you can use the Content Targeters window of the Content task area in the ACC. See the *Matching Content with Your Target Audiences* chapter in the *ATG Personalization Guide for Business Users*.

You can also create targeting rules by hand. You can define the Personalization module's targeting rules either as a property of a rule set service or in a separate **rules file**. A rules file is a text file with a `.rules` extension that you create to specify targeting rules in Personalization module applications. The Personalization module's rule sets use SGML format with special tags to describe the rules. This chapter explains how to write rule sets. It includes the following sections:

Elements of Rule Sets

A rule set is made up of *rules* and *sorting directives*. The rules define the possible matches between items in your repository (documents) and a user (based on the user's profile attributes). Rules are either *accept rules* or *reject rules*. If an accept rule is true with respect to a target object (a document in your repository), then that target object is included in the targeting results. If a reject rule is true with respect to a target document, then that target object is excluded from the targeting results.

Targeting rules use *operators*, like “and,” “equals,” and “includes,” to establish relationships between properties of target objects (the documents or other items in your repository) and properties of source objects (such as the profiles of your users). See Rule Tag Operations for details about all the operators you can use in a rule set.

Accept Rules

The accept rules in your rule set determine what content should be delivered to your users. The rules test properties of the content items and properties of your user's profile. If the relationships among these properties result in a match, then the system delivers the content item to the user.

Example: Targeting Content

For example, suppose you have a repository that includes news items about companies in different industrial sectors, and you want to deliver items about companies in the rubber industry sector to users whose profiles identify them as analysts who cover the rubber industry. First, you can write a rule that identifies which of the items in your repository relate to the rubber industry, like this:

```
<rule op=eq name="Rubber sector">
  <valueof target="industrySector">
  <valueof constant="rubber">
</rule>
```

This rule is true with respect to a document if the `industrySector` property of the document has the value "rubber." When you use the `target` type in a `<valueof>` tag, you are testing properties of items in your repository. See <valueof> Tag.

Example: Targeting Users

Next, you can write a rule that identifies whether one of the industry sectors covered by the user is the rubber industry, like this:

```
<rule op=includes name="Rubber analysts">
  <valueof bean="Profile.sectorsCovered">
  <valueof constant="rubber">
</rule>
```

This rule is true with respect to a user if the `sectorsCovered` property of his or her profile includes the value "rubber." The `<valueof bean="...">` tag points to the user's profile and tests the indicated property against the constant in the next tag. See <valueof> Tag.

Example: A Complete Accept Rule

Now, you can put the two rules together in an **and** rule within the `<accepts>` tag to match the content to the user:

```
<accepts>
  <rule op=and>
    <rule op=eq name="Rubber sector">
      <valueof target="industry sector">
      <valueof constant="rubber">
    </rule>
    <rule op=includes name="Rubber analysts">
      <valueof bean="Profile.sectorsCovered">
      <valueof constant="rubber">
    </rule>
  </rule>
</accepts>
```

Reject Rules

The accept rules in the previous example defined the content that the targeting operation should include. You can modify your rule set to *exclude* content, using reject rules. For example, suppose you wanted to deliver content only about rubber companies outside the United States. You could add a reject rule to your rule set like this:

```
<rejects>
  <rule op=eq name="U.S. companies">
    <valueof target="companyCountry">
    <valueof constant="United States">
  </rule>
</rejects>
```

This reject rule is true if the value of the companyCountry property of a document is “United States”. Any document that this rule identifies as relating to a U.S. company is excluded from the results of the targeting operation, and is *not* displayed to the user.

Combining the Accept Rules and Reject Rules

When a targeting operation specified by a given rule set is performed, the Personalization module combines the accept and reject rules as follows to obtain the overall targeting rule:

```
(AND (OR <accept rules>)
     (NOT (OR <reject rules>)))
```

In other words, a target object satisfies the overall targeting rule if it satisfies the accept rules but does not satisfy the reject rules.

Sorting Directives

A rule set can also include *sorting directives*, which determine the order in which targeting results should be returned for display, based on properties of the target objects. The application can then either display the targeting results in that order, or select items from the targeting results based on the order created by the sorting directives. For example, to sort the results of the previous example by company name, in alphabetical order, you write a sorting directive like this and add it to your rule set:

```
<sortby>
  <sortbyvalue value="company" dir="ascending">
</sortby>
```

See <sortby> Tag and <sortbyvalue> Tag for details about sorting directives.

Example: A Complete Rule Set

Here is an example of a complete rule set, putting together all the elements discussed in this section:

```
<ruleset>
<!-- This rule delivers items about non-U.S. rubber companies to analysts who
     cover the rubber industry -->

<!-- Accept rules -->

<accepts>
  <rule op=eq name="Rubber sector">
    <valueof target="industry sector">
    <valueof constant="rubber">
  </rule>
  <rule op=includes name="Rubber analysts">
    <valueof bean="Profile.sectorsCovered">
    <valueof constant="rubber">
  </rule>
</accepts>

<!-- Reject rule -->

<rejects>
  <rule op=eq name="U.S. companies"
    <valueof target="companyCountry">
    <valueof constant="United States">
  </rule>
</rejects>

<!-- Sorting Directive -->

<sortby>
  <sortbyvalue value="company" dir=ascending>
</sortby>
</ruleset>
```

Including Elements from Other Sources

Rule sets can also include rules and sorting directives from other sources, or even incorporate an entire other rule set. See Including Rule Sets, Rules, and Sorting Directives.

Rule Set Structure

A rule set has the following basic structure:

```
<ruleset>
  <accepts>
    <rule ...> ... </rule>
```

```
      ...
    </accepts>
    <rejects>
      <rule ...> ... </rule>
      ...
    </rejects>
    <includes>
      <ruleset src=...> ... </ruleset>
      ...
    </includes>
    <sortby>
      <sortbyvalue ...>
      ...
    </sortby>
    <site type=...>
    </site>
  </ruleset>
```

The only SGML tags allowed in a rule set are the following:

- `<ruleset></ruleset>`
- `<accepts></accepts>`
- `<rejects></rejects>`
- `<includes></includes>`
- `<rule></rule>`
- `<valueof>`
- `<sortby></sortby>`
- `<sortbyvalue>`
- `<site></site>`

No other SGML constructs are allowed, except for comments. Comments may occur anywhere, except within tags themselves.

A rule set contains one `<ruleset>` tag. The `<ruleset>` tag must have at least one and at most five child tags: one of `<accepts>`, `<rejects>`, or `<includes>` tags must be present. The `<sortby>` and `<site>` tags are optional. Multiple `<accepts>`, `<rejects>`, `<includes>`, `<sortby>`, and `<site>` tags are not allowed.

The `<accepts>` and `<rejects>` tags must contain one or more `<rule>` tags. A `<rule>` tag must include an **op** (operation) attribute and one or more `<valueof>` tags. You can create complex rules by nesting multiple `rule` tags within a `<rule>` tag.

You can also create complex rules by incorporating other rule sets with the `<includes>` tag. An `<includes>` tag contains references to other rule sets to be included in the rule set. It must have at least one child; each child must be a `<ruleset>` tag with a `src` attribute. See Including Rule Sets, Rules, and Sorting Directives for details. The `<ruleset>` tag may also contain a `<sortby>` tag, which in turn may contain one or more <sortbyvalue> tags to establish sorting directives. The <ruleset> tag may also include a <site> tag, which specifies the Web site to use as a filter (applies to multisite ATG environments).

Rules Tag Syntax

The function and syntax of each of these tags is described below:

- <ruleset> Tag
- <accepts> Tag
- <rejects> Tag
- <includes> Tag
- <rule> Tag
- Rule Tag Attributes
- Rule Tag Operations
- <valueof> Tag
- <sortby> Tag
- <sortbyvalue> Tag
- <site> Tag

<ruleset> Tag

The <ruleset> tag is the required top-level parent tag. It must have at least one and at most four child tags: At least one of either an <accepts> tag, a <rejects> tag, or an <includes> tag must be present; the <sortby> tag is optional. Multiple <accepts>, <rejects>, <includes>, or <sortby> tags are not allowed.

A <ruleset> tag can also be used inside an <includes> tag. All <ruleset> tags used inside an <includes> tag must use the src attribute to specify what rule set is to be included. See <includes> Tag and Including Rule Sets, Rules and Sorting Directives for details.

The <ruleset> tag must be matched by a closing </ruleset> tag.

<accepts> Tag

The <accepts> tag is a container for all the *accept rules* in the rule set. If an accept rule is true with respect to a target object (a document in your repository), then that target object is included in the targeting results (unless the target object is rejected by a reject rule). The <accepts> tag must have at least one child; each child must be a <rule> tag.

The <accepts> tag must be matched by a closing </accepts> tag.

<rejects> Tag

Similarly, the `<rejects>` tag is a container for the *reject rules*. If a `reject` rule is true with respect to a target document, then that target object is excluded from the targeting results. Like the `<accepts>` tag, the `<rejects>` tag has at least one `<rule>` child.

The `<rejects>` tag must be matched by a closing `</rejects>` tag.

<includes> Tag

You may want to include entire rule sets within another rule set. You can do this with the `<includes>` tag. The `<includes>` tag has the following syntax:

```
<includes>
  <ruleset src="{rule set name}"></ruleset>
   ...
</includes>
```

Each `<includes>` tag has one or more child tags. Each child tag must be a `<ruleset>` tag, and each `<ruleset>` tag must use the `src` attribute to specify the Nucleus path of the rule set to include. See Including Rule Sets, Rules, and Sorting Directives for more information.

The `<includes>` tag must be matched by a closing `</includes>` tag.

<rule> Tag

The `<rule>` tag has the following syntax:

```
<rule op={operation} [src={path}][name={name}] [tag={tag}]>
  {child tags}
</rule>
```

or this:

```
<rule src={path}></rule>
```

The `<rule>` tag must be matched by a closing `</rule>` tag.

Rule Tag Attributes

The `<rule>` tag can take four attributes:

- op
- src
- name
- tag

A rule must include either an op attribute or a `src` attribute. The op attribute specifies an operation to be performed on the child tags, which must be either `<rule>` or `<valueof>` tags. Rule tag operations define a relationship between properties, components, or other rules. The allowed operations are described below, in Rule Tag Operations.

The `src` attribute allows you to incorporate rules that have already been defined in another rule set. For more information about using the `src` attribute, see Including Rule Sets, Rules, and Sorting Directives.

The `name` and `tag` attributes are optional; they are used by the ATG Business Control Center and the ACC to label a rule and attach any client data to it.

Referring to Profile Attributes in Rules

The Personalization module uses, by default, a `TargetingSourceMap` service that maps user profiles to the name "Profile." Assuming your profiles have a Nucleus address of `/atg/userprofiling/Profile`, a rule can therefore refer to the `name` property of a profile with `bean="Profile.name"` instead of `bean="/atg/userprofiling/Profile.name"`. See the Setting Up Targeting Services chapter for more information.

Rule Tag Operations

You can use any one of the following operators in a `<rule>` tag. The operator strings are case-insensitive; for example, `startsWith`, `STARTSWITH`, and `startswith` are equivalent.

Note that some of these operators are not supported by certain repositories. For example, HTML and XML repositories do not support the `includesAll` operator. For more information, see the *ATG Repository Guide*.

- `eq, neq, lt, gt, lteq, gteq`
- `contains, startsWith, endsWith`
- `includes, notIncludes`
- `includesAny, notIncludesAny`
- `includesAll, notIncludesAll`
- `isOneOf, isNotOneOf`
- `includesItem`
- `and, or, not, any`
- `matchId`
- `elementAt`
- `indexOf`
- `count`
- `inSchedule`
- `inFolders`

- isBetween, isNotBetween
- isNull, isNotNull
- textSearch

The purpose and syntax of each of the rule tag operators is discussed below, with examples of each.

eq, neq, lt, gt, lteq, gteq

Represent an equality operation on the child tags.

Operator string	Operation
eq	equal to
neq	not equal to
lt	less than
gt	greater than
lteq	less than or equal to
gteq	greater than or equal to

Rules with each of these operators must have exactly two child tags. For eq and neq operators, there is no restriction on the types of child tags; however, they are generally expected to be compatible (i.e., one shouldn't compare a Boolean value to an Integer). If the operation is not either eq or neq, the children must both be either Numbers, Strings or Dates.

Example: Using the lteq operator with a Date Constant - This example rule tests whether the `lastLogin` property indicates that the last time the user logged in was before 1998.

```
<rule op=lteq>
  <valueof bean="Profile.lastLogin">
  <valueof constant="12/31/97 11:59 pm">
</rule>
```

Example: Using the gt operator – This example rule tests whether the user has more money than sense.

```
<rule op=gt>
  <valueof bean="Profile.money">
  <valueof bean="Profile.sense">
</rule>
```

contains, startsWith, endsWith

Represent a string pattern matching operation on the child tags (i.e., does the first string contain, start with, or end with the second string?). Rules with the `contains`, `startsWith`, or `endsWith` attributes must have exactly two child tags. Both child tags must represent String values.

Example: Comparing strings with the startsWith operator – This example rule tests whether the `longestJourney` property in a profile starts with the string “a single step”.

```
<rule op=startsWith>
    <valueof bean="Profile.longestJourney">
    <valueof constant="a single step">
</rule>
```

Note that these string matching operations are case-sensitive. In the example above, the rule would not find a profile whose `longestJourney` property starts with “A single step”. For case-insensitive matching, you can use the `containsIgnoreCase`, `startsWithIgnoreCase`, and `endsWithIgnoreCase` operators.

includes, notIncludes

Represents a membership operation on the child tags (i.e., does the first child contain the second child as a member?). Rules with the `includes` or `notIncludes` operator must have exactly two child tags. The first child must represent an array or a Vector object. The second child can represent any non-null Object value. Note that not every element of the first child has to be of the same type as the second child; elements of different types are simply ignored during membership testing.

Example: includes Operator with a Constant Array – In this example rule, `[book, article, thesis]` is a constant array expression. The rule tests whether the value of the `docType` property of the target repository item is one of `book`, `article`, or `thesis`.

```
<rule op=includes>
  <valueof constant="[book, article, thesis]">
  <valueof target="docType">
</rule>
```

includesAny, notIncludesAny

Compares the memberships of the child tags. An `includesAny` rule evaluates as true if at least one of the members of the second child tag is also a member of the first child tag. A `notIncludesAny` rule evaluates as true if none of the members of the second child tag is a member of the first child tag. Rules with the `includesAny` or `notIncludesAny` operator must have exactly two child tags, both of which are array or Vector objects.

Example: includesAny Operator – In this example rule, `keywords` and `favoriteSubjects` are each String arrays. The rule tests whether any of the values of the `keywords` property of the target repository item match any of the values of the `favoriteSubjects` property of the profile.

```
<rule op=includesAny>
  <valueof bean="Profile.favoriteSubjects">
  <valueof target="keywords">
</rule>
```

includesAll, notIncludesAll

Compares the memberships of the child tags. An includesAll rule evaluates as true if all of the members of the second child tag are also members of the first child tag. A notIncludesAll rule evaluates as true if at least one of the members of the second child tag is not a member of the first child tag. Rules with the includesAll or notIncludesAll operator must have exactly two child tags, both of which are array or Vector objects.

Example: includesAll Operator – In this example rule, keywords and favoriteSubjects are each String arrays. The rule tests whether all of the values of the keywords property of the target repository item are also members of the value of the favoriteSubjects property of the profile. If any values of the keywords property are not found in the favoriteSubjects property, the rule returns false.

```
<rule op=includesAll>
  <valueof bean="Profile.favoriteSubjects">
  <valueof target="keywords">
</rule>
```

isOneOf, isNotOneOf

The same as includes, notIncludes, but with the order of the child tag arguments reversed. Represents a membership operation on the child tags (i.e., is the first child a member of the second child?). Rules with the isOneOf or isNotOneOf operator must have exactly two child tags. The first child can represent any non-null Object value. The second child must represent an array or a Vector object. Note that not every element of the second child has to be of the same type as the first child; elements of different types are simply ignored during membership testing.

Example: isOneOf Operator with a Constant Array – In this example rule, [book, article, thesis] is a constant array expression. The rule tests whether the value of the docType property of the target repository item is one of book, article, or thesis.

```
<rule op=isOneOf>
  <valueof target="docType">
  <valueof constant="[book, article, thesis]">
</rule>
```

includesItem

Evaluates to true if the Collection property contains an element that matches the query.

For example, if the item has an addresses property that contains multiple addresses, the following rule returns true for all items whose addresses property includes an address with the zip code 90210.

```
<rule op="includesItem">
  <valueof target="addresses">
  <rule op="eq">
    <valueof target="postalCode">
    <valueof constant="90210">
  </rule>
</rule>
```

and, or, not, any

Represent a logical **and**, **or**, or **not** operation on the child tags. Rules that use the **and** or the **or** operators must have at least one child tag. Rules that use the **not** operator must have exactly one child tag. Each of the child tags must represent a Boolean expression. The **any** operator is the same as the **or** operator, except that if the **any** operator is used in a rule with no child tags, then the rule evaluates as true.

matchId

Represents a lookup operation on the child tags (i.e., return all the items that have the IDs specified). May have one or more child tags. Each child tag must be a `<valueof constant="...">` tag that specifies a valid ID for the content repository.

Example: `matchId` Operator – This rule returns items whose ID is either 14427 or 14428:

```
<rule op=matchId>
    <valueof constant="14427">
    <valueof constant="14428">
</rule>
```

elementAt

Represents a lookup operation on the child tags (i.e., get me the element from the second child at the index specified by the first child). Must have exactly two child tags. The first child must represent an Integer object. The second child must represent a value that will eventually evaluate to an array or a Vector object. Rules that use the `elementAt` operator cannot include subrules that use the `<valueof target="...">` tag.

indexOf

Represents a reverse lookup operation on the child tags (i.e., what is the index of the first child in the second child). Rules using the `indexOf` attribute must have exactly two child tags. The first child can represent any non-null Object value. The second child must represent a value that will eventually evaluate to an array or a Vector object. Rules that use the `indexOf` operator cannot include subrules that use the `<valueof target="...">` tag. Note that not every element of the second child has to be of the same type as the first child; elements of different types are simply ignored during equality testing. The tag itself eventually evaluates to an Integer object (with a value of -1 if the element is not found in the array or Vector).

Example: `indexOf` Operator - This example rule demonstrates the use of the `indexOf` operator:

```
<rule op=lteq>
  <!-- minimum clearance level required -->
  <valueof constant="5">
  <rule op=indexOf>
    <valueof bean="Profile.securityClearance">
    <valueof constant="[uncleared, confidential, restricted,
                        classified, critical_sensitive,
                        secret, top_secret]">
  </rule>
</rule>
```

count

Represents a count, or size, operation on the child tag (i.e., what is the number of elements in the child?). Rules using the `count` operator must have exactly one child tag. The child tag must represent a value that will eventually evaluate to an array or a Vector object. Rules that use the `count` operator cannot include subrules that use the `<valueof target="...">` tag. The tag itself eventually evaluates to an Integer object.

Example: count Operator - This example demonstrates the `count` operator. This rule assumes you have a `pagesViewed` property in your Profile that stores an array of site pages, and evaluates as `true` if the user has viewed three or fewer pages.

```
<rule op=lteq>
  <valueof constant="3">
    <rule op=count>
      <valueof bean="Profile.pagesViewed">
    </rule>
</rule>
```

inSchedule

Represents an operation that determines whether the given time occurs in the given schedule. This attribute is used for such things as targeting to specific dates or days of the week. Rules using the `inSchedule` operator must have exactly two child tags. The first child tag must represent a value that will eventually evaluate to a Date object. The second child tag must represent a value that will eventually evaluate to a `CalendarSchedule` object. See the *Scheduler Services* section in the *Core Dynamo Services* chapter of the *ATG Programming Guide* for information about the syntax for `CalendarSchedule` objects. Rules that use the `inSchedule` operator cannot include subrules that use the `<valueof target="...">` tag.

Example: inSchedule Operator - This example demonstrates the `inSchedule` operator. This rule assumes you have a `/atg/Calendar` object of class Date, and evaluates as `true` if `/atg/Calendar.today` is a Monday in January, February, or March:

```
<rule op=inSchedule>
  <valueof bean="/atg/Calendar.today">
  <valueof constant="0-2 * 1 * *">
</rule>
```

inFolders

Represents a lookup operation (i.e., is the item in any of the folders specified in the child tags. The child tags represent folder IDs. The exact syntax and semantics of a folder ID depend on the type of content repository.

Example: inFolders Operator - This example demonstrates the `inFolders` operator. This rule evaluates as true for items found in either the `someFolder` or the `anotherFolder` folders:

```
<rule op=inFolders>
  <valueof constant="someFolder">
  <valueof constant="anotherFolder">
</rule>
```

isBetween, isNotBetween

Represents a range operation on the first child tag (i.e., does the first child tag fall between the second and third child tag?). Rules using the `isBetween` or `isNotBetween` operators must have exactly three child tags. The children must be Numbers, Strings or Dates. This is an inclusive operation; in other words, a rule evaluates to true if the first child tag is greater than or equal to the second child tag and less than or equal to the third child tag.

Example: isBetween Operator – This example demonstrates the `isBetween` operator. This rule evaluates to true if the value of the `age` property is between 21 and 55, inclusive.

```
<rule op=isBetween>
  <valueof bean="Profile.age">
  <valueof constant="21">
  <valueof constant="55">
</rule>
```

isNull, isNotNull

Represents a null check operation on the child tag. Rules using the `isNull` or `isNotNull` operator must have exactly one child tag. This tag eventually evaluates to a Boolean object.

Example: isNull Operator – This example demonstrates the `isNull` operator. This rule evaluates to true if the value of the `birthday` property is null.

```
<rule op=isNull>
  <valueof bean="Profile.birthday">
</rule>
```

textSearch

Takes three or four rule nodes. The first three represent the search pattern, the search string format, and the minimum score for matches. The fourth one (which is optional) is a target rule node that specifies a property to search against. For example:

```
<rule op="textsearch">
    <valueof constant="denim">
    <valueof constant="ORACLE_CONTEXT">
    <valueof constant="50">
    <valueof target="description">
</rule>
```

<valueof> Tag

A `<rule>` tag can contain one or more `valueof` tags, depending on the rule's operation attribute. The `<valueof>` tag has the following syntax

```
<valueof {type}="{string}">
```

Note that the `<valueof>` tag does not take an end tag.

valueof Types

You can use the following types and attributes in a `<valueof>` tag:

- target
- constant
- bean

target

Represents a property of the repository item object (for example, an HTML page in the repository) that is a potential member of the result set of a targeting operation.

Example: `<valueof target="author">` could refer to the `author` property of a document in the repository. This tag might be used to test whether the author of a repository item is the same person as the visitor in a rule like this:

```
<rule op=eq>
  <valueof bean="Profile.name">
  <valueof target="author">
</rule>
```

constant

Represents a constant value, which does not require evaluation (for example, “42”, “true”, “snowboarding”, “4/10/98 5:59 PM” or “[1, 2, 3]”, which is how an array with constant elements is represented). The string portion of the tag is parsed sequentially to determine its data type, in the following priority.

1. The tag is first parsed as an Integer. If that fails, the tag is parsed in turn as a:
2. Double
3. Boolean
4. Date (in the same way that Date objects are parsed in `.properties` files)
5. `CalendarSchedule`. See the *Scheduler Services* section in the *Core Dynamo Services* chapter of the *ATG Programming Guide* for information about the syntax for `CalendarSchedule` objects.
6. Array (assuming a syntax like [1, 2, 3], as in the example above)
7. String

Example: `<valueof constant="18">`. This tag might be used to compare an age property in a user’s profile to an Integer constant in a rule like this:

```
<rule op=lt>
  <valueof bean="Profile.age">
  <valueof constant="18">
</rule>
```

This rule is satisfied if the age property of the source profile is less than the integer constant, 18.

Example: `<valueof constant="[Maine, New Hampshire, Vermont, Massachusetts, Connecticut, Rhode Island]">`. This tag might be used to determine whether the `state` property in a user’s profile is one of the New England states included in a constant array, in a rule like this:

```
<rule op=includes>
  <valueof constant="[Maine, New Hampshire, Vermont, \
                      Massachusetts, Connecticut, Rhode Island]">
  <valueof bean="Profile.state">
</rule>
```

Note the use of the backslash (\) character as a line continuation character in the `<valueof constant="...">` tag.

bean

Represents a Nucleus component or a property value of a Nucleus component. The syntax of the string portion of the tag is: `{component}.{property}`. The string can include more than one property, separated by dots.

You typically use this type to represent a property of the user profile object you are targeting a rule set against.

Example: `<valueof bean="/atg/userprofiling/Profile.gender">` could refer to the `gender` property of a profile bean. This tag might be used to test whether a visitor is female in a rule like this:

```
<rule op=eq>
  <valueof bean="/atg/userprofiling/Profile.gender">
  <valueof constant="female">
</rule>
```

This rule is satisfied if the value of the `gender` property of the source profile is the string constant `female`.

The default configuration of the `TargetingSourceMap` service maps the Nucleus path of `/atg/userprofiling/Profile` to the source name “Profile.” This lets you simplify tags like the previous example to `<valueof bean="Profile.gender">`. See the Setting Up Targeting Services chapter for more information.

Using Indexed Property Values

For `bean` and `target` types, you can also refer to indexed property values. For example, each of the following is valid:

```
<valueof bean="/atg/Test/Person.hobbies[0]">
<valueof target="keywords[bean: /atg/Test/Person.favoriteKeyword]">
```

As in regular `<valueof>` tags in JHTML files, you can use `bean:` strings inside tags to refer to properties. However, you cannot use `param:` strings in a targeting rule’s `<valueof>` tag.

Target Nodes Inside Boolean Expressions

You cannot have rules that use the `<valueof target="...">` tag as direct children of rules that use the `and`, `or`, or `not` operators, even if the target property is supposed to eventually evaluate to a Boolean value. For example, this rule is invalid:

```
<rule op=or name="Wrong!">
  <valueof target="isAvailable">
  <valueof bean="Profile.isAuthor">
</rule>
```

Instead, you must do the following:

```
<rule op=or>
  <rule op=eq>
    <valueof target="isAvailable">
    <valueof constant="true">
  </rule>
  <valueof bean="Profile.isAuthor">
</rule>
```

<sortby> Tag

The `<sortby>` tag is a container for all the *sorting directives*, which specify how the targeting results should be sorted. You can include a sorting directive that has been previously defined, or which is defined dynamically, using the `src` attribute with the `<sortby>` tag. See Including Rule Sets, Rules, and Sorting Directives.

The `<sortby>` tag must either use the `src` attribute or have at least one child. Each child must be a `<sortbyvalue>` tag. The order of the child `<sortbyvalue>` tags implies the sorting order: that is, the first child defines the primary sorting criterion, the second child defines the secondary sorting criterion, and so forth.

The `<sortby>` tag must be matched by a closing `</sortby>` tag.

<sortbyvalue> Tag

The `<sortby>` tag includes one or more `<sortbyvalue>` tags, which you can use to specify a property of the target object to be used in sorting the results of a targeting operation. The `<sortbyvalue>` tag has the following syntax:

```
<sortbyvalue value="{string}" [dir={direction}]>
```

The `value` attribute specifies the name of the property of the target object to sort by. The optional `dir` attribute specifies the sorting direction, one of `ascending` or `descending`; if unspecified, the direction defaults to `ascending`.

Example - This sorting directive indicates that the targeting results should be sorted by the `lastModified` property of the target objects, in descending order.

```
<sortbyvalue value="lastModified" dir=descending>
```

Note that the `<sortbyvalue>` tag does not take an end tag.

<site> Tag

The optional `<site>` tag is designed for use in environments that support several Web sites. It identifies the Web site to use as a filter for the targeting operation. If no `<site>` tag is specified, the query automatically applies to the current site (the site the user is visiting). For more information on multisite environments, refer to the *ATG Multisite Administration Guide*.

The `<site>` tag has the following syntax:

```
<site type=...>
  ...
</site>
```

The `<site>` tag must be matched by a closing `</site>` tag.

This tag has one attribute, `type`, which can have the following values:

- `current`
- `any`
- `shareable`

current

Specifies the current site (the site the user is visiting) as the filter for the query.

```
<site type=current></site>
```

Items that have no site membership are included in a query if the value is `current`. You can also include these items by omitting the `<site>` tag and making sure the user has an empty site context.

any

The any setting can be used alone, in which case all registered sites are included in the targeting query. It can also be used with one or more child tags that identify specific sites to include in the query (effectively a list of sites to use). Only items that have at least one site specified will be included. If an item has no site membership, it will be ignored.

```
<site type=any></site>
```

Or

```
<site type=any>
  <valueof site="{siteId1}">
  <valueof site="{siteId2}">
</site>
```

The `siteId` properties are strings stored in `/atg/multisite/SiteRepository`. The following example shows the `siteIds` for ATG Store US and ATG Store Germany from the ATG Commerce Reference Store application:

```
site type=any>
  <valueof site="storeSiteUS">
  <valueof site="storeSiteDE">
</site>
```

For more information on the `siteId` property, refer to the *ATG Multisite Administration Guide.*

shareable

Identifies one or more shareable types to use for the targeting query. Shareable types are resources such as shopping carts that can be shared among sites in a multisite environment. In this case, the query for the targeting operation includes only sites that are configured to share the specified type. For example, the targeter could look for content only in sites that share a shopping cart.

```
<site type=shareable>
  <valueof constant="{shareableTypeId}">
</site>
```

The first example includes all sites that share a shopping cart with the current site. The `shareableTypeId` is from ATG Commerce Reference Store:

```
<site type=shareable>
  <valueof constant="atg.shoppingCart">
</site>
```

The shareable type can also exist in a site other than the current site. The second example includes all sites that share a shopping cart with ATG Store Germany:

```
<site type=shareable>
  <valueof constant="atg.shoppingCart">
  <valueof site="storeSiteDE">
</site>
```

Null Values in Rules

Any null value encountered by a targeter is interpreted to mean “this value is unknown.” If any such unknown values are encountered during rule evaluation or query execution, the corresponding repository items are not included in the targeting results. For example, consider the rule

```
<rule op=eq>
  <valueof bean="Profile.ageGroup">
  <valueof target="ageGroup">
</rule>
```

This rule might encounter null values in two cases:

- If `Profile.ageGroup` evaluates to null, we deduce that the user did not specify his age, and his `ageGroup` is therefore unknown. In this case, no content should be returned by this rule.
- If `Profile.ageGroup` evaluates to `Teenagers`, the only items returned from the repository query should be the ones with `ageGroup` equal to `Teenagers`. A content item with a null `ageGroup` value should not be returned, since its `ageGroup` property is considered to be unknown.

The less intuitive case is the case with rules that use negative rule operators, such as neq. For example:

```
<rule op=neq>
  <valueof bean="Profile.ageGroup">
  <valueof target="forbiddenAgeGroup">
</rule>
```

Here again,

- If Profile.ageGroup evaluates to null, the ageGroup is unknown, and no content is returned.
- If Profile.ageGroup evaluates to Teenagers, the only items returned from the repository query should be the ones which have the forbiddenAgeGroup property set to something other than Teenagers - but not null. Items whose forbiddenAgeGroup property is null should **not** be returned, since their forbiddenAgeGroup is unknown.

The principle behind this treatment of null values is that when the user doesn't specify a value for one of his properties, or when the content creator does not specify a value for one of the content attributes, that unspecified value in actuality may or may not match the rule condition we are trying to satisfy. If there is a possibility that the condition may not be satisfied, we always err on the conservative side and do not include the questionable people or content. If you want to include items with null values, then add a term to the rule with an **or** rule operator, like this:

```
<rule op=or>
  <rule op=eq>
    <valueof bean="Profile.ageGroup">
    <valueof target="ageGroup">
  </rule>
  <rule op=isNull>
    <valueof target="ageGroup">
  </rule>
</rule>
```

This rule returns all content items whose ageGroup attribute matches the profile's ageGroup, and also all content items whose ageGroup attribute is null.

Creating a Rule Set for a Profile Group that Includes Roles

You can use global and organizational roles to define membership of a profile group. For example, you can create a profile group that includes anyone assigned the global role "administrator." You can then use that profile group within a content targeter, effectively allowing you to deliver personalized content according to the role of the person viewing the page.

The following example shows a rule set for a profile group that includes a global role named admin.

```
<rule op="includesItem">
   <valueof target="roles">
```

```
        <rule op="or">
           <rule op="and">
               <rule op="equals">
                   <valueof target="name">
                   <valueof constant="admin">
               </rule>
               <rule op="equals">
                  <valueof target="type">
                  <valueof constant="role">
               </rule>
           </rule>
        </rule>
</rule>
```

The `<value of target="type">` setting determines whether the role is global or organizational, allowing the Personalization module to distinguish between a global role and an organizational role that have the same name.

Note: If you want to use global roles within rule sets, as shown above, all global role names must be unique. For more information, refer to *Adding a New Role* in the *ATG Personalization Guide for Business Users*.

The same restriction does not apply to organizational roles, which need to be unique only within their organization.

The following example shows a rule set for a profile group that includes an organizational (relative) role named `buyer`. This role belongs to an organization named `MyCompany`.

```
<rule op="includesItem">
   <valueof target="roles">
        <rule op="or">
            <rule op="and">
                  <rule op="eq">
                    <valueof target="name">
                    <valueof constant="buyer">
                  </rule>
                  <rule op="eq">
                    <valueof target="relativeTo.name">
                    <valueof constant="MyCompany">
                  </rule>
            </rule>
        </rule>
</rule>
```

Note also that you can create profile groups that include roles by using the Targeting > Profile Groups window in the ACC. However, the ACC supports the creation of `<rule op=or>` rules only; in other words, if you create a profile group that includes people assigned to Role A and Role B, everyone assigned either role will be a member of the profile group. If you want to create rules that use other operators, for example, <rule op=and>, write the rule by hand as described in this chapter. See Rule Tag Operations for more information.

Including Rule Sets, Rules, and Sorting Directives

You can compose rule sets from already defined rules or sorting directives. To include an already-defined rule set or sorting directive in your rule set, you can use the src attribute with the <ruleset>, <rule>, and <sortby> tags. The syntax for the three tags in this form is:

```
<ruleset src={path}></ruleset>
<rule src={path}></rule>
<sortby src={path}></sortby>
```

where the *path* is the Nucleus path of the rule, rule set, or sorting directives to be included.

Note that if you include a rule set using the <ruleset src="..."> tag, the tag must be contained by an includes tag. Also, when you include a rule set with the <ruleset src="..."> tag, only the rules from that rule set are included; any sorting directives in the included rule set are ignored. You can, of course, include them using the <sortby src="..."> tag.

Examples: src Attribute

For example, here is a rule set that combines a Males18-24 rule set and a Females18-24 rule set which have been defined elsewhere, and sorts the results according to the sorting criteria dynamically specified by the user. In this example, Males18-24.ruleSet and Females18-24.ruleSet refer to rule sets that you have defined elsewhere, and sortby is a property of the UserPreferences component that can be generated dynamically.

```
<ruleset>
  <includes>
    <ruleset src="/atg/rules/Males18-24.ruleSet"></ruleset>
    <ruleset src="/atg/rules/Females18-24.ruleSet"></ruleset>
  </includes>
  <sortby src="/atg/rules/UserPreferences.sortby"></sortby>
</ruleset>
```

If you don’t want to include the entire rule set into your rules, use the <rule src="..."> tag to include only a particular rule. The <rule src="..."> tag can appear anywhere within the accepts or rejects block of your rule set, not just the top level. For example:

```
<accepts>
  <rule op=and>
    <rule src="IsMale.rules"></rule>
    <rule op=eq>
```

```
      <valueof target="gender">
      <valueof constant="male">
    </rule>
  </rule>
</accepts>
```

Complex Rules Example

Here is an example of a rule set that demonstrates use of many of the tags and operators discussed in this chapter. It also shows how you can form complex rules that combine rules and nest one rule inside another.

The `accept` rules test whether content that is available and whether the `author` property of the repository item matches the `favoriteAuthor` attribute in the source profile.

The `reject` rules exclude content that is rated R if the user profile's age is less than 16 and exclude content that is rated X if the user profile's age is less than 18.

The sorting directive declared in the `<sortby>` tag specifies that the targeting results are to be sorted first by the `lastModified` property, from most recent to oldest, and secondarily by the `name` property of the target objects, in alphabetical order.

When you put it all together, this rule outputs content:

- if it is available AND
- if the content's author matches the profile's favorite author
- but not if the content is rated R and the profile's age is less than 16
- and also not if the content is rated X and the profile's age is less than 18

```
<ruleset>
  <!-- accept rules -->
  <accepts>
    <rule op=and>
      <rule op=eq>
        <valueof target="isAvailable">
        <valueof constant="true">
      </rule>
      <rule op=eq>
        <valueof target="author">
        <valueof bean="Profile.favoriteAuthor">
      </rule>
    </rule>
  </accepts>

  <!-- reject rules -->
  <rejects>
    <rule op=and name="Rated R">
      <rule op=lt>
        <valueof bean="Profile.age">
        <valueof constant="16">
      </rule>
      <rule op=eq>
        <valueof target="rating">
        <valueof constant="R">
      </rule>
    </rule>
    <rule op=and name="Rated X">
      <rule op=lt>
        <valueof bean="Profile.age">
        <valueof constant="18">
      </rule>
      <rule op=eq>
        <valueof target="rating">
        <valueof constant="X">
      </rule>
    </rule>
  </rejects>

  <!-- sorting directives -->
  <sortby>
    <sortbyvalue value="lastModified" dir=descending>
    <sortbyvalue value="name" dir=ascending>
  </sortby>
</ruleset>
```

9 Setting Up Targeting Services

When you create a targeter using the ATG Business Control Center or the ACC, the system automatically sets up the components needed to make your targeting rules work. When you create targeting rules by hand, as described in the previous chapter, you need to create or configure these components yourself. This chapter describes how to set up various components for delivering targeted content.

This chapter contains the following sections:

Setting Up a RuleSetService

To set up a RuleSetService for your rule set, create an atg.targeting.RuleSetService component. This component can reference a rules file, or it can itself include your targeting rules as a property. Give the component a name that helps you link it with the rules file. Suppose you have created a rules file that contains rules to target content for New England snowboarders called NewEnglandSnowboarders.rules in the directory <ATG10dir>/home/targeting/rulesets. Set up the Rule Set Service for this rules file by creating a component you could name NewEnglandSnowboardersRuleSet. A Rule Set Service component has the following properties:

rulesFilePath

If your Rule Set Service refers to a rules file, set this property to the file path of the rules file. This path can be an absolute path or a relative path starting from your <ATG10dir>/home directory.

ruleSet

If you want to include your rules as a property of the component, set this property to the value of your targeting rules. The syntax for rules defined in this property is the same as for rules defined in a rules file, except that you must indicate new lines with the backslash (\) character. (The backslash must be the last character in the line; additional spaces after the backslash will prevent the rule set from working properly.) For example:

```
ruleSet=<ruleset> \
           <accepts> \
            <rule op=and> \
             <rule op=eq name="Rubber sector"> \
               <valueof target="industry sector"> \
               <valueof constant="rubber"> \
             </rule> \
             <rule op=includes name="Rubber analysts"> \
               <valueof bean="Profile.sectorsCovered"> \
               <valueof constant="rubber"> \
             </rule> \
           </accepts> \
           <sortby> \
             <sortbyvalue value="company" dir=ascending> \
           </sortby> \
           </ruleset>
```

Do not use both the rulesFilePath property and the ruleSet property in the same Rule Set Service. If you want to combine rules set in your properties file with the ruleSet property with rules defined in a rules file, use the includes tag or the src attribute to incorporate the rules defined in the rules file into the rules defined in the ruleSet property.

When you create rules using the ATG Business Control Center or the ACC, the system creates a Rule Set Service using the rulesets property to define the rule, rather than pointing to a separate rules file with the rulesFilePath property. Rule Set Service components created by the UIs are instances of atg.targeting.DynamicContentTargeter and are saved in the /atg/registry/RepositoryTargeters folder.

useTranslatedPath

This property is set to false by default. If you use the rulesFilePath property to specify a path to a rules file, and the path includes the system property atg.dynamo.root, as shown in the example below, you must set the useTranslatedPath property to true so that the path can be parsed correctly:

```
rulesFilePath={atg.dynamo.root}/...
```

For more information on using the atg.dynamo.root system property to access files from application module code, refer to the *ATG Installation and Configuration Guide*.

updatesEnabled

This property is set to true by default. This setting instructs the system to check the rules file for changes, at an interval set by the rulesFileCheckSeconds property.

rulesFileCheckSeconds

If the updatesEnabled property is set to true (as it is by default), this property sets the time interval after which to check whether the rules file has changed; if 0, the check will be performed on each request.

Example

Here is an example of a RuleSetService properties file for our NewEnglandSnowboarders.rules rules file.

```
$class=atg.targeting.RuleSetService

# Path of the rules file
rulesFilePath=targeting/rulesets/NewEnglandSnowboarders.rules

# Should we check whether the rules file has changed?
updatesEnabled=true

# Time interval after which to check whether the rules file has
# changed; if 0, the check will be performed on each request.
rulesFileCheckSeconds=0
```

Setting Up a RuleBasedRepositoryTargeter Service

Once you've set up a Rule Set Service for your rules, you implement targeting with the rules by setting up a RuleBasedRepositoryTargeter service that uses the Rule Set Service. Do this by creating a targeter component as an instance of atg.targeting.RuleBasedRepositoryTargeter. Continuing with the example in Setting Up a RuleSetService, you would create a component with a name like NewEnglandSnowboardersTargeter.

A RuleBasedRepositoryTargeter component can have the following properties:

repository

The repository that holds content objects to match against the profiles in the source.

ruleSetService

The Nucleus path name of the Rule Set Service used by the targeter component.

Example

The following example shows how a NewEnglandSnowboardersTargeter.properties file could look.

```
$class=atg.targeting.RuleBasedRepositoryTargeter
repository=/atg/adapter/html/TargetedContent
ruleSetService=/atg/targeting/NewEnglandSnowboardersRuleSet
```

Note that when you create a targeter component using the ATG Business Control Center or the ACC, the component you create is an instance of atg.targeting.DynamicContentTargeter with a Nucleus address of /atg/registry/RepositoryTargeters.

Note also that multsite features are not supported by the ACC.

Setting Up a RuleBasedRepositoryItemGroup Service

You can create groups of repository items, using the atg.targeting.RuleBasedRepositoryItemGroup class. A repository item group has the same properties as a RuleBasedRepositoryTargeter: repository and ruleSetService.

Create your RuleBasedRepositoryItemGroup component in /atg/registry/RepositoryGroups, so that the system's group registry can find it and make it available to the ATG Business Control Center or the ACC. Note that when you create a content group component through the UIs, the component you create is an instance of atg.targeting.DynamicContentGroup with a Nucleus address of /atg/registry/RepositoryGroups.

Setting Up a TargetingSourceMap Service

The previous chapter, Creating Rules for Targeting Content, described how to write a rule that compares properties of a user profile to a constant, or to properties of other objects, as in the following rule:

```
<rule op=includes name="Rubber analysts">
  <valueof bean="/atg/userprofiling/Profile.sectorsCovered">
```

```
        <valueof constant="rubber">
      </rule>
```

Writing a rule in that style requires you to provide the full Nucleus path of the user profile or other source object. You can simplify your references to profiles or other source objects by using a TargetingSourceMap service. The TargetingSourceMap maps source names to their Nucleus paths.

The Personalization module uses, by default, a TargetingSourceMap service that maps user profiles to the name “Profile.” Each of the targeting servlet beans described in the next chapter uses this /atg/targeting/TargetingSourceMap by default:

```
$class=atg.targeting.TargetingSourceMap
sourceMap+=Profile=/atg/userprofiling/Profile
```

The sourceMap property is a comma-delimited list of names for source objects, with the Nucleus path of each. Now, your rule set can refer to the name property of a profile with <valueof bean="Profile.name"> instead of <valueof bean="/atg/userprofiling/Profile.name">.

The Personalization module also uses another TargetingSourceMap service to make available short names for the Request object and the CurrentDate object, at /atg/targeting/TargetingSourceMap:

```
$class=atg.targeting.TargetingSourceMap
sourceMap+=\
          Request=/OriginatingRequest,\
          Today=/atg/dynamo/service/CurrentDate
```

Using TargetingResults

After you’ve set up your Rule Set Service, targeter component, and optionally a TargetingSourceMap service, you can use the Personalization module’s targeting servlet beans to access the targeter service from a content page (.jhtml or .jsp file). Targeting servlet beans are described in the *Serving Targeted Content with ATG Servlet Beans* chapter of the *ATG Page Developer’s Guide*.

The atg.targeting package includes a class, atg.targeting.TargetingResults, that you can use to perform targeting operations outside the context of a targeting servlet bean on a content page. TargetingResults is useful in cases where you have a potentially large number of results. TargetingResults is a Nucleus service that can be configured with the specific Targeter and NameResolver, and used to produce the corresponding TargetingEnumeration objects.

A TargetingResults component uses the following properties:

Property	Value
`targeter`	A targeter that operates over the appropriate repository.
`sourceMap`	A TargetingSourceMap for resolving names of source objects used by targeting rules. See Setting up a TargetingSourceMap Service.
`bufferSize`	Number of items to target for in each iteration. If 0, only one targeting operation will be performed, with all the targeting results returned at once. Otherwise, targeting will be performed in chunks, with the specified number of elements returned each time.

The `targeter` property should point to a targeter service that operates over a repository. You can configure the targeter service through the Targeting and Segmentation interface in the ATG Business Control Center (see the *ATG Business Control Center User's Guide*), you can use the Targeting > Content Targeters window of the ACC (see the *ATG Personalization Guide for Business Users*), or you can configure the targeter by hand (see Creating Rules for Targeting Content in this manual).

The `bufferSize` property lets you limit the maximum number of items that are returned at a time. The read-only `results` property is used to extract the results as an Enumeration. When the Enumeration elements are accessed, targeting is performed as needed to extract the next result array of size `bufferSize`. If `bufferSize` is 0, the entire targeting result array is obtained the first time an Enumeration element is asked for. Thus, if the number of items in the targeting results is potentially large, this allows you to split one expensive targeting operation into many inexpensive ones. This is especially useful if, for example, you are only interested in the first 10 elements in the result set.

The `TargetingResults.getResults()` method returns an Enumeration that is an instance of `atg.targeting.TargetingEnumeration`. `TargetingEnumeration` is simply an implementation of `Enumeration` that is created with, at a minimum, a `Targeter` and a `NameResolver`, and allows you to access targeting results with `Enumeration` methods.

Defining Profile and Content Groups

This section describes how to create profile and content groups by hand. For information about creating profile and content groups through the ACC interface, see the *ATG Personalization Guide for Business Users*. For information about creating content groups through the ATG Business Control Center, refer to the *ATG Business Control Center User's Guide*.

Notes:

- In ATG installations that include ATG Content Administration and the ATG Business Control Center, user segments replace profile groups as a way of grouping site visitors. User segments contain the same basic functionality as profile groups but provide additional features, including a more flexible rule editor. User segments are intended to supersede profile groups and are the recommended tool for grouping site visitors in installations where they are available. For detailed information on user segments, refer to the *ATG Business Control Center User's Guide*.

- You cannot create a profile or content group, either by hand or through the UIs, that compares a target or a repository property to a bean property. While you can use bean properties in rule sets in targeters, you cannot do the same in profile or content groups. This is because profile and content groups may be evaluated without an accompanying request or session. For example, when you are sending targeted e-mail to a collection of people, there is no way for the ATG system to evaluate a reference to what could be a request or a session-scoped bean. The properties to which you compare a repository or a target property must be other target properties or constants.

Profile Groups

You can use the ACC's Targeting > Profile Groups window to define rules that group user profiles, based on the available profile attributes in your profile template. Each profile group you define becomes a boolean attribute of each user profile. For example, if you define a profile group named `Retirees`, members of that profile group will have a `retirees` attribute with a value of `true`. You can create rules that target group members like this:

```
<rule op=eq>
  <valueof target="retirees">
  <valueof constant="true">
</rule>
```

Profile groups created in the ACC's Targeting > Profile Groups window are of class `atg.targeting.DynamicContentGroup`. You can also define a profile group by hand, by creating a component whose class is `atg.targeting.RuleBasedRepositoryItemGroup`, and placing the component in the configuration tree at `/atg/registry/RepositoryGroups/UserProfiles`.

Profile groups that you create by hand appear in the Profile Groups window (along with any profile groups created in that window), but these groups are not identified by the profile group icon (👥), and cannot be edited in the Profile Groups window. (Only profile groups created in the Profile Groups window can be edited there; these profile groups store rules in a special format because of the way the ACC interface represents them.) However, profile groups created by hand are still available in the Targeting > Content Targeters window to use in creating targeters.

Profile groups created by hand are also available as non-assignable roles in the Dynamo User Directory. Since membership in a profile group is determined by a set of rules, you cannot arbitrarily assign users to a profile group in the ACC screens that manipulate the User Directory. You can query the User Directory to find users who are assigned to a specific profile group. For more information see Working with the Dynamo User Directory in this guide.

To create a profile group by hand:

1. Create a rule set service that defines the profiles to include in the profile group. See the Setting Up a RuleSetService section.
2. In the ACC, select Pages and Components > by Path.
3. Select `/atg/registry/RepositoryGroups/UserProfiles`.
4. From the File menu, select New Component.

5. In the Select Component Template dialog box, select Dynamo from the Modules list, and then select Generic Component from the Templates list. Click OK.
6. In the Generic Component Wizard, specify the class as `atg.targeting.RuleBasedRepositoryItemGroup`. Click Finish.
7. In the Input New Component Name dialog box, enter a name for the profile group, and click OK.

   The new component opens in the Component Editor.
8. In the Component Editor, set the Configured Value of the `repository` property to the Nucleus address of the profile repository; for example, `/atg/userprofiling/ProfileAdapterRepository`.
9. Set the Configured Value of the `ruleSetService` property to the Nucleus address of the rule set service you created in step 1; for example, `/atg/targeting/AggressiveInvestorsRuleSetService`.

Content Groups

You can use the ACC's Targeting > Content Groups window to define rules that group content items, based on the available item descriptors for your content repository. Installations that include ATG Content Administration can use the Targeting and Segmentation interface in the ATG Busines Control Center to perform this task. Note that only the ATG Business Control Center supports multisite content groups.

Content groups can help simplify your targeting rules. For example, you could create a content group with information for guest users, and then write a targeter for displaying the content in this group.

Content groups created in the UIs are of class `atg.targeting.DynamicContentGroup`. You can also define a content group by hand, by creating a component whose class is `atg.targeting.RuleBasedRepositoryItemGroup`, and placing the component in the configuration tree at `/atg/registry/RepositoryGroups/`<*repository name*>, where <*repository name*> is a folder that has the same name as the content repository that contains the content included in the content group.

Content groups that you create by hand appear in the UIs but cannot be edited there. However, they are still available in the UIs to use in creating targeters. Note, however, that multsite features are not supported by the ACC.

To create a content group by hand:

1. Create a rule set service that defines the content to include in the content group. See the Setting Up a RuleSetService section.
2. In the ACC, select Pages and Components > by Path.
3. Select `/atg/registry/RepositoryGroups/<`*`repositoryname`*`>`, where <*`repositoryname`*> is a folder that has the same name as the repository that contains the content included in the content group.
4. From the File menu, select New Component.
5. In the Select Component Template dialog box, select Dynamo from the Modules list, and then select Generic Component from the Templates list. Click OK.

6. In the Generic Component Wizard, specify the class as `atg.targeting.RuleBasedRepositoryItemGroup`. Click Finish.
7. In the Input New Component Name dialog box, enter a name for the content group, and click OK.

   The component is created, and opens in the Component Editor.
8. In the Component Editor, set the Configured Value of the `repository` property to the Nucleus address of the content repository; for example, `/atg/adapter/html/TargetedContent`.
9. Set the Configured Value of the `ruleSetService` property to the Nucleus address of the rule set service you created in step 1; for example, `/atg/targeting/StockFundsRuleSetService`.

Managing User Segments

As mentioned in the previous section, profile groups created in the ACC are of class `atg.targeting.DynamicContentGroup`. If your system includes ATG Content Administration, you can also create profile groups through the ATG Business Control Center, as described in the *ATG Business Control Center User's Guide*. Profile groups created in the ATG Business Control Center are called **user segments**, and are of class `atg.targeting.html.UserSegment`, which is subclass of a subclass of `atg.targeting.DynamicContentGroup`. Note that user segments created in the ATG Business Control Center cannot be edited in the ACC.

Some sites may define a large number of user segments. To help manage these segments, the ATG platform uses **segment lists**, which are specific subsets of the set of segments defined on the site. For example, ATG Customer Intelligence uses segment lists to determine which segments to use for creating reports.

Segment lists are predefined for specific features and applications, but you need to specify the actual segments to include in these lists through the ATG Business Control Center. For more information, see the *ATG Business Control Center Administration and Development Guide*.

Segment lists are stored in the Personalization Repository, located at `/atg/userprofiling/PersonalizationRepository`. Each segment list is represented by a single repository item of type `userSegmentList`, which has these properties:

- `displayName` – The name used in the ATG Business Control Center for the segment list; e.g., the `displayName` for the `CommerceReporting` list is Commerce Reports.
- `description` – The description of the segment list displayed in the ATG Business Control Center
- `groups` – A comma-separated list of the names of the segments; e.g., `YoungMen,Audiophiles,EarlyAdopters`

Segment lists are versioned assets, and the Personalization Repository (on a system that includes ATG Content Administration) is a versioned repository that is included in the `DPS.Versioned` module. Therefore, when you build the application for your ATG Content Administration environment, you must be sure to include this module. Note that when you build the application for your production environment, you should not include the `DPS.Versioned` module; the Personalization Repository on the production site is a non-versioned repository that can be queried at runtime.

How Segment Lists Are Used

A given feature that has a segment list associated with it (such as Affinity Selling) can use the segment list by invoking the `/atg/userprofiling/UserSegmentListManager` component, which determines which segments in the segment list the current user is a member of. This determination involves evaluating the rule set for each segment in the segment list. The feature can then restrict the data it considers to just those segments. For example, Affinity Selling can return the products with the highest affinity to a specified product, taking into account only the purchases of users in a segment the current user is a part of.

Conflict Resolution

This section covers features that resolve conflicts between content items delivered to pages by targeting servlet beans. Conflict resolution works as follows:

- Content repository administrators designate one or more properties to be used as **conflict topics**, **conflict tags** and optional **conflict priority**, possibly adding these properties to the repository configuration.
- A developer configures one or more **conflict filter** components in Nucleus to make use of the above conflict data.
- Page developers deploy the above filters by passing their names as optional parameters to targeting servlet beans in their pages (e.g., `TargetingForEach`). As the page is rendered, each servlet bean employs its corresponding conflict filter to remove items from the result set returned by a targeter or slot.
- Content creators supply the above conflict tags through the standard repository administration user interface.

Programming Interface

Conflict filter components implement the `atg.service.filter.ItemFilter` interface. The targeting servlet beans call this interface's `filterItems` method, passing in the targeter's result set plus the same name resolver that was passed to the targeter:

```
public Object[] filterItems(Object[] pItems, NameResolver pResolver)
```

In addition, conflict filter components implement the `atg.service.filter.ItemHistory` interface. The targeting servlet beans call this interface's `recordItems` method, passing in the result of the `filterItems` method to keep the history of all displayed items for further conflict resolutions:

```
public void recordItems(Object[] pItems);
```

Architecture and Implementation

A simple implementation of the ItemFilter and ItemHistory interfaces is provided in the atg.service.filter.TopicHistoryConflictFilter class, which maintains a historical record of all items that have been allowed through it so far. On invocation of its filterItems method, it filters out any items that conflict either with other items in the same result set, or with any items that have been allowed through the filter on previous invocations.

Typically this object will be a session or request-scoped Nucleus component. If session-scoped, it provides conflict resolution over the course of the session; if request-scoped, it provides conflict resolution within the serving of a single page.

The algorithm used to resolve conflicts is as follows: each item is analyzed by looking at a set of specific DynamicBeans properties, whose names are supplied by the conflict filter configuration. The analysis determines these attributes for each item:

conflict topic
The name of a “topic” to which the item applies. For example, a soft drink promotion might belong to the topic softDrinks, while a promotion for automobile parts might belong to the topic autoParts.

conflict tag
A string that is used to resolve conflicts within a given topic. For example, a promotion for Cogswell Cogs auto parts might have a conflict tag of Cogswell, while a Spacely Sprockets auto parts promotion might have a conflict tag of Spacely.

conflict priority
An integer that determines priority within a given topic.

Two items are considered to be in conflict if their conflict topics are the same, but their conflict tags differ. Thus, Cogswell Cogs and Spacely Sprockets promotions in the above examples would be considered in conflict with each other. On the other hand, a soft drink promotion would not conflict with an auto parts promotion, despite their conflict tags being different, because their conflict topics are not the same.

If two conflicting items belong to a single targeting result set being filtered, and conflict priorities can be determined, then the item with the greater priority wins. If conflict priorities cannot be determined, then a random choice is made.

If two items conflict and one item has already been recorded in the filter’s history as having passed the filter, then the other item is rejected.

The configurable properties of a TopicHistoryConflictFilter are as follows:

itemTopicProperty
The name of an item DynamicBeans property which provides the conflict topic. If an item lacks this property, it will always pass the filter.

itemTagProperty
The name of an item DynamicBeans property which provides the conflict tag by default. If an item lacks this property, it will always pass the filter.

itemPriorityProperty (optional)
The name of an item DynamicBeans property which provides the conflict priority. If

omitted, then conflict priorities are ignored and conflicts within a result set are always resolved randomly.

Different conflict filters may be deployed in different servlet beans on the same site, if desired, to provide conflict resolution services that are customized for particular purposes.

The Personalization module includes a `/atg/targeting/ConflictFilter` component that is a session-scoped conflict filter of type `TopicHistoryConflictFilter` pre-configured to use the item property names `conflictTopic` and `conflictTag`.

Invoking in Servlet Beans

All the targeted servlet beans accept the optional parameter `filter`. If supplied, this filter is applied to result sets returned from the targeter specified by the servlet bean’s `targeter` parameter.

Using Slots to Deliver Content

Slots are Nucleus components that you can use to display and manage dynamic content on your Web site. You use targeting servlet beans to include slots on your site pages, and you use scenarios to fill them with content.

For more information, see Using Slots.

10 Using Targeted E-mail

You can use the Targeted E-mail services included with the Personalization module to compose and deliver e-mail using the same profile groups and targeting rules you use to deliver content on your Web sites. For example, you can use targeted e-mail to:

- Send a confirmation message to a new user who registers at a site.
- Notify frequent customers of special sales.
- Notify all users who have not logged in to a site in several months that their accounts will be closed soon.
- Send out a mass mailing with each message tailored to its recipient.

Before you begin this chapter, you may want to review the discussion of ATG's e-mail services in the *ATG Programming Guide*.

Note: While this chapter discusses accessing targeted e-mail programmatically, the Scenarios module includes extensive capabilities for working with targeted e-mail through the ACC interface. See Using Scenario Actions for more information.

This chapter includes the following sections:

Creating Targeted E-mail

You create targeted e-mail using the `atg.userprofiling.email.TemplateEmailInfoImpl` class. This class draws the message body of an e-mail from a page template, invokes a `MessageContentProcessor` component to process the content, and then passes the resulting JavaMail object to the `TemplateEmailSender` component, which sends the message. The properties of a `TemplateEmailInfoImpl` object store values that are specific to an individual e-mail campaign, so you should create a separate instance of this class for each campaign.

You can use targeted e-mail to produce e-mail messages that are personalized for different profiles. The e-mail content is based on a JSP or JHTML template, specified with the `templateURL` property of a `TemplateEmailInfoImpl` object. Using the dynamic elements of the Page template, you can customize the e-mail for each of your users according to their profile attributes. Here is a JSP example :

```
<html>
<dsp:importbean bean="/atg/userprofiling/Profile"/>

<p>Dear <dsp:valueof bean="Profile.firstName">Customer</dsp:valueof>,
<p>We're writing to tell you about some exciting new offers ...

</html>
```

The template yields different message text depending on the attributes of the recipient's profile. You can use ATG's targeting servlet beans to personalize the content of the message as well. If the template page contains a targeting servlet bean, the servlet bean behaves just as it would in any other page, rendering content targeted to the current profile.

The `TemplateEmailInfoImpl` object contains the standard e-mail message attributes as properties. When it invokes the `createMessage` method, the specified e-mail attributes create and fill in a `javax.mail.Message` object, which can then be sent to the desired recipient using the `TemplateEmailSender` service, described in the next section.

The following table describes key properties of the `TemplateEmailInfoImpl` class:

Property	**Function**
`altTemplateURL`	Specifies the URL of a template containing a text version of an HTML message. See Sending Message Content as Both Text and HTML.
`batched`	Enables distributed mailing features for this mailing. See Distributing a Mailing Across Multiple Servers.
`templateURL`	The URL of the template page. Example: `templateURL=/en/email/newfund.jsp` If the template contains links to other URLs on your site, you must specify them as absolute URLs in order for the email recipients to be able to access the linked pages. Use the full `http://server:port/...` form of the URL, for example `<img src="http://myserver:80/images/logo.gif>` rather than `<img src="images/logo.gif>` and `<a href="http://myserver:80/help/index.html>help</a>` rather than `<a href="help/index.html>help</a>`.
`mailingName`	A name used to identify an e-mail campaign Default: `Tuesday meeting`

messageFrom	“From” field of the e-mail message Default: management@example.com
messageTo	A single e-mail address that overrides the emailAddress property of the target profile Default: test@example.com
messageReplyTo	“Reply to” field of the e-mail message Default: hrgroup@example.com
messageCc	“Cc” field of the e-mail message Default: bill@example.com
messageBcc	“Bcc” field of the e-mail message Default: some-one@example.com
messageSubject	“Subject” field of the e-mail message Default: Mandatory Meeting, Tuesday at 3:00
messageAttachments	Any files to attach to the e-mail message Default: E:/mailattach/agenda.doc
contentProcessor	MessageContentProcessor responsible for processing the message content Default: /atg/userprofiling/email/HtmlContentProcessor
fillFromTemplate	Overrides the properties set in a JHTML or JSP file and fills the e-mail with values from an e-mail template Default: true
siteId	(Multisite environments) The site ID to use for the message. Retrieved from the Site Context Manager.

Note that TemplateEmailInfoImpl is a subclass of the abstract class TemplateEmailInfo. If you want to create your own template e-mail implementation, you can create a subclass of TemplateEmailInfo that implements all of its methods, and use it as you would use TemplateEmailInfoImpl.

Creating a Targeted E-mail Template

The page template for your targeted e-mail is specified by the templateURL property of the TemplateEmailInfoImpl object. You create this template like any other page. Because the template page is passed to the standard ATG servlet pipeline, it can include ATG servlet beans and additional JHTML or JSP tags.

Here is a sample JSP targeted e-mail template. It displays a few profile attributes and contains a targeting servlet bean. Assuming the targeting rules used in the `TargetingForEach` servlet bean's targeter service depend on the attributes of the `Profile` component, the targeting results displayed in the e-mail message will be different for each profile.

```
<dsp:importbean bean="/atg/userprofiling/Profile"/>

<p>Dear <dsp:valueof bean="Profile.firstName"/>
<dsp:valueof bean="Profile.lastName"/>,

<p>Thank you for your order! It has been shipped today to:

<blockquote><pre>
<dsp:valueof bean="Profile.address"/><br>
<dsp:valueof bean="Profile.city"/>, <dsp:valueof bean="Profile.State"/>
<dsp:valueof bean="Profile.zipCode"/>
</pre></blockquote>

<p>Since your last order, we've introduced some great new products. If you enjoy
your new <dsp:valueof bean="Profile.lastProductShipped"/>, you may also be
interested in ordering some of these great widgets:<p>

<dsp:droplet name="/atg/targeting/TargetingForEach">
  <dsp:param bean="/targeters/WidgetTargeter" name="targeter"/>
  <dsp:oparam name="output">
    <dsp:valueof param="element.name"/><br>
  </dsp:oparam>
</dsp:droplet>

<p>Thank you for shopping with us.

<p>Sincerely,
The Customer Service Team
help@example.com
http://www.example.com
```

Note that if your template page contains links to other URLs on your site, you must specify them as absolute URLs; otherwise the e-mail recipients will not be able to access the linked pages. Use the full `http://`*`server`*`:`*`port`*`/...` form of the URL. For example, in JSP code:

```
<dsp:img src="http://myserver:80/images/logo.gif">
<dsp:a href="http://myserver:80/help/index.html">help</dsp:a>
```

The following example will **not** work, because it uses relative URLs:

```
<dsp:img src="images/logo.gif">
<dsp:a href="help/index.html">help</dsp:a>
```

Specifying E-mail Fields in the Template

By default, the Personalization module takes the values for various fields of an e-mail message from properties of the `TemplateEmailInfoImpl` object. This behavior means that all messages using the same object will have identical properties. For example, suppose the value of the `messageSubject` property of the `TemplateInfoImpl` component is “Hello”. All e-mail messages created by this component will contain the subject line “Hello”.

You can override the values of these properties in a specific e-mail template by doing the following:

1. In the template, include parameters with the same names as the corresponding `TemplateEmailInfoImpl` properties.
2. Set the `fillFromTemplate` value of the `TemplateEmailInfoImpl` object to `true`.

For example, you can set a different subject line for a specific mailing by including a statement as follows in the template for that mailing:

JSP example:

```
<dsp:setvalue value="Your order has shipped" param="messageSubject"/>
```

JHTML example:

```
<setvalue param="messageSubject" value="Your order has shipped">
```

Because these parameters are evaluated for each message individually, they can include dynamic content. For example, you could set the message subject as follows.

JSP example:

```
<dsp:setvalue value='<%="Order " + request.getParameter("order.id")+
" has shipped"%>' param="messageSubject"/>
```

JHTML example:

```
<setvalue param="messageSubject"
  value="Order 'request.getParameter("order.id")' has shipped">
```

There are seven parameters that you can use in this way to override the corresponding `TemplateEmailInfoImpl` properties:

- `mailingName`
- `messageFrom`
- `messageTo`
- `messageReplyTo`
- `messageSubject`
- `messageCc`
- `messageBcc`

You can use any combination of these parameters in an e-mail template; if any of these parameters is not set, the system uses the corresponding property value instead.

For example, you could globally set the `messageCc` property to Cc yourself, but you could override this property if you wanted to Cc or Bcc other people to make them aware of a particular e-mail campaign. The `messageCc` and `messageBcc` parameters are defined as a string of e-mail addresses separated by commas. For example, you could add the following lines to a template file.

JSP example:

```
<dsp:setvalue value=lewis@example.com,everyone@example.com
 param="messageCc"/>
<dsp:setvalue value=management@example.com,bob@example.com
 param="messageBcc"/>
```

JHTML example:

```
<setvalue param="messageCc" value=lewis@example.com,everyone@example.com>
<setvalue param="messageBcc" value=management@example.com,bob@example.com>
```

Note: Property values you set through an e-mail template are not persisted to the database. Only the values that are set in the `TemplateEmailInfoImpl` object are persisted.

Specifying a MessageContentProcessor

The `TemplateEmailInfoImpl`'s `contentProcessor` property points to a `MessageContentProcessor` object, which is responsible for actually setting the e-mail message content given the rendered message body. Implementing this functionality in a separate class allows you to implement different schemes for processing content.

`MessageContentProcessor` is an abstract class. The Personalization module includes two concrete subclasses of `MessageContentProcessor`: `SimpleContentProcessor` and `HtmlContentProcessor`. You can also implement your own content-processing scheme by creating a subclass of `MessageContentProcessor` that implements all of its methods.

SimpleContentProcessor

`SimpleContentProcessor` doesn't do any processing on the passed-in message content but simply uses the content as is to set the `javax.mail.Message` content. The MIME type of the content in the `Message` is specified by the service's `contentType` property. For instance, if `contentType` is "text/html," the rendered page is sent as HTML; if the `contentType` is "text/plain," it will be sent as simple text (no word wrapping).

HtmlContentProcessor

`HtmlContentProcessor` is a more sophisticated implementation of `MessageContentProcessor`. The `HtmlContentProcessor` does the following:

1. Takes in content of type "text/html"
2. Optionally, converts it into content of type "text/plain"

3. Performs word wrapping on the plain text version of the content
4. Determines whether to send the `Message` content as "text/html", "text/plain", or "multipart/alternative" with "text/plain" and "text/html" parts

You can configure the `HtmlContentProcessor` with the following properties:

Property	Function	Default
`sendAsHtml`	If true, send message as HTML; if `sendAsText` is also true, send as multipart/alternative with text/plain and text/html parts.	true
`sendAsText`	If true, send message as plain text; if `sendAsHtml` is also true, send as multipart/alternative with text/plain and text/html parts.	true
`indentWidth`	Indentation width used in text/plain version of message.	5
`lineWidth`	Line width used in text/plain version of message.	70
`hrChar`	Character to use to create horizontal rules in text/plain version of message.	-
`liChar`	Character to use to introduce list items in text/plain version of message.	-

HtmlToTextConverter

If the `sendAsText` property in `HtmlContentProcessor` is true for a given e-mail message, the Personalization module uses an instance of the class `atg.userprofiling.email.HtmlToTextConverter` to convert the message to plain text. Note, however, that the converter supports the conversion of content included in the following HTML tags only:

```
title
h1
h2
h3
h4
h5
h6
p
br
hr
blockquote
pre
ul
ol
li
```

See also Sending Message Content as Both Text and HTML.

Sending Message Content as Both Text and HTML

As described in the previous section, the HtmlContentProcesser and HtmlTextConverter allow you to send a message as both text and HTML. However, formatting results for text messages converted from HTML may be unpredictable. For situations where you need to guarantee a properly formatted message in both styles, you can configure the altTemplateURL property in the TemplateEmailInfo object. As the value for this property, specify a URL that points to an alternative text template for the HTML message. The text representation is generated in the same session as the HTML representation.

In addition, in the JSP or JHTML page that renders the alternative text template, set the MIME type of the response to text/plain. For more information, see the *ATG Page Developer's Guide*.

Note that this feature is available only if you are creating and sending mailings programmatically; it is not available for scenario-generated mailings.

Creating the Recipient List

After you have created the TemplateEmailInfoImpl object, you need to create a targeted list of people to send the mail to. This list of recipient profiles can be a Collection, an Enumeration, an array of profile objects, or a group of String-based e-mail addresses. You can also create a mixed list of profile objects and String e-mail addresses. Typically, however, you would obtain the recipient profile list by targeting the profile repository for profiles that satisfy some particular criteria. Two ways to get a recipient profile list are:

- Using the TargetingResults Class
- Targeting with Profile Groups

After you have defined your recipient list, call the TemplateEmailSender's sendEmailMessage method, passing in the TemplateEmailInfoImpl object corresponding to the e-mail and a Collection, Enumeration, or array representing the recipient list.

Using the TargetingResults Class

The atg.targeting package includes a class, atg.targeting.TargetingResults, that you can use to generate targeted recipient lists. TargetingResults is a Nucleus service that can be configured with the specific Targeter and NameResolver, and used to produce the corresponding TargetingEnumeration objects.

Set the targeter property of the TargetingResults component to point to a targeter that operates over a profile repository (such as /atg/userprofiling/ProfileAdapterRepository). You can configure the targeter service in the ATG Business Control Center (see the *ATG Business Control Center User's Guide*), in the ACC (see the *Matching Content with Your Target Audience* chapter in the *ATG Personalization Guide for Business Users*), or by hand (see the Creating Rules for Targeting Content and Setting Up Targeting Services chapters in this manual).

Once you have defined the TargetingResults object, you can obtain a recipient list by calling TargetingResults.getResults().

Targeting with Profile Groups

Another way to obtain the recipient list is by asking a profile group for its members. You can create profile groups using the ACC (see the *Creating Profile Groups* chapter in the *ATG Personalization Guide for Business Users*) or programmatically (see the Setting Up Targeting Services chapter in this manual). Assuming you have a pointer to a `profileGroup` (of type `RepositoryItemGroup`), you can create a list of recipients that includes all the members of the profile group:

```
Object[] recipients = profileGroup.getGroupMembers();
```

Sending Targeted E-mail

Once you have created your messages and identified the list of recipients, you use a `TemplateEmailSender` (`atg.userprofiling.email.TemplateEmailSender`) to send out the mailing. There is also a `TemplateEmailListener` component that listens for events such as success or failure in sending e-mails. See Handling E-mail Problems.

`TemplateEmailSender` is the service responsible for sending targeted e-mail. Its `sendEmailMessage` method takes a `TemplateEmailInfo` object together with a list (`Collection`, `Enumeration`, or array) of recipient profiles.

Note that, because `TemplateEmailSender`'s methods take any `TemplateEmailInfo` object, you can use either the `TemplateEmailInfoImpl` subclass or a custom subclass you have created.

The `TemplateEmailSender` does the following:

- It renders the page given by the `TemplateEmailInfo`'s `templateURL` for each of the profiles or e-mail addresses provided.
- It invokes the `TemplateEmailInfo`'s `createMessage` method to obtain the corresponding JavaMail message.
- It sends the resulting message to each corresponding user (by way of the `EmailMessageSender` component).

The actual sending of the messages is handled by an `atg.service.email.EmailMessageSender` component (for example, `/atg/dynamo/service/SMTPEmail`), which is specified by `TemplateEmailSender`'s `emailMessageSender` property.

The Page is rendered once for each user by creating a `DynamoHttpServletRequest` and sending it down the ATG servlet pipeline. (In the default configuration; it is also possible to implement other content rendering schemes by specifying different values for the service's `templateRendererServlet` property.) That is, the template page is rendered simply as if it were a normal ATG page requested through a browser. See the *Request Handling and the Servlet Pipeline* chapter in the *ATG Programming Guide* for more information about how the servlet pipeline handles requests.

The default instance of `DynamoHttpServletRequest` associates the recipient profile object with the current session. In other words, it binds the profile object to the session-scoped `/atg/userprofiling/Profile`, and allows the page to refer to the profile's attributes, as well as to include targeting servlet beans that display content targeted to the current profile. Note that the profile must have attributes for the user's e-mail address and locale (although the locale property value can be null). In addition, the profile must have an attribute named `receiveEmail`, and the value of this attribute must be yes or null.

