Contents

Preface .....................................................................................................................................................9

1 Introduction to the Development Environment for GlassFish Server Add-On Components .........................................................15
   GlassFish Server Modular Architecture and Add-On Components ........................................... 15
   OSGi Alliance Module Management Subsystem ....................................................................... 16
   Hundred-Kilobyte Kernel .............................................................................................................. 16
   Overview of the Development Process for an Add-On Component ..................................... 17
      Writing HK2 Components ........................................................................................................ 17
      Extending the Administration Console .................................................................................. 17
      Extending the asadmin Utility ................................................................................................. 18
      Adding Monitoring Capabilities ............................................................................................. 18
      Adding Configuration Data for a Component ....................................................................... 18
      Adding Container Capabilities ............................................................................................... 19
      Packaging and Delivering an Add-On Component ............................................................... 19

2 Writing HK2 Components ..................................................................................................................21
   HK2 Component Model ............................................................................................................. 21
   Services in the HK2 Component Model ................................................................................... 22
   HK2 Runtime ............................................................................................................................. 22
      Scopes of Services .................................................................................................................. 22
      Instantiation of Components in HK2 ..................................................................................... 23
      HK2 Lifecycle Interfaces ......................................................................................................... 23
   Inversion of Control .................................................................................................................... 24
      Injecting HK2 Components ..................................................................................................... 24
      Extraction ............................................................................................................................... 25
      Instantiation Cascading in HK2 ............................................................................................. 26
   Identifying a Class as an Add-On Component ......................................................................... 26
Using the Apache Maven Build System to Develop HK2 Components .............................................. 27

3 Extending the Administration Console ........................................................................................... 29
   Administration Console Architecture ............................................................................................. 30
   Implementing a Console Provider ................................................................................................. 30
   About Administration Console Templates .................................................................................... 31
   About Integration Points .................................................................................................................. 32
   Specifying the ID of an Add-On Component .................................................................................. 32
   Adding Functionality to the Administration Console ...................................................................... 33
      Adding a Node to the Navigation Tree ....................................................................................... 34
      Adding Tabs to a Page .................................................................................................................. 36
      Adding a Task to the Common Tasks Page ............................................................................... 39
      Adding a Task Group to the Common Tasks Page ................................................................. 40
      Adding Content to a Page ........................................................................................................... 42
      Adding a Page to the Administration Console ........................................................................ 44
   Adding Internationalization Support .............................................................................................. 45
   Changing the Theme or Brand of the Administration Console ...................................................... 45
   Creating an Integration Point Type ............................................................................................... 47
   ▼ To Create an Integration Point Type ........................................................................................ 47

4 Extending the asadmin Utility ......................................................................................................... 49
   About the Administrative Command Infrastructure of GlassFish Server .................................... 49
   Adding an asadmin Subcommand ................................................................................................. 50
      Representing an asadmin Subcommand as a Java Class ............................................................ 50
      Specifying the Name of an asadmin Subcommand ................................................................ 50
      Ensuring That an AdminCommand Implementation Is Stateless ............................................. 51
      Example of Adding an asadmin Subcommand ....................................................................... 51
   Adding Parameters to an asadmin Subcommand ....................................................................... 52
      Representing a Parameter of an asadmin Subcommand ............................................................ 52
      Identifying a Parameter of an asadmin Subcommand ................................................................. 53
      Specifying Whether a Parameter Is an Option or an Operand .................................................. 53
      Specifying the Name of an Option ............................................................................................. 53
      Specifying the Acceptable Values of a Parameter .................................................................... 54
      Specifying the Default Value of a Parameter .......................................................................... 55
      Specifying Whether a Parameter Is Required or Optional ....................................................... 55
Example of Adding Parameters to an asadmin Subcommand ............................................... 55
Adding Message Text Strings to an asadmin Subcommand .................................................... 56
Enabling an asadmin Subcommand to Run ............................................................................. 59
Setting the Context of an asadmin Subcommand ................................................................. 59
Changing the Brand in the GlassFish Server CLI ..................................................................... 59
Examples of Extending the asadmin Utility ......................................................................... 61

5 Adding Monitoring Capabilities .......................................................................................... 67
Defining Statistics That Are to Be Monitored ......................................................................... 67
  Defining an Event Provider ...................................................................................................... 68
  Sending an Event .................................................................................................................... 73
Updating the Monitorable Object Tree .................................................................................... 74
  Creating Event Listeners ........................................................................................................ 75
  Representing a Component’s Statistics in an Event Listener Class .................................. 75
  Subscribing to Events From Event Provider Classes ............................................................ 77
  Registering an Event Listener ............................................................................................... 78
Dotted Names for an Add-On Component’s Statistics .............................................................. 79
Example of Adding Monitoring Capabilities ............................................................................ 80

6 Adding Configuration Data for a Component ....................................................................... 87
How GlassFish Server Stores Configuration Data ................................................................ 87
Defining an Element ................................................................................................................ 88
  ▼ To Define an Element ........................................................................................................... 88
Defining an Attribute of an Element ....................................................................................... 89
  Representing an Attribute of an Element ............................................................................... 89
  Specifying the Data Type of an Attribute .............................................................................. 89
  Identifying an Attribute of an Element .................................................................................. 89
  Specifying the Name of an Attribute .................................................................................... 90
  Specifying the Default Value of an Attribute ........................................................................ 90
  Specifying Whether an Attribute Is Required or Optional ................................................ 90
  Example of Defining an Attribute Is Required or Optional .................................................. 91
Defining a Subelement ............................................................................................................ 91
  ▼ To Define a Subelement ...................................................................................................... 91
Validating Configuration Data ................................................................................................. 92
Initializing a Component’s Configuration Data ....................................................................... 94
Preface

This document explains how to use published interfaces of Oracle GlassFish Server to develop add-on components for GlassFish Server. This document explains how to perform only those tasks that ensure that the add-on component is suitable for GlassFish Server.

This document is for software developers who are developing add-on components for GlassFish Server. This document assumes that the developers are working with a distribution of GlassFish Server. Access to the source code of the GlassFish project is not required to perform the tasks in this document. This document also assumes the ability to program in the Java language.

This preface contains information about and conventions for the entire Oracle GlassFish Server (GlassFish Server) documentation set.

GlassFish Server 3.0.1 is developed through the GlassFish project open-source community at https://glassfish.dev.java.net/. The GlassFish project provides a structured process for developing the GlassFish Server platform that makes the new features of the Java EE platform available faster, while maintaining the most important feature of Java EE: compatibility. It enables Java developers to access the GlassFish Server source code and to contribute to the development of the GlassFish Server. The GlassFish project is designed to encourage communication between Oracle engineers and the community.

The following topics are addressed here:

- “GlassFish Server Documentation Set” on page 10
- “Related Documentation” on page 11
- “Typographic Conventions” on page 12
- “Symbol Conventions” on page 12
- “Default Paths and File Names” on page 13
- “Documentation, Support, and Training” on page 14
- “Searching Oracle Product Documentation” on page 14
- “Third-Party Web Site References” on page 14
GlassFish Server Documentation Set

The GlassFish Server documentation set describes deployment planning and system installation. The Uniform Resource Locator (URL) for GlassFish Server documentation is http://docs.sun.com/coll/1343.13. For an introduction to GlassFish Server, refer to the books in the order in which they are listed in the following table.

<table>
<thead>
<tr>
<th>Book Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release Notes</td>
<td>Provides late-breaking information about the software and the documentation. Includes a comprehensive, table-based summary of the supported hardware, operating system, Java Development Kit (JDK), and database drivers.</td>
</tr>
<tr>
<td>Quick Start Guide</td>
<td>Explains how to get started with the GlassFish Server product.</td>
</tr>
<tr>
<td>Installation Guide</td>
<td>Explains how to install the software and its components.</td>
</tr>
<tr>
<td>Upgrade Guide</td>
<td>Explains how to upgrade to the latest version of GlassFish Server. This guide also describes differences between adjacent product releases and configuration options that can result in incompatibility with the product specifications.</td>
</tr>
<tr>
<td>Administration Guide</td>
<td>Explains how to configure, monitor, and manage GlassFish Server subsystems and components from the command line by using the asadmin(1M) utility. Instructions for performing these tasks from the Administration Console are provided in the Administration Console online help.</td>
</tr>
<tr>
<td>Application Deployment Guide</td>
<td>Explains how to assemble and deploy applications to the GlassFish Server and provides information about deployment descriptors.</td>
</tr>
<tr>
<td>Your First Cup: An Introduction to the Java EE Platform</td>
<td>Provides a short tutorial for beginning Java EE programmers that explains the entire process for developing a simple enterprise application. The sample application is a web application that consists of a component that is based on the Enterprise JavaBeans specification, a JAX-RS web service, and a JavaServer Faces component for the web front end.</td>
</tr>
<tr>
<td>Application Development Guide</td>
<td>Explains how to create and implement Java Platform, Enterprise Edition (Java EE platform) applications that are intended to run on the GlassFish Server. These applications follow the open Java standards model for Java EE components and APIs. This guide provides information about developer tools, security, and debugging.</td>
</tr>
<tr>
<td>Add-On Component Development Guide</td>
<td>Explains how to use published interfaces of GlassFish Server to develop add-on components for GlassFish Server. This document explains how to perform only those tasks that ensure that the add-on component is suitable for GlassFish Server.</td>
</tr>
</tbody>
</table>
### Related Documentation

Javadoc tool reference documentation for packages that are provided with GlassFish Server is available as follows:

- The API specification for version 6 of Java EE is located at [http://download.oracle.com/docs/cd/E17410_01/javaee/6/api/](http://download.oracle.com/docs/cd/E17410_01/javaee/6/api/).
- The API specification for GlassFish Server 3.0.1, including Java EE 6 platform packages and nonplatform packages that are specific to the GlassFish Server product, is located at: [https://glassfish.dev.java.net/nonav/docs/v3/api/](https://glassfish.dev.java.net/nonav/docs/v3/api/).

Additionally, the following resources might be useful:

- The [Java EE Specifications](http://java.sun.com/javaee/technologies/index.jsp)
- The [Java EE Blueprints](http://java.sun.com/reference/blueprints/)

For information about creating enterprise applications in the NetBeans Integrated Development Environment (IDE), see [http://www.netbeans.org/kb/](http://www.netbeans.org/kb/).
For information about the Java DB for use with the GlassFish Server, see http://developers.sun.com/javadb/.

The GlassFish Samples project is a collection of sample applications that demonstrate a broad range of Java EE technologies. The GlassFish Samples are bundled with the Java EE Software Development Kit (SDK), and are also available from the GlassFish Samples project page at https://glassfish-samples.dev.java.net/.

### Typographic Conventions

The following table describes the typographic changes that are used in this book.

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories, and onscreen computer output</td>
<td>Edit your .login file. Use ls -a to list all files. machine_name% you have mail.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>What you type, contrasted with onscreen computer output</td>
<td>machine_name% su Password:</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>A placeholder to be replaced with a real name or value</td>
<td>The command to remove a file is rm filename.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Book titles, new terms, and terms to be emphasized (note that some emphasized items appear bold online)</td>
<td>Read Chapter 6 in the User's Guide. A cache is a copy that is stored locally. Do not save the file.</td>
</tr>
</tbody>
</table>

### Symbol Conventions

The following table explains symbols that might be used in this book.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>Contains optional arguments and command options.</td>
<td>ls [-l]</td>
<td>The -l option is not required.</td>
</tr>
<tr>
<td>{ }</td>
<td>Contains a set of choices for a required command option.</td>
<td>-d (y</td>
<td>n)</td>
</tr>
</tbody>
</table>
TABLE P–3  Symbol Conventions  (Continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$( )</td>
<td>Indicates a variable reference.</td>
<td>$(com.sun.javaRoot)</td>
<td>References the value of the com.sun.javaRoot variable.</td>
</tr>
<tr>
<td>-</td>
<td>Joins simultaneous multiple keystrokes.</td>
<td>Control-A</td>
<td>Press the Control key while you press the A key.</td>
</tr>
<tr>
<td>+</td>
<td>Joins consecutive multiple keystrokes.</td>
<td>Ctrl+A+N</td>
<td>Press the Control key, release it, and then press the subsequent keys.</td>
</tr>
<tr>
<td>→</td>
<td>Indicates menu item selection in a graphical user interface.</td>
<td>File → New → Templates</td>
<td>From the File menu, choose New. From the New submenu, choose Templates.</td>
</tr>
</tbody>
</table>

Default Paths and File Names

The following table describes the default paths and file names that are used in this book.