The following table describes key properties of the `TemplateEmailSender` class:

Property	**Function**
`batchEmailPeriodicService`	The name of the component that performs a number of periodic tasks as part of a performing a distributed mailing. For more information, see Distributing a Mailing Across Multiple Servers.
`batchIfPossible`	Property that enables distributed mailing automatically as long as a given mailing does not include transient profiles. For more information, see Distributing a Mailing Across Multiple Servers.
`emailAddressPropertyName`	Name of the e-mail address property in the profile. Default: `/atg/userprofiling/PropertyManager .emailAddressPropertyName`
`emailEncodingMap`	List of mappings between character sets used in e-mail templates and in messages (see *Setting the E-mail Encoding* in the *Internationalizing a Dynamo Web Site* chapter of the *ATG Programming Guide*.) Default: `SJIS=iso-2022-jp,EUC=iso-2022-jp`
`emailMessageSender`	`EmailMessageSender` component responsible for sending the e-mail. Default: `/atg/dynamo/service/SMTPEmail`
`emailStatusInvalidOptionValue`	The String value for the `emailStatus` profile property, used to flag the status as invalid. Default: `invalid`.
`emailStatusValidOptionValue`	The String value for the `emailStatus` profile property, used to flag the status as valid. Default: `valid`.
`enabledAsTemplateMailServer`	Determines whether this server should participate in a distributed mailing. For more information, see Distributing a Mailing Across Multiple Servers.
`encodingTyper`	`PageEncodingTyper` object that determines the encoding used in the e-mail. Default: `/atg/dynamo/servlet/pagecompile/EncodingTyper`
`localePropertyName`	Name of the locale property in the profile. Default: `locale`

`profilePath`	Session-scoped `Profile` object. Default: `/atg/userprofiling/Profile`
`requestSetupServlet`	`HeadPipelineServlet` responsible for setting up the request. Default: `/atg/dynamo/servlet/pipeline/DynamoHandler`
`templateEmailListeners`	List of listeners notified of any targeted e-mail events. Default: There is no existing component for this class
`templateEmailPersister`	`TemplateEmailPersisterImpl` component responsible for recording information about a mailing so that the mailing can be restarted if necessary. Default: `/atg/userprofiling/email/TemplateEmailPersister`
`templateRendererServlet`	Servlet responsible for rendering the template page. Default: `/atg/dynamo/servlet/pipeline/DynamoHandler`

Sending E-mail to Users Without Profiles

In some situations, you may want to send e-mail to a user or a group of users who have not yet registered and who do not have profiles. For example, you might want to send a gift certificate to a group of e-mail addresses promising a promotion if the recipients register at your Web site. To send e-mail to a user without a profile, use the `TemplateEmailInfoImpl.messageTo` property. You can set the `messageTo` property globally, or in a JSP or JHTML file. If you set it globally, `messageTo` overrides the `emailAddress` property specified in the e-mail template. To set `messageTo` in a JSP file, add the following lines to the JSP template:

```
<dsp:setvalue paramvalue="message.profile.giftEmailAddress"
 param="messageTo"/>
<setvalue param="fillfromTemplate" value="true">
```

Here is the same example in JHTML:

```
<setvalue param="messageTo"
 value="param:message.profile.giftEmailAddress">
<setvalue param="fillfromTemplate" value="true">
```

This setting is local to a specific template and does not affect other mailings. The `fillfromTemplate` parameter forces the sender to override any property set in the template file.

Using the TemplateEmailSender Class

You can also send e-mail to e-mail addresses that do not have matching profiles by specifying the e-mail addresses as an array of Strings. In the same way that you can send e-mail to a profile group, you can programmatically specify a list of e-mail addresses:

```
Object []recipients = {"bill@example.com", "sam@example.com"};
TemplateEmailSender.sendEmailMessage(TemplateEmailInfo, recipients);
```

In this example, the `TemplateEmailSender` component checks each element in the `recipients` [] and if the element is of type String , the `TemplateEmailSender` sends e-mail to the e-mail address.

You can also mix and match Profile objects with String based e-mail addresses when calling `TemplateEmailSender.sendEmailMessage`:

```
Object[] profileRecipients = profileGroup.getGroupMembers();
Object[] emailRecipients = {"bill@example.com", "sam@example.com"};
Object[] recipients = addArrays(profileRecipients, emailRecipients);
TemplateEmailSender.sendEmailMessage(TemplateEmailInfo, recipients);
```

For more information on working with users who don't have profiles, see Tracking Guest Users in the Working with User Profiles chapter.

Viewing, Canceling or Resuming a Mailing

The targeted e-mail system includes ATG Dynamo Server Admin features that you can use to examine or cancel mailings. In addition, you can use the Admin to view individual batches of a distributed mailing or resume a distributed mailing that you previously canceled.

1. Start the ATG Dynamo Server Admin as described in the *ATG Installation and Configuration Guide*.
2. Click the Component Browser link and navigate to the `/atg/userprofiling/email/TemplateEmailSender/` page.

The Mailings by Status display shows information similar to the following:

Mailing Type	# Local Mailings	# Distributed Mailings	Action
All	3	18	Show/Remove Local/Remove Distributed
Failed	0	0	Show/Remove Local/Remove Distributed
In Progress	1	4	Show/Remove Local/Remove Distributed
Canceled	0	0	Show/Remove Local/Remove Distributed
Pending	0	5	Show/Remove Local/Remove Distributed
Completed	2	9	Show/Remove Local/Remove Distributed

In all cases, the numbers are the instances of each mailing type that have occurred since the `TemplateEmailSender` component was started.

Notes:

- Mailings you send through scenarios are automatically batched by the Scenario Manager, and each batch appears as a separate mailing.
- Only the mailings sent by this instance of the `TemplateEmailSender` appear in the list. In a multiple server environment, therefore, many more mailings may exist than are shown in this display.

Use the **Show** link to display a list of individual instances of a mailing type. Use the Remove link to delete all instances of that mailing type from the database.

After you click Show for a particular mailing type, you can choose to cancel or pause an individual mailing by clicking the Cancel link for that mailing. For distributed mailings, a Resume link then appears. You can use Resume to start the mailing from the point at which it was stopped (note that the mailing is not restarted from the beginning). When you select Resume, the batches are returned to the pool so that they can be reclaimed by the distributed mail servers (see Distributing a Mailing across Multiple Servers for more information).

Important: Be aware that it may take a few minutes for canceling a mailing to take effect. It is not recommended that you resume a mailing immediately after canceling it; this behavior can generate exceptions and cause deadlocks to occur. Wait until the cancellation is complete before resuming. (To check whether a cancellation is complete, look at the database and ensure that the Status entries for the mailing are set to 2.)

For distributed mailings, you can also view specific batches of the mailing by clicking View Batches after you use the Show link.

Use the **Remove Local** link to remove mailings from this sender.

Use the **Remove Distributed** link to remove distributed mailings from all senders in this cluster. Use this option with caution. As noted above, the display shows only the mailings that were generated by this instance of the `TemplateEmailSender`. If you click Remove Distributed, you remove all mailings from any instance that uses the same database as the sender on the server where you are viewing this list.

Avoiding E-Mail Fatigue

In some cases you may want to limit the frequency with which e-mails are sent to users. You can do so by setting the `daysContactFatigue` or `hoursContactFatigue` property in the `TemplateEmailSender` component. These properties specify an amount of time that must pass between mailings to any user. For example, if you set the `daysContactFatigue` property to 5, a user is excluded from a mailing if five days have not passed since a mailing was sent to him or her. The `TemplateEmailSender` uses the value of the `lastEmailed` property in the user profile to determine if the required amount of time has passed.

Note that the daysContactFatigue and hoursContactFatigue properties eliminate all e-mails. If you want to override the setting for an essential e-mail such as receipt or order confirmation, set the ignoreContactFatigue property in the TemplateEmailInfoImpl object to true for the given mailing.

Improving Performance for SQL JMS Mailings

When targeted e-mail senders initiate a mailing, the EmailManager fires an OutboundEmailMessage that includes the javax.mail Message object. If your sites use SQL JMS for mailings, and you send out a large number of e-mails, mailings can take a long time to complete because the javax.mail Message object needs to be written to the SQL database that is used for message persistence.

If your mailings do not require the javax.mail Message object, you can improve performance by setting the serializeOutboundEmailContent property in the /atg/userprofiling/DPSMessageSource component to false. This setting omits the javax.mail Message object from the message.

Handling E-mail Problems

The Personalization module includes several services that you can use in your application to handle situations when a mailing fails. Common situations the application may encounter include:

- Failed E-mail
- Bounced E-mail
- Stopped E-mail Campaigns

Failed E-mail

Each time TemplateEmailSender sends an e-mail message, it generates an event of class atg.userprofiling.email.TemplateEmailEvent, which indicates whether or not the message was successfully sent. For example, suppose you create a mailing with 10 recipients, and 1 message does not get sent because the recipient has an invalid e-mail address. TemplateEmailSender generates 10 template e-mail events, 9 indicating success and 1 indicating failure.

When a message fails, TemplateEmailSender looks at the e-mail address of the failed message, finds all profiles that use that address, and marks those profiles as being ineligible for receiving e-mail by setting the emailStatus property to invalid.

You can have your application perform specific actions depending on whether a message is successfully sent. For example, you could keep track of the messages that failed and delete the corresponding profiles. To facilitate this type of tracking, the TemplateEmailSender component has a templateEmailListeners property, which is a standard list of listeners notified of any template e-mail events. Any service that you add to this property is notified of each template e-mail event, which your application can then handle appropriately.

Note that if a message is sent successfully, this means only that the message was sent to a legal address (that is, an address that has a legal form, such as someone@some.example.com). A message sent successfully may still bounce if the address is not the actual address of any mailbox.

Bounced E-mail

To detect bounced e-mail, ATG provides the `atg.service.email.pop.POP3Service` class. This service implements a standard POP3 e-mail client. It periodically checks for incoming mail on a specified POP3 server, and it determines which messages are bounced-back messages. For each bounced message, it generates an event of class `atg.service.email.pop.MailBounceEvent`, which contains information about the bounced message.

The Personalization module includes an `atg.userprofiling.email.EmailManager` class which handles these events. An `EmailManager` component registers itself with the `POP3Service` as a listener. When the `POP3Service` generates a mail bounce event, the `EmailManager` looks at the e-mail address of the bounced message, finds all profiles that use that address, and marks those profiles as being ineligible for receiving e-mail by setting the `emailStatus` property to `invalid`.

Once a user's profile is marked as ineligible to receive e-mail, future mailings will skip this user. This cuts down on the number of bounced messages and provides a mechanism for flagging users with invalid e-mail addresses. For example, when a user logs into a site, you could check if the user's e-mail address is marked as invalid, and if it is, prompt the user to enter a new address. When the new address is submitted, your application should then set the `emailStatus` property to `valid` to include this user in future mailings.

Note also that the `InboundEmail` event in the Scenarios module is fired whenever the `POP3Service` receives an e-mail, and it contains various properties that you can use to create scenarios triggered by the arrival of bounced e-mails. For example, you could set up a scenario that notifies your customer service department when a bounced e-mail is received. For more information, refer to InboundEmail Event.

Enabling Bounced E-mail Detection

To enable bounced e-mail detection, you must configure the `/atg/dynamo/service/POP3Service` component to retrieve mail from a POP3 e-mail server. The following table summarizes the key properties of this component:

Property	**Function**
`username`	User name for the e-mail account. Default: `dynasend`
`password`	Password for the e-mail account. Default: `dynasend`
`host`	Name of the POP server, for example `pop.mydomain.com`.
`schedule`	Interval at which the component checks the POP server for new e-mail. Default: `every 15 seconds`
`scheduler`	The name of the service that the component uses to schedule the task of checking for new e-mail. Default: `/atg/dynamo/service/Scheduler`

`attachmentDir`	Directory for the POP client to store e-mail attachments. Default: `/tmp`
`removeBouncedEMail`	Indicates whether to delete bounced-back e-mail messages from the server after retrieving them. Default: `true`
`removeEmail`	Indicates whether to delete e-mail messages (other than bounced-back messages) from the server after retrieving them. Default: `false`

`emailExaminers`	An array of `EmailExaminer` components that you want the `POP3Service` to use when determining if an email is bounced. (See below.)
	Default:
	`/atg/dynamo/service/SendmailExaminer`
	`/atg/dynamo/service/EximExaminer`
	`/atg/dynamo/service/MSExchange6Examiner`
	`/atg/dynamo/service/MSExchange5Examiner`

In addition to the `POP3Service` component, you must also have an `EmailManager` component running. To ensure that both of these services start up when your ATG application starts, add them to the `initialServices` property of an `atg.nucleus.InitialService` component, such as the `/atg/dynamo/service/Initial` component.

Note that there must be only one instance of each of these services on a site. In other words, if an application has a single database but multiple servers running ATG products, only one of the ATG servers should run these services.

You must also configure and enable the specific `emailExaminer` components that you want the POP3Service to use to identify bounced e-mail messages. See the next section, Detecting Bounced E-mail Messages from Different MTAs.

Detecting Bounced E-mail Messages from Different MTAs

The format of each bounced message that the `POP3Service` receives can vary depending on the type of MTA (Mail Transfer Agent) that sent it. The `POP3Service` handles these different formats by means of one or more implementations of the `atg.service.email.examiner.EmailExaminer` interface. Several classes that implement this interface are supplied by default with the Personalization module:

- `atg.service.email.examiner.SendmailEmailExaminer`. See SendmailExaminer Component for more information.
- `atg.service.email.examiner.EximEmailExaminer`. See EximExaminer Component.

- atg.service.email.examiner.RegExEmailExaminer. You can use this class to detect bounced messages from other MTAs by configuring regular expressions. (The examiner looks for configurable string expressions inside the text of the bounced message to determine the various properties of the e-mail.) Two default implementations are provided for interpreting bounced Microsoft Exchange Server messages: /atg/dynamo/service/MSExchange5Examiner and /atg/dynamo/service/MSExchange6Examiner.

You specify the examiner component or components you want to use to check bounced e-mail messages in the emailExaminers property of the POP3Service component.

SendmailExaminer Component

The /atg/dynamo/service/SendmailExaminer component is an implementation of the atg.service.email.examiner.SendmailEmailExaminer class that you can use to detect bounced e-mail messages returned by sendmail MTA systems.

The component determines whether a message is bounced by looking for specific String expressions within the message text. For example, the following text would allow this component to determine that the message is bounced:

```
<<< 550 5.1.1 <noone@example.com>... User unknown
```

The SendmailExaminer component looks for the String <<< n, where n is any of the reply codes specified in the component's monitoredBounceReplyCodes property (550, in this example). If the String exists, the message is considered bounced.

In addition, it checks for any reply codes that are specified in the monitoredBounceReplyCodes property (see below).

An RFC 1893 enhanced status code appears after the bounce indicator String. In this example, the value is 5.1.1. This implementation assumes a space character after the reply code followed by a status code that is 5 characters long. The bounced e-mail address (here, noone@example.com) appears between the < and > characters after the bounced indicator. The error message appears after the first ... that is found after the bounce indicator. In this example, the error message is User unknown. Note that the text of the error message is a mail server-specific interpretation of the status code, and therefore the error message varies from one mail server to another.

The SendmailExaminer component is not enabled by default. It contains the following key properties:

Property	Description
MTAName	The MTA format to which this examiner corresponds. Default: sendmail

monitoredBounceReplyCodes	An array of Strings that represent RFC 821 reply codes from bounced e-mail messages. When an examiner checks an e-mail message to see if it is bounced, it looks for specific text and a reply code. If the monitoredBounceReplyCodes property is not set, a message is considered bounced if its text includes the characters <<< followed by any reply code that starts with 5. If the monitoredBounceReplyCodes property is set, the same characters are looked for, but the component performs an additional check to ensure that the reply code for the message matches one in this property. If it does not, the message is not considered bounced.

EximExaminer Component

The /atg/dynamo/service/EximExaminer component is an implementation of the atg.service.email.examiner.EximEmailExaminer class that you can use to detect bounced e-mail messages returned by exim MTA systems.

The component determines whether a message is bounced by looking for the following header in the message:

```
X-failed-recipients
```

In addition, it checks for any reply codes that are specified in the monitoredBounceReplyCodes property (see below).

This component is not enabled by default. It contains the following key properties:

Property	**Description**
MTAName	The MTA format to which this examiner corresponds. Default: exim

`monitoredBounceReplyCodes`	An array of Strings that represent RFC 821 reply codes from bounced e-mail messages. When an examiner checks an e-mail message to see if it is bounced, it looks for specific text and a reply code. If the `monitoredBounceReplyCodes` property is not set, a message is considered bounced if it includes a specific message header (X-failed-recipients). If the `monitoredBounceReplyCodes` property is set, the component performs an additional check to ensure that the reply code for the message matches one in this property. If it does not, the message is not considered bounced.

Identifying Soft-Bounced Messages

A "soft bounce" can be defined as a transient delivery failure, one where a message is bounced for reasons that are usually temporary, for example the user's mailbox is full or a server is too busy to handle the request. In such cases you may want to resend the mailing at a later time or otherwise identify the bounced e-mail recipient as only temporarily unavailable. The `softBounceReplyCodes` and `softBounceEnhancedStatusCodes` properties in the `EmailExaminer` implementations described above can be used to track soft bounce messages, which are identified by their RFC 821 and 1893 error codes.

Retrieving Tracking Data from a Bounced E-mail

You can send tracking data with a mailing for subsequent retrieval from a bounced message. To add tracking data to a mailing, add name/value pairs to the `trackingData` Map property in the `TemplateEmailInfoImpl` object. Alternatively, call the appropriate methods in the `atg.service.email.TrackableEmailSender` interface, which is implemented by the `atg.service.email.SMTPEmailSender` class (`/atg/dynamo/service/SMTPEmail` or `/atg/dynamo/service/EmailListenerQueue` components). As described earlier, when the `POP3Service` receives a bounced e-mail, it sends out bounce events to any configured listeners. The `trackingData` Map property in the bounce events contains any tracking data that could be retrieved from the bounced message.

Stopped E-mail Campaigns

If a mailing is stopped for any reason (for example, if the server sending out the mailing crashes), you can rerun the mailing when the server restarts. However, if any of the mail messages were sent successfully, some users might receive multiple copies of the mail. The Personalization module includes services you can use to handle this situation. These services record information about the users to whom the mailing was sent and whether or not the mailing completed successfully. When you restart the server, the system can check for any incomplete mailings and restart them from the point where the failure occurred. This behavior ensures that each user receives only one copy of a mailing.

There are two services in `atg.userprofiling.email` that handle restarting stopped e-mail campaigns, `TemplateEmailRestarter` and `TemplateEmailPersisterImpl`. When the `TemplateEmailRestarter` service starts up, it checks the database for incomplete mailings and restarts any that it finds. Typically, you want to invoke TemplateEmailRestarter when the server starts up, so you should add this service to the initialServices property of an atg.nucleus.InitialService component, for example the /atg/dynamo/service/Initial component.

There should be one instance of the TemplateEmailRestarter service running on each ATG server on a site. Each ATG server must have a unique DRP port number so the TemplateEmailRestarter service can determine which server created a given mailing. Configure the migrationList property to point to the list of UniqServerIds of the mailing that should be migrated to that server. For example:

```
migrationList=64.69.0.100:8851
```

This setting moves all mailings on host 64.69.0.100 at DRP port 8851 to the current server. To test that this migration works, you can create a set of mailings on one server, change the DRP port number, and then edit the TemplateEmailRestarter component to migrate from the old DRP port to the new one. You can verify the change by looking in the database.

The TemplateEmailPersisterImpl service does the actual work of recording information about a mailing as it is being sent out and checking the status of mailings when the ATG system starts up. The dps_mailing table has a property called num_profiles that is the number of profiles to which a particular mailing should be sent, and another property, num_sent, which is the number of e-mails sent in that mailing. Two more properties, num_errors and num_skipped, record respectively the number of messages that could not be sent because of errors that occurred during the mailing, and the number of recipients to whom message could not be sent, for example because the emailStatus profile property was set to invalid.

E-mails are sent in order, and each one is assigned a unique ID, which is the mailing_id property of the dps_mailing and dps_email_address tables. The TemplateEmailPersisterImpl service takes the list of recipients from the dps_user_mailing and the dps_email_address tables and begins resending e-mails from the number recorded as the sum of the num_sent, num_errors, and num_skipped properties.

This service is a property of both the TemplateEmailSender service (which uses it to record information about mailings) and TemplateEmailRestarter (which uses it to check the status of mailings). When you configure the TemplateEmailSender and TemplateEmailRestarter components, be sure to set their templateEmailPersister properties to the Nucleus pathname of a TemplateEmailPersisterImpl component, such as /atg/userprofiling/email/TemplateEmailPersister.

For information on setting up persistent targeted e-mail with a composite repository that uses an LDAP repository as its primary view, see Configuring Targeted E-mail for a Composite Repository.

For distributed mailings, information about the batches within a mailing is persisted to the dps_mail_batch table by the /atg/userprofiling/email/TemplateEmailBatchPersister component. Information about the servers used to perform a distributed mailing is persisted to the dps_mail_server table by the /atg/userprofiling/email/TemplateEmailBatchServerPersister component. See Distributing a Mailing Across Multiple Servers for more information.

Note that, if you use the Scenarios module, mailings sent through the Scenario window of the ACC automatically handle persistence. However, if you are creating mailings programmatically, you must use a TemplateEmailSender.sendEmailMessage() method that includes an argument for enabling persistence. For example, if you are sending targeted e-mail that contains a promotion, the e-mail is handled in a separate thread and you have to ensure that the slot that holds the promotion is persistent, otherwise the promotion will not be sent with the e-mail. A persistence argument could look like this:

```
sendEmailMessage(TemplateEmailInfo pEmailInfo,
     Enumeration pRecipients,
     boolean pRunInSeparateThread,
     boolean pPersist);
```

If either `RunInSeparateThread` or `Persist` evaluates to true, the e-mail has the potential to run in a separate thread, and the slots associated with the e-mail must be persistent. For more information on slots, see *Using Slots* in the *Creating Scenarios* chapter of the *ATG Personalization Guide for Business Users*.

Distributing a Mailing across Multiple Servers

The distributed e-mail feature, also called "batched e-mail," allows you to manage a large mailing efficiently by splitting it into multiple batches or ranges. Rendering threads running on participating machines (or on multiple processors on a single machine) take batches from the queue. They then generate and send the messages in essentially the same way as single threads do for a non-distributed mailing.

Note that you can use distributed e-mail only for mailings to registered users; you cannot use it for mailings whose recipient list includes any transient profiles. This limitation exists because distributed e-mail works with the template e-mail system's persistence mechanism, and transient profiles are not persisted to the database. (You can, however, use distributed e-mail for anonymous profiles if you use the `persistentAnonymousProfiles` feature. See Tracking Guest Users for more information.)

Configuring a Distributed E-mail Server

To configure a machine to participate as a distributed e-mail server:

1. Set up targeted e-mail as described earlier in this chapter.
2. Set the `enabledAsTemplateMailServer` property in the `atg/userprofiling/email/TemplateEmailSender` component to true.
3. Make sure that the `TemplateEmailSender` is in the `initialServices` list so that it will be started when the server is launched.
4. Check that caching is configured as shown for the following repository item descriptors in the `userProfile.xml` file.

   ```
   <item-descriptor name="mailing" cache-mode="locked" />
   <item-descriptor name="mailBatch" cache-mode="locked" />
   <item-descriptor name="mailServer" cache-mode="disabled" />
   ```

   Note the cache mode is set automatically in the `liveconfig` configuration layer, so if you are deploying a live site, you can simply add the `liveconfig` configuration layer to the environment for all the distributed e-mail servers. See the *Configuring Dynamo* chapter of your application server's *ATG Installation and Configuration Guide* for information about the `liveconfig` configuration layer.

However, if you are working in a non-production environment that does not include the `liveconfig` configuration, you must set the cache mode manually as shown above.

5. Check that lock managers are configured to prevent conflicts among threads. Specifically, check that you have a `ServerLockManager` enabled on the machine that acts as the lock server. On each machine that will render e-mail threads, make sure you have a `ClientLockManager` pointing to the `ServerLockManager`. For more information, see *Locked Caching* in the *ATG Repository Guide*.

6. If you are sending mailings programmatically, make sure that the template e-mail system is set up to perform persistent mailings. See Stopped E-mail Campaigns for more information. (If you are using scenarios to send the mailings, persistence is handled automatically.)

Setting Up a Mailing to Use Distributed E-mail Features

If you are using a scenario to send the distributed mailing, you do not need to carry out any configuration steps for the mailing itself. If you are sending the mailing programmatically, perform the following steps:

1. Set the `batched` property in the `TemplateEmailInfoImpl` object to true on the machine that will initiate the mailing. Do not set this property to true if the mailing's recipient list includes transient profiles.

2. Optionally, you can also set the `batchIfPossible` property of the `TemplateEmailSender` component on each machine to true. With this property enabled, the `batched` property is set to true automatically if a mailing's recipient list contains no transient profiles.

How Distributed Mailings Work

A distributed mailing is handled as follows: a request to start sending a mailing is received by one of the machines that you have designated as a distributed e-mail server. The machine (the send owner) creates entries in the `dps_mailing` and `dps_user_mailing` tables as it would for a non-distributed mailing. In addition, it separates the mailing into batches, creating entries for them in a `dps_mail_batch` table. Each batch entry specifies the ID of the parent mailing, the starting index of the first recipient in the batch, and the number of recipients in the batch. The `dps_mail_batch` table also includes a `unique_server_id` column, which contains the ID of the server that has claimed the batch for rendering.

When another machine that you have designated as a distributed e-mail server is started, it creates or updates a time entry in the `dps_mail_server` table. Then, at set intervals, the `TemplateEmailPeriodicService` component on that server does the following:

1. Updates the timestamp in the `dps_mail_server` table.

2. Checks to make sure that the previous machine shown in the `dps_mail_server` list has not timed out.

3. If the previous machine has timed out, it clears the `uniq_server_id` from all of that machine's in-progress batch mailings, thereby returning them to the pool of unclaimed batches.

4. Checks the `dps_mail_batch` table for available batch jobs. If jobs are available, it spawns threads to execute them (unless the threads already exist). You can control the number of threads that are spawned; for more information, see Performance Tuning Considerations for Distributed E-mail.

   The rendering threads use a lock in the `ClientLockManager` to avoid having two threads claim the same batch (which would trigger a concurrent update exception).

Each thread then does the following:

1. Claims the first job in the `dps_mail_batch` table by entering a value in the `uniq_server_id` column of that table.

2. Executes the job as it would in a non-distributed mailing, except that it updates the count in the `dps_mail_batch` table instead of in the `dps_mailing` table. (This behavior allows the system to avoid conflict over the `dps_mailing` row representing the mailing.)

3. When the batch is complete, it merges the counts for the batch into the entry in the `dps_mailing` table that represents the parent mailing. Once again, it uses a lock to prevent concurrent updates.

4. Changes the status in the `dps_mail_batch` entry to "complete."

5. Checks for another job.

Performance Tuning Considerations for Distributed E-mail

This section describes some recommendations for tuning your system to achieve and maintain optimal performance of distributed e-mail. Note that performance varies inevitably from case to case; therefore you should consider these values as a starting point only and experiment with the parameters listed to determine the values that best suit your configuration.

Note also that distributed e-mail is resource intensive. In addition to configuring the parameters described here, ensure that your database, network, and SMTP servers can handle the volume of traffic generated by distributed e-mail. It may be necessary to scale your hardware at each level of the targeted e-mail architecture to achieve the level of performance you require.

- **VM heap size**: Increasing the heap size can yield a measurable performance improvement. Suggested initial value: 1024MB per server.
- **Instance count**: ATG recommends having one distributed e-mail server per available CPU. ATG also suggests that the CPUs reside on a machine dedicated to distributed e-mail.
- **CPU binding**: Leaving all processes and processors unbound may provide the highest throughput for distributed mailings. For more information, refer to the documentation for your application server.

- **Thread handler count on distributed e-mail server**
  `numBatchThreads` property in the
  `/atg/userprofiling/email/BatchEmailPeriodicService` component

  The `numBatchThreads` property sets the number of threads that render e-mail. Suggested initial value: 20.

- **E-mail batch size**
  `emailsPerBatch` property in the
  `/atg/userprofiling/email/TemplateEmailBatchPersister` component

  This property specifies the number of recipient addresses that can be claimed at one time for e-mail rendering. Suggested initial value: 100.

- **E-mail chunking for SMTP server communication**
  `createMessagesBatchSize` property in the
  `/atg/userprofiling/email/TemplateEmailSender` component

  The e-mail handlers send messages to the SMTP server in chunks, and the `createMessagesBatchSize` property specifies the size of the chunk. ATG suggests setting this value to at least 4 (each rendered e-mail batch sent to the SMTP server contains four outbound messages).

- **Distributed e-mail status updates**
  `updateBatchCountsEveryNMessages` and
  `updateProfileStatusesEveryNMessages` properties in the
  `/atg/userprofiling/email/TemplateEmailSender` component

  These properties control the frequency of status updates to the database. Suggested initial value: the size of the e-mail chunks that are sent to the SMTP server.

- **Scenario maximum batch size**
  `maxBatchSize` property in the `/atg/scenario/ScenarioManager` component

  For very large mailings, tuning the `maxBatchSize` property helps control the size of the e-mail recipient list, which must be retrieved from the database. Suggested initial value: 1000 (the default setting). For more information, see Minimizing the Number of Collective Elements. Note that it is not necessary to change this setting if you are creating and sending mailing programmatically rather than through scenarios.

Deleting Mailings

Once an e-mail campaign is complete, you may want to delete it from the database. You can delete mailings manually through the ATG Dynamo Server Admin (see Viewing, Canceling or Resuming a Mailing), but you can also use either of the mechanisms described in this section to delete completed mailings automatically.

The `TemplateEmailPersisterImpl` class has a `deleteWhenComplete` property. If the value of this property is `true`, the `TemplateEmailPersisterImpl` service deletes the mailing immediately after it completes successfully. If the mailing does not complete successfully, or if the `deleteWhenComplete` property is set to false, the mailing is not deleted. The property is set to `false` by default.

In addition, the Personalization module includes an atg.userprofiling.email.TemplateEmailDeleter class that can delete mailings at scheduled intervals. You can use the mailingsToDelete property of the TemplateEmailDeleter service to specify the type of mailings to delete: completed, canceled, or failed mailings, or any combination of these.

The TemplateEmailDeleter service actually invokes the TemplateEmailPersisterImpl service to delete mailings, but the two deletion mechanisms operate independently. You can use the TemplateEmailDeleter service to schedule e-mail deletions even if the deleteWhenComplete property of TemplateEmailPersisterImpl service is set to false. Or you can set deleteWhenComplete to true (so that completed mailings are deleted immediately), and set up TemplateEmailDeleter to periodically clean up after failed or canceled mailings.

Typically, you want to invoke TemplateEmailDeleter when the server starts up, so you should add this service to the initialServices property of an atg.nucleus.InitialService component, such as the /atg/dynamo/service/Initial component.

The following table summarizes the properties of TemplateEmailDeleter:

Property	Function
templateEmailPersister	TemplateEmailPersisterImpl service that is invoked to delete the mailings. Default: /atg/userprofiling/email/TemplateEmailPersister
mailingsToDelete	List of status values for mailings eligible for deletion; can be any combination of completed, canceled, and failed. Default: completed
thisServerOnly	Indicates whether to delete only the mailings that are run from the server that this TemplateEmailDeleter is running on. Default: true
scheduler	Scheduler service used to schedule deletion of mailings. Default: /atg/dynamo/service/Scheduler
schedule	The interval at which mailings are deleted. Default: every 60 minutes

If your installation has multiple ATG servers connected to the same database, you must set the thisServerOnly property to the appropriate value. If you want to use TemplateEmailDeleter on a single server to delete mailings run from all of the servers connected to the database, set the thisServerOnly property to false. If you want to have one instance of TemplateEmailDeleter per server to delete only the mailings run from that server, set the thisServerOnly property to true.

Targeted E-mail Demo

The Scenarios module includes a demonstration Targeted E-mail application for evaluation purposes. This demo provides a working example of how you can use the Targeted E-mail API to create and send targeted e-mail. It comes with some pre-defined example mailings, and has an HTML interface to let you create your own. You can view the demo at `http://localhost:`*`port`*`/QuincyFunds/EmailDemo`. (The default port numbers on JBoss, Oracle WebLogic, and IBM WebSphere are 8080, 7001, and 9080, respectively. For more information, see the *ATG Quincy Funds Demo Documentation*.)

11 Personalization Module Tracking

Running a Web site on the Personalization module creates any number of events that the system can recognize and act upon. You can use events to change component properties and to implement logging and reporting on the performance of your sites. For example, you could trigger an event each time a user logs in. You could use this event to log and report on the number of people logging in. You could also have a profile attribute named `lastActivity`, and use the login event to reset the value of the `lastActivity` property in all users' profiles each time they log in.

The ACC's Targeting >Tracking Sensors window provides an interface for creating actions that are implemented in response to specific events. This chapter describes the Personalization module components that exist in the ACC's Tracking area. It includes the following sections:

Personalization Events

Action Handlers

Note: The tracking mechanisms described in this chapter are disabled by default, because scenarios and scenario recorders provide a more comprehensive means for performing event tracking (see Using Scenario Recorders). However, if you have existing ATG applications that use tracking sensors, you can enable tracking sensors by setting the Nucleus property `sendD4StyleEvents` to `true`. For more information, see *Implementing Event*s in the *Converting DPS Code* chapter of the *Dynamo 4.5-5.0 Migration Guide*.

Correspondingly, the Tracking Sensors window does not appear by default in the ACC. To display the Targeting > Tracking Sensors window, go to the People and Organizations > Control Center Groups window and enable the `Targeting: Tracking Sensors` Group UI Access Privilege for the appropriate group of users. If you do not have administrator privileges in the ACC you may not be able to modify this setting.

Personalization Events

To implement Personalization events you need to understand three main concepts:

- The Event Registry

  The event registry component is located in Nucleus at `/atg/registry/EventHandlers`. The event registry scans a directory in the config path that is specified in its `handlerPath` property. It loads any handlers in this directory that extend `GenericHandler`. The event registry points each handler component to listen to its default event channel.

The event registry also includes a pointer to an EventDistributor component.

- The Event Distributor

  The EventDistributor component is located in Nucleus at /atg/dynamo/service/event/Distributor. The EventDistributor recognizes a tree of registered event sources. Each path in the event source tree is referred to as an **event channel**. An event channel is a list of event sources, starting with the root source (/). You can register an event handler to listen to any node in the event source hierarchy. The Personalization module is configured to recognize the following event channels:

  - session
  - page
  - content
  - content type

- Event Triggers

  Event Triggers are servlets that trigger events based on your specifications. The Personalization module comes with some event triggers that are already configured. You can also create your own event triggers and configure them to trigger events in several different ways. See the Event Triggers and Setting up Event Triggers sections in this chapter for more information.

Event Triggers

The Personalization module has eight types of events that are configured and available out-of-the-box:

Event Channel	Event Name	Triggered when . . .
session	login	User logs in
session	logout	User logs out
session	register	User registers
session	new session	Set by SessionEventTrigger servlet
session	expire session	Set by SessionEventTrigger servlet when session expiration event is received from SessionManager
page	page view	A page is displayed to the user; triggered by PageViewServletTrigger pipeline servlet, or by a SendPageEvent servlet bean
content	content view	A content item is displayed to the user; triggered by targeting servlet beans
content-type	content-type view	A content item of a particular content type is displayed to the user; triggered by targeting servlet beans

Setting Up Event Triggers

You can trigger each type of event in one or more ways.

Triggering Session Login, Logout, or Register Events

A session login, logout, or register event is triggered by default whenever a user successfully submits a form created with the Profile Form Handler methods. See the *Working with Forms and Form Handlers* chapter in the *ATG Programming Guide* for more information about how the Profile Form Handler handles login, logout, and registration events.

Triggering New Session Events

The Personalization module adds a new servlet, `SessionEventTrigger`, to the ATG servlet pipeline. This pipeline servlet checks each request to see if it represents the start of a new session. If so, it triggers a new session event, which is sent to the Event Distributor's session channel.

Triggering Page View Events

The `PageViewServletTrigger` is another pipeline servlet. This servlet triggers a page view event for each page that is served.

Triggering an event for every page viewed could adversely affect performance on a large site. Instead, you may want to trigger page view events only for particular pages. To do this you can disable the `PageViewServletTrigger` by setting its `broadcastPageViewedEvents` property to `false`. Then, set the page view event trigger in each Page, using an ATG Servlet Bean with the Nucleus path `/atg/userprofiling/SendPageEvent`, as in this example:

```
<droplet bean="/atg/userprofiling/SendPageEvent"></droplet>
```

Any time a page with this tag is rendered, the `SendPageEvent` component triggers a page view event and sends it to the Event Distributor's page channel.

You can configure the `SendPageEvent` component with an optional parameter named `pageviewed`. The value of `pageviewed` sets a name for the page being viewed in the `path` property of the `PageViewedEvent`. If you do not set a value for the `pageviewed` parameter, then this value defaults to the `requestURI` property of the request object.

By default, the `PageViewServletTrigger` pipeline servlet and `SendPageEvent` servlet bean strip out any query arguments from the URI in the page event. If you want to retain the query arguments in the URI, set the `removeURIArguments` property of `/atg/userprofiling/PageEventTrigger` to `false`.

Page view events are triggered each time the ATG server downloads a page. They are not repeatedly triggered in the following instances unless a user deliberately decides to reload a page:

- when a user views a page that was cached by their browser,
- when the page has an HTML rather than a JHTML or JSP extension,
- if the user allows setting cookies instead of using a session ID.

Triggering View Item Events

Each of the targeting servlet beans described in the *Serving Targeted Content with ATG Servlet Beans* chapter of the *ATG Page Developer's Guide* can trigger a JMS `ViewItem` event for each repository item returned by the targeter. For example, suppose a page with a `TargetingForEach` servlet bean returns an array of seven content items. When the system serves this page, seven `ViewItem` events are sent.

You can enable this behavior for any targeting servlet by setting the following parameters in the servlet:

```
<param name="fireViewItemEvent" value="true">
```

For more information on the JMS `ViewItem` event, see ViewItem Event in the part of this manual that describes scenario features.

Note: Earlier versions of ATG products used `fireContentEvent` and `fireContentTypeEvent` servlet bean parameters instead of the `fireViewItemEvent` parameter. These earlier parameters were triggered when the targeter returned content repository items (but not items from other repositories). They sent content view and content-type view events to the Event Distributor's content channel and content-type channel, respectively. Although these parameters still work as intended in any existing targeters, they have been deprecated in favor of the more flexible `fireViewItemEvent` parameter, which is triggered when an item from any repository (including, for example, the profile repository) is viewed.

If an existing targeter uses the `fireContentEvent` and `fireContentTypeEvent` parameters, and you want to prevent the targeter from firing any events, you must set the parameters as follows in addition to turning off the `fireViewEvent` parameter described above:

```
<param name="fireContentEvent" value="false">
<param name="fireContentTypeEvent" value="false">
```

TargetedContentTrigger Servlet Bean

The `atg.targeting.TargetedContentTrigger` servlet bean is designed to be nested in the `<oparam>` tag of another targeting servlet. You can use it to enable or disable an event trigger in a particular targeting servlet bean instance.

The `TargetedContentTrigger` expects the following parameters to be available from the servlet bean that contains it:

`targeter`
a `RepositoryTargeter` serving the content

`element`
a `RepositoryItem` representing the viewed content

The `TargetedContentTrigger` by default fires both a content event and a content type event. If you don't want to fire one of the events, you can set one of the following parameters to false:

`fireContentEvent`—set to false to prevent the content event from firing

`fireContentTypeEvent`—set to false to prevent the content type event from firing

For example, if you want to set the `TargetedContentTrigger` servlet bean to trigger a content view event, and not a content type event, use the following code.

JSP example:

```
<dsp:droplet name="/atg/targeting/RepositoryLookup">
  <dsp:param bean="/atg/userprofiling/SendPageEvent" name="targeter"/>
  <dsp:param value="viewedContent" name="element"/>
    <dsp:oparam name="output">
       <dsp:droplet name="/atg/targeting/TargetedContentTrigger">
          <dsp:param value="false" name="fireContentTypeEvent"/>
       </dsp:droplet>
    </dsp:oparam>
</dsp:droplet>
```

JHTML example:

```
<droplet bean="/atg/targeting/RepositoryLookup">
  <param name="targeter" value="bean:/atg/userprofiling/SendPageEvent">
  <param name="element" value="viewedContent">
    <oparam name="output">
       <droplet bean=/atg/targeting/TargetedContentTrigger>
          <param name="fireContentTypeEvent" value="false">
       </droplet>
    </oparam>
</droplet>
```

For more information, see *Nesting Servlet Beans* in the *Using ATG Servlet Beans* chapter of the *ATG Page Developer's Guide*.

Note, however, that the `fireContentEvent` and `fireContentTypeEvent` parameters used by the `TargetedContentTrigger` have been deprecated. Refer to the previous section, Triggering View Item Events, for more information.

Event Action Queue

If an event is logged to a file or a database with every request, and the events are handled synchronously, the file can become a performance bottleneck. To make the events asynchronous, you must create a queue between the event source and the event sink. By default, Personalization module events are sent to the event Action Queue component, located at `/atg/dynamo/service/event/handlers/ActionQueue`. These events are handled either when the queue fills up or at a time that you can schedule.

You can configure events to trigger immediately, without passing to the Action Queue, by setting the `queueActions` property in the Action Handler to `false`.

Action Handlers

One primary use for events is changing properties of components in response to an event. The easiest way to implement this is with the Targeting > Tracking Sensors task area of the ACC. See *Tracking Visitor Behavior* in the *ATG Personalization Guide for Business Users*. The Personalization module implements tracking sensors through **action handlers**. An action handler component implements `atg.service.event.ConfigurableActionHandler`. Action handler components have an `actions` property that lists an array of the actions (`atg.service.event.ConfigurableAction`) to take in response to an event.

Event Properties

An event's properties depend on its event channel. These properties represent objects available to action handler components when the event is triggered. The available properties are:

Event Channel	Properties
session	profile, request, session
page	profile, request, path
content	profile, request, content item
content-type	profile, request, content item

ConfigurableAction Properties

Each `ConfigurableAction` component has the following properties that define the action:

Property	Function
`modifiedPropertyName`	The name of the property to modify. The value of this property should specify both the component and the property to modify. The only properties that can be modified are those of the event that was fired. For example, to modify the `member` property of a `Profile` component, use `modifiedPropertyName=Profile.member`.

`operator`	How to modify the property. The following operators are available: `APPEND` edits the property value by appending a string to the existing value `APPEND ONCE` appends a value to an array only if the value doesn't exist already in the array `ADD` changes numerical property values `SUBTRACT` changes numerical property values `ASSIGN` replaces the old property value with a new one
`modifierPropertyName`	The modifier property (i.e. what to add, subtract, append, etc. to the modified property). Use this for a dynamic modifier that is a property of a bean. The property can be any of the properties of any of the beans that are properties of the event. For instance, a page event has `profile`, `request`, and `path` properties. You can use any of the properties of the `profile`, `request` or `path` objects as modifiers. The data type of the modifier property should match that of the modified property.
`modifierStaticValue`	The modifier value, i.e. what to add, subtract, append, etc to the modified property. Use this for a static value. The Personalization module accepts a string value for this property and converts it to the appropriate data type of the `modifiedPropertyName`.

For example, suppose you have a gardening news page and a profile attribute named `likesGardens`. You can set an event trigger on your gardening news page to send a page view event. Then, create an Action Handler that refers to a `ConfigurableAction` component with the following properties:

```
modifiedPropertyName=Profile.likesGardens
operator=ASSIGN
modifierStaticValue=true
```

This `ConfigurableAction` sets the `likesGardens` property to true when a page view event is fired by the gardening news page. In the alternative, if the `likesGardens` property were an integer type, rather than a Boolean, you could use these properties:

```
modifiedPropertyName=Profile.likesGardens
operator=ADD
modifierStaticValue=1
```

to increment the `likesGardens` property in response to the event.

For another example, suppose you want to keep track of the last page a visitor saw, so that you can return him to the same page on his next visit. You might have a `lastPageView` attribute in your profile template. You can have a `ConfigurableAction` that uses page event to set this attribute:

```
modifiedPropertyName=Profile.lastPageView
operator=ASSIGN
modifierPropertyName=URLPath
```

If you want a profile property that tracks each page a visitor sees, you might use the APPEND ONCE operator to keep a list of pages viewed:

```
modifiedPropertyName=Profile.pageViewed
operator=APPEND ONCE
modifierPropertyName=URLPath
```

Default Action Handlers

By default, the Personalization module comes with two action handlers configured:

- `/atg/registry/EventHandlers/actions/AssignLastActivityAction`, which automatically sets a value of the `lastActivity` profile attribute on login
- `/atg/registry/EventHandlers/actions/AssignRegistrationDateAction`, which automatically sets a value of the `registrationDate` profile attribute on registration

If your profile template does not include the `lastActivity` and `registrationDate` attributes, then the presence of these action handlers can cause errors. To disable these action handlers, set the `registerAtStartup` property to `false` on the `/atg/registry/EventHandlers/LoginHandler` and `/atg/registry/EventHandlers/RegisterHandler` components.

12 Personalization Module Logging

When you send events to the ATG logging system, you can record useful information about the operation of your Web application. The Personalization module's logging system handles three categories of log entries:

- page requests (from URLs)
- user events (such as new session, login, etc)
- content viewed from Content Repositories.

Each of these three categories of log entries has its own components to generate raw log entries. Each log entry is then passed through instances of Data Collection services, which store the entries in flat files or in SQL database tables. To understand how the logging services manage log entries after creation, see the *Logging and Data Collection* chapter of the *ATG Programming Guide*.

This chapter includes the following topics:

Note: The logging mechanisms described in this chapter are disabled by default because scenario recorders provide a more comprehensive means for performing logging tasks (see Using Scenario Recorders). However, if you have existing ATG applications that use logging, you can enable the logging mechanisms described in this chapter by setting the Nucleus property `sendD4StyleEvents` to `true`. For more information, see *Implementing Event*s in the *Converting DPS Code* chapter of the *Dynamo 4.5-5.0 Migration Guide*.

Logging Events

Logging events are handled by an Action Handler located at `/atg/registry/EventHandlers/LoggingEventHandler`. This logging handler is configured to listen for events on the page, session, and content event channels with a property like this:

```
channels=/page/,\
         /session/,\
         /content/
```

For each event channel, the `LoggingEventHandler` has a property that points to a logging service:

Event Channel	Property Name	Logging Service
page	`requestLogger`	`/atg/reporting/RequestLogging`
session	`userEventLogger`	`/atg/reporting/UserEventLogging`
content	`contentViewedLogger`	`/atg/reporting/ContentViewedLogging`

Logging Services

All logging and reporting components live in the Nucleus folder `/atg/reporting/`. There are three log generators that define three different logging events. These events are distributed through standard Java Event Listeners and are contained in seven different logging sinks. Each of the raw log entries are uniquely identified in their category (i.e. request, user event and content viewed log entries).

The three main logging services are:

- `/atg/reporting/RequestLogging`
- `/atg/reporting/UserEventLogging`
- `/atg/reporting/ContentViewedLogging`

Request Logging

The `RequestLogging` service is an instance of `atg.reporting.datacollection.RequestLogging`. This interface defines methods that generate a request log entry and send it to an assigned set of Data Collection listeners. In addition, the ID of the log entry is returned and associated with the request by the `get/setRequestId` methods. This request ID allows content viewed log events to associate what content is extracted from a Content Management System with a specific request. A centralized event handler listens for Page Events and then generates the request log entry. These Page Events are, by default, fired for every HTTP request that is serviced by the ATG server. Obviously this may generate more data than is required. You can reconfigure the `RequestLogging` service to generate page events only for the pages in which you are interested. For more information, see the Personalization Module Tracking chapter in this manual.

User Event Logging

The UserEventLogging service tracks the events generated by a user's typical life cycle on a Web site. These include:

- creating a new session
- ending a session
- logging in
- registering
- logging out of the site

The UserEventLogging service is an instance of atg.reporting.datacollection.UserEventLogging. The UserEventLogging service provides methods that generate different log event types when a user creates a session, logs into a site, or does any of the other actions described above. The RequestLogging event handler also listens for Session Events and generates a log entry associated with each type of event using the UserEventLogging component.

Content Viewed Logging

The ContentViewedLogging service is similar to the RequestLogging service. It generates log events created when a user accesses content pulled from an Open Content Adapter (OCA). These actions usually occur when a targeter is configured to fire Content Viewed events for each piece of content returned from the targeting operation. The ContentViewedLogging service requires the current request ID to generate a content viewed log entry. It uses the RequestLogging service to extract this ID from the HttpServletRequest object. The ContentViewedLogging service is invoked from the same event listener as the RequestLogging service, but it looks for Content Events and generates the appropriate log entries associated with the content request.

Log Entry IDs

All logging services use a log entry ID generator service to generate unique IDs for each logging entry type. This component uses a database table (dps_logging_id_counter) to keep track of current log entry IDs. These ids are generated in batches for each ATG server to minimize the amount of database contention. The batch size can be configured through the logging service's idBurnFactor property. The default value is 100, but servers that are in a production environment might want to increase this value (e.g. to 1000) depending on load.

Data Listeners and Queues

Every logging service extends the GenericDataListenerService class and has a dataListeners property. This property defines which Data Collection services operate on the log entry objects generated from the logging services. The DataListenerQueue component configures each logging service to send logging events to its own unique listeners.

For example, the UserEventLogging service has this property:

```
dataListeners=userevents/QueueSink
```

Each QueueSink has a dataListeners property that specifies how to log events. By default, the system sends all log entries directly to the FileSink, as specified by this property:

```
dataListeners=FileSink
```

To send all log entries to a database instead, configure the dataListeners property as follows:

```
dataListeners=SQLSink
```

See Logging to a Database for more information about changing the Personalization module's configuration to use SQLSinks rather than FileSinks.

Each event channel is logged separately, so each of the following sinks are available:

```
userevent/SQLSink
userevent/FileSink
requests/SQLSink
requests/FileSink
contentviewed/SQLSink
contentviewed/FileSink
```

If a site generates a very large load of log entries, you may not be able to put all log entries into the database fast enough. This could result in the queue backing up and the potential loss of data. One way to avoid this problem is to use a flat file as your logging queue sink. You can create a routine to import the data from the flat file into your database at appropriate intervals for use in reporting.

The purpose of the queues in the logging system is to optimize the request handling time for the user. Storing the logged data in the scope of a request can be a relatively time consuming operation. Each queue has its own thread management that allows it to issue log entries to the various output sinks outside the scope of the request thread. This allows the request to hand off the log entry very quickly and it does not get slowed down by the data store of the logging data.

The discrete log entries are stored persistently in:

- a flat file, through an instance of atg.service.datacollection.FormattingFileLogger or
- a relational database, through an instance of atg.service.datacollection.SQLTableLogger.

By default, the queue forwards the log entries to a file sink. SQL sinks have been pre-configured in the Personalization module's installation, but have not been activated.

Once the log entry reaches the file or SQL sink, the Data Collection formatting syntax is used to determine how to represent the log entry. Each logging service uses a specific JavaBean to represent the log entry. The properties of the JavaBeans can be used in the formatting syntax. For more information on the properties available for logging, see the *ATG API Reference*.

Configuring QueueSinks

You can configure the behavior of each `QueueSink` with the `schedule` and `dataItemThreshold` properties. The `schedule` property sets a time schedule for flushing the queue and writing all the events in the queue to a log. The `dataItemThreshold` sets a maximum number of events to accumulate in the queue before writing all the events in the queue to a log. The queue is flushed when it either meets the schedule or reaches the item threshold. Both of these properties are set centrally for all QueueSinks in the `SQLFlushSchedule` and `dataItemThreshold` properties of the `/atg/reporting/Configuration` component. You can also customize each `QueueSink` with its own properties.

FileSinks

The `FileSink` components are `atg.service.datacollection.FormattingFileLogger` class components that log events to a flat file. A `FileSink` has properties that specify the directory and file name prefix of the event log. For example, the `requests/FileSink` component has these properties:

```
logFileDir^=../Configuration/dataLogFileDir
logFileName=request_
```

Other properties of the `FileSink` components refer to the `/atg/reporting/Configuration` component to specify the exact format of the log file and log file name and the log's rotation schedule. By default, the log files include timestamps in their file names, so you may see a user events log file name like this:

```
userevents_02-09-2001_18-36-03-55.data
```

For more information about logging services, see the *Logging and Data Collection* chapter in the *ATG Programming Guide*.

SQLSinks

The `SQLSink` components are `atg.service.datacollection.SQLTableLogger` class components that log events to a SQL database table. A `SQLSink` has two properties that specify the database table and column mapping to use when logging events to the database. For example, the `requests/SQLSink` component has these properties:

```
tableName=dps_requests
SQLColumnMappings=id:id,timestampAsDate:timestamp,\
                  sessionId:sessionid,name:name,memberAsDBValue:member
```

The `SQLSink` also has a read-only property that sets the SQL INSERT statement to use when writing log events to the database.

Log Files

By default, the Personalization module sends all logging data into flat files to optimize performance for large sites. Depending on the load of a Web site, you may or may not want to store all logging data directly in a database. Typically, large sites perform nightly bulk copy operations from the flat file logs into databases if they want their data available for querying. The Personalization module's log file names are time-stamped down to the millisecond, to make them more or less unique on server restarts.

All aspects of the file names for the logs can be configured. This is especially useful if you want to uniquely identify each log file per ATG server. In addition, you can set a file sink's schedule property to automatically flush and close files currently being used and rotate to a new log file. By default, this schedule is set in the `logFileRotationSchedule` property of `/atg/reporting/Configuration` to perform this operation at 1 AM every night. See the *Logging and Data Collection* chapter in the *ATG Programming Guide* for more information about formatting and scheduling logs.

Configuring Log Operations

You may want to modify some of the configuration properties in production environments to optimize performance. To help aid in this across all the logging services there is a global component at `/atg/reporting/Configuration`, which has a set of properties that the different logging services link to. This lets you change logging behavior for logging services globally, rather than changing the configuration of each logging service separately. The following table describes the properties that you can globally configure:

Property Name	**Function**
`enableLogging`	Flag to determine if the logging operations should be enabled. To disable all logging for the server set this to false. Default: `true`
`dataItemThreshold`	The number of items to log at once (in a batch). Default: `10`
`SQLFlushSchedule`	The schedule that the SQL loggers should use to periodically flush their data to the database. Default: `every 10 minutes in 10 minutes`
`summarizationThreshold`	The number of log entries to collect before generating a summarized log entry. Default: `100`
`summarizationFlush Schedule`	The schedule that the summarizers should use to output their current statistics. Default: `every 10 minutes in 10 minutes`
`logFileRotation Schedule`	Schedule to determine when the log files should be rotated. (See the *Core Dynamo Services* chapter in the *ATG Programming Guide*.) The default setting has the logs rotated every day at 1am. Default: `calendar * * * 1 0`

`idBurnFactor`	How many log entry IDs should be reserved in one batch. Default: `100`
`dataLogFileDir`	The directory in which to place all the log data files. Default: `logs`
`scheduler`	The centralized ATG scheduler service. Default: `/atg/dynamo/service/Scheduler`
`timestampLogFileName`	Adds a time-stamp to all the names of the log files. Default: `true`
`logFileExtension`	Uses this extension for the log file. Default: `data`
`timestampDateFormat`	Format for the time stamp, based on `java.text.SimpleDateFormat`. The default value formats dates by `month-day-year_hour-minute_second-millisecond`. Default: `MM-dd-yyyy_HH-mm-ss-SS`

Enabling and Disabling Logging

The Personalization module is configured to log all events in the page, session, and content event channels. This means logging every page request and every content item for every user. You may or may not want to log every such event. You can configure logging to limit the events that are captured in the logging system, which may improve performance at the cost of losing some information.

For example, you may to log only the page view events triggered by certain specific pages. To implement this behavior:

1. Disable the `PageViewServletTrigger`, which triggers page view events for all pages served.
2. Embed the `/atg/userprofiling/SendPageEvent` servlet bean in each page that you want to trigger a page view event.

You can also disable logging of an event channel by deleting that channel from the `channels` property of the `/atg/registry/EventHandlers/LoggingEventHandler` component.

You can disable all logging at startup by setting the `enableLogging` property in the `/atg/reporting/Configuration` component to `false`.

Logging to a Database

The default logging configuration logs to flat files. If you want to change the default setup and have everything logged into a database instead, there a few simple configuration changes required.

1. Make sure the appropriate database tables are defined. The DDL for these tables are defined in the `<ATG10dir>/DPS/sql/db_components/<`*`DB_PLATFORM`*`>/logging.ddl` file.
2. If this is the first time you are creating all the logging tables, you need to initialize the appropriate user event and ID tables with the `logging_init.sql` file located in the same directory as the `logging.ddl` file. If you used the file `<ATG10dir>/DPS/sql/install_dps/<`*`DB_PLATFORM`*`>/init_db.sql` to initialize the database, then you do not need to separately create the logging tables.
3. Set the `dataListeners` property of the following components to change the destination from the file sinks to the SQL sinks:

Component Name	Value
/atg/reporting/requests/QueueSink	/atg/reporting/requests/SQLSink
/atg/reporting/requests/RequestNameSummarizerQueueSink	/atg/reporting/requests/RequestNameSummarizerSQLSink
/atg/reporting/requests/SessionSummarizerQueueSink	/atg/reporting/requests/SessionSummarizerSQLSink
/atg/reporting/userevents/QueueSink	/atg/reporting/userevents/SQLSink
/atg/reporting/userevents/EventTypeSummarizerQueueSink	/atg/reporting/userevents/EventTypeSummarizerSQLSink
/atg/reporting/contentviewed/QueueSink	/atg/reporting/contentviewed/SQLSink
/atg/reporting/contentviewed/ContentIdSummarizerQueueSink	/atg/reporting/contentviewed/ContentIdSummarizerSQLSink

As an alternative, you can create a properties file for the component in the `localconfig` directory or another appropriate directory in your configuration path and set the `dataListeners` property to the value from the table above.

Limiting Input to the Database

The amount of data generated and processed through each non-summarized `QueueSink` is much greater than the summarized `QueueSinks`. Therefore, in production environments, the load might preclude sending all of the `QueueSink's` logging events directly to the database, but the summarized logging events could be sent directly to the database. You can choose to configure each `QueueSink` individually as needed.

By default, the summarizer components are disabled. You can enable the summarizer components and disable the non-summarized loggers by setting the `dataListeners` property for the `RequestLogging`, `UserEventLogging` and `ContentViewedLogging` components. For example, the following configuration would only log summarized data:

Component Name	dataListeners Property Value

`/atg/reporting/RequestLogging`	`requests/RequestNameSummarizer,`
	`requests/SessionSummarizer`

`/atg/reporting/UserEventLogging`	`userevents/EventTypeSummarizer`
`/atg/reporting/ContentViewedLogging`	`contentviewed/ContentIdSummarizer`

You can modify the `dataListeners` property by editing the value of the data event set in the Events tab of the Component Editor. You can enable the summarizers while retaining the non-summarized loggers by adding the values in the table above to the respective `dataListeners` properties.

Generating Reports

You can generate reports on your Web application activity from SQL table data. This SQL table data can be generated from one or both of the following sources:

- Data coming from and manipulated by the Personalization module
- Data generated by the Batch Reporting Service

The Batch Reporting Service takes content information out of a repository and places it into a dynamically generated SQL table. It also generates a table of content groups, a table mapping content to content groups, and a table mapping profiles to profile groups. These tables are necessary for any content or profile related reports.

Batch Reporting Service

The Batch Reporting Service makes it possible to correlate information from Personalization module applications to SQL database tables. You run the Batch Reporting Service at intervals to allow new information to be gathered from a repository into a dynamically generated database table. This service formats information into tables as follows:

- Collects content from repository
- Generates content groups
- Maps content to content groups
- Maps profiles to profile groups

These tables are necessary for content or profile related reports to be generated with up-to-date information. The Batch Reporting Service is a Dynamo service, so you can schedule it to run at convenient times.

The Batch Reporting Service component is located at:

```
/atg/reporting/tools/BatchReportingService
```

The Batch Reporting Service is initially turned off. You can turn on the Batch Reporting Service by setting the following property:

```
enableBatchReporting=true
```

Also, you need to add the Batch Reporting Service to Initial Services.

You can configure the Batch Reporting Service to do the following:

- set the schedule on which the Batch Reporting Service gathers information
- point to a repository
- point to a registry service

Note: You should not run the Batch Reporting Service on an ATG instance that serves a live site.

Setting the Schedule

The `schedule` property of the Batch Reporting Service determines when and how often it queries the system for reporting information. The default setting of this property is:

```
schedule=calendar * . * 4 0
```

This calendar frequency is every morning at 4:00 am. Calendar takes frequency in the following order:

```
month  day  weekday  hour  (military)  minute
```

In the above example: the first asterisk (*) means every month. The period (.) means no day specified. The second asterisk (*) means all weekdays. The 4 means 4:00. The 0 is for no minutes.

Note: When you set the Batch Reporting Service to run periodically, keep in mind that it puts a load on the system. You should consider running it at off-peak hours (like the middle of the night). Also, you should not run the Batch Reporting Service on a server that manages your public Web site.

Pointing to a Repository (Content)

The `repository` property specifies the Nucleus address of the content repository whose activity you want to report on. For example:

```
repository=/atg/adapter/html/TargetedContent
```

Pointing to a Registry Service

The `registry` property specifies the Personalization module component that registers content groups and profile groups. By default, this property is set as follows:

```
registry=/atg/registry/RepositoryGroups
```

Fine Tuning Updates

You can configure the Batch Reporting Service to fine tune updates on content, content traits, content groups, and profile groups so that only a limited percentage of content is copied over from the repository. This reduces load on the system and speeds up Batch Reporting Service updates.

By default, all **Content** information is transferred from the repository by the following property setting:

```
transferAllContent=true
```

To limit the transfer of content so that only content pieces that have been requested by site visitors are transferred, set `transferAllContent` to `false`.

By default, all **Content Traits** information is transferred from the repository by the following property setting:

```
transferAllContentTraits=true
```

To specify content traits to report on, set them as values of the `contentTraitList` property:

```
contentTraitList=
```

By default, all **Content Groups** information is transferred from the repository by the following line:

```
transferAllContentGroups=true
```

To specify the content groups to transfer, set them as values of the `contentGroups` property:

```
contentGroups=
```

By default, all **Profile Groups** information is transferred from the repository by the following line:

```
transferAllProfileGroups=true
```

To specify profile groups to transfer, set them as values of the `profileGroups` property:

```
profileGroups=
```

Part II: Scenarios Module Programming

The Scenarios module extends the content targeting capabilities of the Personalization module, providing a set of advanced targeting features that you can use to plan and manage personalized customer relationships. You use the Scenarios module to do the following:

- Create scenarios, which are event-driven campaigns designed to manage interactions between site visitors and content over a long period of time.
- Create workflows, which are similar to scenarios, but can be customized to model a wide variety of business processes.

Note that the Scenarios module is included with the ATG platform.

The Scenarios module section of this guide contains the following chapters:

Adding Custom Events, Actions, and Conditions to Scenarios
Shows how to add custom scenario elements.

Filtering Collections
Describes how to filter objects in a collection and cache the resultant content. Also describes how to create custom collection filters.

Using Profile Markers
Explains how to set up and use markers on profiles.

Defining and Tracking Business Processes
Describes how to define a business process as a series of stages, track activity within the business process, and report on the activity for a specified time frame.

Creating and Configuring Workflows
Describes how to use the Workflow API to create new workflow types for modeling business processes.

Managing Workflows on Multiple Servers
Shows how to set up a process editor server, global servers, and individual servers for workflows. Also includes a list of the workflow repository item descriptors that you should configure to use locked caching.

Setting Up Security Access for Workflows
Describes the levels of workflow access control you can set up for both ACC users and site users.

Configuring the ATG Expression Editor
Describes how to define or modify the grammar that makes up part of the ACC such as the targeting rules editor or the scenario editor.

Note: You (or your business users) create scenarios, reports, and workflows in the ACC. This manual does not cover these tasks but focuses instead on the configuration aspects of the Scenarios module. For detailed information on creating scenarios, reports, and workflows, please refer to the *ATG Personalization Guide for Business Users*.

13 Overview of the Scenarios Module

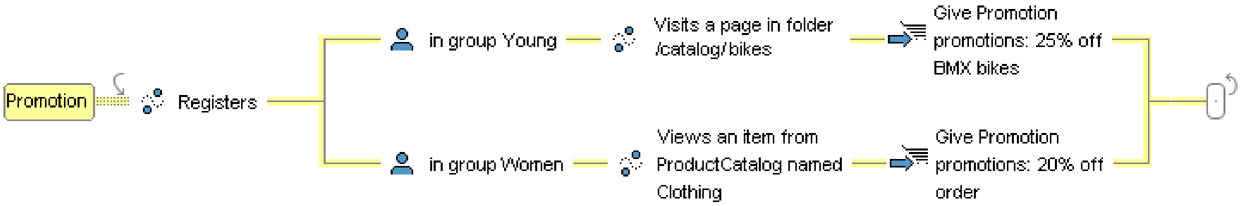
This chapter describes the features and implementation of the scenario server (also referred to as the scenario engine), which is the server component responsible for processing and executing scenarios.

You create scenarios in the ACC. For detailed information, refer to *Creating Scenarios* in the *ATG Personalization Guide for Business Users*. **Important:** If you want to edit scenarios through the ACC on the management server (the server where ATG Content Administration and the ATG Business Control Center are installed), you must run the ATG platform with the Preview layer. For more information, see the *ATG Business Control Center Administration and Development Guide*.

You cannot create scenarios through the ATG Business Control Center.

Scenario Basics

Consider the following simple example of a scenario:


Promotion
Registers
in group Young
Visits a page in folder /catalog/bikes
Give Promotion promotions: 25% off BMX bikes
in group Women
Views an item from ProductCatalog named Clothing
Give Promotion promotions: 20% off order

This scenario gives a one-time promotion to newly registered users of a bike store. Young people will get 25% off BMX bikes if they visit any bike-related pages in the store; female users will get 20% off their order if they view any clothing items.

One can think of a scenario as a channel or pipe through which subjects (users) flow. At the start of the scenario, the set of all users (whether currently active or not) are in the pipe. As subjects move down the channel, they encounter scenario elements that affect their progress in various ways. Events (such as “Registers”) prevent the subject from proceeding further until the event occurs. Condition elements (such as “in group Young”) narrow the set of subjects passing through the pipe; only those subjects who satisfy the condition can proceed further. Action elements (such as “Give Promotion”) carry out some action (usually relative to the subject). Finally, the pipe may fork, in which case subjects flow down multiple branches in the fork, which diverge and then rejoin. There are two kinds of forks: regular (the subject leaves the fork as soon as it arrives at the join point via one or more paths), and synchronized (the subject leaves the fork only when all paths in the fork arrive at the join point).

In the example, if the user doesn't qualify as either young or female, he does not proceed past the "Registers" element in the scenario. A young male proceeds via the upper branch in the fork, where the scenario must wait for him to visit an appropriate page before giving him the promotion; likewise, an old female proceeds via the lower branch. A user who is both young and female proceeds via both branches; the promotion she gets depends on which of the two events ("Visits a page" or "Views an item") happens first. Because the fork is regular, once she does receive one of the promotions, she will exit the fork and exit the scenario.

Scenario Processing

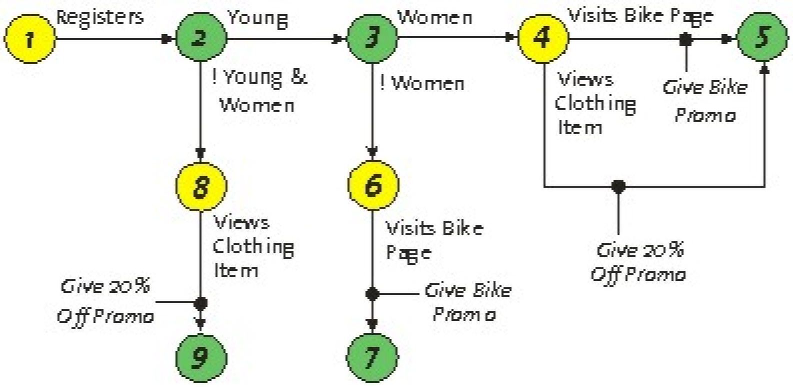
The scenario server executes scenarios as follows. Note first of all that scenarios are event driven. Once an event occurs, users can quickly proceed through all the consecutive conditions and actions until they either encounter another event, or exit the scenario. Thus, the scenario server itself is event driven; events in the ATG system are JMS messages, and the scenario server acts as a `MessageSink`. You can configure it (through Patch Bay) to listen to various events (by default, it receives all the standard ATG events). As each event arrives, the scenario server determines, for each scenario, which user or users are affected, and advances those individuals through the scenario until the next wait is encountered. It then waits for the next event to arrive, and so on.

In order to keep track of where in the scenario the users are, the scenario server must maintain some state for every individual going through the scenario—most notably, it must maintain the individual's location in the scenario. (Another piece of state is a user-specific map of scenario variables, which can be set via the "Set variable" scenario action; for more information, refer to the *ATG Personalization Guide for Business Users*.)

Maintaining the per-user scenario state is complicated by forking. The example above shows that, because of forking, a user can be in two different places in the scenario at once, waiting for two different events to occur. A much more complicated scenario might have many forks, and forks inside forks, so that keeping track of all the states associated with just a single user could become quite problematic. To simplify the matter, the scenario server does not actually attempt to execute scenarios the way they are defined. Instead, it transforms each scenario into an equivalent scenario state machine (SSM). This behavior requires an additional processing step when the scenario is first created and started, but it provides some important advantages.

The main virtue of the scenario state machine is that a given individual is always in exactly one SSM state, and progresses through a strictly linear sequence of states with no forking. In addition, the state machine is constructed in a way that provides various useful guarantees concerning its structure.

The state machine for our promotion example can be described as follows. An SSM has two kinds of states: waiting (shown in yellow) and intermediate (green). Waiting states are where the scenario execution must stop in order to wait for the next event (or several events, as is the case with state 4 below).



Intermediate states are used to channel users to different paths through the state machine. For instance, users who arrive at state 2 in the example proceed to either state 3 or state 8, depending on whether they are young or old and female (old males don't proceed beyond state 2, since there is no transition for them to take). Note that the conditions (or filters) leading out of the intermediate state are always mutually exclusive; thus, any individual user follows a single linear path through the SSM. Scenario actions take place as part of the SSM transitions; for example, a user is given a bike promotion as part of the transition from state 4 to state 5, and so on.

Scenario Definition Files

When a scenario is created or modified in the ACC, its definition is saved in the scenario registry (located at the Nucleus path `/atg/registry/data/scenarios`) as a file with the extension `.sdl`. Scenarios are defined using the XML-based Process Definition Language. The Process Definition Language DTD (`pdl.dtd`) describes the formal syntax of the language. (The PDL DTD is located in `<ATG10dir>\DSS\lib\classes.jar`. For an example of a scenario definition file, look at one of the `.sdl` files in your scenario registry.) The scenario server reads the `.sdl` files from the registry and parses them into scenario objects.

After the scenario is parsed, each segment of the scenario is converted into an SSM.

Scenario Execution

This section describes how the scenario server handles the progression of site visitors through a scenario.

Individual Scenario Instances

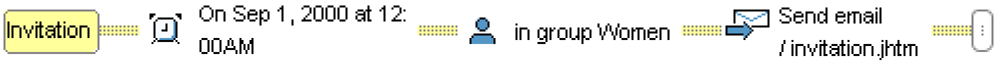
As noted earlier, the scenario server has to maintain some amount of state information for every individual going through each scenario segment. This state is stored in the profile repository. Specifically, the user item descriptor in the `/atg/userprofiling/ProfileAdapterRepository` component (and the associated `/atg/userprofiling/userProfile.xml` template file) has a `scenarioInstances` property, which contains a set of individual scenario instances (repository items of type `individualScenario`) currently associated with that user.

Each individual scenario instance maintains the state associated with a particular scenario segment through which the user is progressing. Individual scenario instances have the following properties:

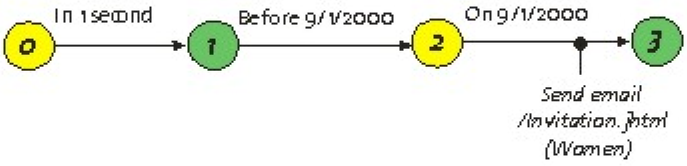
- `id`: the repository ID of this individual scenario instance.
- `processName`: the name of the scenario.
- `modificationTime`: the time the scenario was last modified (see the next section).
- `segmentName`: the name of the scenario segment.
- `creatorId`: the repository ID of the scenario instance which created this instance (see below).
- `state`: the ID of the SSM state this scenario instance is currently in.
- `subject`: the pointer back to the "user" repository item.
- `contextStrings`, `contextBooleans`, `contextLongs`, `contextDoubles`, `contextDates`: each of these properties is a map of scenario variables of the specified type; for example, `contextBooleans` is a map of boolean context variables which have been set for this scenario instance.

Collective Scenario Instances

In addition to per-user state, a scenario segment has some global, or collective, states—that is, states associated with the segment as a whole. Consider the following example:



Here, the scenario waits for a particular date and time and then sends an e-mail invitation to all the site's female users. The state machine for this scenario looks like this:



Notice that, unlike in the promotion example, there are no user-specific events here. Up until the transition to state 3, where the scenario sends e-mail to all women, there is no mention of users at all. Thus, the scenario server cannot use individual scenario instances to maintain the scenario state while scenario execution moves through states 0, 1, and 2. Instead, it uses a collective scenario instance, which is simply another repository item, of type `collectiveScenario`. The properties of collective instances are a subset of the individual instance properties:

- `id`: the repository ID of this collective scenario instance
- `processName`: the name of the scenario

- `modificationTime`: the time the scenario was last modified (see the next section)
- `segmentName`: the name of the scenario segment
- `creatorId`: the repository ID of the scenario instance which created this instance (see below)
- `state`: the ID of the SSM state this scenario instance is currently in

When a scenario segment is first started, a single collective scenario instance, called the root instance, is always created and placed in the initial SSM state (in the example, state 0). After 1 second goes by, the root instance moves on to the next state, and so on, until the entire scenario state can no longer be represented by the collective instance, and individual instances must be created to hold per-user state. In the promotion example, the collective instance only proceeds as far as state 1 - after a user registers, an individual instance is created for him, and is subsequently responsible for maintaining that user's scenario state. In the invitation example, however, the root instance proceeds as far as state 2; when taking the transition to state 3, it is replaced with individual instances, one for each of the female users.

In addition, collective scenario instances allow for scenarios which apply to anonymous, as well as registered, users.

Note that, because of database query overhead, individual scenario instances are generally more efficient than collective ones in terms of site performance. For more information, refer to Designing Effective Scenarios.

Scenario Initialization

When a scenario is created in the ACC, the corresponding SDL file is created in the scenario registry, and the `updateProcess` method is called on the `/atg/scenario/ScenarioManager` component. At this point, the scenario is registered with the Scenario Manager, and (assuming the scenario has been enabled) it starts running. Scenario registration basically amounts to transforming each scenario segment into an SSM, adding the resultant scenario/SSM information to the Scenario Manager, and updating the Scenario Manager's internal message map to include the new event information.

The message map is used by the Scenario Manager to efficiently determine the following:

- Which scenario segments are waiting for a particular event type
- Where in the segment the scenario instance must be in order to be affected by the event
- Whether the instance must be individual or collective
- What event filter must be satisfied for the event to trigger the transition to the next SSM state

For example, suppose that our two scenario examples (promotion and invitation) are the only two scenarios registered with the server. When a "Registers" event is received by the Scenario Manager, the message map can be used to determine that the only segment waiting for it is the Promotion segment; the only instances affected by the event are the collective scenario instances in state 1. On the other hand, if a "Visits page" event comes in, the message map will reveal that only individual scenario instances in states 4 and 6 of the Promotion SSM are affected by the event; in addition, the event must satisfy the filter "in folder '/products/bikes/'" for the transition to be triggered.

Scenario Event Handling

The Scenarios module distinguishes between the following event types:

- Individual event: an event specific to an individual user, such as “Visits page” or “Views item”
- Global event: an event that applies to all users, such as “Dynamo starts” or “Items requested by slot”
- Individual timer: an event that completes a time wait for an individual scenario instance
- Collective timer: an event that completes a time wait for a collective scenario instance
- Batch timer: an event that completes a time wait for a batch of individual scenario instances

The `ScenarioManager` component is an instance of class `atg.scenario.ScenarioManagerService`, which implements the `atg.dms.patchbay.MessageSink` interface. The Scenario Manager is configured via Patch Bay to receive all standard JMS messages sent out during ATG product operation. Each of the event types listed above comes in on a different input port.

For more information on default scenario events, refer to Using Scenario Events.

For information on how to add your own event elements, refer to Adding Custom Events, Actions, and Conditions to Scenarios.

Workflows

In addition to scenarios, the Scenarios module includes another mechanism for modeling processes, called a **workflow**. Workflows are similar to scenarios, but can be applied to a wider range of processes. The workflow API enables you to create new custom types of workflows that are tailored to specific processes, and which include their own unique actions and tasks. When you edit a workflow in the ACC, the editor is automatically configured to reflect the specific type of workflow you are editing.

Many different types of business processes can be exposed as workflows, with examples ranging from commerce order fulfillment to management of customer support calls. The tools for creating new workflow types are included in the Scenarios module, although the module is not preconfigured with any workflow types. ATG Content Administration provides a default workflow that manages the lifecycle of a publishing project. It might be helpful to examine this workflows as an example.

For more information about workflows, see Creating and Configuring Workflows.

Internal Scenario Manager

You can create and maintain distinct sets of scenarios for internal users, such as customer service representatives, and external users (customers or other external site visitors). Doing so requires the configuration of an internal scenario manager that runs against the internal user profile repository. The internal scenario manager is started by the `DSS.InternalUsers` module, which is provided with the ATG platform and some ATG applications.

Setting up the internal scenario manager is very similar to setting up the scenario manager that is started by the standard DSS module. Unless you specify otherwise, the first server that is started is assumed to be the process editor server. See Configuring the Scenario Manager for more information.

The component that represents the internal scenario manager is `/atg/scenario/InternalScenarioManager`, and the subject for its scenarios is a User item from the internal profile repository The internal scenario manager configuration file is `internalScenarioManager.xml`, which is located by default in `<ATG10dir>\DSS\InternalUsers\config.jar`.

Scenarios that you create while the internal scenario manager is running are stored in the following directory: `<ATG10dir>\home\localconfig\atg\registry\data\internalscenarios`.

Note that the scenario manager and the internal scenario manager listen for message events on different ports. This behavior ensures that events that reference the external user profile repository are sent to the external scenario manager, and events that reference the internal profile repository are sent to the internal scenario manager.

Note also that the ACC is configured by default to point to the external user profile repository. If you want to create scenarios that run against internal user profiles, you must start your application with the `DSS.InternalUsers.ACC` module. This module requires the `DSS.InternalUsers` module.

14 Configuring Scenarios

This chapter describes the steps you must perform to configure scenarios for use in an ATG application. The steps are as follows:

1. Configure the Scenario Manager, the Nucleus component that controls many aspects of scenario behavior in the Scenarios module. As well as defining settings in the Scenario Manager itself, this task includes configuring related components, and for multiple server environments it also includes determining which server will acts as the process editor server.
2. Configure SQL repository caching for the Scenarios module. This task involves setting up Lock Managers for use in the Scenarios module and setting the cache mode for appropriate item descriptors in the `userProfile.xml` file.

Both these steps are required for the Scenarios module to work correctly.

In addition, if you want your ATG application to send e-mail messages to specific recipients, you must set up scenario E-mail Sender components.

This chapter contains the following sections:

In addition, this chapter contains the section Monitoring and Debugging Scenarios, which shows how to log and view scenario-related activities.

Configuring the Scenario Manager

Scenarios are managed and run by the Scenario Manager service. The following sections describe how the Scenario Manager works and explain how to configure it.

Scenario Manager Configuration File

The configuration file `scenarioManager.xml` is the place where information common to all scenario servers is specified. This file uses the Process Manager DTD, located in `<ATG10dir>\DSS\lib\classes.jar`. The `scenarioManager.xml` file defines the configuration of the process editor server, the global server, and individual servers as described below. In addition, it contains the specification for the event, action, and condition registries, including custom elements. The registries are described in more detail in Adding Custom Events, Actions, and Conditions to Scenarios.

Global, Individual, and Process Editor Servers

A cluster of ATG servers must always contain the following:

- exactly one process editor server
- zero or more global scenario servers
- zero or more individual scenario servers

Each individual server handles all the individual, or visitor-specific, events (such as "Visits Page") for the profiles whose sessions are on that server. For example, suppose you have a simple scenario that records all the page visit events to a dataset. When someone visits a page, the corresponding individual server receives the page visit event and performs the scenario action that records the data.

Global scenario servers, in addition to handling the individual events for any of their own sessions, handle global events (such as "Dynamo Starts"), and timer events (such as "Wait 3 hours"). In general, they handle any operations that are not visitor-specific. For example, if you have a scenario that sends e-mail to all your visitors, the scenario action to send the e-mail is executed on a global server.

The process editor server is a specific instance of a global server. In addition to handling global events and the individual events for any of its own sessions, it is also responsible for starting and stopping scenarios. You can create and edit scenarios only in the ACC that is connected to the process editor server.

Important Note About Creating and Viewing Scenarios in the ACC

As stated above, you can create and edit scenarios only in the ACC that is connected to the process editor server. However, your scenarios are not automatically visible in an ACC connected to another server (individual or global). To make them visible, make sure that your standard deployment procedure for scenarios copies your entire configuration (including all scenario XML files) across all scenario servers in your system. If you follow this procedure, scenarios created on the process editor server do appear with read-only access in an ACC connected to another server.

Using the Same Instance of ATG 10.0.2 for More than One Cluster of Servers

You can use the same instance of ATG 10.0.2 for a single cluster of scenario servers, but do not use the same instance for more than one cluster. In other words, you can have a cluster of scenario servers running on one or more machines that all use the same installation of ATG 10.0.2, but additional clusters, including those that you use for production or staging, should be set up to run on separate installations of ATG 10.0.2. This configuration is required because any changes you make to a scenario on a server in one cluster are written to the top-level `localconfig` directory for that installation. Those changes are propagated to all other servers using that installation, including servers in other clusters. This behavior is desirable among servers in a single cluster, but it is undesirable in other situations—for example, between servers in your development and production environments. This caveat also applies to servers that you use to run workflows.

Configuring the Process Editor Server

The default Scenario Manager configuration does not specify a process editor server. The first server that is started is assumed to be the process editor server. This behavior works well in a single-server environment, where one ATG server is responsible for handling all scenario events and actions, both individual and global.

Once you move to a multiple ATG server configuration, however, it is highly recommended that you designate one of your servers as the process editor server. Again, if none is specified, the first server that is started is assumed to be the process editor server. However, designating a server in advance is preferable because it gives you greater control over your ATG environment.

The identity of the process editor server is specified in the Scenario Manager's XML configuration file, which is located in the `/atg/scenario/scenarioManager.xml` file in `<ATG10dir>/DSS/config/config.jar`. (Note that this file uses the Process Manager DTD located in `<ATG10dir>\DSS\lib\classes.jar`.) To override the default server configuration, create a file with the same name and the same path in top-level `localconfig` directory for your cluster of servers. The following shows an example of the file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<process-manager-configuration>

 <process-editor-server>
  <server-name>dyn1:8850</server-name>
 </process-editor-server>

</process-manager-configuration>
```

Specify the name of the ATG process editor server in the `server-name` tag. Use the name returned by the `/atg/dynamo/service/ServerName` component's `serverName()` method (typically, the name is of the form `hostname:drp-port`). For more information about the `atg.service.ServerName` class, see *Referring to Dynamo Servers* in the *Nucleus: Using Nucleus* section of the *ATG Programming Guide*.

Note the following important points about specifying the DRP port number:

- The process editor server must the use the DRP port so that it can be accessed by the `ServerName` component. You do not have to enable DRP (Dynamo Request Protocol) itself; in this case, the Scenarios module just uses it to create a `ServerName` for the process editor server.
- For situations where you have more than one ATG product instance running on the same server, you must specify a unique DRP port for each instance to guarantee that each `hostname:drp-port` combination will be unique. This step is required even if the servers do not use DRP.
- Non-DAS application servers, such as Oracle WebLogic, do not use DRP. However, even if you are running the Scenarios module on a non-DAS application server, you must specify a DRP port so that the Scenarios module can create a unique name for the process editor server. For more information, refer to the *ATG Installation and Configuration Guide*.

To verify that your process editor server has been configured correctly, start both the process editor server and all other global and individual scenario servers (see below). As each server starts up, an appropriate message appears in the ATG console window; for example, “Initializing process editor server dyn1:8850,” “Initializing global scenario server dyn2:8850,” or “Initializing individual scenario server dyn3:8850.” Make sure that there is exactly one process editor server running and that all other scenario servers are designated as global or individual.

See The Scenario Registry and Scenario Definition Files for information on changing the process editor server from one server to another.

Configuring Global Scenario Servers

In a multiple ATG server configuration, it is recommended that you have at least one global server in addition to the process editor server (see above).

You define the identity of the global server in the `scenarioManager.xml` file that you created in your `localconfig` directory and used to configure the process editor server (see above). The following shows an example of the file with the addition of the `<global-server>` tag. Here, two servers, `dyn2:8850` and `dyn8:8850`, have been defined as global.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE process-manager-configuration
        PUBLIC "-//Art Technology Group, Inc.//DTD Process Manager//EN"
        'http://www.atg.com/dtds/processmanager/processmanager_1.0.dtd'>

<process-manager-configuration>

 <process-editor-server>
  <server-name>dyn1:8850</server-name>
 </process-editor-server>

 <global-server>
  <server-name>dyn2:8850</server-name>
 </global-server>

 <global-server>
  <server-name>dyn8:8850</server-name>
 </global-server>

</process-manager-configuration>
```

Specify the name of the global ATG server in the `server-name` element within the `global-server` tag. Use the name returned by the `/atg/dynamo/service/ServerName` component’s `serverName()` method (typically, the name is of the form `hostname:drp-port`). For more information about the `atg.service.ServerName` class, see *Referring to Dynamo Servers* in the *Using Nucleus* section of the *ATG Programming Guide*.

You can define as many global servers as you require.

Note: Because running global scenario servers places an additional burden on your ATG server, consider configuring global servers to not accept any user sessions. Do this by setting the `drpEnabled` property to `false` in the server's `/atg/dynamo/Configuration` component. Specifically, this may be a good idea if you anticipate creating a lot of scenarios that will run primarily on global servers – for example, scenarios that send e-mail to a large number of visitors at a predetermined time. For more information on session management, refer to the *ATG Installation and Configuration Guide*.

Configuring Individual Scenario Servers

No additional steps are necessary. By default, all servers handle individual scenario events and actions; therefore, any server that is not specified as either the process editor server or a global server is automatically an individual server.

The Scenario Registry and Scenario Definition Files

When you create scenarios in the ACC, the system writes the scenarios out to definition files in the scenario registry, which exists on the process editor server. The definition files are in XML format and have the extension `.sdl`. The process editor server reads these scenario definitions at startup and whenever you modify any scenario through the ACC.

The scenario registry is located at `/atg/registry/data/scenarios` in your config path. (It may be located across several config path layers depending on how you package your applications; for example, the scenarios that come with the Quincy Funds demo are located in the Quincy Funds config layer.) By default, however, any scenarios that you create or edit in the ACC are saved as SDL files in the `localconfig` layer. If you want the ACC to write SDL files to a layer other than `localconfig`, you must manually edit the `defaultForUpdates` property in the `/CONFIG.properties` files in each config layer. Set the property to `false` in the `localconfig` layer and to `true` in the layer where you want the files to be written.

Under normal circumstances, you do not have to edit the `.sdl` files in any way. However, if you change your process editor server from one ATG server to another, you must copy the entire scenario registry from the old process editor server to the new one. (If you do not perform this step, the new process editor server will not have access to the scenario definition files. Consequently it will be unable to recognize any existing scenarios and will disable any that may be currently running.)

Configuring the ScenarioManager Component

The main Nucleus component responsible for scenario operations on each scenario server is `/atg/scenario/ScenarioManager`. The following table lists the properties of the `ScenarioManager` component:

Property	Description
`$class`	Class name Default: `atg.scenario.ScenarioManagerService`

`configurationFile`	The location of the scenario configuration XML file Default: `/atg/scenario/scenarioManager.xml`
`editOnlyMode`	Specifies that this scenario server should not run any scenarios or process any events. This property is ignored if it is set on the `ScenarioManager` component of a process editor server. To disable the component entirely, see Disabling the Scenario Manager Component.
`enabled`	Determines whether the component will start along with a server. See Disabling the Scenario Manager Component.
`globalConfigurationFile`	The location of the Dynamo Messaging System file that contains `message-source` and `message-sink` definitions for global events. Default: `/atg/dynamo/messaging/dynamoMessagingSystemDSSGlobal.xml`
`globalServer`	Read-only flag that indicates whether this scenario server is a global server (see Global, Individual, and Process Editor Servers). Default: `false`. Set to `true` at runtime for global servers and for the process editor server.
`loggingDebug`	Writes debug information about scenarios to the `debug.log` file and to the ATG console window. Default: `false` For more information on this property and the other debugging properties in this component, see Monitoring and Debugging Scenarios.
`loudMissingElementMessages`	Displays an informative message rather than a long series of errors in cases where a server attempts to run scenario created by another server. See Monitoring and Debugging Scenarios for more information.

`maxBatchSize`	The maximum batch size to use when performing batch operations on scenario instances. Batch operations come into play when you perform scenario actions on a large number of profiles (for example, sending an e-mail message to a large group of users). It may not be possible to perform the entire operation in a single transaction because databases have a fixed size for their transaction logs. Thus, the action is executed in batches over multiple transactions. The batch size needs to be large enough to minimize the overhead of independent queries, but small enough so that you do not fill up the transaction log when processing changes for each of the items in the batch. See also Minimizing the Number of Collective Elements. Default: `1000`
`maxMessageDeliveryAttempts`	The maximum number of times the system attempts to deliver the same message if an error occurs during delivery. Set to `-1` to have the system try an unlimited number of times. Default: `1`
`messageRegistryComponentName`	The Nucleus path of the `MessagingManager` component that is used to obtain information about DMS messages Default: `/atg/dynamo/messaging/MessagingManager`
`subjectRepository`	The Nucleus path of the profile repository where profile and scenario information is stored Default: `/atg/userprofiling/ProfileAdapterRepository`
`processEditorServer`	Read-only flag that indicates whether this server is the process editor server (see Global, Individual, and Process Editor Servers). Default: `false`. Set to `true` at runtime for the process editor server.
`processRegistry`	The Nucleus path of the registry that stores scenario definitions Default: `/atg/registry/Scenarios`
`PDLParser`	The Nucleus path of the `SDLParser` component, which interprets the scenario definitions Default: `/atg/scenario/SDLParser`
`serverName`	The Nucleus path of the `ServerName` component that provides this ATG server's name Default: `/atg/dynamo/service/ServerName`
`transactionManager`	The Nucleus path of the `TransactionManager` component Default: `/atg/dynamo/transaction/TransactionManager`

useEventRegistry	Determines whether the Scenarios module gets event information from the scenario event registry (in the scenarioManager.xml file) or the Dynamo Messaging System registry. If true, the scenario event registry is used. This property exists to allow compatibility with versions of ATG products before 6.0.0. For more information, refer to Using Scenario Events. Default: False

In addition, the following properties of ScenarioManager specify the item descriptor and property names that are used to store scenario-related information in the profile repository. These must correspond to the names specified in the repository template. Do not change them unless you make the corresponding changes in the repository template as well.

Property	Description
subjectIdProperty	Name of the profile repository property that contains the item's repository ID. Default: id
subjectProcessInstancesProperty	Name of the profile repository item's property that is used to store the set of scenario instances associated with the profile. Default: scenarioInstances
subjectType	Name of the repository item descriptor that describes a user profile. Default: user

Configuring the SDLParser Component

As well as the ScenarioManager component, configure the component /atg/scenario/SDLParser, which the system uses to interpret the scenario definition files produced when you create scenarios in the ACC.

This component, of class atg.scenario.definition.SDLParser, has the following key property:

Property	Description
XMLToolsFactory	The Nucleus path of the XMLToolsFactory component used to parse the SDL files. Default: /atg/dynamo/service/xml/XMLToolsFactory

Configuring SQL Repository Caching for Scenarios

The default cache mode for all scenario item descriptors is `simple`. This setting is ideal for a single ATG server environment, but if your installation is running more than one ATG server, you must set the SQL repository cache mode to `locked` for the following item descriptors:

- `individualScenario`
- `collectiveScenario`
- `scenarioInfo`
- `scenarioTemplateInfo`
- `scenarioMigrationInfo`
- `individualScenarioTransition`
- `collectiveScenarioTransition`

The cache mode is set to locked automatically in the Scenario module's `liveconfig` configuration layer, so if you are deploying to a live site, you can simply add the `liveconfig` configuration layer to the environment for all your ATG servers to guarantee that the cache mode will be set correctly for these item descriptors. See the *Configuring for Production* chapter in the *ATG Installation and Configuration Guide* for information about the `liveconfig` configuration layer.

However, if you are working in a non-production, multiple-server environment that does not include the `liveconfig` configuration, you must set the cache mode manually as described below. If you do not set the cache mode to locked for these item descriptors, several scenario servers could attempt to process the same event, and serious errors will occur.

To complete this step manually, make the following changes to `/atg/userprofiling/userProfile.xml`:

```
--- localconfig/atg/userprofiling/userProfile.xml:
<gsa-template>
    <item-descriptor name="individualScenario" cache-mode="locked"/>
    <item-descriptor name="collectiveScenario" cache-mode="locked"/>
    <item-descriptor name="scenarioInfo" cache-mode="locked"/>
    <item-descriptor name="scenarioTemplateInfo" cache-mode="locked"/>
    <item-descriptor name="scenarioMigrationInfo" cache-mode="locked"/>
    <item-descriptor name="individualScenarioTransition"
        cache-mode="locked"/>
    <item-descriptor name="collectiveScenarioTransition"
        cache-mode="locked"/>
</gsa-template>
```

In addition, in order for locked mode caching to work in a multiple server environment, you must configure the following Lock Manager components, as described in the *Enabling the Repository Cache Lock Managers* section of the *ATG Installation and Configuration Guide*.

- `/atg/dynamo/service/ServerLockManager`
- `/atg/dynamo/service/ClientLockManager`

Be sure to set the useLockServer property of each ClientLockManager to true. (This property is set to true automatically in the liveconfig layer, but if your environment does not include liveconfig, you must set the property manually.)

In addition, verify that the lockManager property in the atg/userprofiling/ProfileAdapterRepository component is set as follows:

```
lockManager=atg/dynamo/service/ClientLockManager
```

Scenario Caching with Session Federation

If your implementation uses session federation (see *Session Federation Overview* in the *ATG Installation and Configuration Guide*) consider the following situation: a site visitor triggers a scenario by logging into server A and then follows a link to server B. If session federation is enabled, the person will have a profile object on both servers. If another scenario event (for example, a timer event) occurs, both servers will try to advance the visitor to the next state in a scenario state machine. This behavior could cause conflict or unexpected results.

If you use session federation on servers that process scenarios, you can prevent this type of conflict by setting the SQL repository cache mode to locked for all scenario tables. See the previous section, Configuring SQL Repository Caching for the Scenarios Module, for more information.

Setting Up Scenario E-mail Sender Components

As part of creating scenarios in the ACC, you can include Send Email elements that tell the system to send e-mail messages to specific recipients (see SendEmail Action). To send these messages, the Scenarios module uses two TemplateEmailSender components, which you can configure as part of setting up your ATG site:

- The IndividualEmailSender component handles messages that you send to individual site visitors (for example, a message that you send in response to a customer placing an order). This component is located by default at /atg/scenario/IndividualEmailSender.
- The CollectiveEmailSender component handles messages that you send to groups of visitors (for example, a message that you send to a group of qualified recipients inviting them to register for a seminar). By default, this component is a GenericReference whose componentPath property is set to /atg/userprofiling/email/TemplateEmailSender. You can, however, configure it as a separate e-mail sender component if required.

  This component is located by default at /atg/scenario/CollectiveEmailSender.

  For more information, see the targeted e-mail chapters of this guide.

Setting Up TemplateEmailInfo Objects for Scenarios

In addition to configuring the two components described above, you must also create an instance of the class atg.userprofiling.email.TemplateEmailInfoImpl for every e-mail that you send through a scenario. The Scenarios module contains a default instance of this class, /atg/scenario/DefaultTemplateEmailInfo, that you can use as a model. Note that, when you specify the URL of the e-mail template to use for the mailing, you do not need to specify the context root; the SendEmail scenario action (see SendEmail Action) is designed to ensure that the template URL is correctly formed.

For more information on the TemplateEmailInfoImpl class, refer to Creating Targeted E-mail in the Personalization section of this guide.

Configuring BatchEmailListener and EmailListenerQueue Components

Note also that, when you set up your production site, it is recommended that you also configure and use ATG's BatchEmailListener and EmailListenerQueue components, which provide optimal performance for e-mail handling. These components are described in the *Email Sender* chapter of the *ATG Programming Guide*.

Monitoring and Debugging Scenarios

To help you monitor scenarios and debug them if necessary, enable the loggingDebug property in the /atg/scenario/ScenarioManager component (set to false by default). If you set it to true, the system logs information such as the events each scenario is capturing, the condition elements that are being satisfied, and the action elements that are triggered. The information is stored in the debug.log file and it also appears in the ATG console window.

Note that you might not want to keep this setting enabled for long periods of time, especially for live sites, because tracking information on each scenario can affect performance.

The ScenarioManager component contains additional properties you can use for different levels of scenario debugging as described below:

- debugProcessNames

  Allows you to debug specific scenarios rather than all of them. Set this property to the folder path of the scenario or scenarios that you want to debug, minus the part of the registry where all scenarios are located. You can specify individual scenarios or all scenarios in a given folder. For example:

  ```
  debugProcessNames=/DSSDemo/InvestorRelations/MyScenario
  ```

  logs information for the specified scenario. Note that the path comes from the location of this scenario's SDL file under the /atg/registry/data/scenarios directory. You can determine the path by looking for the scenario in the ACC using the By Folders view.

  ```
  debugProcessNames=/scenario1, /scenario2
  ```

  logs information only for scenarios called scenario1.sdl and scenario2.sdl

  ```
  debugProcessNames=/recorders/*,/scenario1
  ```

logs information for all scenarios in folder /recorders/, plus /scenario1.sdl.

The default setting is *, which logs information on all scenarios. Note that this property is available only if both loggingDebug and loggingDebugProcesses are enabled.

- loggingDebugMessages

  Includes in the debug log file information about the Java Message Service (JMS) messages sent and received by the Scenario Manager.

  The default setting is true.

- loggingDebugProcesses

  Includes additional scenario information in the debug log file such as the time that individual instances are created, transitions between states, and filter checking.

  The default setting is false.

- loggingDebugUpdates

  Includes information about scenario updates in the debug log file.

  The default setting is true.

- loudMissingElementMessages

  In some circumstances, a server may read scenario definitions generated by another server, attempt to run them, and be unable to do so because the required elements do not exist in its configuration path. For example, an ATG Service agent server could detect and attempt to run the campaign (scenario) definitions that ATG Outreach has deployed to a production server. In this case, a long series of errors about unknown elements is sent to the console when the agent server is started.

  Setting the loudMissingElementMessages property to false displays an informative message rather than the errors. Note that it is set to false in the DSS liveconfig layer.

For more information on configuring the ScenarioManager component, see Configuring Scenarios.

As an alternative to editing the ScenarioManager component directly, you can enable many of these debugging properties through the Configure Server Debugging dialog box in the ACC. To access this dialog box, display the Scenarios > Scenarios window, and then select Tools > Configure Server Debugging. This dialog box contains the following options:

- **Include debugging information in server log**: Enables the loggingDebug property.
- **Log JMS messages received by process manager**: Enables the loggingDebugMessages property.
- **Log information related to process updates**: Enables the loggingDebugUpdates property.
- **Enable logging of information about individual processes**: Enables the loggingDebugProcesses property.

Viewing Scenario Information in the ATG Dynamo Server Admin Page

In addition, you can use the Component Browser in the ATG Dynamo Server Admin page to see information about your scenarios. The main Nucleus component responsible for scenario operations is located at `/atg/scenario/ScenarioManager`. To examine the scenarios handled by this service, point your Web browser to the ATG Dynamo Server Admin page at:

```
http://localhost:port/dyn/admin/nucleus/atg/scenario/ScenarioManager/
```

(The default port numbers on JBoss, Oracle WebLogic, and IBM WebSphere are 8080, 7001, and 9080, respectively. For more information, see *Connecting to the Dynamo Administration UI* in the *ATG Installation and Configuration Guide*.)

The Examine Scenarios table shows, for each scenario, its status; last modification time; number of collective and individual instances; and debugging flags. A scenario’s status is displayed as “running” if it is enabled or as “disabled” if it is disabled.

Click the **List instances** link in the table to view, for the given scenario, the repository items that maintain the scenario state. Individual scenario instances maintain the scenario state for individual visitor profiles, and collective scenario instances maintain the state for the scenario as a whole. The following shows an example of the information you see for each scenario:

```
Scenario /QuincyFunds/InvestorRetention/WelcomeMail.sdl:

  Scenario info:

    1000020: class atg.adapter.gsa.GSAItem, item=scenarioInfo[ id=1000020,
    creationTime=960937282000, pdl=[B@683bfc, stateMachineVersion=2,
    author=admin, processStatus=2, id=1000020,
    processName=/QuincyFunds/InvestorRetention/WelcomeMail.sdl, lastModifiedBy=,
    modificationTime=964051587174]

Segment New Members:

Collective scenario instances:

    1.1000009: class atg.adapter.gsa.GSAItem, item=collectiveScenario[ id=1000009,
    segmentName=New Members, state=2, id=1000009,
    processName=/QuincyFunds/InvestorRetention/WelcomeMail.sdl,
    modificationTime=964051587174, creatorId=null]

Individual scenario instances:
    1.11000015: class atg.adapter.gsa.GSAItem, item=individualScenario
    [id=11000015, segmentName=New Members, contextLongs={},
    contextDates={}, state=3, createdByRecurringEvent=false, subject=(reference
    to item=user:210001), id=11000015,
    processName=/QuincyFunds/InvestorRetention/WelcomeMail.sdl,
    contextBooleans={}, modificationTime=964051587174, contextDoubles={},
    contextStrings={}, creatorId=1000009]
```

Click the name of the scenario in the table to view an internal text-based representation of the elements in the given scenario. You might find this version helpful if you need to include a description of a scenario in an internal bug report, for example. The following shows the scenario definition information for the New Members segment in the `WelcomeMail` scenario:

```
Scenario /QuincyFunds/InvestorRetention/WelcomeMail.sdl:

Process enabled=true modificationTime=1028236820204

     Segment New Members:
       0s: branch
         1: event atg.dps.Register
         2: time-wait in 5 mins
         3: condition (eq subject:investors true)
         4: condition (eq subject:receiveEmail yes)
         5: condition (eq subject:locale en_US)
         6: condition (isNotNull subject:email)
         7: action sendEmail[template=/QuincyFunds/en/email/welcome.jsp]
         8: fork
           8.1: branch
             8.1.1: event atg.dps.Login
           8.2: branch
             8.2.1: time-wait in 4 weeks 2 days
             8.2.2: action
              sendEmail[template=/QuincyFunds/en/email/newmemberoffer.jsp]
```

The link also shows you the possible states for people passing through each scenario (see Scenario Basics for more information about scenario states). The following example shows the scenario states for the New Members segment of the `WelcomeMail` scenario:

```
StateMachine New Members:
  1[start]
    (1) in 1 secs --> 2[1]:

  2[1]
    (1) atg.dps.Register --> 3[2]:

  3[2]
    (1) in 5 mins --> 4[8.2.1((eq subject:investors true)&(eq subject:receiveEmail
         yes)&(eq subject:locale en_US)&(isNotNull subject:email)),8.1.1((eq
         subject:investors true)&(eq subject:receiveEmail yes)&(eq subject:locale
         en_US)&(isNotNull subject:email))]:
         sendEmail[template=/QuincyFunds/en/email/welcome.jsp]((eq
         subject:investors true)&(eq subject:receiveEmail yes)&(eq subject:locale
         en_US)&(isNotNull subject:email))

  4[8.2.1((eq subject:investors true)&(eq subject:receiveEmail yes)&(eq
    subject:locale en_US)&(isNotNull subject:email)),8.1.1((eq subject:investors
    true)&(eq subject:receiveEmail yes)&(eq subject:locale en_US)&(isNotNull
    subject:email))]
    (1) ((eq subject:investors true)&(eq subject:receiveEmail yes)&(eq
        subject:locale en_US)&(isNotNull subject:email)) --> 5[8.2.1,8.1.1]:

  5[8.2.1,8.1.1]
    (1) atg.dps.Login --> 6[8++]:
    (2) in 4 weeks 2 days --> 6[8++]:
         sendEmail[template=/QuincyFunds/en/email/newmemberoffer.jsp]

  6[8++]
```

Setting the Web Application Context Root for Scenarios

The context root for a J2EE Web application is set in the application's `application.xml` file. In some situations you may have more than one Web application deployed on the same server, which then requires areas of ATG functionality such as the Scenarios module to be aware of more than one context root. (Specifically, some scenario event and action elements contain references to pages or e-mail templates that may be stored in multiple context roots.) To manage context roots in multiple Web applications, the system maintains a registry called a `WebAppRegistry`.

The base class for this registry is `atg.service.webappregistry.WebAppRegistry`. Different implementations of this class supply the settings that the registry requires. Two key implementations are the `StaticWebAppRegistry` and the `ServletContextWebAppRegistry`, which are described below. For information on other implementations, refer to the *ATG API Reference*.

Note that, as described in this section, the document picker in the ACC scenario editor uses the Web application name (the value of the `<display-name>` tag in the `web.xml` file) to resolve the context root for any documents that you add to scenarios. For this reason, you cannot add documents (for example, email templates) from unnamed Web applications. These applications are shown in gray in the scenario editor's document picker, and they cannot be selected.

StaticWebAppRegistry

The `atg/registry/webappregistry/StaticWebAppRegistry` component (class `atg.service.webappregistry.StaticWebAppRegistry`) configures the registry of deployed Web applications by reading a Map of `display-name=context-root` mappings or a list of Web application components from its properties file.

To use the `display-name=context-root` mappings to supply Web application information, simply set the `contextRootMap` property inside the `atg/registry/webappregistry/StaticWebAppRegistry` properties file. Example:

```
contextRootMap=\ Quincy Funds J2EE DAF Demo=QuincyFunds,
     \ MotorpriseJSP=Motorprise
```

You set the ContextRootMap value to the value of the <display-name> and <context-root> tags in the application's web.xml file, usually located in the WEB-INF directory (for example, the web.xml file for Quincy Funds is located at <ATG10dir>\DSSJ2EEDemo\j2ee-apps\QuincyFunds\web-app\WEB-INF).

To use the list of Web application components to configure the registry, you create a separate component of class atg.service.webappregistry.WebApp for each application, and then you list the components in the preConfiguredWebApps property in the StaticWebAppRegistry properties file, as shown here:

```
preConfiguredWebApps=\/atg/registry/webappregistry/QuincyFundsWebApp,
      \ /atg/registry/webappregistry/MotorpriseWebApp
```

The following example shows the properties file for the /atg/registry/webappregistry/QuincyFundsWebApp component:

```
QuincyFundsWebApp.properties

$class=atg.service.webappregistry.WebApp

properties=\ display-name=QuincyFunds,\ appState=started,\
context-root=QuincyFunds,\ web-uri=web_app,\
path=d:/work/5.6/Dynamo/DSSJ2EEDemo/j2ee-apps/QuincyFunds
```

ServletContextWebAppRegistry

The atg/registry/webappregistry/ServletContextWebAppRegistry component (class atg.service.webappregistry.ServletContextWebAppRegistry) registers Web applications defined by a ServletContext. It is used by NucleusServlet (class atg.nucleus.servlet.NucleusServlet) to register Dynamo-specific Web applications, which it does by reading information from the Web applications started at runtime. For more information, refer to the *ATG Repository Guide*.

Updating the Context Root for Scenarios

Occasionally it may be necessary to change an application's context root after you have already created and enabled scenarios for that application. You make the change by adjusting the context root setting in the application.xml file for the given application. However, the WebAppRegistry described in the previous section is not updated automatically, so the Scenarios module is not aware of any change you make to this file. To avoid scenario execution errors, therefore, note that you must either manually update the WebAppRegistry or manually update the context root value in the application's web.xml file.

The following example shows the web.xml file setting for the Quincy Funds demo:

```
<context-param>
  <param-name>context-root</param-name>
  <param-value>QuincyFunds</param-value>
</context-param>
```

Disabling the Scenario Manager Component

To disable the scenario manager component for a server:

1. Create or edit the `ScenarioManager.properties` file for the server.

   ```
   <ATG10dir>/home/servers/servername/localconfig/
   atg/scenario/ScenarioManager.properties
   ```

   If you are disabling the InternalScenarioManager component for an agent server, make this configuration in the `InternalScenarioManager.properties` file.

2. Set the `enabled` property to `false`. If the enabled property is not present in the file, add it.

   ```
   enabled=false
   ```

3. Restart the server.

Verify that the server has started with the scenario manager component disabled by checking the server's log file for the following message.

```
2010-08-20 13:34:01,290 INFO  [nucleusNamespace.atg.scenario.ScenarioManager]
(main) This ProcessManager is disabled and will not execute processes or handle
events
```

15 Setting Up Security Access for Scenarios

As you can for any product in the ATG suite, you can grant or deny access to the features of the Scenarios module by displaying or hiding menu items in the main ACC window. You use the People and Organizations > Control Center Groups option to configure access to menu items. For more information, see the *ATG Programming Guide*.

For those groups of users who do have access to the Scenarios module menu items, you can set a more granular layer of security by controlling access to scenario folders, individual scenarios, and scenario templates. For example, suppose your company sets up scenarios that deliver separate sets of promotions to customers in your East and West regions. You decide that you want business users to be able to edit scenarios for their own region only. You set up two scenarios folders called East and West and copy scenarios into their appropriate folder. Then you define access rights for each folder.

In addition, you can grant or deny access to individual repository items or to individual properties of a repository item. Note, however, that you cannot do this through the ACC; you must do it by defining access control lists (ACLs) manually as described in *Secured Repositories* in the *ATG Repository Guide*.

This chapter describes how to use the ACC to set up access control for scenario folders, scenarios, and scenario templates. It includes the following sections:

Using the ACC to Set Scenario Access Rights

As noted above, you can use the ACC to set access rights to scenarios and scenario templates. To see the access control lists (ACLs) that are created through the ACC, look in `<ATG10dir>/home/localconfig/registry/data/` and locate the appropriate folder and `.sec` file (or `.tsc` file for scenario templates). Open the file to view the ACL.

The following figure shows a sample .sec file for a scenario:

```
acl=Admin$user$design:list;
Admin$user$developer:list,write_owner;
Admin$user$manager:read_acl;
```

For more information on ACLs, refer to *Secured Repositories* in the *ATG Repository Guide*.

Note that you can view only ACL information for Scenarios module features through the secure registry. To view the repository- based ACLs, look at the secure repository definition file (by default, `secured-test-repository.xml`, described in the *ATG Repository Guide*) and locate the `acl-property` for a given item. The ACL for that item is stored in the table and column listed in the `acl-property`.

Defining Access Control for Scenario Folders

The following procedure shows how to set access rights to scenario folders.

1. Start the ACC and select Scenarios > Scenarios.
2. Select the scenario folder for which you want to define access.
3. Select File > Set Access Rights On Folder. The Set Access Rights dialog box appears.

   The checkboxes show the different levels of access rights you can grant or remove:

   - List: Controls whether a this folder will appear as the result of a query. Note that, even if a user does not have List access, he or she could still access this folder by requesting it specifically.
   - Read: Controls the ability to view (but not add to or edit) this folder and its contents.
   - Write: Controls the ability to add or edit scenarios in this folder.
   - Delete: Controls the ability to remove this folder and its contents from the repository. This access right corresponds to the DESTROY access right in the Access Control List. For more information, see *Secured Repositories* in the *ATG Repository Guide*.
   - View owner: Controls the ability to view the name of the person who owns this folder, for example in the Author column of the Scenarios > Scenario window. Corresponds to READ-OWNER in the Access Control List.
   - Set owner: Controls the ability to change the owner of this folder. The owner access right exists essentially as a way of identifying the person or group who created the folder. Corresponds to WRITE-OWNER in the Access Control List.
   - View access rights: Controls the ability to view the access control list for this folder. (This access right is automatically granted to the owner of the folder.) Corresponds to READ-ACL in the Access Control List.
   - Set access rights: Controls the ability to change the access rights for this folder. (This access right is automatically granted to the owner of the folder.) Corresponds to WRITE-ACL in the Access Control List.

The list on the left shows the people for whom you can define access rights. The list can include individual users, organizations, roles, or profile groups. To add new items to this list, click Add, and select the items that represent the people for whom you want to define access. Remember that a member of an organization or a role inherits its access rights.

4. Use the checkboxes to grant or deny appropriate access rights for each user, organization, role, or profile group in the list on the left.

Important: Bear in mind a user's access rights for content repositories as well as his or her access rights for scenario folders. For example, you might give a user write access to a specific scenario folder, but does he or she also need access to one or more content repositories? To create slot elements, for example, a user may need read access to a repository that stores image files. Make sure that users have access to all required content, and in particular, check that their access rights for different repositories do not conflict.

Defining Access Control for a Scenario

This procedure is similar to the previous one, except that the access rights you define here apply to a specific scenario rather than to a scenario folder.

1. In the Scenarios window, select the name of the scenario for which you want to define access. Note: Do not display the scenario itself; just highlight its row in the scenario list. The easiest way to do this is to click in any column of data for the scenario except the name column.
2. Display the right-click menu and select Set Access Rights On Scenario. The Set Access Rights dialog box appears. (For scenario templates, the name of the menu item changes appropriately.)
3. Use the checkboxes to grant or deny appropriate access rights for each user, organization, role, or profile group in the list on the left. For more information, see Defining Access Control for Scenario Folders.

Important: Bear in mind a user's access rights for content repositories as well as his or her access rights for scenarios. For example, you might give a user write access to a specific scenario, but does he or she also need access to one or more content repositories? To create slot elements, for example, a user may need read access to a repository that stores image files. Make sure that users have access to all required content, and in particular, check that their access rights for different repositories do not conflict.

You can set access control for individual scenario templates in the same way that you can control access to individual scenarios and scenario folders.

Making a Scenario Read Only

In addition to using the security measures described in this chapter to control access to scenarios, you can make individual scenarios read only. Any user who has security access that allows him or her to open the scenario will be able to do so but will not be able to make any changes to its contents. However, scenarios marked as read only can still be enabled, disabled, or deleted by any user.

To make a scenario read only, perform the following steps:

1. Open the `.sdl` file for the scenario you want to make read only. These files are usually located in `<ATG10dir>\home\localconfig\atg\registry\data\scenarios\DSS`.
2. Add the following attribute to the `<process>` element:

   ```
   readonly="true"
   ```

   For example:

   ```
   <process author="admin" creation-time="1070401076069" enabled="false"
   last-modified-by="admin" modification-time="1070401076159"
   readonly="true">
   ```

Note that you may need to restart the server to have this change take effect.

16 Designing Effective Scenarios

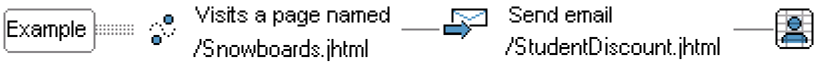
Scenarios are a powerful personalization tool, and they have been designed to give you as much flexibility as possible in the way you set them up. However, the performance that you get from your scenarios depends greatly on their design; a scenario that appears to cause a site page to load slowly, for example, can perform much better if you substitute or rearrange the elements that it contains.

This chapter makes some recommendations about the best way to position scenario elements for optimal performance.

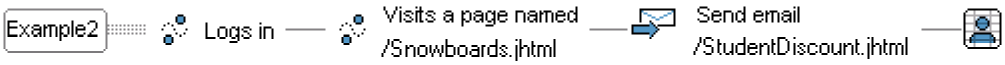
Excluding Anonymous Visitors

Scenarios that apply to all site visitors require changes to every profile in the profile repository, and this behavior can be time consuming, especially if a site attracts a large number of visitors. If your scenario does not apply to anonymous visitors, include a Logs In element at the beginning of the scenario so that only those people who log in as members will be included.

Replace this scenario:

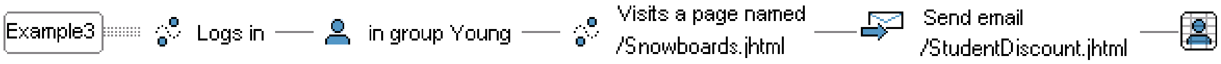

Example
Visits a page named /Snowboards.jhtml
Send email /StudentDiscount.jhtml

With this one:


Example2
Logs in
Visits a page named /Snowboards.jhtml
Send email /StudentDiscount.jhtml

Minimizing the Number of Visitors Included

Whether a scenario includes anonymous visitors or not, it is always best to minimize the number of visitors to whom the scenario applies, especially if you have a large number of site visitors. Consider the following example:


Example3
Logs in
in group Young
Visits a page named /Snowboards.jhtml
Send email /StudentDiscount.jhtml

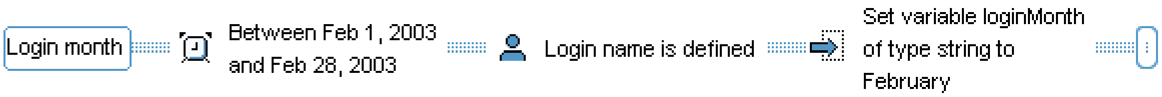
Here, the e-mail message you want to send applies only to visitors who are students. You estimate that almost all the people who are likely to respond to the mailing also belong to a profile group called Young, which you have set up for anyone under 35 years of age. Therefore, rather than having the scenario apply to anyone who logs in and visits the /Snowboards page, you can safely narrow its focus to people who belong to the profile group Young, thereby reducing the number of profiles that the Scenarios module must change as part of processing this scenario.

Bear in mind, however, that using profile groups to minimize the number of scenario participants does not always make a scenario perform more efficiently. Depending on their position in a scenario, profile group elements ("People in group X") can behave as collective elements, and in this case they can have a negative effect on performance. See the next section, Minimizing the Number of Collective Elements, for more information.

Minimizing the Number of Collective Elements

Collective elements apply to all users in a scenario, and they require the Scenarios module to generate line-by-line queries of the `dps_user` and `dps_user_scenario` tables. These queries can run slowly if the number of profiles is high.

Consider the following example:


Login month
Between Feb 1, 2003 and Feb 28, 2003
Login name is defined
Set variable loginMonth of type string to February

This scenario generates a query similar to the following:

```
SELECT DISTINCT t1.id,t1.id
FROM dps_user t1
WHERE ((t1.login IS NOT NULL)
AND NOT EXISTS (SELECT * FROM dps_user_scenario
WHERE id = t1.id AND ind_scenario_id IN
( SELECT tt1.id FROM dss_ind_scenario tt1
WHERE (tt1.creator_id = '4000001'))))
ORDER BY t1.id ASC;
```

The query searches the `dps_user` table for profiles that have a defined login name, and it also searches the `dps_user_scenario` table to make sure none of those profiles already has a scenario instance created for this scenario. As more profiles are found that match the query, more scenario instances are created, which in turn adds more lines to the `dps_user_scenario` table. For sites with an unusually large number of profiles, this process can eventually become very slow.

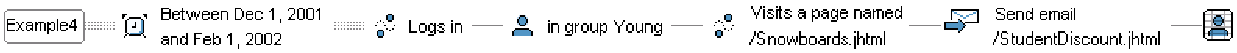
You can improve performance by checking that the `maxBatchSize` setting in the Scenario Manager component is set to at least the default value, 1000, and adjusting it if necessary A higher setting enables the Scenarios module to process more profiles with each iteration of the query.

To examine the profile queries that your scenarios generate, turn on the `loggingDebug` property for the `/atg/userprofiling/ProfileAdapterRepository` component. The queries appear in the `<ATG10dir>\home\logs\debug.log` file.

Avoiding Scenarios that Run Indefinitely

Scenarios are not restricted to one session only and will not stop simply because a visitor logs out or otherwise ends her session. In the example shown above, if a visitor logs in but never displays the /Snowboards page, information about her scenario state will remain indefinitely in the profile database.

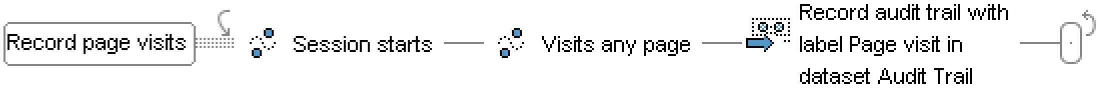
To avoid this behavior, use Time elements to limit the amount of time for which a scenario is active. For example, you could write this scenario as follows:

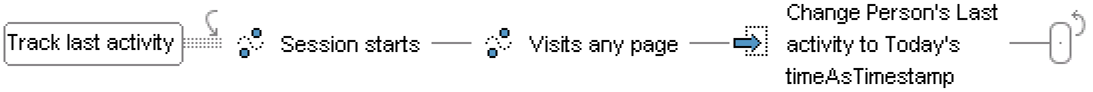

Example4
Between Dec 1, 2001 and Feb 1, 2002
Logs in
in group Young
Visits a page named /Snowboards.jhtml
Send email /StudentDiscount.jhtml

In general, limit a scenario’s lifespan wherever possible.

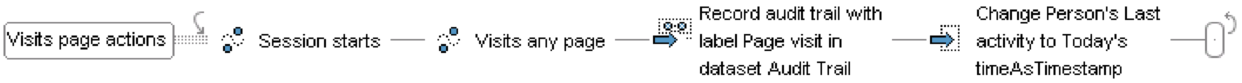
Combining Scenarios Wherever Practical

As expected, the number of scenarios you have enabled affects the performance of your sites. However, it is important to note that performance is affected less by the total number of scenarios than by the number of times a particular Event type occurs across all the scenarios you have defined. For example, if you have 20 scenarios, each of which includes a Visits page element, you may notice that pages load more slowly because the site must process 20 Visits page elements whenever anyone displays a page. To improve performance, create a single scenario that contains one Visits page event followed by several actions. The following images illustrate this idea:


Record page visits
Session starts
Visits any page
Record audit trail with label Page visit in dataset Audit Trail



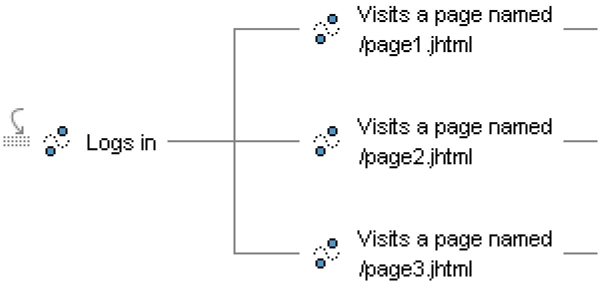
Consider combining these two segments into a single segment, as follows:


Visits page actions
Session starts
Visits any page
Record audit trail with label Page visit in dataset Audit Trail
Change Person's Last activity to Today's timeAsTimestamp

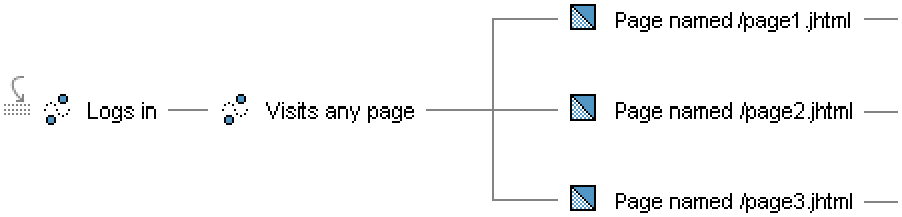
Minimizing the Number of Paths through a Fork

As described in the overview chapter, the Scenarios module creates transient states for every visitor passing through a scenario and stores information about those states in the profile repository. In the case of scenarios that contain fork elements, the Scenarios module creates a state for every possible path a visitor could follow through a fork. If you have a large number of site visitors, and you create fork elements that present many possible paths, out-of-memory errors can occasionally occur.

For this reason, design fork elements carefully so that you minimize the number of possible paths the fork can contain. One way to do this is by reducing the number of branches that start with events. In the following example, a series of branches start with a Visits element:


Logs in
Visits a page named /page1.jhtml
Visits a page named /page2.jhtml
Visits a page named /page3.jhtml

Rewrite scenarios such as this one by placing the event before the fork, and then specifying the page in a series of condition elements after the fork:


Logs in
Visits any page
Page named /page1.jhtml
Page named /page2.jhtml
Page named /page3.jhtml

17 Using Scenario Events

This section describes the default event elements you can insert into a scenario segment (see *Creating a Scenario: Basic Steps* in the *ATG Personalization Guide for Business Users*).

Scenario event elements are triggered when the Scenario Manager component receives events sent as Dynamo Message System messages. The description of each event element below includes the name of the message that triggers the event and the component or part of the system that is responsible for sending the message. The optional parameters that appear in the scenario editor for a specific event correspond to the properties of the message that represents that event, including properties inherited from any parent classes. For example, the parameters that you can select in the scenario editor to further define the Logs In event are the properties associated with the `DPSMessageSource` component. For more information, refer to the Dynamo Message System chapter in the *ATG Programming Guide*.

All scenario events have corresponding entries in the Dynamo Message System registry. In addition, the `scenarioManager.xml` file contains a scenario event registry, which serves to identify the events that you want to associate with scenarios (in contrast to, for example, events that are associated with workflows in ATG Content Administration). The event registry defines various configuration settings for each scenario event, including the message context, which determines whether an event is individual (in other words, it applies to individual users passing through the scenario, such as a `PageVisits` event) or global (it applies to all users, such as a Startup event).

Note that for compatibility with ATG products before version 6.0, the event registry in the `scenarioManager.xml` file is turned off by default, and the event information in the Dynamo Message System registry is used instead. To use the `scenarioManager.xml` event registry, set the `useEventRegistry` property to true in the `ScenarioManager` component.

For more information on both the Dynamo Message System registry and the event registry, refer to Adding Custom Events, Actions, and Conditions to Scenarios.

Scenario events can be collective (they affect or are triggered by everyone progressing through the scenario) or individual (they affect or are triggered only by individual users).

Many scenario events are site aware, meaning they can be configured to apply to specific Web sites in an ATG environment that manages more than one site.

The remaining sections of this chapter describe the default collective and individual scenario events available with the Scenarios module.

Collective events:

- InboundEmail Event

- Shutdown Event
- Startup Event

GSAInvalidation Event Individual events:

- ClickThrough Event
- FormSubmission Event
- SlotItemRequest Event
- Referrer Event
- Login Event
- Logout Event
- Register Event
- AdminRegister Event
- StartSession Event
- EndSession Event
- ProfilePropertyUpdate Event
- AdminProfilePropertyUpdate Event
- ProfileUpdate Event
- AdminProfileUpdate Event
- ViewItem Event
- PageVisit Event
- ScenarioEnd Event
- SiteChanged Event
- ProfileMarkerAdded Event
- ProfileMarkerRemoved Event
- ProfileMarkerReplaced Event
- Business Stage Reached Event

InboundEmail Event

This event is triggered whenever an e-mail message is received by the `atg/dynamo/service/POP3Service` component.

Class name	`atg.userprofiling.dms.InboundEmailMessage`
JMS name	`atg.dps.InboundEmail`

Display name	An email is received
Message scope	Global (collective)
Message source	Component: `/atg/userprofiling/DPSMessageSource` Class: `atg.userprofiling.dms.DPSMessageSource`
Component that calls the message source	`/atg/userprofiling/email/EmailManager`
How this event is triggered	This event is triggered by the `/atg/dynamo/service/POP3Service` component when an email is received.
How to turn this event off	Set to false the `fireInboundEmailEvents` property in the `/atg/dynamo/service/POP3Service` component.

The message that triggers this event contains the following properties:

Property	**Type**	**Scenario editor label**
`messageSubject`	`java.lang.String`	`messageSubject`
	The subject line of the message. Example: *An e-mail is received where* `messageSubject` *is Enroll me!*	
`originalSubject`	`java.lang.String`	`originalSubject`
	The original subject line of the message. The original subject is the subject line without the word “Re:” that may be added if this e-mail message is a response to another. Example: *An e-mail is received where* `originalSubject` *is Come back, we miss you!* In this case, an incoming message with the subject line *Re: Come back, we miss you!* will trigger this event.	
`messageFrom`	`java.lang.String`	`messageFrom`
	The e-mail address of the person who sent the message (the “From” field in an e-mail). Example: *An e-mail is received where* `messageFrom` *includes atg.com* In this case, the system watches for incoming e-mail from the domain atg.com.	
`messageTo`	`java.lang.String[]`	`messageTo`

	An array containing the e-mail addresses of the recipients of the message (the “To” field in an e-mail).	
messageCc	java.lang.String[]	messageCc
	An array containing the e-mail addresses of the recipients included in the “CC” field of the message.	
messageRecipients	java.lang.String[]	messageRecipients
	An array containing all e-mail addresses in both the “To” and “CC” fields.	
messageReplyTo	java.lang.String	messageReplyTo
	The Reply-To e-mail address of the message.	
receivedDate	java.util.Date	receivedDate
	The date on which the message was received by the POP3 server.	
bounced	boolean	bounced
	Indicates whether the e-mail message is one that was returned because it was sent to an invalid address.	
message	javax.mail.Message	message
	The `javax.mail.Message` object that represents the entire contents of the e-mail message.	
bouncedEmailAddress	java.lang.String	bouncedEmailAddress
	If an e-mail message is bounced, this property is set to the e-mail address of the person to whom the message was sent. This property is set only if the bounced property is set to true.	
bouncedReplyCode	java.lang.String	bouncedReplyCode
	The RFC 821 reply code of the bounced e-mail. The reply code indicates why the message was bounced. This property is set only if the bounced property is set to true.	
bouncedStatusCode	java.lang.String	bouncedStatusCode
	The enhanced RFC 1893 status code of the bounced e-mail. The status code is similar to the reply code and can give more specific information about the nature of the bounced email. This property is set only if the bounced property is set to true.	
bouncedErrorMessage	java.lang.String	bouncedErrorMessage

	A String property indicating why the message was bounced. The String is a mail server-specific interpretation of the enhanced RFC 1893 status code. This property is set only if the bounced property is set to true.

The four bounced* properties described above can be null even if the bounced property is set. The values of these properties are determined by specific EmailExaminer classes that parse the bounced e-mail messages. Sometimes the values cannot be determined, and in these cases the properties do not get set.

For more information on how the Personalization module detects and handles bounced e-mail messages, refer to Bounced E-mail.

Shutdown Event

This event is triggered whenever the Dynamo server shuts down.

Class name	atg.nucleus.dms.DASMessage
JMS name	atg.das.Shutdown
Display name	Dynamo shuts down
Message scope	Global (collective)
Message source	Component: /atg/dynamo/messaging/DynamoMessageSource Class: atg.nucleus.dms.DASMessageSource
Component that calls the message source	/Nucleus
How this event is triggered	Nucleus itself sends this message when the Dynamo server shuts down.
How to turn this event off	This event is always fired; you cannot turn it off.

Startup Event

This event is triggered whenever the Dynamo server is started.

Class name	`atg.nucleus.dms.DASMessage`
JMS name	`atg.das.Startup`
Display name	Dynamo starts
Message scope	Global (collective)
Message source	Component: `/atg/dynamo/messaging/DynamoMessageSource` Class: `atg.nucleus.dms.DASMessageSource`
Component that calls the message source	`/Nucleus`
How this event is triggered	Nucleus itself sends this message when the Dynamo server is started.
How to turn this event off	This event is always fired; you cannot turn it off.

GSAInvalidation Event

This event is triggered when SQL repository items are flushed from the cache by the JMS Distributed Cache Invalidator service, usually called as a result of the operation of an external content management system's deployment tools. For more information on distributed cache invalidation, refer to the *ATG Repository Guide*.

Class name	`atg.adapter.gsa.invalidator.GSAInvalidationMessage`
JMS name	`atg.das.GSAInvalidation`
Display name	GSA repository cache invalidation received
Message scope	Global
Message source	Component: `/atg/dynamo/service/GSAInvalidatorService` Class: `atg.adapter.gsa.invalidator.GSAInvalidatorService`
Component that calls the message source	It is called by a remotely executed RMI Client program, `/atg/adapter/gsa/invalidator/GSAInvalidatorClient`.
How this event is triggered	Triggered when an external content management system's content deployment tools update repository database tables with new data, requiring a cache invalidation of the ATG repositories that use these tables.

How to turn this event off	This event cannot be turned off; it is always fired if the JMS Distributed Cache Invalidator service is enabled and a specified cache invalidation occurs.

The message that triggers this event contains the following properties:

Property	Type	Scenario editor label
`itemDescriptorName`	`java.lang.String`	`itemDescriptorName`
	A specific item type. The invalidation of cached items of that type triggers this event.	
`itemId`	`java.lang.String`	`itemId`
	The ID of a specific cached repository item. Invalidation of that item triggers this event.	
`repositoryPath`	`java.lang.String`	`repositoryPath`
	The Nucleus address of a cached `GSARepository` component. Invalidation of the entire repository triggers this event.	

ClickThrough Event

This event is triggered whenever a site visitor clicks a link in a site page or an e-mail message.

Examples: `Clicks link to page /promo.jsp`

`Clicks link from page /index.jsp [to page /promo.jsp]`

`Clicks link with sources BikePromo [from page /index.jsp] [to page /promo.jsp]`

The `ClickThrough` event is similar to the `PageVisit` event, except that it is specifically designed to help you track and report on the links that visitors follow. The message that corresponds to this event contains detailed data about the link that the visitor clicked; for example, it contains information about both the page where the link is located and the page to which it leads. For more information, see the description later in this section of the properties that the message contains.

The `ClickThrough` event works only for links that include an anchor tag containing a `dsource` parameter. The following shows an example of the JSP code for this type of link:

```
<dsp:a href="content/welcome.jsp">
  <dsp:param value="OnSale, MemberDiscount" name="dsource"/>
```

```
      Click here to see this month's discounts!
    </dsp:a>
```

The values that the page developer specifies for the `dsource` parameter (in the example, `OnSale` and `MemberDiscount`) can be any string. The Scenarios module uses them to identify and distinguish the links for purposes such as reporting. For example, the page developer might give several links the same `dsource` value to identify that they belong to a similar group, such as links that point to a specific set of content pages.

Note that at least one `dsource` value must be specified in the link for the event to be triggered.

The event is triggered only once for any clicked link, regardless of how many `dsource` values there are. For example, if the `dsource` value is `"OnSale, MemberDiscount"`, only one `ClickThrough` message is sent.

In addition to the `dsource` parameter, you can optionally include a `dreferrer` parameter as shown in the following JSP example:

```
<dsp:a href="destination/page.jsp">
  <dsp:param value="source/page.jsp" name="dreferrer"/>
  <dsp:param value="OnSale, MemberDiscount" name="dsource"/>
  Click here to see this month's discounts!
</dsp:a>
```

The `dreferrer` parameter defines the `sourcePath` of a `ClickThrough` event (see below for more information about the `sourcePath` property). In normal circumstances, the `sourcePath` is determined by getting the value of the “Referrer” HTTP header. If this header is unavailable, the system can use the value of the `dreferrer` parameter instead.

You can also trigger the `ClickThrough` event from links that you embed in targeted e-mail messages. Use the same JSP or JHTML code (in other words, `dsource` parameters in anchor tags) that you would use for links in site pages. Note that the `dreferrer` feature is especially useful for links that you embed in targeted e-mail messages because no “Referrer” header is sent when a visitor clicks this type of link.

The following table shows additional information about the `ClickThrough` event.

Class name	`atg.userprofiling.dms.ClickThroughMessage`
JMS name	`atg.dps.ClickThrough`
Display name	Clicks a Link
Message context	Request
Message scope	Individual
Message source	Component: `/atg/userprofiling/DPSMessageSource` Class: `atg.userprofiling.dms.DPSMessageSource`

Component that calls the message source	/atg/userprofiling/PageEventTrigger
How this event is triggered	This event is triggered when a request containing a URL with a dsource parameter passes through the ATG servlet pipeline and is processed by the /atg/dynamo/servlet/pipeline/PageViewServletTrigger, which calls the PageEventTrigger. See the important note at the bottom of this section.
How to turn this event off	Set to false the broadcastClickThroughEvents property in component /atg/dynamo/servlet/pipeline/PageViewServletTrigger.

The message that triggers this event contains the following properties:

Property	Type	Scenario editor label
sourcePath	java.lang.String []	to page…
	The pathname of the page that contained the clicked link. Note that the pathname does not include the protocol or domain of the URL. For example, if a visitor clicks a link on the page http://www.example.com/index.jsp, and that link is set up to fire a ClickThrough event, the sourcePath will be /index.jsp.	
destinationPath	java.lang.String	from page…
	The pathname of the page that is represented by the clicked link. Again, the protocol and domain are not included.	
sourceNames	java.lang.String	where source name list…
	A user-defined, comma-separated list of keywords that signifies some sort of information association with the clicked link. At least one keyword is required in order for a ClickThrough message to be fired.	
profileId	java.lang.String	Does not appear.
	The profile ID of the visitor who clicked the link.	
siteId	java.lang.String	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

Important: As described above, this event is triggered when a URL containing a dsource parameter is intercepted in the ATG servlet pipeline by the PageViewServletTrigger. This behavior means that the event will be fired not only when a visitor clicks a given link but also when he or she reloads a page containing that link. If you use the ClickThrough event for reporting purposes, and you want to track the number of times it occurs, note that the total count will include page reloads as well as clicks. Similarly, if you use this event to trigger scenario actions such as sending a targeted e-mail or populating a slot with content, and you design the scenario so that the action can occur more than once for the same site visitor, bear in mind that the action will also occur when a site visitor reloads a page containing a dsource URL.

FormSubmission Event

This event is triggered whenever a visitor submits a form on a site page.

Example: *A form is submitted where* formName *is* decemberSurvey

When the page developer adds a form to a site page, he or she includes a formName parameter in the JSP or JHTML code for the form. The Scenarios module uses the value of the formName parameter to identify the form that the visitor submits. If the page developer does not specify a formName value, the value of the absoluteName property is used instead; for example, the formName of the /atg/userprofiling/ProfileFormHandler component defaults to /atg/userprofiling/ProfileFormHandler.

Note that forms do not trigger this event by default; you must set the sendMessages property of the form handler component to true to have this event work correctly.

Class name	atg.nucleus.dms.formSubmissionMessage
JMS name	atg.das.FormSubmission
Display name	A form is submitted
Message context	Request
Message scope	Individual
Message source	Component: /atg/dynamo/messaging/DynamoMessageSource Class: atg.nucleus.dms.DASMessageSource
Component that calls the message source	An instance of /atg/droplet/GenericFormHandler
How this event is triggered	Triggered by the form handler component after the visitor submits the form.
How to turn this event off	Set to false the SendMessages property of the form handler component. Note, however, that this property is false by default.

The message that triggers this event contains the following properties:

Property	Type	Scenario editor label
formName	java.lang.String	where formName is…
	The name of the form that the visitor submits.	
siteId	java.lang.String	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

SlotItemRequest Event

This event is triggered whenever a visitor performs an action on the site (for example, displays a given page) that causes an active slot to request content items. See Using Slots for more information (especially note the differences between active and passive slots).

Class name	atg.scenario.dms.SlotItemRequestMessage
JMS name	atg.dss.SlotItemRequest
Display name	Items requested
Message context	Request
Message scope	Individual
Message source	Component: /atg/scenario/DSSMessageSource Class: atg.scenario.dms.DSSMessageSource
Component that calls the message source	None. The slot component calls the message source directly.
How this event is triggered	The slot component itself generates the message when it is called upon to provide content. It does so only if the slot is an active slot and has no more items to show.
How to turn this event off	Make the slot component passive instead of active by setting the generation property in the component to passive.

The message that triggers this event contains the following properties:

Property	Type	Scenario editor label
slotName	java.lang.String	by slot <slot_name>
	The name of the slot that requests content items.	
siteId	java.lang.String	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

Referrer Event

This event is triggered whenever a visitor arrives at the site or at a specific page on the site by clicking a link from a site or page with given attributes (for example, a specific URL).

Class name	atg.userprofiling.dms.ReferrerMessage
JMS name	atg.dps.Referrer
Display name	Is referred by external site
Message context	request
Message scope	individual
Message source	Component: /atg/userprofiling/DPSMessageSource Class: atg.userprofiling.dms.DPSMessageSource
Component that calls the message source	/atg/dynamo/servlet/sessiontracking/SessionEventTrigger
How this event is triggered	The SessionEventTrigger sits in the pipeline after the /atg/userprofiling/ProfileRequestServlet and waits for new sessions.
How to turn this event off	Set to false the broadcastReferrerEvents property in the /atg/dynamo/servlet/sessiontracking/SessionEventTrigger component.

The message that triggers this event contains the following properties:

Property	Type	Scenario editor label
`referrerURL`	`java.lang.String`	`referrerURL`
	A fully qualified URL. Example: `http://myhost:8080/test.jsp`	
`referrerPath`	`java.lang.String`	`referrerPath`
	The path relative to the Web application (or document root). Example: `test.jsp`	
`referrerSite`	`java.lang.String`	`referrerSite`
	The URL minus the protocol and the `referrerPath`. Example: `myhost:8080`	
`siteId`	`java.lang.String`	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

Login Event

This event is triggered whenever a visitor logs into the site. For more information on the Personalization module's login features, refer to Tracking Users.

Class name	`atg.userprofiling.dms.DPSMessage`
JMS name	`atg.dps.Login`
Display name	Logs in
Message context	Request
Message scope	Individual
Message source	Component: `/atg/userprofiling/DPSMessageSource` Class: `atg.userprofiling.dms.DPSMessageSource`
Component that calls the message source	`/atg/userprofiling/ProfileEventTrigger`
How this event is triggered	Triggered by the `postLoginUser()` method of the `ProfileFormHandler`.
How to turn this event off	Set to false the `broadcastLoginEvents` property in the `/atg/userprofiling/ProfileEventTrigger` component.

The message that triggers this event contains the following properties:

Property	Type	Scenario editor label
`profileId`	`java.lang.String`	`profileId`
	The profile ID of the visitor who logs in.	
`siteId`	`java.lang.String`	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

Logout Event

The Scenarios module watches for a visitor to log out of the site.

Class name	`atg.userprofiling.dms.DPSMessage`
JMS name	`atg.dps.Logout`
Display name	Logs out
Message context	Request
Message scope	Individual
Message source	Component: `/atg/userprofiling/DPSMessageSource` Class: `atg.userprofiling.dms.DPSMessageSource`
Component that calls the message source	`/atg/userprofiling/ProfileEventTrigger`
How this event is triggered	Triggered by the `preLogoutUser()` method of the `ProfileFormHandler`.
How to turn this event off	Set to false the `broadcastLogoutEvents` property in the `/atg/userprofiling/ProfileEventTrigger` component.

The message that triggers this event contains the following properties:

Property	Type	Scenario editor label
`profileId`	`java.lang.String`	`profileId`
	The profile ID of the visitor who logs out.	
`siteId`	`java.lang.String`	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

Register Event

This event is triggered whenever a previously anonymous visitor registers at the site.

Class name	`atg.userprofiling.dms.DPSMessage`
JMS name	`atg.dps.Register`
Display name	Registers
Message context	Request
Message scope	Individual
Message source	Component: `/atg/userprofiling/DPSMessageSource` Class: `atg.userprofiling.dms.DPSMessageSource`
Component that calls the message source	`/atg/userprofiling/ProfileEventTrigger`
How this event is triggered	Triggered by the `postCreateUser()`.method of the `ProfileFormHandler`.
How to turn this event off	Set to false the `broadcastRegisterEvents` property in the `/atg/userprofiling/ProfileEventTrigger` component.

The message that triggers this event contains the following properties:

Property	Type	Scenario editor label
`profileId`	`java.lang.String`	`profileId`
	The profile ID of the visitor who registers.	

Property	Type	Scenario editor label
`siteId`	`java.lang.String`	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

AdminRegister Event

This event is triggered whenever an administrator registers users at the site by way of a multi profile form handler.

Class name	`atg.userprofiling.dms.AdminRegisterMessage`
JMS name	`atg.dps.AdminRegister`
Display name	Profile registered by admin
Message context	Session
Message scope	Individual
Message source	Component: `/atg/userprofiling/DPSMessageSource` Class: `atg.userprofiling.dms.DPSMessageSource`
Component that calls the message source	`/atg/userprofiling/ProfileEventTrigger`
How this event is triggered	Triggered by the `postCreateUser()`.method of the `MultiProfileAddForm` class.
How to turn this event off	Set to false the `broadcastAdminRegisterEvents` property in the `/atg/userprofiling/ProfileEventTrigger` component.

The message that triggers this event contains the following properties:

Property	Type	Scenario editor label
`adminProfileId`	`java.lang.String`	`adminProfileId`
	The profile ID of the administrator who performs the batch registration.	
`siteId`	`java.lang.String`	Does not appear.

Property	Type	Scenario editor label
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

StartSession Event

This event is triggered when a site visitor starts a new session.

Class name	`atg.userprofiling.dms.DPSMessage`
JMS name	`atg.dps.StartSession`
Display name	Session Starts
Message context	Request
Message scope	Individual
Message source	Component: `/atg/userprofiling/DPSMessageSource` Class: `atg.userprofiling.dms.DPSMessageSource`
Component that calls the message source	`/atg/dynamo/servlet/sessiontracking/SessionEventTrigger`
How this event is triggered	The `SessionEventTrigger` sits in the pipeline after the `/atg/userprofiling/ProfileRequestServlet` and waits for new sessions.
How to turn this event off	Set to false the `broadcastNewSessionEvents` property in the `/atg/dynamo/servlet/sessiontracking/SessionEventTrigger` component.

The message that triggers this event contains the following properties:

Property	Type	Scenario editor label
`profileId`	`java.lang.String`	`profileId`
	The profile ID of the visitor who starts a session.	
`siteId`	`java.lang.String`	Does not appear.

Property	Type	Scenario editor label
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

EndSession Event

This event is triggered whenever a visitor’s session ends or expires.

Class name	`atg.userprofiling.dms.EndSessionMessage`
JMS name	`atg.dps.EndSession`
Display name	Session ends
Message context	Session
Message scope	Individual
Message source	Component: `/atg/userprofiling/DPSMessageSource` Class: `atg.userprofiling.dms.DPSMessageSource`
Component that calls the message source	`/atg/dynamo/servlet/sessiontracking/SessionEventTrigger`
How this event is triggered	This event is triggered by the `SessionEventTrigger`, which sits in the pipeline after the `/atg/userprofiling/ProfileRequestServlet`.
How to turn this event off	Set to false the `broadcastExpiredSessionEvents` property in `/atg/dynamo/servlet/sessiontracking/SessionEventTrigger`.

The message that triggers this event contains the following properties:

Property	Type	Scenario editor label
`profile`	`atg.repository.RepositoryItem`	`profile's`…
	The profile repository item corresponding to the visitor whose session is expiring.	
`profileId`	`java.lang.String`	`profileId`

Property	Type	Scenario editor label
	The profile ID of the visitor whose session is expiring.	
`sessionId`	`java.lang.String`	`sessionId`
	The ID of the session that is expiring.	
`siteId`	`java.lang.String`	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

Note: In some previous versions of the Scenarios module, this message did not contain a `sessionId` property. To determine the session for an `EndSession` event, the Scenarios module used instead the session ID from the request (`context.request.session.id`). However, in some circumstances, it is possible for the session to expire before the `EndSession` message is sent, which means that the session ID is lost, the Scenarios module cannot identify it, and scenarios that use it will not be triggered. For this reason, it is highly recommended that you use the `sessionId` property in this message (`context.message.sessionId`) whenever you need to identify a session for an `EndSession` event. For example, use this property in data mappers that you are using to record `EndSession` event data.

ProfilePropertyUpdate Event

This event is triggered when a site visitor changes the value of a specific profile property by way of an implementation of the profile form handler.

Examples:

Changes Marital status from single to married
Changes Home address's city to Boston
Changes Interests by removing dancing

You can define the properties that trigger the event through the `propertiesToSendUpdateEvents` property of the `ProfileUpdateTrigger` component. By default, a change to any profile property triggers this event. See `/atg/userprofiling/ProfileUpdateTrigger` for more information.

The Personalization module sends a separate instance of the `ProfilePropertyUpdate` message for every value that is changed. For example, if a visitor changes five property values during the same form submission, the Personalization module sends five messages to the Scenario Manager.

See also the description of the Update User Profile element.

Class name	`atg.userprofiling.dms.ProfilePropertyUpdateMessage`

JMS name	`atg.dps.ProfilePropertyUpdate`
Display name	Profile property updated by user
Message context	Request
Message scope	Individual
Message source	Component: `/atg/userprofiling/DPSMessageSource` Class: `atg.userprofiling.dms.DPSMessageSource`
Component that calls the message source	`/atg/userprofiling/ProfileUpdateTrigger`
How this event is triggered	Triggered by the `postUpdateUser()` methods in the `ProfileFormHandler` and `MultiProfileUpdateFormHandler`.
How to turn this event off	Set to false the `generateProfileUpdateEvents` property in the `/atg/userprofiling/ProfileUpdateTrigger` component.

The message that triggers this event contains the following properties:

Property	Type	Scenario editor label
`propertyPath`	`java.lang.String`	`by changing <property list>`
	The name of the property that has changed, for example `maritalStatus` or `homeAddress.city`	
`oldValue`	`java.lang.Object`	`where old value`
	The old value of the property, before it was changed.	
`newValue`	`java.lang.Object`	`where new value`
	The new value of the property, after it was changed.	
`changeSign`	`int`	Does not appear.
	An integer representing whether the new value is greater than, less than, or equal to the old value. A positive `changeSign` value indicates that the new property value is greater than the old, a negative value indicates that it is less than the old, and zero indicates that they are equal or not comparable.	
`changePercentage`	`double`	Does not appear.
	For number type properties only, the absolute value of the percent difference between the old and new values.	
`changeAmount`	`double`	Does not appear.