TABLE P–4  Default Paths and File Names

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
</table>
| as-install  | Represents the base installation directory for GlassFish Server. In configuration files, as-install is represented as follows: | Installations on the Oracle Solaris operating system, Linux operating system, and Mac operating system: 
user’s-home-directory/glassfishv3/glassfish  
Windows, all installations: 
SystemDrive:\glassfishv3\glassfish  |
| as-install-parent | Represents the parent of the base installation directory for GlassFish Server. | Installations on the Oracle Solaris operating system, Linux operating system, and Mac operating system: 
user’s-home-directory/glassfishv3  
Windows, all installations: 
SystemDrive:\glassfishv3 |
| domain-root-dir | Represents the directory in which a domain is created by default. | as-install/domains/ |
| domain-dir   | Represents the directory in which a domain’s configuration is stored. In configuration files, domain-dir is represented as follows: | domain-root-dir/domain-name |
|             | | |
Documentation, Support, and Training

The Oracle web site provides information about the following additional resources:

- Documentation (http://docs.sun.com/)
- Support (http://www.sun.com/support/)
- Training (http://education.oracle.com/)

Searching Oracle Product Documentation

Besides searching Oracle product documentation from the docs.sun.com web site, you can use a search engine by typing the following syntax in the search field:

search-term site:docs.sun.com

For example, to search for “broker,” type the following:

broker site:docs.sun.com

To include other Oracle web sites in your search (for example, java.sun.com, www.sun.com, and developers.sun.com), use sun.com in place of docs.sun.com in the search field.

Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

Note – Oracle is not responsible for the availability of third-party web sites mentioned in this document. Oracle does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Oracle will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.
Introduction to the Development Environment for GlassFish Server Add-On Components

Oracle GlassFish Server enables an external vendor such as an independent software vendor (ISV), original equipment manufacturer (OEM), or system integrator to incorporate GlassFish Server into a new product with the vendor’s own brand name. External vendors can extend the functionality of GlassFish Server by developing add-on components for GlassFish Server. GlassFish Server provides interfaces to enable add-on components to be configured, managed, and monitored through existing GlassFish Server tools such as the Administration Console and the asadmin utility.

The following topics are addressed here:

- “GlassFish Server Modular Architecture and Add-On Components” on page 15
- “OSGi Alliance Module Management Subsystem” on page 16
- “Hundred-Kilobyte Kernel” on page 16
- “Overview of the Development Process for an Add-On Component” on page 17

GlassFish Server Modular Architecture and Add-On Components

GlassFish Server has a modular architecture in which the features of GlassFish Server are provided by a consistent set of components that interact with each other. Each component provides a small set of functionally related features.

The modular architecture of GlassFish Server enables users to download and install only the components that are required for the applications that are being deployed. As a result, start-up times, memory consumption, and disk space requirements are all minimized.

The modular architecture of GlassFish Server enables you to extend the basic functionality of GlassFish Server by developing add-on components. An add-on component is an encapsulated definition of reusable code that has the following characteristics:

- The component provides a set of Java classes.
The component offers services and public interfaces.
The component implements the public interfaces with a set of private classes.
The component depends on other components.

Add-on components that you develop interact with GlassFish Server in the same way as components that are supplied in GlassFish Server distributions.

You can create and offer new or updated add-on components at any time. GlassFish Server administrators can install add-on components and update or remove installed components after GlassFish Server is installed. For more information, see Chapter 10, "Extending GlassFish Server," in Oracle GlassFish Server 3.0.1 Administration Guide.

OSGi Alliance Module Management Subsystem

To enable components to be added when required, GlassFish Server provides a lightweight and extensible kernel that uses the module management subsystem from the OSGi Alliance. Any GlassFish Server component that plugs into this kernel must be implemented as an OSGi bundle. To enable an add-on component to plug in to the GlassFish Server kernel in the same way as other components, package the component as an OSGi bundle. For more information, see “Packaging an Add-On Component” on page 119.

The default OSGi module management subsystem in GlassFish Server is the Apache Felix OSGi framework. However, the GlassFish Server kernel uses only the OSGi Service Platform Release 4 API. Therefore, GlassFish Server supports other OSGi module management subsystems that are compatible with the OSGi Service Platform Release 4 API.

Hundred-Kilobyte Kernel

The Hundred-Kilobyte Kernel (HK2) is the lightweight and extensible kernel of GlassFish Server. HK2 consists of the following technologies:

- **Module subsystem.** The HK2 module subsystem provides isolation between components of the GlassFish Server. The HK2 module subsystem is compatible with existing technologies such as the OSGi framework.

- **Component model.** The HK2 component model eases the development of components that are also services. GlassFish Server discovers these components automatically and dynamically. HK2 components use injection of dependencies to express dependencies on other components. GlassFish Server provides two-way mappings between the services of an HK2 component and OSGi services.

For more information, see Chapter 2, "Writing HK2 Components."
Overview of the Development Process for an Add-On Component

To ensure that an add-on component behaves identically to components that are supplied in GlassFish Server distributions, the component must meet the following requirements:

- If the component generates management data or monitoring data, it must provide that data to other GlassFish Server components in the same way as other GlassFish Server components.
- If the component generates management data or monitoring data, it must provide that data to users through GlassFish Server administrative interfaces such as Administration Console and the asadmin utility.
- The component must be packaged and delivered as an OSGi bundle.

To develop add-on components that meet these requirements, follow the development process that is described in the following sections:

- “Writing HK2 Components” on page 17
- “Extending the Administration Console” on page 17
- “Extending the asadmin Utility” on page 18
- “Adding Monitoring Capabilities” on page 18
- “Adding Configuration Data for a Component” on page 18
- “Adding Container Capabilities” on page 19
- “Packaging and Delivering an Add-On Component” on page 19

Writing HK2 Components

The Hundred-Kilobyte Kernel (HK2) is the lightweight and extensible kernel of GlassFish Server. To interact with GlassFish Server, add-on components plug into this kernel. In the HK2 component model, the functions of an add-on component are declared through a contract-service implementation paradigm. An HK2 contract identifies and describes the building blocks or the extension points of an application. An HK2 service implements an HK2 contract.

For more information, see Chapter 2, “Writing HK2 Components.”

Extending the Administration Console

The Administration Console is a browser-based tool for administering GlassFish Server. It features an easy-to-navigate interface and online help. Extending the Administration Console enables you to provide a graphical user interface for administering your add-on component. You can use any of the user interface features of the Administration Console, such as tree nodes,
Extending the \texttt{asadmin} Utility

The \texttt{asadmin} utility is a command-line tool for configuring and administering GlassFish Server. Extending the \texttt{asadmin} utility enables you to provide administrative interfaces for an add-on component that are consistent with the interfaces of other GlassFish Server components. A user can run \texttt{asadmin} subcommands either from a command prompt or from a script. For more information about the \texttt{asadmin} utility, see the \texttt{asadmin(1M)} man page.

For more information, see Chapter 4, “Extending the \texttt{asadmin} Utility.”

Adding Monitoring Capabilities

Monitoring is the process of reviewing the statistics of a system to improve performance or solve problems. By monitoring the state of components and services that are deployed in the GlassFish Server, system administrators can identify performance bottlenecks, predict failures, perform root cause analysis, and ensure that everything is functioning as expected. Monitoring data can also be useful in performance tuning and capacity planning.

An add-on component typically generates statistics that the GlassFish Server can gather at run time. Adding monitoring capabilities enables an add-on component to provide statistics to GlassFish Server in the same way as components that are supplied in GlassFish Server distributions. As a result, system administrators can use the same administrative interfaces to monitor statistics from any installed GlassFish Server component, regardless of the origin of the component.

For more information, see Chapter 5, "Adding Monitoring Capabilities.”

Adding Configuration Data for a Component

The configuration data of a component determines the characteristics and runtime behavior of a component. GlassFish Server provides interfaces to enable an add-on component to store its configuration data in the same way as other GlassFish Server components. These interfaces are similar to interfaces that are defined in Java Specification Request (JSR) 222: Java Architecture for XML Binding (JAXB) 2.0. By using these interfaces to store configuration data, you ensure that the add-on component is fully integrated with GlassFish Server. As a result, administrators can configure an add-on component in the same way as they can configure other GlassFish Server components.
For more information, see Chapter 6, “Adding Configuration Data for a Component.”

**Adding Container Capabilities**

Applications run on GlassFish Server in containers. GlassFish Server enables you to create containers that extend or replace the existing containers of GlassFish Server. Adding container capabilities enables you to run new types of applications and to deploy new archive types in GlassFish Server.

For more information, see Chapter 7, “Adding Container Capabilities.”

**Packaging and Delivering an Add-On Component**

Packaging an add-on component enables the component to interact with the GlassFish Server kernel in the same way as other components. Integrating a component with GlassFish Server enables GlassFish Server to discover the component at runtime. If an add-on component is an extension or update to existing installations of GlassFish Server, deliver the component through Update Tool.

For more information, see Chapter 8, “Packaging, Integrating, and Delivering an Add-On Component.”
Writing HK2 Components

The Hundred-Kilobyte Kernel (HK2) is the lightweight and extensible kernel of GlassFish Server. To interact with GlassFish Server, add-on components plug in to this kernel. In the HK2 component model, the functions of an add-on component are declared through a contract-service implementation paradigm. An HK2 contract identifies and describes the building blocks or the extension points of an application. An HK2 service implements an HK2 contract.

The following topics are addressed here:

- “HK2 Component Model” on page 21
- “Services in the HK2 Component Model” on page 22
- “HK2 Runtime” on page 22
- “Inversion of Control” on page 24
- “Identifying a Class as an Add-On Component” on page 26
- “Using the Apache Maven Build System to Develop HK2 Components” on page 27

HK2 Component Model

The Hundred-Kilobyte Kernel (HK2) provides a module system and component model for building complex software systems. HK2 forms the core of GlassFish Server’s architecture.

The module system is responsible for instantiating classes that constitute the application functionality. The HK2 runtime complements the module system by creating objects. It configures such objects by:

- Injecting other objects that are needed by a newly instantiated object
- Injecting configuration information needed for that object
- Making newly created objects available, so that they can then be injected to other objects that need it
Services in the HK2 Component Model

An HK2 service identifies the building blocks or the extension points of an application. A service is a plain-old Java object (POJO) with the following characteristics:

- The object implements an interface.
- The object is declared in a JAR file with the META-INF/services file.

To clearly separate the contract interface and its implementation, the HK2 runtime requires the following information:

- Which interfaces are contracts
- Which implementations of such interfaces are services

Interfaces that define a contract are identified by the `org.jvnet.hk2.annotation.Contract` annotation.

```java
@Retention(RUNTIME)
@Target(TYPE)
public @interface Contract {
}
```

Implementations of such contracts should be identified with an `org.jvnet.hk2.annotations.Service` annotation so that the HK2 runtime can recognize them as @Contract implementations.

```java
@Retention(RUNTIME)
@Target(TYPE)
public @interface Service {
    ...
}
```

For more information, see *Service*.

HK2 Runtime

Once Services are defined, the HK2 runtime can be used to instantiate or retrieve instances of services. Each service instance has a scope, specified as singleton, per thread, per application, or a custom scope.

Scopes of Services

You can specify the scope of a service by adding an `org.jvnet.hk2.annotations Scoped` annotation to the class-level of your @Service implementation class. Scopes are also services, so they can be custom defined and added to the HK2 runtime before being used by other services.
Each scope is responsible for storing the service instances to which it is tied; therefore, the HK2 runtime does not rely on predefined scopes (although it comes with a few predefined ones).

```java
@Contract
public abstract class Scope {
    public abstract ScopeInstance current();
}
```

The following code fragment shows how to set the scope for a service to the predefined Singleton scope:

```java
@Service
public Singleton implements Scope {
    ...
}

@Scope(Singleton.class)
@Service
public class SingletonService implements RandomContract {
    ...
}
```

You can define a new Scope implementation and use that scope on your @Service implementations. You will see that the HK2 runtime uses the Scope instance to store and retrieve service instances tied to that scope.

**Instantiation of Components in HK2**

Do not call the `new` method to instantiate components. Instead, retrieve components by using the `ComponentManager` instance. The simplest way to use the `ComponentManager` instance is through a `getComponent(Class<T> contract)` call:

```java
public <T> T getComponent(Class<T> clazz) throws ComponentException;
```

More APIs are available at `ComponentManager`.

**HK2 Lifecycle Interfaces**

Components can attach behaviors to their construction and destruction events by implementing the `org.jvnet.hk2.component.PostConstruct` interface, the `org.jvnet.hk2.component.PreDestroy` interface, or both. These are interfaces rather than annotations for performance reasons.

The `PostConstruct` interface defines a single method, `postConstruct`, which is called after a component has been initialized and all its dependencies have been injected.
The `PreDestroy` interface defines a single method, `preDestroy`, which is called just before a component is removed from the system.