Property	Type	Scenario editor label
	For number type properties only, the absolute difference between the old value and the new value	
`elementsAdded`	`java.lang.Object[]`	Does not appear.
	If the changed property is an array or Collection, `elementsAdded` is the array of items that are members of the new value but not members of the old value.	
`elementsRemoved`	`java.lang.Object[]`	Does not appear.
	If the changed property is an array or Collection, `elementsRemoved` is the array of items that are members of the old value but not members of the new value.	
`profileId`	`java.lang.String`	Does not appear.
	The profile ID of the visitor whose profile property has changed.	
`reportingOldValue`	`java.lang.String`	Does not appear.
	A text representation of the value of the `oldValue` property. This property is used for reporting.	
`reportingNewValue`	`java.lang.String`	Does not appear.
	A text representation of the value of the `newValue` property. This property is used for reporting.	
`reportingChangeSign`	`java.lang.String`	Does not appear.
	A text representation of the value of the `changeSign` property. This property is used for reporting.	
`reportingChangeAmount`	`double`	Does not appear.
	A Double representation of the value of the `changeAmount` property. This property is used for reporting.	
`reportingChangePercentage`	`double`	Does not appear.
	A Double representation of the value of the `changePercentage` property. This property is used for reporting.	
`reportingElementsAdded`	`java.lang.String`	Does not appear.
	A comma-separated representation of the contents of the `elementsAdded` property. This property is used for reporting.	
`reportingElementsRemoved`	`java.lang.String`	Does not appear.
	A comma-separated representation of the contents of the `elementsRemoved` property. This property is used for reporting.	

Property	Type	Scenario editor label
siteId	java.lang.String	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

AdminProfilePropertyUpdate Event

This event is triggered when an administrator changes the value of a specific profile property in a user's profile (or in a batch of profiles) by way of an implementation of the multi profile form handler.

You can define the properties that trigger the event through the propertiesToSendAdminUpdateEvents property of the ProfileUpdateTrigger component. By default, a change to any profile property triggers this event. See /atg/userprofiling/ProfileUpdateTrigger for more information.

Class name	atg.userprofiling.dms.AdminProfilePropertyUpdateMessage
JMS name	atg.dps.AdminProfilePropertyUpdate
Display name	Profile property updated by admin
Message context	Session
Message scope	Individual
Message source	Component: /atg/userprofiling/DPSMessageSource Class: atg.userprofiling.dms.DPSMessageSource
Component that calls the message source	/atg/userprofiling/ProfileUpdateTrigger
How this event is triggered	Triggered by the postUpdateUser() methods in the MultiProfileUpdateFormHandler.
How to turn this event off	Set to false the generateAdminProfileUpdateEvents property in the /atg/userprofiling/ProfileUpdateTrigger component.

This event has the same properties as the ProfilePropertyUpdate event described earlier, with the addition of the following property:

Property	Type	Scenario editor label
`adminProfileId`	`java.lang.String`	`adminProfileId`
	The profile ID of the administrator who performs the update.	
`siteId`	`java.lang.String`	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

ProfileUpdate Event

This event is triggered when a site visitor changes his or her profile by way of an implementation of the profile form handler. It is similar to the `ProfilePropertyUpdate` event, except that it applies to any change to a profile rather than to a change to a single property.

Class name	`atg.userprofiling.dms.ProfileUpdateMessage`
JMS name	`atg.dps.ProfileUpdate`
Display name	Profile updated by user
Message context	Request
Message scope	Individual
Message source	Component: `/atg/userprofiling/DPSMessageSource` Class: `atg.userprofiling.dms.DPSMessageSource`
Component that calls the message source	`/atg/userprofiling/ProfileUpdateTrigger`
How this event is triggered	Triggered by the `postUpdateUser()` methods in the `ProfileFormHandler` and `MultiProfileUpdateFormHandler`.
How to turn this event off	Set to false the `generateProfileUpdateEvents` property in the `/atg/userprofiling/ProfileUpdateTrigger` component.

The message that triggers this event contains the following properties:

Property	Type	Scenario editor label
`changedProperties`	`java.util.List`	`where changedProperties`

Property	Type	Scenario editor label
	A List showing the path of each property that has changed.	
`oldValues`	`java.util.Map`	`oldValues`
	A Map where the map keys are the profile property paths, and the map values are the old property values before they were changed.	
`newValues`	`java.lang.Object`	`newValues`
	A Map where the map keys are the profile property paths, and the map values are the new property values after they were changed.	
`profileId`	`java.lang.String`	`profileId`
	The profile ID of the visitor whose profile has changed.	
`reportingChangedProperties`	`java.lang.String`	`reportingChangedProperties`
	A text representation of the value of the `changedProperties` property. This property is used for reporting.	
`reportingOldValues`	`java.lang.String`	`reportingOldValues`
	A text representation of the contents of the `oldValues` property. This property is used for reporting.	
`reportingNewValues`	`java.lang.String`	`reportingNewValues`
	A text representation of the contents of the `NewValues` property. This property is used for reporting.	
`siteId`	`java.lang.String`	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

AdminProfileUpdate Event

This event is triggered when an administrator changes or deletes a user profile or group of profiles by way of an implementation of the multi profile form handler. It is similar to the `AdminProfilePropertyUpdate` event, except that it applies to any change to a profile rather than to a change to a specific property.

Class name	`atg.userprofiling.dms.AdminProfileUpdateMessage`
JMS name	`atg.dps.AdminProfileUpdate`
Display name	Profile updated by admin
Message context	Session
Message scope	Individual
Message source	Component: `/atg/userprofiling/DPSMessageSource` Class: `atg.userprofiling.dms.DPSMessageSource`
Component that calls the message source	`/atg/userprofiling/ProfileUpdateTrigger`
How this event is triggered	Triggered by the `postUpdateUser()` methods in the `MultiProfileUpdateFormHandler`.
How to turn this event off	Set to false the `generateAdminProfileUpdateEvents` property in the `/atg/userprofiling/ProfileUpdateTrigger` component.

This event has the same properties as the ProfileUpdate event described earlier, with the addition of the following property:

Property	Type	Scenario editor label
`adminProfileId`	`java.lang.String`	`adminProfileId`
	The profile ID of the administrator who performs the update.	
`siteId`	`java.lang.String`	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

ViewItem Event

This event is triggered when a site visitor views an item from any repository (any `RepositoryItem`).

Examples: *Views any item*

Views an item from Funds whose Aggressive Index is 3.

Class name	atg.userprofiling.dms.ViewItemMessage
JMS name	atg.dps.ViewItem
Display name	Views
Message context	Request
Message scope	Individual
Message source	Component: /atg/userprofiling/DPSMessageSource Class: atg.userprofiling.dms.DPSMessageSource
Component that calls the message source	/atg/targeting/TargetedContentTrigger
How this event is triggered	Triggered by the fireViewItemEvents servlet bean parameter in the following targeters: TargetingForEach TargetingFirst TargetingRandom TargetingRange Also triggered by the same parameter in the RepositoryLookup servlet bean. The parameter is set to false by default. See also the note below.
How to turn this event off	Set to false the fireViewItemEvents servlet bean parameter.

Note: The fireContentEvent and fireContentTypeEvent servlet bean parameters included in earlier versions of ATG products (Dynamo 4 and later) were designed to fire events only if the item that the visitor viewed was stored in a content repository (a ContentRepositoryItem). These parameters have been deprecated in favor of the more flexible fireViewItemEvents parameter, which is triggered when an item from any repository (including, for example, the profile repository) is viewed.

Any existing targeters that use the fireContentEvent and fireContentTypeEvent parameters will still work as expected; in other words, they will fire a Dynamo 4-style ContentEvent or ContentTypeEvent if the visitor views a content repository item.

The message that triggers this event contains the following properties:

Property	**Type**	**Scenario editor label**
repositoryName	java.lang.String	an item from <repository_name>
	The name of the repository that contains the viewed item.	

Property	Type	Scenario editor label
itemType	java.lang.String	of type
	The name of the item type (itemDescriptorName) to which this item belongs.	
repositoryId	java.lang.String	Does not appear . The “named” keyword is used to specify the appropriate item.
	The repository ID of the viewed item.	
profileId	java.lang.String	Does not appear.
	The profile ID of the visitor who viewed the item.	
item	atg.repository.RepositoryItem	Does not appear.
	The actual RepositoryItem object that the visitor viewed.	
targeter	atg.targeting.Targeter	Does not appear.
	The targeter that was used to get this item, if available.	
path	java.lang.String	Does not appear.
	The path of the page that contains the item.	
siteId	java.lang.String	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

PageVisit Event

This event is triggered when a site visitor displays a page.

Examples: `Visits any page`

`Visits a page named welcome.jsp`

`Visits a page in folder /Dynamo/Solutions/Pioneer Cycling`

Note that, although the effect of this event is to fire when a specified page is displayed, the event is actually triggered when the Personalization module issues the request for the page; no checking is performed to see whether the page exists.

Class name	`atg.userprofiling.dms.PageVisitMessage`
JMS name	`atg.dps.PageVisit`
Display name	Visits
Message context	Request
Message scope	Individual
Message source	Component: `/atg/userprofiling/DPSMessageSource` Class: `atg.userprofiling.dms.DPSMessageSource`
Component that calls the message source	`/atg/userprofiling/PageEventTrigger`, or `/atg/userprofiling/SendPageEvent` (servlet bean in a JSP)
How this event is triggered	The pipeline servlet `/atg/dynamo/servlet/pipeline/PageViewServletTrigger` sits in the pipeline and calls the `PageEventTrigger` component when a page is viewed. Alternatively, page developers can trigger this event by embedding the `/atg/userprofiling/SendPageEvent` component in a JSP.
How to turn this event off	Set to false the `broadcastPageViewedEvents` property in either the `/atg/dynamo/servlet/pipeline/PageViewServletTrigger` component or the `/atg/userprofiling/SendPageEvent` component (see above).

The message that triggers this event contains the following properties:

Property	**Type**	**Scenario editor label**
`profileId`	`java.lang.String`	`whose profileId…`
	The profile ID of the visitor who visited the page.	

`scenarioPathInfo`	`java.lang.String`	`in folder…`
		`named…`

	The path of the page that triggers the event, including the application name and context root (or the Dynamo document root). Example: `Quincy Funds J2EE DAF Demo:/index.jsp`	

`path`	`java.lang.String`	`in Dynamo folder`
		`with Dynamo path`

	The Dynamo document root path of the page that triggers this event, for example, `/demo/home/index.jhtm`	

Property	Type	Scenario editor label
folder	java.lang.String	whose folder…
	The Dynamo document root folder containing the page that triggers this event, for example /demo/home.	
siteId	java.lang.String	Does not appear.
	The ID of the current site. The value is provided by the Site Context Manager. This parameter applies to multisite environments.	

ScenarioEnd Event

This event is fired when a scenario is terminated for any of the following reasons:

- The scenario is deleted or disabled by an ACC user.
- The scenario is disabled by a DisableScenario action.

Note that the scenario editor does display a Scenario Ends event as an element that users can include in a scenario, but the event does not of course have to be used explicitly in a scenario in order to be fired.

Class name	atg.process.dms.ProcessEndMessage
JMS name	atg.dss.ScenarioEnd
Display name	Scenario Ends
Message scope	Global
Message source	Component: /atg/scenario/DSSMessageSource Class: atg.scenario.dms.DSSMessageSource
Component that calls the message source	/atg/process/ScenarioUpdateTrigger (class atg.process.ProcessUpdateTrigger)

How this event is triggered	The Scenario Manager generates and sends a `ProcessUpdateEvent` to any configured `ProcessUpdateListeners`, including the `ScenarioUpdateTrigger` component. When it receives the event, the `ScenarioUpdateTrigger` component determines whether the event indicates that a scenario has been terminated, and if so, it generates and sends the corresponding JMS message of the type specified by its `ProcessEndMessageType` property (by default, `atg.dss.ScenarioEnd`). Note that only the `ScenarioUpateTrigger` component located on the process editor server performs these steps. The components on other servers receive the event but do not send the JMS message.
How to turn this event off	Set to false the `sendProcessEndMessages` property in the `/atg/process/ScenarioUpdateTrigger` component.

The `ScenarioEnd` message contains the following properties:

Property	Type	Scenario editor label
`endType`	`int`	`endType`
	Indicates the way the scenario was terminated. The value can be `REMOVE` (indicating the scenario was deleted by an ACC user), `DISABLE` (disabled by an ACC user), or `DISABLE_SELF` (disabled as a result of a `DisableScenario` action)	
`processName`	`java.lang.String`	`processName`
	The name of the scenario that was terminated.	

SiteChanged Event

This event is fired when a user navigates from one registered Web site to another, thus causing the site context to change. Note that the event is designed only for use with sites supported by a multisite ATG installation (sites registered and deployed through Site Administration).

Class name	`atg.multisite.dms.SiteChangedMessage`
JMS name	`atg.multisite.SiteChanged`

Display name	Switches Site Context
Message scope	Individual
Message source	Component: `/atg/multisite/SiteSessionEventSender` Class: `atg.multisite.SiteSessionEventSender`
Component that calls the message source	`/atg/dynamo/servlet/dafpipeline/SiteSessionEventTrigger` or `/atg/dynamo/servlet/pipeline/SiteSessionEventTrigger`
How this event is triggered	A `SiteChanged` message is fired when the value of the last visited site in the current site context is different from the current site.
How to turn this event off	Set to false the `sendSiteChangedEvents` property in the `/atg/dynamo/servlet/dafpipeline/SiteSessionEventTrigger` or `/atg/dynamo/servlet/pipeline/SiteSessionEventTrigger` components

The `SiteChanged` message contains the following properties:

Property	**Type**	**Scenario editor label**
`newSiteID`	`java.lang.String`	`Site is switched to`
	The site ID of the current site.	
`oldSiteID`	`java.lang.String`	`Site is switched from`
	The site ID of the last visited site (the site that triggered the event).	

ProfileMarkerAdded Event

This event is triggered when a marker is added to a profile.

Class name	`atg.markers.MarkerAddedEventMessage`
JMS name	`atg.profile.marker.added`
Display name	Profile marker added
Message context	Session

Message scope	Individual
Message source	/atg/markers/RepositoryMarkerMessageSource
Component that calls the message source	/atg/markers/userprofiling/ProfileMarkerManager
How this event is triggered	The ProfileMarkerManager adds a marker to a profile.
How to turn this event off	Set the /atg/markers/userprofiling/ProfileMarkerManager.generateEvents property to false. When you turn off this event, you disable the ProfileMarkerRemoved and ProfileMarkerReplaced events as well.

The event message contains the following properties that pertain to the event itself:

Property	**Type**	**Scenario editor label**
eventDate	java.sql.Timestamp	eventDate
	The date and time at which the event was generated.	
markerAddedData	atg.markers.MarkerData	markerAddedData
	An object representing the profile marker that was added.	
markedItemId	java.lang.String	markedItemId
	The ID for the profile that received the marker. The value in this property is the same as the value in the profileId property. When a user causes a profile marker to be added to his or her own profile, this property is set to the active profile ID.	
markedItemType	java.lang.String	markedItemType
	The type of repository item to which the marker was added.	
markerPropertyName	Java.lang.String	markerPropertyName
	The name of the profile property that stores markers.	
repositoryName	java.lang.String	repositoryName
	The name of the repository with an item that was marked.	
parentSessionId	java.lang.String	parentSessionId

	The session ID for the session during which the event is generated.	
`profileId`	`java.lang.String`	`profileId`
	The ID for the profile that received the marker. The value in this property is the same as the value in the `markedItemId` property. When a user causes a profile marker to be added to his or her own profile, this property is set to the active profile ID.	
`sessionId`	`java.lang.String`	`sessionId`
	The session ID for the session during which the marker is added.	

The event message contains the following properties that pertain to the marker created by the event:

Property	**Type**	**Scenario editor label**
`Key`	`java.lang.String`	`Key`
	The value in the marker's `key` property.	
`Value`	`java.lang.String`	`Value`
	The value in the marker's `value` property.	
`Data`	`java.lang.String`	`Data`
	The value in marker's the `data` property.	
`createDate`	`java.sql.Timestamp`	`createDate`
	The date and time when the marker was created.	
`markerItemId`	`Java.lang.String`	`markerItemId`
	The marker's ID.	
`markerItemType`	`Java.lang.String`	`markerItemType`
	The marker's repository item type.	

ProfileMarkerRemoved Event

This event is triggered when a marker is removed from a profile.

Class name	`atg.markers.MarkerRemovedEventMessage`
JMS name	`atg.profile.marker.removed`
Display name	Profile marker removed
Message context	Session
Message scope	Individual
Message source	`/atg/markers/RepositoryMarkerMessageSource`
Component that calls the message source	`/atg/markers/userprofiling/ProfileMarkerManager`
How this event is triggered	The `ProfileMarkerManager` removes a marker from a profile.
How to turn this event off	Set the `/atg/markers/userprofiling/ProfileMarkerManager.generateEvents` property to `false`. When you turn off this event, you disable the `ProfileMarkerAdded` and `ProfileMarkerReplaced` events as well.

The event message contains the following properties:

Property	**Type**	**Scenario editor label**
`eventDate`	`java.sql.Timestamp`	`eventDate`
	The date and time when the event was generated.	
`markerRemovedData`	`atg.markers.MarkerData`	`markerRemovedData`
	An object representing the marker that was removed.	
`markedItemId`	`java.lang.String`	`markedItemId`
	The ID for the profile from which the marker was removed. The value in this property is the same as the value in the `profileId` property. When a user causes a profile marker to be removed from his or her own profile, this property is set to the active profile ID.	
`markedItemType`	`java.lang.String`	`markedItemType`
	The type of repository item from which the marker was removed.	
`markerPropertyName`	`Java.lang.String`	`markerPropertyName`

	The name of the profile property that held the marker that was removed.	
`repositoryName`	`java.lang.String`	`repositoryName`
	The name of the repository name with the item that held the marker.	
`parentSessionId`	`java.lang.String`	`parentSessionId`
	The session ID for the session during which the event was generated.	
`profileId`	`java.lang.String`	`profileId`
	The ID for the profile from which the marker was removed. The value in this property is the same as the value in the `markedItemId` property. When a user causes a profile marker to be removed from his or her own profile, this property is set to the active profile ID.	
`sessionId`	`java.lang.String`	`sessionId`
	The session ID for the session during which the marker was removed.	

The event message contains the following properties that pertain to the marker removed by the event. If more than one marker is removed, a separate message is generated for each removed marker.

Property	**Type**	**Scenario editor label**
`key`	`java.lang.String`	`Key`
	The `key` property value of the marker being removed.	
`value`	`java.lang.String`	`Value`
	The `value` property value of the marker being removed.	
`data`	`java.lang.String`	`Data`
	The `data` property value of the marker being removed.	
`createDate`	`java.sql.Timestamp`	`createDate`
	The date and time when the marker was created.	
`markerItemId`	`java.lang.String`	`markerItemId`

	The repository ID for the marker being removed.	
markerItemType	java.lang.String	markerItemType
	The repository item type of the marker being removed.	

ProfileMarkerReplaced Event

This event is triggered when one profile marker is replaced by another.

Class name	atg.markers.MarkerReplacedEventMessage
JMS name	atg.profile.marker.replaced
Display name	Profile marker replaced
Message context	Session
Message scope	Individual
Message source	/atg/markers/RepositoryMarkerMessageSource
Component that calls the message source	/atg/markers/userprofiling/ProfileMarkerManager
How this event is triggered	The ProfileMarkerManager replaces one or more markers with another marker.
How to turn this event off	Set the /atg/markers/userprofiling/ProfileMarkerManager.generateEvents property to false. When you turn off this event, you disable the ProfileAddedMarker and ProfileRemovedMarker events as well.

The event message contains the following properties that pertain to the event itself:

Property	**Type**	**Scenario editor label**
eventDate	java.sql.Timestamp	eventDate
	The date on which the event was generated.	
markerAddData	atg.markers.MarkerData	markerAddData

	An object representing the marker that was added.	
`markerReplacedData`	`atg.markers.MarkerData`	`markerReplacedData`
	An object representing the marker that was replaced.	
`markedItemId`	`java.lang.String`	`markedItemId`
	The ID for the profile from which the marker was replaced. The value in this property is the same as the value in the `profileId` property. When a user causes a profile marker to be replaced on his or her own profile, this property is set to the active profile ID.	
`markedItemType`	`java.lang.String`	`markedItemType`
	The type of repository item from which the marker was replaced.	
`markerPropertyName`	`Java.lang.String`	`markerPropertyName`
	The name of the profile property that holds the marker being replaced.	
`repositoryName`	`java.lang.String`	`repositoryName`
	The repository name of Profile item that held the marker.	
`parentSessionId`	`java.lang.String`	`parentSessionId`
	The session ID for the session during which the event was generated.	
`profileId`	`java.lang.String`	`profileId`
	The ID for the profile on which a marker is being replaced. The value in this property is the same as the value in the `markedItemId` property. When a user causes a profile marker to be replaced on his or her own profile, this property is set to the active profile ID.	
`sessionId`	`java.lang.String`	`sessionId`
	The session ID for the session during which the marker is replaced.	

The event message contains two objects, one of which describes the new, replacement marker and the other describes the marker being replaced. The following section describes the properties on these objects:

Property	Type	Scenario editor label
key	java.lang.String	key
	The key property value of the marker.	
value	java.lang.String	value
	The value property value of the marker.	
data	java.lang.String	data
	The data property value of the marker.	
createDate	java.sql.Timestamp	createDate
	The date and time when the marker was created.	
markerItemId	java.lang.String	markerItemId
	The repository ID of the marker.	
markerItemType	java.lang.String	markerItemType
	The repository item type of the marker.	

Business Stage Reached Event

This event is triggered when an object reaches a stage in a business process. For more information, see the Defining and Tracking Business Processes chapter.

Class name	atg.markers.bp.BusinessProcessEventMessage
JMS name	atg.business.process.stage.reached
Display name	Business Process Stage Reached
Message context	session
Message scope	individual
Message source	Component: /atg/markers/RepositoryMarkerMessageSource Class: atg.markers.MarkerMessageSource
Component that calls the message source	/atg/markers/bp/BusinessProcessManager

How this event is triggered	This event is triggered by the `BusinessProcessManager` when an object is marked with a new business process stage.
How to turn this event off	Set to false the `generateEvents` property in the `BusinessProcessConfiguration` or `BusinessProcessManager` component.

The message that triggers this event contains the following properties:

Property	**Type**	**Description**
`businessProcessName`	`String`	The name of the business process in which the stage has been reached.
`businessProcessStage`	`String`	The name of the stage in the business process that has been reached
`businessProcessStageSequence`	`int`	The sequence number of the stage.
`processStartTime`	`Timestamp`	The time the first stage in the business process was reached.

Scenario Events and Transient Properties

As explained in the *SQL Repositories* chapter of the *ATG Repository Guide*, you can set up repository items to have transient attributes, which are readable and writable but are neither stored in nor read from the persistent data store.

For a scenario to access transient profile properties, it must include an event element that establishes a session has begun (for example, a Session Starts or Views event). The event must occur before any subsequent scenario actions that use the transient properties.

Bear in mind also that the scenario can evaluate transient properties as long as the user's session is active. When the session expires, the Personalization module sets the transient property value to null, and the Scenarios module can no longer use it as a criterion for scenario evaluation.

Scenarios and Anonymous Users

In some cases, you may have scenarios that are specifically designed to handle visitors who start as anonymous users and then either log in or register to become site members while they are progressing through the scenario.

The Scenarios module manages the transition from anonymous to registered visitor by means of an instance of `atg.scenario.userprofiling.ScenarioProfileFormHandler`, which is a subclass of `atg.userprofiling.ProfileFormHandler`. The `ScenarioProfileFormHandler` takes the scenario instances associated with an anonymous visitor and attaches them to the persistent visitor when he or she registers or logs in. With this behavior, guest users can progress through a scenario and then, when they become persistent users, the Scenarios module does not lose track of their scenario states.

For the Scenarios module, the `/atg/userprofiling/ProfileFormHandler` component is automatically overridden to point to the default `ScenarioProfileFormHandler`, so there are no configuration steps to perform. However, if you want to extend the `ProfileFormHandler` class and keep the `ScenarioProfileFormHandler` functionality described here, you must subclass `ScenarioProfileFormHandler` instead of `ProfileFormHandler`.

18 Using Scenario Actions

This section describes the default action elements you can insert into a scenario segment. Note that, as for all scenario elements, you create actions through the scenario editor in the ACC (see *Creating a Scenario: Basic Steps* in the *ATG Personalization Guide for Business Users*).

Scenario actions are implementations of the `atg.process.action.Action` interface. In general terms, they represent actions that you want the Scenarios module to perform as a response to some form of user behavior (represented by scenario events) – for example, you might want the system to send a "Welcome" email to anyone who registers at one of your sites. You would create a scenario that includes a Registers event followed by a Send Email action; when the Registers event is triggered by the site visitor, the Send Email action is performed by the Scenarios module.

All scenario actions have corresponding entries in the action registry within the Scenario Manager configuration file (`scenarioManager.xml`). The action registry tags define various settings for each action, including information that determines whether an action is individual or collective, and how you want the system to respond to any errors that occur while an action is being executed.

The following table describes the action registry tags that apply to all action elements. See also the Process Manager document type definition (DTD) located in `<ATG10dir>\DSS\lib\classes.jar`.

Tag	Required	Description

`<action-name>`
`</action-name>`
	Yes	The logical name of the action as passed to an action handler.

`<action-class>`
`</action-class>`
	Yes	A Java class that is an implementation of the `atg.process.action.Action` interface.

`<action-configuration>`
`</action-configuration>`
	No	The Nucleus path of the action's configuration file.

`<icon-resource>`
`</icon-resource>`
	No	A CLASSPATH-relative path to a 24x24 icon resource for a descriptor. Note that, for reasons of backwards compatibility, this is **not** a resource bundle key.

Tag	Required	Description

`<action-terminal>`
`</action-terminal>`
	No	Boolean. Terminal actions are those that cannot be followed by other elements in a scenario, usually for reasons of logic. For example, the Disable Scenario action by definition cannot have any subsequent elements. The scenario editor indicates an action is terminal by showing no connecting line after it and by displaying in red any elements that a user attempts to add.

`<action-execution-policy>`
`</action-execution-policy>`
	No	Determines whether an action applies to individual users in a scenario (individual) or to all users in a scenario (collective). The default value is individual. For more information, see Specifying the <action-execution-policy> Tag.

`<action-error-response>`
`</action-error-response>`
	No	Specifies what to do if an error occurs while the action is being executed. You can specify `delete`, `continue`, or `retry` as the value. For descriptions of these options, see Specifying the <action-error-response> Tag.

`<action-parameter>`
`</action-parameter>`
	At least one	A named parameter passed to an action. Its value is an expression, one that might be used, for example, as the component of a filter. By default, action parameters are Strings.

The following list shows the possible child tags for an <action-parameter> tag:

Tag	Required	Description

`<action-parameter-name>`
`</action-parameter-name>`
	Yes	The PDL name of the parameter.

`<action-parameter-class>`
`</action-parameter-class>`
	No	An optional type for the parameter used to constrain it in the ACC user interface. The default type is String.

`<action-parameter-repository-name>`
`</action-parameter-repository-name>`
	No	If the parameter is a repository ID or an array of repository IDs, this tag supplies the name of the repository.

Tag	Required	Description
<action-parameter-repository-item-type> </action-parameter-repository-item-type>	No	If the parameter is a repository ID or an array of repository IDs, this tag supplies the repository item type. In the ACC, users can select any item of this type or its subtypes for this action.
<action-parameter-expression-construct> </action-parameter-expression-construct>	No	An expression editor grammar construct name that can be used to edit the value of this parameter. If specified, this tag permits the construct to control custom editing of an individual parameter.

For information on the <resource-bundle>, <display-name-resource>, <required>, <expert>, <description>, and <description-resource> tags, refer to the DTD for a standard SQL repository in the *ATG Repository Guide*.

The following example shows the action registry tags for the EmailNotify action:

```
<action>
      <action-name>
        emailNotify
      </action-name>
      <action-class>
        atg.process.action.EmailNotify
      </action-class>
      <action-configuration>
        /atg/scenario/configuration/EmailNotifyConfiguration
      </action-configuration>
      <resource-bundle>
        atg.ui.scenario.TemplateResources
      </resource-bundle>
      <display-name-resource>
        emailNotify.displayName
      </display-name-resource>
      <icon-resource>
        emailNotify.icon
      </icon-resource>
      <description-resource>
        emailNotify.description
      </description-resource>
      <action-execution-policy>
        collective
      </action-execution-policy>
```

```
<action-error-response>
   delete
</action-error-response>

<action-parameter>
  <action-parameter-name>
    recipientIds
  </action-parameter-name>
  <required>
    false
  </required>
  <description-resource>
     emailNotify.recipientIds.description
  </description-resource>
</action-parameter>

<action-parameter>
  <action-parameter-name>
    recipientAddresses
  </action-parameter-name>
  <required>
    false
  </required>
  <description-resource>
     emailNotify.recipientAddresses.description
  </description-resource>
</action-parameter>

<action-parameter>
  <action-parameter-name>
    recipientAddress
  </action-parameter-name>
  <required>
    false
  </required>
  <description-resource>
    emailNotify.recipientAddress.description
  </description-resource>
</action-parameter>

<action-parameter>
  <action-parameter-name>
    template
  </action-parameter-name>
  <required>
    true
  </required>
  <description-resource>
    emailNotify.template.description
  </description-resource>
```

```
      </action-parameter>

      <action-parameter>
        <action-parameter-name>
          application-name
        </action-parameter-name>
        <required>
          false
        </required>
        <description-resource>
          emailNotify.application-name.description
        </description-resource>
      </action-parameter>

</action>
```

The remaining sections of this chapter describe the following default scenario actions:

- Modify Action
- Set Random Action
- Redirect Action
- FillSlot Action
- EmptySlot Action
- Disable Scenario Action
- Record Event Action
- Record Audit Trail Action
- Filter Slot Contents Action
- Add Marker To Profile Action
- Remove All Markers From Profile Action
- Remove Markers From Profile Action
- Add Stage Reached Action
- Remove Stage Reached Action
- E-mail-Related Actions: EmailNotify and SendEmail

Modify Action

The Modify action does any of the following:

- Changes a specified property in the visitor's profile, for example: `Change Person's interests to skating`.

- Sets the value of a scenario variable (the Set Variable action in the scenario editor).
- Changes the value of a scenario variable.
- Changes a person’s parent organization or set of roles.

Display name in the scenario editor: Change

Action Registry Tag	Value
action name	`modify`
configuration component	none
action execution policy	individual
action error response	delete

The Modify action has the following parameters:

Parameter	Required	Description
modified	yes	The property, variable, roles, or organization to change
modifier	yes	The value to set
operator	yes	The type of change (for example, append, assign, or subtract)

Set Variable Action

Note that the Set Variable action in the scenario editor is a variant of the Modify action. You can use the Set Variable action to define a value that you can use, for example, to trigger an action later in the same scenario segment (scenario variables apply only to the segment in which you set them).

Example:

```
Set variable LateNightShopper to Yes
```

The following example from an SDL file shows a Set Variable action that creates a variable called `LateNightShopper` and sets it to the value `VeryLate`.

```
<action-name construct="variable-declaration-action">modify</action-name>
      <action-param name="modified">
        <variable type="java.lang.String">LateNightShopper</variable>
```

```
</action-param>
<action-param name="operator">
  <constant>assign</constant>
</action-param>
<action-param name="modifier">
  <constant>Yes</constant>
</action-param>
```

Scenario variables can be any of the following types:

- String
- Long
- Double
- Flag
- Date

SetRandom Action

The `SetRandom` action assigns a random value between 0.0 and 1.0 to the given expression. The expression must be a `MutableExpression` that evaluates to a Double. Note that this action does not appear in the scenario editor; it is used by the Scenarios module during the process of creating a randomizing fork element.

Action Registry Tag	**Value**
action name	`setRandom`
configuration component	none
action execution policy	individual
action error response	delete

The `SetRandom` action has the following parameters:

Parameter	**Required**	**Description**
`modified`	yes	The expression to modify.

Redirect Action

The Redirect action redirects the current HTTP request to the specified URL.

Example: `Redirect to page with Dynamo path http://myhost:8080/QuincyFunds/en/RegistrationError.jsp`

Action Registry Tag	Value
action name	redirect
configuration component	`/atg/scenario/configuration/RedirectActionConfiguration`
action execution policy	individual
action error response	continue

The Redirect action has the following parameters:

Parameter	Required	Description
`path`	yes	The path of the page to which you want to redirect users. Corresponds to the ACC option `Redirect to page with Dynamo path`… Users specify the fully qualified path of the e-mail template to send. If the document is outside the Dynamo document root because it is part of a non-DAS J2EE application, they must specify the application context root manually. This parameter reflects the way that users specified document paths in versions of Dynamo earlier than 6.0.0 and exists mostly for compatibility with those versions.
`scenarioPathInfo`	yes	The path of the page to which you want to redirect users. Corresponds to the ACC option `Redirect to page with path…`. Users specify the path by selecting the page in a document picker dialog box. The path is automatically prepended with either the application context root (if the template is part of a J2EE application) or a flag indicating that the template is located in the Dynamo document root.

Note: When you add a Redirect action to a scenario, make sure that the action does not conflict with any other redirect activity that may be occurring on the site. For example, if you design a scenario that redirects users after a Login event, make sure your login pages do not perform their own redirect as well. Conflicting redirects could cause unexpected behavior to occur.

RedirectActionConfiguration Component

The following example shows the default properties file for the `RedirectActionConfiguration` component, which you use to define settings for the `Redirect` action.

```
# Version: $Change: 244651 $$DateTime: 2002/06/24 09:19:29 $
$class=atg.scenario.action.RedirectRequestConfiguration

webAppRegistry=/atg/registry/WebApplicationRegistry
```

The `webAppRegistry` property points to the registry of all Web applications that are on the J2EE server where Dynamo is running.

FillSlot Action

Use to display content (or any other repository) items in a given slot. Specify the slot to use and the repository items to show in it.

Example: `Add Items to Slot ProductSlot named Springtrak Shatterproof Helmet, Arribia Bike Shorts, Springtrak Insulated Water Bottle`

You can also specify an existing targeter instead of a list of content items. In this case, the system uses the criteria in the targeter to define the content items to display.

For more information, see Using Slots.

Scenario editor display name: Add Items to Slot

Action Registry Tag	Value
action name	`fillSlot`
configuration component	`/atg/scenario/configuration/SlotActionConfigurati on`
action execution policy	collective
action error response	continue

The `FillSlot` action has the following parameters:

Parameter	Required	Description

slot	yes	The slot to fill. Can be an instance of `atg.scenario.targeting.RepositoryItemSlot` or `atg.scenario.targeting.Slot`. Example:
		`<nucleus-property>`
		`   <nucleus-path>/atg/registry/Slots`
		`    /QFHomePageSlot</nucleus-path>`
		`</nucleus-property>`

values	no	If the slot is of type `atg.scenario.targeting.Slot` (but not `RepositoryItemSlot`), this parameter holds the array of objects that will be added to the slot.

ids	no	The set of repository items to include in this slot. Example:
		`<array type="java.lang.String[]">`
		`   <constant>/repositories/`
		`    Images/promo-signup.html</constant>`
		`</array>`

targeter	no	The targeter to use (if any) to populate the slot. Example:
		`<nucleus-property>`
		`  <nucleus-path>/atg/`
		`   registry/RepositoryTargeters/`
		`   Images/aggressivePromo</nucleus-path>`
		`</nucleus-property>`

priority	no	The relative priority of display for items in this slot. Note that for slots, 0 is the lowest priority and the default setting.

SlotActionConfiguration Component

The following example shows the properties file for the `SlotActionConfiguration` component, which you can use to define various settings for the `FillSlot` and `EmptySlot` actions (see the next section for information on the `EmptySlot` action).

```
# Version: $Change: 233510 $$DateTime: 2002/03/25 20:52:27 $
$class=atg.scenario.action.SlotActionConfiguration

targetingSourceMap=/atg/targeting/TargetingSourceMap
```

EmptySlot Action

Use to clear items that you previously included in a slot. For more information, see Using Slots.

Example: `Remove items from HomePageSlot named helmet.gif`

Scenario editor display name: Remove Items from Slot

Action Registry Tag	Value
action name	`emptySlot`
configuration component	`/atg/scenario/configuration/SlotActionConfigurati on`
action execution policy	collective
action error response	continue

The `EmptySlot` action has the following parameters:

Parameter	Required	Description

slot	yes	The slot to empty. Example:
		`<nucleus-property>`
		`   <nucleus-path>/atg/registry/Slots`
		`    /QFHomePageSlot</nucleus-path>`
		`</nucleus-property>`

values	no	If the slot is of type `atg.scenario.targeting.Slot` (but not `RepositoryItemSlot`), this parameter holds the array of objects that will be removed from the slot.
ids	no	The set of repository items to remove from this slot.

targeter	no	The targeter to use (if any) to remove items from the slot. Example:
		`<nucleus-property>`
		`  <nucleus-path>/atg/`
		`   registry/RepositoryTargeters/`
		`   Images/aggressivePromo</nucleus-path>`
		`</nucleus-property>`

See the previous section for information on the `SlotActionConfiguration` component.

DisableScenario Action

Use this action to disable a scenario for all users who are passing through it when the action occurs. The action has the same effect as manually disabling a scenario through the ACC; for more information, refer to the *ATG Personalization Guide for Business Users*.

Example: `On Dec 31, 2004 at 12:00:00 AM --- Disable Scenario`

Scenario editor display name: Disable scenario

Action Registry Tag	Value
action name	`disableScenario`
configuration component	`/atg/scenario/configuration/DisableScenarioConfiguration`
action terminal	true
action execution policy	collective
action error response	delete

This action has no parameters.

Note that this action disables only the scenario in which it occurs; you cannot disable a scenario from within another scenario.

The action does not disable the scenario itself, because this behavior requires making a change to the scenario's definition file, which can be done only by the process editor server. Instead, the `DisableScenario` action sends an `atg.dss.DisableProcess` message on a SQL JMS topic to all global servers. The message is ignored by all servers except the process editor server, which makes the following changes to the scenario's definition file:

- The `enabled` attribute is set to false.
- The `modification-time` attribute is set to the current time.
- The `last-modified-by` attribute is set to the string `system` to indicate that the scenario was disabled automatically rather than by an ACC user.
- The `modified-by-server` attribute is set to true. This setting allows the process update and process end messages sent by all servers in response to this action to include appropriate update and end code values.

DisableScenarioConfiguration Component

The following example shows the properties file for the `DisableScenarioConfiguration` component, which you can use to define various settings for the `DisableScenario` action:

```
# Version: $Change: 289519 $$DateTime: 2003/09/26 13:51:21 $
$class = atg.process.action.DisableProcessConfiguration

processMessageSource=/atg/scenario/DSSMessageSource
```

RecordEvent Action

Use to track data for inclusion in reports. Allows you to track a scenario event or action and store information about it in a specified dataset.

Example: `Record event Logs In in dataset /logins.xml`

For more information, see Using Scenario Recorders.

Scenario editor display name: Record Event

Action Registry Tag	Value
action name	`recordEvent`
configuration component	`/atg/scenario/configuration/RecordActionConfiguration`
action execution policy	collective
action error response	continue

The `RecordEvent` action has the following parameters:

Parameter	Required	Description
dataset	yes	The XML file that defines the dataset where you want to record the event. Example: `<constant>/clickthrough.xml</constant>`

RecordActionConfiguration Component

The following example shows the properties file for the `RecordActionConfiguration` component, which you can use to define various settings for the `RecordEvent` and `RecordAuditTrail` actions (see the next section for information on the `RecordAuditTrail` action).

```
# Version: $Change: 233510 $$DateTime: 2002/03/25 20:52:27 $
$class=atg.process.action.RecordActionConfiguration

recorderManager=/atg/reporting/DatasetRecorderManager
```

Record Audit Trail Action

Use this action to track data that monitors scenario activities. For more information, see *Recording Scenario Activity* in the *ATG Personalization Guide for Business Users*.

Action Registry Tag	Value
action name	`recordAuditTrail`
configuration component	`/atg/scenario/configuration/RecordActionConfiguration`
action execution policy	collective
action error response	continue

The `RecordAuditTrail` action has the following parameters:

Parameter	Required	Description
`label`	yes	The label you want to use to identify this data in the dataset. `<constant>FirstTimeBuyer</constant>`
`dataset`	yes	The XML file that defines the dataset where you want to record the event. Example: `<constant>/audittrail.xml</constant>`

See the previous section for information on the `RecordActionConfiguration` component.

Filter Slot Contents Action

Use `filerSlot` to remove items from a slot using a collection filter. When you use this action, you specify the slot name and the component that implements the filtering conditions you want to use.

Action Registry Tag	Value
action name	`filterSlot`
configuration component	`/atg/collections/filter/scenario/FilterActionConfiguration`
action execution policy	collective
action error response	continue

The `filterSlot` action has the following parameters:

Parameter	Required	Description
`slot`	yes	The slot holding objects that will be filtered.
`filter`	yes	The collection filtering component that defines the filtering conditions.

Add Marker To Profile Action

The `AddMarkerToProfile` action creates a marker and adds it to a specified property on the active user profile.

Action Registry Tag	Value
action name	`atg.markers.userprofiling.AddMarkerToProfile`
configuration component	`/atg/markers/userprofiling/ProfileScenarioMarkerConfig`
action execution policy	Individual
action error response	Continue

The `AddMarkerToProfile` action has the following parameters:

Parameter	Required	Description
key	yes	The value given to the key property on the marker being created.

`value`	no	The value given to the `value` property on the marker being created.
`data`	no	The value given to the `data` property on the marker being created.
`duplicationMode`	no	The duplication mode used for the marker being created.

Remove All Markers From Profile Action

The `RemoveAllMarkersFromProfile` action removes all markers from a specified property on the active user profile.

Action Registry Tag	Value
action name	`atg.markers.userprofiling.RemoveAllMarkersFromProfile`
configuration component	`/atg/markers/userprofiling/ProfileScenarioMarkerConfig`
action execution policy	Individual
action error response	Continue

Remove Markers From Profile Action

The `RemoveMarkersFromProfile` action removes, from the active user profile, all markers in a specific property that have the particular `key`, `value`, and `data` values you indicate.

Action Registry Tag	Value
action name	`atg.markers.userprofiling.RemoveMarkersFromProfile`
configuration component	`/atg/markers/userprofiling/ProfileScenarioMarkerConfig`
action execution policy	Individual
action error response	Continue

The `RemoveMarkersFromProfile` action has the following parameters:

Parameter	Required	Description
`key`	yes	The `key` property value that a marker must have in order to be removed from the profile.
`value`	no	The `value` property value that a marker must have in order to be removed from the profile.
`data`	no	The `data` property value that a marker must have in order to be removed from the profile.

Add Stage Reached Action

Use to add a stage in a business process to an object. Specify the name of the business process and the stage. For more information, see the Defining and Tracking Business Processes chapter.

Example: `Add stage reached with process name Shopping Process and stage = AddedToCart`

Action Registry Tag	Value
action name	`addBusinessProcessStage`
configuration component	`/atg/markers/bp/scenario/BusinessProcessScenarioConfiguration`
action execution policy	individual
action error response	continue

The Add Stage Reached action has the following parameters:

Parameter	Required	Description
`business process name`	yes	The name of the business process to which a stage is being added.
`stage`	yes	The name for the stage being added.

Remove Stage Reached Action

Use to remove a business process stage or stages from an object. Specify the name of the business process and the stage or stages to be removed. For more information, see the Defining and Tracking Business Processes chapter.

Example: `Remove stage(s) reached with process name Shopping Process and stage = AddedToCart`

Action Registry Tag	Value
action name	`removeBusinessProcessStage`
configuration component	`/atg/markers/bp/scenario/BusinessProcessScenarioConfiguration`
action execution policy	individual
action error response	continue

The Remove Stage Reached action has the following parameters:

Parameter	Required	Description
`business process name`	yes	The name of the business process to which a stage is being removed.
`stage`	yes	The name for the stage being removed.

E-mail-Related Actions: EmailNotify and SendEmail

The actions described in this section allow you to send a targeted e-mail as part of a scenario. The first, `EmailNotify`, is designed for workflow situations in which you want to alert other users in the organization that a specific scenario event has occurred. The second, `SendEmail`, allows you to send e-mail messages to site visitors, for example as part of a targeted e-mail campaign. With both actions, you specify a JSP or JHTML page that serves as the template for the body of the message.

Both actions are site aware, allowing you to send e-mails or notifications with content that is specific to one or more Web sites supported by your system. To enable this behavior, the classes for these actions include an optional `site` property, which they set on the `TemplateEmailInfo` object. This object in turn passes the value to the `TemplateEmailSender`, which retrieves the site context, creates the message, and uses the site context to customize the template (for example by supplying the applicable Web site name in a “Thank you for registering at…” message). See the examples for each action. For more information about setting the site context, refer to the *ATG Multisite Administration Guide*.

- EmailNotify Action
- SendEmail Action

EmailNotify Action

Sends a specified e-mail message to a given recipient or group of recipients. You can define the recipients as users (people with user profiles in the Profile repository) or e-mail addresses.

Scenario editor display name: Send Notification

Examples:

```
Send notification with path My Web App:/en/email/complaint.jsp to address
mgarcia@example.com

Send notification with path My Web App:/en/email/complaint.jsp to address
mgarcia@example.com with site Budget Pet Supplies

Registers where Site is MyStore > Send notification with path Example
Corps Web App:/en/newMember.jsp to people Amy Stevens with site
Event's Site
```

Note that the site can be set by the preceding event, as shown in the last example above.

This element is designed for situations in which you want to notify a person or group in response to an earlier event in the scenario. For example, you might want to notify your Sales department if a site visitor displays a page showing pricing information about a new product. By contrast, the Send E-mail action (see below) is designed for situations in which you want to send a message as part of an e-mail campaign to a targeted group of site visitors.

Note that the `EmailNotify` action is persisted to the database only if the recipient is a user profile. Actions whose recipients are specified as e-mail addresses are not persisted.

Action Registry Tag	Value
action name	`emailNotify`
configuration component	`/atg/scenario/configuration/EmailNotifyConfiguration`
action execution policy	collective
action error response	delete

The `EmailNotify` action has the following parameters:

Parameter	Required	Description
`recipientIds`	no	The profile IDs of the people to whom this notification is sent.
`recipientAddresses`	no	The e-mail addresses to which this notification is sent. (Used in cases where there are multiple recipients).
`recipientAddress`	no	The e-mail address to which this notification is sent.
`template`	no	The path of the template to send. Corresponds to the ACC option `Send notification with Dynamo path…` Users specify the fully qualified path of the e-mail template to send. If the document is outside the Dynamo document root because it is part of a non-DAS J2EE application, they must specify the application context root manually. This parameter reflects the way that users specified document paths in versions of ATG products before 6.0.0 and exists mostly for compatibility with those versions.
`scenarioPathInfo`	no	The path of the template to send. Corresponds to the ACC option `Send notification with path….` Users specify the path of the e-mail template to send by selecting the template in a document picker dialog box. The path is automatically prepended with either the application context root (if the template is part of a J2EE application) or a flag indicating that the template is located in the Dynamo document root.
`site`	no	(Multisite environments) The Web site to use as the context for the template. Corresponds to the ACC option `Send notification with path… to people… with site….`

EmailNotifyConfiguration Component

The following example shows the properties file for the `EmailNotifyConfiguration` component, which you can use to define various settings for the `EmailNotify` action.

```
# Version: $Change: 246405 $$DateTime: 2002/07/10 17:43:59 $
$class = atg.process.action.EmailNotifyConfiguration

defaultEmailInfo=/atg/scenario/DefaultTemplateEmailInfo
emailSender=/atg/scenario/IndividualEmailSender
profileRepository=/atg/userprofiling/ProfileAdapterRepository
webAppRegistry=/atg/registry/WebApplicationRegistry
```

```
# set to true if you want to render an email with the scenario subject's profile
# but send the email to someone other than the scenario subject
# renderTemplateWithProcessSubject=false
```

By default, the profile you can associate with an `EmailNotify` message is the profile of the person to whom it is sent, not the profile of the person passing through the scenario. This behavior means that any profile-related dynamic content that you include in the body of the message will reference the recipient's profile. However, suppose you want to design a scenario that alerts your customer support department whenever a customer makes an unusually large purchase. You want to send the message to a support rep, but you want to include information from the customer's profile in the body of the message (her login, order number, and order total, for example). To do this, you can set the `renderTemplateWithProcessSubject` property of the `EmailNotifyConfiguration` component to true. This setting uses the profile of the person going through the scenario to resolve any content in the body of the message, but sends the message to the person you specify when you create the `EmailNotify` action in the scenario editor.

Note that you can maintain two versions of the `EmailNotify` action, one with the `renderTemplateWithProcessSubject` property set to true and one with the property set to false.

SendEmail Action

Sends a specified e-mail message to the visitors who have reached this point in the scenario. (See also the EmailNotify action.)

Examples:

```
Send e-mail with path My Web App:/en/email/ordershipped.jsp

Send e-mail with path My Web App:/en/email/welcome.jsp with Site
Gourmet Sausages

Logs In with Site Gourmet Sausages > Send Send e-mail with path
My Web App:/en/email/welcomeback.jsp with Site Event's Site
```

Note that the site can be set by the preceding event, as shown in the last example above.

Action Registry Tag	Value
action name	`sendEmail`
configuration component	`/atg/scenario/configuration/SendEmailConfiguration`
action execution policy	individual
action error response	delete

The SendEmail action has the following parameters:

Parameter	Required	Description
template	no	The path of the template to send. Corresponds to the ACC option Send email with Dynamo path… Users specify the fully qualified path of the e-mail template to send. If the document is outside the Dynamo document root because it is part of a non-DAS J2EE application, they must specify the application context root manually. This parameter reflects the way that users specified document paths in versions of ATG products before 6.0.0 and exists mostly for compatibility with those versions.
scenarioPathInfo	no	The path of the template to send. Corresponds to the ACC option Send email with path…. Users specify the path of the e-mail template to send by selecting the template in a document picker dialog box. The path is automatically prepended with either the application context root (if the template is part of a J2EE application) or a flag indicating that the template is located in the Dynamo document root.
ignoringContactFatigue	no	Allows you to override any daysContactFatigue or hoursContactFatigue settings in the TemplateEmailSender. These settings are used to prevent overexposure to mailings, but they can be overridden for important messages. See Avoiding E-mail Fatigue.
site	no	(Multisite environments) The Web site to use as the context for the template. Corresponds to the ACC option Send email with path … with site…

SendEmail Configuration Component

The following example shows the default properties file for the SendEmailConfiguration component, which you use to define various settings for the SendEmail action.

```
# Version: $Change: 244651 $$DateTime: 2002/06/24 09:19:29 $
$class=atg.scenario.action.SendEmailConfiguration

defaultEmailInfo=/atg/scenario/DefaultTemplateEmailInfo
individualEmailSender=/atg/scenario/IndividualEmailSender
```

```
collectiveEmailSender=/atg/scenario/CollectiveEmailSender
webAppRegistry=/atg/registry/WebApplicationRegistry
```

These properties are described below.

Property	Description
`defaultEmailInfo`	The `TemplateEmailInfo` object that supplies the default information for all emails sent out by this action.
`individualEmailSender`	The `TemplateEmailSender` component used to send email in an individual context.
`collectiveEmailSender`	The `TemplateEmailSender` component used to send email in a collective context.
`webAppRegistry`	The registry of all Web applications that are on the J2EE server where Dynamo is running.

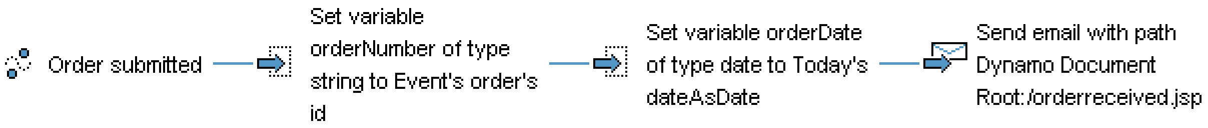
For information on configuring the `TemplateEmailSender` components for use with the `SendEmail` action, refer to Setting Up Scenario E-mail Sender Components.

Identifying Mailings Sent by a Single SendEmail Action

The `batch_exec_id` entry in the `dps_mailing` table contains a unique ID that corresponds to the `SendEmail` action responsible for initiating the mailing. You can use the `batch_exec_id` to group all mailings sent by the same `SendEmail` action, allowing you to identify them easily through the View Grouped display in the ATG Dynamo Server Admin Component Browser.

Accessing Scenario Variables in an E-mail Template

In some cases, you may want to set a variable in a scenario and then reference the variable in the body of an e-mail message that the scenario sends out (using either the `EmailNotify` or the `SendEmail` action). In the following example, you set up a scenario that watches for a customer to make a purchase, tracks the order ID in a variable called `orderNumber` and the date the order was placed in a variable called `orderDate`, and then sends the customer an e-mail that includes both pieces of information. The scenario includes the following elements:



Order submitted

Set variable orderNumber of type string to Event's order's id

Set variable orderDate of type date to Today's dateAsDate

Send email with path Dynamo Document Root:/orderreceived.jsp

You could then add the following lines to the email template `orderreceived.jsp`:

```
<p>Order number
<dsp:valueof param="context.processInstance.contextStrings.orderNumber"/>
<p>was received on
<dsp:valueof param="context.processInstance.contextDates.orderDate"/>
```

(**Note:** The context variable is bound to the template at runtime, so the value of the variable does not appear if you simply preview the template in a Web browser, for example by using the preview features in the ACC. If you want to make sure that the correct values appear, set up a test e-mail address and send the message to it.)

The `valueof` tag references the process execution context, an implementation of the interface `processExecutionContext` that extends the `atg.process` package. The following properties of the context component are available for use in the e-mail template:

- `processInstance`. An instance of an `individualScenario` repository item descriptor. (The definition of this repository item is contained in the default `userProfile.xml` file.)
- `subject`. An instance of the `user` repository item descriptor.
- `messageType`. A String that represents the JMS type of the message that triggered the scenario.
- `individual`. A Boolean property that indicates whether the scenario context is collective (false), which means it applies to all users passing through the scenario, or individual (true), which means it applies only to individual users. Note that, although you can use either value of this boolean property in the body of an e-mail, only individual process instances (see `processInstance` above) will contain a context that can be used to extract the value of a variable set during the scenario.

The page compiler uses Dynamic Beans to reference subproperties of these context elements. In the example above, `contextStrings` and `contextDate` are subproperties of the `processInstance`. Other `processInstance` subproperties that you can reference in this way are as follows:

- `contextBooleans`
- `contextLongs`
- `contextDoubles`

For more information on Dynamic Beans, refer to the *ATG Programming Guide*.

Sending Attachments with Scenario-Based E-mail Messages

You can attach files to an e-mail message that you send with the `EmailNotify` or `SendEmail` scenario action. To do so, edit the associated JSP e-mail template (for example, `DSSJ2EEDemo/j2ee-apps/QuincyFunds/web-app/en/email/newfund.jsp`), adding a line similar to the following:

```
<dsp:setvalue value="C:/meetings/agenda.doc,
C:/meetings/minutes.txt" param="messageAttachments"/>
```

For a JHTML template, add a line similar to the following:

```
<setvalue param="messageAttachments"
value="C:/meetings/agenda.doc,C:/meetings/minutes.txt">
```

19 Using Slots

Slots are containers that you can use to display and manage dynamic items on your Web sites. You use targeting servlet beans to include slots in site pages, and you use scenarios to fill them with content.

Note that you can use targeting servlet beans without slots to display dynamic content. However, slots provide more power and flexibility than targeters. Slots have better caching capabilities, which can make displaying content faster. In addition, because you use scenarios to display items in your slots, you can take advantage of scenario features to help you manage the delivery of dynamic content. For example, scenarios allow you to set up an empty slot that generates its own request for content when a site visitor displays the page. With slots, you can also display content other than repository items.

You can set up slots as components that you register with Nucleus, or you can send them as properties of JMS messages.

The process of creating and setting up a slot has the following steps:

1. Create the slot component.
2. Add an appropriate targeting servlet bean (for example, `TargetingForEach`) to the page or pages where you want the slot to appear. In the `Targeter` property of the bean, specify the slot component. See the *ATG Page Developer's Guide* for information on how to do this.
3. Create a scenario that specifies the content you want to display in the slot and defines the circumstances in which the content appears. See the *ATG Personalization Guide for Business Users* for information on this step.

This chapter contains the following sections:

Creating a Slot as a Nucleus Component

Slots are components of class `atg.scenario.targeting.RepositoryItemSlot` or `atg.scenario.targeting.Slot`. A slot component must have a Nucleus address in the folder `/atg/registry/Slots/`. You can create slot components in two ways:

- By manually creating a `.properties` file
- Through the slot wizard in the ACC

The following is an example of a `.properties` file for a slot component of class `atg.scenario.targeting.RepositoryItemSlot`:

```
$class=atg.scenario.targeting.RepositoryItemSlot
$description=displays fund news to brokers
$scope=session
generation=0
itemDescriptorName=news
maxRenderSize=3
ordering=1
persistent=true
repositoryName=News
retrieval=1
```

For an example of a `.properties` file for a slot of class `atg.scenario.targeting.Slot`, refer to Creating a Slot Component for Objects other than Repository Items.

The following steps show how to create a repository item slot component using the slot wizard in the ACC. Note that you can access this wizard through the Scenarios > Slots window, as described below, or through the Component Editor (Pages and Components > Components by Module > New Component > Scenario Server > Slot). The screens in the wizard are the same regardless of how you access it.

1. Start the ACC.
2. Select Scenarios > Slots.
3. Click New Slot. The slot wizard appears and displays the Specify Slot Name screen.
4. Type a name for this slot. Do not include spaces in the name.
5. Type a brief description of this slot. The description appears in the Description column in the Slots window. You can edit it later if necessary.
6. Click Next, and use the remaining screens in the wizard to configure the slot. The rest of this section describes the properties you can set.

Important: By default, all slots that you create through the slot wizard are instances of the class `atg.scenario.targeting.RepositoryItemSlot`. If you set up your own subclass of `RepositoryItemSlot`, and you want to use the wizard to create a slot from that subclass, you must make a configuration change to the ACC to have it recognize your subclass. Add a string property called `repositoryItemSlotClass` to the `/atg/devtools/ScenarioAgent.properties` file, and set it equal to the name of your class. Your class must be a subclass of `RepositoryItemSlot`.

Content Source

Specify the name of the repository that contains the items you want to show in a slot. (A slot can contain items from only one repository.)

The content source entry in the slot wizard corresponds to the `RepositoryName` property in the slot component's `.properties` file.

In some previous versions of the Scenarios module it was necessary to specify a repository path rather than a name if your application used secure repositories. Specifically, if you specified a secure repository as the content source for the slot, but you stored the content in the corresponding insecure repository, you could not subsequently specify any content items for the slot when you added it to a scenario. Note that it is no longer necessary to specify a repository path in this situation; the Scenarios module can use the `repositoryName` property regardless of whether the slot content is stored in a secure repository or its underlying insecure version. In addition, any slot properties files from older versions that do include a `repositoryPath` property will work without modification.

Content Type

Specify the type of item that you want to display in this slot. When you set up a repository, you define the various item types that it can contain. The setting is important here because a slot can contain items of one type only.

When you add a slot element to a scenario, the ACC presents a menu showing all the items of this type from the repository you specify for the Content Source (see above).

This option corresponds to `itemDescriptorName` (property of type `String)` in the slot component `.properties` file.

Event Generation

The Event Generation setting determines whether an empty slot can issue a request for content items.

If you select Never (the default), the slot displays any items that contributing scenarios have currently defined for it. The slot itself does not issue a request for content items. This type of slot is sometimes referred to as a passive slot. The model for passive slots can be summed up as follows:

1. You create one or more contributing scenarios containing Add Item to Slot elements.
   These elements specify content items that will fill the passive slot in step 3.
2. A visitor displays a page containing the slot.
3. The slot displays the items that are currently defined for it.

If you select When Empty, the slot component can act as a scenario event, which allows it to issue its own requests for items. This type of slot is sometimes referred to as an active slot. When a site visitor views a page containing an active slot, the slot component triggers a scenario that tells the system which content items to display. The model for active slots looks like this:

1. A visitor displays a page containing the active slot.

2. The slot generates a scenario event, triggering one or more scenarios that fill it with items for display.

This behavior means that an empty slot can fill itself with content on demand, which can have advantages for larger sites. For example, suppose you have 50 pages on which you want to display the same slot. With the passive model, you would have to specify a page visit to each page as a triggering event in the contributing scenario, or run the scenario for all pages in the site. With the active model, you do not have to specify pages within the scenario because the slot requests its own items when any of those 50 pages is displayed.

For information on creating an active slot, see *Creating a Scenario for an Active Slot* in the *ATG Personalization Guide for Business Users*.

The Event Generation option corresponds to `generation` (property of type `int`) in the slot component `.properties` file. For Never, specify 1. For When Empty, specify 0.

Scope

All Nucleus components have a scope property that you can set to `global`, `session`, or `request`. In the case of slot components, the way you set the scope affects the results of the item retrieval mode and, indirectly, the order in which visitors see items in the slot.

Scope	Effect
Per Session	Each visitor has a separate list of items, and the order of display persists as the visitor moves from page to page. Session-scoped slots are the most common. This is the default setting.
Global	All site visitors share the same list of items.
Per Request	As with session-scoped slots, each visitor has a separate list of items. However, the order of display does not persist among pages. Each time the visitor requests a page containing the slot, the system starts the cycle of display again.

For more information on the scope property, see the *Nucleus: Organizing JavaBean Components* chapter in the *ATG Programming Guide*.

Item Retrieval

The Item Retrieval setting defines how the system cycles through the content items in the slot.

Item Retrieval	Effect
Destructive (int value 2)	Items display once each. When an item has been displayed, the system removes it from the list of items in the slot. After the system has displayed all items on the list, the slot is empty, unless contributing scenarios add new items.
Rotating (int value 1)	Items display multiple times in rotation. When an item is displayed, the system moves it to the end of the list. The system does not remove items from the list, so the slot never becomes empty. Other contributing scenarios can add content items to the rotation, but to remove any items, you must explicitly clear the slot by adding a Remove Items from Slot element to a scenario.
Static (int value 0)	Similar to destructive mode, except that items are not removed from the list, and the order of the list does not change.

This option corresponds to `retrieval` (property of type `int`) in the slot component .properties file.

Ordering

Use the Ordering setting to change the order in which items appear in the slot. Note that this setting works closely with the At Priority option that you set within the slot element in the scenario.

Ordering	Effect
Show Items in Order They Were Added (int value 0)	The Scenarios module uses the At Priority option from the slot element to define the order. This is the default setting. It corresponds to `sequential` in the `.properties` file.
Show Items in Randomly Shuffled Order (int value 1)	The Scenarios module shuffles the items in the slot and displays them in a random order. It shuffles together only items of the same priority, preserving the priority among groupings. Items are shuffled only once during each instance of the slot's scope. However, if another scenario adds items to the slot during an instance of its display, all the items are shuffled again. This setting corresponds to `random` in the `.properties` file.

This option corresponds to `ordering` (property of type `int`) in the slot component `.properties` file.

For more information on other settings that affect the order of display within a slot, refer to *Using Slot Components* in the *ATG Personalization Guide for Business Users*.

Limit Number of Items Rendered by Slot

Use the Limit Number of Items Rendered by Slot option to define the number of items that can appear in the slot at once. The default value is 2147483647, a deliberately large number that effectively makes the slot show all specified items at the same time. If you changed this value to 1, for example, the Scenarios module would serve only one item to the slot at a time.

Note that the page developer uses a targeting servlet bean to display slots on a page, and the servlet bean also contains settings (the `howMany` and `maxNumber` parameters) that affect the number of items that appear at once. The smallest value defined among the three settings takes precedence. For more information on these parameters, see the *ATG Page Developer's Guide*.

The Limit Number of Items Rendered by Slot option corresponds to `maxRenderSize` (property of type `int`) in the slot component `.properties` file.

Permit Duplicate Content Items

This setting is important if you have multiple scenarios contributing items to the same slot. By default, a content item cannot be added more than once to a single slot. If multiple scenarios do attempt to add an item that already exists in a slot, the item is not duplicated, but its priority is set to the highest of all priorities specified for that item.

Select Permit Duplicate Content Items to change this behavior and allow items to be added more than once. In this case, each instance of the item maintains its own priority.

This option corresponds to `allowDuplicates` (property of type boolean) in the slot component `.properties` file.

Note that this option applies to the number of instances of a given item in the list of items for a slot; it does not affect the Item Retrieval setting, which determines the display cycle for all items in the slot.

Store Slot Persistently in Repository

By default, the system initializes slots every time their corresponding scenario is triggered. For example, assume you have a scenario that watches for a visitor to log into the site, and then displays a “Welcome back!” image. Every time the visitor logs in during the period for which the scenario is active, the slot is initialized and the content is displayed.

In some circumstances, however, you might have a scenario that is triggered once only. For example, you might have a scenario that watches for a visitor to register and then populates a slot with an image containing a link to a promotion. The scenario is intended to wait for the visitor to click the link and then remove the link from the slot. Registration happens only once, so, in this case, the slot is only initialized once. However, the visitor may not click on the link during the first session after he or she registers. Unfortunately the link is displayed during the first session only because the scenario itself is not triggered for each session.

To change this behavior, check the Store Slot Persistently in Repository option. The system stores information about the slot in the profile repository (specifically, in the `slotInstances` property) so that its contents can be displayed to a visitor across multiple sessions.

This option corresponds to the `persistent` property in the slot component .properties file.

Important: As described above, persistent slots are particularly useful when you want to display content from session to session until a specific event occurs. However, persistent slots can have a negative effect on performance because their contents must be maintained by interfacing to the profile repository. For this reason, use them very sparingly and only when persistence is absolutely required.

Do not use this option for globally scoped slots. Persistent slots are associated with a specific user, and the ATG Scenarios code expects them to be session scoped. Array index exceptions occur if you try to display a page that contains a persistent slot that is configured as global.

Creating a Slot Component for Objects other than Repository Items

Slot components can contain repository items, as described previously, or they can contain objects of types String, Date, Long, or Double. Slots that are designed to display these objects are of class `atg.scenario.targeting.Slot`. The following `.properties` file shows a component instance of this class:

```
$class=atg.scenario.targeting.Slot
$description=displays dates
$scope=session
generation=0
maxRenderSize=2
ordering=1
retrieval=1
valueType=java.util.Date
```

The `valueType` property specifies the Class object of the type of values that the slot will display.

The process for creating a slot of this type is almost identical to the process for creating a repository item slot; see the previous section for details. The exceptions are as follows:

- Options specific to repository item slots, for example Content Source, do not appear in the slot wizard.
- `atg.scenario.targeting.Slot` slots cannot be persisted across visitor sessions, so the Store Slot Persistently in Repository option does not appear in the slot wizard.

Editing Slot Components

The Scenarios > Slots window in the ACC gives you a convenient way to edit the settings of any existing slot. From the list of slots in the left pane of the window, select the slot whose settings you want to change. Save the changes when they are complete.

You can also use the Slots window to locate the pages and scenarios that reference a given slot. This feature can be very useful for debugging purposes. To use it, select the slot in the list in the left pane of the Scenarios > Slots window, then display either the Pages or Scenarios tab as needed.

Note: If you create a slot through the Pages and Components area of the ACC, you must restart the ACC to make the slot appear in the list in the Scenarios > Slots window.

Deleting Slot Components

To delete a slot component, use the Pages and Components > Components by Path option in the ACC. Slot components have the path `/atg/registry/Slots`. Locate the slot component to delete and then select Edit > Delete.

Creating a Slot as a Property of a JMS Message

As an alternative to registering slots as Nucleus components, you can create them on the fly and send them as properties of JMS event messages. In the scenario editor, you include an Add Items to Slot element that is filled by extracting the slot property from the preceding event. Note that you can also remove slot items in this way.

This type of slot is temporary and cannot be persisted across sessions. In addition, this technique for creating slots cannot be used for slots that contain repository items (`RepositoryItemSlot`); it can be used only for slots that contain Strings, Dates, Longs, or Doubles (class `Slot`).

The ACC cannot detect the type of content a slot holds if the slot is created as part of a JMS message; for this reason, the person who sets up the slot element within the scenario editor must specify the type of content that the slot is designed to display.

For more information on creating scenarios that include slots created as JMS messages, refer to the *ATG Personalization Guide for Business Users*.

Using Slots in a Multisite Environment

In a multisite environment, all Web sites supported by your ATG instance typically use the same pages (for example, all sites have the same login page, possibly called `login.jsp`). The pages contain code that controls the display of content, making it appropriate for each site. Be aware that any slot you add to a page will therefore display the same content on all your sites unless you configure it otherwise.

There are several methods for configuring slots to display site-specific content.

- One efficient way is to use a single slot component on all sites and add content to it through a targeter that has site-specific rules. Create a scenario that uses the site-specific targeter to fill the slot. For a description of site-specific targeters, refer to the *ATG Business Control Center User's Guide.* For more information on creating scenarios, refer to the *ATG Personalization Guide for Business Users*.

- You can create a different slot for each site and write page logic that checks the site context and determines which slot to render for the current site. This approach is less efficient than the single slot method if you have many sites to maintain.

Note: For slots that are filled by a multisite targeter, be aware of the following behavior with session-scoped slots. A session-scoped slot is filled with the items on the current site when the slot is first requested. Subsequent requests will use the items that were filled on the first request, regardless of the current site. The following situation is therefore possible: a user logs into Site A and views items from the current site via a session-scoped slot. When the user switches to Site B, the slot still displays items from Site A.

For more information on site context and working with pages in a multisite environment, refer to the *ATG Multisite Administration Guide.*

20 Using Scenario Recorders

The recorders that are provided with the Scenarios module give you a way to collect the visitor- and site-related data that you can use for analysis and display in business reports.

Recorders are based on scenario events. (For more information on scenarios, see *Creating Scenarios* in the *ATG Personalization Guide for Business Users*.) Events are JavaBeans, and you can collect data from any JavaBean property that represents a single entry. For example, you can collect data from properties that contain strings or integers but you cannot collect arrays, sets, or hashes.

Each recorder has four required components, as follows:

- A Data Collection object that collects the data before it is logged.
- An mapper that maps the columns in the SQL database to Java data-types.
- A dataset, which bundles groups of data together and filters it based upon sampling criteria.
- A scenario that personalizes the data collection, defining the circumstances under which it takes place. (Note: read-only reports do not require a scenario.)

The Scenarios module provides several standard recorders for performing common data collection tasks. For example, it provides a Page Visit recorder that tracks a range of activities related to personalized site visits. The rest of this chapter describes the process of creating your own recorders.

Creating a Custom Recorder

In this example, a company called SmallCorp has very limited space in its database and wants to track only the names of the pages that customers visit and the time that the visits occur. In addition, SmallCorp wants to sample this data to save space in its database.

SmallCorp's database administrator has already set up the database columns that will hold the recorded data. The column names and descriptions are shown below.

Table: `Dynamo_Recorder`

Column	Description
`datasetId`	The dataset name that records the data. Different datasets can share the same mapper. It is this dataset identification that allows reporting to show actual totals in coordination with sampling.
`PageVisited`	The name of the page that is visited by the customer.
`timestamp`	The date and time when the page was visited.

SmallCorp would take the following steps to create the custom recorder:

1. Create a new Data Collection object of `class=$atg.reporting.dataset.DatasetSQLLogger`.
2. Create a new mapper to listen for the Data Collection event.
3. Create a new dataset to reference the new mapper.
4. Create a new scenario to record data to this table using this dataset.

SmallCorp can then use the data that the recorder collected in reports. The following sections show how to perform the steps outlined above.

Note: If you plan to add new tables to your database for the data you want to log, perform this step before you create the recorder.

Creating a New Data Collection Object

Create an appropriate component of class `atg.reporting.dataset.DatasetSQLLogger`. In this example, SmallCorp would create an `/atg/reporting/dataset/RecordVisitsSQLLogger` component as shown below:

```
# /atg/reporting/dataset/RecordVisitsSQLLogger
#Tue Jul 18 11:37:45 EDT 2000
$class=atg.reporting.dataset.DatasetSQLLogger
schedule=every\ 1\ minute\ in\ 1\ minute
scheduler=/atg/dynamo/service/Scheduler
```

Place the component `.properties` file in the `<ATG10dir>/home/localconfig/atg/reporting/dataset` directory.

Important: Set the scope property for this component to `global`.

Creating the New Mapper

Manually create a new mapper (an XML file) that describes the table and the listener for the table. Place the file in the `<ATG10dir>/home/localconfig/atg/registry/data/mappers` directory. The file that SmallCorp creates in this example (called `dynamo_recorder.xml`) is shown below:

```
<mapper >
  <registry-descriptor>
     <id>dynamo_recorder.xml</id>
     <displayname>Example Page Visit SQL Mapper</displayname>
     <description>Example Page Visit Mapper</description>
  </registry-descriptor>
  <input-filter context="jms">
     <value>atg.dps.PageVisit</value>
  </input-filter>
  <data-listener>/atg/reporting/dataset/RecordVisitsSQLLogger</data-listener>
  <database>
     <transaction-manager>/atg/dynamo/transaction/TransactionManager
      </transaction-manager>
     <datasource>dynamo:/atg/dynamo/service/jdbc/JTDataSource</datasource>
     <table>
        <name>DYNAMO_RECORDER</name>
        <display-name>Refined Page Visits</display-name>

            <mappings>
                <dataset-mapping>
                     <display-name>Visitor</display-name>
                     <column>datasetId</column>
                     <type>java.lang.String</type>
                     <property>datasetId</property>
                </dataset-mapping>

                <timestamp-mapping>
                     <display-name>Timestamp</display-name>
                     <column>timestamp</column>
                     <type>java.sql.Timestamp</type>
                     <property>timestamp</property>
                </timestamp-mapping>

                <property-mapping>
                     <display-name>Path Qa</display-name>
                     <column>PageVisited</column>
                     <type>java.lang.String</type>
                     <property>context.message.path</property>
                </property-mapping>

            </mappings>
     </table>
  </database>
</mapper>
```

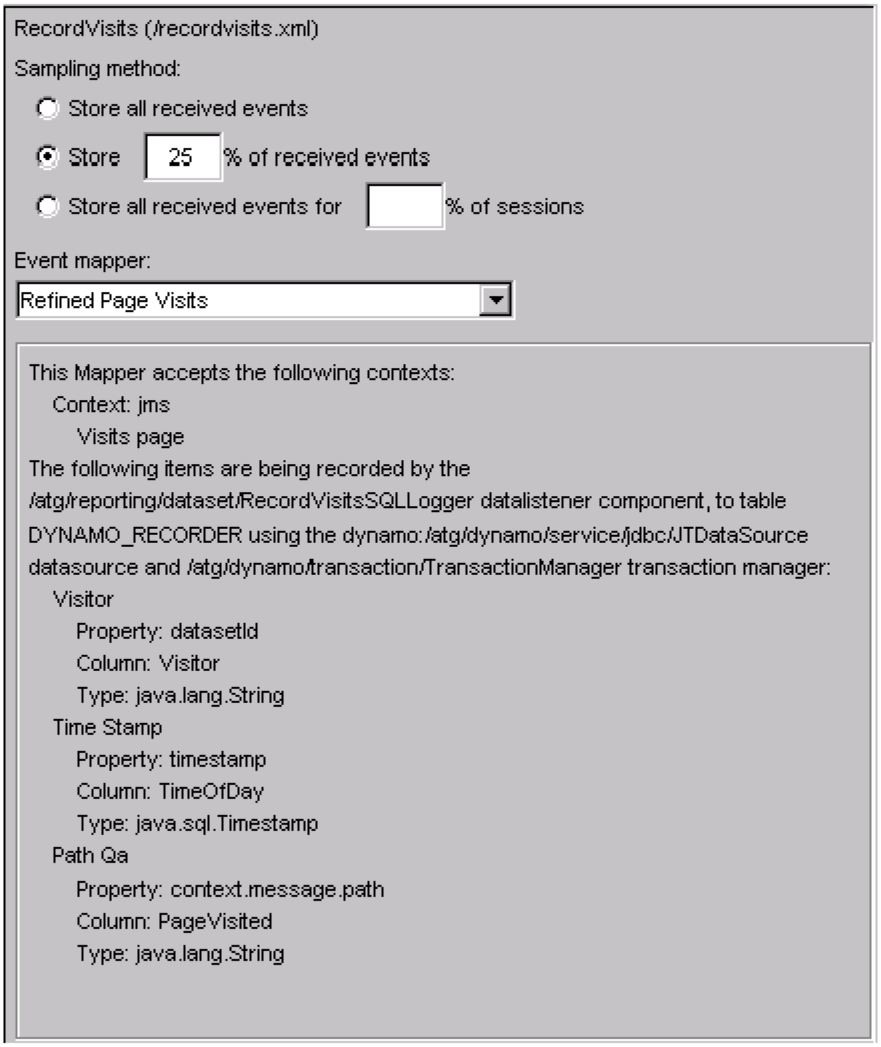
Optionally, you can turn on mapper validation, which checks on application startup for the existence of the custom mappers identified in your code. To turn on validation:

1. In the `atg.reporting.DatasetRecorderManager` component, set the `validateOnStart` property to true.
2. Add the `DatasetRecorderManager` to the `initial.properties` file as one of the `InitialServices` entries.

Creating a New Dataset for a Custom Recorder

The next step in the process of setting up a custom recorder is to create a dataset that references the new mapper and specifies the sampling rate for recording the data.

The dataset that SmallCorp creates looks like this:



RecordVisits (/recordvisits.xml)

Sampling method:

○ Store all received events

● Store 25 % of received events

○ Store all received events for % of sessions

Event mapper:

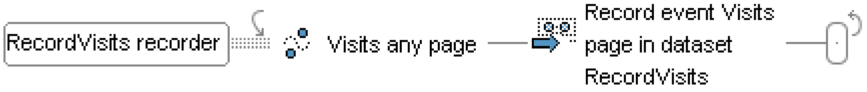
Refined Page Visits

This Mapper accepts the following contexts:
  Context: jms
    Visits page
The following items are being recorded by the /atg/reporting/dataset/RecordVisitsSQLLogger datalistener component, to table DYNAMO_RECORDER using the dynamo:/atg/dynamo/service/jdbc/JTDataSource datasource and /atg/dynamo/transaction/TransactionManager transaction manager:
  Visitor
    Property: datasetId
    Column: Visitor
    Type: java.lang.String
  Time Stamp
    Property: timestamp
    Column: TimeOfDay
    Type: java.sql.Timestamp
  Path Qa
    Property: context.message.path
    Column: PageVisited
    Type: java.lang.String

Note that the name of the mapper as it appears in the Event Mapper list is defined by the <`display-name`> tag in the mapper file.

Creating a Scenario for a Custom Recorder

Using the ACC, create and enable a new scenario to record events to this dataset. The following example shows the scenario that SmallCorp would create for its new recorder:



This scenario tells the system to watch for any page visit event and then record it to the specified dataset.

For detailed information on scenarios, see the *ATG Personalization Guide for Business Users*.

Mapper XML Definition Language

This section describes the XML syntax you can include in event mappers that you create for reports in the Scenarios module.

<mapper>

The `<mapper>` tag (required) is the root tag for defining a mapper object.

It can contain the following child tags:

- `<registry-descriptor>` (required)
- `<input-filter>` (required)
- `<data-listener>` (required)
- `<database>` (required)

<registry-descriptor>

The `<registry-descriptor>` tag (required) is a wrapper for information that identifies this event mapper to the registry.

It can contain the following child tags:

- `<id>` (required). Unique ID for the mapper (used for external reference).
- `<display-name>` (optional). Text that defines the display name of the mapper as it will appear in the ACC (the text appears in the Event Mapper field in the Dataset window).
- `<description>` (optional).

<input-filter>

The <input-filter> tag (required) has one attribute, context, that determines the purpose of the event mapper.

The context attribute (required) can be set to one of the following three values:

- "jms" – use for mappers that you want to accept data from JMS messages.
- "read-only" – use for mappers that do not record data through scenarios but are used instead for reporting on existing repository data.
- "audit" – use for mappers that you want to accept data from audit trail events that you have included in recorder scenarios. For more information, refer to *Creating an Audit Trail* in the *ATG Personalization Guide for Business Users*.

The <input-filter> tag has the following child tags:

- <value> – Required only if you set the context attribute to "jms". It identifies the JMS message to use to collect data. For "read-only" and "audit" context attribute settings, no <value> tag is required.

Example:

```
<input-filter context="jms">
  <value>atg.dps.ProfilePropertyUpdate</value>
</input-filter>
```

<data-listener>

The <data-listener> tag (required) specifies the Nucleus path for the component of class atg.reporting.dataset.DatasetSQLLogger that you created to collect data (see Creating a New Data Collection Object for more information).

Example:

```
<data-listener>/atg/reporting/dataset/ProfileUpdateLoggerQueue
</data-listener>
```

<database>

The <database> tag (required) is a wrapper that describes the destination database for the data you want to record.

It can contain the following child tags:

- <transaction-manager> (required)
- <datasource> (required)
- <table> (required)

<transaction-manager>

The `<transaction-manager>` tag (required) specifies the Nucleus address of the Transaction Manager service to use to track transaction activity for this event mapper.

Example:

```
<transaction-manager>/atg/dynamo/transaction/TransactionManager
</transaction-manager>
```

For more information, see *Transaction Management* in the *ATG Programming Guide*.

<datasource>

The `<datasource>` tag (required) specifies the JDBC connection pool to use to create connections to the database where this event mapper stores the data it records. For more information, refer to the *ATG Installation and Configuration Guide*.

Example:

```
<datasource>dynamo:/atg/dynamo/jdbc/JTDataSource</datasource>
```

<table>

The `<table>` wrapper (required) defines the database table that will be used to store the data that this mapper records. It can contain the following child tags:

- `<name>` (required). The name of the table to use.
- `<display-name>` (optional). Text that defines the display name of the table as it will appear in the ACC (the text appears in the description of the mapper in the Dataset window).
- `<mappings>` (required) . The database columns to which you want to map the properties you are recording.

<mappings>

The `<mappings>` tag is a wrapper that contains one or more dataset mappings that map the property or properties you are recording to appropriate database columns.

It can contain the following child tags:

- `<dataset-mapping>`
- `<timestamp-mapping>`
- `<property-mapping>`
- `<repository-item-mapping>`

<dataset-mapping>

The `<dataset-mapping>` tag contains information that specifies the column to use to identify this dataset within the database. This mapping is required.

This tag can have the following child tags:

- <display-name> (optional) Text that defines the name of this property in the Choose the Axis Items dropdown list (see above).
- <column> (required) The name of the database column to which you want to record this data.
- <type> (required) The Java type of the property to record to the specified column.
- <property> (required) The data to record.

Example:

```
<dataset-mapping hidden="true">
  <display-name>Dataset Id</display-name>
  <column>id</column>
  <type>java.lang.String</type>
  <property>datasetId</property>
</dataset-mapping>
```

The hidden attribute is no longer used by ATG products.

<timestamp-mapping>

The <timestamp-mapping> tag specifies the column to use within this dataset to store information about the time that data is recorded. This mapping is required.

The <timestamp-mapping> tag can have the following child tags:

- <display-name> (optional). <display-name> (optional) Text that defines the name of this property in the Choose the Axis Items dropdown list (see above).
- <column> (required). The name of the database column to which you want to record this data.
- <type> (required). Specifies the Java type for this data. Note that java.sql.Timestamp is the only Java type you can use in a <timestamp-mapping> tag for an Scenarios module event mapper.
- <property> (required). The data to record.

Example:

```
<timestamp-mapping hidden="true">
  <display-name>Time Stamp</display-name>
  <column>clocktime</column>
  <type>java.sql.Timestamp</type>
  <property>timestamp</property>
</timestamp-mapping>
```

The hidden attribute is no longer used by ATG products.

<property-mapping>

The <property-mapping> tag specifies the column to use to store the given piece of data as well as the actual data to record.

This tag can have the following child tags:

- <display-name> (optional). Text that defines the name of this property in the Choose the Axis Items dropdown list (see above).
- <column> (required). The name of the database column to which you want to record this data.
- <type> (required). The Java type of the data:
    - `java.lang.String`
    - `java.lang.Number` (and its subclasses)
    - `java.sql.Timestamp`
- <property> (required). The data to record. Refers to the `atg.process.ProcessExecutionContext` object, which is the object that gets passed to the mapper. Use the following attributes to specify the object and the property you are recording:
    - `context.request` – a property of the current HTTP request object if the scenario segment is being executed in the context of an HTTP request. Example: `<property>context.request.session.id</property>`
    - `context.response` – a property of the current HTTP response, if the scenario segment is being executed in the context of an HTTP request.

      Example: `<property>context.response.SC_NOT_FOUND</property>`
    - `context.message` – a property of the JMS message that triggered the recorder scenario. Example: `<property>context.message.reportingOldValues </property>`

      Note that many of the JMS messages that come with the ATG system contain properties designed to make reporting easier. These properties are typically String representations of data that could not normally be recorded because of `type` restrictions for Scenarios module data collection (`java.lang.String`, `java.lang.Number`, and `java.sql.Timestamp` only). The property shown above, `reportingOldValues`, is an example; it allows you to record a List property, `oldValues`, as a String.

      Usually the names of these properties start with `reporting`, which gives you an easy way to identify them in the API.
    - `context.profile` – a profile property of the user going through the scenario segment if the segment is being executed in the context of an individual user. Example: `<property>context.profile.repositoryid</property>`.
    - `context.label` – for use in event mappers where you set the `<input-filter>` tag `context` attribute to `"audit"`. The label is a String that is mapped

by default to the `label` column in the `dss_audit_trail` table. The data that is recorded is the Label value that you specify within the Record Audit Trail scenario action.
Example: `<property>context.label</property>`

Example 1:

```
<property-mapping>
  <display-name>Old Values</display-name>
  <column>oldvalues</column>
  <type>java.lang.String</type>
  <property>context.message.reportingOldValues</property>
</property-mapping>
```

Example 2:

```
<property-mapping>
  <display-name>Label</display-name>
  <column>label</column>
  <type>java.lang.String</type>
  <property>context.label</property>
</property-mapping>
```

The `hidden` attribute is no longer used by ATG products.

<repository-item-mapping>

The `<repository-item-mapping>` tag specifies the column to use to store the given piece of data as well as the actual data to record. Use this tag for data that is stored as a repository item (for example, profile data).

This tag can have the following child tags:

- `<display-name>` (optional). Text that defines the name of this property in the Choose the Axis Items dropdown list (see above).
- `<column>` (required). The name of the database column to which you want to record this data.
- `<type>` (required). The Java type of the data.
- `<property>` (required). The data to record. See the description of the `<property-mapping>` tag for information about the `context` attribute.
- `<component>` (required). The Nucleus component that represents the repository that stores the data. Example:
  `<component>/atg/userprofiling/ProfileAdapterRepository</component>`

Example:

```
<repository-item-mapping>
  <display-name>Profile Id</display-name>
  <column>profileid</column>
  <type>java.lang.String</type>
  <property>context.profile.repositoryid</property>
  <component>/atg/userprofiling/ProfileAdapterRepository</component>
</repository-item-mapping>
```

The `hidden` attribute is no longer used by ATG products.

Sample Mapper XML File

The following example shows the Profile Update SQL Mapper (`<ATG10dir>/DSS/config/atg/registry/data/mappers/profileupdatemapper.xml`), which is included by default with the Scenarios module. This mapper maps the properties of a Personalization module `ProfileUpdateEvent` to the specified database table.

For additional examples, look at the mapper XML files located in the `<ATG10dir>/DSS/config/atg/registry/data/mappers` directory.

```
<mapper>

  <registry-descriptor>
    <id>/profileupdatemapper.xml</id>
    <display-name>Profile Update SQL Mapper</display-name>
    <description> This maps the properties of a DPS ProfileUpdateEvent
      to a table in a database
    </description>
  </registry-descriptor>

  <input-filter context="jms">
    <value>atg.dps.ProfileUpdate</value>
  </input-filter>

  <data-listener>/atg/reporting/dataset/ProfileUpdateLoggerQueue</data-listener>

  <database>
    <transaction-manager>/atg/dynamo/transaction/TransactionManager
      </transaction-manager>
    <datasource>dynamo:/atg/dynamo/service/jdbc/JTDataSource</datasource>
    <table>
      <name>dss_dps_update</name>
      <display-name>
        DPS Profile Update Event Dataset Table
      </display-name>

      <mappings>
```

```
<dataset-mapping hidden="true">
  <display-name>Dataset Id</display-name>
  <column>id</column>
  <type>java.lang.String</type>
  <property>datasetId</property>
</dataset-mapping>

<timestamp-mapping hidden="true">
  <display-name>Time Stamp</display-name>
  <column>clocktime</column>
  <type>java.sql.Timestamp</type>
  <property>timestamp</property>
</timestamp-mapping>

<property-mapping>
  <display-name>Session Id</display-name>
  <column>sessionid</column>
  <type>java.lang.String</type>
  <property>context.request.session.id</property>
</property-mapping>

<property-mapping>
  <display-name>Changed Properties</display-name>
  <column>changedproperties</column>
  <type>java.lang.String</type>
  <property>context.message.reportingChangedProperties</property>
</property-mapping>

<property-mapping>
  <display-name>Old Values</display-name>
  <column>oldvalues</column>
  <type>java.lang.String</type>
  <property>context.message.reportingOldValues</property>
</property-mapping>

<property-mapping>
  <display-name>New Values</display-name>
  <column>newvalues</column>
  <type>java.lang.String</type>
  <property>context.message.reportingNewValues</property>
</property-mapping>

<repository-item-mapping>
  <display-name>Profile Id</display-name>
  <column>profileid</column>
  <type>java.lang.String</type>
  <property>context.profile.repositoryid</property>
  <component>/atg/userprofiling/ProfileAdapterRepository</component>
</repository-item-mapping>
```

```
      </mappings>
    </table>
  </database>
</mapper>
```

21 Adding Custom Events, Actions, and Conditions to Scenarios

When business managers create scenarios in the ACC, they build the scenarios from a range of elements that include events and actions. Events are instances of site visitor behavior that the business manager wants the scenario to watch for; for example, he or she can create a scenario that watches for a visitor to register at the site or to go to a specific page. “Registers” and “Visits page” are both examples of scenario events. Actions, on the other hand, are instances of system behavior - often what the system does in response to an event. For example, a scenario might watch for visitors to register, and then send an e-mail. “Send E-mail” is an example of a scenario action. Conditions are filters that qualify the previous event element in a scenario; for example, you could follow the event “Views any item” with the condition “Browser’s type is Netscape Navigator.”

If the events, actions, and conditions that come with the Scenarios module do not meet your requirements, you can add your own. This section explains the process of creating custom elements so that they appear in the ACC for your business managers to use:

Note: As described in this chapter, scenario events are instances of JMS messages, and the procedure for creating a custom event therefore involves editing various files that configure the sending and receiving of messages. The examples in this chapter assume your application uses SQL JMS as the JMS provider. For information on the code to add to these files if you use other JMS providers (for example, for applications running on IBM WebSphere), refer to *Dynamo Messaging System* in the *ATG Programming Guide*.

Adding Custom Events

By default, all the standard JMS messages for DAF and the Personalization module appear as event elements in the Scenario area of the ACC. (If your product suite includes ATG Commerce, additional commerce-related events appear.) If the standard events do not meet all your company’s needs, you can configure the Scenario Manager to recognize custom events. To add your custom event (in other words, your custom JMS message) to the Scenarios module, do the following:

1. Create a JavaBean class that represents your custom JMS message.

2. Create a Nucleus component implementing `MessageSource` that will generate your custom JMS message.
3. Add your custom event to the scenario event registry.
4. Add your message and message source to the appropriate Dynamo Message System (DMS) configuration file.
5. Configure the Scenario Manager to receive your message.

Example: Adding Clickthrough Tracking To Your Application

This example shows how to add an event to the system that will be fired whenever a user clicks on a particular link in a Web page. For instance, there might be several different links on a site that all lead to the same page, but you want to trigger different scenario actions depending on which of these links is clicked. Alternatively, you might simply want to record all clickthrough events to a dataset for later analysis.

To do this, we will create a JMS message called `test.scenario.LinkMessage` to represent a clickthrough. The message contains the logical name of the location that was clicked. Scenarios can then be created which intercept these clickthrough events and detect their locations. We will also create a Nucleus component named `/test/scenario/LinkMessageSource` that can be instructed to fire a `LinkMessage` by using a special <A> tag to set its location property. This permits any link in a page to selectively fire messages, as in this JHTML example:

```
<A HREF="PromotionDetail.jhtml"
   BEAN="/test/scenario/LinkMessageSource.location"
   VALUE="Teaser">See details now!</A>
```

The example above provides a link to `PromotionDetail.jhtml` that, when clicked, fires a `LinkMessage` that carries the location `Teaser`.

Here is the same example in JSP code:

```
<dsp:a bean="/test/scenario/LinkMessageSource.location"
       value="Teaser" href="PromotionDetail.jsp">See details now!</dsp:a>
```

Note: This section was originally written for version 5.0 of the Scenarios module. A similar clickthrough event has since been added to the product as a standard scenario element. We have kept the example, however, because the basic procedure you follow is the same regardless of the event you are adding.

Creating the LinkMessage class

A JMS object message class is very simple: it is a serializable JavaBean that carries the message properties. Our `LinkMessage` class will have only one property: `location`.

```
package test.scenario;

/**
           * JMS message that conveys data about a link clickthrough event.
```

```
 *
 **/
public class LinkMessage implements java.io.Serializable
{
  //----------------------------------------
  // CONSTRUCTORS
  //----------------------------------------
  public LinkMessage(String pLocation)
  {
    setLocation(pLocation);
  }
  //----------------------------------------
  // PROPERTY ACCESSORS
  //----------------------------------------
  String mLocation;
  //-------------------------------------
  /**
   * Sets the logical name of the clickthrough location.
   **/
  public void setLocation(String pLocation) {
    mLocation = pLocation;
  }
  //-------------------------------------
  /**
   * Returns the logical name of the clickthrough location.
   **/
  public String getLocation() {
    return mLocation;
  }
}
```

Creating the LinkMessageSource Component

The next step is to create a `MessageSource` implementation that fires our new message and can be triggered by setting its `location` property from a JSP or a JHTML page. It will use a fixed Patch Bay port named `Link` and a message type of `test.scenario.Link`. (For more information, see *Patch Bay: Configuring the Dynamo Message System* in the *ATG Programming Guide*.)

```
package test.scenario;
import javax.jms.*;
import javax.transaction.*;
import java.io.Serializable;
import atg.dms.patchbay.MessageSource;
import atg.dms.patchbay.MessageSourceContext;
import atg.nucleus.GenericService;

/**
 * A Nucleus component used for firing LinkMessages.
 */
public
class LinkMessageSource extends GenericService implements MessageSource
{
  //-------------------------------------
  // Constants
  //-------------------------------------
  public static final String PORT_NAME = "Link";
  public static final String MESSAGE_TYPE = "test.scenario.Link";
  //-------------------------------------
  // Member Variables
  //-------------------------------------
  /** Message source context for message creation */
  MessageSourceContext mMessageSourceContext;
  /** Flag gating whether this message source is active or not. */
  boolean mSendingMessages = false;

  //-------------------------------------
  // Properties
  //-------------------------------------
  public MessageSourceContext getMessageSourceContext ()
  {
    return mMessageSourceContext;
  }

  //----------------------------------------
  // MessageSource Interface Implementation
  //----------------------------------------
  public void setMessageSourceContext (MessageSourceContext pContext)
  {
    mMessageSourceContext = pContext;
  }
  public void startMessageSource ()
  {
    mSendingMessages = true;
  }
  public void stopMessageSource ()
  {
    mSendingMessages = false;
  }
  //----------------------------------------
  // Public Methods
  //----------------------------------------
  //----------------------------------------
  /**
   * Called by Dynamo when a link of the form <A HREF="..."
   * BEAN="/test/scenario/LinkMessageSource.location" VALUE="...">>
   * is clicked.
   *
   * @param pLocation the symbolic name of the link location being
   * fired, as given in the value tag above.
   */
  public void setLocation (String pLocation)
  {
    fireLinkMessage(pLocation);
  }

  /**
   * Fires a LinkMessage reporting a clickthrough location.
   */
  public void fireLinkMessage(String pLocation)
  {
    try {
      ObjectMessage message;
      message = mMessageSourceContext.createObjectMessage(PORT_NAME);
      message.setJMSType(MESSAGE_TYPE);
      message.setObject(new LinkMessage(pLocation));
      mMessageSourceContext.sendMessage(PORT_NAME, message);
    }
    catch (JMSException jmse) {
      if (isLoggingError())
        logError(jmse);
    }
  }
}
```

We also need to configure the above class as a Nucleus component by creating a file named `test/scenario/LinkMessageSource.properties` in the `localconfig` directory. There are no properties to configure; just specify the component's class:

```
$class=test.scenario.LinkMessageSource
```

Adding Your Message to the Appropriate DMS Configuration File

There are two Dynamo Message System configuration files that you may have to edit as part of setting up a custom event:

- The `dynamoMessagingSystem.xml` file, located in your configuration path at `/atg/dynamo/messaging/dynamoMessagingSystem.xml`. Each product in the ATG suite adds its own message definitions to this file. The default contents of the Scenarios module version of this file are shown below.
- The `dynamoMessagingSystemDSSGlobal.xml` file, which is specifically used to define message source and message sinks for global scenario events. The default file is located in your configuration path at `/atg/dynamo/messaging/dynamoMessagingSystemDSSGlobal.xml`. Note that the name of the file is specified by the `globalConfigurationFile` property of the `ScenarioManager` component.

For information on how the system combines the contents of these files, refer to the Dynamo Message System chapter in the *ATG Programming Guide*.

By default, the Scenarios module adds the following to the `dynamoMessagingSystem.xml` file:

- `message-registry` definitions for all JMS messages that come with the Scenarios module, including global JMS messages.
- `message-source` definitions for the following ports:
  - `IndividualTimers`
  - `SlotItemRequest` port
- `message-sink` definitions for the following ports:
  - `IndividualEvents`

`ProcessUpdates` The default `dynamoMessagingSystemDSSGlobal.xml` file contains the following:

- `message-source` definitions for the following ports:
  - `CollectiveTimers`
  - `BatchTimers`
  - `ProcessUpdates`
  - `SegmentStartTimers`
- `message-sink` definitions for the following ports:
  - `GlobalEvents`
  - `IndividualTimers`
  - `CollectiveTimers`
  - `BatchTimers`
  - `SegmentStartTimers`

You can extend either configuration by creating a file with the same name in your `localconfig` directory or within your custom application module.

If you are adding a global event, do the following:

- Add a `message-source` definition for this event. If you want this event to be listened to by the Scenario Manager, you must add the definition to the `dynamoMessagingSystemDSSGlobal.xml` file (see Configuring the Message Source). If you want the event to be listened to by other message sinks (for example, custom applications that you have set up to use the contents of Scenarios module messages), add the definition to the `dynamoMessagingSystem.xml` file.
- Add a `message-registry` definition for this event to the `dynamoMessagingSystem.xml` file (see Configuring the Scenario Manager to Receive Your Message).

For an individual event, add both the `message-source` and `message-registry` definitions to the `dynamoMessagingSystem.xml` file.

Configuring the Message Source

As part of adding your custom message, you have to create and configure a message source (using the <message-source> tag in the DMS configuration file), which will be responsible for generating your messages. The <output-port> specifies the destination topic or queue where your message will be placed. For more information, refer to the Dynamo Message System chapter in the *ATG Programming Guide*.

Both global and individual event messages can be sent to queues or to topics as appropriate. However, global events to which you want the Scenario Manager to listen must be sent to queues.

Typically you make up a new name to go with a new message type that you create. Ours will be called localdms:/local/test/Link.

The following example shows how to configure the LinkMessageSource component to send messages to the destination topic localdms:/local/test/Link. If you are creating a custom global event, and you want to use the Scenario Manager as the message sink for this event, add the code to your extension of the file dynamoMessagingSystemDSSGlobal.xml. If you are creating a custom individual event, or a global event for use by message sinks other than the Scenario Manager, add the code to your extension of the dynamoMessagingSystem.xml file. (See Adding Your Message to the Appropriate DMS Configuration File for more information.)

```
<message-source>
  <nucleus-name>
    /test/scenario/LinkMessageSource
  </nucleus-name>
  <output-port>
    <port-name>
      Link
    </port-name>
    <output-destination>
      <provider-name>
        local
      </provider-name>
      <destination-name>
        localdms:/local/test/Link
      </destination-name>
      <destination-type>
        Topic
      </destination-type>
    </output-destination>
  </output-port>
</message-source>
```

Configuring the Scenario Manager to Receive Your Message

You also use the DMS configuration files to configure the Scenario Manager to receive your custom messages.

The Scenario Manager acts as a message sink for messages that can occur in scenarios. It has two different input ports to which your messages can be sent: GlobalEvents and IndividualEvents. Our Link example is an individual event. The following, taken from the dynamoMessagingSystem.xml file, shows how to configure the Scenario Manager to receive events from our destination topic, localdms:/local/test/Link:

```
<message-sink>
   <nucleus-name>
     /atg/scenario/ScenarioManager
   </nucleus-name>
   <input-port>
     <port-name>
       IndividualEvents
     </port-name>
     <input-destination>
       <provider-name>
         local
       </provider-name>
       <destination-name>
         localdms:/local/test/Link
       </destination-name>
       <destination-type>
         Topic
       </destination-type>
     </input-destination>
   </input-port>
</message-sink>
```

If you are creating a global event that you want the Scenario Manager to listen for, you would add the same code to the dynamoMessagingSystemDSSGlobal.xml file instead, replacing IndividualEvents with GlobalEvents in the <port-name> tag. You would also specify Queue rather than Topic here as the destination type.

Note: For information on how to determine whether your event is global or individual, see Adding Your Message to the Scenario Event Registry.

Adding Your Message to the Scenario Event Registry

The Scenario Manager definition file, scenarioManager.xml, contains an event registry that specifies various pieces of configuration information for the scenario events in your system. You add your custom message to this registry as part of the creation process.

Note that, for compatibility with ATG products before version 6.0, the event registry in the scenarioManager.xml file is turned off by default. To enable the scenario event registry, set the useEventRegistry property to true in the ScenarioManager component. If you do not enable the scenario event registry, add the custom message to the DMS message registry instead (see the next section).

The following example shows the event registry entries for the clickthrough event we are adding:

```
<event-registry>

<event>
  <jms-type>
    test.scenario.Link
  </jms-type>
  <message-class>
    test.scenario.LinkMessage
  </message-class>
  <message-context>
    request
  </message-context>
  <message-scope>
    individual
  </message-scope>
  <display-name>
    Link clicked
  </display-name>
  <description>
    Message generated when user clicks a link
  </description>
</event>

</event-registry>
```

The <message-context> tag defines the nature of the message's originating context. The following values are recognized:

- request: the message originates in a request thread, and request- or session-specific values (for example, request, response, profile) may be resolved via JNDI.
- session: the message originates in a session-specific context, and session-specific values (for example, profile) may be resolved via JNDI.

If the <message-context> tag is omitted, no assumptions are made concerning the message's context. For example, the "Dynamo Starts" event does not specify a message context, while the "Visits Page" event has the request context, because it occurs in the context of a user request. Our custom message has a request context, since it clearly occurs in the context of a request.

The <message-scope> tag defines whether the event you are adding is a global event or an individual event. Global events (for example, the Dynamo Starts event) apply to all visitors who have reached that stage of the scenario. Individual events (for example, the Visits Page event) apply to specific visitors only. Thus, if a Dynamo Starts event occurs in a scenario, all visitors at the corresponding place in the scenario are affected by the event. On the other hand, if a Visits Page event occurs, only the people who visit the page are affected. Effectively, individual events act as filters that narrow the number of people who progress through the scenario.

Typically, a message with a request or session context represents an individual event, while a message with no context specified represents a global event. The default value for the <message-scope> tag is individual.

The <display-name>tag is used by the scenario editor interface in the ACC. It determines the label that appears for the event in the Events dropdown menu. Any message you add to the registry in this way appears by default in the events menu. If there are messages appearing as events that you would rather not show, you can include a <hidden> element, set to true, that prevents them from appearing.

Adding the Message to the DMS Message Registry

Add the custom message to the DMS message registry (the <message-registry> section of the DMS configuration file). For both individual and global events, add the message to the dynamoMessagingSystem.xml file.

(**Note:** If the scenario event registry is enabled, as described in the previous section, you must add the message to both the event registry and the DMS message registry. If you add the message to the event registry only, the event appears in the ACC, but its properties do not appear in scenario condition elements.)

For example, the LinkMessage message would be configured as follows:

```
<message-registry>
  <message-family>
    <message-family-name>
      test
    </message-family-name>
    <message-type>
      <jms-type>
        test.scenario.Link
      </jms-type>
      <message-class>
        test.scenario.LinkMessage
      </message-class>
      <message-context>
        request
      </message-context>
      <display-name>
        Link clicked
      </display-name>
      <description>
        Message generated when user clicks a link
      </description>
    </message-type>
  </message-family>
</message-registry>
```

For more information, refer to *DMS Configuration File Document Type Definition* in the *ATG Programming Guide*.

Note that the steps described so far are not specific to the Scenario Manager. This procedure is the same for adding any message to DMS.

Declaring the Local DMS Topic

If you are creating a custom individual event that uses a Local DMS topic as its destination, the final piece of required DMS configuration is to inform the Local DMS system of our new topic name. This is done by adding a `<topic-name>` element as follows to the `dynamoMessagingSystem.xml` file:

```
<local-jms>
  <jndi-prefix>
    /local
  </jndi-prefix>
  <topic-name>
    /test/Link
  </topic-name>
</local-jms>
```

Declaring the SQLDMS Topic or Queue

If you are creating a custom individual or custom global event that uses a SQLDMS topic or queue as its destination, the final step in the configuration process is to add the new queue to the `/atg/dynamo/messaging/SqlJmsProvider.properties` file in your configuration path. The following example shows the default `SqlJmsProvider.properties` file for the Scenarios module.

```
requiredTopicNames+=\
     sqldms/DSSTopic/ScenarioUpdateEvents,\
     sqldms/DSSTopic/IndividualTimerEvents,\
     sqldms/DSSTopic/CollectiveTimerEvents,\
     sqldms/DSSTopic/BatchTimerEvents

requiredQueueNames+=\
     sqldms/DSSQueue/IndividualTimerEvents,\
     sqldms/DSSQueue/CollectiveTimerEvents,\
     sqldms/DSSQueue/BatchTimerEvents
```

Putting It All Together

The following complete `dynamoMessagingSystem.xml` configuration file uses a <message-source> element to hook up our `/test/scenario/LinkMessageSource` component to the Local DMS topic `localdms:/local/test/Link`. It uses a `<message-sink>` element to enable the Scenario Manager to receive messages on this topic. A `<message-type>` element provides descriptive information about the `LinkMessage` message, and finally a `<topic-name>` element declares the topic to Local JMS:

```
<dynamo-message-system>
  <patchbay>
    <message-source>
      <nucleus-name>
        /test/scenario/LinkMessageSource
      </nucleus-name>
      <output-port>
        <port-name>
          Link
        </port-name>
        <output-destination>
          <provider-name>
            local
          </provider-name>
          <destination-name>
            localdms:/local/test/Link
          </destination-name>
          <destination-type>
            Topic
          </destination-type>
        </output-destination>
      </output-port>
    </message-source>
    <message-sink>
      <nucleus-name>
        /atg/scenario/ScenarioManager
      </nucleus-name>
      <input-port>
        <port-name>
          IndividualEvents
        </port-name>
        <input-destination>
          <provider-name>
            local
          </provider-name>
          <destination-name>
            localdms:/local/test/Link
          </destination-name>
          <destination-type>
            Topic
          </destination-type>
        </input-destination>
      </input-port>
    </message-sink>
  </patchbay>
  <message-registry>
    <message-family>
      <message-family-name>
        test
      </message-family-name>
      <message-type>
        <jms-type>
          test.scenario.Link
        </jms-type>
        <message-class>
          test.scenario.LinkMessage
        </message-class>
        <message-context>
          request
        </message-context>
        <display-name>
          Link clicked
        </display-name>
        <description>
          Message generated when user clicks a link
        </description>
      </message-type>
    </message-family>
  </message-registry>
  <local-jms>
    <jndi-prefix>
      /local
    </jndi-prefix>
    <topic-name>
      /test/Link
    </topic-name>
  </local-jms>
</dynamo-message-system>
```

Associating Profiles with Individual Custom Events

If you want to use an individual user's profile properties as qualifiers for triggering a custom scenario event, and the event occurs outside a request thread, the message that represents the event must include a `getSubject()` method (to allow access to the profile) and contain any of the following properties, of type `RepositoryItem`:

- `profile`
- `profileId`
- `subject`
- `subjectId`

These properties enable both the scenario engine and the ACC to recognize that the event is associated with individual users. As a result, the People Whose condition element, which you can insert into the scenario after the custom event, will pick up and display all dynamic properties in the user's profile, allowing you to use them to qualify the custom event.

For more information on accessing properties of Nucleus components from with a scenario event, refer to Exposing Nucleus Components for Use in Custom Bean Expressions.

Adding Custom Actions

You can extend the Scenario Manager to support custom scenario actions. The procedure has the following steps:

1. Add the new action to the Scenario Manager configuration file.
2. Implement the action interface (`atg.process.action.Action`).

The example in this section adds an action that simply logs some information about its parameters and its evaluation context to the Dynamo information log. Such an action is very handy for debugging purposes, among other things, and it illustrates some of the general principles of custom scenario actions.

Adding the Action to the Scenario Manager Configuration File

The Scenario Manager's configuration file is located in your config path at `/atg/scenario/scenarioManager.xml`. You can add a new action to this configuration by creating a file with the same name in your `localconfig` directory. Below is an example of a `scenarioManager.xml` file that you would have to define in order to add the test action.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE process-manager-configuration
        PUBLIC "-//Art Technology Group, Inc.//DTD Process Manager//EN"
        'http://www.atg.com/dtds/processmanager/processmanager_1.0.dtd'>

<process-manager-configuration>

  <action-registry>
    <action>
      <action-name>
        Log Info
      </action-name>
      <action-class>
        test.scenario.LogAction
      </action-class>
      <description>
        prints out information about the action's context and parameters
      </description>
      <action-execution-policy>
        individual
      </action-execution-policy>
      <action-error-response>
        continue
      </action-error-response>
      <action-parameter>
        <action-parameter-name>
          logString
        </action-parameter-name>
        <action-parameter-class>
          java.lang.String
        </action-parameter-class>
        <required>
          true
        </required>
        <description>
          a string value to evaluate and display
        </description>
      </action-parameter>
      <action-parameter>
        <action-parameter-name>
          logInteger
        </action-parameter-name>
        <action-parameter-class>
          java.lang.Integer
        </action-parameter-class>
        <required>
          true
        </required>
        <description>
          an integer value to evaluate and display
        </description>
      </action-parameter>
    </action>
  </action-registry>

</process-manager-configuration>
```

The action specification contains the action’s name, its class (which must implement the `atg.process.action.Action` interface), and its description. In addition, you must set two other important properties for your action in these tags:

- `<action-execution-policy>`
- `<action-error-response>`

Specifying the <action-execution-policy> Tag

For `<action-execution-policy>`, you can specify individual or collective as the value. It determines whether an action must be executed in the context of a specific visitor or visitors (individual), or if it can be executed once for an entire collection of visitors participating in a scenario (collective). For example, the `SendEmail` action is marked as individual because it must have a person to whom the e-mail is sent. On the other hand, the `RecordEvent` action is collective, because it simply records event information to a dataset, regardless of who is going through the scenario.

Specifying the <action-error-response> Tag

The tag <action-error-response> specifies what to do if an error occurs during the action's execution. You can specify delete, continue, or retry as the value.

- If the value is continue, the error is logged, but the scenario continues to the next element.
- If the value is delete, the error is logged, and the scenario stops executing for the user in question (the scenario instance is deleted). For example, the Send Email action causes the scenario instance to be deleted when an error occurs, while the Record Event action doesn't, because a failure to record an event is not considered fatal.
- If the value is retry, the scenario instance does not continue but stays in its original state. (If the error occurs while the scenario is in an intermediate state, the scenario reverts to the original waiting state.) The next time the triggering event occurs, the transition is taken and the action is executed again. This cycle continues until the action is executed without an error.

Adding Parameters to a Scenario Action

You can also specify any number of action parameters for an action. Each parameter declares a name and the parameter value type. Our test action, for example, has two parameters, one of type String (with name "logString"), and another of type Integer (with name "logInteger").

To require users to specify a given parameter, include a <required> tag for that parameter and set it to true (see the example above). This tag is optional. If you include it, the ACC displays an error message if users do not specify the parameter.

Users can specify action parameters as either explicit values or expressions when they create scenarios in the ACC. For example, for a parameter of type Integer, they can type an explicit integer value, such as "249," or select an expression that evaluates to an integer, such as "Person's age." When the system executes the action, it evaluates the parameter expressions in the current scenario context; thus the expression "Person's age" is evaluated to the age in the profile of the person going through the scenario.

For action parameters of type String or String[], you can include the optional tags <action-parameter-repository-name> and <action-parameter-repository-item-type>. Without these tags, users must manually type the repository ID for the item they want when they fill in a parameter. With these tags, however, the ACC displays a dialog box from which users can choose the appropriate item by its display name. The system then passes the repository ID to the action. The following example is from the Give Promotions action in ATG Commerce; it shows how the Promotions action parameter is set up to use these tags:

```
<action>
      <action-name>
        promotion
      </action-name>
      <action-class>
        atg.commerce.promotion.PromotionAction
      </action-class>
```

```
    <display-name>
      Give Promotion
    </display-name>
    <description>
      gives a profile a promotion
    </description>
    <action-execution-policy>
      individual
    </action-execution-policy>
    <action-error-response>
      continue
    </action-error-response>
    <action-parameter>
      <action-parameter-name>
        promotions
      </action-parameter-name>
      <action-parameter-class>
        java.lang.String[]
      </action-parameter-class>
      <action-parameter-repository-name>
        ProductCatalog
      </action-parameter-repository-name>
      <action-parameter-repository-item-type>
        promotion
      </action-parameter-repository-item-type>
      <description>
        the promotions to be added to the profile
      </description>
    </action-parameter>
  </action>
```

Implementing the Action Interface

The `<action-class>` tag above contains the class name of the `atg.process.action.Action` implementation that is invoked when your custom action occurs in a scenario. The Action interface refers to several other classes and interfaces. At a minimum, familiarize yourself with the following interfaces/classes before implementing your action:

- `atg.process.expression.Expression`
- `atg.process.action.Action`
- `atg.process.action.ActionImpl`
- `atg.process.ProcessExecutionContext`

The Expression interface represents an expression as described above (for example, “Person’s age,” or the value “249”). An Expression can be evaluated using a `ProcessExecutionContext`. Thus, the expression “Person’s age” returns the age in the current profile when evaluated against the current scenario execution context. You can also query the `ProcessExecutionContext` explicitly for the current user profile, request, and so on (see an example of this below).

The Action interface has an initialize method that is called when the action is first created. This method takes a Map of parameters. The keys in the Map are the String parameter names; the values are the Expression objects representing parameter values. The initialize method should store these parameter expressions for later evaluation. For example, suppose you create a scenario that contains a `LogAction`, with parameters configured as follows: `logString` is simply the string "this is a test," and `logInteger` is the expression "Person's age." When the scenario is created, the action's initialize method is called with a Map which contains two key/value pairs: `"logString"/Expression`, representing the string "this is a test," and `"logInteger"/Expression`, representing "Person's age."

When the action is executed on a particular user or collection of users, one of its execute methods gets called. The first version of the execute method takes a `ProcessExecutionContext`. The action may use this context object to evaluate its parameter Expressions, or it may obtain information directly from the context (for example, the context contains the current `DynamoHttpServletRequest`, which can be used to evaluate any Nucleus expression, among other things).

The second version of the execute method takes an array of `ProcessExecutionContext` objects. It is called when several scenario instances are traveling through the scenario at the same time; typically, it just calls the first version of the method for each of the given context objects.

The class `atg.process.action.ActionImpl` is an abstract Action implementation that is provided to make implementing simple actions easier. It provides methods for storing, retrieving, and evaluating parameter Expressions so you do not have to re-implement the same logic for each action. It also implements both versions of the execute method in terms of an abstract method, `executeAction`, which takes a single `ProcessExecutionContext`. Thus, to implement your action, you need to implement only the methods `initialize` and `executeAction`.

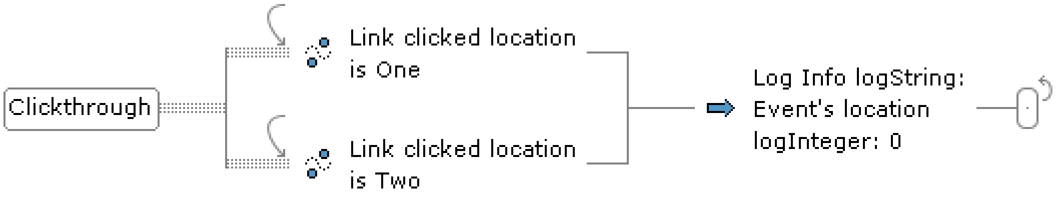
The `LogAction` example below inherits from `ActionImpl` and demonstrates its use. In its initialize method, it uses the method `storeRequiredParameter` to store the Expressions for parameters `logString` and `logInteger`. Then, in the `executeAction` method, it uses the method `getParameterValue` to evaluate the parameter expressions and print them out. In addition, the test action prints out all of the values available from the context.

```
package test.scenario;
import java.util.Map;
import atg.servlet.DynamoHttpServletRequest;
import atg.process.ProcessExecutionContext;
import atg.process.ProcessException;
import atg.process.action.ActionImpl;
/**
 * Custom scenario action that logs its parameters.
 *
 * @version $Revision$
 **/
public class LogAction extends ActionImpl {
  //-----------------------------------
  // Constants
  //-----------------------------------
  /** parameter: logString **/
  public static final String PARAM_LOG_STRING = "logString";
  /** parameter: logInteger **/
  public static final String PARAM_LOG_INTEGER = "logInteger";
  //-------------------------------------
  // ActionImpl overrides
  //-------------------------------------
  //-------------------------------------
  /**
   * Initializes the action with the given parameters. The keys in
   * the parameter Map are the String parameter names; the values are
   * the Expression objects representing parameter values.
   *
   * @exception ProcessException if the action could not be properly
   * initialized - for example, if not all of the required parameters
   * are present in the Map
   **/
  public void initialize(Map pParameters)
    throws ProcessException
  {
    storeRequiredParameter(pParameters, PARAM_LOG_STRING, String.class);
    storeRequiredParameter(pParameters, PARAM_LOG_INTEGER, Integer.class);
  }
  //-------------------------------------
  /**
   * Executes this action in the given single process execution
   * context. Called by both of the execute methods.
   *
   * @exception ProcessException if the action can not be executed
   **/
  protected void executeAction(ProcessExecutionContext pContext)
    throws ProcessException
  {
    // Get our request so that we can log stuff.
    DynamoHttpServletRequest request = pContext.getRequest();
    // use the context to evaluate the action's parameters
    String str = (String) getParameterValue(PARAM_LOG_STRING, pContext);
    Integer num = (Integer) getParameterValue(PARAM_LOG_INTEGER, pContext);
    request.logInfo("string value = " + str);
    request.logInfo("integer value = " + num);
    // the following objects are available from the context
    request.logInfo("scenario instance = " + pContext.getProcessInstance());
    request.logInfo("subject = " + pContext.getSubject());
    request.logInfo("message = " + pContext.getMessage());
    request.logInfo("request = " + request);
    request.logInfo("response = " + pContext.getResponse());
  }
  //-------------------------------------
}
```

Putting It All Together

After your action is implemented, and you update the configuration file to include it, the action appears in the Scenarios area of the ACC. Your business managers can then add it to scenarios just like any other action.

Here is an example scenario that includes both the custom `LinkMessage` event and the custom `LogAction` action:


Clickthrough
Link clicked location is One
Link clicked location is Two
Log Info logString: Event's location logInteger: 0

This scenario waits for a clickthrough with a `location` of either One or Two to occur. Whenever this happens, it logs the event's location code to the `info.log` file using our custom log action. The following test page can be created with the name `link.jsp` to test this scenario:

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


<HTML> <HEAD>
<TITLE>Link Event Test</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<H1>Link Event Test</H1>
<P>Actions:
<UL>
  <LI><dsp:a bean="/test/scenario/LinkMessageSource.location" value="One"
       href="link.jsp">One</dsp:a></LI>
  <LI><dsp:a bean="/test/scenario/LinkMessageSource.location" value="Two"
       href="link.jsp">Two</dsp:a></LI>
  <LI><dsp:a bean="/test/scenario/LinkMessageSource.location" value="Three"
       href="link.jsp">Three</dsp:a></LI>
</UL>

</BODY> </HTML>

<%/* Version: $Change: 224215 $$DateTime: 2001/12/27 14:00:56 $*/%>


</dsp:page>
```

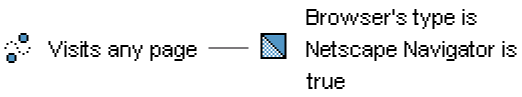
Here is the same example in JHTML:

```
<HTML> <HEAD>
<TITLE>Link Event Test</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<H1>Link Event Test</H1>
<P>Actions:
<UL>
  <LI><A HREF="link.jhtml" BEAN="/test/scenario/LinkMessageSource.location"
                           VALUE="One">One</A></LI>
  <LI><A HREF="link.jhtml" BEAN="/test/scenario/LinkMessageSource.location"
                           VALUE="Two">Two</A></LI>
  <LI><A HREF="link.jhtml" BEAN="/test/scenario/LinkMessageSource.location"
                           VALUE="Three">Three</A></LI>
</UL>
</BODY> </HTML>
```

Click any of the links and then examine the `info.log` file (`<ATG10dir>\home\logs`). Clicking link One or Two generates the output specified by the custom action. Clicking link Three produces no output.

Adding Custom Conditions

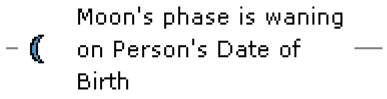
In the Scenarios module, condition elements serve as filters that further qualify the previous scenario event or action. For example, if you include a Views event that is triggered when a visitor displays any page, you can follow it with a condition that further qualifies the event according to the type of browser the visitor uses to generate the request:



Visits any page

Browser's type is Netscape Navigator is true

This section describes how to add your own custom condition elements if the default set does not meet your requirements. In the previous section of this chapter, the example showed how to add a custom action, `LogAction`, that printed out the contents of its parameters. This section develops that example and shows how to add a condition called `moonPhase` that performs the print action only if the moon is in a given phase on a particular date.



Moon's phase is waning on Person's Date of Birth

The procedure for adding a custom condition is similar to the procedure for adding a custom action element as described earlier. It can be summarized as follows:

1. Add the new condition to the condition registry in the Scenario Manager configuration file.
2. Extend the abstract class `atg.process.filter.ExpressionFilter` for your new condition.

Adding the New Condition to the Scenario Manager Configuration File

The Scenario Manager configuration file is located in your config path at `/atg/scenario/scenarioManager.xml`. You can add a new condition to this configuration by creating a file with the same name in your `localconfig` directory (`<ATG10dir>/home/localconfig/atg/scenario/scenarioManager.xml`).

Below is an example of a `scenarioManager.xml` file showing the condition registry entries that you would define in order to add the new condition. (Note that sample also shows the action definition, which is essentially the same as the example from the previous section of this chapter except that it uses the `Resources.properties` file shown later to determine display names for the ACC.)

The first operand of the condition is a String code of "1", "2", "3" or "4" that determines the phase to be checked for (new, waxing, full, or waning, respectively). The second operand is a scenario expression of type Date that determines the date on which the moon's phase will be evaluated and checked. If the moon was, is, or will be in the specified phase on the specified date, the condition evaluates to true; otherwise, it is false.

Note that the order of the operands is important. The filter class that evaluates the condition (see Extending the ExpressionFilter Class for the New Condition) processes the operands in the order in which you define them here.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE process-manager-configuration
        PUBLIC "-//Art Technology Group, Inc.//DTD Process Manager//EN"
        'http://www.atg.com/dtds/processmanager/processmanager_1.0.dtd'>

<process-manager-configuration>

  <action-registry>
    <action>
      <action-name>
        logInfo
      </action-name>
      <action-class>
        astroweb.LogAction
      </action-class>
      <resource-bundle>
        astroweb.Resources
      </resource-bundle>
      <display-name-resource>
       logAction.displayName
```

```
        </display-name-resource>
        <description-resource>
          logAction.decription
        </description-resource>
        <action-execution-policy>
          individual
        </action-execution-policy>
        <action-error-response>
          continue
        </action-error-response>

        <action-parameter>
          <action-parameter-name>
            logString
          </action-parameter-name>
          <action-parameter-class>
            java.lang.String
          </action-parameter-class>
          <required>
            true
          </required>
          <display-name-resource>
            logAction.logString.displayName
          </display-name-resource>
          <description-resource>
            logAction.logString.description
          </description-resource>
        </action-parameter>

        <action-parameter>
          <action-parameter-name>
            logInteger
          </action-parameter-name>
          <action-parameter-class>
            java.lang.Integer
          </action-parameter-class>
          <required>
            true
          </required>
          <display-name-resource>
            logAction.logInteger.displayName
          </display-name-resource>
          <description-resource>
            logAction.logInteger.description
          </description-resource>
        </action-parameter>
      </action>
    </action-registry>

    <!--
```

```
  Register a custom condition named "moonPhase" that determines
  whether the moon is in a given phase on a particular date.
-->

  <condition-registry>
    <condition>
      <condition-name>
        moonPhase
      </condition-name>
      <filter-class>
        astroweb.MoonFilter
      </filter-class>
      <resource-bundle>
        astroweb.Resources
      </resource-bundle>
      <display-name-resource>
        moonPhase.displayName
      </display-name-resource>
      <description-resource>
        moonPhase.description
      </description-resource>
      <icon-resource>
        moonPhase.icon
      </icon-resource>

      <action-parameter>
        <action-parameter-name>
         phase
        </action-parameter-name>
        <display-name-resource>
         moonPhase.phase.displayName
        </display-name-resource>
        <action-parameter-class>
         java.lang.String
        </action-parameter-class>
        <required>
         true
        </required>
        <description-resource>
         moonPhase.phase.description
        </description-resource>
      </action-parameter>

      <action-parameter>
        <action-parameter-name>
         date
        </action-parameter-name>
        <display-name-resource>
         moonPhase.date.displayName
        </display-name-resource>
        <action-parameter-class>
         java.util.Date
        </action-parameter-class>
        <required>
         true
        </required>
        <description-resource>
         moonPhase.date.description
        </description-resource>
      </action-parameter>

    </condition>
  </condition-registry>

</process-manager-configuration>
```

Adding an Icon to a Custom Element

Optionally, you can create an icon that will appear next to your custom condition in the ACC. In the example above, the `<icon-resource>` tags reference the file `astroweb/moon-phase.gif`, specified along with other display elements in the resource file shown in the next section.

Using a Resource Bundle Properties File to Define Display Elements

To ease the process of internationalizing the ACC, the custom condition uses a resource bundle properties file to determine display names for this condition. The file created for this condition, which is called `Resources.properties`, has the following contents:

```
#
# Resource bundle properties used by example custom scenario elements
#
logAction.displayName=Log data
logAction.description=test action to log values
logAction.logString.displayName=with string value
logAction.logString.description=a string value to evaluate and display
logAction.logInteger.displayName=and integer value
logAction.logInteger.description=an integer value to evaluate and display
moonPhase.displayName=Moon
moonPhase.description=lunar calendar condition
moonPhase.icon=astroweb/moon-phase.gif
moonPhase.phase.displayName=in phase
moonPhase.phase.description=1:new, 2:waxing, 3:full, 4:waning
moonPhase.date.displayName=on date
moonPhase.date.description=the date for which the phase is checked
vmSystem.displayName=Virtual Machine
```

(Note that the `vmSystem.displayName=Virtual Machine` line is included here only for the purposes of the custom bean expression example later in this chapter.)

For more information, refer to *Resource Bundles* in the *ATG Programming Guide*.

Extending the ExpressionFilter Class for the New Condition

After you have added the condition to the `scenarioManager.xml` file, you extend the abstract class `atg.process.filter.ExpressionFilter` for your new condition. The `ExpressionFilter` class evaluates the arguments in the condition and returns true or false.

The following code sample shows how you would extend this class for the custom condition in our example:

```
package astroweb;

import atg.process.ProcessException;
import atg.process.ProcessExecutionContext;
import atg.process.filter.Filter;
import atg.process.filter.ExpressionFilter;
import atg.process.expression.Expression;

import atg.beans.*;

import java.text.*;
import java.util.*;

/**
 * Filter example for testing the moon's phase as of a specific date.
 */
public class MoonFilter extends ExpressionFilter
{
  //-------------------------------------
  // Constants
  //-------------------------------------

  // calendar test type enumeration
  public static final int NEW = 0;
  public static final int WAXING = 1;
  public static final int FULL = 2;
  public static final int WANING = 3;

  //-------------------------------------
  // Fields
  //-------------------------------------

  /** The type of calendar test being performed, as per above constants */
  private int mTestType;

  //-------------------------------------
  // ExpressionFilter overrides
```

```
//-------------------------------------

//-------------------------------------
/**
 * Initializes this ExpressionFilter, given its operator and
 * operands.  The default implementation of this method simply sets
 * the operator and operands properties.
 *
 * @param pOperator the filter operator, not needed in this example
 * @param pOperands the operands to the filter.  pOperands[0] is an
 * Integer giving the test type, and pOperands[1] is a Date-typed
 * expression giving the date on which the moon's phase is to be
 * tested.
 *
 * @exception ProcessException if the operands argument is invalid
 **/

public void initialize(String pOperator, Expression[] pOperands)
  throws ProcessException
{
  super.initialize(pOperator, pOperands);

  // Verify that we have the expected number of operands: two
  int nOperands = (pOperands == null ? 0 : pOperands.length);
  if (nOperands != 2) {
    throw new ProcessException("Wrong number of operands: " + nOperands);
  }

  // Precompute our test type operand for efficiency, since it's a
  // constant.  First verify that we can get the values without an
  // execution context
  Expression exTestType = pOperands[0];
  if (!exTestType.canGetValue(null))
    throw new ProcessException("Unable to precompute test type");

  // Now look up the values, cast them to the correct types, and
  // cache values in data members.
  //
  // The rigorous type checking isn't strictly necessary since the
  // scenario editor grammar should constrain the types of our operands,
  // but a little extra sanity checking never hurts.
  Object value = null;
  try
  {
    value = exTestType.getValue(null);
    Integer testType = (Integer)value;

    if (testType == null)
      throw new ProcessException("Test type is null");
    mTestType = testType.intValue();
  }
  catch (ClassCastException cce) {
    throw new ProcessException("Test type had unexpected type: " +
    value.getClass().getName());
  }

  System.out.println("Cached mTestType = " + mTestType);
}


//---------------------------------------
/**
 * Evaluates this filter in the given process execution context.
 * The context may not yet contain all of the information necessary
 * to evaluate the filter - specifically, it may be missing the
 * particular scenario instance and/or profile that the scenario is
 * being executed on.  If that is the case, the filter is evaluated
 * as much as possible, and the simplified filter is returned.
 *
 * <p>The possible return values of this method are as follows:
 * <ul>
 * <li><code>Filter.TRUE</code> - if the filter can be fully
 * evaluated, and is satisfied in the given context</li>
 * <li><code>Filter.FALSE</code> - if the filter can be fully
 * evaluated, and is not satisfied in the given context</li>
 * <li><code>null</code> - if the filter cannot be evaluated because
 * of a null expression encountered during evaluation (e.g., filter
 * refers to a profile property which evaluates to null)</li>
 * </ul>
 *
 * @exception ProcessException if there is a problem evaluating the
 * filter (other than information missing from the context)
 **/
protected Filter evaluate(ProcessExecutionContext pContext)
  throws ProcessException
{
  Expression[] operands = getOperands();

  // Verify that all variable operand values are available

  Expression exDate = operands[1];
  if (!exDate.canGetValue(pContext))
    return this;

  // Get dynamic operand values and convert them to the expected
  // types, as above.
  Object value;
  Date dateValue;
  if ((value = exDate.getValue(pContext)) == null)
    return null;

  try {
    dateValue = (Date) value;
  }
  catch (ClassCastException cce) {
    throw new ProcessException("Date value was of wrong type: " +
    value.getClass().getName());
  }

  MoonCalendar mc = new MoonCalendar(dateValue);
  boolean result = false;

  System.out.println("Age of moon = " + mc.getAge());

  switch(mTestType)
  {
  case NEW:
    result = mc.isNew();
    break;
  case WAXING:
    result = mc.isWaxing();
    break;
  case FULL:
    result = mc.isFull();
    break;
  case WANING:
    result = mc.isWaning();
    break;
  default:
    throw new ProcessException("Unknown test type value: " + mTestType);
  }

  return result ? Filter.TRUE : Filter.FALSE;
}
}
```

The next code sample is included for completeness. It constructs a JavaBean representing the calculator that determines the phase of the moon for any given date.

```
package astroweb;

import java.util.Date;

/**
 * A simple calendar calculator that determines lunar phase
 * information.
 **/

public class MoonCalendar
{
  // Calendrical and astronomical constants
  private static final long MILLIS_PER_DAY = 86400000;
  private static final long JULIAN_EPOCH = 2440588; // Jaunary 1, 1970
  private static final double LUNAR_PERIOD = 29.530588853;
  private static final double NEW_MOON_BASELINE = 2451550.1;

  /**
   * The age of the moon, expressed as a number of days, based on the
   * given date, normalized to lie in the range 0..LUNAR_PERIOD.
   **/
  private double mAge;


  //----------------------------------------
  /**
   * Construct a MoonCalendar for some given date, precalculating the
   * moon's age.
   */
  public MoonCalendar(Date date)
  {
    long julianDate = JULIAN_EPOCH + (date.getTime() / MILLIS_PER_DAY);
    double tempVal = (julianDate - NEW_MOON_BASELINE) / LUNAR_PERIOD;
    tempVal -= Math.floor(tempVal);
    if (tempVal < 0)
      tempVal += 1;
    mAge = tempVal * LUNAR_PERIOD;
  }

  //----------------------------------------
  /**
   * Construct a MoonCalendar for the current time
   */
  public MoonCalendar()
  {
    this(new Date());
  }

  //----------------------------------------
  /**
   * Return the age of the moon in days.
   */
  public double getAge()
  {
    return mAge;
  }
  //----------------------------------------
  /**
   * Determine if the moon is waxing
   */
  public boolean isWaxing()
  {
    return mAge > 0.5 && mAge < 14.0;
  }

  //----------------------------------------
  /**
   * Determine if the moon is waning
   */
  public boolean isWaning()
  {
    return mAge > 15.0 && mAge < 29.0;
  }

  //----------------------------------------
  /**
   * Determine if the moon is full
   */
  public boolean isFull()
  {
    return mAge > 14.0 && mAge < 15.0;
  }

  //----------------------------------------
  /**
   * Determine if the moon is new
   */
  public boolean isNew()
  {
    return mAge > 29.0 || mAge < 0.5;
  }
}
```

Extending the Expression Editor

At this point, the custom condition is ready for use by your business managers. Optionally, however, you could extend the Scenarios module's grammar expression editor to provide more elegant handling of the new custom condition in the ACC. This process involves two steps:

1. Create an XML file that defines the custom expression grammar. In our example, the file is `astroweb/astroweb-grammar.xml`.
2. Add a grammar registry entry specifying the location of the expression grammar XML file to the `scenarioManager.xml` file.

The next sections in this chapter describe these steps.

For detailed information on options for extending the expression editor for use in the Scenarios module and other ATG products, refer to the next chapter, Configuring the ATG Expression Editor.

Creating an Expression Grammar Definition File

The following sample shows the file `astroweb/astroweb-grammar.xml`.

```
 <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
  <?xcl-stylesheet resource="atg/ui/scenario/expression/scenario-grammar.xsl"?>
  <?xcl-stylesheet resource="atg/ui/expreditor/xcl/grammar.xsl"?>
- <!--
This grammar definition supplies the details of how our example's
custom action and custom condition will be treated in the UI.
  -->

<context>
 <!--
 Define the grammar of action "logInfo".  Our intention is to
  constrain both operands to literals, rather than general scenario
  expressions, and to require the logInteger to be unsigned.
  -->
<sequence id="action-logInfo">
 <!--
 Attach an XML template to this grammar node, specifying the
    SDL that it will emit.  This template will generate an <action-name>
    tag inside the enclosing <action> tag, followed by SDL emitted by
    children of this node.

  -->
<xml-template>
  <action-name>logInfo</action-name>
  <apply-xml-templates />
  </xml-template>
  <!--
 Provide a leading token that will show up as a choice in the
    Actions menu of the scenario editor.

  -->
<token>
  <description>Log test data</description>
  </token>
  <!--  include a token describing the string to be logged
  -->
<token>
  <description>with label</description>
  </token>
  <!--  then, a literal supplying the logString parameter
  -->
<literal>
```

```
<xml-template>
  <action-param name="logString">
  <constant>
  <apply-xml-templates />
  </constant>
  </action-param>
  </xml-template>
  <required />
  </literal>
  <!--  then a token describing the int to be logged
  -->
  <token>
  <description>and value</description>
  </token>
  <!--
 then, a literal supplying the integer parameter.  The
    <default> tag is required for all non-String literals as it
    determines the data type.

  -->
<literal>
<xml-template>
<action-param name="logInteger">
<constant>
  <apply-xml-templates />
  </constant>
  </action-param>
  </xml-template>
  <unsigned-integer-editor />
  <required />
<default>
  <value type="java.lang.Integer">1</value>
  </default>
  </literal>
  </sequence>
<!--
 Define the grammar of condition "moonPhase".  Note that the id
  of this grammar element is the prefix "condition-", followed by the
  condition name given in scenarioManager.xml.  We would like to
  support the following syntax:

     Moon's phase is (new | waxing | full | waning) (on DATE-EXPR | now)

  The MoonFilter class expects two operands; the first operand is a
  stringified integer relating to the choice of phase (new, waxing,
  ...), while the second is an arbitrary expression of type Date.

  -->
<sequence id="condition-moonPhase">
<!--
```

```
 Attach an XML template to this grammar node, specifying the
    SDL that it will emit inside an enclosing <condition> tag.  This
    template says that a <filter operator="moonPhase"> tag should
    enclose all XML generated by children of this grammar node.

  -->
<xml-template>
<filter operator="moonPhase">
  <apply-xml-templates />
  </filter>
  </xml-template>
<!--
 Provide a leading token that will show up as a choice in the
    Conditions menu of the scenario editor.

  -->
<token>
  <description>Moon's phase is</description>
  </token>
<!--
 Provide a choice of tokens for the moon phase, the first
    condition operand.  Each token's <description> governs how it is
    presented in the scenario editor choice list, while each token's
    XML template supplies an SDL fragment defining a constant value,
    to be generated when that token is the current choice.
  -->
<choice>
<!--  (new | waxing | full | waning)
  -->
<token>
  <xml-template>
    <constant type="java.lang.Integer">0</constant>
  </xml-template>
  <description>new</description>
</token>
<token>
  <xml-template>
    <constant type="java.lang.Integer">1</constant>
  </xml-template>
  <description>waxing</description>
</token>
<token>
  <xml-template>
    <constant type="java.lang.Integer">2</constant>
  </xml-template>
  <description>full</description>
</token>
<token>
  <xml-template>
    <constant type="java.lang.Integer">3</constant>
  </xml-template>
  <description>waning</description>
</token>
</choice>
<!--
 Provide a choice of two possible values for the second
    operand of the condition.  The first choice is a token whose
    displayed description is "now", which is a short hand for an SDL
    expression equivalent to "Today's timeAsDate".  The second choice
    is an arbitrary scenario expression of type java.util.Date.

    Note that the "now" choice must come first, because the SDL for
    "now" can match either of the two choices' templates; we want it
    to match the more specific of the two.

  -->
<choice>
<!--  (on DATE-EXPR | now)
  -->
<token>
<!--  now
  -->
 <xml-template>
  <jndi-property>
   <jndi-url>dynamo:/atg/dynamo/service/CurrentDate</jndi-url>
   <property-name>timeAsDate</property-name>
  </jndi-property>
 </xml-template>
 <description>now</description>
<!--  only show the word "now" in the dropdown choice list
  -->
<hidden />
</token>
<sequence>
<!--  on DATE-EXPR
  -->
<token>
  <editor-text>on</editor-text>
<!--  show ellipsis in dropdown choice list only
  -->
  <description>on...</description>
  </token>
  <scenario-expression type="java.util.Date" />
  </sequence>
  </choice>
  </sequence>
  </context>
```

Extending the Grammar Extension Class

Create an extension of the class `atg.ui.scenario.expression.DefaultGrammarExtension` that simply identifies how to locate the grammar extension XML file, which in our example is `astroweb/astroweb-grammar.xml`. The following sample shows the extension created for our custom condition:

```
package astroweb;

import atg.ui.scenario.expression.*;

public class AstrowebGrammarExtension extends DefaultGrammarExtension
{
  //----------------------------------------
  /**
   * Construct a grammar extension that references our example custom
   * expression grammar definition.
   */
  public AstrowebGrammarExtension()
  {
    // Specify the grammar file.
    super("astroweb.astroweb-grammar");

    // Note: the grammar file is specified with no suffix and with a
    // dot-qualified package name, since it is localized in a
    // ResourceBundle-like way.  If the user's locale is en_US, for
    // instance, the following files will be searched for, in this
    // order:
    //
    //   astroweb/astroweb-grammar_en_US.xml
    //   astroweb/astroweb-grammar_en.xml
    //   astroweb/astroweb-grammar.xml
  }
}
```

Specifying the Location of the Expression Grammar XML File

The final step is to add a grammar registry entry to the `scenarioManager.xml` file that specifies the location of the expression grammar XML file. The following sample shows the additions you would make to the `scenarioManager.xml` file:

```
<grammar-registry>
   <grammar-extension-file>astroweb.astroweb-grammar
   </grammar-extension-file>
</grammar-registry>
```

Note that the grammar file is specified with no suffix and with a dot-qualified package name to make localization easier. If the user’s locale is `en_US`, for instance, the following files will be searched for, in this order:

```
astroweb/astroweb-grammar_en_US.xml
astroweb/astroweb-grammar_en.xml
astroweb/astroweb-grammar.xml
```

Configuring Actions and Conditions through Properties Files

The action registry in the `scenarioManager.xml` file contains an optional tag called `<action-configuration>` that allows you to configure an action through the `.properties` file of a Nucleus component. The following example shows the `<action-configuration>` entry for the standard `SendEmail` action:

```
<action-configuration>
    /atg/scenario/configuration/SendEmailConfiguration
</action-configuration>
```

The following example shows the .properties file of the `SendEmailConfiguration` component:

```
# Version: $Change: 244651 $$DateTime: 2002/06/24 09:19:29 $
$class=atg.scenario.action.SendEmailConfiguration

defaultEmailInfo=/atg/scenario/DefaultTemplateEmailInfo
individualEmailSender=/atg/scenario/IndividualEmailSender
collectiveEmailSender=/atg/scenario/CollectiveEmailSender
webAppRegistry=/atg/registry/WebApplicationRegistry
```

The `individulaEmailSender` property, for example, allows you to specify the email sender component to use (here, `/atg/scenario/IndividualEmailSender`).

If you create a custom action and you want it to be configurable, override the `configure` method in your implementation of the `atg.process.action.Action` interface to extract the required information from the configuration object that you create. Note that the method must cast the object to the expected type. If you create a custom action and it does not need a flexible configuration, you can omit the `<action-configuration>` tag and specify the names and paths of any necessary components directly in the action registry. The configure method, in this case, throws an `UnsupportedObjectException`. For more information on this method, refer to the *ATG API Reference*.

A similar technique exists for configuring scenario conditions. The condition registry in the `scenarioManager.xml` file contains an optional tag called `<filter-configuration>` that you can use to specify the name and path of a component that contains configuration properties for a scenario condition. The `atg.process.filter.ExpressionFilter` class contains a `configure` method that you can override to use the settings in the configuration component. Again, the default implementation throws an `UnsupportedObjectExpression` so that you can choose to omit the `<filter-configuration>` tag if your condition does not need to be configurable through a .properties file.

Exposing Nucleus Components for Use in Custom Bean Expressions

This section describes how to expose a JNDI-accessible JavaBean so that you can use its properties within any scenario expression. The example below shows how to configure a custom expression that exposes the standard Nucleus component `/VMSystem`. (This component provides access to global properties of the JVM such as the `freeMemory` bean property, which shows the amount of free memory exposed.)

You can then use that component in any scenario element, allowing you to create expressions such as the following:

```
Set variable foo to Virtual Machine's freeMemory

Virtual Machine's freeMemory is greater than 3000000
```

To expose a component for use in scenario elements, add it to the bean expression registry in the `scenarioManager.xml` file, located in your config path at `/atg/scenario/scenarioManager.xml`. The following code sample shows the bean expression registry entries you would make for the `/VMSystem` component.

The JNDI path of any Nucleus component is its path preceded by a “dynamo:” prefix.