**EXAMPLE 2–1**  Example Implementation of `PostConstruct` and `PreDestroy`

```java
@Service(name="com.example.container.MyContainer")
public class MyContainer implements Container, PostConstruct, PreDestroy {
    @Inject
    Logger logger;
    ...
    public void postConstruct() {
        logger.info("Starting up.");
    }
    ...
    public void preDestroy() {
        logger.info("Shutting down.");
    }
}
```

## Inversion of Control

Inversion of control (IoC) refers to a style of software architecture where the behavior of a system is determined by the runtime capabilities of the individual, discrete components that make up the system. This architecture is different from traditional styles of software architecture, where all the components of a system are specified at design-time. With IoC, discrete components respond to high-level events to perform actions. While performing these actions, the components typically rely on other components to provide other actions. In an IoC system, components use injection to gain access to other components, and extraction to make component variables available to the system.

## Injecting HK2 Components

Services usually rely on other services to perform their tasks. The HK2 runtime identifies the `@Contract` implementations required by a service by using the `org.jvnet.hk2.annotations.Inject` annotation. `Inject` can be placed on fields or setter methods of any service instantiated by the HK2 runtime. The target service is retrieved and injected during the calling service's instantiation by the component manager.

The following example shows how to use `@Inject` at the field level:

```java
@Inject
ConfigService config;
```

The following example shows how to use `@Inject` at the setter level:
@Inject
public void set(ConfigService svc) {...}

Injection can further qualify the intended injected service implementation by using a name and scope from which the service should be available:

@Inject(Scope=Singleton.class, name="deploy")
AdminCommand deployCommand;

## Extraction

Although all services are automatically placed into a scope for later retrieval, a component may need to extract more than itself. One practical way of doing so is to use a factory service. For simplicity, however, the HK2 runtime extracts all fields or getter methods annotated with the org.jvnet.hk2.annotations.Extract annotation.

The following example shows how to use @Extract at the field level:

@Extract
ConfigService config;

The following example shows how to use @Extract at the getter level:

@Extract
public ConfigService getConfigService() {...}

Extraction, like injection, can also use the name and scope annotation fields to further qualify the extracted Contract implementation.

Extracted fields and properties are made available to other service instances by exporting them to the org.jvnet.hk2.component.Habitat instance. Habitat instances can be injected into other components, and the components can then extract and use the data contained in the Habitat instance.

@Inject
protected Habitat habitat;
...
public void doSomething(String name) {
    ...
    ConfigService config = habitat.getComponent(ConfigService.class);
    ...
}
Instantiation Cascading in HK2

Injection of instances that have not been already instantiated triggers more instantiation. You can see this as a component instantiation cascade where some code requests for a high-level service will, by using the @Inject annotation, require more injection and instantiation of lower level services. This cascading feature keeps the implementation as private as possible while relying on interfaces and the separation of contracts and providers.

**EXAMPLE 2-2**  Example of Instantiation Cascading

The following example shows how the instantiation of DeploymentService as a Startup contract implementation will trigger the instantiation of the ConfigService.

```java
@Contract
public interface Startup {...}

Iterable<Startup> startups;
startups = componentMgr.getComponents(Startup.class);

@Service
public class DeploymentService implements Startup {
    @Inject
    ConfigService config;
}

@Service
public class ConfigService implements ... {...}
```

Identifying a Class as an Add-On Component

GlassFish Server discovers add-on components by identifying Java programming language classes that are annotated with the org.jvnet.hk2.annotation.Service annotation.

To identify a class as an implementation of an GlassFish Server service, add the org.jvnet.hk2.annotation.Service annotation at the class-definition level of your Java programming language class.

```java
@Service
public class SamplePlugin implements ConsoleProvider {
    ...
}
```

The @Service annotation has the following elements. All elements are optional.

name
The name of the service. The default value is an empty string.
scope
The scope to which this service implementation is tied. The default value is org.jvnet.hk2.component.PerLookup.class.

factory
The factory class for the service implementation, if the service is created by a factory class rather than by calling the default constructor. If this element is specified, the factory component is activated, and Factory.getObject is used instead of the default constructor. The default value of the factory element is org.jvnet.hk2.component.Factory.class.

EXAMPLE 2–3  Example of the Optional Elements of the @Service Annotation

The following example shows how to use the optional elements of the @Service annotation:

@Service (name="MyService",
    scope=com.example.PerRequest.class,
    factory=com.example.MyCustomFactory)
public class SamplePlugin implements ConsoleProvider {
    ...
}

Using the Apache Maven Build System to Develop HK2 Components

If you are using Maven 2 to build HK2 components, invoke the auto-depends plug-in for Maven so that the META-INF/services files are generated automatically during build time.

EXAMPLE 2–4  Example of the Maven Plug-In Configuration

<plugin>
    <groupId>com.sun.enterprise</groupId>
    <artifactId>hk2-maven-plugin</artifactId>
    <configuration>
        <includes>
            <include>com/sun/enterprise/v3/**</include>
        </includes>
    </configuration>
</plugin>

EXAMPLE 2–5  Example of META-INF/services File Generation

This example shows how to use @Contract and @Service and the resulting META-INF/services files.

The interfaces and classes in this example are as follows:
package com.sun.v3.annotations;
@Contract
public interface Startup {...}

package com.wombat;
@Contract
public interface RandomContract {...}

package com.sun.v3;
@Service
public class MyService implements Startup, RandomContract, PropertyChangeListener {
   ...
}

These interfaces and classes generate this META-INF/services file with the MyService content:

com.sun.v3.annotations.Startup
com.wombat.RandomContract
Extending the Administration Console

The Administration Console is a browser-based tool for administering GlassFish Server. It features an easy-to-navigate interface and online help. Extending the Administration Console enables you to provide a graphical user interface for administering your add-on component. You can use any of the user interface features of the Administration Console, such as tree nodes, links on the Common Tasks page, tabs and sub-tabs, property sheets, and JavaServer Faces pages. Your add-on component implements a marker interface and provides a configuration file that describes how your customizations integrate with the Administration Console.

This chapter refers to a simple example called console-sample-ip that illustrates how to provide Administration Console features for a hypothetical add-on component. Instructions for obtaining and using this example are available at the example's project page (http://wiki.glassfish.java.net/Wiki.jsp?page=V3SampleIpProject). When you check out the code, it is placed in a directory named glassfish-samples/v3/plugin/adminconsole/console-sample-ip/ in your current directory. In this chapter, path names for the example files are relative to this directory.

The following topics are addressed here:

- “Administration Console Architecture” on page 30
- “About Administration Console Templates” on page 31
- “About Integration Points” on page 32
- “Specifying the ID of an Add-On Component” on page 32
- “Adding Functionality to the Administration Console” on page 33
- “Adding Internationalization Support” on page 45
- “Changing the Theme or Brand of the Administration Console” on page 45
- “Creating an Integration Point Type” on page 47
Administration Console Architecture

The Administration Console is a web application that is composed of OSGi bundles. These bundles provide all the features of the Administration Console, such as the Web Applications, Update Center, and Security content. To provide support for your add-on component, create your own OSGi bundle that implements the parts of the user interface that you need. Place your bundle in the modules directory of your GlassFish Server installation, along with the other Administration Console bundles.

To learn how to package the Administration Console features for an add-on component, go to the modules directory of your GlassFish Server installation and examine the contents of the files named console-componentname-plugin.jar. Place the console-sample-ip project bundle in the same place to deploy it and examine the changes that it makes to the Administration Console.

The Administration Console includes a Console Add-On Component Service. The Console Add-On Component Service is an HK2 service that acts as a façade to all the Administration Console add-on components. The Console Add-On Component Service queries the various console providers for integration points so that it can perform the actions needed for the integration (adding a tree node or a new tab, for example). The interface name for this service is org.glassfish.api.admingui.ConsolePluginService.

For details about the Hundred-Kilobyte Kernel (HK2) project, see "Hundred-Kilobyte Kernel" on page 16 and "HK2 Component Model" on page 21.

Each add-on component must contain a console provider implementation. This is a Java class that implements the org.glassfish.api.admingui.ConsoleProvider interface and uses the HK2 @Service annotation. The console provider allows your add-on component to specify where your integration point configuration file is located. This configuration file communicates to the Console Add-On Component Service the customizations that your add-on component makes to the Administration Console.

Implementing a Console Provider

The org.glassfish.api.admingui.ConsoleProvider interface has one required method, getConfiguration. The getConfiguration method returns the location of the console-config.xml file as a java.net.URL. If getConfiguration returns null, the default location, META-INF/admingui/console-config.xml, is used. The console-config.xml file is described in “About Integration Points” on page 32.

To implement the console provider for your add-on component, write a Java class that is similar to the following example.

**EXAMPLE 3–1   Example ConsoleProvider Implementation**

This example shows a simple implementation of the ConsoleProvider interface:
package org.glassfish.admingui.plugin;
import org.glassfish.api.admingui.ConsoleProvider;
import org.jvnet.hk2.annotations.Service;
import java.net.URL;
@Service
public class SamplePlugin implements ConsoleProvider {
    public URL getConfiguration() { return null; }
}

This implementation of getConfiguration returns null to specify that the configuration file is in the default location. If you place the file in a nonstandard location or give it a name other than console-config.xml, your implementation of getConfiguration must return the URL where the file can be found.

You can find this example code in the file project/src/main/java/org/glassfish/admingui/plugin/SamplePlugin.java.

About Administration Console Templates

GlassFish Server includes a set of templates that make it easier to create JavaServer Faces pages for your add-on component. These templates use Templat for JavaServer Faces Technology (https://jsftemplating.dev.java.net/), which is also known as JSFTemplating.

Examples of JSFTemplating technology can be found in the following sections of this chapter:
- “Creating a JavaServer Faces Page for Your Node” on page 35
- “Creating JavaServer Faces Pages for Your Tabs” on page 38
- “Creating a JavaServer Faces Page for Your Task” on page 40
- “Creating a JavaServer Faces Page for Your Task Group” on page 41
- “Creating a JavaServer Faces Page for Your Page Content” on page 43
- “Adding a Page to the Administration Console” on page 44
About Integration Points

The integration points for your add-on component are the individual Administration Console user interface features that your add-on component will extend. You can implement the following kinds of integration points:

- Nodes in the navigation tree
- Elements on the Common Tasks page of the Administration Console
- JavaServer Faces pages
- Tabs and sub-tabs

Specify all the integration points in a file named console-config.xml. In the example, this file is in the directory project/src/main/resources/META-INF/admingui/. The following sections describe how to create this file.

In addition, create JavaServer Faces pages that contain JSF code fragments to implement the integration points. In the example, these files are in the directory project/src/main/resources/. The content of these files depends on the integration point you are implementing. The following sections describe how to create these JavaServer Faces pages.

For reference information on integration points, see Appendix A, "Integration Point Reference."

Specifying the ID of an Add-On Component

The console-config.xml file consists of a console-config element that encloses a series of integration-point elements. The console-config element has one attribute, id, which specifies a unique name or ID value for the add-on component.

In the example, the element is declared as follows:

```xml
<console-config id="sample">
  ...
</console-config>
```

You will also specify this ID value when you construct URLs to images, resources and pages in your add-on component. See “Adding a Node to the Navigation Tree” on page 34 for an example.

For example, a URL to an image named my.gif might look like this:

```xml
<sun:image url="/resource/sample/images/my.gif" />
```
The URL is constructed as follows:
- `/resource` is required to locate any resource URL.
- `sample` is the add-on component ID. You must choose a unique ID value.
- `images` is a folder under the root of the add-on component JAR file.

**Adding Functionality to the Administration Console**

The integration-point elements in the `console-config.xml` file specify attributes for the user interface features that you choose to implement. The example file provides examples of most of the available kinds of integration points at this release. Your own add-on component can use some or all of them.

For each integration-point element, specify the following attributes.

- **id**
  - An identifier for the integration point.

- **parentId**
  - The ID of the integration point's parent.

- **type**
  - The type of the integration point.

- **priority**
  - A numeric value that specifies the relative ordering of integration points for add-on components that specify the same `parentId`. A lower number specifies a higher priority (for example, 100 represents a higher priority than 400). The integration points for add-on components are always placed after those in the basic Administration Console. You might need to experiment to place the integration point where you want it. This attribute is optional.

- **content**
  - The content for the integration point, typically a JavaServer Faces page. In the example, you can find the JavaServer Faces pages in the directory `project/src/main/resources/`.

---

**Note** - The order in which these attributes are specified does not matter, and in the example `console-config.xml` file the order varies. To improve readability, this chapter uses the same order throughout.