```
 <?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE process-manager-configuration
        PUBLIC "-//Art Technology Group, Inc.//DTD Process Manager//EN"
        'http://www.atg.com/dtds/processmanager/processmanager_1.0.dtd'>

 <process-manager-configuration>
   <bean-expression-registry>
     <bean-expression>
       <bean-expression-name>
          VMSystem
       </bean-expression-name>
       <jndi-url>
          dynamo:/VMSystem
       </jndi-url>
       <bean-info-provider-path>
          /VMSystem
       </bean-info-provider-path>
       <resource-bundle>
         astroweb.Resources
       </resource-bundle>
       <display-name-resource>
          vmSystem.displayName
       </display-name-resource>
     </bean-expression>
```

```
  </bean-expression-registry>
</process-manager-configuration>
```

Note this example uses the same `Resources.properties` file as the custom condition example to define the component's name for display in the scenario editor. The relevant line is as follows:

```
vmSystem.displayName=Virtual Machine
```

22 Filtering Collections

Collection filtering is the process of reducing objects in a collection based on some condition. Each collection filtering component defines a condition (for example, start date) and, when necessary, a criteria value (5/19/2010). When a collection filtering component receives a collection, it determines which objects satisfy the condition by having matching criteria (objects that have a start date on or before 5/19/2010). All matching objects become part of the result set collection while the remaining objects are "filtered out" or discarded.

When you want to filter a collection based on several conditions, you can create one complex filter. Another approach is to define one collection filtering component per condition and a filter chain component to organize the filters into a sequence chain. The chain produces one result set that satisfies all of the conditions specified by the filters within it.

A collection used in collection filtering can be made up of any grouping of objects. This grouping is passed to a collection filtering component by a scenario or a servlet bean in a JSP that is defined specifically for this purpose. Once the collection filter processes the collection, it returns the resultant collection to the invoking resource. When that resource is a servlet bean, you have the option of caching the resultant content in order to optimize performance for repeat collections accessed multiple times.

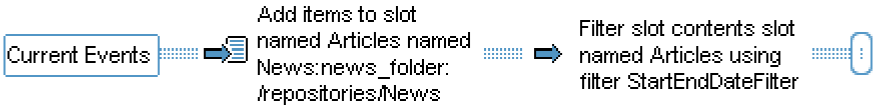
You can also access the collection filtering classes in the `atg.service.collections.filter` package using the API. See *ATG API Reference* for information.

ATG Commerce provides additional collection filtering components to ATG Commerce customers. See the *ATG Commerce Guide to Setting Up a Store* for more information.

This chapter describes collection filtering in the following sections:

How Collection Filtering Works

This example describes how a collection filter limits the contents in a slot using a scenario. Consider a Web site that provides articles about various topics. To ensure that only current articles are displayed to users, each article has a start and end date property. Articles that are active on the current day are added to a slot.


Current Events
Add items to slot named Articles named News:news_folder: /repositories/News
Filter slot contents slot named Articles using filter StartEndDateFilter

The scenario begins with an event that adds all News RepositoryItems to the Articles slot. The next action, Filter Slot Contents, passes the items in Articles to StartEndDateFilter. To learn more about this action, see Filter Slot Contents Action.

Rather than use a scenario to filter a slot, you can use a JSP to access a filled slot and filter the contents within it. The StartEndDateFilterDroplet accesses the StartEndDateFilter component, which performs the filtering task and adds the resultant collection to the Articles slot:

```
<dsp:droplet name="/atg/collections/filter/droplet/StartEndDateDroplet">
   <dsp:param name="collection" beanvalue="Articles"/>
   <dsp:param name="collectionIdentifierKey" value="date"/>

   <dsp:oparam name="output">
      Featured Articles:<p>
      <dsp:droplet name="/atg/targeting/TargetingForEach">
         <dsp:param bean="/atg/registry/Slots/Articles" name="targeter"/>
         <dsp:oparam name="output">
            <dsp:valueof param="filteredCollection"/>
         </dsp:oparam>
      </dsp:droplet>
   </dsp:oparam>

   <dsp:oparam name="empty">
      There are no articles today.
   </dsp:oparam>
</dsp:droplet>
```

Each article has a start and end date, which StartEndDateFilter compares to the current date. All articles that are “started” and have not yet “ended” remain in the slot. All others are removed by the filter from the slot.

Using Collection Filtering Classes

Each collection filtering class relies on the base abstract class atg.service.collections.filter.CachedCollectionFilter for the ability to filter collections. This class is designed to receive a collection, remove items from the collection, and create a result set. It’s possible to pass a Profile to this class so that subclasses can use Profile property values as filter conditions.

Other objects and properties may also be passed and used as filter conditions (see Passing Additional Parameters to a Filter (Filtering in a Multisite Environment) for more details). Finally, this class is capable of caching the filtered collection.

When you want to filter objects in a collection, you use a subclass of CachedCollectionFilter. ATG comes with several subclasses already implemented:

- atg.service.collections.filter.StartEndDateFilter limits a collection by removing objects that are inactive. You can find an instance of it in /atg/registry/CollectionFilters/StartEndDateFilter and a servlet bean for accessing it from a JSP in /atg/collections/filter/droplet/StartEndDateFilterDroplet.
- atg.service.collections.filter.ChainedFilter lets you chain several filters together so the resultant collection contains items that meet the conditions of all filters in the chain. There are no sample components in ATG Relationship Management, but ATG Commerce includes a sample filtering component and servlet bean. See the *Filtering Collections of Products* chapter of the *ATG Commerce Guide to Setting Up a Store* for details about this and other components of CachedCollectionFilter.
- atg.commerce.gifts.GiftlistSiteFilter filters gift lists and gift items in a multisite environment. You can find an instance of this class in /atg/registry/CollectionFilters/GiftlistSiteFilter and a servlet bean for accessing it in /atg/commerce/collections/filter/droplet/GiftlistSiteFilterDroplet. For more information, see Passing Additional Parameters to a Filter (Filtering in a Multisite Environment).

Using StartEndDateFilter

The atg.service.collections.filter.StartEndDateFilter class is used to evaluate the objects in a collection based on start and end date properties. ATG Relationship Management includes one component of this class: /atg/registry/CollectionFilters/StartEndDateFilter.

StartEndDateFilter has two properties, startDatePropertyName and endDatePropertyName that map to Date properties on the objects in the collection. By default, these properties are set to startDate and endDate respectively, anticipating that the objects in the collection have properties with these names. Filtering is based on whether objects are “started” or have “ended,” relative to the date the collection filter is executed.

For example, an object remains in the result set collection when the following are both true:

- The object’s startDate value is a date that occurs before the day the page is requested or is null. Objects without a startDate property also remain.
- The object’s endDate value is a date that occurs after the day the page is requested or is null. Objects without an endDate property also remain.

You can access the StartEndDateFilter component through the /atg/collections/filter/droplet/StartEndDateFilterDroplet servlet bean. When you use the servlet bean, the resultant content is cached by default. For more information on caching, see Caching Filtered Content.

Using ChainedFilter

The atg.service.collections.filter.ChainedFilter is used to create a chain of collection filters that apply their conditions to a collection of objects. There are no instances of ChainedFilter included with ATG Relationship Management.

The ChainedFilter class passes the objects to the first collection filtering component defined in filters property for processing. The resultant collection is then passed on to the next collection filter defined in filters and so on until all collection filters are executed and a final result set collection is produced.

Caching Filtered Content

When you filter a collection in a JSP, you can cache the filtered results so that future executions of that page need not run the collection filter to produce an identical result set. A filter that uses caching attempts to reuse content cached by a previous invocation that filtered an identical collection. When no cached content can be reused, the filter executes and saves its resultant collection to the cache.

To enable caching, you need to set five properties. For example, to cache content from StartEndDateFilter, the following properties require these values:

1. StartEndDateFilter.cacheEnabled= true
2. StartEndDateFilter.cache= /atg/collections/filter/FilterCache

   Note that this property can be set to any Cache component designed to work with collection filters.
3. FilterCache.cacheAdapter=/atg/collections/filter/FilterCacheAdapter
4. StartEndDateFilterDroplet.consultCache=true
5. StartEndDateFilterDroplet.updateCache=true

Note: The values described in this list are the default settings for the StartEndDateFilter implementation.

FilterCache is a component of class atg.service.cache.Cache designed to handle caching for collection filtering components. You can use FilterCache to configure the caching settings.

FilterCache has a Map object that uses a key (CollectionCacheKey) to describe the filtering conditions and a value to hold the filtered results. Specifically, the CollectionCacheKey is an object that consists of:

- A reference to the collection filtering component

- A unique string that represents the unfiltered collection identifier key. All references to a specific collection should use the same key. When you use a collection filtering servlet bean, you specify this key as an input parameter.
- A context object created by the collection filter to hold context criteria used by a filter instance. For example, a `Date` context object holding the current day's date is used by the `StartEndDateFilter` to determine whether a collection of objects are active on a specific day. Not all collection filtering components have context objects.

Here's an example of how caching could work in the `StartEndDateFilter`. The first call to a JSP that uses the `StartEndDateFilterDroplet` servlet bean relies on `StartEndDateFilter` to generate a collection of active objects. When caching is enabled, `FilterCache` saves the following information in the Map key: the `StartEndDateFilter` component, the `catid-53009-curtains` string, and a `Date` object that specifies the date the filter was executed. The "active" objects, those being the objects that remain after filtering, are saved to the Map as a value.

Subsequent renderings of that JSP will cause ATG 10.0.2 to compare the cached `CacheCollectionKey` key to the newly generated one. When the keys match, the filter returns the cached collection. By default, a cache is flushed once a day, which is appropriate for the `StartEndDateFilter` since the cached content it generates is only relevant for one day.

Caching For Chained Filters

Caching for chained collection filters is slightly more complex because there are two opportunities for caching: caching for the first collection filter in the chain and caching for the chain itself.

Consider a filter chain that chains together three filters: `StartEndDateFilter`, `AFilter`, and `BFilter`. The filter chain passes the collection to `StartEndDateFilter`, which, in turn, passes a reduced collection to `AFilter`, and so on from `AFilter` to `BFilter`. The final collection is returned to filter chain instance.

Caching the content produced by each collection filter that's part of a chain is inefficient because the context largely influences the output produced be a collection filter. A change to the context might change the output of one collection filter, which would alter the output of all collection filters that follow it in the chain. For example assume that on March 15, `StartEndDateFilter` returns a collection that includes objects `A`, `B`, `C`, and `D`. `AFilter` processes this collection and produces a subset, which `BFilter` reduces further to produce a final collection set. Caching the content for each individual filter is wasteful if you consider that March 16 could cause `StartEndDateFilter` to return a completely different set of objects, such as `G`, `H`, `T`, and `R`. Cached content produced by downstream collection filters would not apply.

As a result, caching is available at two points during the chained filter lifecycle: for the first collection filter's output collection and the chained filter's final output collection. In order to cache the chain results, the cached Map is constructed in a slightly different way. The `CacheCollectionKey` has an array of context objects, one for each collection filter in the chain. Since the array must have a context object that represents each collection filter, but not all collection filters have a context keys, the array contains the following:

- A context key for each collection filter that produces a context key while filtering the collection

- A null placeholder for each collection filter that filters the collection, but doesn't have a context object
- A reference to the filter itself for each collection filter that can't filter the collection

To determine whether cached content can be reused, the current `CacheCollectionKey` is compared to previously generated ones.

In this example, cached content would be saved for `StartEndDateFilter` and the `ChainedFilter` instance, assuming that caching is enabled for both of them. However, regardless of the caching settings on the remaining collection filters in the chain, they will never be cached because when they are executed, their `consultCache` and `updateCache` servlet bean properties are explicitly set to `false`.

Determining When To Cache Filter Content

Caching is most effective for content that requires expensive, re-usable resources or computations to generate it. Some collection filtering components, such as `StartEndDateFilter`, are ideal for caching because the content they produce applies to a broad spectrum of users for a sizable period of time. It is more efficient for `StartEndDateFilter` to cache its result once a day rather than compute the same collection of objects for each request.

Caching is a waste of resources for collection filtering components when the input collection changes frequently. A site wide search, for example, would render a list of results, the contents of which would not likely be identical to other result sets. Another poor candidate is a collection filter that takes a user's birthday as a criterion because the cached content is only reused for users with the exact same birthday. In this instance, too many resources are spent to store content in the cache and too few users access it for caching to be worthwhile. Consider that balance when you are deciding when to use filter caching.

Although caching isn't appropriate for some collection filters that use Profile conditions such as a zip code or items on a wish list because such content is too specific, caching is appropriate for other Profile-related conditions, such as those that use state or gender.

Configuring FilterCache

Use the following `FilterCache` properties to configure caching for your collection filters. `FilterCache` is a global component that applies for all collection filters. If you'd like different cache settings for each collection filter, create other components based on the `atg.service.cache.Cache` class and configure them appropriately.

When you are setting these properties, keep in mind that each cached item is a collection. The accumulation of many, large cached items can degrade performance.

Property	Description
`cacheAdapter`	The component that handles modifications to cached content, when such modifications are necessary. This property must be set to `/atg/collections/filer/FilterCacheAdapter`.

`maximumCacheEntries`	The maximum number of entries in the cache. 0 = Cache nothing. -1 = Unlimited (Default)
`maximumEntryLifetime`	The maximum time, in milliseconds, that an entry can exist in the cache. 0 = Cache nothing. -1 = Cached entries never expire (Default)

Note: To empty the cache immediately, invoke the `flush()` method by locating `FilterCache` in the Admin UI and clicking the **flush** link in the Methods section. You can also invoke this method directly through the API.

Implementing Custom Collection Filters

It is likely that you will want to create custom collection filters that filter objects based on the conditions that meet the specific needs of your Web application. When creating collection filters, you may want to keep them simple and modular by limiting them to work with one or two conditions only. Remember that you can link as many filters together as you like in a chain.

To create custom collection filters, complete the following tasks:

1. Create a collection filtering component. See Creating Custom Collection Filters.
2. Configure your collection filtering component. See Configuring Custom Collection Filters.
3. Create a resource that can access the collection filtering component. See Accessing a Collection Filtering Component.

To demonstrate how to create a custom collection filter, this discussion walks you through implementing a fictitious filter called `MembershipFilter`. Consider that a site has two levels of membership: basic and premier. This filter ensures that members only see articles suited for their level of membership by removing all articles from a slot that are marked as appropriate for another membership level . Assume that the articles and user Profiles have a enumerated property named `membershipLevel` with values `0` (basic) and `1` (premier)

Creating Custom Collection Filters

Keep in mind that when you define a collection filter, you specify a type of object to which it should be applied. Any objects in a collection that have a type other than the specified type will be ignored by the collection filter and included in the result set.

To create a collection filtering component, do the following:

1. Create a class that is a subclass of `atg.service.collections.filter.CachedCollectionFilter`
2. Implement the `shouldApplyFilter()` method, which is used to determine whether a collection filter should be applied given the execution context. For example, the `MembershipFilter` would access a user's membership level and decide if the filter is relevant for that user. Since `MembershipFilter` supports basic and premier members, which are the only kinds of membership permitted by the site, this method will always return `true`. If, in the future, the site permits additional membership levels, you would need to revise this method.
3. Implement the `generateFilteredCollection()` method, which is a method used to create an output collection and add the objects that meet the filter condition to it. You need to code this method to specify the data type of the output collection.

   It is recommended that you use the `generateNewCollectionObject()` method to configure the output collection data type to be the same as the input collection, barring certain restrictions: an input type of `atg.adapter.gsa.ChangeAwareList` creates an output type of `ArrayList`, and an input type of `atg.adapter.gsa.ChangeAwareSet` creates an output type of `HashSet`.
4. Implement the `generateContextKey()` method, which is a method used to create a context key object that holds the criteria value for a particular filtering instance. For example, the object created by `MembershipFilter` could be an Integer object that represents values 0 (when a user is a basic member) or 1 (when a user is a premier member). For collections that don't have context specific values, this method should return null.
5. Add other methods and properties as necessary. There are no other necessary properties or methods required for the `MembershipFilter` example.
6. Create a component for that class, such as `/atg/registry/CollectionFilters/MembershipFilter`. See the *ATG Programming Guide* for instructions on creating components.

The `shouldApplyFilter()`, `generateFilteredCollection()`, and `generateContextKey()` methods, along with several other `CachedCollectionFilter` methods, are all overloaded methods with two signatures, one that includes a `pExtraParameters` map and another that doesn't. The `pExtraParameters` map allows you to pass additional parameters to a filter so it can do its work. The version of the method that uses `pExtraParameters` contains the logic required to perform the given tasks. The method that doesn't include `pExtraParameters` simply calls the method that does and passes in a null map. This use of overloading allows code that calls the original methods to continue to function properly while providing the flexibility to include additional filter parameters when needed. When writing your own implementations, you should implement the version of each method that includes the `pExtraPameters` map, so as not to break this logic. For more information, see Passing Additional Parameters to a Filter (Filtering in a Multisite Environment).

Configuring Custom Collection Filters

Configure the properties on your custom collection filter as follows:

1. If you want your component to cache its content, see Caching Filtered Content for information on which properties to set.

2. In the component, specify values to properties that determine the filtering conditions as well as any other properties for which the default values are insufficient for your filter. Here's the default values for MembershipFilter:

```
#/atg/registry/CollectionFilters/MembershipFilter
$class=atg.service.collections.filter.MembershipFilter
$scope=global
cache=/atg/collections/filter/FilterCache
cacheEnabled=true
```

Accessing a Collection Filtering Component

There are two ways to access a collection filtering component. You can create a scenario that has a Filter Slot Contents action element using your custom filter component. By default, your collection filtering component is visible to Filter Slot Contents as long as it lives in the /atg/registry/CollectionFilters directory.

The other option is to access the custom collection filter in a JSP using a servlet bean. To do this, create a servlet bean that is an instance of atg.service.collections.filter.droplet.CollectionFilter (MembershipFilterDroplet) and implement it in a JSP. By setting MembershipFilterDroplet.filter to /atg/registry/CollectionFilters/MembershipFilter, you make it the default filter so that you need not specify the filter input parameter in your JSP MembershipFilter implementation. Refer to the *ATG API Reference* for more information about the atg.service.collections.filter.droplet.CollectionFilter class.

Passing Additional Parameters to a Filter (Filtering in a Multisite Environment)

There may be occasions when you need to pass additional parameters to a filter in order for the filter to do its work. For example, when filtering collections in a multisite environment, you must specify which sites' items should be evaluated. To do this, the filtering classes in a multisite environment require two additional pieces of information:

- A list of site IDs that defines the set of sites whose items should be evaluated.
- A site scope setting that controls whether filtering activity is limited to the specified site(s), limited to sites in a sharing group, or not limited at all.

To facilitate the passing of additional parameters, many of the methods in the atg.service.collections.filter.CachedCollectionFilter and atg.service.collections.filter.ChainedFilter filtering classes accept a pExtraParameters map. pExtraParameters is just a map that allows you to pass additional information to a filter so that it can do its work. All of these methods are overloaded with two signatures, one that uses the pExtraParameters map and another that doesn't. The method that uses pExtraParameters contains the logic required to perform the given tasks. The method that doesn't include pExtraParameters simply calls the method that does and passes in a null map. This use of overloading allows code that calls the original methods to continue to function properly while providing the flexibility to include additional filter parameters when needed.

You can choose to populate the `pExtraParameters` map in any way that makes sense for your implementation, although typically it is done through a combination of servlet bean properties and JSP settings. The example below describes how to pass site ID and site scope parameters to `/atg/registry/CollectionFilters/GiftlistSiteFilter`, a component that filters gift lists and gift items in a multisite environment. A servlet bean, `/atg/registry/CollectionFilters/GiftlistSiteFilterDroplet`, creates the `pExtraParameters` map with the site ID and site scope, and passes it along when it calls `GiftlistSiteFilter`.

- The keys of the map are specified by configuring an `extraParametersName` property in the servlet bean's `.properties` file. For example, the `GiftlistSiteFilterDroplet.properties` file defines two keys in the `pExtraParameters` map, `siteIds` and `siteScope`:

```
$class=atg.service.collections.filter.droplet.CollectionFilter
$scope=global
updateCache=true
consultCache=true
extraParameterNames=siteIds,siteScope
filter=/atg/registry/CollectionFilters/GiftlistSiteFilter
```

- The values for each key are set in the JSP. The following example sets values for the `siteIds` and `siteScope` keys defined by the `GiftlistSiteFilterDroplet` servlet bean (note that the `GiftlistSiteFilter` uses default values when `siteIds` or `siteScope` are not explicitly set by the servlet bean):

```
<dsp:droplet name="/atg/commerce/collections
/filter/droplet/GiftlistSiteFilterDroplet">
  <&-- Specify the collection to filter, the site context, and the
       site scope. --%>
  <dsp:param name="collection" bean="Profile.giftlists"/>
  <dsp:param name="siteIds" value="siteA,siteB"/>
  <dsp:param name="siteScope" value="current"/>

  <dsp:oparam name="output">

    <%-- The droplet stores the filtered list it in the JSP parameter
         filteredCollection. The getvalueof tag transfers that value
         to the JSTL variable filteredGiftlists. --%>
    <dsp:getvalueof var="filteredGiftLists" param="filteredCollection" />

    <%-- See if the new collection is empty. If not, iterate
through it. --%>
    <c:if test="${not empty filteredGiftLists}">
      <c:forEach var="giftlist" items="${filteredGiftLists}">

        <%-- Do whatever you want to do with each giftlist in here. --%>
```

```
          </c:forEach>
        </c:if>
      </dsp:oparam>
   </dsp:droplet>
```

- Using this key and value information, the servlet bean's `service()` method builds the `pExtraParameters` map and passes the map when it calls the filter. In cases where there are no extra parameters in the request, the `service()` method calls the original version of the method that doesn't use the `pExtraParameters` map.

Out of the box, the only filter ATG Commerce includes that uses the `pExtraParameters` map is the `GiftlistSiteFilter` filter, mentioned above. You can find additional information on this filter, and the servlet bean that calls it, in the *Filtering Gift Lists* section of the *ATG Commerce Programming Guide*. Keep in mind that, while the `pExtraParameters` map was initially developed to support site-aware filters, it can be used to pass any additional parameters that your filters require.

23 Using Profile Markers

A profile marker is a marker RepositoryItem that attaches to a user profile. Use profile markers when you want to “mark” a user’s profile with some information. A user might perform some action, for example navigate to one of your sites by way of an advertisement on another site. If you were to create a mechanism that extracts the site name and advertisement ID from the URL, when the user browses to your site, that information could be stored in a marker on the user’s profile. Any markers added to a user’s transient profile will be copied to his or her permanent profile automatically when he or she registers or logs in. You can design your sites to change their appearance or behavior based on the profile markers in a user profile. You can also gather data that could be used for comparing the number of users diverted to your sites from specific partner sites in order to determine which partner is most helpful to you.

Another use of profile markers is demonstrated in the business process tracking feature. Each business process is made up of stages. When a user reaches a stage, a marker is assigned to his or her profile. For more information, see Defining and Tracking Business Processes.

This chapter describes how to work with profile markers, which are markers assigned to profiles. Markers can be assigned to any RepositoryItem as long as that RepositoryItem has a property for holding markers, a Marker Manager component configured appropriately, and mechanisms are in place for adding, removing and locating markers.

You can store any information in a profile marker that’s relevant to your application. Each profile marker has several properties, the most important of which are key, value and data. You should use these properties to hold string values that define the marker or represent the particular circumstances in which the marker is assigned.

The key property distinguishes one kind of profile marker from another. Consider the example described above about the user who browses to a site from a partner site. In this instance, a key might be set to partner and value to travelSiteA. It’s a good idea to think of the key as a marker type, which will be used as an organizing principle for the marker data you are collecting. The key property might be a superset of the value.

Similarly, the data property might be a subset of value. The data property might hold a request parameter specifying the particular banner link on travel site A that directed the user to your site. To find a particular marker on a profile, you provide its key (partner), value (travelSiteA) and data (image2399), although if you are looking for any marker with a specific key, you need only provide that key.

Assigning a marker to a profile is one way to store non-standard information in a profile; the other way is to create custom profile properties. Custom properties typically hold one piece of data, meaning you’ll need one custom property for each type of data you want to store. When you use markers, several can be stored in one profile property. Markers have keys, each of which can signify a specific application purpose and are used when sorting markers Also, there are several conditions, actions, events, and servlet beans designed to create, locate, and remove markers from profiles in an efficient process. Unless the information you gather is simple and limited to one purpose, in most cases you should use markers instead of creating custom properties.

Here's how profile markers work:

1. Configure a Profile Marker Manager. When a scenario determines that a profile marker needs to be created, the scenario prompts a Profile Marker Manager to create a marker instance and assign it to a particular profile. The Profile Marker Manager facilitates all interaction between markers and profiles, such as creating markers and removing them from profiles as well as querying profiles for specific markers.

   See Configuring the Profile Marker Manager.

2. Create one or more scenarios that define when markers will be created and attached to a profile. Rather than creating a scenario for this purpose, you can code a JSP to use a servlet bean that accomplishes the same task.

   See Marking a User Profile.

3. Once you have marked profiles, your application can check profiles for specific markers or use scenarios that respond based on the markers in a profile. Create scenarios (or use servlet beans) for this purpose.

   See Using Marked Profiles.

4. To ensure optimal performance, it's a good idea to create scenarios (or use servlet beans) that remove markers when you no longer need them.

   See Removing Profile Markers.

For information on how to accomplish these tasks directly through the API, see the `atg.markers` package in the *ATG API Reference*.

Configuring the Profile Marker Manager

The Profile Marker Manager is a component that manages how markers interact with profiles. When your application adds a new marker, removes an existing one, or queries for information about the markers on a profile, the Profile Marker Manager coordinates these actions by identifying the marked item(`profile`), the property on that item that holds markers (`markers`), and the type of item that will act as a marker (`marker`).

When adding markers to a profile, the Profile Marker Manager applies the marker duplication and validation rules you configure it to use. These settings and others specified in the Profile Marker Manager are considered to be default values and are overridden when contrary values are provided in a scenario or servlet bean.

By default, one Profile Marker Manager called `atg/markers/userprofiling/ProfileMarkerManager` manages all activities relating to profile markers. You may choose to create additional Profile Marker Manager components each of which would be designed to use the particular validation and duplication rules necessary for a particular type of profile marker. Another reason for having multiple Profile Marker

Managers is so markers can be saved to more than one profile property. Each Profile Marker Manager works with one particular profile property.

To configure the Profile Marker Manager, see the following sections:

Setting a Duplication Mode

Setting up Marker Validation

Defining Profile Marker Manager Properties

Setting a Duplication Mode

The Profile Marker Manager specifies a default duplication mode that is used for the markers it creates when the scenario (or servlet bean) used to add the marker doesn't provide an alternate mode. The Profile Marker Manager accepts three values for the `defaultDuplicationMode` property:

- `ALLOW_DUPLICATES`
- `REPLACE_DUPLICATES`
- `NO_DUPLICATES`

When a Profile Marker Manager is prompted to create a marker that is identical to one that already exists in a given profile, the Profile Marker Manager can discard the new marker (`NO_DUPLICATES`), replace the original with the duplicate (`REPLACE_DUPLICATES`),or accept the new marker in addition to the original (`ALLOW_DUPLICATES`). Fewer markers cause quicker response in queries for a particular marker in a profile, but may slow the marker creation and removal processes.

You can change your duplication mode at any time. If the Profile Marker Manager previously permitted duplicate markers (`ALLOW_DUPLICATES`), you can adjust it to replace all old markers with a new duplicate (`REPLACE_DUPLICATES`).

When you specify a preference for unique markers (`NO_DUPLICATES` or `REPLACE_DUPLICATES`), the Profile Marker Manager relies on a compare component to determine whether a new marker is a duplicate of any existing ones. By default, the `defaultMarkerDuplicateComparator` property is set to `/atg/markers/CompareByDefaultProperties`, which identifies two markers as identical when they have the same values in their `key`, `value`, and `data` properties respectively. ATG 10.0.2 comes with another component, `/atg/markers/userprofiling/CompareByKeyAndValue` that judges duplication based on the values in the `key` and `value` properties.

Defining Uniqueness

You can create your own components that judge uniqueness based on the marker properties you identify or the business logic you define. One approach is to provide multiple compare components, each of which is tailored to the uniqueness criteria required for a particular type of profile marker (as identified by marker `key`).

To create a compare component that determines equality based on marker property values:

1. Create a component of class `atg.markers.CompareByProperties`.
2. Set the `propertiesTocompare` property to the marker property or properties that will be compared to determine equality.

3. If you have multiple compare components, update the markerDuplicateComparators map property to use each one: set the key to the profile marker key value and the value to a compare component.

To create a compare component that determines equality based on the business logic you define:

1. Create a subclass of atg.markers.MarkerDuplicateComparator and create a component of that class.
2. If you have multiple compare components, update the markerDuplicateComparators map property to use each one: set the key to the profile marker key value and the value to a compare component.

Setting up Marker Validation

When your application creates a profile marker, the only information you are required to provide is a value to the key property. You may prefer that your profile markers conform to stricter standards, for example: all markers must have a key value that's at least six characters. Or, your validation requirements may be based on application-specific business logic that is unrelated to marker properties. Create a validation class that enforces the validation standards appropriate for your profile markers.

The validation standards you define might vary based on the type of profile marker: profile markers with key X must have a value for key and value where as profile markers with key Y must have a Date value for expirationDate, which is a custom property you defined. You must match each validation component to a key; there is no way to create one validation component for all profile markers created by a given Profile Marker Manager.

Follow these steps to define new validation standards:

1. Create a class for each set of validation standards. This class should implement the atg.markers.MarkerValidatorContainer interface. Create a component of each class.
2. Update the Profile Marker Manager markerValidators map property to use each validation component you defined: set the key to the profile marker key value and the value to a validation component. For example:

   ```
   markerValidators=X=/atg/KeyXValidator,Y=/atg/KeyYValidator
   ```

1. Toggle the Profile Marker Manager alwaysValidate property to true only if you have defined a validation component for each possible key. If not, keep this set to false: those profile markers that are mapped to validation components will use them regardless.
2. If your validation standards require profile marker properties in addition to key, value, and data to be populated, add those property names to the Profile Marker Manager component requiredExtendedProperties property.

Defining Profile Marker Manager Properties

The Profile Marker Manager has the following configurable properties:

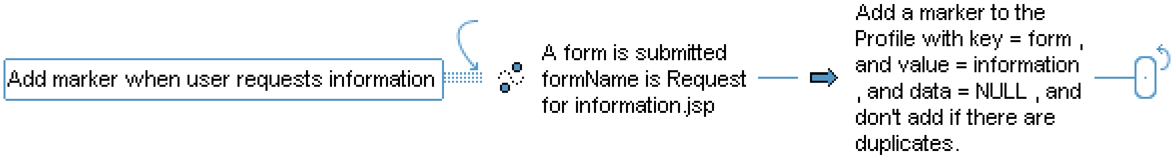
Property	Description
addMessageJMSType **Default value**: atg.profile.marker.added	The type of JMS message that is sent when a marker is attached to a profile.
alwaysValidate **Default value**: false	Indicates whether validation standards are defined for all marker keys. When set to false, profile markers use validation components when their key is mapped to a validation component. Set this property to true only if all profile markers are mapped to use validation components. These mappings are specified in the markerValidatiors property.
defaultDuplicationMode **Default value**: ALLOW_DUPLICATES	The mode that specifies whether duplicate markers can exist on a profile. Options include: - ALLOW_DUPLICATES - REPLACE_DUPLICATES (new marker replaces original marker) - NO_DUPLICATES (original marker remains)
defaultMarkedItemType **Default value**: /atg/userprofiling/ProfileTools.defaultProfileType	The type of RepositoryItem to which this Profile Marker Manager attaches markers.
defaultMarkerDuplicateComparator **Default value**: /atg/markers/CompareByDefaultProperties	The component used to determine if one marker is the same as any other on a given profile.
defaultMarkerItemType **Default value**: marker	The type of marker RepositoryItem to attach to a profile.
defaultMarkerPropertyName **Default value**: markers	The name of the Profile property that holds markers managed by this Profile Marker Manager.
defaultMarkerSortPropertyName **Default value**: creationDate	The marker property whose value is used to sort the markers on a profile when the profile property that holds markers is a Set and the markers it contains need to be ordered.

generateEvents **Default value**: false	Indicates whether the Profile Marker Manager generates ProfileMarkerAdded, ProfileMarkerRemoved, and ProfileMarkerReplaced events.
markerDuplicateComparators No default value	The map that specifies each profile marker key and the component used to determine marker uniqueness for markers with that key.
markerMessageSource **Default value**: /atg/markers/ RepositoryMarkerMessageSource	The component that sends messages generated by the Profile Marker Manager.
markerValidators No default value	The map that specifies each profile marker key and the component used to validate markers with that key.
profilePropertyNames **Default value**: markers, businessProcessMarkers	The names of the profile properties that can hold markers. When a user logs in, markers in these properties are copied from a transient profile to a permanent one.
removeMessageJMSType **Default value**: atg.profile.marker.removed	The type of JMS message sent when a profile marker is removed from a profile.
replaceMessageJMSType Default value: atg.profile.marker.replaced	The type of JMS message sent when one marker replaces another on a profile.
repository **Default value**: /atg/userprofiling/ ProfileAdapterRepository	The repository that contains both profiles and markers.
requiredExtendedProperties No default value	The extended profile marker properties to which values must be provided in order for new markers to be created. Use this property only if you extend the marker repository item to include new properties for which you require values upon creation.
transactionManager **Default value**: /atg/dynmo/TransactionManager	The Transaction Manager that controls the transactions used by the Profile Marker Manager.

Note: The Profile Marker Manager has one event listener property called `swapEventListener` that holds the listeners that will detect a `ProfileSwapEvent` after one is sent by a Profile Form Handler and detected by the Profile Marker Manager.

Marking a User Profile

To use the profile marker feature, you need to define the circumstances under which a user profile will be marked. You can create a scenario that uses an `AddMarkerToProfile` action to mark a profile as follows:



Add marker when user requests information

A form is submitted formName is Request for information.jsp

Add a marker to the Profile with key = form , and value = information , and data = NULL , and don't add if there are duplicates.

In this example, a profile marker is added to a profile when a user submits a request for information. For more information on the `AddMarkerToProfile` action, see the *Using Action Elements in Scenarios* section in the *Creating Scenarios* chapter of the *ATG Personalization Guide for Business Users*.

Another way to accomplish the same task is by using the `AddMarkerToProfileDroplet` servlet bean on a form success page, which is the page that displays after a form has been submitted without error. Your code would look like this:

```
<dsp:droplet
 name="/atg/markers/userprofiling/droplet/AddMarkerToProfileDroplet">
   <dsp:param name="key" value="form"/>
   <dsp:param name="value" value="information"/>
   <dsp:param name="duplicationMode" value="NO_DUPLICATES"/>

</dsp:droplet>
```

For more information on the `AddMarkerToProfileDroplet` servlet bean, see the *ATG Page Developer's Guide*.

Using Marked Profiles

Once profiles have markers, you can use them in a number of ways. You can design a JSP to display text or initiate some action depending on whether a user has a particular profile marker. Or, you can cause a scenario to advance only if some profile marker activity has occurred. Refer to the following topics for implementation instructions:

- Using Servlet Beans to Find Markers on Profiles

- Advancing a Scenario If a Profile Has Markers
- Advancing a Scenario Based on Profile Marker Events

Using Servlet Beans to Find Markers on Profiles

There are several servlet beans you can use to look up the markers on a given user profile. When you use these servlet beans, you can specify an input parameter that indicates the profile you want to work with. If you choose not to, the active user profile is used.

You can check a profile for a particular marker by specifying the marker's `key`, `value`, or `data` property values:

```
<dsp:droplet name="/atg/markers/userprofiling/droplet/ProfileHasMarkerDroplet">
   <dsp:param name="key" value="partner"/>
   <dsp:param name="value" value="travelSiteA"/>
   <dsp:param name="data" bean="ProfileHasMarkerDroplet.ANY_VALUE"/>

   <dsp:oparam name="false">
      Check out our partner sites!
   </dsp:oparam>

</dsp:droplet>
```

The `ProfileHasLastMarkerDroplet` servlet bean locates the last marker attached to a given profile. If you'd like, you can indicate that the last marker should be returned only if it has the particular `key`, `value`, `data`, or other marker property value you specify. This servlet bean lets you access the marker itself. For example:

```
<dsp:droplet
 name="/atg/markers/userprofiling/droplet/ProfileHasLastMarkerDroplet">
   <dsp:param name="key" value="partner"/>
   <dsp:param name="value" bean="ProfileHasLastMarkerDroplet.ANY_VALUE"/>
   <dsp:param name="data" bean="ProfileHasLastMarkerDroplet.ANY_VALUE"/>

   <dsp:oparam name="true">
      <dsp:valueof paramvalue="marker.value">
      is one of our favorite affiliates. Learn why!
   </dsp:oparam>

</dsp:droplet>
```

Finally, the `ProfileHasLastMarkerWithKey` lets you find the last marker added to a profile that has a particular `key`. Other marker property values can act as parameters for this servlet bean as well.

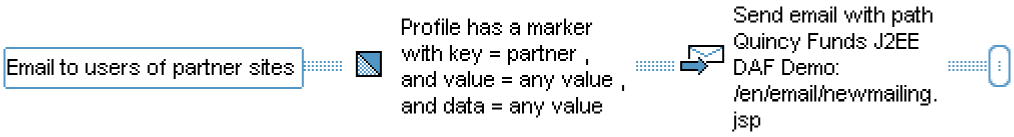
```
<dsp:droplet
 name="/atg/markers/userprofiling/droplet/ProfileHasLastMarkerWithKeyDroplet">
   <dsp:param name="key" value="partner"/>
   <dsp:param name="value" value="travelSiteA"/>
   <dsp:param name="data" bean="ProfileHasLastMarkerWithKeyDroplet.ANY_VALUE"/>

   <dsp:oparam name="true">
      When you book your hotel room with us, you'll receive a discount on air fare
      if you book your flight with travel site A!
   </dsp:oparam>

</dsp:droplet>
```

For more information on the servlet beans described here, see *Appendix B: ATG Servlet Beans* of the *ATG Page Developer's Guide*.

Advancing a Scenario If a Profile has Markers

Use profile marker conditions in scenarios when you want the scenario to advance only if the active user has specific markers. There are three profile marker-related conditions available to you.

When you want a scenario to respond based on the existence of profile markers on a profile, use the `Profile has a marker` condition:



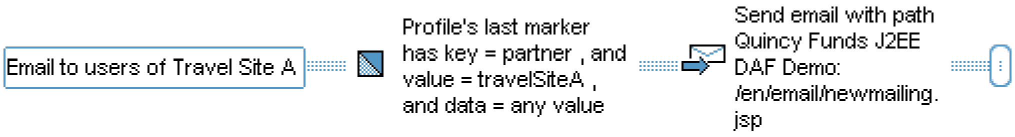
Email to users of partner sites

Profile has a marker
with key = partner ,
and value = any value ,
and data = any value

Send email with path
Quincy Funds J2EE
DAF Demo:
/en/email/newmailing.
jsp

In this example, users who have a profile marker with a key of `partner` will receive an email. You are given the option to set `value` and `data` to `any value` so that the values in these properties aren't used to narrow the scope of the condition.

The `Profile's last marker` condition lets you indicate that only users whose last marker has the `key`, `value`, and `data` values you specify should advance to the next scenario element. For example:

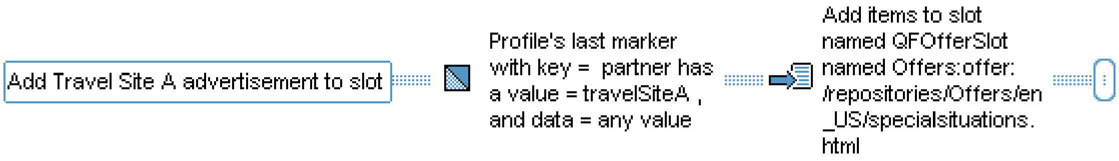


Email to users of Travel Site A

Profile's last marker
has key = partner , and
value = travelSiteA ,
and data = any value

Send email with path
Quincy Funds J2EE
DAF Demo:
/en/email/newmailing.
jsp

A third condition called `Profile's last marker with key` determines that only users whose last marker with a particular `key` has the `value` and `data` values specified here may advance to the next scenario element. This example demonstrates that users whose last partner profile marker has a value of `travelSiteA` will see an advertisement for that travel site:


Add Travel Site A advertisement to slot
Profile's last marker with key = partner has a value = travelSiteA , and data = any value
Add items to slot named QFOfferSlot named Offers:offer: /repositories/Offers/en_US/specialsituations. html

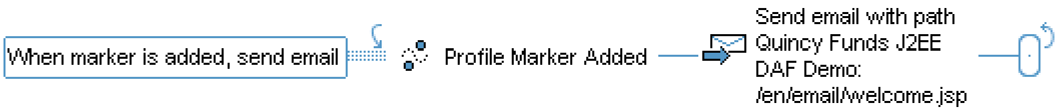
For more information on the actions described here, see the *Using Condition Elements in Scenarios* section in the *Creating Scenarios* chapter of the *ATG Personalization Guide for Business Users*.

Advancing a Scenario Based on Profile Marker Events

ATG 10.0.2 includes three scenario elements that listen for the following events:

- a marker being added to a profile (`Profile Marker Added`)
- a marker being removed from a profile (`Profile Marker Removed`)
- one marker being replaced on a profile with another marker (`Profile Marker Replaced`)

You create a scenario that advances when it hears a profile marker event. Here's a scenario that listens for a profile added event and when it detects one, sends an email:


When marker is added, send email
Profile Marker Added
Send email with path Quincy Funds J2EE DAF Demo: /en/email/welcome.jsp

For more information on the events described here, see the *Using Event Elements in Scenarios* section in the *Creating Scenarios* chapter of the *ATG Personalization Guide for Business Users*.

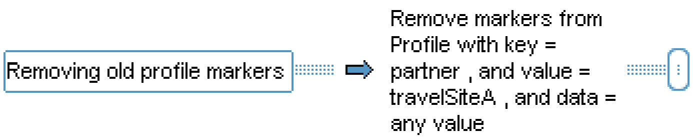
Removing Profile Markers

It's a good idea to set up a mechanism that removes profile markers when they are no longer relevant to your application. You may use markers to monitor some user behavior, but once you've gathered the data you need, flush markers from the profiles in one of the ways described here.

In general, you remove markers from a given profile either by removing specific markers or all of them at once. If, instead, you want to remove markers from all profiles simultaneously, invoke the `deleteMarkers` method on the `atg.markers.userprofiling.ProfileMarkerManager` class. See *ATG API Reference* for more information.

Specific Profile Markers From a Profile

In order to remove specific markers from an active user profile, create a scenario that uses the `Remove markers from profile` action and specify the `key` used by the markers you want to remove. You can provide additional information such as a `value` or `data` value: markers that have the values for the properties you provide will be removed from the active user profile. For example, remove all `partner` profile markers that have a value of `travelSiteA`:



Removing old profile markers

Remove markers from Profile with key = partner , and value = travelSiteA , and data = any value

For more information on the `Remove markers from profile` action, see the *Using Action Elements in Scenarios* section in the *Creating Scenarios* chapter of the *ATG Personalization Guide for Business Users*.

You can also delete particular markers from a profile using the `RemoveMarkersFromProfileDroplet`:

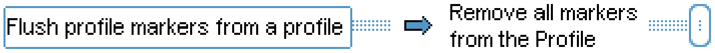
```
<dsp:droplet
 name="/atg/markers/userprofiling/droplet/RemoveMarkersFromProfileDroplet">
   <dsp: param name="key" value="partner"/>
   <dsp: param name="value" value="travelSiteA"/>
   <dsp:param name="data" bean="RemoveMarkersFromProfileDroplet.ANY_VALUE"/>

   <dsp:oparam name="output">
      Book your flight with travel site B!
   </dsp:oparam>

</dsp:droplet>
```

When you remove markers using the servlet bean, you choose the profile from which you want to remove markers. Although the default is the active user profile, it's possible to specify a different one. Also, in addition to using the `key`, `value`, and `data` property values to identify the marker you want to remove, you can further define the marker by including other marker properties and their values.

For more information on the `RemoveMarkersFromProfileDroplet` servlet bean, see *Appendix B: ATG Servlet Beans* of the *ATG Page Developer's Guide*.

All Profile Markers on a Profile

The quickest way to flush markers from a profile is to remove all at once. In order to remove all markers from an active user profile, design a scenario that uses the `Remove all markers from the profile` action:


Flush profile markers from a profile
Remove all markers from the Profile

For more information on the `Remove all markers from the profile` action, see the *Using Action Elements in Scenarios* section in the *Creating Scenarios* chapter of the *ATG Personalization Guide for Business Users*.

You can also remove all markers from a profile using the `RemoveAllMarkersFromProfileDroplet`:

```
<dsp:droplet
 name="/atg/markers/userprofiling/droplet/RemoveAllMarkersFromProfileDroplet">

  <dsp:param name="output">
     There were <dsp:param name="markerCount"/> markers on your
     profile.
  </dsp:param>

</dsp:droplet>
```

Using this servlet bean to remove markers gives you more control over the items you remove than the corresponding scenario action. As demonstrated in the servlet bean example, you can find out the number of markers that are removed and display it to users. You can also specify the ID for the profile containing markers you want removed or rely on the default value, which causes markers from the active user profile to be removed.

For more information on the `RemoveAllMarkersFromProfileDroplet` servlet bean, see *Appendix B: ATG Servlet Beans* of the *ATG Page Developer's Guide*.

24 Defining and Tracking Business Processes

ATG's Adaptive Customer Engine includes a business tracking feature that lets you define a business process as a series of stages, track activity within the business process, and report on the activity for a specified time frame. This enables you to track progress or failure to progress in the business process and to personalize a user's experience based on where they are in a business process.

How Business Process Tracking Works

A business process should be defined as a series of stages applied to business object. The business object is defined as a repository item type (for example, a profile or an order). As the business object reaches a new stage, the corresponding repository item is marked to indicate the process and stage reached, and a message is sent, identifying the business process, the repository item, and the new business process stage being reached.

Defining a Business Process

To define a business process and track the progress of items through the business process:

1. Identify the object of the business process (a profile, order, or other type of repository item).
2. Create and configure a `BusinessProcessConfiguration` component to define the business process. See Creating a BusinessProcessConfiguration Component.
3. Add the `BusinessProcessConfiguration` component to the `businessProcessConfigurations` property of the `BusinessProcessManager`. See Configuring the BusinessProcessManager Component.
4. Set up a process for adding business process stage markers to your business process object. You can do this using scenarios, servlet beans in JSPs, or directly through the API. See Marking Business Process Stages.
5. Create a scenario recorder to record business process stage reached events, together with reports derived from the dataset. See Reporting on Business Processes.

Creating a BusinessProcessConfiguration Component

Each business process is defined by a configuration component that extends the `atg.markers.bp.BusinessProcessConfiguration` abstract class. Two subclasses of the `BusinessProcessConfiguration` class already exist:

- `atg.markers.bp.ProfileBasedProcessConfiguration` tracks a profile through a business process. By default, a process of this type tracks the session's profile.
- `atg.commerce.markers.bp.OrderBasedProcessConfiguration` tracks an order through a business process. By default, a process of this type tracks the shopping cart's current order. This class is only available to ATG Commerce customers.

The configuration component defines the business process using the following properties:

Property Name	**Description**
`businessProcessName`	The name of the business process
`duplicationMode`	How to handle attempts to add duplicate markers. Valid values are: - `ALLOW_DUPLICATES` - no duplication check - `REPLACE_DUPLICATES` - delete all duplicate markers, and add the new marker - `NO_DUPLICATES` - if the marker already exists, discard the new one. This is the default value. Keeping the default (`NO_DUPLICATES)` means that once a business stage has been reached, no change is made if that stage is reached again.
`enabled`	Boolean used to indicate whether a business process is active. Disabling an active business process causes all associated tasks to halt. For example, the achievement of new stages in a disabled business process will not be tracked. When you re-enable a business process, it begins at the point to which it had stopped. Default is `true`.
`generateEvents`	Boolean used to determine whether a business process is able to process events. Default is `true`.
`markedItemType`	The type of `RepositoryItem` to which the markers are attached. This property is used for deleting all markers attached by a given business process. The default value is derived from the value of another component property: - For `ProfileBasedProcessConfiguration`, the value is provided by the `ProfileRepositoryMarkerManager.markedItemType` property. Typically, the value is `user`. - For `OrderBasedProcessConfiguration`, the value is provided by the `OrderTools.defaultOrderType` property. Typically, the value is `order` for Consumer Commerce, and `b2border` for Business Commerce.
`stageNames`	An array of the names of the stages that make up the business process.

The two important properties here are stageNames and duplicationMode. You need to define the stages of the business process that you might want to track. Make sure that your application design will allow you to mark these stages using a scenario, a servlet bean in a JSP, or directly using the business process tracking API. See Marking Business Process Stages and the atg.markers.bp package in the *ATG API Reference* for more information.

Configuring the BusinessProcessManager Component

Business processes are managed by a component of class atg.markers.bp.BusinessProcessManager. The BusinessProcessManager acts as a registry of business processes and handles creating, querying, and removing business process markers. It also sends business process event messages. A BusinessProcessManager component is configured out of the box at /atg/markers/bp/BusinessProcessManager. The BusinessProcessManager has the following configurable properties:

Property Name	Description
businessProcessConfigurations	An array of business process configuration components. There should be one for each business process managed by this component.
generateEvents	Boolean. Should we generate business process events? Default is true.

If you create a new business process, you need to add the BusinessProcessConfiguration component to the businessProcessConfigurations property of the BusinessProcessManager component.

Marking Business Process Stages

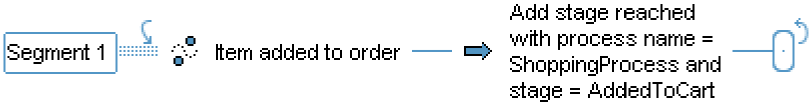
ATG includes page-based and scenario-based tools that let you add, remove, and check for business process stages. These include the following servlet beans and scenario elements:

Task	Servlet Bean	Scenario Element
Add a business process stage	AddBusinessProcessStage Servlet Bean	Add Stage Reached Action
Remove a business process stage	RemoveBusinessProcessStage Servlet Bean	Remove Stage Reached Action
Check if a business process stage has been reached	HasBusinessProcessStage Servlet Bean	Has Reached Stage

Check the most recent business process stage that has been reached	`MostRecentBusinessProcessStage` Servlet Bean	Most Recent Stage Reached

For reference information about the business process servlet beans, see *Appendix B: ATG Servlet Beans* in the *ATG Page Developer's Guide*. For information about the business process scenario elements, see Using Scenario Events and Using Scenario Actions chapters.

For example, you could create a scenario that uses the Add Stage Reached action to add the `AddedToCart` process stage when the user adds an item to an order:


Segment 1
Item added to order
Add stage reached with process name = ShoppingProcess and stage = AddedToCart

You could accomplish the same thing by including the `AddBusinessProcessStage` servlet bean to a commerce page:

```
<dsp:droplet name="AddBusinessProcessStage">
  <dsp:param name="businessProcessName" value="ShoppingProcess"/>
  <dsp:param name="businessProcessStage" value="AddedToCart"/>
</dsp:droplet>
```

Each business process stage servlet bean has a `businessProcessName` property. This property lets you create instances of the servlet bean that are specific to a single business process. If you so this, you can use elements like this:

```
<dsp:droplet name="AddRegistrationStage">
  <dsp:param name="businessProcessStage" value="ViewedTermsAndConditions"/>
</dsp:droplet>
```

This makes your pages easier to read and avoids the need to specify the business process each time.

Deleting Business Process Content

When you are no longer using a business process or content derived from it, you should delete the associated content. ATG 10.0.2 provides two API methods in the `atg.markers.bp.BusinessProcessManager` class for business process content deletion. The `BusinessProcessManager` has access to both the individual `RepositoryItems` involved in the business process as well as the governing business process itself so you can delete business process content from both contexts:

- removeBusinessProcessStage deletes business process content from a particular RepositoryItem
- deleteBusinessProcessMarkers deletes content provided by a given business process

Both methods take arguments

```
(pBusinessProcessName, pBusinessProcessStage)
```

where pBusinessProcessName is the name of a business process and pBusinessProcessStage is a business process stage. See atg.markers.bp.BusinessProcessManager in the *ATG API Reference* for more information.

RepositoryItem-Based Deletion

To delete business process content from a specific RepositoryItem, use removeBusinessProcessStage. For example, the following call would remove content provided to a RepositoryItem by StageA of a business process called MyBusinessProcess:

```
removeBusinessProcessStage("MyBusinessProcess","StageA")
```

It's also possible to remove content provided by all stages of MyBusinessProcess from a particular RepositoryItem using this call:

```
removeBusinessProcessStage("MyBusinessProcess",
"MarkerConstants.ANY_VALUE")
```

Business Process-Based Deletion

Alternatively, you can approach business process content deletion from the perspective of the business process. You can delete content associated with stages from a given business process on every affected RepositoryItem. The BusinessProcessManager refers to the BusinessProcessConfiguration.markedItemType property to assist in deleting business process content using this method.

Eliminate content added by a particular stage, such as StageB of MyBusinessProcess, from all RepositoryItems, as follows:

```
deleteBusinessProcessMarkers("MyBusinessProcess","StageB")
```

To delete content supplied by all stages of MyBusinessProcess from all RepositoryItems, use this code:

```
deleteBusinessProcessMarkers("MyBusinessProcess",
"MarkerConstants.ANY_VALUE")
```

Reporting on Business Processes

Tracking business processes allows you to generate, study, and learn from reports about what happens at different stages of the business process. You can detect when users are dropping out of a business process and take steps to encourage them to resume the business process.

To generate reports about business processes, you should:

1. Create a scenario recorder, including a scenario, data mapper, and dataset to store business process information. See Using Scenario Recorders.
2. Create reports based on the dataset.

25 Creating and Configuring Workflows

The Scenarios module includes a mechanism for modeling business processes called a **workflow**. Workflows are similar to scenarios, but can be applied to a wider range of processes. One key difference between scenarios and workflows is the capability for creating new types of workflows that are tailored to specific processes, and that include their own unique actions and tasks.

The tools for creating new workflow types are included in the Scenarios module, although the module is not preconfigured with any workflow types. ATG Content Administration provides a workflow that manages the lifecycle of a publishing project. You might find it helpful to use this workflow as an example.

Many different types of business processes can be exposed as workflows, with examples ranging from commerce order fulfillment to management of customer support calls. This chapter discusses the various aspects of creating a workflow type.

Overview of Workflows

Workflows, like scenarios, are based on ATG's process engine architecture. Most workflow classes are analogous to scenario classes.

A major difference, however, is that while you can create custom events and actions for scenarios, there is only a single scenario type. With the workflow classes, you can create any number of different workflow types, each with its own actions and tasks, repository items, and database tables. When you edit a workflow in the ACC, the editor is automatically configured to reflect the specific type of workflow you are editing.

Creating a Workflow Type

You create a workflow type as follows:

1. Choose the workflow subject type.

   The subject type is a repository item that represents the main element in the process that will be modeled by the workflow. For example, an expense report could be a workflow subject type, with the workflow modeling the steps involved in processing the report.

2. Create the repository items and database schema.

Each workflow type must define its own database schema and repository items. This provides isolation between the data stored for different types of processes, allowing for independent deployment and management of data associated with any given workflow type. For example, the data associated with registration workflows is stored in one set of tables, while the data for ATG Content Administration workflows resides in a different set of tables. The two sets of data can be managed independently (for example, you could clean out one set of tables completely without affecting the other).

When you create a new type of workflow, you create several new database tables, as well as the associated repository item descriptors. See the Registration Workflow section for information about the tables and repository items used by registration workflows.

3. Configure the workflow components.

   Each workflow type requires you to configure a number of Nucleus components and XML files, as discussed throughout this chapter.

4. Create the workflow definition in the ACC.

   For each workflow type, the workflow editor is customized to expose the relevant item type and its properties, and to display only those custom expressions and actions that have been configured for the corresponding subject type.

   To accomplish this, ATG 10.0.2 maintains a global registry of ACC workflow agent components at `/atg/registry/WorkflowAgents`. When you create and configure a new type of workflow, you must add the corresponding workflow agent component to this global registry. From the workflow agent, the ACC can determine the corresponding workflow subject type and all the custom process configuration information. Because the agent component also points to the registry in which the workflows themselves are stored, the ACC can load all of the existing workflow definitions corresponding to each subject type, and write out new workflow definitions to the appropriate registries.

   For information about editing workflows in the ACC, see the *ATG Personalization Guide for Business Users*.

Workflow Classes

The following are the classes and interfaces that you use to work with workflows:

atg.workflow.WorkflowManager

The primary interface that exposes features of the workflow system. The WorkflowManager can be queried to obtain global workflow status information as well as the underlying workflow process information (via the associated ProcessManager). It also supports creation of workflow views that expose user-specific features of the workflow system.

atg.workflow.WorkflowView

This interface exposes workflow features to a notional user of the workflow system, such as a specific user or Persona. A workflow view provides status information about workflow instances, tasks, and outcomes, allows new workflow instances to be started, and permits task outcomes to be fired to existing workflow instances. All of these operations are implicitly performed from the point of view of the user, and the view exposes only those workflow objects that are accessible to the user. It is also possible for a `WorkflowView` to be global, that is, not associated with any user. A global view performs no access control, thereby providing an unconstrained view of the workflow system.

atg.workflow classes and interfaces

Used to expose various workflow-related information. The descriptor classes (`WorkflowDescriptor`, `TaskDescriptor`, `OutcomeDescriptor`) describe workflows, their tasks, and outcomes. The `TaskInfo` class keeps track of the runtime information associated with workflow tasks. `TaskQueryOptions` is a helper class used by both `WorkflowManager` and `WorkflowView` when querying for tasks. The `WorkflowAccessRights` and `TaskAccessRights` interfaces expose the access rights associated with workflows and tasks. The `WorkflowConfiguration` class stores configuration information that is shared by multiple workflow components.

atg.workflow.servlet classes

Include the workflow servlet bean and form handler classes. These are described in more detail in the Workflow Servlet Beans and Workflow Task Form Handler sections.

atg.process.ProcessManager

This interface includes several methods that provide workflow support. Specifically, these are the `getProcessInstanceInfos` methods, which can be used to obtain runtime information about the process instances currently going through a process.

atg.process classes

Support ProcessManager methods and their use by the workflow system: `ProcessInstanceInfo`, `ProcessElementInfo`, and `ProcessWaitState`.

For more information about these classes and interfaces, see the *ATG API Reference*.

Shared Components

Creating a workflow type involves creating and configuring many components and XML files that are specific to the workflow type, as shown in the Configuring the Registration Workflow Type section. In addition, there are several workflow-related Nucleus components that are used by all custom workflow types. These components require no additional configuration, but are available for other components to make reference to as part of their configuration:

/atg/registry/factories/WorkflowFactory

A UI-related component that is shared by all workflow types. The `factory` property of each workflow registry must point to this component (for example, the `/atg/registry/RegistrationWorkflows` component shown in the Registration Workflow section).

/atg/registry/filters/WdlFilter

A filter for `.wdl` files. The `filter` property of each workflow registry must point to this component (for example, the `/atg/registry/RegistrationWorkflows` component shown in the Registration Workflow section).

/atg/workflow/process/ components

An e-mail sending component (`WorkflowEmailSender`) and the associated template e-mail information (`DefaultTemplateEmailInfo`). These are used by the `emailNotify` and `emailNotifyTaskActors` standard workflow actions.

/atg/workflow/process/configuration/ components

These supply the default configurations for some of the standard workflow actions. There are five standard actions that come with the Scenarios module and that can be included in the workflow process manager configuration for any workflow type: `emailNotify`, `emailNotifyTaskActors`, `recordEvent`, `recordAuditTrail`, and `deleteSubject`. The `EmailNotifyConfiguration` component configures the `emailNotify` action; the `RecordActionConfiguration` component configures the `recordEvent` and `recordAuditTrail` actions. Note that the `emailNotifyTaskActors` and `deleteSubject` actions do not have default configurations shared by all workflow types, because they must be configured separately for each workflow type, as demonstrated in the Registration Workflow section.

Registration Workflow

This section uses the EcoVida registration workflow type to show how to create and configure a workflow type. The EcoVida demo application was provided with earlier versions of some ATG products. Although the demo is not included in the ATG 10.0.2 installation, you can download the configuration files for its workflow module (`EcoVida.workflow`) from the Developer Network on ATG's Web site. Be aware that the module was designed to work with ATG 6 products, so running it against later ATG environments is not supported. However, the way a workflow type is configured has not changed substantially in ATG 10.0.2, so looking at the registration workflow's configuration files can still be very helpful for understanding and debugging workflow setup.

This workflow type models the process of registering a new partner (builder or reseller) in a portal community:

1. The prospective reseller comes to the EcoVida partner portal and submits an application.
2. The EcoVida portal initiates an approval workflow.
3. Because the partner is in the east region, the application is routed to the Eastern regional channel manager for review. Mandy, the channel manager, claims the task.

4. Mandy completes the reseller approval task.
5. The workflow then routes the registration request to a finance manager to perform a credit check. Frank, a finance manager, claims the task and does a credit check.
6. Frank approves the reseller.
7. Mandy is notified that the partner has passed the credit check and been approved.
8. A personalized e-mail is sent to the partner informing them of their approval.
9. The workflow ends.

To implement this use case, EcoVida includes a company registration page, where a prospective partner can specify the company's name, address, region, and contact information, as well as bank account information. A registration approval workflow is initiated upon the completion of the registration form. The subject of the registration workflow is the registration request, which is a repository item (of type `company-registration-request`) created upon the form submission.

The workflow consists of two tasks, and the associated outcomes and actions. The first task is the channel review, initially assigned to the `channel-manager` role. Upon approval, the workflow progresses to the credit check task, which is assigned to the `finance-manager` role. If the company passes the credit check, a series of custom actions are executed, which create all the necessary partner accounts, including the new organization and organizational roles, a portal community, and a user account for the company's contact.

Various e-mail messages are sent to the user or the channel manager depending on the outcomes of the two tasks. Finally, the registration request is deleted, and the workflow ends.

In order for the channel review task to be executed by the channel manager associated with the appropriate region, EcoVida defines three profile groups: `ChannelManagerEast`, `ChannelManagerCentral`, and `ChannelManagerWest`. The registration request is dynamically routed to the appropriate profile group depending on the zip code submitted during registration.

EcoVida also includes a `registration_approval` gear that allows EcoVida employees to perform tasks along the registration approval workflow. This gear is accessible only to channel managers and Finance. It allows the user to display all of the active workflow tasks that are accessible to him or her, claim and release tasks, and execute task outcomes. In this simple UI, once a task has been claimed by the user, only he or she can execute the task. The user can also release the task back into the pool, to allow others to claim and execute it.

Configuring the Registration Workflow Type

The `EcoVida.workflow` module, obtainable from the ATG Developer Network, includes all the configuration files necessary to define a new workflow type. The following is the list of these files:

```
/atg/
    dynamo/
        messaging/
            MessagingManager.properties
            SqlJmsProvider.properties
            dmsRegistrationWorkflowGlobal.xml
            dynamoMessagingSystem.xml
        security/
            admin-accounts.xml
    registry/
        RegistrationWorkflows.properties
        WorkflowAgents.properties
        data/
            workflows/
                registration/
                    Registration.wdl
    workflow/
        registration/
            WorkflowConfiguration.properties
            WorkflowManager.properties
            WorkflowMessageSource.properties
            WorkflowSubjectAccess.properties
            WorkflowSubjectLookup.properties
            WorkflowTaskFormHandler.properties
            WorkflowTaskQuery.properties
            WorkflowView.properties
            process/
                WDLParser.properties
                WorkflowAgent.properties
                WorkflowClusterManager.properties
                WorkflowDefinitionRegistry.properties
                WorkflowProcessManager.properties
                WorkflowSubjectFinder.properties
                workflowProcessManager.xml
                configuration/
                    AddUserToCommunityConfiguration.properties
                    CreateAccountsConfiguration.properties
                    CreateCommunityConfiguration.properties
                    CreateOrganizationConfiguration.properties
                    DeleteResgistrationConfiguration.properties
                    EmailNotifyTaskActorsConfiguration.properties
                    SetHomeCommunityConfiguration.properties
    userprofiling/
        userprofile.xml
    ecovida/
        commerce/
            b2b/
                profile/
                    CompanyRegistrationFormHandler.properties
```

You can use the registration workflow configuration files as a template for creating your own workflow types. Most of the properties configured by these files are the same for all workflow types. The properties you may need to modify are discussed below.

/atg/dynamo/messaging/

The MessagingManager.properties file adds the workflow process manager for the registration workflow type to the fileCombiners property:

```
filecombiners+=/atg/workflow/registration/process/WorkflowProcessManager
```

The SqlJmsProvider.properties file adds the SQL JMS topics and queues used by the registration workflow type to the requiredTopicNames and requiredQueueNames properties:

```
requiredTopicNames+=\
   sqldms/RegistrationWorkflowTopic/WorkflowUpdateEvents,\
   sqldms/RegistrationWorkflowTopic/WorkflowMigrationUpdateEvents
requiredQueueNames+=\
   sqldms/RegistrationWorkflowQueue/IndividualTimerEvents,\
   sqldms/RegistrationWorkflowQueue/CollectiveTimerEvents,\
   sqldms/RegistrationWorkflowQueue/BatchTimerEvents
```

The XML files in /atg/dynamo/messaging/ are the Patch Bay configuration files necessary for hooking up the WorkflowProcessManager and WorkflowMessageSource components as message sources and sinks. The standard Patch Bay configuration file, dynamoMessagingSystem.xml, is used on all process servers. The other Patch Bay configuration file is used on global process servers only, and is XML-combined with the standard configuration file on startup. For the registration workflow type, this file is named dmsRegistrationWorkflowGlobal.xml.

/atg/dynamo/security/

The admin-accounts.xml file specifies security privilege information for the ACC task that is used for editing workflows.

/atg/registry/

RegistrationWorkflows.properties creates a component of class atg.service.registry.SimpleRegistry that sets the root path for storing registry files for the workflow type:

```
configurationRootPath=/atg/registry/data/workflows/registration
```

The WorkflowAgents component is the registry (used by the ACC) of all workflow agents. You add the agent components of your custom workflow types to the workflowAgentPaths property of this component. (By default, this list is empty.) The WorkflowAgent.properties file includes this line:

```
workflowAgentPaths+=/atg/workflow/registration/process/WorkflowAgent
```

/atg/registry/data/workflows/registration/

The Registration.wdl file stores information used by the ACC. It is created automatically by the ACC.

/atg/workflow/registration/

The /atg/workflow/registration/ components include the WorkflowManager itself and the associated helper components.

The WorkflowConfiguration component has a number of properties that other components make reference to. The WorkflowConfiguration.properties file includes:

```
subjectRepository=/atg/userprofiling/ProfileAdapterRepository
subjectType=company-registration-request
processRegistry=/atg/registry/RegistrationWorkflows
workflowViewPath=/atg/workflow/registration/WorkflowView
processManagerConfigurationFile=
   /atg/workflow/registration/process/workflowProcessManager.xml
dmsGlobalConfigurationFile=
   /atg/dynamo/messaging/dmsRegistrationWorkflowGlobal.xml
```

WorkflowMessageSource is a message source that fires all workflow-related JMS messages for registration workflows.

The WorkflowSubjectAccess component provides access to the workflow's task information data. The component is configured to point to the workflow subject type and the repository it is stored in:

```
subjectRepository^=WorkflowConfiguration.subjectRepository
subjectType^=WorkflowConfiguration.subjectType
```

The WorkflowView component implements the atg.workflow.WorkflowView interface described in the Workflow Classes section; it provides session-scoped access to workflow features. This component is optional, as it is also possible to create workflow views programmatically using the WorkflowManager API, but it provides a convenient way to access workflow information from the point of view of the current user.

The WorkflowTaskQuery and WorkflowSubjectLookup servlet beans, and the request-scoped WorkflowTaskFormHandler, are optional as well; they can be used to construct custom workflow UIs. The registration approval gear uses the WorkflowTaskQueryDroplet to display active tasks accessible to the current user. It then uses the WorkflowTaskFormHandler to operate on these tasks: claim and release tasks, and fire outcomes. All of this is done in JSPs, with no need for the direct use of the workflow API, or custom code. See the Workflow Servlet Beans and Workflow Task Form Handler sections for more information about these components.

/atg/workflow/registration/process/

The /atg/workflow/registration/process/ components include the WorkflowProcessManager and the associated helper components.

The WorkflowAgent component is configured to point to the registry component for the workflow type:

```
processRegistry=/atg/registry/RegistrationWorkflows
```

The process manager XML configuration file, workflowProcessManager.xml, defines all of the standard workflow events and actions, plus the custom actions specific to registration workflows. It also contains configuration information for customizing the registration workflow editor in the ACC.

/atg/workflow/registration/process/configuration/

The /atg/workflow/registration/process/configuration/ components configure the emailNotifyTaskActors and deleteRegistration standard workflow actions, plus the custom actions createAccounts, createOrganization, createCommunityFromTemplate, addUserToCommunity, and setHomeCommunity. Note that you can find the source code for these custom actions, along with other Java source files, in the src/ subdirectory of the Ecovida.workflow module.

/atg/userprofiling/

The registration repository item types are part of the user profile repository, and are therefore defined in the file /atg/userprofiling/userProfile.xml. (Other workflow types may use other repositories.) This file is XML-combined with the other user profile configuration files, and defines all the necessary workflow-related item types. In addition, it defines two new properties for the company-registration-request item type: workflowInstances (pointing to a set of individual workflow instances), and workflowTaskInfos (pointing to a set of WorkflowTaskInfo items). A third property, id, is also required for workflows, but is not defined in this file because the company-registration-request item type already has an id property. All these properties must be defined for a subject type in order for the workflow and process engines to function correctly.

In addition to the process-related item descriptors, the workflow repository schema also defines several additional item types that are required for all workflow types. One of these is the workflowTaskInfo item descriptor, whose purpose is to store workflow task-related information on a per-subject basis. The intent is for generalized workflow subjects to have a workflowTaskInfos property pointing to a set of workflowTaskInfo items. The other item types are as follows:

```
individualWorkflow
collectiveWorkflow
workflowInfo
workflowMigrationInfo
collectiveWorkflowTransition
individualWorkflowTransition
workflowDeletion
workflowMigration
workflowServerId
```

For information about the properties of these item types, see the /atg/userprofiling/userProfile.xml file in the Ecovida.workflow module.

The names for these item types are specified by the WorkflowProcessManager component. If you want to use different names in your repository, you must change the corresponding properties of that component.

As mentioned above, each workflow type must have its own set of database tables. The script for creating the tables for the registration workflow is <ATG10dir>/EcoVida/sql/db_components/*<database-vendor>*/registration_workflow_ddl.sql.

/atg/ecovida/commerce/b2b/profile/

The CompanyRegistrationFormHandler.properties file creates a component of class atg.ecovida.commerce.b2b.profile.CompanyRegistrationFormHandler. This class contains the code that initiates the workflow and performs dynamic routing of the channel review task by region. This code is executed after the registration request item has been created, in the form handler's postCreateItem method. It demonstrates how the workflow API can be used both to start the workflow and to affect its tasks.

Workflow Servlet Beans

The workflow API includes two servlet beans that you can use to create custom UIs for manipulating workflows, atg.workflow.servlet.WorkflowTaskQueryDroplet and atg.workflow.servlet.WorkflowInstanceQueryDroplet. Note that the Scenarios module includes these classes but does not include any Nucleus components of these classes. Each component of one of these classes is typically customized (by setting certain properties) to work with a specific workflow type. For example, the registration workflow type uses a customized version of WorkFlowTaskQueryDroplet.

The configuration for each new workflow type typically includes some of the workflow-related servlet bean components. This section describes the two workflow servlet beans, plus two other servlet beans that may be used to facilitate the development of custom workflow UIs. For more information about these servlet beans, see the *ATG Page Developer's Guide*.

WorkflowTaskQueryDroplet

The WorkflowTaskQuery component in the registration workflow is a servlet bean of class atg.workflow.servlet.WorkflowTaskQueryDroplet. You can use this servlet bean to perform workflow task queries; for example, you could create a UI that displays the ID of the workflow subject and the names of all tasks (including completed and inactive tasks), with links from each task to a page where the task can be viewed in more detail, and where operations can be performed on the task. The task's identifying information (its process name, segment name, subject ID, and task element ID) can be extracted from the TaskInfo and passed to this page as parameters. These parameters can then be used as inputs to the WorkflowTaskFormHandler, which can display task information and perform operations on the task, including firing task outcomes. See the Workflow Task Form Handler section for more on WorkflowTaskFormHandler.

WorkflowInstanceQueryDroplet

In some cases, rather than viewing all workflow tasks, you may to want to view all existing workflow instances. The atg.workflow.servlet.WorkflowInstanceQueryDroplet servlet bean class provides this capability. For example, suppose you have a workflow type for managing orders, and you want view all the outstanding orders and their status. Suppose, also, that an order workflow involves several simultaneously active tasks. A query made using the WorkflowTaskQueryDroplet might return multiple active tasks for each order, but the WorkflowInstanceQueryDroplet will return a single entry for each workflow instance. (The registration workflow does not use an instance of this servlet bean, because a registration workflow can have only one active task at a time.)

ItemLookupDroplet

The WorkflowSubjectLookup component in the registration workflow is a servlet bean of class atg.repository.servlet.ItemLookupDroplet that is configured to work with the workflow's repository and item type. This servlet bean is useful for looking up properties of the workflow subject given the subject ID. In the following example, the taskInfo parameter points to a TaskInfo object returned by a WorkflowTaskQueryDroplet:

```
<dsp:droplet name="WorkflowSubjectLookup">
  <dsp:param name="id" param="taskInfo.subjectId"/>
  <dsp:oparam name="output">
    <dsp:valueof param="element.employeeFirstName"/>
    <dsp:valueof param="element.employeeLastName"/>
  </dsp:oparam>
</dsp:droplet>
```

GetDirectoryPrincipal

The Personalization module includes a servlet bean of class atg.userprofiling.GetDirectoryPrincipal, located at /atg/dynamo/droplet/GetDirectoryPrincipal, which you can use for resolving user directory principals. Although it is not specific to workflows, the GetDirectoryPrincipal servlet bean has some workflow-related uses:

- You can use its principal output parameter as an input to the WorkflowTaskQueryDroplet. For example, you could use these two servlet beans together to find all active tasks that can be executed, claimed, and released by the Approver role.
- You can use its persona output parameter to specify the newOwnerName property of the WorkflowTaskFormHandler, which is described in detail in the next section. The name to pass to the form handler can be accessed as the value of persona.name in the servlet bean.

Workflow Task Form Handler

In addition to workflow servlet beans, the configuration for each new workflow type typically includes a request-scoped component of class atg.workflow.servlet.WorkflowTaskFormHandler. This form handler is used to perform workflow task operations such as setting a task's owner and firing outcomes.

Read-only properties

The WorkflowTaskFormHandler class has four properties that together uniquely identify the task being operated on by the form handler: processName, segmentName, subjectId, and taskElementId. It also has the following read-only properties that can be accessed once the task's identifying information has been specified:

Property	Description
taskDescriptor	TaskDescriptor object that describes the task
outcomeDescriptors	list of OutcomeDescriptor objects that describe all the possible outcomes of the task
taskInfo	TaskInfo object that describes the given task instance

Submit methods

The form handler's submit methods operate on the task on behalf of the WorkflowView associated with the form. The following submit methods are supported:

Submit method	Description
setTaskPriority	Sets the task's priority to the value of the newPriority property
setTaskOwner	Sets the task's owner to the atg.userdirectory.DirectoryPrincipal corresponding to the atg.security.Persona name specified via the newOwnerName property
claimTask	Claims the task on behalf of the current user's view
releaseTask	Releases the task on behalf of the current user's view
fireOutcome	Fires the task outcome identified by the outcomeElementId property

The following properties of the WorkflowTaskFormHandler are typically set in the properties file for this component:

Property	Description
workflowManager	WorkflowManager that provides access to the workflow system
workflowView	Session-scoped WorkflowView on behalf of which the task is to be manipulated
subjectRepository	MutableRepository that stores workflow subjects
subjectType	Workflow subject item type
subjectClassName	Name of the class to use when instantiating the object to be returned by the subject property; the default is atg.workflow.servlet.WorkflowTaskFormSubject (see below for details)

The following properties are typically set in the JSP:

Property	Description
`processName`	Name of the workflow process
`segmentName`	Name of the workflow process segment
`subjectId`	Repository ID of the workflow subject
`taskElementId`	ID of the workflow element of the task
`newPriority`	New priority value to be set by the `setTaskPriority` submit method
`newOwnerName`	Unique `atg.security.Persona` name of new owner, to be set by the `setTaskOwner` submit method
`outcomeElementId`	ID of the outcome element to be fired by the `fireOutcome` submit method
`updateSubjectOnSubmit`	flag indicating whether the workflow subject should be updated when the submit methods are executed (see below for details)

Navigation properties

The `WorkflowTaskFormHandler` also has a set of properties that are used to control navigation after a form operation has been completed. These properties specify the URLs to redirect to on certain error and success conditions. If the value for a particular condition is not set, the form is left on the page defined as the action for that form (in other words, no redirect takes place). Each operation has its own `successURL` and `errorURL` properties. Thus the following properties are available:

- `setTaskPrioritySuccessURL`
- `setTaskPriorityErrorURL`
- `setTaskOwnerSuccessURL`
- `setTaskOwnerErrorURL`
- `claimTaskSuccessURL`
- `claimTaskErrorURL`
- `releaseTaskSuccessURL`
- `releaseTaskErrorURL`
- `fireOutcomeSuccessURL`
- `fireOutcomeErrorURL`

These properties can be set in the properties file of the form handler or by hidden tags in the JSP.

Updating Subject Properties

When operating on tasks, it is common to want to update the workflow subject as it gets advanced through the workflow. For example, a manager in an expense report workflow may be required to submit a comment as part of approving or rejecting an expense report. The comment field is typically a property of the workflow subject (the expense report).

To facilitate subject updates without the need for custom code, `WorkflowTaskFormHandler` exposes a special `subject` property of class `atg.workflow.servlet.WorkflowTaskFormSubject` (or a subclass specified via the `subjectClassName` property). `WorkflowTaskFormSubject` is a subclass of `atg.repository.RepositoryFormHandler` and, like its parent class, exposes the subject's properties via a `value` dictionary. Using the `subject` property, you can set workflow subject values as follows:

```
<dsp:textarea bean="WorkflowTaskFormHandler.subject.value.comment"
              cols="35" rows="5" default=""></dsp:textarea>
```

When the `updateSubjectOnSubmit` property is set to true, the values set via the `subject` property are copied over to the workflow subject when any of the submit methods are executed.

For maximum flexibility, the `WorkflowTaskFormHandler` class can also be overridden to perform operations before and after the work done by the submit methods. Each submit method has a pair of corresponding `pre` and `post` operations. Thus the following protected methods can be overridden:

- `preSetTaskPriority`
- `postSetTaskPriority`
- `preSetTaskOwner`
- `postSetTaskOwner`
- `preClaimTask`
- `postClaimTask`
- `preReleaseTask`
- `postReleaseTask`
- `preFireOutcome`
- `postFireOutcome`

In the default implementation, each `pre` method checks to see if the `updateSubjectOnSubmit` property is set to `true`, and if so, updates the subject by calling `handleUpdate` on the `WorkflowTaskFormSubject` object. The `post` methods do nothing by default.

Firing Task Outcomes

Most workflow tasks have several possible outcomes. Typically, a single form with multiple submit buttons is used to fire one of those outcomes. In these situations, the `outcomeElementId` property cannot be set via a hidden tag, since each submit button needs to specify its own outcome element ID.

Instead, the outcome element ID must be passed in as the submit value. For example, here is how you might display multiple submit buttons using a `ForEach` servlet bean:

```
<dsp:droplet name="/atg/dynamo/droplet/ForEach">
  <dsp:param name="array" bean="WorkflowTaskFormHandler.outcomeDescriptors"/>
  <dsp:oparam name="output">
    <dsp:getvalueof id="locale" bean="/OriginatingRequest.requestLocale.locale"
                    idtype="java.util.Locale">
      <dsp:getvalueof id="outcomeDescriptor" param="element"
                      idtype="atg.workflow.OutcomeDescriptor">
        <dsp:input type="submit" bean="WorkflowTaskFormHandler.fireOutcome"
                   value="<%= outcomeDescriptor.getDisplayName(locale) %>"
                   submitvalue="<%= outcomeDescriptor.getOutcomeElementId() %>"
                   name="<%= outcomeDescriptor.getName() %>"/>
      </dsp:getvalueof>
    </dsp:getvalueof>
  </dsp:oparam>
</dsp:droplet>
```

To support this functionality, the `WorkflowTaskFormHandler` class defines a `setFireOutcome` method, which is called immediately before `handleFireOutcome` when the form is submitted. This method simply sets the `outcomeElementId` property using the submitted value.

26 Managing Workflows on Multiple Servers

This chapter describes the configuration tasks necessary for using workflows on multiple ATG 10.0.2 servers. It contains the following sections:

Designating a Process Editor Server for Workflows

If your environment includes more than one ATG 10.0.2 server running workflows, you must take steps to ensure that the servers do not conflict with each other when they attempt to handle the sending and receiving of workflow messages. The configuration process is the same as the process for scenarios: you designate one server as the process editor server and then assign either global or individual workflow editor status to the remaining servers.

Specify the identity of the process editor server in the Workflow Process Manager configuration file, `workflowProcessManager.xml`. Create a file with the same name in the top-level `<ATG10dir>/home/localconfig/atg/epub/workflow/process` directory (in other words, in the directory that applies to the whole cluster). The following shows an example of the process editor server setting you would add to this file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<process-manager-configuration>

<process-editor-server>
  <server-name>dyn1:88500</server-name>
</process-editor-server>

</process-manager-configuration>
```

For detailed information, refer to Configuring the Scenario Manager.