The following topics are addressed here:
- “Adding a Node to the Navigation Tree” on page 34
- “Adding Tabs to a Page” on page 36
- “Adding a Task to the Common Tasks Page” on page 39
- “Adding a Task Group to the Common Tasks Page” on page 40
Adding a Node to the Navigation Tree

You can add a node to the navigation tree, either at the top level or under another node. To add a node, use an integration point of type `org.glassfish.admingui:navNode`. Use the `parentId` attribute to specify where the new node should be placed. Any tree node, including those added by other add-on components, can be specified. Examples include the following:

- **tree**
  - At the top level
- **applicationServer**
  - Under the GlassFish Server node
- **applications**
  - Under the Applications node
- **resources**
  - Under the Resources node
- **configuration**
  - Under the Configuration node
- **webContainer**
  - Under the Web Container node
- **httpService**
  - Under the HTTP Service node

**Note** – The `webContainer` and `httpService` nodes are available only if you installed the web container module for the Administration Console (the `console-web-gui.jar` OSGi bundle).

If you do not specify a `parentId`, the new content is added to the root of the integration point, in this case the top level node, `tree`.

**EXAMPLE 3-2  Example Tree Node Integration Point**

For example, the following `integration-point` element uses a `parentId` of `tree` to place the new node at the top level.

```xml
<integration-point
  id="sampleNode"
  parentId="tree"
  type="org.glassfish.admingui:treeNode"
```
EXAMPLE 3–2  Example Tree Node Integration Point  (Continued)

    priority="200"
    content="sampleNode.jsf"

This example specifies the following values in addition to the parent ID:

- The `id` value, `sampleNode`, specifies the integration point ID.
- The `type` value, `org.glassfish.admingui:treeNode`, specifies the integration point type as a tree node.
- The `priority` value, `200`, specifies the order of the node on the tree.
- The `content` value, `sampleNode.jsf`, specifies the JavaServer Faces page that displays the node.

The example `console-config.xml` file provides other examples of tree nodes under the Resources and Configuration nodes.

**Creating a JavaServer Faces Page for Your Node**

A JavaServer Faces page for a tree node uses the tag `sun:treeNode`. This tag provides all the capabilities of the Project Woodstock tag `webui:jsf:treeNode`.

EXAMPLE 3–3  Example JavaServer Faces Page for a TreeNode

In the example, the `sampleNode.jsf` file has the following content:

```xml
<sun:treeNode
    id="treeNode1"
    text="SampleTop"
    url="/sample/page/testPage.jsf?name=SampleTop"
    imageURL="/resource/sample/images/sample.png"
>
    <sun:treeNode
        id="treeNodeBB"
        text="SampleBB"
        url="/sample/page/testPage.jsf?name=SampleBB"
        imageURL="/resource/sample/images/sample.png"
    />
</sun:treeNode>
```

This file uses the `sun:treeNode` tag to specify both a top-level tree node and another node nested beneath it. In your own JavaServer Faces pages, specify the attributes of this tag as follows:

- `id`
  - A unique identifier for the tree node.
text
The node name that appears in the tree.

url
The location of the JavaServer Faces page that appears when you click the node. In the
element, most of the integration points use a very simple JavaServer Faces page called
testPage.jsf, which is in the src/main/resources/page/ directory. Specify the
integration point id value as the root of the URL; in this case, it is sample (see “Specifying the
ID of an Add-On Component” on page 32). The rest of the URL is relative to the
src/main/resources/ directory, where sampleNode.jsf resides.

The url tag in this example passes a name parameter to the JavaServer Faces page.

imageURL
The location of a graphic to display next to the node name. In the example, the graphic is
always sample.png, which is in the src/main/resources/images/ directory. The URL for
this image is an absolute path, /resource/sample/images/sample.png, where sample in the
path is the integration point id value (see “Specifying the ID of an Add-On Component” on
page 32).

Adding Tabs to a Page

You can add a tab to an existing tab set, or you can create a tab set for your own page. One way
to add a tab or tab set is to use an integration point of type
org.glassfish.admingui:serverInstTab, which adds a tab to the tab set on the main
GlassFish Server page of the Administration Console. You can also create sub-tabs. Once again,
the parentId element specifies where to place the tab or tab set.

EXAMPLE 3–4  Example Tab Integration Point

In the example, the following integration-point element adds a new tab on the main
GlassFish Server page of the Administration Console:

```xml
<integration-point
    id="sampleTab"
    parentId="serverInstTabs"
    type="org.glassfish.admingui:serverInstTab"
    priority="500"
    content="sampleTab.jsf"
/>```
This example specifies the following values:

- The id value, sampleTab, specifies the integration point ID.
- The parentId value, serverInstTabs, specifies the tab set associated with the server instance. The GlassFish Server page is the only one of the default Administration Console pages that has a tab set.
- The type value, org.glassfish.admingui:serverInstTab, specifies the integration point type as a tab associated with the server instance.
- The priority value, 500, specifies the order of the tab within the tab set. This value is optional.
- The content value, sampleTab.jsf, specifies the JavaServer Faces page that displays the tab.

**EXAMPLE 3-5  Example Tab Set Integration Points**

The following integration-point elements add a new tab with two sub-tabs, also on the main GlassFish Server page of the Administration Console:

```xml
<integration-point
    id="sampleTabWithSubTab"
    parentId="serverInstTabs"
    type="org.glassfish.admingui:serverInstTab"
    priority="300"
    content="sampleTabWithSubTab.jsf"
/>

<integration-point
    id="sampleSubTab1"
    parentId="sampleTabWithSubTab"
    type="org.glassfish.admingui:serverInstTab"
    priority="300"
    content="sampleSubTab1.jsf"
/>

<integration-point
    id="sampleSubTab2"
    parentId="sampleTabWithSubTab"
    type="org.glassfish.admingui:serverInstTab"
    priority="400"
    content="sampleSubTab2.jsf"
/>
```
These examples specify the following values:

- The id values, `sampleTabWithSubTab`, `sampleSubTab1`, and `sampleSubTab2`, specify the integration point IDs for the tab and its sub-tabs.
- The parentId of the new tab, `serverInstTabs`, specifies the tab set associated with the server instance. The parentId of the two sub-tabs, `sampleTabWithSubTab`, is the id value of this new tab.
- The type value, `org.glassfish.admingui:serverInstTab`, specifies the integration point type for all the tabs as a tab associated with the server instance.
- The priority values specify the order of the tabs within the tab set. This value is optional. In this case, the priority value for `sampleTabWithSubTab` is 300, which is higher than the value for `sampleTab`. That means that `sampleTabWithSubTab` appears to the left of `sampleTab` in the Administration Console. The priority values for `sampleSubTab1` and `sampleSubTab2` are 300 and 400 respectively, so `sampleSubTab1` appears to the left of `sampleSubTab2`.
- The content values, `sampleTabWithSubTab.jsf`, `sampleSubTab1.jsf`, and `sampleSubTab2.jsf`, specify the JavaServer Faces pages that display the tabs.

**Creating JavaServer Faces Pages for Your Tabs**

A JavaServer Faces page for a tab uses the tag `sun:tab`. This tag provides all the capabilities of the Project Woodstock tag `webui:jsf:tab`.

**EXAMPLE 3-6  Example JavaServer Faces Page for a Tab**

In the example, the `sampleTab.jsf` file has the following content:

```xml
<sun:tab id="sampleTab" immediate="true" text="Sample First Tab">
  <!command
    setSessionAttribute(key="serverInstTabs" value="sampleTab");
    gf.redirect(page="#{request.contextPath}/page/tabPage.jsf?name=Sample%20First%20Tab");
  />
</sun:tab>
```

**Note** – In the actual file there are no line breaks in the `gf.redirect` value.

In your own JavaServer Faces pages, specify the attributes of this tag as follows:

- **id**
  A unique identifier for the tab, in this case `sampleTab`.

- **immediate**
  If set to true, event handling for this component should be handled immediately (in the Apply Request Values phase) rather than waiting until the Invoke Application phase.
The tab name that appears in the tab set.

The JSF page displays tab content differently from the way the page for a node displays node content. It invokes two handlers for the command event: `setSessionAttribute` and `gf.redirect`. The `gf.redirect` handler has the same effect for a tab that the `url` attribute has for a node. It navigates to a simple JavaServer Faces page called `tabPage.jsf`, in the `src/main/resources/page/` directory, passing the text “Sample First Tab” to the page in a `name` parameter.

The `sampleSubTab1.jsf` and `sampleSubTab2.jsf` files are almost identical to `sampleTab.jsf`. The most important difference is that each sets the session attribute `serverInstTabs` to the base name of the JavaServer Faces file that corresponds to that tab:

```java
setSessionAttribute(key="serverInstTabs" value="sampleTab");
setSessionAttribute(key="serverInstTabs" value="sampleSubTab1");
setSessionAttribute(key="serverInstTabs" value="sampleSubTab2");
```

### Adding a Task to the Common Tasks Page

You can add either a single task or a group of tasks to the Common Tasks page of the Administration Console. To add a task or task group, use an integration point of type `org.glassfish.admingui:commonTask`.

See “Adding a Task Group to the Common Tasks Page” on page 40 for information on adding a task group.

**EXAMPLE 3-7 Example Task Integration Point**

In the example `console-config.xml` file, the following `integration-point` element adds a task to the Deployment task group:

```xml
<integration-point
    id="sampleCommonTask"
    parentId="deployment"
    type="org.glassfish.admingui:commonTask"
    priority="200"
    content="sampleCommonTask.jsf"
/>
```
This example specifies the following values:

- The id value, sampleCommonTask, specifies the integration point ID.
- The parentId value, deployment, specifies that the task is to be placed in the Deployment task group.
- The type value, org.glassfish.admingui:commonTask, specifies the integration point type as a common task.
- The priority value, 200, specifies the order of the task within the task group.
- The content value, sampleCommonTask.jsf, specifies the JavaServer Faces page that displays the task.

### Creating a JavaServer Faces Page for Your Task

A JavaServer Faces page for a task uses the tag sun:commonTask. This tag provides all the capabilities of the Project Woodstock tag webui:jsf:commonTask.

**EXAMPLE 3-8 Example JavaServer Faces Page for a Task**

In the example, the sampleCommonTask.jsf file has the following content:

```xml
<sun:commonTask
    text="Sample Application Page"
    toolTip="Sample Application Page"
    onClick="return admingui.woodstock.commonTaskHandler('treeForm:tree:applications:ejb',
    '#{request.contextPath}/sample/page/testPage.jsf?name=Sample%20Application%20Page');">
</sun:commonTask>
```

**Note** – In the actual file, there is no line break in the onClick attribute value.

This file uses the sun:commonTask tag to specify the task. In your own JavaServer Faces pages, specify the attributes of this tag as follows:

- **text**  The task name that appears on the Common Tasks page.
- **toolTip**  The text that appears when a user places the mouse cursor over the task name.
- **onClick**  Scripting code that is to be executed when a user clicks the task name.

### Adding a Task Group to the Common Tasks Page

You can add a new group of tasks to the Common Tasks page to display the most important tasks for your add-on component. To add a task group, use an integration point of type org.glassfish.admingui:commonTask.
EXAMPLE 3–9  Example Task Group Integration Point

In the example console-config.xml file, the following integration-point element adds a new task group to the Common Tasks page:

```xml
<integration-point
    id="sampleGroup"
    parentId="commonTasksSection"
    type="org.glassfish.admingui:commonTask"
    priority="500"
    content="sampleTaskGroup.jsf"
/>
```

This example specifies the following values:
- The id value, sampleGroup, specifies the integration point ID.
- The parentId value, commonTasksSection, specifies that the task group is to be placed on the Common Tasks page.
- The type value, org.glassfish.admingui:commonTask, specifies the integration point type as a common task.
- The priority value, 500, specifies the order of the task group on the Common Tasks page. The low value places it at the end of the page.
- The content value, sampleTaskGroup.jsf, specifies the JavaServer Faces page that displays the task.

Creating a JavaServer Faces Page for Your Task Group

A JavaServer Faces page for a task group uses the tag `sun:commonTasksGroup`. This tag provides all the capabilities of the Project Woodstock tag `webui:jsf:commonTasksGroup`.

EXAMPLE 3–10  Example JavaServer Faces Page for a Task Group

In the example, the sampleTaskGroup.jsf file has the following content:

```xml
<sun:commonTasksGroup title="My Own Sample Group">
    <sun:commonTask
        text="Go To Sample Resource"
        toolTip="Go To Sample Resource"
        onClick="return admingui.woodstock.commonTaskHandler('form:tree:resources:treeNode1',
            '#{request.contextPath}/sample/page/testPage.jsf?name=Sample%20Resource%20Page');">
    </sun:commonTask>
    <sun:commonTask
        text="Sample Configuration"
        toolTip="Go To Sample Configuration"
        onClick="return admingui.woodstock.commonTaskHandler('form:configuration:sampleConfigNode',
        ```


EXAMPLE 3–10  Example JavaServer Faces Page for a Task Group  (Continued)

'#{request.contextPath}/sample/page/testPage.jsf?name=Sample%20Configuration%20Page'};
</sun:commonTask>
</sun:commonTasksGroup>

Note – In the actual file, there are no line breaks in the onClick attribute values.

This file uses the sun:commonTasksGroup tag to specify the task group, and two
sun:commonTask tags to specify the tasks in the task group. The sun:commonTasksGroup tag has
only one attribute, title, which specifies the name of the task group.

Adding Content to a Page

You can add content for your add-on component to an existing top-level page, such as the
Configuration page or the Resources page. To add content to one of these pages, use an
integration point of type org.glassfish.admingui:configuration or
org.glassfish.admingui:resources.

EXAMPLE 3–11  Example Resources Page Implementation Point

In the example console-config.xml file, the following integration-point element adds new
content to the top-level Resources page:

<integration-point
    id="sampleResourceLink"
    parentId="propSheetSection"
    type="org.glassfish.admingui:resources"
    priority="100"
    content="sampleResourceLink.jsf"
/>

This example specifies the following values:

- The id value, sampleResourceLink, specifies the integration point ID.
- The parentId value, propSheetSection, specifies that the content is to be a section of a
  property sheet on the page.
- The type value, org.glassfish.admingui:resources, specifies the integration point type
  as the Resources page.
To add content to the Configuration page, specify the type value as
org.glassfish.admingui:configuration.

- The priority value, 100, specifies the order of the content on the Resources page. The high value places it at the top of the page.
- The content value, sampleResourceLink.jsf, specifies the JavaServer Faces page that displays the new content on the Resources page.

Another integration-point element in the console-config.xml file places similar content on the Configuration page.

Creating a JavaServer Faces Page for Your Page Content

A JavaServer Faces page for page content often uses the tag sun:property to specify a property on a property sheet. This tag provides all the capabilities of the Project Woodstock tag webui:jsf:property.

EXAMPLE 3-12 Example JavaServer Faces Page for a Resource Page Item

In the example, the sampleResourceLink.jsf file has the following content:

```xml
<sun:property>
  <sun:hyperlink
    toolTip="Sample Resource"
    url="/sample/page/testPage.jsf?name=Sample%20Resource%20Page" >
    <sun:image url="/resource/sample/images/sample.png" />
    <sun:staticText text="Sample Resource" />
  </sun:hyperlink>
</sun:property>

<sun:property>
  <sun:hyperlink
    toolTip="Another"
    url="/sample/page/testPage.jsf?name=Another" >
    <sun:image url="/resource/sample/images/sample.png" />
    <sun:staticText text="Another" />
  </sun:hyperlink>
</sun:property>
```

The file specifies two simple properties on the property sheet, one above the other. Each consists of a sun:hyperlink element (a link to a URL). Within each sun:hyperlink element is nested a sun:image element, specifying an image, and a sun:staticText element, specifying the text to be placed next to the image.
Each `sun:hyperlink` element uses a `tooltip` attribute and a `url` attribute. Each `url` attribute references the `testPage.jsf` file that is used elsewhere in the example, specifying different content for the name parameter.

You can use many other kinds of user interface elements within a `sun:property` element.

### Adding a Page to the Administration Console

Your add-on component may require new configuration tasks. In addition to implementing commands that accomplish these tasks (see Chapter 4, “Extending the asadmin Utility”), you can provide property sheets that enable users to configure your component or to perform tasks such as creating and editing resources for it.

**EXAMPLE 3–13  Example JavaServer Faces Page for a Property Sheet**

Most of the user interface features used in the example reference the file `testPage.jsf` or (for tabs) the file `tabPage.jsf`. Both files are in the `src/main/resources/page/` directory. The `testPage.jsf` file looks like this:

```xml
<!composition template="/templates/default.layout" guiTitle="TEST Sample Page Title">
<!define name="content">
<sun:form id="propertyForm">
<sun:propertySheet id="propertySheet">
  <sun:propertySheetSection id="propertySection">
    <sun:property id="prop1" labelAlign="left" noWrap="true" overlapLabel="false" label="Test Page Name:" >
      <sun:staticText text="$pageSession{pageName}"
        <!beforeCreate
          getRequestValue(key="name" value=>$page(pageName));
          /!>
        </sun:staticText>
    </sun:property>
  </sun:propertySheetSection>
</sun:propertySheet>
</sun:form>
</define>
</composition>
```

The page uses the `composition` directive to specify that the page uses the `default.layout` template and to specify a page title. The page uses additional directives, events, and tags to specify its content.
Adding Internationalization Support

To add internationalization support for your add-on component to the Administration Console, you can place an event and handler like the following at the top of your page:

```xml
:initPage
    setResourceBundle(key="yourI18NKey" bundle="bundle.package.BundleName")
/>
```

Replace the values `yourI18NKey` and `bundle.package.BundleName` with appropriate values for your component.

Changing the Theme or Brand of the Administration Console

To change the theme or brand of the Administration Console for your add-on component, use the integration point type `org.glassfish.admingui:customtheme`. This integration point affects the Cascading Style Sheet (CSS) files and images that are used in the Administration Console.

EXAMPLE 3–14  Example Custom Theme Integration Point

For example, the following integration point specifies a custom theme:

```xml
<integration-point
    id="myOwnBrand"
    type="org.glassfish.admingui:customtheme"
    priority="2"
    content="myOwnBrand.properties"
/>
```

The `priority` attribute works differently when you specify it in a branding integration point from the way it works in other integration points. You can place multiple branding add-on components in the `modules` directory, but only one theme can be applied to the Administration Console. The `priority` attribute determines which theme is used. Specify a value from 1 to 100; the lower the number, the higher the priority. The integration point with the highest priority will be used.

Additional integration point types also affect the theme or brand of the Administration Console:

- `org.glassfish.admingui:masthead`
  Specifies the name and location of the include masthead file, which can be customized with a branding image. This include file will be integrated on the masthead of the Administration Console.
Overview

The previous section described various AdminUI styling concepts and how AdminUI uses them to layout theAdministrationConsole.

In this section, we will look at the AdminUI configuration files that control these concepts. In particular, we will focus on the AdminUI include files. We will see how the AdminUI use include files to load and render content such as a theme.

Branding

The most flexible way of brand integration is through the use of include files. If you need to look at all the branding elements defined in your AdminUI include files, you can use the AdminUI include browser. Example of how you might want to configure this feature is shown below:

EXAMPLE3–15  Example of Branding Integration Points

For example, you might specify the following integration points. The content for each integration point is defined in an include file.

```xml
<integration-point
  id="myOwnBrandMast"
  type="org.glassfish.admingui:masthead"
  priority="80"
  content="branding/masthead.inc"
/>

<integration-point
  id="myOwnBrandLogImg"
  type="org.glassfish.admingui:loginimage"
  priority="80"
  content="branding/loginimage.inc"
/>

<integration-point
  id="myOwnBrandLogFm"
  type="org.glassfish.admingui:loginform"
  priority="80"
  content="branding/loginform.inc"
/>

<integration-point
  id="myOwnBrandVersInf"
  type="org.glassfish.admingui:versioninfo"
  priority="80"
  content="branding/versioninfo.inc"
/>
```

To provide your own CSS and images to modify the global look and feel of the entire application (not just the Administration Console), use the theming feature of Project Woodstock (https://woodstock.dev.java.net/docs/specs/ThemeFS.html). Create a theme JAR file with all the CSS properties and image files that are required by your Woodstock.
component. Use a script provided by the Woodstock project to clone an existing theme, then modify the files and properties as necessary. Once you have created the theme JAR file, place it in the WEB-INF/lib directory of the Administration Console so that the Woodstock theme component will load the theme. In addition, edit the properties file specified by your integration point (MyOwnBrand.properties, for example) to specify the name and version of your theme.

Creating an Integration Point Type

If your add-on component provides new content that you would like other people to extend, you may define your own integration point types. For example, if you add a new page that provides tabs of monitoring information, you might want to allow others to add their own tabs to complement your default tabs. This feature enables your page to behave like the existing Administration Console pages that you or others can extend.

▼ To Create an Integration Point Type

1 Decide on the name of your integration point type.
The integration point type must be a unique identifier. You might use the package name of your integration point, with a meaningful name appended to the end, as in the following example:
org.company.project:myMonitoringTabs

2 After you have an integration point ID, use handlers to insert the integration point implementation(s).
Include code like the following below the place in your JavaServer Faces page where you would like to enable others to add their integration point implementations:
<event>
  <afterCreate
    getUIComponent(clientId="clientId:of:root"
      component=>$attribute{rootComp});
    includeIntegrations(type="org.company.project:myMonitoringTabs"
      root="#{rootComp}");
  />
</event>

Change clientId:of:root to match the clientId of the outermost component in which you want others to be able to add their content (in this example, the tab set is the most likely choice). Also include your integration point ID in place of org.company.project:myMonitoringTabs. If you omit the root argument to includeIntegrations, all components on the entire page can be used for the parentId of the integration points.
To enable others to use this integration point, document it at the GlassFish Integration Point wiki page (http://wiki.glassfish.java.net/Wiki.jsp?page=V3IntegrationPoint).

Document the integration point only if your content is publicly available.

You or others can now provide an integration point that will be integrated into this page.

See Also For more information, see the JSF Templating API documentation (https://jsftemplating.dev.java.net/nonav/javadoc/index.html).
Extending the asadmin Utility

The asadmin utility is a command-line tool for configuring and administering GlassFish Server. Extending the asadmin utility enables you to provide administrative interfaces for an add-on component that are consistent with the interfaces of other GlassFish Server components. A user can run asadmin subcommands either from a command prompt or from a script. For more information about the asadmin utility, see the asadmin(1M) man page.

The following topics are addressed here:

- “About the Administrative Command Infrastructure of GlassFish Server” on page 49
- “Adding an asadmin Subcommand” on page 50
- “Adding Parameters to an asadmin Subcommand” on page 52
- “Adding Message Text Strings to an asadmin Subcommand” on page 56
- “Enabling an asadmin Subcommand to Run” on page 59
- “Setting the Context of an asadmin Subcommand” on page 59
- “Changing the Brand in the GlassFish Server CLI” on page 59
- “Examples of Extending the asadmin Utility” on page 61

About the Administrative Command Infrastructure of GlassFish Server

To enable multiple containers to be independently packaged and loaded, the administrative command infrastructure of GlassFish Server provides the following features:

- **Location independence.** Administration subcommands can be loaded from any add-on component that is known to GlassFish Server.
- **Extensibility.** Administrative subcommands that are available to GlassFish Server are discovered on demand and not obtained from a preset list of subcommands.
- **Support for the HK2 architecture.** Subcommands can use injection to express their dependencies, and extraction to provide results to a user. For more information, see Chapter 2, “Writing HK2 Components.”
Adding an asadmin Subcommand

An asadmin subcommand identifies the operation or task that a user is to perform. Adding an asadmin subcommand enables the user to perform these tasks and operations through the asadmin utility.

The following topics are addressed here:

■ "Representing an asadmin Subcommand as a Java Class” on page 50
■ "Specifying the Name of an asadmin Subcommand” on page 50
■ "Ensuring That an AdminCommand Implementation Is Stateless” on page 51
■ "Example of Adding an asadmin Subcommand” on page 51

Representing an asadmin Subcommand as a Java Class

Each asadmin subcommand that you are adding must be represented as a Java class. To represent an asadmin subcommand as a Java class, write a Java class that implements the org.glassfish.api.admin.AdminCommand interface. Write one class for each subcommand that you are adding. Do not represent multiple asadmin subcommands in a single class.

Annotate the declaration of your implementations of the AdminCommand interface with the org.jvnet.hk2.annotations.Service annotation. The @Service annotation ensures that the following requirements for your implementations are met:

■ The implementations are eligible for resource injection and resource extraction.
■ The implementations are location independent, provided that the component that contains them is made known to the GlassFish Server runtime.

For information about how to make a component known to the GlassFish Server runtime, see “Integrating an Add-On Component With GlassFish Server” on page 120.

Specifying the Name of an asadmin Subcommand

To specify the name of the subcommand, set the name element of the @Service annotation to the name.

Note – Subcommand names are case-sensitive.

Subcommands that are supplied in GlassFish Server distributions typically create, delete, and list objects of a particular type. For consistency with the names of subcommands that are supplied in GlassFish Server distributions, follow these conventions when specifying the name of a subcommand:

■ For subcommands that create an object of a particular type, use the name create-object.
For subcommands that delete an object of a particular type, use the name `delete-object`.

For subcommands that list all objects of a particular type, use the name `list-objects`.