Note that the same server can act as the process editor server for both workflows and scenarios. If you do elect to use the same server for both purposes, specify the same <server-name> setting for the process editor server in the workflowProcessManager.xml file and the scenarioManager.xml file.

Designating Global and Individual Workflow Servers

After you have designated one server as your process editor server for workflows (see the previous section), you can then determine whether the remaining servers should act as global or individual servers. You define a server as global or individual in the top-level localconfig copy of the workflowProcessManager.xml file (see the previous section).

For detailed information on this configuration process, refer to Configuring the Scenario Manager.

Configuring Caching for Workflows

If you are running workflows on multiple servers, you must set the cache-mode property to locked for the following workflow item descriptors on each server:

- individualWorkflow
- collectiveWorkflow
- workflowInfo
- workflowTemplateInfo
- workflowMigrationInfo
- individualWorkflowTransition
- collectiveWorkflowTransition

These repository item descriptors are contained in one of the custom repositories that you set up as part of creating a workflow type. To set the cache mode, add the following settings to the XML file that you use to define the repository that contains these items:

```
<gsa-template>
  <item-descriptor name="individualWorkflow" cache-mode="locked"/>
  <item-descriptor name="collectiveWorkflow" cache-mode="locked"/>
  <item-descriptor name="workflowInfo" cache-mode="locked"/>
  <item-descriptor name="workflowTemplateInfo" cache-mode="locked"/>
  <item-descriptor name="workflowMigrationInfo" cache-mode="locked"/>
  <item-descriptor name="individualWorkflowTransition" cache-mode="locked"/>
  <item-descriptor name="collectiveWorkflowTransition" cache-mode="locked"/>
</gsa-template>
```

In addition, in order for locked mode caching to work in a multiple server environment, you must configure Lock Manager components as described in the *Enabling the Repository Cache Lock Managers* section of the *ATG Installation and Configuration Guide*.

27 Setting Up Security Access for Workflows

For workflows, you can define the following levels of security access:

- Access to the Workflow menu items in the ACC. This setting determines whether users can create, edit, or view workflows in the workflow editor. See Allowing ACC Users to Edit Workflows.
- The ability to execute a specific workflow, which determines whether a user can initiating the project or process that contains the workflow. See Allowing Site Users to Execute Workflows.
- The ability to complete specific tasks within a workflow, which determines whether a user has access to buttons or other controls that would indicate a given task has been initiated and thus advance the workflow to the next element. See Giving Site Users Access to Workflow Tasks.

The mechanism that workflows use to handle security is the access control list, described in *Secured Repositories* in the *ATG Repository Guide*. For workflows, access control entries are stored as strings in the workflow definition XML file and then parsed by the `atg.security package` into an `AccessControlList` object.

The following example shows an access control entry from the `registration.wdl` file, which is the definition file for the EcoVida registration workflow:

```
<attribute name="atg.workflow.acl">
  <constant>Profile$role$2900189:execute</constant>
</attribute>
```

Allowing ACC Users to Edit Workflows

ACC users create, edit, and view workflows through the workflow editor. You can grant or deny access to the workflow editor for any workflow type in your system by enabling or disabling the menu item that corresponds to that workflow type. For example, to prevent users from editing or creating more instances of the Registration workflow type in the EcoVida demo, you would simply disable the Workflow > Registration item on the main ACC navigation menu.

As you do for any other ACC menu item, you enable or disable menu items for specific groups of ACC users; the given items do not appear if users do not belong to a group that has explicit access to them. To enable or disable the Workflow menu item that corresponds to a particular workflow type, complete the following steps:

1. In the ACC, select People and Organizations > Control Center Groups.
2. Select the group of users whose access rights you want to change.
3. Edit the checkbox that corresponds to the Workflow menu item for which you want to set access. For example, to deny access to the Registration workflow type in the EcoVida demo, deselect `Workflow:Registration`.

For more information, see *Configuring Access Privileges* in the *ATG Programming Guide*.

Allowing Site Users to Execute Workflows

To “execute” a workflow means to complete the steps that cause the first element in a given workflow to occur. For example, in ATG Content Administration, users execute a workflow by creating a project to which that workflow is assigned. When a user creates a project, ATG Content Administration checks to make sure the user has the proper access rights to execute the workflow associated with the project type. If the user does not have the proper privileges, the project is not created.

You assign workflow execution rights by user directory principal – in other words, you can assign this right to organizations, roles, profile groups, or individuals. For more information on user directory principals, see Working with the Dynamo User Directory.

Follow these steps to assign workflow execution rights:

1. Open the workflow.
2. Right-click the first element in the workflow and select Edit Details from the pop-up menu. The Workflow Attributes window opens.
3. Click the Set Access Rights button. The Set Access Rights window opens.
4. Uncheck the Use Default Security Policy Behavior checkbox, which indicates that no user access list has been specified and therefore that all users can execute the workflow.
5. Click on Add to add a new principal to the access rights list. The Add Principal box opens.
6. Select the type of principal you want to add:

Principal	Description
Organizations	Gives workflow execution rights to all members of the specified organization. Note that a child organization automatically inherits the access rights of the parent organization. You can override the inherited rights by explicitly setting the access rights for the child organization. For more information about working with organizations, see the *Setting Up Visitor Profiles* chapter of *ATG Personalization Guide for Business Users*.
Roles	Gives workflow execution rights to anyone assigned the specified role.
Groups	Gives workflow execution rights to all members of the specified profile group.
Individuals	Gives workflow execution rights to the specified individuals (user profiles). You can list individual users according to the organizations to which they belong (List Individuals by Organization), or you can search for users that match certain criteria (List by Expression).

7. Grant access rights to the principal you chose by checking the Execute access right in the list. To deny access rights, by clicking again so there is an X in the box. An empty box indicates that no user access rights are set.
8. Click OK.
9. Select File > Save to save your changes to the workflow.

Giving Site Users Access to Workflow Tasks

For individual tasks within a workflow, you can give site users the following types of access privileges:

- You can allow a user to execute a task. For example, assume you have set up a site page that lets all your technical support representatives see a list of pending customer cases that require technical involvement. You can include a Claim button on the page that allows a site user (a support representative, in this case) to assign herself a given case. The Claim button is displayed only to those users who have Execute access rights to the task element that is triggered when the button is clicked. In addition, you can set up a Release button that allows users to return a task they previously claimed to the list of available tasks.

  To allow a user to execute a task (which includes the ability to claim and release it), give him or her Execute access rights as described below.

- You can allow users to assign a task to other people. This process is the same as allowing a user to claim a task, but it enables a user such as a manager to nominate another user for a given task. Users who have Write access can assign tasks to other users.

  To implement this feature, you must perform the following steps:

- Enable the task's Assignable option as described below.
- Give Write access to the appropriate users as described below.
- Add an Assign button (or its equivalent) to an appropriate site page. When users with Write access for this task click the Assign button, a list of users appears to whom the task can be assigned. This list includes the principals that have been granted Execution access rights to the task. For example, if you grant Execution privileges to a profile group, all members of this group appear in the list. The page developer should configure the Assign control to call `WorkflowView.setTaskOwner(processName, segmentName, subject id, tasked, user)` where `user` is an instance of `atg.userdirectory.User`.

To give a user Execute or Write access for a task:

1. Display the appropriate workflow in the ACC.
2. Right-click the task whose security access you want to change.
3. Follow the procedure described in the previous section, Allowing Site Users to Execute Workflows.

To make a task assignable:

1. Display the appropriate workflow in the ACC.
2. Right-click the task whose access you want to change.
3. Click Edit Details.
4. Check the Assignable option.
5. Close the task element.
6. Select File > Save to save your changes to the workflow.

28 Configuring the ATG Expression Editor

The ATG expression editor is a collection of code resources that together define the parts of the ACC interface where users can manipulate grammar expressions within areas such as targeting rules and scenarios. This section describes how to define or modify the grammar that drives the editor. It includes the following sections:

Overview of the ATG Expression Editor

Each instance of the expression editor in the ATG user interface is configured at runtime by an XCL template, which exposes the grammar as a graph of Java objects implementing the `atg.ui.expreditor.Construct` interface. In this graph, each Construct represents one of the following fundamental types of grammatical constructs:

- **Token**: a unitary word or phrase within the expression such as `People whose`, `named`, or `is not`. The content of a token cannot be edited by the user directly, but tokens can be selected from a list or from a dialog box. A token may have a visible representation different from its intrinsic value, which is always of String type.
- **Literal**: a constant value that the user can supply and freely edit within an expression, such as a number or a string. In the expression `People whose name is Nate`, the term `Nate` would be a literal supplied by the user.

- **Choice**: a set of alternative constructs, exactly one of which is allowed to occupy a particular place in the expression. The user is permitted to select the member of the choice set. For example, in the alternative expressions `People whose name is Nate` and `People whose name is not Nate`, the tokens `is` and `is not` are members of a choice.
- **Sequence**: a sequence of a fixed number of constructs that occur in a fixed order. For example, the expression `People whose name is Nate` might be defined as a sequence whose components are the token `People whose`, a choice of properties (of which one is `name`), a choice of {`is, is not`} and a literal.

As XCL templates, expression grammars are defined by XML data files in the CLASSPATH, which in turn can be compiled into .ser files by the XCL toolkit for greater efficiency and speed. The XCL stylesheet used by expression grammar templates provides XML elements that can be used to define grammatical concepts, including the basic construct types given above.

For example, the following XML file would serve to define the sequence mentioned above. It includes a number of details that will be explained later in this section.

```
<?xcl-stylesheet resource="atg/ui/expreditor/xcl/grammar.xsl"?>
<sequence id="example-sequence">
  <token><description>People whose</description></token>
  <choice>
    <token value="name"/>
    <token value="email"/>
    <token value="gender"/>
  </choice>
  <choice>
    <token value="eq"><description>is</description></token>
    <token value="neq"><description>is not</description></token>
  </choice>
  <literal/>
</sequence>
```

Grammar Template Files

This section explains the basics of grammar template files in preparation for exploring the details of grammar definition.

Templates, Filenames and Localization

An XCL expression grammar template is referenced within the code like a `ResourceBundle`: the code specifies the template as if it were a class name, and XCL searches in the CLASSPATH for the corresponding .ser or .xml file, looking for any localized variants that are present.

XCL looks for .ser files before .xml files, and it also performs a most-specific-locale-first search. For example, in the US English locale, if the code is looking for a template named `atg.ui.expreditor.targeting.query-grammar`, the following files will be searched for in this order:

1. `atg/ui/expreditor/targeting/query-grammar_en_US.ser`
2. `atg/ui/expreditor/targeting/query-grammar_en_US.xml`
3. `atg/ui/expreditor/targeting/query-grammar_en.ser`
4. `atg/ui/expreditor/targeting/query-grammar_en.xml`
5. `atg/ui/expreditor/targeting/query-grammar.ser`
6. `atg/ui/expreditor/targeting/query-grammar.xml`

Unlike for a `ResourceBundle`, here there is no attempt to combine files. The first file found by the above search logic will be used in its entirety, and any other matching files will be ignored.

Stylesheet Preamble

Each expression grammar must begin with the following XML processing instructions, which define the textual encoding used by the file, and the XCL stylesheet to be used, defining such XML elements as sequence, choice, and so on:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?xcl-stylesheet resource="atg/ui/expreditor/xcl/grammar.xsl"?>
```

The encoding attribute of the XML instruction may take on any of the values shown in Supported Character Encodings, permitting grammar definitions to be expressed in any desired textual encoding.

Textual Inclusion

A grammar definition file may textually include another grammar definition by use of the XCL `include` element. For example, the following line includes the contents of the `dynamo-grammar` template definition into the file in which it occurs:

```
<include template="atg.ui.expreditor.dynamo.dynamo-grammar"/>
```

Grammar Templates in the ATG Product Distribution

The following grammar templates are distributed with ATG products:

Distribution file	Purpose	Template Name

`DPS/lib/classes.jar`	Query expressions in Content Repository Admin UI	`atg.ui.expreditor.targeting`
		`.`
		`query-grammar`

`DPS/lib/classes.jar`	Common definition of date-related subexpressions	`atg.ui.expreditor.dynamo.` `date-grammar`

Distribution file	Purpose	Template Name
`DPS/lib/classes.jar`	Common definition of Dynamo-related subexpressions	`atg.ui.expreditor.dynamo.dynamo-grammar`
`DSS/lib/classes.jar`	Scenario expressions supported by the Scenarios module	`atg.ui.scenario.expression.expression-grammar`
`DCS/lib/classes.jar`	PMDL expression grammar for ATG Commerce promotions editor	`atg.ui.commerce.pricing.pricing-grammar`
`DCS/lib/classes.jar`	Scenario expression extensions for ATG Commerce	`atg.ui.commerce.scenario.commerce-expression-grammar`

Serialization of Templates

Templates can be serialized to .ser files to avoid the runtime overhead of parsing. To serialize a template, use the following command:

```
java atg.ui.xuill.Xuill -serialize template-name
```

The input-file is the pathname of the template file (with a .xml suffix). A corresponding .ser file will be created in the current directory.

To use this command, the ATG distribution's `DAS/lib/classes.jar` and `DAS-UI/lib/uiclasses.jar` files must be in your CLASSPATH.

Grammar Definition Fundamentals

This section describes how to use the four fundamental types of grammatical constructs outlined above:

- Tokens
- Literals
- Choices
- Sequences

Tokens

Tokens are non-editable words or phrases within an expression, specified by the `token` element in XML. A token has two main properties: its `value` and its `description`. The relationship between these properties is similar to the relationship between a property's `name` and `display name`.

The `value` of a token is a String representing the token's identity from the application's point of view. It is specified by an optional `value="..."` attribute of `token`. A value need not be supplied in the case of noise words that merely pad out the expression or provide a linguistic handle for a choice. A value should not be localized.

The description of a token is the text used to display the token in the expression editor user interface, both in the body of the actual expression and in choice lists. The description is specified by the text contained in a <description>...</description> child element of token. In general it should be localized. If it is not provided, the token's value will be used as the description.

The editor text of a token is optional descriptive text that will be displayed in the expression body, but not in choice lists. It is specified by the text contained in a <editor-text>...</editor-text> child element.

A <hidden/> child element, if present, denotes that the token is to be displayed only in choice lists and hidden in the expression body. It is used for choice elements that affect the structure of subsequent parts of the expression, but which would visually disrupt the expression if visible. It is often used for the first element of a sequence that itself is one alternative of a choice.

Some examples:

```
<token value="1"/>
```

This example shows a token whose value and description are equal to the string "1"

```
<token value="US">
  <description>United States</description>
</token>
```

This example shows a token whose value is the ISO country code "US", and whose description in the user interface is "United States". In a French locale, the description might be altered to "États-Unis".

Literals

Literals are values that may be edited through the expression editor user interface. They are specified by the literal XML element.

A literal has a default value, which can be specified by <default><value type="class">...</value></default>. The type of this default value determines the data type that the user is permitted to enter. If no default is given, the default is the empty string.

A literal's description is identical in nature and specification to the description of a token: it determines how the literal is presented in choice lists, and it also acts as a descriptive prompt whenever the literal is selected for editing.

The following example shows a literal of type Integer whose default value is the number 1.

```
<literal>
  <default><value type="java.lang.Integer">1</value></default>
  <description>Enter a relative priority</description>
</literal>
```

Choices

Choices are specified by the `choice` XML element, whose children may be any set of construct elements.

A `choice` always has a default child element that is initially selected. If not specified, this is the first element, but it can be overridden by enclosing any desired child in a `<default-choice>...</default-choice>` wrapper.

Choices must have at least one child.

Sequences

Sequences are specified by the `sequence` XML element, which can have any set of constructs as its children.

A sequence must have at least one child.

Creating a Grammar by Composing Constructs

The basic technique of defining an expression editor grammar is to define individual tokens and literals that represent building blocks of the desired expression language, and compose them into larger structures made up of sequence and choice elements. These structures can in turn be composed into still more complex sequences and choices, and so on.

Because tokens and literals cannot be decomposed into smaller pieces, they are called terminal constructs (as in the terminal branches of a tree). Sequences and choices are nonterminals.

An expression grammar usually has a single top-level construct whose contained sequences and choices represent all possible expressions in the grammar.

Let’s return to our earlier example grammar:

```
<?xcl-stylesheet resource="atg/ui/expreditor/xcl/grammar.xsl"?>
<sequence id="example-sequence">
  <token><description>People whose</description>
  <choice>
    <token value="name"/>
    <token value="email"/>
    <token value="gender"/>
  </choice>
  <choice>
    <token value="eq"><description>is</description></token>
    <token value="neq"><description>is not</description></token>
  </choice>
  <literal/>
</sequence>
```

This grammar's top level construct is given the name `example-sequence`, and it permits the following expressions:

```
People whose name is ____

People whose email is ____

People whose gender is ____

People whose name is not ____

People whose email is not ____

People whose gender is not ____
```

The expression editor always displays a valid expression at all times; consequently, some element of a choice must always be "chosen". There is an initial default for each choice, which is normally the first element of the choice, but this may be overridden. Thus, in our example, the first of the above possibilities will be shown when the editor is initially displayed.

Structure and Presentation of Choices

In the expression editor user interface, a set of choices is represented by a list that has one entry for each of the leading terminals of the choice. Whenever the user selects the leading terminal of some element of a choice, the user interface will display a list of all other leading terminals that can occupy that particular place in the expression. Each entry in the list shows the corresponding terminal's description text, as defined in the grammar.

In our example grammar given above, the leading terminals for a person's property are `name`, `email` and `gender`. Consequently, when the user selects `name`, all three of these alternatives will be displayed.

This concept seems trivial, but the set of choices can be determined in complex ways since one can have choices of sequences, or choices of other choices. These are handled as follows:

- The leading terminals of a sequence are the leading terminals of the first element of the sequence.
- The leading terminals of a choice are the union of the sets of leading terminals of each element of the choice.
- Whenever the leading terminal of a sequence is selected, this automatically causes all the subsequent elements of the sequence to be included in the expression.

To demonstrate these principles, let's modify our example:

```
<?xcl-stylesheet resource="atg/ui/expreditor/xcl/grammar.xsl"?>
<sequence id="example-sequence">
  <token><description>People</description></token>
  <choice id="filter-choice">
    <sequence id="property-filter">
      <token><description>whose</description></token>
      <choice>
        <token value="name"/>
        <token value="email"/>
        <token value="gender"/>
      </choice>
      <choice>
        <token value="eq"><description>is</description></token>
        <token value="neq"><description>is not</description></token>
      </choice>
      <literal/>
    </sequence>
    <choice id="location-or-food-choice">
      <sequence id="location-filter">
        <token><description>living in</description></token>
        <choice>
          <token value="US"><description>United States</description></token>
          <token value="JP"><description>Japan</description></token>
      </sequence>
      <sequence id="food-filter">
        <token><description>who eat</description></token>
        <choice>
          <token value="meat"/>
          <token value="vegetables"/>
      </sequence>
    </choice>
  </choice>
</sequence>
```

The first element of an expression in this grammar is, of course, People. The second element can be chosen from any of the following:

- • `whose` (the leading terminal of `property-filter`)
- • `living in` (the leading terminal of `location-filter`)
- • `who eat` (the leading terminal of `food-filter`)

It is clear enough that `whose` is a valid choice: after `People` comes a construct named `filter-choice`, the first of whose choices is a sequence that begins with the leading terminal `whose`. The selection of `whose` automatically includes the other elements of `property-filter` in the expression—in other words, picking `whose` includes the entire sequence `whose name is` _____ in the edited expression.

But note that `living in` and `who eat` are presented as alternatives to `whose`, even though `location-filter` and `food-filter` are not direct alternatives to `property-filter` within the same choice. This occurs because they are elements of `location-or-food-choice`, which is an alternative to `property-filter`. The set of alternatives to `property-filter` must therefore expand to include all the leading terminals of `location-or-food-choice`.

If the alternative `who eat` is selected, it alters two different choices in the grammar: the current element of `filter-choice` becomes `location-or-food-choice`, and the current element of `location-or-food-choice` becomes `food-filter`.

Defining and Referring to Labeled Constructs

It is frequently the case that a particular construct will be a child of more than one parent construct. For instance, consider our example grammar—it may be the case that a choice of people properties is needed in multiple expressions within the same grammar.

The `id="..."` attribute may be attached to any construct in order to label the construct with a unique ID. The presence or absence of an ID does not affect the nature of the labeled construct.

Once labeled, a construct can be referred to within any nonterminal construct (choice or sequence) using the notation `<rule name="..."/>`. To rework our initial example so that it compares two different people properties, we might do the following:

```
<?xcl-stylesheet resource="atg/ui/expreditor/xcl/grammar.xsl"?>
  <sequence id="example-sequence">
    <token><description>People whose</description></token>
    <rule name="people-property-choice"/>
    <choice>
      <token value="eq"><description>is</description></token>
      <token value="neq"><description>is not</description></token>
    </choice>
    <rule name="people-property-choice"/>
  </sequence>
  <choice id="people-property-choice">
    <token value="name"/>
    <token value="email"/>
    <token value="gender"/>
  </choice>
```

A labeled construct need not be defined in a “standalone” manner as in the above example; it can be directly included in one parent construct as a regular child, but also referenced by other parent constructs by the `<rule name=".../>` notation.

Circular reference chains between constructs are legal, and are in fact an important tool for grammars, since the resulting recursion permits the user to form arbitrarily complex expressions. However, there are two cases to avoid, since they would cause the user interface to enter infinite loops of one kind or another:

- The leading element of a sequence must never refer to its parent, directly or indirectly.
- The default element of a choice must never refer to its parent, directly or indirectly.

Important: Many constructs are labeled so that the application code itself can identify them. In general, construct IDs should be assumed to be significant to the application and should not be changed.

Advanced Features

This section describes some additional features you can use to define an expression editor grammar.

Custom Expression Classes

You can label any construct with a `<expression-class>...</expression-class>` child to specify a Java class for the construct's runtime expression. The class must implement `atg.ui.expreditor.Expression`. This feature is typically used to impose additional behaviors and validity constraints for special constructs, such as choices that are automatically populated with bean `property-name` tokens as their children.

Custom Editor Classes

You can label any literal with a `<editor-class>...</editor-class>` child to specify an actual AWT component used to display the literal within the expression editor, implementing `atg.ui.expreditor.TerminalExpressionEditor`. This feature permits special-purpose editing of literal values.

Custom Assistant Classes

You can label any terminal with an `<assistant-class>...</assistant-class>` child to specify a Java class for the literal's runtime assistant, implementing `atg.ui.expreditor.TerminalAssistant`. This feature permits special-purpose editing of literals or tokens via custom widgets that are displayed in a popup window above or below the selected terminal.

Placeholders

Some applications may use the expression editor in a special mode in which placeholders may be defined. A placeholder is a part of some expression that can be given a name by the user at runtime. Placeholders are used to support user-defined "template" expressions in which named portions of the template can be later changed.

In a grammar definition file, you can enclose any construct in a `<placeholder>...</placeholder>` wrapper to indicate that it should be treated as a placeholder.

Required Terminals

A terminal construct may be marked as required by including the child XML expression `<required/>`. This forces the expression editor to require the expression to have a non-null, non-empty value.

Eliminating Spaces

Normally a space is displayed immediately after each terminal. To suppress this space, include the child XML expression <nospace/>.

Verbose Terminals

Some verbose terminals should not be shown except when an expression is being actively edited. The child XML expression <verbose/> is used to denote such terminals.

Unsigned Integer

A literal element may possess the child XML expression <unsigned-integer-editor> to indicate that no sign should be permitted.

Scenario UI Expression Grammar Configuration

A grammar extension to the Scenarios module has special considerations. It specifies an XML file containing an expression grammar definition just like those described above, with information specific to this module, including:

- Grammar elements that correspond to custom scenario actions, events and conditions
- Additional information governing the generation of XML fragments that provide the PDL (Process Description Language) corresponding to grammar elements
- Special-purpose tags for scenario-related grammatical expressions and attributes

Note that the chapter Adding Custom Events, Actions, and Conditions to Scenarios contains a detailed example of extending the grammar editor for scenarios. It shows how to add a custom condition to the grammar editor. The example is in the section Extending the Expression Editor.

Scenario Grammar Extension Header

All grammar extension definitions must include an additional processing instruction at their top. The header of a grammar definition file for the Scenarios module looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?xcl-stylesheet resource="atg/ui/scenario/expression/
 scenario-grammar.xsl"?>
<?xcl-stylesheet resource="atg/ui/expreditor/xcl/grammar.xsl"?>
```

Defining Expressions for Custom Events

The grammar element for a custom event must have a special id: the prefix event-message- followed by the JMS message type string for the event. The <element-icon> and <declared-message-type> tags must also be used to provide an icon (specified as a class loader resource path) and the message type that will be associated with the event. The icon is optional; a generic icon will be used if none is specified.

For instance, if you were defining an expression for the custom event `MyProject.MyEvent`, you would define the event in the grammar as follows:

```
<sequence id="event-message-MyProject.MyEvent">
  <element-icon>icon-resource-path</element-icon>
  <declared-message-type>MyProject.MyEvent</declared-message-type>
  ...
</sequence>
```

Defining Expressions for Custom Actions

The grammar element for a custom action must have a special `id`: the prefix `action-` followed by the action's name, as specified in `scenarioManager.xml`. The optional `<element-icon>` tag may be used to provide an icon (specified as a class loader resource path). For instance, if you were defining an expression for the custom event `MyAction`, you would define the event in the grammar as follows:

```
<sequence id="action-MyAction">
  <element-icon>icon-resource-path</element-icon>
  ...
</sequence>
```

Defining Expressions for Custom Conditions

The grammar element for a custom condition must have a special `id`: the prefix `condition-` followed by the condition's name as specified in `scenarioManager.xml`. The optional `<element-icon>` tag may be used to provide an icon (specified as a class loader resource path). For instance, if you were defining an expression for the custom event `MyCondition`, you would define the event in the grammar as follows:

```
<sequence id="condition-MyCondition">
  <element-icon>icon-resource-path</element-icon>
  ...
</sequence>
```

Associating XML Templates with Grammar Elements

If you simply defined the expression grammar for custom elements, and nothing more, the system would have no way of knowing what PDL (Process Description Language) is associated with the grammar elements, and the extension would not work. Special XML template information must be associated with grammar elements that correspond to PDL fragments. This information allows the editor to do the following:

1. Parse an existing scenario element's PDL and configure an expression appropriately to reflect its contents. The DOM for a scenario element is traversed and recursively matched against the fragments in the expression's XML templates.
2. Generate the appropriate PDL for a scenario element given the choices and literal values specified in the expression editor. The expression is traversed, and the XML

template information drives the recursive construction of a DOM for the scenario element.

In general, an XML template for a grammar element consists of one or more literal fragments of XML, in which an "attachment point" may be defined for the application of XML templates belonging to child elements.

For parsing, a grammar element's template is matched against the DOM element being parsed, to determine whether the element is applicable. If the match succeeds, the attachment point determines a descendant of the matched DOM element that will be recursively matched against the grammar element's children. For generation, the template's XML is generated whenever the grammar element is encountered in a traversal of the expression. The attachment point determines the place within the generated DOM within where the grammar element's children will attach their own generated XML. If a grammar element has no XML template, the current DOM element being parsed or generated is simply passed to the element's children.

The XML templates of a sequence's children are processed in the order in which the children are defined. All templates of all the children must match, or the sequence is not considered to match as a whole.

The XML templates of a choice's children are treated differently. On XML parsing, the choice's currently chosen child is determined by finding the first child whose XML template matches the DOM fragment being parsed. On XML generation, only the currently chosen child's XML templates will be processed.

The XML template for atomic constructs like literals and tokens is used to define only their associated non-varying XML, if any. For parsing, the textual child of the current DOM fragment is taken as the value of the construct, and the type attribute is taken as its type. On generation, the value of the literal or token is emitted as a textual child of the construct and the type of the value is emitted as an attribute named type.

Specification of XML templates

XML templates are specified for a grammar element by placing one of the following tags within it:

Tag	Purpose
`<xml-template>`	Encloses an XML fragment that is associated with the grammar element. If the grammar element has children with their own XML templates, then an embedded `<apply-xml-templates/>` element determines the location in the fragment at which the children's associated XML will be spliced in.
`<xml-empty-template/>`	Specifies that the grammar element is associated with the absence of any parsed or emitted XML. This is only meaningful when other alternative constructs in a choice have non-empty XML templates.

Tag	Purpose
`<xml-template-attribute name="..."/>`	Specifies that the grammar element is associated with an XML attribute.

Here is an example to clarify the use of XML templates. Suppose you have a custom scenario action whose expression grammar takes the following form:

```
Perform action in ( this way | that way ) on [literal]
```

This action will be represented as a sequence beginning with the token `Perform MyAction in`, followed by a choice of tokens `this way` and `that way`, followed by the on token, and finally a literal for the user to supply a value. In the case where the token `this way` is chosen, the expression will correspond to the following PDL fragment (all of which would be automatically enclosed in an <`action`> element by the scenario editor):

```
<action-name>MyAction</action-name>
<action-param name="mode">
  <constant>this</constant>
</action-param>
<action-param name="value">
  <constant type="java.lang.Integer">value</constant>
</action-param>
```

(In the case where `that way` is chosen, the PDL would be identical but the value of the mode argument would be `that`.)

The expression grammar for such a custom action could look like this:

```
<sequence id="action-MyAction">
  <!-- XML template specifying the action name.  Child elements' XML will
       be processed following the <action-name> element. -->
  <xml-template>
    <action-name>myAction</action-name>
    <apply-xml-templates/>
  </xml-template>
  <!-- "noise" token that simply acts as a choice handle in the UI;
       it is not associated with any PDL. -->
  <token><description>Perform action</description></token>
  <!-- further "noise" token providing connective to choice.  We
       separate it because "Perform action" is nicer in a dropdown
       list than "Perform action in". -->
  <token><description>in</description></token>
  <!-- choice of mode arguments -->
  <choice>
    <!-- XML template specifying the "mode" parameter.  This XML
```

```
         template is processed where the <apply-xml-templates/>
         element occurred in the parent sequence's template, so the
         parameter will follow the <action-name> element. -->
    <xml-template>
      <action-param name="mode">
        <constant><apply-xml-templates/></constant>
      </action-param>
    </xml-template>
    <!-- mode tokens.  Because these have values, the tokens are
         associated with XML text nodes at the point where the
         <apply-xml-templates/> element occurred in the parent
         choice's template, i.e. within the <constant> element. -->
    <token value="this"><description>this way</description></token>
    <token value="that"><description>that way</description></token>
  </choice>
  <token><description>on</description></token>
  <literal>
    <!-- XML template for the "value" parameter.  Note that the
         value of the literal will be associated with an XML text
         node processed at the point of <apply-xml-templates/>. -->
    <xml-template>
      <action-param name="value">
        <constant><apply-xml-templates/></constant>
      </action-param>
    </xml-template>
    <required/>
    <default><value type="java.lang.Integer">1</value></default>
  </literal>
</sequence>
```

Standard XML Template Patterns

The following examples illustrate typical XML template patterns used in scenario grammar elements.

Actions

```
<sequence id="action-myAction">
  ...
  <xml-template>
    <action-name>myAction</action-name>
    <apply-xml-templates/>
  </xml-template>
  ...
</sequence>
```

The above XML template will cause the PDL fragment `<action-name>myAction</action-name>` to be associated with this custom action. The embedded `<apply-xml-templates/>` element specifies that the sequence's children will parse and generate their own XML following the `<action-name>` element.

(Note: the enclosing `<action>` element is automatically generated by the scenario editor and requires no template).

Events

```
<sequence id="event-message-MyProject.MyEvent">
  ...
  <xml-template>
    <event-name>MyProject.MyEvent</event-name>
    <apply-xml-templates/>
  </xml-template>
  ...
</sequence>
```

Similarly to the action example, the XML template here provides an `<event-name>` element to be generated with an enclosing `<event>` element.

Conditions

```
<sequence id="condition-myCondition">
  ...
  <xml-template>
    <filter operator="myCondition">
      <apply-xml-templates/>
    </filter>
  </xml-template>
  ...
</sequence>
```

The XML template here provides a `<filter>` element to be generated with an enclosing `<condition>` element. Note that the operator attribute must correspond to the condition name, and that the positioning of `<apply-xml-templates/>` causes the sequence's children to be generated within the `<filter>` PDL element, not after it.

Literal Constants and Token Constants

```
<literal>
  <xml-template>
    <action-param name="logInteger">
      <constant><apply-xml-templates/></constant>
    </action-param>
  </xml-template>
  <default><value type="java.lang.Integer">1</value></default>
</literal>
```

In this typical pattern, a literal is associated with an XML template that specifies a <constant> PDL element. The <apply-xml-templates/> element is positioned such that the literal's value will be generated within the <constant> tag, and its type will be attached to that tag as an attribute.

A similar technique applies to tokens that represent constants.

Tokens for PDL Fragment Choices

```
<token>
  <xml-template>
    <jndi-property>
      <jndi-url>dynamo:/atg/dynamo/service/CurrentDate</jndi-url>
      <property-name>timeAsDate</property-name>
    </jndi-property>
  </xml-template>
  <description>now</description>
</token>
```

In this pattern, a token is not associated with any value, but its XML template serves to generate a particular PDL fragment outright, without variation. Because the token has no value, the <apply-xml-templates/> element is not used in the template.

Special-Purpose Grammar Extension Tags

This section describes additional tags that you can use within a grammar extension for the Scenarios module.

Constraining a Sequence to a JMS Message Type

It is possible to constrain a sequence to a JMS message type. This effectively makes the sequence invisible within the user interface unless it is used in a place in the scenario where the given JMS message is in scope as the most recent event. This is done by providing the element <required-message-sequence message-type="..."/> as a child of the grammar element. For instance, one can state that a custom condition may only be used as an antecedent of a given event:

```
<required-message-sequence
    id="condition-MyCondition"
    message-type="MyProject.MyEvent">
  ...
</required-message-sequence>
```

Including a Generic Scenario Subexpression

Scenario user interface expressions incorporate the notion of a "generic subexpression." Such an expression may be a constant, or a variable, or a property of the current event, or of the subject, or of a globally visible bean. To include a generic subexpression as a child of your custom grammar construct, embed this tag:

```
        <scenario-expression type="..."/>
```

where the `type` attribute is a Java class name denoting the type to which the subexpression should be constrained.

Including an Array of RepositoryItem IDs

A special tag can be used to provide a token whose value is an array of String IDs of repository items:

```
        <repository-item-set
            repository-name="..."
            repository-item-type="..."
            />
```

This tag is automatically associated with an `<array>` element in PDL.

Commerce-Related Grammar Configuration

If you are extending the grammar of the Scenarios module for use with ATG Commerce scenario elements, you can make use of some additional specialized tags.

The header for a grammar extension definition file that includes ATG Commerce scenario elements looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?xcl-stylesheet resource="atg/ui/commerce/scenario/commerce-grammar.xsl"?>
<?xcl-stylesheet resource="atg/ui/scenario/expression/scenario-grammar.xsl"?>
<?xcl-stylesheet resource="atg/ui/expreditor/xcl/grammar.xsl"?>
```

Constraining a Sequence to an Order-Related Event

You can specify a sequence whose contained nodes are invisible in the user interface unless the scenario element that owns the sequence can see an event in its scope that has an `order` bean property of type `atg.commerce.order.Order`. To specify this type of sequence, use the `<required-order-sequence/>` element.

Constraining a Sequence to a Commerce-Item-Related Event

You can specify a sequence whose contained nodes are invisible in the user interface unless the scenario element that owns the sequence can see an event in its scope that has an `item` bean property of type `atg.commerce.order.CommerceItem`. To specify this type of sequence, use the `<required-commerce-item-sequence/>` element.

Suggestions for Localization

The following list shows the main tasks to perform when you localize extensions to the grammar editor:

1. Examine the XML files for the grammar templates that ship in the ATG product distribution (the source .xml files are in the distribution along with the .ser files), and consider their relationship to the behavior of the user interface.
2. Prepare locale-specific files that correspond to the ATG template names described earlier in this document, using the language and optional country name to form the filename according to the search rules.
3. Select a suitable character encoding and declare it in the grammar.
4. Translate all description and editor-text elements using the above encoding to represent the translation.
5. Reorder the children of sequence elements where feasible—in general, this is only permitted where the children are either noise words (value-less tokens) or constructs labeled with unique IDs.
6. Eventually, after development and testing, serialize the XML files into correspondingly named .ser files.

Supported Character Encodings

If a name is given in parentheses, this is the value that must be supplied in the encoding attribute in the xml instruction on the first line of the file. For example to select Japanese Shift JIS, use this instruction:

```
<?xml version="1.0" encoding="Shift_JIS" standalone="no"?>
```

The full list of supported XML encodings can be found at:

```
ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets
```

Note: Not all encodings may be supported by the XML tools provided with the ATG product suite.

Part III: Web Services for Personalization and Scenarios

The ATG product suite includes a set of Web services that you can use to make calls to Dynamo from non-ATG applications. The services provided with ATG Personalization and ATG Scenarios allow you to retrieve and change user profile information, perform content targeting operations, and send profile-related messages to the Dynamo Messaging System. For example, users can log into an ATG application and perform profile updates; you can retrieve the profile ID of a given user; you can display content items that are tailored to the current user; and you can fire an event message if a user views a specific piece of content.

Many of the se Web services were designed to have the same functionality as their non-Web service counterparts. For example, standard Profile Form Handler operations have parallel Web service operations. This behavior helps you reproduce any code extensions that you have made to non-Web service classes.

This chapter contains the following information:

Web Services Module for ATG Personalization and Scenarios

The default Web services for Personalization and Scenarios are packaged in the `DPS.WebServices` module in the ATG Personalization layer. The code that is called for these Web services is located in the DAF, Personalization, or Scenarios layer, depending on the service.

For information on how to include this module in your application, refer to the *ATG Installation and Configuration Guide*.

User Profiling Web Services

The user profiling Web services provided with the ATG product suite allow you to access the ATG profile repository from a non-ATG application. You can use these Web services to perform login and logout operations, create new profiles and change existing ones, and manage passwords.

This section contains the following topics:

Note: In the sections that follow, some Web service descriptions include URLs with the variables *`hostname:port`*. For *`hostname`*, use the name of the machine running your application. For *`port`*, enter the number of the port that your application server uses to handle HTTP requests on that machine.

ProfileServices Component

The `/atg/userprofiling/ProfileServices` component (class `atg.userprofiling.ProfileServices`) manages many of the Web service functions related to user profiling. Note that some of these functions mimic the behavior of standard ATG profile form handlers; this behavior is designed to help you reproduce any profile form handler custom code as extensions of out-of-the-box Web services.

The following list shows the properties of the `ProfileServices` component.

- `transactionManager`

  Type: `javax.transaction.TransactionManager`

  The service that manages any transactions used to execute repository methods on this instance.

  Default: `/atg/dynamo/transaction/TransactionManager` (set in DPS module)

- `mappingManager`

Type: atg.repository.xml.ItemDescriptorMappingManager

The component that manages mapping files based on repository item descriptor name combinations. Any methods that return repository items consult this service to retrieve the mapping files to use when transforming these items into XML. Note that this behavior assumes the useDefaultMappingFiles property is set to true.

Default: /atg/repository/xml/ItemDescriptorMappingManager (set in DPS module)

- xmlGetService

  Type: atg.repository.xml.GetService

  The service that turns repository items into XML.

  Default: /atg/repository/xml/GetService (set in DPS module)

- xmlAddService

  Type: atg.repository.xml.AddService

  The service that adds repository items in XML format to a repository.

  Default: /atg/repository/xml/AddService (set in DPS module)

- xmlUpdateService

  Type: atg.repository.xml.UpdateService

  The service that takes repository items in XML format and updates them in their corresponding repositories.

  Default: /atg/repository/xml/UpdateService (set in DPS module)

- profileTools

  Type: atg.userprofiling.ProfileTools

  The profile tools service that provides access to common profiling functions, as well as access to other profile components such as the PropertyManager and ProfileRepository.

  Default: /atg/userprofiling/ProfileTools (set in DPS module)

- updateEventListeners

  Type: atg.userprofiling.ProfileUpdateListener[]

  An array of ProfileUpdateListeners that are notified when a profile is updated. Note that the generateUpdateEvents property must be set to true for this behavior to occur.

  Default: /atg/userprofiling/ProfileUpdateTrigger (set in DPS module)

- generateLoginEvents

  Type: boolean

  If true, the LoginUser Web service fires login events after a user has been successfully logged in.

  Default: true (set in DPS module)

- `generateUpdateEvents`

  Type: boolean

  If true, the `UpdateUser` Web service fires update events after a user profile has been successfully updated.

  Default: true (set in DPS module)

- `generateRegisterEvents`

  Type: boolean

  If true, the `CreateUser` Web service fires register events after a user profile has been successfully created.

  Default: true (set in DPS module)

- `generateLogoutEvents`

  Type: boolean

  If true, the `logoutUser` service fires logout events after a user has been successfully logged out.

  Default: true (set in DPS module)

- `expireSessionOnLogout`

  Type: boolean

  Behaves the same way as the `expiresSessionOnLogout` property in the `ProfileFormHandler` class. See the description of the `handleLogout` method in Profile Form Handlers.

  Default: true (set in DPS module)

- `usingLDAPProfile`

  Type: boolean

  Behaves the same way as the `usingLDAPProfile` property in the `ProfileFormHandler` class. It indicates whether the profile in use is stored in an LDAP directory.

  Default: false (set in DPS module)

- `useDefaultMappings`

  Type: boolean

  If true, methods that return `Repo2Xml` items map properties according to the default mappings configured in the `ItemDescriptorMappingManager` (unless the Web service calls a variation of a method that takes a mapping file location as an argument).

  Default: true (set in DPS module)

- `logoutProfileType`

  Type: String

Behaves the same way as the logoutProfileType property in the ProfileFormHandler class. It specifies the item descriptor type of the profile defaulted to when the current user logs out.

Default: user (set in DPS module)

- createProfileType

  Type: String

  Behaves the same way as the createProfileType property in the ProfileFormHandler class. It specifies the item descriptor type of the profile created when a new profile is added to the repository.

  Default: user (set in DPS module)

- loginProfileType

  Type: String

  Behaves the same way as the loginProfileType property in the ProfileFormHandler class. It specifies the item descriptor type of the profile accessed when a user logs in.

  Default: user (set in DPS module)

- propertiesToCopyOnLogin

  Type: String[]

  Behaves the same way as the propertiesToCopyOnLogin property in the ProfileFormHandler class. It specifies the property values to copy from the anonymous profile to the persistent profile when a user logs in.

  Default: null (set in DPS module)

- propertiesToAddOnLogin

  Type: String[]

  Behaves the same way as the propertiesToAddOnLogin property in the ProfileFormHandler class. It specifies the properties to add from the anonymous profile to the persistent profile when a user logs in.

  Default: scenarioInstances, slotInstances (set in DSS module)

- maxAuthenticationWait

  Type: long

  Indicates how long a password encryption conversation is allowed to last, which is the length of time between calls to getPasswordHashKey and loginUser. If the length of time between these two calls exceeds the maxAuthenticationWait property value, the login is invalid. Note that this behavior assumes the login attempt is encrypted (in other words, that pIsPasswordEncrypted is true for the loginUser call).

  Default: 30000 (30 seconds) (set in DPS module)

- badPasswordDelay

  Type: long

Behaves the same way as the badPasswordDelay property in the ProfileFormHandler class. It specifies the number of milliseconds to wait before proceeding after a user submits a password that fails authentication.

Default: 1000 (1 second) (set in DPS module)

- profilePath

  Type: String

  The component path of the Profile component.

  Default: /atg/userprofiling/Profile (set in DPS module)

- requestLocalePath

  Type: String

  The component path of the RequestLocale component.

  Default: /atg/dynamo/servlet/RequestLocale (set in DPS module)

- allowEncryptedPasswords

  Type: boolean

  If true, allows login attempts that use passwords encrypted on the client. Even if this property is true, the canClientEncryptPasswords() method still needs to return true to verify that the application itself can handle passwords encrypted in this way.

  Default: true (set in DPS module)

GetProfileId Web Service

The GetProfileId Web service finds the profile that matches the login supplied by the method call and returns its profile ID. For an example of how to invoke this Web service in a client application, see Example: Using the GetProfileId Web Service in an Axis Client.

EAR file	userprofilingWebServices.ear
WAR file	usersession.war
Context root	userprofiling/usersession
Servlet	getProfileId
Web Service URL	http://*hostname:port*/userprofiling/usersession/getProfileId
WSDL URL	http://*hostname:port*/userprofiling/usersession/getProfileId ?WSDL
Web Service Class	webservice.GetProfileIdSEIImpl
Input Parameters	String Login
Output	String ProfileId

Nucleus Component	/atg/userprofiling/ProfileServices (class atg.userprofiling.ProfileServices)
Method	getProfileId(String pLogin)
Executes within a session	Yes
Security FunctionalName	profileInfoOperation

getProfileId Method

The GetProfileId Web service calls the getProfileId method in the underlying atg.userprofiling.ProfileServices implementation.

The getProfileId method does the following:

1. Calls ProfileTools.getItem() with the given login.
2. Returns the ID of the found item or null if no item exists with that login.

Security Recommendation

It is recommended that you restrict the ability to call the GetProfileId Web service to administrative users.

GetProfile Web Service

The GetProfile Web service finds the profile that matches the supplied profile ID and returns it as a RepositoryItem in XML form (a Repo2Xml item). For more information, see Returning Repository Items as Repo2Xml Items. You can configure a mapping file for this XML item on the component that contains the getProfileId method. (It is recommended, for example, that you define the mapping file so that the password property is omitted when the item is returned.)

By default, the profile repository that the GetProfile Web service accesses is the /atg/userprofiling/ProfileAdapterRepository (specifically, the user item descriptor).

EAR file	userprofilingWebServices.ear
WAR file	usersession.war
Context root	userprofiling/usersession
Servlet	getProfile
Web Service URL	http://*hostname:port*/userprofiling/usersession/getProfile
WSDL URL	http://*hostname:port*/userprofiling/usersession/getProfile?WSDL

Web Service Class	`webservice.GetProfileSEIImpl`
Input Parameters	String `ProfileId`
Output	String `ProfileAsXML`
Nucleus Component	`/atg/userprofiling/ProfileServices` (class `atg.userprofiling.ProfileServices`)
Method	`getProfile(String pProfileId)`
Executes within a session	Yes
Security `FunctionalName`	`profileInfoOperation`

getProfile Method

The `GetProfile` Web service calls the `getProfile` method in the underlying `atg.userprofiling.ProfileServices` implementation. The method invokes `getItem` on the profile repository and transforms the result into a `Repo2Xml` item. An HTTP request is not required for this method.

Security Recommendation

It is recommended that you restrict the ability to call the `GetProfileId` Web service to administrative users.

LoginUser Web Service

The `LoginUser` Web service authenticates the identity of the user for whom the service was called, returning the user's profile ID if authentication is successful.

EAR file	`userprofilingWebServices.ear`
WAR file	`usersession.war`
Context root	`userprofiling/usersession`
Servlet	`loginUser`
Web Service URL	http://*hostname:port*/userprofiling/usersession/loginUser
WSDL URL	http://*hostname:port*/userprofiling/usersession/loginUser?WSDL
Web Service Class	`webservice.LoginUserSEIImpl`

Input Parameters	String Login String Password boolean IsPasswordEncrypted
Output	String ProfileId
Nucleus Component	/atg/userprofiling/ProfileServices (class atg.userprofiling.ProfileServices)
Method	loginUser(String pLogin, String pPassword, boolean pIsPasswordEncrypted)
Executes within a session	Yes
Security FunctionalName	loginOperation.

loginUser Method

The LoginUser Web service calls the loginUser method in the underlying atg.userprofiling.ProfileServices implementation. The loginUser method behaves the same way as the handleLogin method in the ProfileFormHandler (see The ProfileForm Class). Note that loginUser should be called only in the context of an HTTP request; otherwise an error occurs.

loginUser takes the supplied login name and password and uses them to locate a valid profile. To do this, it calls the following methods:

- preLoginUser(DynamoHttpServletRequest pRequest, DynamoHttpServletResponse pResponse)
- doLoginUser(DynamoHttpServletRequest pRequest, DynamoHttpServletResponse pResponse)
- postLoginUser(DynamoHttpServletRequest pRequest, DynamoHttpServletResponse pResponse)

The preLoginUser and postLoginUser methods are similar to the ProfileFormHandler's preLoginUser and postLoginUser methods, which are stubs designed to allow subclasses to control login logic before and after the login process. You can write extensions to this code by overriding these methods with your own custom subclasses.

Unlike the ProfileFormHandler, the loginUser Web service immediately propagates to the caller any errors that occur during processing. Errors are not stored and shown to the user because the caller in this case is an RPC client that does not have access to a request/response pair. (The ProfileFormHandler, by contrast, expects to have its errors shown on an HTML page where the user can correct them and resubmit.) This error-handling behavior is used by all profile-related Web services that mimic ProfileFormHandler functionality.

The loginUser method acts as follows:

1. If pIsPasswordEncrypted is true, the loginUser method checks to see if a valid password encryption conversation has occurred in this session. If not, a ServletException is thrown.
2. loginUser calls the preLoginUser method, which checks that the session associated with the current profile is not transient and then does the following:
    - If the current profile's login does not match the login that was passed to the Web service, the current profile is logged out, and its session is expired.
    - If the login and password given to the Web service match those in the current profile, it is assumed that the same user is logging in again. In this case, an exception is thrown so that login events and profile cookies are not resent. In addition, the securityStatus of the profile is reset to the login securityStatus (if securityStatus is enabled).
    - If the password passed to the Web service does not match the password stored for the given login name, a ServletException is thrown.
3. loginUser calls doLoginUser, which attempts to authenticate the user based on the given credentials. It forward-hashes the stored password for the given user with a hashKey initialized during the password encryption conversation and compares it to the password argument. If this check succeeds, the RepositoryItem for that user is set as the current profile's data source, and repository properties are copied and/or added from the guest user's profile to the authenticated user's profile.
4. The profile ID is returned for the user who just logged in.
5. If the password encryption comparison fails, indicating that either the login name or the password was invalid, null is returned by doLoginUser.
6. The loginUser method calls the postLoginUser method, which sends a login event if configured to do so (set generateLoginEvents to true in the ProfileServices component). It also sends profile cookies if necessary, sets the security status for the logged-in profile, and changes the request locale to reflect the logged-in profile's locale.

Security Recommendation

Users are not authenticated before the LoginUser Web service is called, so there is no specific security policy recommendation for this service.

LogoutUser Web Service

The LogoutUser Web service attempts to log out the user for whom the logoutUser method is called (the user associated with the current session).

EAR file	userprofilingWebServices.ear
WAR file	usersession.war
Context root	userprofiling/usersession

Servlet	`logoutUser`
Web Service URL	`http://`*`hostname:port`*`/userprofiling/usersession/logoutUser`
WSDL URL	`http://`*`hostname:port`*`/userprofiling/usersession/logoutUser?WSDL`
Web Service Class	`webservice.LogoutUserSEIImpl`
Input Parameters	None
Output	None
Nucleus Component	`/atg/userprofiling/ProfileServices` (class `atg.userprofiling.ProfileServices`)
Method	`logoutUser()`
Executes within a session	Yes
Security `FunctionalName`	`logoutOperation`.

logoutUser Method

The `LogoutUser` Web service calls the `logoutUser` method in the underlying `atg.userprofiling.ProfileServices` implementation. The `logoutUser` method behaves the same way as the `handleLogout` method in the `ProfileFormHandler` (see The ProfileForm Class). Note that `logoutUser` should be called only in the context of an HTTP request; otherwise an error occurs.

`logoutUser` invokes the following methods:

- `preLogoutUser(DynamoHttpServletRequest pRequest, DynamoHttpServletResponse pResponse)`
- `doLogoutUser(DynamoHttpServletRequest pRequest, DynamoHttpServletResponse pResponse)`
- `postLoginUser(DynamoHttpServletRequest pRequest, DynamoHttpServletResponse pResponse)`

These methods are similar to the `ProfileFormHandler`’s `preLogoutUser`, `handleLogout`, and `postLogoutUser` methods.

The `logoutUser` method acts as follows:

1. Calls `preLogoutUser`, which sets up a logout event and revokes the current user’s identity.
2. Calls `doLogoutUser`.
3. Calls `postLogoutUser`, which fires a logout event and expires the current session if configured to do so. (You can control this behavior through the

generateLogoutEvents and expireSessionOnLogout properties in the ProfileServices component.)

Security Recommendation

It is recommended that you restrict the ability to call the LogoutUser service to appropriate users. For example, you would generally not want anonymous or guest users to be able to invoke LogoutUser.

CreateUser Web Service

The CreateUser Web service attempts to create a new profile from the information in the input ProfileAsXML String, which is a Repo2XML item. The profile ID for the new profile is returned.

Any password present in the ProfileAsXML argument must be cleartext, so it is recommended you always use a secure connection to call this service.

EAR file	userprofilingWebServices.ear
WAR file	usersession.war
Context root	userprofiling/usersession
Servlet	createUser
Web Service URL	http://*hostname:port*/userprofiling/usersession/createUser
WSDL URL	http://*hostname:port*/userprofiling/usersession/createUser?WSDL
Web Service Class	webservice.CreateUserSEIImpl
Input Parameters	String ProfileAsXML
Output	String ProfileId
Nucleus Component	/atg/userprofiling/ProfileServices (class atg.userprofiling.ProfileServices)
Method	createUser(String pProfileAsXml)
Executes within a session	Yes
Security FunctionalName	createOperation

createUser Method

The createUser Web service calls the createUser method in the underlying atg.userprofiling.ProfileServices implementation. The createUser method behaves the same way as the handleCreate method in the ProfileFormHandler (see The ProfileForm Class). Note that createUser should be called only in the context of an HTTP request; otherwise an error occurs.

createUser invokes the following methods:

- preCreateUser(DynamoHttpServletRequest pRequest, DynamoHttpServletResponse pResponse)
- doCreateUser(DynamoHttpServletRequest pRequest, DynamoHttpServletResponse pResponse)
- postCreateUser(DynamoHttpServletRequest pRequest, DynamoHttpServletResponse pResponse)

These methods are similar to the ProfileFormHandler's preCreateUser, createUser, and postCreateUser methods.

The createUser method acts as follows:

1. Calls preCreateUser.
2. Calls doCreateUser, which uses a Repo2Xml AddService to add the given item to the repository. If successful, the item is then set as the data source for the current profile. Any password present in the given item is also encrypted for storage. (As mentioned above, it is expected that passwords given in the pProfileAsXML are cleartext.)
3. Calls postCreateUser, which sets profile cookies if required to do so. postCreateUser also fires a register event if you set the generateRegisterEvents to true in the ProfileServices component.

Security Recommendation

The user has no credentials before the CreateUser Web service is called, so there is no specific security policy recommendation for it.

UpdateUser Web Service

The UpdateUser Web service attempts to change a profile using information passed in as the ProfileAsXML argument (a Repo2XML item).

Note: Do not use this service for updating passwords. If a password property were included in the Repo2Xml item for this ProfileAsXML argument, the value would be persisted as it appeared in the file; the password string would not be encrypted, resulting in invalid login attempts for the user in the future. To change a password, use the setPassword Web service instead.

EAR file	userprofilingWebServices.ear
WAR file	usersession.war
Context root	userprofiling/usersession

Servlet	updateUser
Web Service URL	http://*hostname:port*/userprofiling/usersession/updateUser
WSDL URL	http://*hostname:port*/userprofiling/usersession/updateUser?WSDL
Web Service Class	webservice.UpdateUserSEIImpl
Input Parameters	String ProfileAsXML
Output	None
Nucleus Component	/atg/userprofiling/ProfileServices (class atg.userprofiling.ProfileServices)
Method	updateUser(String pProfileAsXml)
Executes within a session	Yes
Security FunctionalName	xmlProfileOwnerOperation

updateUser Method

The updateUser Web service calls the updateUser method in the underlying atg.userprofiling.ProfileServices implementation. The updateUser method behaves the same way as the handleUpdate method in the ProfileFormHandler (see The ProfileForm Class). Note that updateUser should be called only in the context of an HTTP request; otherwise an error occurs.

updateUser invokes the following methods:

- preUpdateUser(DynamoHttpServletRequest pRequest, DynamoHttpServletResponse pResponse)
- doUpdateUser(DynamoHttpServletRequest pRequest, DynamoHttpServletResponse pResponse)
- postUpdateUser(DynamoHttpServletRequest pRequest, DynamoHttpServletResponse pResponse)

These methods are similar to the ProfileFormHandler's preUpdateUser, updateUser, and postUpdateUser methods.

The updateUser method acts as follows:

1. Calls preUpdateUser, which sets up a ProfileUpdateEvent but does not fire it.
2. Calls doUpdateUser, which uses a Repo2Xml update service to update the given pProfileAsXML.

3. Calls postUpdateUser, which may revoke profile cookies if the user's autoLogin property changes from true to false. It also fires a ProfileUpdateEvent if configured to do so (set generateUpdateEvents to true in the ProfileServices component).

Security Recommendation

Apply the atg.userprofiling.ProfileAsXMLOwnerPolicy to this service. This security policy verifies that the user associated with the current session matches the user whose profile is being updated. The functional name (xmlProfileOwnerOperation) differs from the profileOwnerOperation used by other services that are based on profile ownership because the security argument in this service is a Repo2Xml item rather than a profile ID. For more information, see ProfileAsXMLOwnerPolicy.

Set Password Web Service

The SetPassword Web service changes the password of the user specified by the ProfileId argument. It requires the user's current password for security reasons. The service checks the value of the supplied password (OriginalPassword) against the value stored for this user in the profile repository. If the values match, the password is changed to the new password.

Do not encrypt OriginalPassword or NewPassword. Both are expected to be cleartext as they are forward-hashed for comparison with the stored password value.

EAR file	userprofilingWebServices.ear
WAR file	usersession.war
Context root	userprofiling/usersession
Servlet	setPassword
Web Service URL	http://*hostname:port*/userprofiling/usersession/setPassword
WSDL URL	http://*hostname:port*/userprofiling/usersession/setPassword?WSDL
Web Service Class	webservice.SetPasswordSEIImpl
Input Parameters	String ProfileId String OriginalPassword String NewPassword
Output	None
Nucleus Component	/atg/userprofiling/ProfileServices (class atg.userprofiling.ProfileServices)
Method	setPassword(String pProfileId, String pOriginalPassword, String pNewPassword)
Executes within a session	Yes

Security FunctionalName	profileOwnerOperation

setPassword Method

The SetPassword Web service calls the setPassword method in the underlying atg.userprofiling.ProfileServices implementation. Note that setPassword should be called only in the context of an HTTP request; otherwise an error occurs. The method does the following:

1. Calls preSetPassword.
2. Calls doSetPassword, which compares the password currently stored for the user (specified by the supplied profile ID) with the given pOldPassword. If the check succeeds, the user's password is set to pNewPassword.
3. Calls postSetPassword.

Security Recommendation

Apply the atg.userprofiling.ProfileOwnerPolicy to this Web service. This policy requires that the ProfileId argument match the profile ID of the user who is calling the method. This behavior ensures that a password can be changed only by the person who owns the profile. For more information, see ProfileOwnerPolicy.

SetContactInfo Web Service

The SetContactInfo Web service sets or changes the contact information of the user specified by the ProfileId argument to the value of the supplied ContactInfoAsXML item. If the given ContactInfoAsXML represents an item that does not yet exist in the repository, it is added. This service assumes that the User item descriptor for your ProfileAdapterRepository component defines a single-valued address property.

EAR file	userprofilingWebServices.ear
WAR file	usersession.war
Context root	userprofiling/usersession
Servlet	setContactInfo
Web Service URL	http://*hostname:port*/userprofiling/usersession/setContactInfo
WSDL URL	http://*hostname:port*/userprofiling/usersession/setContactInfo?WSDL
Web Service Class	webservice.SetContactInfoSEIImpl
Input Parameters	String ProfileId String ContactInfoAsXML

Output	None
Nucleus Component	/atg/userprofiling/ProfileServices (class atg.userprofiling.ProfileServices)
Method	setContactInfo(String pProfileId, String pContactInfoAsXml)
Executes within a session	Yes
Security FunctionalName	profileOwnerOperation

setContactInfo Method

The SetContactInfo Web service calls the setContactInfo method in the underlying atg.userprofiling.ProfileServices implementation. This method attempts to set the contact information of the user who is specified by the profile ID to the given value, which is represented as a Repo2Xml item. Use the PropertyManager.contactInfoPropertyName to specify the property where contact information is stored in your profile repository configuration. This property should be of type atg.repository.RepositoryItem.

The setContactInfo method acts as follows:

1. Checks that the given pProfileId resolves to a valid profile.
2. Takes the pContactInfoAsXML and uses it to get or create a RepositoryItem.
3. Sets the contact information property of the user represented by pProfileId to the RepositoryItem returned by the previous step.

Security Recommendation

Apply the atg.userprofiling.ProfileOwnerPolicy to this Web service. This policy requires that the ProfileId argument match the profile ID of the user who is calling the method. This behavior ensures that contact information can be changed only by the person who owns the profile. For more information, see ProfileOwnerPolicy.

SetLocale Web Service

The SetLocale Web service changes the locale property of the user represented by the specified profile ID to the value of the LocaleName argument. If the profile ID matches the ID of the user associated with the current session, the RequestLocale component is also changed appropriately.

EAR file	userprofilingWebServices.ear
WAR file	usersession.war
Context root	userprofiling/usersession

Servlet	`setLocale`
Web Service URL	`http://`*`hostname:port`*`/userprofiling/usersession/setLocale`
WSDL URL	`http://`*`hostname:port`*`/userprofiling/usersession/setLocale?WSDL`
Web Service Class	`webservice.SetLocaleSEIImpl`
Input Parameters	String `ProfileId` String `LocaleName`
Output	None
Nucleus Component	`/atg/userprofiling/ProfileServices` (class `atg.userprofiling.ProfileServices`)
Method	`setLocale(String pProfileId, String pLocaleName)`
Executes within a session	Yes
Security `FunctionalName`	`profileOwnerOperation`

setLocale Method

The `SetLocale` Web service calls the `setLocale` method in the underlying `atg.userprofiling.ProfileServices` implementation. This method attempts to set the locale of the user who is specified by the profile ID to the given value. Use the `PropertyManager.localePropertyName` to specify the property where the locale is stored in your profile repository configuration.

The `setLocale` method acts as follows:

1. Checks that the given `pProfileId` resolves to a valid profile.
2. If the locale property exists, sets the locale property of the given user to `pLocaleName`.
3. If the user whose locale is being changed is also the user associated with the current session, the `RequestLocale` component is also changed to reflect the new locale.

Security Recommendation

Apply the `atg.userprofiling.ProfileOwnerPolicy` to this Web service. This policy requires that the `ProfileId` argument match the profile ID of the user who is calling the method. This behavior ensures that locale information can be changed only by the person who owns the profile. For more information, see ProfileOwnerPolicy.

CanClientEncryptPasswords Web Service

The `CanClientEncryptPasswords` Web service is used as part of the optional client-side encryption feature (see Using Client-Side Password Encryption) that you can use with the `LoginUser` Web service. It is a utility service that checks to see if a client is configured and able to encrypt passwords for sending via an ATG Web service. Currently, only the `LoginUser` Web service can handle passwords encrypted by the client.

EAR file	`userprofilingWebServices.ear`
WAR file	`usersession.war`
Context root	`userprofiling/usersession`
Servlet	`canClientEncryptPasswords`
Web Service URL	`http://`*`hostname:port`*`/userprofiling/usersession/canClientEncryptPasswords`
WSDL URL	`http://`*`hostname:port`*`/userprofiling/usersession/canClientEncryptPasswords?WSDL`
Web Service Class	`webservice.CanClientEncryptPasswordsSEIImpl`
Input Parameters	None
Output	boolean `CanEncryptPasswords`
Nucleus Component	`/atg/userprofiling/ProfileServices` (class `atg.userprofiling.ProfileServices`)
Method	`canClientEncryptPasswords()`
Executes within a session	Yes
Security `FunctionalName`	`loginOperation`

canClientEncryptPasswords Method

The `CanClientEncryptPasswords` Web service calls the `canClientEncryptPasswords` method in the underlying `atg.userprofiling.ProfileServices` implementation.

The `canClientEncryptPasswords` method checks that all of the following are true:

1. The `allowEncryptedPasswords` property of `ProfileServices` is set to true.
2. The password hasher configured for the application supports temporary encryption keys.
3. The password hasher configured for the application is supported.

In all other cases, this method returns false.

For more information on the way this method is used, refer to Using Client-Side Password Encryption.

Security Recommendation

You can apply any appropriate security policy to the CanClientEncryptPasswords Web service. As this service is used in conjunction with the LoginUser service, it is suggested that you use the same security policy for both.

GetPassWordHashKey Web Service

The GetPassWordHashKey Web service is used as part of the optional client-side encryption feature (see Using Client-Side Password Encryption). It returns a temporary hashKey used by the client to encrypt a password for a single authentication call. The client forward-encrypts the password with this hash key. The server then stores the hash key in the current session so you do not have to return it to the LoginUser service.

EAR file	userprofilingWebServices.ear
WAR file	usersession.war
Context root	userprofiling/usersession
Servlet	getPasswordHashKey
Web Service URL	http://*hostname:port*/userprofiling/usersession/getPasswordHashKey
WSDL URL	http://*hostname:port*/userprofiling/usersession/getPasswordHashKey?WSDL
Web Service Class	webservice.GetPasswordHashKeySEIImpl
Input Parameters	None
Output	String PasswordHashKey
Nucleus Component	/atg/userprofiling/ProfileServices (class atg.userprofiling.ProfileServices)
Method	getPasswordHashKey()
Executes within a session	Yes
Security FunctionalName	loginOperation

getPasswordHashKey Method

The GetPasswordHashKey Web service calls the getPasswordHashKey method in the underlying atg.userprofiling.ProfileServices implementation.

The getPasswordHashKey method acts as follows:

1. Returns null if not called in the context of a request.
2. Sets up session attributes to indicate that a password encryption conversation has started.
3. If the password hasher does not support hash keys, this method returns null (in other words, `PasswordHasher.getPasswordHashKey()` returns null). If hash keys are supported, the method returns the value provided by the password hasher and sets that hash key as a session attribute of the current session.

For more information on the way this method is used, refer to Using Client-Side Password Encryption.

Security Recommendation

You can apply any appropriate security policy to the `GetPasswordHashKey` Web service. As this service is used in conjunction with the `LoginUser` service, it is suggested that you use the same security policy for both.

GetPasswordHashAlgorithm Web Service

The `GetPasswordHashAlgorithm` Web service is used as part of the optional client-side encryption feature (see Using Client-Side Password Encryption). It returns the name of the algorithm that the containing application on the server uses to hash passwords, for example MD5, SHA, or SSHA. (Note that only MD5 is currently supported.) Along with a `hashKey`, this service allows the client to encrypt the password and pass it to the `LoginUser` or `CreateUser` Web service over unsecured transport mechanisms.

EAR file	`userprofilingWebServices.ear`
WAR file	`usersession.war`
Context root	`userprofiling/usersession`
Servlet	`getPasswordHashAlgorithm`
Web Service URL	`http://`*`hostname:port`*`/userprofiling/usersession/getPasswordHashAlgorithm`
WSDL URL	`http://`*`hostname:port`*`/userprofiling/usersession/getPasswordHashAlgorithm?WSDL`
Web Service Class	`webservice.GetPasswordHashAlgorithmSEIImpl`
Input Parameters	None
Output	String `PasswordHashAlgorithm`
Nucleus Component	`/atg/userprofiling/ProfileServices` (class `atg.userprofiling.ProfileServices`)
Method	`getPasswordHashAlgorithm()`

Executes within a session	Yes
Security FunctionalName	loginOperation

getPasswordHashAlgorithm Method

The GetPasswordHashAlgorithm Web service calls the getPasswordHashAlgorithm method in the underlying atg.userprofiling.ProfileServices implementation. It is assumed that you call canClientEncryptPasswords() before calling this method, or that you know that this application can encrypt passwords.

The getPasswordHashAlgorithm method acts as follows:

1. If the PasswordHasher used for this application is not supported, the getPasswordHashAlgorithm method throws a ServletException. (Note that only MD5 is currently supported.)
2. Returns the password-hashing algorithm used for this application. Each password hasher has a unique way of encrypting passwords, so clients that attempt password encryption need to know the procedure that their password hasher uses.

Security Recommendation

You can apply any appropriate security policy to the GetPasswordHashAlgorithm Web service. As this service is used in conjunction with the LoginUser service, it is suggested that you use the same security policy for both.

Content Targeting Web Services

One content targeting Web service, RecommendContent, is provided by default with the ATG product suite. It allows you to use a content targeter (created using the Personalization module) or slot (created in the Scenarios module) to display text or images that are appropriate for the person associated with the current user profile.

This section contains the following topics:

Note: In the sections that follow, some Web service descriptions include URLs with the variables *hostname:port*. For *hostname*, use the name of the machine running your application. For *port*, enter the number of the port that your application server uses to handle HTTP requests on that machine.

TargetingServices Component

The /atg/targeting/TargetingServices component (class atg.targeting.TargetingServices) manages functions related to the Web services that perform targeting operations (by default, the RecommendContent Web service). The component has the following properties:

- xmlGetService

  Type: atg.repository.xml.GetService

  The service that turns RepositoryItems into XML

  Default: /atg/repository/xml/GetService (set in DPS module)

- mappingManager

  Type: atg.repository.xml.ItemDescriptorMappingManager

  The component that manages mapping files based on repository item descriptor name combinations. Any methods that return repository items consult this service to retrieve the mapping files to use when transforming these items into XML. Note that this behavior assumes the useDefaultMappingFiles property is set to true.

  Default: /atg/repository/xml/ItemDescriptorMappingManager (set in DPS module)

- useDefaultMappings

  Type: boolean

  If true, methods that return Repo2Xml items use the default mappings configured in the ItemDescriptorMappingManager (unless the Web service calls a variation of a method that takes a mapping file location as an argument).

  Default: true (set in DPS module)

For more information on using mapping files to manage Repo2Xml items, refer to Returning RepositoryItems as Repo2Xml Items.

RecommendContent Web Service

The RecommendContent Web service performs a content targeting operation, taking a slot or targeter path as input and returning *n* content items, where *n* is the value of the given howMany parameter. The content is returned in XML form as an array of Repo2Xml items. A value of -1 for the howMany parameter indicates that there is no limit to the number of returned results.

If called in the context of an HTTP request, this service can resolve both request- and session-scoped targeter and slot components. If no HTTP request is present, the service can resolve only globally-scoped components.

EAR file	userprofilingWebServices.ear
WAR file	usersession.war
Context root	userprofiling/usersession

Servlet	`recommendContent`
Web Service URL	`http://`*`hostname:port`*`/userprofiling/usersession/recommendContent`
WSDL URL	`http://`*`hostname:port`*`/userprofiling/usersession/recommendContent?WSDL`
Web Service Class	`webservice.ExecuteRepositoryTargeterSEIImpl`
Input Parameters	String `PathToSlotOrTargeter`. Represents the Nucleus component path of the targeter or slot to execute. int `StartingIndex`. Represents the 0-based index at which to start when returning values. For example, passing in "3" here returns an array of items starting with the fourth item that is returned from the targeting operation (in other words, the first three items are skipped). int `HowMany`. Indicates how many items should be returned in total from this targeter. If this value is -1, all items are returned, starting from the `StartingIndex` value.
Output	`String[] ContentAsXML`
Nucleus Component	`/atg/targeting/TargetingServices` (class `atg.targeting.TargetingServices`)
Method	`executeRepositoryTargeter (String pPathToSlotOrTargeter, int pStartingIndex, int pHowMany)`
Executes within a session	Yes
Security `FunctionalName`	`targetingOperation`

executeRepositoryTargeter Method

The `RecommendContent` Web service calls the `executeRepositoryTargeter` method in the underlying `atg.targeting.TargetingServices` implementation. This method does the following:

1. Resolves the given `pTargeterPath` by attempting to use the current request as a `NameResolver`.
2. If a slot or targeter is found, the `target()` method is called. `pStartingIndex` and `pMaxResults` are passed in as arguments.
3. Any results that are returned are changed into `Repo2Xml` items and passed back as a String.

Security Recommendation

There is no specific security policy recommendation for the `RecommendContent` Web service; it is assumed that you want all users to whom the targeter or slot applies to be able to call this service.

Messaging Web Services

The ATG product suite includes some Web services that you can use to send JMS messages to the Dynamo Messaging System. The default services send `PageVisit` and `Clickthrough` event messages, which you can use in a number of ways in your ATG application – for example, you can set up a scenario that is triggered whenever a `PageVisit` message with specific parameters is received.

This section contains the following topics:

Note: In the sections that follow, some Web service descriptions include URLs with the variables *`hostname:port`*. For *`hostname`*, use the name of the machine running your application. For *`port`*, enter the number of the port that your application server uses to handle HTTP requests on that machine.

MessagingImporter Component and ReceiveObjectMessage() Method

The default Web services that send JMS messages do so by calling the `receiveObjectMessage()` method exposed by the `/atg/dynamo/messaging/MessagingImporter` component. The `MessagingImporter` component is an instance of class `atg.dms.patchbay.MessageImporter`, which is part of the Web service infrastructure in the DAF layer.

The `receiveObjectMessage()` method takes three parameters:

- the message object
- a String indicating the `JMSType` of the message
- a String indicating the Patch Bay port name to use

The Web services that call this method take a message object as their input parameter.

For more information, refer to *JMS Web Services* in the *ATG Web Services and Integration Framework Guide*.

ContentViewed Web Service

The `ContentViewed` Web service sends a `PageVisitMessage` to Patch Bay.

EAR file	`userprofilingWebServices.ear`
WAR file	`messaging.war`
Context root	`userprofiling/messaging`
Servlet	`contentViewed`
Web Service URL	`http://`*`hostname:port`*`/userprofiling/messaging/contentViewed`

WSDL URL	http://*hostname:port*/userprofiling/messaging/contentViewed?WSDL
Web Service Class	webservice.SendPageVisitSEIImpl
Input Parameters	atg.userprofiling.dms.PageVisitMessage pMessage
Output	None
Nucleus Component	/atg/dynamo/messaging/MessagingImporter (class atg.dms.patchbay.MessageImporter)

Method	receiveObjectMessage (Object pMessage, String pJMSType, String pInputPortName)
	Note that for the default ContentViewed service:
	JMSType = atg.dps.PageVisit
	InputPortName = IndividualEvents

Executes within a session	Yes
Security FunctionalName	messagingOperation

Setting PageVisit Message Properties for a Web Service

When PageVisit JMS messages are sent within an ATG application rather than to an ATG application from a Web service, the values of various message properties are determined by the PageEventTrigger component on the ATG server. When you send these messages through a Web service, however, the PageEventTrigger is not available, and you must set them explicitly on the PageVisit object created by the client application.

Note that setting these values incorrectly will prevent any scenarios that are waiting for these messages from being fired.

Use the same settings for the PageVisit message properties that you would use for a message sent within an ATG application, with the following important exception: use the path property rather than scenarioPathInfo property to specify the path of the page that was visited. This behavior is necessary because only the PageEventTrigger contains the necessary logic for specifying the correct value for scenarioInfoPath.

Important: When you set up a scenario to watch for this event, do not use the document picker (Choose document...) to select the page to include in the Visits element; pages specified this way automatically use the scenarioInfoPath property. Instead, use the whose path or with Dynamo path option in the element to specify the path to the page, and make sure that the value you enter matches the value that you send in the message from the Web service. Both of the following examples are valid ways to specify the correct path in the scenario element:

```
Visits a page whose path is /en/index.jsp
```

Or

```
Visits a page with Dynamo path /en/index.jsp
```

For pages that are outside the Dynamo document root (the page is part of a non-DAS J2EE application) make sure the `path` property includes the application context root.

For more information on the `PageVisit` message, refer to PageVisit Event.

ContentConsumed Web Service

The `ContentConsumed` Web service sends a `ClickThroughMessage` to Patch Bay.

EAR file	`userprofilingWebServices.ear`
WAR file	`messaging.war`
Context root	`userprofiling/messaging`
Servlet	`contentConsumed`
Web Service URL	`http://`*`hostname:port`*`/userprofiling/messaging/contentConsumed`
WSDL URL	`http://`*`hostname:port`*`/userprofiling/messaging/contentConsumes?WSDL`
Web Service Class	`webservice.SendClickThroughSEIImpl`
Input Parameters	`atg.userprofiling.dms.ClickThroughMessage pMessage`
Output	None
Nucleus Component	`/atg/dynamo/messaging/MessagingImporter` (class `atg.dms.patchbay.MessageImporter`)

Method	`receiveObjectMessage (Object pMessage, String pJMSType, String pInputPortName)`
	Note that for the default `ContentConsumed` service:
	`JMSType = atg.dps.ClickThrough`
	`InputPortName = IndividualEvents`

Executes within a session	Yes
Security `FunctionalName`	`messagingOperation`

Refer to ClickThrough Event for information on the properties to set in the ClickThrough message object.

Example: Using the GetProfileId Web Service in an Axis Client

The following example shows how you could retrieve a profile ID from the ATG system by using the GetProfileId Web service in an Apache Axis client application. For more examples of client-side code, refer to the *ATG Web Services and Integration Framework Guide*.

For more information on using Axis tools to build client applications, visit http://ws.apache.org/axis/.

The class that builds clients is org.apache.axis.wsdl.WSDL2Java, which takes a WSDL file as one of its arguments. The Axis client generates classes that you can then use to make a call to the GetProfileId Web service. For example:

```
java org.apache.axis.wsdl.WSDL2Java -o c:\my\client\classes
/path/to/getProfileId.wsdl
```

This command creates a class called com.atg.www.webservices.GetProfileIdSEIServiceLocator. This class is responsible for locating and returning a handle to the service you want to invoke. It returns an item of type com.atg.www.GetProfileIdSEI, which is an interface that contains the single method you need. You can then call this method with the arguments that are appropriate.

The following example shows how you could use the generated classes as a standalone method within a client application:

```
import com.atg.www.webservices.*;

public callGetProfileId(String pProfileId) {
      // This finds our Web service
      GetProfileIdServiceLocator locator = new GetProfileIdServiceLocator();

      try {
        // This represents the Web service that we want to act on
        GetProfileIdSEI getProfileService = locator.getGetProfileIdSEIPort();

         // Now we can invoke our method on it...we know the method name
         // and args by either looking at the WSDL, or at the
         // Axis-generated code
         return getProfileService.getProfileId(pProfileId);
      }
      catch(javax.xml.rpc.ServiceException exc) {}
      catch(java.rmi.RemoteException exc) {}
      catch(com.atg.www.javax_servlet.ServletException exc) {}
      catch(com.atg.www.atg_security.SecurityException exc) {} }
```

Note that the last two exceptions shown are generated by Axis, but you can just catch regular exceptions if the type of exception being thrown is not important to your application.

Returning RepositoryItems as Repo2Xml Items

The Web services provided with the Personalization layer return repository items as `Repo2Xml` items. You can manipulate the returned item by means of mapping files that allow you to do the following:

- Block certain properties from being returned.
- Change the property names in the resulting `Repo2Xml` item.

For example, assume you are returning a `user` item, but you do not want the client calling the method to see the value of the `scenarioInstances` property. You could set up a mapping file for the `user` item descriptor that omits the `scenarioInstances` property when `Repo2Xml` items are created The syntax for these mapping files is defined in the *ATG Web Services and Integration Framework Guide*.

Applying Mapping Files to Repo2Xml Items

Each Web service method that returns `Repo2Xml` items has an additional method with an optional String argument for a mapping file name. For example, the `getProfile()` method has two signatures: `getProfile(String pProfileId)`, and `getProfile(String pProfileId, String pMappingFile)`. However, only the former is generated by default. If you want to use the second String and apply a mapping file, you must configure an `ItemDescriptorMapping` as described in the *ATG Web Services and Integration Framework Guide*. In addition, ensure that the `useDefaultMappings` property is set to true for the component that the Web service calls.

Profile-Related Security Policies for Web Services

The `atg.security.StandardSecurityPolicy` class, which is provided as part of the Dynamo Application Framework layer, supplies the core logic for controlling access to objects secured using access control lists (ACLs). For more information on the `StandardSecurityPolicy`, see the *Managing Access Control* chapter of the *ATG Programming Guide*. The security policies used for profile-related Web services, which are described in this section, extend the `StandardSecurityPolicy`, appending the ACL returned by the `StandardSecurityPolicy` with additional access control entries (ACEs) that either grant or deny access to specific personae. (Personae can be users, roles or organizations.)

For more information on how security policies work with Web services, refer to *Web Service Security* in the *ATG Web Services and Integration Framework Guide* and *Repository Web Service Security* in the *ATG Repository Guide*.

This section contains the following topics:

AppendACLPolicy

This class is abstract and provides the basis for appending entries to the StandardSecurityPolicy ACL. It extends the getEffectiveAccessControlList method of the StandardSecurityPolicy class to append entries to the ACL returned by the StandardSecurityPolicy instance.

The abstract methods for this class are detailed here.

getAdditionalACL

This abstract method returns the ACL that is appended to the StandardSecurityPolicy ACL. The getAdditionalACL method is an extension of the getEffectiveAccessControlList method that appends the ACL of the StandardSecurityPolicy instance with additional ACEs. The abstract getAdditionalACL method returns the ACL to which this is appended. If the StandardSecurityPolicy does not return an ACL, the return value of getAdditionalACL is used. If both are null, null is returned and access is granted, which is the default behavior of the StandardSecurityPolicy.