For example, GlassFish Server provides the following subcommands for creating, deleting, and listing HTTP listeners:

- create-http-listener
- delete-http-listener
- list-http-listeners

You must also ensure that the name of your subcommand is unique. To obtain a complete list of the names of all `asadmin` subcommands that are installed, use the `list-commands` subcommand. For a complete list of `asadmin` subcommands that are supplied in GlassFish Server distributions, see *Oracle GlassFish Server 3.0.1 Reference Manual*.

### Ensuring That an AdminCommand Implementation Is Stateless

To enable multiple clients to run a subcommand simultaneously, ensure that the implementation of the AdminCommand interface for the subcommand is stateless. To ensure that the implementation of the AdminCommand interface is stateless, annotate the declaration of your implementation with the `org.jvnet.hk2.annotations Scoped` annotation. In the `@Scoped` annotation, set the scope as follows:

- To instantiate the subcommand for each lookup, set the scope to `PerLookup.class`.
- To instantiate the subcommand only once for each session, set the scope to `Singleton`.

### Example of Adding an asadmin Subcommand

**EXAMPLE 4-1  Adding an asadmin Subcommand**

This example shows the declaration of the class `CreateMyContainer` that represents an `asadmin` subcommand that is named `create-mycontainer`. The subcommand is instantiated for each lookup.

```java
package com.example.mycontainer;

import org.glassfish.api.admin.AdminCommand;
...
import org.jvnet.hk2 annotations.Service;
...
import org.jvnet.hk2 annotations Scoped;
import org.jvnet.hk2 component PerLookup;
```
Adding Parameters to an \texttt{asadmin} Subcommand

The parameters of an \texttt{asadmin} subcommand are the options and operands of the subcommand.

- **Options** control how the \texttt{asadmin} utility performs a subcommand.
- **Operands** are the objects on which a subcommand acts. For example, the operand of the \texttt{start-domain(1)} subcommand is the domain that is to be started.

The following topics are addressed here:

- “Representing a Parameter of an \texttt{asadmin} Subcommand” on page 52
- “Identifying a Parameter of an \texttt{asadmin} Subcommand” on page 53
- “Specifying Whether a Parameter Is an Option or an Operand” on page 53
- “Specifying the Name of an Option” on page 53
- “Specifying the Acceptable Values of a Parameter” on page 54
- “Specifying the Default Value of a Parameter” on page 55
- “Specifying Whether a Parameter Is Required or Optional” on page 55
- “Example of Adding Parameters to an \texttt{asadmin} Subcommand” on page 55

Representing a Parameter of an \texttt{asadmin} Subcommand

Represent each parameter of a subcommand in your implementation as a field or as the property of a JavaBeans specification setter method. Use the property of a setter method for the following reasons:

- To provide data encapsulation for the parameter
- To add code for validating the parameter before the property is set
Identifying a Parameter of an `asadmin` Subcommand

Identifying a parameter of an `asadmin` subcommand enables GlassFish Server to perform the following operations at runtime on the parameter:

- **Validation.** The GlassFish Server determines whether all required parameters are specified and returns an error if any required parameter is omitted.
- **Injection.** Before the subcommand runs, the GlassFish Server injects each parameter into the required field or method before the subcommand is run.
- **Usage message generation.** The GlassFish Server uses reflection to obtain the list of parameters for a subcommand and to generate the usage message from this list.
- **Localized string display.** If the subcommand supports internationalization and if localized strings are available, the GlassFish Server can automatically obtain the localized strings for a subcommand and display them to the user.

To identify a parameter of a subcommand, annotate the declaration of the item that is associated with the parameter with the `org.glassfish.api.Param` annotation. This item is either the field or setter method that is associated with the parameter.

To specify the properties of the parameter, use the elements of the `@Param` annotation as explained in the sections that follow.

Specifying Whether a Parameter Is an Option or an Operand

Whether a parameter is an option or an operand determines how a user must specify the parameter when running the subcommand:

- If the parameter is an option, the user must specify the option with the parameter name.
- If the parameter is an operand, the user may omit the parameter name.

To specify whether a parameter is an option or an operand, set the `primary` element of the `@Param` annotation as follows:

- If the parameter is an option, set the `primary` element to `false`. This value is the default.
- If the parameter is an operand, set the `primary` element to `true`.

Specifying the Name of an Option

The name of an option is the name that a user must type on the command line to specify the option when running the subcommand.
The name of each option that you add in your implementation of an `asadmin` subcommand can have a long form and a short form. When running the subcommand, the user specifies the long form and the short form as follows:

- The short form of an option name has a single dash (`-`) followed by a single character.
- The long form of an option name has two dashes (`--`) followed by an option word.

For example, the short form and the long form of the name of the option for specifying terse output are as follows:

- Short form: `-m`
- Long form: `--monitor`

**Note** – Option names are case-sensitive.

### Specifying the Long Form of an Option Name

To specify the long form of an option name, set the `name` element of the `@Param` annotation to a string that specifies the name. If you do not set this element, the default name depends on how you represent the option.

- If you represent the option as a field, the default name is the field name.
- If you represent the option as the property of a JavaBeans specification setter method, the default name is the property name from the setter method name. For example, if the setter method `setPassword` is associated with an option, the property name and the option name are both `password`.

### Specifying the Short Form of an Option Name

To specify the short form of an option name, set the `shortName` element of the `@Param` annotation to a single character that specifies the short form of the parameter. The user can specify this character instead of the full parameter name, for example `-m` instead of `--monitor`. If you do not set this element, the option has no short form.

### Specifying the Acceptable Values of a Parameter

When a user runs the subcommand, the GlassFish Server validates option arguments and operands against the acceptable values that you specify in your implementation.

To specify the acceptable values of a parameter, set the `acceptableValues` element of the `@Param` annotation to a string that contains a comma-separated list of acceptable values. If you do not set this element, any string of characters is acceptable.
Specifying the Default Value of a Parameter

The default value of a parameter is the value that is applied if a user omits the parameter when running the subcommand.

To specify the default value of a parameter, set the `defaultValue` element of the `@Param` annotation to a string that contains the default value. If you do not set this element, the parameter has no default value.

Specifying Whether a Parameter Is Required or Optional

Whether a parameter is required or optional determines how a subcommand responds if a user omits the parameter when running the subcommand:

- If the parameter is required, the subcommand returns an error.
- If the parameter is optional, the subcommand runs successfully.

To specify whether a parameter is optional or required, set the `optional` element of the `@Param` annotation as follows:

- If the parameter is required, set the `optional` element to `false`. This value is the default.
- If the parameter is optional, set the `optional` element to `true`.

Example of Adding Parameters to an `asadmin` Subcommand

This example shows the code for adding parameters to an `asadmin` subcommand with the properties as shown in the table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Represented As</th>
<th>Acceptable Values</th>
<th>Default Value</th>
<th>Optional or Required</th>
<th>Short Name</th>
<th>Option or Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>--originator</td>
<td>A field that is named originator</td>
<td>Any character string</td>
<td>None defined</td>
<td>Required</td>
<td>None</td>
<td>Option</td>
</tr>
<tr>
<td>--description</td>
<td>A field that is named mycontainerDescription</td>
<td>Any character string</td>
<td>None defined</td>
<td>Optional</td>
<td>None</td>
<td>Option</td>
</tr>
<tr>
<td>--enabled</td>
<td>A field that is named enabled</td>
<td>true or false</td>
<td>false</td>
<td>Optional</td>
<td>None</td>
<td>Option</td>
</tr>
</tbody>
</table>
Adding Message Text Strings to an asadmin Subcommand

A message text string provides useful information to the user about an asadmin subcommand or a parameter.

To provide internationalization support for the text string of a subcommand or parameter, annotate the declaration of the subcommand or parameter with the org.glassfish.api.I18n annotation. The @I18n annotation identifies the resource from the resource bundle that is associated with your implementation.

To add message text strings to an asadmin subcommand, create a plain text file that is named LocalStrings.properties to contain the strings. Define each string on a separate line of the file as follows:

key=string
key
A key that maps the string to a subcommand or a parameter. The format to use for key depends on the target to which the key applies and whether the target is annotated with the @I18n annotation. See the following table.

<table>
<thead>
<tr>
<th>Target</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subcommand or parameter with the @I18n annotation</td>
<td>subcommand-name.command.resource-name</td>
</tr>
<tr>
<td>Subcommand without the @I18n annotation</td>
<td>subcommand-name.command</td>
</tr>
<tr>
<td>Parameter without the @I18n annotation</td>
<td>subcommand-name.command.param-name</td>
</tr>
</tbody>
</table>

The replaceable parts of these formats are as follows:

- **subcommand-name** The name of the subcommand.
- **resource-name** The name of the resource that is specified in the @I18n annotation.
- **param-name** The name of the parameter.

string
A string without quotes that contains the text of the message.

**Note** – To display the message strings to users, you must provide code in your implementation of the execute method to display the text. For more information about implementing the execute method, see “Enabling an asadmin Subcommand to Run” on page 59.

**EXAMPLE 4-3  Adding Message Strings to an asadmin Subcommand**

This example shows the code for adding message strings to the create-mycontainer subcommand as follows:

- The create-mycontainer subcommand is associated with the message Creates a custom container. No internationalization support is provided for this message.
- The --originator parameter is associated with the message The originator of the container. No internationalization support is provided for this message.
- The --description parameter is associated with the message that is contained in the resource mydesc, for which internationalization is provided. This resource contains the message text A description of the container.
- The --enabled parameter is associated with the message Whether the container is enabled or disabled. No internationalization support is provided for this message.
The \texttt{--containername} parameter is associated with the message \texttt{The container name}. No internationalization support is provided for this message.

The addition of the parameters \texttt{originator}, \texttt{description}, \texttt{enabled} and \texttt{containername} to the subcommand is shown in Example 4–2.

```java
package com.example.mycontainer;

import org.glassfish.api.admin.AdminCommand;
...
import org.glassfish.api.I18n;
import org.glassfish.api.Param;
import org.jvnet.hk2.annotations.Service;
...
import org.jvnet.hk2.annotations.Scoped;
import org.jvnet.hk2.component.PerLookup;

/**
 * Sample subcommand
 */
@Service(name="create-mycontainer")
@Scoped(PerLookup.class)
public Class CreateMycontainer implements AdminCommand {
...

@Param
String originator;

@Param(name="description", optional=true)
@I18n("mydesc")
String mycontainerDescription

@Param (acceptableValues="true,false", defaultValue="false", optional=true)
String enabled

@Param(primary=true)
String containername;
...
}
```

The following message text strings are defined in the file \texttt{LocalStrings.properties} for use by the subcommand:
Enabling an asadmin Subcommand to Run

To enable an asadmin subcommand to run, implement the execute method in your implementation of the AdminCommand interface. The declaration of the execute method in your implementation must be as follows.

```java
public void execute(AdminCommandContext context);
```

Pass each parameter of the subcommand as a property to your implementation of the execute method. Set the key of the property to the parameter name and set the value of the property to the parameter's value.

In the body of the execute method, provide the code for performing the operation that the command was designed to perform. For examples, see Example 4-6 and Example 4-7.

Setting the Context of an asadmin Subcommand

The org.glassfish.api.admin.AdminCommandContext class provides the following services to an asadmin subcommand:

- Access to the parameters of the subcommand
- Logging
- Reporting

To set the context of an asadmin subcommand, pass an AdminCommandContext object to the execute method of your implementation.

Changing the Brand in the GlassFish Server CLI

The brand in the GlassFish Server command-line interface (CLI) consists of the product name and release information that are displayed in the following locations:

- In the string that the `version(1)` subcommand displays
- In each entry in the `server.log` file

If you are incorporating GlassFish Server into a new product with an external vendor's own brand name, change the brand in the GlassFish Server CLI.
To change the brand in the GlassFish Server CLI, create an OSGi fragment bundle that contains a plain text file that is named `src/main/resources/BrandingVersion.properties`.

In the `BrandingVersion.properties` file, define the following keyword-value pairs:

- `product_name=product-name`
- `abbrev_product_name=abbrev-product-name`
- `major_version=major-version`
- `minor_version=minor-version`
- `build_id=build-id`
- `version_prefix=version-prefix`
- `version_suffix=version-suffix`

Define each keyword-value pair on a separate line of the file. Each value is a text string without quotes.

The meaning of each keyword-value pair is as follows:

- `product_name=product-name`
  Specifies the full product name without any release information, for example, Oracle GlassFish Server.
- `abbrev_product_name=abbrev-product-name`
  Specifies an abbreviated form of the product name without any release information, for example, GlassFish Server.
- `major_version=major-version`
  Returns the product major version, for example, 3
- `minor_version=minor-version`
  Specifies the product minor version, for example, 0.
- `build_id=build-id`
  Specifies the build version, for example, build 17.
- `version_prefix=version-prefix`
  Specifies a prefix for the product version, for example, v.
- `version_suffix=version-suffix`
  Specifies a suffix for the product version, for example, Beta.

**EXAMPLE 4-4  BrandingVersion.properties File for Changing the Brand in the GlassFish Server CLI**

This example shows the content of the `BrandingVersion.properties` for defining the product name and release information of Oracle GlassFish Server 3.0.1, build 17. The abbreviated product name is glassfish-server.

```
product_name=Oracle GlassFish Server
abbrev_product_name=glassfish-server
major_version=3
```
EXAMPLE 4-4  BrandingVersion.properties File for Changing the Brand in the GlassFish Server CLI (Continued)

minor_version=0.1
build_id=build 17

Examples of Extending the asadmin Utility

EXAMPLE 4-5 asadmin Subcommand With Empty execute Method

This example shows a class that represents the asadmin subcommand create-mycontainer.

The usage statement for this subcommand is as follows:

```
asadmin create-mycontainer --originator any-character-string
[--description any-character-string]
[--enabled {true|false}] any-character-string
```

This subcommand uses injection to specify that a running domain is required.

```
package com.example.mycontainer;

import org.glassfish.api.admin.AdminCommand;
import org.glassfish.api.admin.AdminCommandContext;
import org.glassfish.api.I18n;
import org.glassfish.api.Param;
import org.jvnet.hk2.annotations.Service;
import org.jvnet.hk2.annotations.Inject;
import org.jvnet.hk2.annotations.Scoped;
import org.jvnet.hk2.component.PerLookup;

/**
 * Sample subcommand
 */
@Service(name="create-mycontainer")
@Scoped(PerLookup.class)
public Class CreateMycontainer implements AdminCommand {
    @Inject
    Domain domain;

    @Param
    String originator;

    @Param(name="description", optional=true)
    @I18n("mydesc")
    String mycontainerDescription
```
The following message text strings are defined in the file LocalStrings.properties for use by the subcommand:

create-mycontainer.command=Creates a custom container
create-mycontainer.command.originator=The originator of the container
create-mycontainer.command.mydesc=A description of the container
create-mycontainer.command.enabled=Whether the container is enabled or disabled
create-mycontainer.command.containername=The container name

EXAMPLE 4–6 asadmin Subcommand for Retrieving and Displaying Information

This example shows a class that represents the asadmin subcommand list-runtime-environment. The subcommand determines the operating system or runtime information for GlassFish Server.

The usage statement for this subcommand is as follows:

```
asadmin list-runtime-environment{runtime|os}
```
Examples of Extending the `asadmin` Utility

### Example 4-6

```java
import org.jvnet.hk2.annotations.Service;
import org.jvnet.hk2.annotations.Inject;
import org.jvnet.hk2.annotations.Scoped;
import org.jvnet.hk2.component.PerLookup;

import java.lang.management.ManagementFactory;
import java.lang.management.OperatingSystemMXBean;
import java.lang.management.RuntimeMXBean;

/**
 * Demos asadmin CLI extension
 *
 */
@Service(name="list-runtime-environment")
@Scoped(PerLookup.class)
public class ListRuntimeEnvironmentCommand implements AdminCommand {

// this value can be either runtime or os for our demo
@Param(primary=true)
String inParam;

public void execute(AdminCommandContext context) {

ActionReport report = context.getActionReport();
report.setActionExitCode(ExitCode.SUCCESS);

// If the inParam is 'os' then this subcommand returns operating system 
// info and if the inParam is 'runtime' then it returns runtime info.
// Both of the above are based on mxbeans.

if ("os".equals(inParam)) {
    OperatingSystemMXBean osmb = ManagementFactory.getOperatingSystemMXBean();
    report.setMessage("Your machine operating system name = " + osmb.getName());
} else if ("runtime".equals(inParam)) {
    RuntimeMXBean rtmb = ManagementFactory.getRuntimeMXBean();
    report.setMessage("Your JVM name = " + rtmb.getVmName());
} else {
    report.setActionExitCode(ExitCode.FAILURE);
    report.setMessage("operand should be either 'os' or 'runtime'");
}
}
```
EXAMPLE 4–7 asadmin Subcommand for Updating Configuration Data

This example shows a class that represents the asadmin subcommand configure-greeter-container. The subcommand performs a transaction to update configuration data for a container component. For more information about such transactions, see “Creating a Transaction to Update Configuration Data” on page 97.

The usage statement for this subcommand is as follows:

```
asadmin configure-greeter-container --instances instances [-language language] [-style style]
```

The acceptable values and default value of each option of the subcommand are shown in the following table. The table also indicates whether each option is optional or required.

<table>
<thead>
<tr>
<th>Option</th>
<th>Acceptable Values</th>
<th>Default value</th>
<th>Optional or Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>--instances</td>
<td>An integer in the range 1–10</td>
<td>5</td>
<td>Required</td>
</tr>
<tr>
<td>--language</td>
<td>english, norsk, or francais</td>
<td>norsk</td>
<td>Optional</td>
</tr>
<tr>
<td>--style</td>
<td>formal, casual, or expansive</td>
<td>formal</td>
<td>Optional</td>
</tr>
</tbody>
</table>

Code for the container component is shown in “Example of Adding Container Capabilities” on page 110.

Code that defines the configuration data for the container component is shown in “Examples of Adding Configuration Data for a Component” on page 99.

```java
package org.glassfish.examples.extension.greeter.config;

import org.glassfish.api.admin.AdminCommand;
import org.glassfish.api.admin.AdminCommandContext;
import org.glassfish.api.Param;
import org.jvnet.hk2.annotations.Service;
import org.jvnet.hk2.annotations.Inject;
import org.jvnet.hk2.config.Transactions;
import org.jvnet.hk2.config.ConfigSupport;
import org.jvnet.hk2.config.SingleConfigCode;
import org.jvnet.hk2.config.TransactionFailure;
import java.beans.PropertyVetoException;

@Service(name = "configure-greeter-container")
public class ConfigureGreeterContainerCommand implements AdminCommand {
```
@Param(acceptableValues = "1,2,3,4,5,6,7,8,9,10", defaultValue = "5")
String instances;
@Param(acceptableValues = "english,norsk,francais", defaultValue = "norsk",
optional = true)
String language;
@Param(acceptableValues = "formal,casual,expansive", defaultValue = "formal",
optional = true)
String style;
@Inject
GreeterContainerConfig config;

public void execute(AdminCommandContext adminCommandContext) {
    try {
        ConfigSupport.apply(new SingleConfigCode<GreeterContainerConfig>() {
            public Object run(GreeterContainerConfig greeterContainerConfig)
                throws PropertyVetoException, TransactionFailure {
                greeterContainerConfig.setNumberOfInstances(instances);
                greeterContainerConfig.setLanguage(language);
                greeterContainerConfig.setStyle(style);
                return null;
            }
        }, config);
    } catch (TransactionFailure e) {
    }
}
Monitoring is the process of reviewing the statistics of a system to improve performance or solve problems. By monitoring the state of components and services that are deployed in the GlassFish Server, system administrators can identify performance bottlenecks, predict failures, perform root cause analysis, and ensure that everything is functioning as expected. Monitoring data can also be useful in performance tuning and capacity planning.

An add-on component typically generates statistics that the GlassFish Server can gather at runtime. Adding monitoring capabilities enables an add-on component to provide statistics to GlassFish Server in the same way as components that are supplied in GlassFish Server distributions. As a result, system administrators can use the same administrative interfaces to monitor statistics from any installed GlassFish Server component, regardless of the origin of the component.

The following topics are addressed here:

- “Defining Statistics That Are to Be Monitored” on page 67
- “Updating the Monitorable Object Tree” on page 74
- “Dotted Names for an Add-On Component’s Statistics” on page 79
- ”Example of Adding Monitoring Capabilities” on page 80

Defining Statistics That Are to Be Monitored

At runtime, your add-on component might perform operations that affect the behavior and performance of your system. For example, your component might start a thread of control, receive a request from a service, or request a connection from a connection pool. Monitoring the statistics that are related to these operations helps a system administrator maintain the system.

To provide statistics to GlassFish Server, your component must define events for the operations that generate these statistics. At runtime, your component must send these events when
performing the operations for which the events are defined. For example, to enable the number of received requests to be monitored, a component must send a “request received” event each time that the component receives a request.

A statistic can correspond to single event or to multiple events.

- Counter statistics typically correspond to a single event. For example, to calculate the number of received requests, only one event is required, for example, a “request received” event. Every time that a “request received” event is sent, the number of received requests is increased by 1.

- Timer statistics typically correspond to multiple events. For example, to calculate the time to process a request, two requests, for example, a “request received” event and a “request completed” event.

Defining statistics that are to be monitored involves the following tasks:

- “Defining an Event Provider” on page 68
- “Sending an Event” on page 73

### Defining an Event Provider

An event provider defines the types of events for the operations that generate statistics for an add-on component.

GlassFish Server enables you to define an event provider in the following ways:

- By writing a Java Class. Define an event provider this way if you have access to the source code of the component for which you are defining an event provider.

- By writing an XML fragment. Define an event provider this way if you do not have access to the source code of the component for which you are defining and event provider.

### Defining an Event Provider by Writing a Java Class

To define an event provider, write a Java language class that defines the types of events for the component. Your class is not required to extend any specific class or implement any interfaces.

To identify your class as an event provider, annotate the declaration of the class with the `org.glassfish.external.probe.provider.annotations.ProbeProvider` annotation.

To create a name space for event providers and to uniquely identify an event provider to the monitoring infrastructure of GlassFish Server, set the elements of the `@ProbeProvider` annotation as follows:

```
moduleProviderName
    Your choice of text to identify the application to which the event provider belongs. The value of the `moduleProviderName` element is not required to be unique.
```
For example, for event providers from Oracle GlassFish Server, `moduleProviderName` is `glassfish`.

**moduleName**
Your choice of name for the module for which the event provider is defined. A module provides significant functionality of an application. The value of the `moduleName` element is not required to be unique.

In GlassFish Server, examples of module names are `web-container`, `ejb-container`, `transaction`, and `webservices`.

**probeProviderName**
Your choice of name to identify the event provider. To uniquely identify the event provider, ensure that `probeProviderName` is unique for all event providers in the same module.

In GlassFish Server, examples of event—provider names are `jsp`, `servlet`, and `web-module`.

### Defining Event Types in an Event Provider Class

To define event types in an event provider class, write one method for each type of event that is related to the component. The requirements for each method are as follows:

- The return value of the callback methods must be `void`.
- The method body must be empty. You instantiate the event provider class in the class that invokes the method to send the event. For more information, see “Sending an Event” on page 73.
- To enable the event to be used as an Oracle Solaris DTrace probe, each parameter in the method signature must be a Java language primitive, such as `Integer`, `boolean`, or `String`.

Annotate the declaration of each method with the `org.glassfish.external.probe.provider.annotations.Probe` annotation.

By default, the type of the event is the method name. If you overload a method in your class, you must uniquely identify the event type for each form of the method. To uniquely identify the event type, set the `name` element of the `@Probe` annotation to the name of the event type.

**Note** – You are not required to uniquely identify the event type for methods that are not overloaded.

### Specifying Event Parameters

To enable methods in an event listener to select a subset of values, annotate each parameter in the method signature with the `org.glassfish.external.probe.provider.annotations.ProbeParam` annotation. Set the `value` element of the `@ProbeParam` annotation to the name of the parameter.
Example of Defining an Event Provider by Writing a Java Class

EXAMPLE 5–1  Defining an Event Provider by Writing a Java Class

This example shows the definition of the TxManager class. This class defines events for the start and end of transactions that are performed by a transaction manager.

The methods in this class are as follows:

onTxBegin
  This method sends an event to indicate the start of a transaction. The name of the event type that is associated with this method is begin. A parameter that is named txId is passed to the method.

onCompletion
  This method sends an event to indicate the end of a transaction. The name of the event type that is associated with this method is end. A parameter that is named outcome is passed to the method.

```java
import org.glassfish.external.probe.provider.annotations.Probe;
import org.glassfish.external.probe.provider.annotations.ProbeParam;
import org.glassfish.external.probe.provider.annotations.ProbeProvider;
@ProbeProvider(moduleProviderName="examplecomponent",
moduleName="transaction", probeProviderName="manager")
public class TxManager {

  @Probe("begin")
  public void onTxBegin(
    @ProbeParam("{txId}") String txId
  ){};

  @Probe ("end")
  public void onCompletion(
    @ProbeParam("{outcome}") boolean outcome
  ){};
}
```

Defining an Event Provider by Writing an XML Fragment

To define an event provider, write an extensible markup language (XML) fragment that contains a single probe-provider element.