```
protected abstract AccessControlList getAdditionalACL(Object pSecuredObject) ;
```

Return Value

This abstract method returns the ACL that is appended to the StandardSecurityPolicy ACL.

MethodParameterPolicy

This is an abstract class that extends the AppendAclPolicy class. It provides the base implementation for policies that depend on incoming parameter values to determine access rights to the secured object. In the case of Web services, the secured object is always the Web service method. It provides a concrete implementation of getAdditionalACL that appends all the Personae returned by the abstract getPersonae method.

The abstract getPersonae method takes a map of parameter values created by the getMethodParametersFromSecuredObject method and the incoming SecuredObject instance. You can use the incoming parameter values, and the SecuredObject if necessary, to determine the Personae that should have access to the SecuredObject.

For example, a Web service might provide the ability to edit a profile. If you want to allow both the profile owner and users with the Admin role to make edits, the getPersonae method could return both Personae, resulting in an ACL that looks something like the following:

```
Profile$user$1234:execute;Profile$role$Admin:execute
```

The access rights for each `persona` appended to the ACL are defined in the static variable `NEW_PERSONA_RIGHTS`.

Important: If no `Personae` are returned from the `getPersonae` method, the ACL is appended with `deny` access for everyone.

Refer to the *ATG API Reference* for information on the abstract methods for this class.

ProfileOwnerPolicy

The `/atg/userprofiling/security/ProfileOwnerPolicy` component (class `atg.userprofiling.security.ProfileOwnerPolicy`) is a security policy designed for situations in which you want only the owner of a profile to be able to perform operations on that profile. The `ProfileOwnerPolicy` examines the supplied profile object to check that it matches the profile associated with the current session; it then appends the ACL with the owner of the profile.

This policy takes a method argument containing a profile object of type `String` or `RepositoryItem`.

By default, the `ProfileOwnerPolicy` looks for profile objects named `pProfileId`, `Profile`, `profileId`, and `profile`, in that order, and uses the first corresponding object that it finds. You can change these names by editing the value of the `profileParameterNames` property in the `ProfileOwnerPolicy` component.

ProfileAsXMLOwnerPolicy

The `/atg/userprofiling/security/ProfileAsXMLOwnerPolicy` component (class `atg.userprofiling.security.ProfileAsXMLOwnerPolicy`) is a security policy that is provided with the Personalization layer. It is similar to the `ProfileOwnerPolicy`, but it expects a method argument that contains a profile in `Repo2Xml` form. It examines this `Repo2Xml` item to check that the profile associated with the current session matches the profile in the method argument.

The behavior provided by this policy can be useful for the `UpdateUser` service, where you may want to ensure that only the owner of a given profile is allowed to update it. For example, if a user whose ID is 700 attempts to call the `updateUser` method with a `Repo2Xml` item that represents a profile with ID 900, the `ProfileAsXMLOwnerPolicy` prevents the method from being called. Specifically, the `ProfileAsXMLOwnerPolicy` looks for method arguments named `pProfileAsXML` and `ProfileAsXML` in that order. If either of those arguments is present, it uses the value for that argument to determine if the method caller has permission to execute the method.

By default, the `ProfileAsXMLOwnerPolicy` looks for profile objects named `pProfileAsXML`, `ProfileAsXML`, and `profileAsXML`, in that order, and uses the first corresponding object that it finds. You can change these names by editing the value of the `profileParameterNames` property in the `ProfileAsXMLOwnerPolicy` component.

RelativeRoleByProfileOrgPolicy

The /atg/userprofiling/security/RelativeRoleByProfileOrgPolicy component (class atg.userprofiling.security.RelativeRoleByProfileOrgPolicy) is a security policy implementation that extends the abstract class atg.userprofiling.security.RelativeRoleByOrganizationPolicy (see *ATG API Reference* for more information). It allows you to grant access to users with specific relative roles (also called organizational roles – for more information, see Working with the Dynamo User Directory). The roles allowed access are those assigned to the parent organization of the profile supplied in the input argument.

This policy takes a method argument containing a profile object of type String or RepositoryItem.

By default, the RelativeRoleByProfileOrgPolicy looks for profile objects named pProfileId, Profile, profileId, and profile, in that order, and uses the first corresponding object that it finds. You can change these names by editing the value of the profileParameterNames property in the RelativeRoleByProfileOrgPolicy component.

Assume you have a Web service that you want to be used exclusively by supervisors. You create a security policy for it called SupervisorsOnly that is an implementation of RelativeRoleByProfileOrgPolicy.

You configure the SupervisorsOnly component with a roleFunctionName property set to a single value:

```
roleFunctionNames=supervisor
```

When a user calls the Web service, the security policy creates an ACL that grants access to the supervisor role in the user's parent organization:

```
$Profile:role:supervisorRoleId
```

The security sub-system grants access if the calling user has an assigned relative role with the ID supervisorRoleId; otherwise access is denied.

Defining Security Functions and Policies

Each default Web service includes a functionalName setting. You can use this setting to apply a single security policy across many services at once; for example, you could apply the ProfileOwnerPolicy to all Web services whose security function setting is profileOwnerOperation. You can define these functional names and security policy relationships in the Web services administration interface.

ATG does not provide any security policy associations for the default Web services. You must determine which security policy you want to associate with each functional name, and then use the Web services administration interface to set up the relationship. Some suggestions for appropriate policies are provided in the description of each Web service in this chapter. For more information on security policies, refer to *Creating and Editing Security Configurations* in the *ATG Web Services and Integration Framework Guide*.

After you set up security policy associations, or if you change any of the default functional names, you must regenerate and then redeploy the affected Web services. For more information on this procedure, see the *ATG Web Services and Integration Framework Guide.*

The following list shows the `functionalName` setting for each Web service:

Web Service	**Functional Name**
`GetProfileId`	`profileInfoOperation`
`GetProfile`	`profileInfoOperation`
`LoginUser`	`loginOperation`
`LogoutUser`	`logoutOperation`
`UpdateUser`	`xmlProfileOwnerOperation`
`CreateUser`	`createOperation`
`SetContactInfo`	`profileOwnerOperation`
`SetPassword`	`profileOwnerOperation`
`SetLocale`	`profileOwnerOperation`
`CanClientEncryptPasswords`	`loginOperation`
`GetPasswordHashKey`	`loginOperation`
`GetPasswordHashAlgorithm`	`loginOperation`
`RecommendContent`	`targetingOperation`
`ContentViewed`	`messagingOperation`
`ContentConsumed`	`messagingOperation`

Using Client-Side Password Encryption

Standard practice for most applications that transmit passwords is to use a secure protocol such as HTTPS, which handles password encryption internally. The `LoginUser` Web service, however, includes an option that permits a client-side application to encrypt passwords and then log in a user without using a secure protocol. **Important:** ATG strongly recommends using a secure protocol for all operations that transmit confidential information, including logins. The client-side password encryption feature built into the `LoginUser` service exists as an option for customers with unusual login requirements, and it is not recommended for general use.

To use this feature, client applications must follow very specific ATG-defined rules for encrypting cleartext passwords. If a password is encrypted in a way the ATG server does not understand, the login authentication will fail. Note also that the rules for encryption vary according to the password hasher your application uses; for example, the `MD5PasswordHasher` hashes passwords in a different way from the `SaltedDigestPasswordHasher`.

Not all password hashers permit the client-side encryption of passwords. Applications whose password hashers do not provide a temporary encryption key to be used for logins cannot use client-side encryption. This temporary encryption key is needed so that the client does not send a password string that is identical to the stored password in the database. The CanClientEncryptPasswords Web service determines if the current application allows client-side encryption. Currently, only the **MD5PasswordHasher** has well-defined rules for client-side encryption, and it is the only password hasher supported for this feature.

The process of passing an encrypted password from a client application to the `LoginUser` Web service is called a **password encryption conversation**. A restriction of this feature is that the entire password encryption conversation must be completed within a certain time limit, defined by the `maxAuthenticationWait` property of the `ProfileServices` component. This time limit starts the moment a client calls `GetPasswordHashKey` and ends the moment that same client calls `LoginUser`.

The following procedure describes how to have your application use the client-side password encryption feature to log in a user. Note that the procedure assumes your application uses the `MD5PasswordHasher`.

1. Call the `CanClientEncryptPasswords`Web service.
2. If `CanClientEncryptPasswords` returns false, use HTTPS and send the password as cleartext. Call the `LoginUser` service using the following parameters: `loginUser(login, cleartext_password, false)`.
3. If `CanClientEncryptPasswords` returns true, call the `getPasswordHashAlgorithm` service.
4. Use the returned algorithm to encrypt the user's password.
5. Call the `GetPasswordHashKey` service. Note that the conversation timer is started here. The client application has `maxAuthenticationWait` milliseconds to call the `LoginUser` service. If `LoginUser` is not called quickly enough, an error is thrown when the service is eventually called.
6. Encrypt a combination of the returned `hashKey` and the encrypted password from step 4, again using the algorithm returned from step 3.
7. Call the `LoginUser` service using the following parameters: `loginUser(login, encrypted_password_from_step_6, true)`.

Appendix A: Database Tables

This appendix describes the default database schema used by the Personalization module and the Scenarios module. The following sections describe the database tables:

Personalization Module Database Schema

The Personalization module's database schema includes the following types of tables:

User Data Tables

The Personalization module uses the following tables to store user data. Note that the tables shown, which have the prefix dps, apply to external profiles and are referenced by the ProfileAdapterRepository. A parallel set of tables exists for internal users with the prefix dpi, for example dpi_user. These tables are referenced by the InternalProfileRepository.

- dps_user
- dps_contact_info
- dps_user_address
- dps_other_addr
- dps_credit_card
- dps_usr_creditcard

dps_user

This table contains information associated with a Personalization module user.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the user.	
`login`	WVARCHAR(20)	NOT NULL UNIQUE
	The user's login name.	
`auto_login`	NUMERIC(1)	NULL CHECK `(auto_login in (0,1))`
	Determines whether to perform autologin for this user. This value is set by the `autoLoginPropertyName` property of the `/atg/userprofiling/PropertyManager` component.	
`password`	VARCHAR(35)	NULL
	The user's password. Note that this field must be at least 35 characters long if the Personalization module stores a hash of the password and not the actual value.	
`member`	NUMERIC(3)	NULL CHECK `(member in (0,1))`
	Whether or not the user is a member.	
`first_name`	WVARCHAR(40)	NULL
	The user's first name.	
`middle_name`	WVARCHAR(40)	NULL
	The user's middle name.	
`last_name`	WVARCHAR(40)	NULL
	The user's last name.	
`user_type`	INT	NULL
	The user's type, either (1) investor, (2) broker, or (3) guest.	
`locale`	INT	NULL
	The user's locale.	
`lastactivity_date`	TIMESTAMP	NULL
	The time the user last accessed the Web site.	

registration_date	TIMESTAMP	NULL
	The time the user registered at the Web site.	
email	WVARCHAR(40)	NULL
	The user's e-mail address.	
email_status	INT	NULL
	The status of the user's e-mail address, for example whether valid or invalid.	
receive_email	INT	NULL
	Determines whether or not the user is to receive e-mails.	
gender	INT	NULL
	The user's gender.	
date_of_birth	TIMESTAMP	NULL
	The user's date of birth.	
securityStatus	INT	NULL
	Indicates how a user was assigned to this site, either (0) ANONYMOUS, (1) URL-PARAM, (2) AUTO-SIGNIN, (3) HTTP-BASIC-AUTH, (4) EXPLICIT-SIGNIN, (5) SECURE_SIGNIN, or (6) CERTIFICATE.	

dps_contact_info

This table contains information about a user's contact info. There can be multiple contact infos for a single user, and all this information is stored in this table.

Column	**Data Type**	**Constraint**
id	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with the contact information.	
user_id	VARCHAR(40)	NULL
	The unique identifier associated with the user who owns the contact information.	
prefix	WVARCHAR(40)	NULL
	The name prefix (for example, a title) of this user.	
first_name	WVARCHAR(40)	NULL

	The first name of this user.	
`middle_name`	WVARCHAR(40)	NULL
	The middle name of this user.	
`last_name`	WVARCHAR(40)	NULL
	The last name of this user.	
`suffix`	WVARCHAR(40)	NULL
	The name suffix of this user.	
`job_title`	WVARCHAR(100)	NULL
	The job title of the user at this address.	
`company_name`	WVARCHAR(40)	NULL
	The company name of the user at this address.	
`address1`	WVARCHAR(50)	NULL
	The street and number of this address.	
`address2`	WVARCHAR(50)	NULL
	The street and number of this address.	
`address3`	WVARCHAR(50)	NULL
	The street and number of this address.	
`city`	WVARCHAR(30)	NULL
	The city of this address.	
`state`	WVARCHAR(20)	NULL
	The state of this address.	
`postal_code`	WVARCHAR(10)	NULL
	The postal code of this address.	
`county`	WVARCHAR(40)	NULL
	The county of this address.	
`country`	WVARCHAR(40)	NULL
	The country of this address.	
`phone_number`	WVARCHAR(15)	NULL
	The phone number of the user at this address.	

`fax_number`	WVARCHAR(15)	NULL
	The fax number of the user at this address.	

dps_user_address

This table contains information about a user's address.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the user. References `dps_user(id)`.	
`home_addr_id`	VARCHAR(40)	NULL
	The unique identifier associated with the user's home address.	
`billing_addr_id`	VARCHAR(40)	NULL
	The unique identifier associated with the user's billing address.	
`shipping_addr_id`	VARCHAR(40)	NULL
	The unique identifier associated with the user's shipping address.	

dps_other_addr

This table contains addresses used by profiles. ATG Commerce applications use this table to store a list of addresses in the table `dps_contact_info`. The Personalization module can also use this table to set up a one-to-many relationship by adding a table similar to the table `dps_contact_info` and modifying the `userProfile.xml` template.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the user who owns the address. References `dps_user(id)`.	
`tag`	WVARCHAR(42)	NOT NULL
(primary key)	The type of address, for example work or home.	
`address_id`	VARCHAR(40)	NOT NULL
	The unique identifier associated with the address.	

dps_credit_card

(External profiles only.) This table contains information about a user's credit card. There can be multiple contact credit cards for a single user.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with this credit card information.	
`credit_card_number`	VARCHAR(40)	NULL
	The credit card number.	
`credit_card_type`	VARCHAR(40)	NULL
	The type of credit card (MasterCard, Visa, etc.)	
`expiration_month`	VARCHAR(20)	NULL
	The month the credit card expires.	
`exp_day_of_month`	VARCHAR(20)	NULL
	The day of the month the credit card expires.	
`expiration_year`	VARCHAR(20)	NULL
	The year the credit card expires.	
`billing_addr`	VARCHAR(40)	NULL
	The billing address of the credit card.	

dps_usr_creditcard

(External profiles only.) This table contains credit card information associated with a user.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the user.	
`tag`	WVARCHAR(42)	NOT NULL
(primary key)	The type of credit card, for example MasterCard, Visa, etc.	
`credit_card_id`	VARCHAR(40)	NOT NULL

	The unique identifier associated with the credit card. REFERENCES `dps_credit_card(id)`.

User Directory Tables

The Personalization module uses the following tables to store information about the organizations and roles that make up a user directory. The tables shown here correspond to external users. A parallel set of tables, with the prefix dpi, exists for internal users.

- `dps_child_folder`
- `dps_folder`
- `dps_organization`
- `dps_org_ancestors`
- `dps_org_chldorg`
- `dps_org_role`
- `dps_relativerole`
- `dps_role`
- `dps_rolefold_chld`
- `dps_role_rel_org`
- `dps_user_org`
- `dps_user_org_anc`
- `dps_user_roles`

dps_child_folder

This table stores a list of a role folder's child folders.

Column	Data Type	Constraint
`folder_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier for this role folder. References `dps_folder(folder_id)`.	
`child_folder_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier for the child folder. References `dps_folder(folder_id)`.	

dps_folder

This table stores information that defines a folder in the user directory.

Column	Data Type	Constraint
`folder_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier for this folder.	
`type`	INT	NOT NULL
	Identifies the type of folder. Values can be as follows: `2000` (`genericFolder`), `2001` (`roleFolder`), or `2002` (`orgFolder`)	
`name`	WVARCHAR(254)	NOT NULL
	The user-specified name for this folder.	
`parent`	VARCHAR(40)	NULL
	The ID of this folder's parent folder. References `dps_folder(folder_id)`.	
`description`	WVARCHAR(254)	NULL
	The user-specified description for this folder.	

dps_organization

This table stores information that defines a user directory organization.

Column	Data Type	Constraint
`org_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier for this organization.	
`name`	WVARCHAR(254)	NOT NULL
	The user-specified name for this organization.	
`description`	WVARCHAR(254)	NULL
	The user-specified description for this organization.	
`parent_org`	VARCHAR(40)	NULL
	The ID of this organization's parent organization. References `dps_organization(org_id)`.	

dps_org_ancestors

This table stores a list of an organization's ancestor organizations.

Column	Data Type	Constraint
`org_Id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of this organization. References `dps_organization(org_id)`.	
`sequence_num`	INT	NOT NULL
(primary key)	The index of this ancestor organization in the list.	
`anc_org`	VARCHAR(40)	NOT NULL
	The unique identifier of the ancestor organization. References `dps_organization(org_id)`	

dps_org_chldorg

This table stores a list of an organization's child organizations.

Column	Data Type	Constraint
`org_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier for this organization. References `dps_organization(org_id)`.	
`child_org_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier for the child organization. References `dps_organization(org_id)`.	

dps_org_role

This table stores a list of roles assigned to an organization.

Column	Data Type	Constraint
`org_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier for this organization. References `dps_organization(org_id)`.	
`atg_role`	VARCHAR(40)	NOT NULL

(primary key)	The unique identifier of the role. References `dps_role(role_id)`.	

dps_relativerole

This table stores information that defines an organizational role.

Column	Data Type	Constraint
`role_id`	VARCHAR(40)	NOT NULL
(primary key)	The ID of this organizational role. References `dps_role(role_id)`.	
`dps_function`	WVARCHAR(40)	NOT NULL
	The value of the `function` property of this organizational role.	
`relative_to`	VARCHAR(40)	NOT NULL
	The organization to which this role is relative. References `dps_organization(org_id)`.	

dps_role

This table stores information that defines a global or organizational role.

Column	Data Type	Constraint
`role_id`	VARCHAR(40)	NOT NULL
(primary key)	The ID of this role.	
`type`	INT	NOT NULL
	The type of role. The value can be 2000 (global role) or 2001 (organizational role).	
`version`	INT	NOT NULL
	The revision number for this role.	
`name`	WVARCHAR(254)	NOT NULL)
	The user-specified name for this role.	
`description`	WVARCHAR(254)	NULL
	The user-specified description for this role.	

dps_rolefold_chld

This table stores a list of the global roles in a role folder.

Column	Data Type	Constraint
`rolefold_id`	VARCHAR(40)	NOT NULL
(primary key)	The ID of this role folder. References `dps_folder(folder_id)`.	
`role_id`	VARCHAR(40	NOT NULL
(primary key)	The ID of this role. References `dps_role(role_id)`.	

dps_role_rel_org

This table stores a list of the organizational roles assigned to an organization.

Column	Data Type	Constraint
`organization`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier for this organization. References `dps_organization(org_id)`.	
`sequence_num`	INT	NOT NULL
(primary key)	The index of this organizational role in the list.	
`role_id`	VARCHAR(40)	NOT NULL
	The unique identifier for this organizational role. References `dps_role(role_id)`.	

dps_user_org

This table stores a list of the users who are assigned to an organization.

Column	Data Type	Constraint
`organization`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier for this organization. References `dps_organization(org_id)`.	
`user_id`	VARCHAR(40)	NOT NULL

(primary key)	The unique identifier for the user assigned to the organization. References `dps_user(id)`.

dps_user_org_anc

This table stores a list of a user's ancestor organizations.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of this user. References `dps_user(id)`.	
`sequence_num`	INT	NOT NULL
(primary key)	The index of this ancestor organization in the user's list.	
`anc_org`	VARCHAR(40)	NOT NULL
	The unique identifier of the ancestor organization. References `dps_organization(org_id)`.	

dps_user_roles

This table extends the user profile to include references to roles assigned to this user.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier for this user. References `dps_user(id)`.	
`atg_role`	VARCHAR(40)	NOT NULL
(primary key)	A role assigned to this user. References `dps_role(role_id)`.	

Logging and Reporting Tables

The Personalization module uses the following tables to store logging and reporting information. Note that these tables exist for use with external user profiles only.

- `dps_con_req`
- `dps_con_req_sum`
- `dps_group`
- `dps_log_id`

- `dps_pgrp_con_sum`
- `dps_pgrp_req_sum`
- `dps_request`
- `dps_reqname_sum`
- `dps_session_sum`
- `dps_user_event`
- `dps_user_event_sum`
- `dps_event_type`

dps_con_req

This table contains reporting information about requests for content.

Column	Data Type	Constraint
`id`	NUMERIC(19)	NOT NULL
(primary key)	The unique identifier associated with this content request.	
`timestamp`	TIMESTAMP	NOT NULL
	The date the content is requested.	
`requestid`	NUMERIC(19)	NULL
	The ID associated with the session in which the request occurs.	
`contentid`	VARCHAR(255)	NOT NULL
	The path name of the requested content.	

dps_con_req_sum

This table contains reporting information about viewed content.

Column	Data Type	Constraint
`contentid`	VARCHAR(255)	NOT NULL
	The path of the requested content item.	
`member`	NUMERIC(1)	NOT NULL CHECK `(member in (0,1))`

	Indicates whether or not the user who will receive the content is a member.	
`summarycount`	INT	NOTNULL
	The number of requests for this content item.	
`fromtime`	TIMESTAMP	NOT NULL
	The beginning of the time span that is summarized.	
`totime`	TIMESTAMP	NOT NULL
	The end of the time span that is summarized.	

dps_group

This table contains information about reporting groups.

Column	**Data Type**	**Constraint**
`id`	INT	NOT NULL
(primary key)	The unique identifier of the group used in reporting.	
`name`	WVARCHAR(64)	NOT NULL UNIQUE
	The name of the group used in reporting.	

dps_log_id

This table provides the counters for the primary keys for each logging table.

Column	**Data Type**	**Constraint**
`tablename`	VARCHAR(30)	NOT NULL
(primary key)	The name of the reporting table.	
`nextid`	NUMERIC(19)	NOT NULL
	The next ID the logging table should use for a new row.	

dps_pgrp_con_sum

This table contains a record of summarized profile groups and the content they accessed in a content group.

Column	Data Type	Constraint
groupname	WVARCHAR(64)	NOT NULL
	The name of the profile group.	
contentname	WVARCHAR(64)	NOT NULL
	The name of the content group	
summarycount	INT	NOT NULL
	The number of times between the fromtime and totime records that the members of the named profile group accessed the content in the named content group.	
fromtime	TIMESTAMP	NOT NULL
	The beginning of the time span that is summarized.	
totime	TIMESTAMP	NOT NULL
	The end of the time span that is summarized.	

dps_pgrp_req_sum

This table contains a record of summarized profile groups and their requests for content from a content group.

Column	Data Type	Constraint
groupname	WVARCHAR(64)	NOT NULL
	The name of the profile group.	
contentname	WVARCHAR(255)	NOT NULL
	The name of the content item.	
summarycount	INT	NOT NULL
	The number of times between the fromtime and totime records that the members of the named profile group accessed the content in the named content group.	
fromtime	TIMESTAMP	NOT NULL
	The beginning of the time span that is summarized.	
totime	TIMESTAMP	NOT NULL
	The end of the time span that is summarized.	

dps_request

This table contains all of the request log entries.

Column	Data Type	Constraint
`id`	NUMERIC(19)	NOT NULL
(primary key)	The unique identifier associated with the request item.	
`timestamp`	TIMESTAMP	NOT NULL
	The date on which the request was made.	
`sessionid`	VARCHAR(32)	NULL
	The ID of the session that made the request.	
`name`	VARCHAR(255)	NOT NULL
	The name of the user who made the request.	
`member`	NUMERIC(1)	NOT NULL CHECK (`member in (0,1)`)
	Indicates whether a member or a guest made the request that is logged.	

dps_reqname_sum

This table contains records of summarized request names.

Column	Data Type	Constraint
`name`	VARCHAR(255)	NOT NULL
	The name of the request.	
`member`	NUMERIC(1)	NOT NULL CHECK (`member in (0,1)`)
	Indicates whether the user is a member or a guest.	
`summarycount`	INT	NOT NULL
	The number of times between the `fromtime` and `totime` records that the members of the named profile group accessed the content in the named content group.	
`fromtime`	TIMESTAMP	NOT NULL

	The beginning of the time span that is summarized.	
totime	TIMESTAMP	NOT NULL
	The end of the time span that is summarized.	

dps_session_sum

This table contains records of summarized sessions.

Column	Data Type	Constraint
`sessionid`	VARCHAR(32)	NULL
	The session ID.	
`member`	NUMERIC(1)	NOT NULL CHECK `(member in (0,1))`
	The value of the member attribute of the session profile.	
`summarycount`	INT	NOT NULL
	The total number of events that occurred during the session.	
`fromtime`	TIMESTAMP	NOT NULL
	The beginning of the time span that is summarized.	
`totime`	TIMESTAMP	NOT NULL
	The end of the time span that is summarized.	

dps_user_event

This table contains a record of any user event.

Column	Data Type	Constraint
`id`	NUMERIC(19)	NOT NULL
(primary key)	The primary key for this logged event.	
`timestamp`	TIMESTAMP	NOT NULL
	The time that this event occurred.	
`sessionid`	VARCHAR(32)	NULL
	The ID associated with the session that triggered this event.	

eventtype	INT	NOT NULL
	The type of event, for example, login, logout, registration, content view, purchase, etc. References dps_event_type(id).	
profileid	VARCHAR(25)	NULL
	The ID associated with the user who triggered the event.	
member	NUMERIC(1)	NOT NULL CHECK (member in (0,1))
	Whether or not the user is a member or a guest.	

dps_user_event_sum

This table contains records of summarized user event types.

Column	Data Type	Constraint
eventtype	INT	NOT NULL
	The type of event that is summarized, for example, login, logout, registration, content view, purchase, etc. References dps_event-type(id).	
summarycount	INT	NOT NULL
	The number of times the event occurred.	
fromtime	TIMESTAMP	NOT NULL
	The beginning of the time span that is summarized.	
totime	TIMESTAMP	NOT NULL
	The end of the time span that is summarized.	

dps_event_type

This table contains information about the default D4-style events.

Column	Data Type	Constraint
id	INTEGER	NOT NULL
(primary key)	The ID associated with the event type.	

name	VARCHAR(32)	NOT NULL UNIQUE
	The name of the event.	

Targeted E-mail Tables

The Personalization module uses the following tables to store information about targeted e-mail. The tables shown here correspond to external users. A parallel set of tables, with the prefix dpi, exists for internal users.

- dps_mailing
- dps_mail_server
- dps_mail_batch
- dps_user_mailing
- dps_email_address

dps_mailing

This table contains information about any targeted e-mail campaigns that are sent to users.

Column	**Data Type**	**Constraint**
id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with this mailing campaign.	
name	WVARCHAR(80)	NULL
	The name of the mailing campaign.	
subject	WVARCHAR(80)	NULL
	The subject line of the e-mail.	
uniq_server_id	VARCHAR(255)	NULL
	The unique ID of the ATG server (IP address: DRP port) that sent out the e-mail.	
from_address	WVARCHAR(255)	NULL
	The value of the “From” header in the e-mail.	
replyto	WVARCHAR(255)	NULL
	The value of the “Reply To” header in the e-mail.	
template_url	VARCHAR(255)	NULL

	The URL of the targeted e-mail template that is used for the e-mails in this campaign.	
`alt_template_URL`	VARCHAR(255)	NULL
	The URL of a template containing an alternative text version of an HTML message.	
`batch_exec_id`	VARCHAR(40)	NULL
	An ID that distinguishes all mailings created from the same invocation of a `SendEmail` scenario action. See Identifying Mailings Sent by a Single SendEmail Action.	
`cc`	LONG VARCHAR	NULL
	The value of the “CC” header in this e-mail.	
`bcc`	LONG VARCHAR	NULL
	The value of the “BCC” header in this e-mail.	
`send_as_html`	INT	NULL
	Determines whether or not the e-mail was sent in HTML format.	
`send_as_text`	INT	NULL
	Determines whether or not the e-mail was sent in text format.	
`params`	LONG VARBINARY	NULL
	Serialized map of template parameters used when rendering the e-mail template for a specific campaign. This is the value of the `templateParameters` property of the `atg.userprofiling.TemplateEmailInfo` object.	
`start_time`	TIMESTAMP	NULL
	The time at which the SMTP server started sending e-mails.	
`end_time`	TIMESTAMP	NULL
	The time at which the SMTP server stopped sending e-mails.	
`status`	INT	NULL
	The status of the e-mail campaign, whether PENDING, INPROGRESS, CANCELED, COMPLETE, or FAILED.	
`num_profiles`	INT	NULL
	The total number of profiles to which this e-mail was sent.	
`num_sent`	INT	NULL

	The total number of e-mails sent.	
`num_bounces`	INT	NULL
	The total number of e-mails that bounced back to the server.	
`num_errors`	INT	NULL
	The number of errors that occurred while sending the e-mails.	
`num_skipped`	INT	NULL
	The number of recipients to whom the mailing could not be sent (for example, because their `emailStatus` profile property was set to `invalid`).	
`fill_from_templ`	NUMERIC(1)	NULL
	The value of the `fillFromTemplate` property of the `atg.userprofiling.TemplateEmailInfo` object. This value is true if information about the e-mail is extracted from the parameter values set in the e-mail template.	
`is_batched`	TINYINT	NULL
	Identifies this mailing as a distributed mailing.	
`batch_size`	INT	NULL
	The number of messages in this batch.	

dps_mail_server

This table is used to identify the servers participating in a distributed mailing.

Column	Data Type	Constraint
`uniq_server_id`	VARCHAR(254)	NOT NULL
(primary key)	The unique identifier associated with each participating server.	
`last_updated`	TIMESTAMP	NULL
	The timestamp set by this server's `batchEmailPeriodicService`.	

dps_mail_batch

This table is used by the distributed mailing feature.

<table>
<tr><th>Column</th><th>Data Type</th><th>Constraint</th></tr>
<tr><td rowspan="2">mailing_id
(primary key)</td><td>VARCHAR(40)</td><td>NOT NULL</td></tr>
<tr><td colspan="2">The unique identifier associated with the batch. References dps_mailing(id).</td></tr>
<tr><td rowspan="2">start_idx
(primary key)</td><td>INT</td><td>NOT NULL</td></tr>
<tr><td colspan="2">An index that identifies the first profile in this batch.</td></tr>
<tr><td rowspan="2">uniq_server_id</td><td>VARCHAR(254)</td><td>NULL</td></tr>
<tr><td colspan="2">The unique ID of the server (IP address: DRP port) that claimed the batch.</td></tr>
<tr><td rowspan="2">start_time</td><td>TIMESTAMP</td><td>NULL</td></tr>
<tr><td colspan="2">The time at which the server started sending the e-mails in this batch.</td></tr>
<tr><td rowspan="2">end_time</td><td>TIMESTAMP</td><td>NULL</td></tr>
<tr><td colspan="2">The time at which the server stopped sending the e-mails in this batch.</td></tr>
<tr><td rowspan="2">status</td><td>INT</td><td>NULL</td></tr>
<tr><td colspan="2">The status of the batch mailing, whether PENDING, INPROGRESS, CANCELED, COMPLETE, or FAILED.</td></tr>
<tr><td rowspan="2">num_profiles</td><td>INT</td><td>NULL</td></tr>
<tr><td colspan="2">The total number of profiles to which this mailing was sent.</td></tr>
<tr><td rowspan="2">num_sent</td><td>INT</td><td>NULL</td></tr>
<tr><td colspan="2">The total number of e-mails sent.</td></tr>
<tr><td rowspan="2">num_bounces</td><td>INT</td><td>NULL</td></tr>
<tr><td colspan="2">The total number of e-mails that bounced back to the server.</td></tr>
<tr><td rowspan="2">num_errors</td><td>INT</td><td>NULL</td></tr>
<tr><td colspan="2">The number of errors that occurred while sending the e-mails.</td></tr>
<tr><td rowspan="2">num_skipped</td><td>INT</td><td>NULL</td></tr>
<tr><td colspan="2">The number of recipients to whom the mailing could not be sent (because, for example, their emailStatus profile property was set to invalid).</td></tr>
</table>

`is_summarized`	TINYINT	NULL
	Indicates whether the counts for this batch have been merged back into the parent mailing.	

dps_user_mailing

This table contains information about users who are receiving e-mail as part of an e-mail campaign.

Column	**Data Type**	**Constraint**
`mailing_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the e-mail campaign. References `dps_mailing(id)`.	
`user_id`	VARCHAR(40)	NOT NULL
	The ID of a user who received an e-mail from the e-mail campaign. References `dps_user(id)`.	
`idx`	INT	NOT NULL
(primary key)	An index used to order the users in this campaign.	

dps_email_address

This table stores information that allows you to send e-mail to specific e-mail addresses. A profile is not required.

Column	**Data Type**	**Constraint**
`mailing_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the e-mail campaign. References `dps_mailing(id)`.	
`email_address`	VARCHAR(255)	NOT NULL
	The user's e-mail address.	
`idx`	INT	NOT NULL
(primary key)	An index used to order the users in this campaign.	

Personalization Module Scenario Tables

The Personalization module uses the following tables to store user-related information about scenarios. The tables shown here, with the prefix dps, apply to external users. A parallel set of tables exists with the prefix dpi (for example, `dpi_user_scenario`) for internal users.

- `dps_scenario_value`
- `dps_user_scenario`
- `dps_user_slot`

dps_scenario_value

This table contains a map of values that can be used in scenarios to persist scenario-related per-user information. For example, one segment in a scenario may store a value under a key in this map, and another segment (or scenario) may look up or change that value. These values are similar to scenario variables but are persisted across scenario segments. Both the keys and the values in this map are Strings.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the user.	
`tag`	VARCHAR(42)	NOT NULL
(primary key)	The key for a value in the scenario map.	
`scenario_value`	VARCHAR(100)	NULL
	The value for this key.	

dps_user_scenario

This table associates a user with individual scenario instances.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL
	The unique identifier associated with the user. References `dps_user(id)`.	
`ind_scenario_id`	VARCHAR(25)	NOT NULL
(primary key)	The ID associated with the scenario instance. References `dss_ind_scenario(id)`.	

dps_user_slot

This table stores information that associates a persistent slot with a user in the profile repository. (Specifically, the data corresponds to the value of the slotInstances property in the user profile.)

Column	Data Type	Constraint
id *(primary key)*	VARCHAR(40)	NOT NULL
	The unique identifier associated with the user. References dps_user(id).	
profile_slot_id *(primary key)*	VARCHAR(25)	NOT NULL
	The slot associated with the user. References dss_profile_slot(id).	

Scenarios Module Database Schema

The database schema for the Scenarios module includes the following types of tables:

Collective and Individual Scenario Instance Tables

Scenario Info Tables

Template Info Tables

Scenario Transition and Deletion Tables

Slot Tables

Server ID Tables

Scenario Xref Tables

Scenario Migration Tables

Event Message Tables

Business Process Tracking Tables

Collective and Individual Scenario Instance Tables

- dss_coll_scenario
- dss_ind_scenario

- dss_scenario_strs
- dss_scenario_bools
- dss_scenario_longs
- dss_scenario_dbls
- dss_scenario_dates

dss_coll_scenario

This table contains information about collective scenario instances. A collective scenario instance maintains the state of any scenario segment that applies to all users (in other words, any that are not specific to individual users). When the Scenarios module begins to execute a scenario segment, it creates a single collective scenario instance, called the root instance, to handle the execution. It may create other collective instances later during execution of that segment, for example to handle recurrent timer events such as a scenario element that occurs every Monday at 2 PM.

Column	Data Type	Constraint
id	VARCHAR(25)	NOT NULL
(primary key)	The ID associated with this collective scenario instance.	
scenario_name	WVARCHAR(255)	NULL
	The name of the scenario.	
modification_time	NUMERIC(19)	NULL
	The last time the scenario was modified.	
segment_name	WVARCHAR(255)	NULL
	The name of the scenario segment.	
creator_id	VARCHAR(25)	NULL
	The unique identifier for the scenario instance that created this collective scenario instance. This value is null if this is the root instance.	
state	VARCHAR(16)	NULL
	The state of this collective scenario instance in the scenario state machine.	

dss_ind_scenario

This table contains information about individual scenario instances. Each scenario instance maintains the state associated with a particular scenario segment that is currently processing a user. Individual scenario instances are formed from collective scenario instances, typically when user-specific events occur, for example when a user visits a particular page.

Column	Data Type	Constraint
`id`	VARCHAR(25)	NOT NULL
(primary key)	The ID associated with this individual scenario instance.	
`scenario_name`	WVARCHAR(255)	NULL
	The name of the scenario.	
`modification_time`	NUMERIC(19)	NULL
	The last time the scenario was modified.	
`segment_name`	WVARCHAR(255)	NULL
	The name of the scenario segment.	
`creator_id`	VARCHAR(25)	NULL
	The ID associated with the scenario that created this individual scenario instance.	
`state`	VARCHAR(16)	NULL
	The state of this individual scenario instance in the scenario state machine.	
`user_id`	VARCHAR(25)	NOT NULL
	The ID associated with the user to whom this scenario instance belongs.	

dss_scenario_strs

This table contains a map of any String variables that have been set for a given scenario instance.

Column	Data Type	Constraint
`id`	VARCHAR(25)	NOT NULL
(primary key)	The ID associated with the individual scenario instance. References `dss_ind_scenario (id)`.	
`tag`	WVARCHAR(25)	NOT NULL
(primary key)	The name of the scenario variable.	
`context_str`	VARCHAR(255)	NULL
	The string value of the scenario variable.	

dss_scenario_bools

This table contains a map of any Boolean scenario variables that have been set for a given scenario instance.

Column	Data Type	Constraint
`id`	VARCHAR(25)	NOT NULL
(primary key)	The ID associated with the individual scenario instance. References `dss_ind_scenario (id)`.	
`tag`	WVARCHAR(25)	NOT NULL
(primary key)	The name of the scenario variable.	
`context_bool`	NUMERIC(1)	NULL
	The Boolean value of the scenario variable.	

dss_scenario_longs

This table contains a map of any Long scenario variables that have been set for a given scenario instance.

Column	Data Type	Constraint
`id`	VARCHAR(25)	NOT NULL
(primary key)	The ID associated with the individual scenario instance. References `dss_ind_scenario (id)`.	
`tag`	WVARCHAR(25)	NOT NULL
(primary key)	The name of the scenario variable.	
`context_long`	NUMERIC(19)	NULL
	The long value of the scenario variable.	

dss_scenario_dbls

This table contains a map of any Double scenario variables that have been set for a given scenario instance.

Column	Data Type	Constraint
`id`	VARCHAR(25)	NOT NULL
(primary key)	The ID associated with the individual scenario instance. References `dss_ind_scenario (id)`.	
`tag`	WVARCHAR(25)	NOT NULL
(primary key)	The name of the scenario variable.	
`context_dbl`	NUMERIC(15, 4)	NULL
	The double value of the scenario variable.	

dss_scenario_dates

This table contains a map of any Date scenario variables that have been set for a given scenario instance.

Column	Data Type	Constraint
`id`	VARCHAR(25)	NOT NULL
(primary key)	The ID associated with the individual scenario instance. References `dss_ind_scenario (id)`.	
`tag`	WVARCHAR(25)	NOT NULL
(primary key)	The name of the scenario variable.	
`context_date`	TIMESTAMP	NULL
	The date value of the scenario variable.	

Scenario Info Tables

This section describes the `dss_scenario_info` table.

dss_scenario_info

This table contains information about scenarios. A entry in this table corresponds to a single SDL (Scenario Definition Language) file in the scenario registry.

Column	Data Type	Constraint
`id` *(primary key)*	VARCHAR(25)	NOT NULL

	The ID associated with this scenario information.	
`scenario_name`	WVARCHAR(255)	NULL
	The name of the scenario.	
`scenario_status`	INT	NULL
	The current scenario status, either (1) disabled, or (2) running.	
`modification_time`	NUMERIC(19)	NULL
	The last time this scenario was modified.	
`creation_time`	NUMERIC(19)	NULL
	The time at which this scenario was created.	
`author`	VARCHAR(25)	NULL
	The login name of the ACC user who created this scenario.	
`last_modified_by`	VARCHAR(25)	NULL
	The login name of the ACC user who last edited this scenario.	
`sdl`	LONG VARBINARY	NULL
	The serialized contents of the scenario's SDL file.	

Template Info Tables

This section describes the `dss_template_info` table.

dss_template_info

This table stores information about scenario templates.

Column	Data Type	Constraint
`id`	VARCHAR(25)	NOT NULL
(primary key)	The unique identifier of this scenario template.	
`template_name`	WVARCHAR(255)	NULL
	The user-specified name for the template.	
`modification_time`	NUMERIC(19)	NULL
	The time when the template was last modified.	
`creation_time`	NUMERIC(19)	NULL

	The time when the template was originally created.	
author	VARCHAR(25)	NULL
	The login name of the person who created the template.	
last_modified_by	VARCHAR(25)	NULL
	The login name of the person who last edited the template.	
sdl	LONG VARBINARY	NULL
	The scenario definition file that represents this template.	

Scenario Transition and Deletion Tables

- dss_coll_trans
- dss_ind_trans
- dss_deletion
- dss_del_seg_name

dss_coll_trans

This table contains information about pending collective scenario transitions. A collective transition is a transition that a collective scenario instance takes from one state in a scenario state machine to another. When the Scenarios module cannot complete such a transition in a single transaction (for example, when the transition initiates an e-mail campaign to a large number of users), it stores the pending transition in the database and completes it over multiple transactions.

Column	**Data Type**	**Constraint**
id	VARCHAR(25)	NOT NULL
(Primary key)	The ID associated with the pending collective transition.	
scenario_name	WVARCHAR(255)	NULL
	The name of the scenario.	
modification_time	NUMERIC(19)	NULL
	The last time this scenario was modified.	
server_id	VARCHAR(40)	NULL
	The ID of the server handling the transition. The value is a combination of a server name and drpPort number.	
message_bean	LONG VARBINARY	NULL

	The serialized JMS message bean that caused this scenario transition to occur.	
`event_type`	VARCHAR(255)	NULL
	The type of the JMS event that caused this scenario transition to occur.	
`segment_name`	WVARCHAR(255)	NULL
	The name of the scenario segment.	
`state`	VARCHAR(16)	NULL
	The state in the scenario state machine of the collective scenario instance at the start of the transition.	
`coll_scenario_id`	VARCHAR(25)	NULL
	The unique identifier for the collective scenario instance that is taking the transition. References `dss_coll_scenario(id)`.	
`step`	INT	NOT NULL
	Indicates where the transition is currently. The value can be actions1 (1), actions2 (2), nextState1 (3), or nextState2 (4). This value changes over multiple transactions as transition actions execute and move to the next scenario state.	
`current_count`	INT	NULL
	For a given transition step, indicates where the transition is currently. For example, as multiple transition actions are executed, this value changes to indicate the index of the current action.	
`last_query_id`	VARCHAR(25)	NULL
	The repository ID last used in a repository query. When a transition action is executed, for example, it is executed on a batch of individual scenario instances at a time. This value is used to keep track of the location in the process of cycling through the scenario instances.	

dss_ind_trans

This table contains information about pending individual scenario transitions. An individual transition is a transition taken from one scenario state machine state to another, by all the individual scenario instances in the originating state. The Scenarios module cannot typically complete such a transition in a single transaction because of the potentially large number of scenario instances involved, so it stores the pending transition in the database and completes it over multiple transactions.

Column	Data Type	Constraint
`id`	VARCHAR(25)	NOT NULL
(primary key)	The ID associated with the pending individual transition.	
`scenario_name`	WVARCHAR(255)	NULL
	The name of the scenario.	
`modification_time`	NUMERIC(19)	NULL
	The last time this scenario was modified.	
`server_id`	VARCHAR(40)	NULL
	The ID of the server handling the transition. The value is a combination of a server name and `drpPort` number.	
`message_bean`	LONG VARBINARY	NULL
	The serialized JMS message bean that caused this scenario transition to occur.	
`event_type`	VARCHAR(255)	NULL
	The type of the JMS event that caused this scenario transition to occur.	
`segment_name`	WVARCHAR(255)	NULL
	The name of the scenario segment.	
`state`	VARCHAR(16)	NULL
	The state in the scenario state machine of the individual scenario instances at the start of the transition.	
`last_query_id`	VARCHAR(25)	NULL
	The repository ID last used in a repository query. When a transition action is executed, for example, it is executed on a batch of individual scenario instances at a time. This value is used to keep track of the location in the process of cycling through the scenario instances.	

dss_deletion

This table contains information about pending scenario deletions. An entry is created in this table when a scenario is marked as disabled, which means that all the scenario's instances must be deleted. The Scenarios module typically cannot complete this deletion in a single transaction because of the potentially large number of scenario instances involved, so it stores the pending deletion in the database and completes the operation over multiple transactions.

Column	Data Type	Constraint
id	VARCHAR(25)	NOT NULL
(primary key)	The ID associated with the pending deletion.	
scenario_name	WVARCHAR(255)	NULL
	The name of the scenario.	
modification_time	NUMERIC(19)	NULL
	The last time this scenario was modified.	

dss_del_seg_name

This table stores a list of the names of scenario segments that are pending deletion.

Column	Data Type	Constraint
id	VARCHAR(25)	NOT NULL
(primary key)	The unique identifier for the scenario that contains the segment.	
idx	INT	NOT NULL
(primary key)	An index used to order the segments pending deletion.	
segment_name	VARCHAR(255)	NULL
	The name of the segment pending deletion.	

Slot Tables

- dss_profile_slot
- dss_slot_items
- dss_slot_priority

dss_profile_slot

This table stores information from the properties of the `ProfileSlot` item descriptor in the `userProfile.xml` file.

Column	Data Type	Constraint
`id`	VARCHAR(25)	NOT NULL
(primary key)	The unique identifier of the persistent slot.	
`slot_name`	VARCHAR(255)	NULL
	The pathname of the persistent slot.	
`item_offset`	NUMERIC(19)	NULL
	Index into the list of slot repository items.	
`user_id`	VARCHAR(25)	NOT NULL
	The ID of the user with whom this slot is associated.	

dss_slot_items

This table stores a list of repository items contained in a persistent slot. It is populated from the `slotItems` property in the `userProfile.xml` file.

Column	Data Type	Constraint
`slot_id`	VARCHAR(25)	NOT NULL REFERENCES dss_profile_slot(id)
(primary key)	The unique identifier of the persistent slot.	
`item_id`	VARCHAR(255)	NULL
	The ID of the repository item contained by the slot.	
`idx`	INT	NOT NULL
(primary key)	An index used to order the repository items in the list.	

dss_slot_priority

This table stores information about the priority of display for each repository item in a persistent slot. It is populated from the `slotItemPriorities` property in the `userProfile.xml` file.

Column	Data Type	Constraint
`slot_id`	VARCHAR(25)	NOT NULL
(primary key)	The unique identifier of the persistent slot. References `dss_profile_slot(id)`.	

`idx`	INTEGER	NOT NULL
(primary key)	An index used to order the priority entries in the list.	
`priority`	NUMERIC(19)	NOT NULL
	The priority level of each repository item in the slot.	

Server ID Tables

This section describes the `dss_server_id` table.

dss_server_id

This table stores information that the `ScenarioClusterManager` service uses to identify the process editor server in a multiple server configuration.

Column	Data Type	Constraint
`server_id`	VARCHAR(40)	NOT NULL
(primary key)	A combination of a server name and `drpPort` number that is used to uniquely identify each scenario server. Example: `mymachine.example.com:8850`.	
`server_type`	INT	NOT NULL
	The scenario server type. Possible values are 0 for the process editor server, 1 for a global server , and 2 for an individual server.	

Scenario Xref Tables

This section describes the `dss_xref` table.

dss_xref

This table stores information that the ACC uses to display scenarios according to a specific view (for example, display all scenarios that were created on the same date or that contain a given event element).

Column	Data Type	Constraint
`id`	VARCHAR(25)	NOT NULL
(primary key)	The unique identifier of the scenario.	

scenario_name	VARCHAR(255)	NULL
	The user-specified scenario name.	
reference_type	VARCHAR(30)	NULL
	The data type of the scenario characteristic to use to match against the display criteria.	
reference_target	VARCHAR(255)	NULL
	The characteristic of the scenario to match against the display criteria.	

Scenario Migration Tables

These tables are used by the Scenarios module to allow changes to be made to a running scenario without losing the status of any users progressing through it.

- dss_scen_mig_info
- dss_scen_mig_info_seg
- dss_migration
- dss_mig_seg_name

dss_scen_mig_info

This table stores migration information for each user progressing though a given scenario.

Column	Data Type	Constraint
id	VARCHAR(25)	NOT NULL
(primary key)	The unique identifier of the scenario.	
scenario_info_id	VARCHAR(25)	NOT NULL
	References dss_scenario_info(id)	
scenario_name	VARCHAR(255)	NULL
	The user-specified scenario name.	
modification_time	NUMERIC(19)	NULL
	The time the scenario was changed in the ACC.	
psm_version	INTEGER	NULL
	The version number of the state machine.	

`sdl`	LONG VARBINARY	NULL
	The SDL file for the scenario being migrated.	
`migration_status`	INTEGER	NULL
	Indicates whether the scenario migration is in progress (1) or complete (2).	

dss_scen_mig_info_seg

This table is stores information about scenario segments being migrated.

Column	**Data Type**	**Constraint**
`id`	VARCHAR(25)	NOT NULL
(primary key)	The unique identifier of the scenario. References `dss_scen_mig_info(id)`.	
`idx`	INTEGER	NOT NULL
(primary key)	An index used to order the segment names.	
`segment_name`	VARCHAR(255)	NULL
	The user-specified segment name	

dss_migration

This table is used to store information about a pending scenario migration operation that changes the modification time of all the individual scenario instances associated with an old version of a scenario (indicated by the old modification time).

Column	**Data Type**	**Constraint**
`id`	VARCHAR(25)	NOT NULL
(primary key)	The unique identifier of the scenario.	
`scenario_name`	WVARCHAR(255)	NULL
	The user-specified scenario name.	
`old_mod_time`	NUMERIC(19)	NULL
	The modification time from which the scenario is being migrated.	

`new_mod_time`	NUMERIC(19)	NULL
	The modification time to which the scenario is being migrated.	

dss_mig_seg_name

This table is used to store information about scenario segments whose modification time is being migrated.

Column	Data Type	Constraint
`id`	VARCHAR(25)	NOT NULL
(primary key)	The unique identifier of the scenario. References `dss_migration(id)`.	
`idx`	INTEGER	NOT NULL
(primary key)	An index used to order the segment names.	
`segment_name`	VARCHAR(255)	NULL
	The user-specified segment name.	

Event Message Tables

The Scenarios module uses the following tables to store event messages:

- `dss_das_event`
- `dss_audit_trail`
- `dss_dps_event`
- `dss_dps_page_visit`
- `dss_dps_view_item`
- `dss_dps_click`
- `dss_dps_referrer`
- `dss_dps_inbound`
- `dss_dps_admin_reg`
- `dss_dps_property`
- `dss_dps_admin_prop`
- `dss_dps_update`
- `dss_dps_admin_up`

dss_das_event

This table contains information used by the DAS Event SQL mapper to capture `atg.das.Startup` and `atg.das.Shutdown` messages.

Column	Data Type	Constraint
`id`	VARCHAR(32)	NOT NULL
	The ID associated with the dataset.	
`timestamp`	TIMESTAMP	NULL
	The time the event occurred.	
`sessionid`	VARCHAR(32)	NULL
	The session ID of the session generating the event.	

dss_audit_trail

This table contains information used by the DSS Audit Trail mapper to capture audit trail event messages.

Column	Data Type	Constraint
`id`	VARCHAR(32)	NOT NULL
	The ID associated with the dataset.	
`timestamp`	TIMESTAMP	NULL
	The time the event occurred.	
`label`	WVARCHAR(255)	NULL
	The audit trail label.	
`profileid`	VARCHAR(25)	NULL
	The profile ID of the user generating the event.	

dss_dps_event

This table contains information used by the DPS Event SQL mapper to capture `atg.dps.Login`, `atg.dps.Logout`, `atg.dps.Register`, `atg.dps.StartSession` and `atg.dps.EndSession` messages.

Column	Data Type	Constraint
`id`	VARCHAR(32)	NOT NULL
	The ID associated with the dataset.	
`timestamp`	TIMESTAMP	NULL
	The time the event occurred.	
`sessionid`	VARCHAR(32)	NULL
	The session ID of the session generating the event.	
`profileid`	VARCHAR(25)	NULL
	The profile ID of the user generating the event.	

dss_dps_page_visit

This table contains information used by the Page Visit SQL mapper to capture `atg.dps.PageVisit` messages.

Column	Data Type	Constraint
`id`	VARCHAR(32)	NOT NULL
	The ID associated with the dataset.	
`timestamp`	TIMESTAMP	NULL
	The time the event occurred.	
`sessionid`	VARCHAR(32)	NULL
	The session ID of the session generating the event.	
`path`	VARCHAR(255)	NULL
	The path of the page being viewed.	
`profileid`	VARCHAR(25)	NULL
	The profile ID of the user generating the event.	

dss_dps_view_item

This table is used by the View Item SQL mapper to capture `atg.dps.ViewItem` messages.

Column	Data Type	Constraint
id	VARCHAR(32)	NOT NULL
	The ID associated with the dataset.	
timestamp	TIMESTAMP	NULL
	The time the event occurred.	
sessionid	VARCHAR(32)	NULL
	The session ID of the session generating the event.	
repositoryname	WVARCHAR(255)	NULL
	The name of the repository that contains the item.	
folder	VARCHAR(255)	NULL
	The repository folder that contains the item.	
itemtype	VARCHAR(255)	NULL
	The type of the item.	
repositoryid	VARCHAR(255)	NULL
	The repository ID of the item.	
itemdescriptor	VARCHAR(255)	NULL
	The item descriptor type of the item.	
page	VARCHAR(255)	NULL
	The path of the page that contains the item.	
profileid	VARCHAR(25)	NULL
	The profile ID of the user generating the event.	

dss_dps_click

This table is used by the ClickThrough SQL mapper to capture `atg.dps.clickThrough` messages.

Column	Data Type	Constraint
id	VARCHAR(32)	NOT NULL
	The ID associated with the dataset.	
timestamp	TIMESTAMP	NULL

	The time the event occurred.	
`sessionid`	VARCHAR(32)	NULL
	The session ID of the session generating the event.	
`destinationpath`	VARCHAR(255)	NULL
	The path name of the page that contains the `dsource` parameter.	
`sourcenames`	VARCHAR(255)	NULL
	A string of all of the `sourceName` values separated by a comma.	
`sourcepath`	VARCHAR(255)	NULL
	The path name of the page that is requested when the link is clicked.	
`profileid`	VARCHAR(25)	NULL
	The profile ID of the user generating the event.	

dss_dps_referrer

This table is used by the Referrer SQL mapper to capture `atg.dps.Referrer` messages.

Column	**Data Type**	**Constraint**
`id`	VARCHAR(32)	NOT NULL
	The ID associated with the dataset.	
`timestamp`	TIMESTAMP	NULL
	The time the event occurred.	
`sessionid`	VARCHAR(32)	NULL
	The session ID of the session generating the event.	
`referrerpath`	VARCHAR(255)	NULL
	The path of the referring page relative to the Web application (or document root).	
`referrersite`	VARCHAR(255)	NULL
	The URL of the referring page minus the protocol and path.	
`referrerpage`	VARCHAR(255)	NULL

	The fully qualified URL of the referring page.	
`profileid`	VARCHAR(25)	NULL
	The profile ID of the user generating the event.	

dss_dps_inbound

This table is used by the Inbound Email SQL mapper to capture `atg.dps.InboundEmail` messages.

Column	Data Type	Constraint
`id`	VARCHAR(32)	NOT NULL
	The name of the dataset that is recording the event.	
`timestamp`	TIMESTAMP	NULL
	The time of the event.	
`messagesubject`	VARCHAR(255)	NULL
	The text in the subject line of the email.	
`originalsubject`	VARCHAR(255)	NULL
	The original subject of the email, without “Re:” if present.	
`messagefrom`	VARCHAR(64)	NULL
	The email address of the sender.	
`messageto`	VARCHAR(255)	NULL
	A comma-separated list of all of the recipients in the “To:” field of the e-mail.	
`messagecc`	VARCHAR(255)	NULL
	A comma-separated list of all of the recipients in the “Cc:” field of the e-mail.	
`messagereplyto`	VARCHAR(64)	NULL
	The email address in the “Reply-To:” header of the e-mail.	
`receiveddate`	NUMERIC(19)	NULL
	The date that the email was received. This is set to the value of the “Delivery-date:” header if it is present. Otherwise, it uses the `MimeMessage.getReceivedDate()` method. The date is represented in milliseconds since Jan 1, 1970.	

bounced	VARCHAR(6)	NULL
	True if the email was returned as undeliverable. False otherwise.	
bounceemailaddr	VARCHAR(64)	NULL
	The e-mail address of the bounced message.	
bouncereplycode	VARCHAR(10)	NULL
	The RFC 821 reply code of the bounced email.	
bounceerrormess	VARCHAR(255)	NULL
	A String property indicating why the message was bounced.	
bouncestatuscode	VARCHAR(10)	NULL
	The enhanced RFC 1893 status code of the bounced email.	

dss_dps_admin_reg

This table is used by the Admin Register SQL mapper to capture `atg.dps.AdminRegister` messages.

Column	Data Type	Constraint
id	VARCHAR(32)	NOT NULL
	The ID associated with the dataset.	
clocktime	TIMESTAMP	NULL
	The time the batch registration event occurred.	
sessionid	VARCHAR(32)	NULL
	The session ID of the session generating the batch registration event.	
adminprofileid	VARCHAR(25)	NULL
	The profile ID of the admin generating the batch registration event.	
profileid	VARCHAR(25)	NULL
	The profile IDs of the users who were registered.	

dss_dps_property

This table is used by the Profile Property Update SQL mapper to capture `atg.dps.ProfilePropertyUpdate` messages.

Column	Data Type	Constraint
`id`	VARCHAR(32)	NOT NULL
	The ID associated with the dataset.	
`clocktime`	TIMESTAMP	NULL
	The time the event occurred.	
`sessionid`	VARCHAR(32)	NULL
	The session ID of the session generating the event.	
`propertypath`	VARCHAR(254)	NULL
	The path of each property that has changed.	
`oldvalue`	VARCHAR(254)	NULL
	The old property values before they were changed.	
`newvalue`	VARCHAR(254)	NULL
	The new property values after they were changed.	
`changesign`	VARCHAR(16)	NULL
	If the property is a Comparable type, this column indicates if the value has increased or decreased. 0 indicates either “no change” or “not comparable.” A positive value indicates an increase. A negative value indicates a decrease.	
`changeamount`	NUMERIC(19, 7)	NULL
	If the property is a Number type, this column indicates the absolute value of the difference between the old value and the new value.	
`changepercentage`	NUMERIC(19, 7)	NULL
	If the property is a Number type, this column indicates the absolute value of the percent difference between the old value and the new value.	
`elementsadded`	VARCHAR(254)	NULL

	If the property is an array or Collection type, this column indicates the elements that were added as part of the profile update (elements that are members of `newvalue` but not members of `oldvalue`).	
`elementsremoved`	VARCHAR(254)	NULL
	If the property is an array or Collection type, this column indicates the elements that were removed as part of the profile update (elements that are members of `oldvalue` but not members of `newvalue`).	
`profileid`	VARCHAR(25)	NULL
	The profile ID of the user whose profile was changed.	

dss_dps_admin_prop

This table is used by the Admin Profile Property Update SQL mapper to capture `atg.dps.AdminProfilePropertyUpdate` messages.

Column	Data Type	Constraint
`id`	VARCHAR(32)	NOT NULL
	The ID associated with the dataset.	
`clocktime`	TIMESTAMP	NULL
	The time the event occurred.	
`sessionid`	VARCHAR(32)	NULL
	The session ID of the session generating the event.	
`propertypath`	VARCHAR(254)	NULL
	The path of each property that has changed.	
`oldvalue`	VARCHAR(254)	NULL
	The old property values before they were changed.	
`newvalue`	VARCHAR(254)	NULL
	The new property values after they were changed.	
`changesign`	VARCHAR(16)	NULL

	If the property is a Comparable type, this column indicates if the value has increased or decreased. 0 indicates either “no change” or “not comparable.” A positive value indicates an increase. A negative value indicates a decrease.	
`changeamount`	NUMERIC(19, 7)	NULL
	If the property is a Number type, this column indicates the absolute value of the difference between the old value and the new value.	
`changepercentage`	NUMERIC(19, 7)	NULL
	If the property is a Number type, this column indicates the absolute value of the percent difference between the old value and the new value.	
`elementsadded`	VARCHAR(254)	NULL
	If the property is an array or Collection type, this column indicates the elements that were added as part of the profile update (elements that are members of `newvalue` but not members of `oldvalue`).	
`elementsremoved`	VARCHAR(254)	NULL
	If the property is an array or Collection type, this column indicates the elements that were removed as part of the profile update (elements that are members of `oldvalue` but not members of `newvalue`).	
`adminprofileid`	VARCHAR(25)	NULL
	The profile ID of the admin who made the change.	
`profileid`	VARCHAR(25)	NULL
	The profile ID of the user whose profile was changed.	

dss_dps_update

This table is used by the Profile Update SQL mapper to capture `atg.dps.ProfileUpdate` messages.

Column	Data Type	Constraint
`id`	VARCHAR(32)	NOT NULL
	The ID associated with the dataset.	
`clocktime`	TIMESTAMP	NULL
	The time the event occurred.	

`sessionid`	VARCHAR(32)	NULL
	The session ID of the session generating the event.	
`changedproperties`	LONG VARCHAR	NULL
	The properties that were changed.	
`oldvalues`	LONG VARCHAR	NULL
	The old property values before they were changed.	
`newvalues`	LONG VARCHAR	NULL
	The new property values after they were changed.	
`profileid`	VARCHAR(25)	NULL
	The profile ID of the user who updated his or her profile.	

dss_dps_admin_up

This table is used by the Admin Profile Update SQL mapper to capture `atg.dps.AdminProfileUpdate` messages.

Column	**Data Type**	**Constraint**
`id`	VARCHAR(32)	NOT NULL
	The ID associated with the dataset.	
`clocktime`	TIMESTAMP	NULL
	The time the event occurred.	
`sessionid`	VARCHAR(32)	NULL
	The session ID of the session generating the event.	
`changedproperties`	LONG VARCHAR	NULL
	The properties that were changed.	
`oldvalues`	LONG VARCHAR	NULL
	The old property values before they were changed.	
`newvalues`	LONG VARCHAR	NULL
	The new property values after they were changed.	
`adminprofileid`	VARCHAR(25)	NULL
	The profile ID of the admin who updated the profiles.	

profileid	VARCHAR(25)	NULL
	The profile ID of the user whose profile was changed.	

Business Process Tracking Tables

The Scenarios module uses the following tables to store information about business processes:

- drpt_stage_reached
- dss_user_bpmarkers

drpt_stage_reached

This table contains information associated with a business process stage.

Column	Data Type	Constraint
id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the business process stage reached dataset.	
owner_id	VARCHAR(40)	NOT NULL
	The ID of the RepositoryItem that is assigned a marker when this stage is reached.	
process_start_time	TIMESTAMP	NOT NULL
	The time a RepositoryItem obtains a marker for a business process stage.	
event_time	TIMESTAMP	NOT NULL
	The time the stage is reached.	
bp_name	VARCHAR(225)	NOT NULL
	The business process name.	
bp_stage	VARCHAR(225)	NULL
	The business process stage name.	
is_transient	NUMERIC (1,0)	NOT NULL
	Indicates if the user who reached the business stage is logged in.	
bp_stage_sequence	INTEGER	NOT NULL
	The index of this stage in the business process.	

dss_user_bpmarkers

This table contains information associated with markers used for business process tracking purposes.

Column	Data Type	Constraint
`marker_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the marker.	
`profile_id`	VARCHAR(40)	NOT NULL
(primary key)	The ID of the Profile for that has a marker.	
`marker_key`	VARCHAR(100)	NOT NULL
	The name of the business process associated with the marker.	
`marker_value`	VARCHAR(100)	NULL
	The name of the business process stage associated with the marker.	
`marker_data`	VARCHAR(100)	NULL
	This column is not currently in use.	
`creation_date`	TIMESTAMP	NULL
	The date the business process stage is reached and the marker is assigned to the Profile.	
`version`	INTEGER	NOT NULL
	The marker repository version number. This information is managed and used internally by the repository.	
`marker_type`	INTEGER	NULL
	This column is not currently in use.	

Index