To create a name space for event providers and to uniquely identify an event provider to the monitoring infrastructure of GlassFish Server, set the attributes of the probe-provider element as follows:

moduleProviderName
  Your choice of text to identify the application to which the event provider belongs. The value of the moduleProviderName attribute is not required to be unique.
For example, for event providers from Oracle GlassFish Server, moduleName is glassfish.

moduleName
Your choice of name for the module for which the event provider is defined. A module provides significant functionality of an application. The value of the moduleName attribute is not required to be unique.

In GlassFish Server, examples of module names are web-container, ejb-container, transaction, and webservices.

probeProviderName
Your choice of name to identify the event provider. To uniquely identify the event provider, ensure that probeProviderName is unique for all event providers in the same module.

In GlassFish Server, examples of event—provider names are jsp, servlet, and web-module.

Within the probe-provider element, add one probe element for each event type that you are defining. To identify the event type, set the name attribute of the probe element to the type.

To define the characteristics of each event type, add the following elements within the probe element:

class
This element contains the fully qualified Java class name of the component that generates the statistics for which you are defining events.

method
This element contains the name of the method that is invoked to generate the statistic.

signature
This element contains the following information about the signature if the method:

return-type (paramater-type-list)

return-type
The return type of the method.

paramater-type-list
A comma-separated list of the types of the parameters in the method signature.

probe-param
The attributes of this element identify the type and the name of a parameter in the method signature. One probe-param element is required for each parameter in the method signature. The probe-param element does not contain any data.

The attributes of the probe-param element are as follows:

type
Specifies the type of the parameter.
name
   Specifies the name of the parameter.

return-param
   The type attribute of this element specifies the return type of the method. The return-param element does not contain any data.

EXAMPLE 5–2  Defining an Event Provider by Writing an XML Fragment

This example defines an event provider for the glassfish:web:jsp component. The Java class of this component is com.sun.enterprise.web.jsp.JspProbeEmitterImpl. The event provider defines one event of type jspLoadedEvent. The signature of the method that is associated with this event is as follows:

    void jspLoaded (String jsp, String hostName)

    <probe-provider moduleProviderName="glassfish" moduleName="web" probeProviderName="jsp">
      <probe name="jspLoadedEvent">
        <class>com.sun.enterprise.web.jsp.JspProbeEmitterImpl</class>
        <method>jspLoaded</method>
        <signature>void (String,String)</signature>
        <probe-param type="String" name="jsp"/>
        <probe-param type="String" name="hostName"/>
        <return-param type="void"/>
      </probe>
    </probe-provider>

Packaging a Component’s Event Providers

Packaging a component’s event providers enables the monitoring infrastructure of GlassFish Server to discover the event providers automatically.

To package a component’s event providers, add an entry to the component’s META-INF/MANIFEST.MF file that identifies all of the component’s event providers. The format of the entry depends on how the event providers are defined:

- If the event providers are defined as Java classes, the entry is a list of the event providers’ class names as follows:

  probe-provider-class-names : class-list

  class-list A comma-separated list of the fully qualified Java class names of the component’s event providers.

- If the event providers are defined as XML fragments, the entry is a list of the paths to the files that contain the XML fragments as follows:

  probe-provider-xml-file-names : path-list
A comma-separated list of the paths to the XML files relative to the root of the archive in the JAR file.

**EXAMPLE 5–3 Manifest Entry for Event Providers That Are Defined as Java Classes**

This example shows the entry in the META-INF/MANIFEST.MF file of a component whose event provider is the `org.glassfish.pluggability.monitoring.ModuleProbeProvider` class.

```xml
probe-provider-class-names: org.glassfish.pluggability.monitoring.ModuleProbeProvider
```

### Sending an Event

At runtime, your add-on component might perform an operation that generates statistics. To provide statistics about the operation to GlassFish Server, your component must send an event of the correct type when performing the operation.

To send an event, instantiate your event provider class and invoke the method of the event provider class for the type of the event. Instantiate the class and invoke the method in the class that represents your add-on component. Ensure that the method is invoked when your component performs the operation for which the event was defined. One way to meet this requirement is to invoke the method for sending the event in the body of the method for performing the operation.

**EXAMPLE 5–4 Sending an Event**

This example shows the code for instantiating the `TxManager` class and invoking the `onTxBegin` method to send an event of type `begin`. This event indicates that a component is about to begin a transaction.

The `TxManager` class is instantiated in the constructor of the `TransactionManagerImpl` class. To ensure that the event is sent at the correct time, the `onTxBegin` method is invoked in the body of the `begin` method, which starts a transaction.

The declaration of the `onTxBegin` method in the event provider interface is shown in Example 5–1.

```java
... public class TransactionManagerImpl {
    ... public void begin() {
        txManager txProvider = new TxManager();
        ...
    }
}`
Updating the Monitorable Object Tree

A monitorable object is a component, subcomponent, or service that can be monitored. GlassFish Server uses a tree structure to track monitorable objects.

Because the tree is dynamic, the tree changes as components of the GlassFish Server instance are added, modified, or removed. Objects are also added to or removed from the tree in response to configuration changes. For example, if monitoring for a component is turned off, the component's monitorable object is removed from the tree.

To enable system administrators to access statistics for all components in the same way, you must provide statistics for an add-on component by updating the monitorable object tree. Statistics for the add-on component are then available through the GlassFish Server administrative commands `get(1)`, `list(1)`, and `set(1)`. These commands locate an object in the tree through the object's dotted name.

For more information about the tree structure of monitorable objects, see "How the Monitoring Tree Structure Works" in Oracle GlassFish Server 3.0.1 Administration Guide.

To make an add-on component a monitorable object, you must add the add-on component to the monitorable object tree.

To update the statistics for an add-on component, you must add the statistics to the monitorable object tree, and create event listeners to gather statistics from events that represent these statistics. At runtime, these listeners must update monitorable objects with statistics that these events contain. The events are sent by event provider classes. For information about how to create event provider classes and send events, see "Defining Statistics That Are to Be Monitored" on page 67.

Updating the monitorable object tree involves the following tasks:

- "Creating Event Listeners" on page 75
- "Representing a Component's Statistics in an Event Listener Class" on page 75
- "Subscribing to Events From Event Provider Classes" on page 77
- "Registering an Event Listener" on page 78

EXAMPLE 5–4  Sending an Event  (Continued)

```java
String txId = createTransactionId();
....
    txProvider.onTxBegin(txId); //emit
}
...
```
Creating Event Listeners

An event listener gathers statistics from events that an event provider sends. To enable an add-on component to gather statistics from events, create listeners to receive events from the event provider. The listener can receive events from the add-on component in which the listener is created and from other components.

To create an event listener, write a Java class to represent the listener. The listener can be any Java object.

An event listener also represents a component's statistics. To enable the Application Server Management Extensions (AMX) to expose the statistics to client applications, annotate the declaration of the class with the `org.glassfish.gmbalManagedObject` annotation.

Ensure that the class that you write meets these requirements:

- The return value of all callback methods in the listener must be `void`.
- Because the methods of your event provider class may be entered by multiple threads, the listener must be thread safe.
- The listener must have the same restrictions as a Java Platform, Enterprise Edition (Java EE) application. For example, the listener cannot open server sockets, or create threads.

A listener is called in the same thread as the event method. As a result, the listener can use thread locals. If the monitored system allows access to thread locals, the listener can access thread locals of the monitored system.

---

**Note** – A listener that is not registered to listen for events is never called by the framework. Therefore, unregistered listeners do not consume any computing resources, such as memory or processor cycles.

Representing a Component's Statistics in an Event Listener Class

Represent each statistic as the property of a JavaBeans specification getter method of your listener class. Methods in the listener class for processing events can then access the property through the getter method. For more information, see “Subscribing to Events From Event Provider Classes” on page 77.

To enable AMX to expose the statistic to client applications, annotate the declaration of the getter method with the `org.glassfish.gmbal.ManagedAttribute` annotation. Set the id element of the `@ManagedAttribute` annotation to the property name all in lowercase.
The data type of the property that represents a statistic must be a class that provides methods for computing the statistic from event data.

The `org.glassfish.external.statistics.impl` package provides the following classes to gather and compute statistics data:

**AverageRangeStatisticImpl**
Provides standard measurements of the lowest and highest values that an attribute has held and the current value of the attribute.

**BoundaryStatisticImpl**
Provides standard measurements of the upper and lower limits of the value of an attribute.

**BoundedRangeStatisticImpl**
Aggregates the attributes of `RangeStatisticImpl` and `BoundaryStatisticImpl` and provides standard measurements of a range that has fixed limits.

**CountStatisticImpl**
Provides standard count measurements.

**RangeStatisticImpl**
Provides standard measurements of the lowest and highest values that an attribute has held and the current value of the attribute.

**StatisticImpl**
Provides performance data.

**StringStatisticImpl**
Provides a string equivalent of a counter statistic.

**TimeStatisticImpl**
Provides standard timing measurements.

**EXAMPLE 5–5**  Representing a Component’s Statistics in an Event Listener Class

This example shows the code for representing the `txcount` statistic in the `TxListener` class.

```java
import org.glassfish.external.statistics.CountStatistic;
import org.glassfish.external.statistics.impl.CountStatisticImpl;
...
import org.glassfish.gmbal.ManagedAttribute;
import org.glassfish.gmbal.ManagedObject;
...

@ManagedObject
public class TxListener {
    private CountStatisticImpl txCount = new CountStatisticImpl("TxCount",
        "count", "Number of completed transactions");
```
Representing a Component's Statistics in an Event Listener Class  (Continued)

```java
...  
@ManagedAttribute(id="txcount")
public CountStatistic getTxCount()
{
    return txCount;
}
}
```

**Subscribing to Events From Event Provider Classes**

To receive events from event provider classes, a listener must subscribe to the events. Subscribing to events also specifies the provider and the type of events that the listener will receive.

To subscribe to events from event provider classes, write one method in your listener class to process each type of event. To specify the provider and the type of event, annotate the method with the `org.glassfish.external.probe.provider.annotations.ProbeListener` annotation. In the `@ProbeListener` annotation, specify the provider and the type as follows:

```
"module-providername:module-name:probe-provider-name:event-type"
```

- **module-providername**
  The application to which the event provider belongs. This parameter must be the value of the `moduleProviderName` element or attribute in the definition of the event provider. See "Defining an Event Provider by Writing a Java Class" on page 68 and "Defining an Event Provider by Writing an XML Fragment" on page 70.

- **module-name**
  The module for which the event provider is defined. This parameter must match the value of the `moduleName` element or attribute in the definition of the event provider. See "Defining an Event Provider by Writing a Java Class" on page 68 and "Defining an Event Provider by Writing an XML Fragment" on page 70.

- **probe-provider-name**
  The name of the event provider. This parameter must match the value of the `probeProviderName` element or attribute in the definition of the event provider. See "Defining an Event Provider by Writing a Java Class" on page 68 and "Defining an Event Provider by Writing an XML Fragment" on page 70.

- **event-type**
  The type of the event. This type is defined in the event provider class. For more information, see "Defining Event Types in an Event Provider Class" on page 69.

Annotate each parameter in the method signature with the `@ProbeParam` annotation. Set the `value` element of the `@ProbeParam` annotation to the name of the parameter.
In the method body, provide the code to update monitoring statistics in response to the event.

**EXAMPLE 5–6** Subscribing to Events From Event Provider Classes

This example shows the code for subscribing to events of type `begin` from the `tx` component. The provider of the component is `TxManager`. The body of the `begin` method contains code to increase the transaction count `txcount` by 1 each time that an event is received.

The definition of the `begin` event type is shown in Example 5–1.

The code for sending `begin` events is shown in Example 5–4.

The instantiation of the `txCount` object is shown in Example 5–5.

```java
...
import org.glassfish.external.probe.provider.annotations.ProbeListener;
import org.glassfish.external.probe.provider.annotations.ProbeParam;
import org.glassfish.gmbal.ManagedObject;
...
@ManagedObject
public class TxListener {
    ...
    @ProbeListener("examplecomponent:transaction:manager:begin")
    public void begin(
        @ProbeParam("txId")
        String txId)
    {
        txCount.increment();
    }
}
```

**Registering an Event Listener**

Registering an event listener enables the listener to receive callbacks from the GlassFish Server event infrastructure. The listener can then collect data from events and update monitorable objects in the object tree. These monitorable objects form the basis for monitoring statistics.

Registering an event listener also makes a component and its statistics monitorable objects by adding statistics for the component to the monitorable object tree.

At runtime, the GlassFish Server event infrastructure registers listeners for an event provider when the event provider is started and unregisters them when the event provider is shut down. As a result, listeners have no dependencies on other components.

To register a listener, invoke the static `org.glassfish.external.probe.provider.StatsProviderManager.register` method in the class that represents your add-on component. In the method invocation, pass the following information as parameters:
Dotted Names for an Add-On Component’s Statistics

The GlassFish Server administrative commands `get(1)`, `list(1)`, and `set(1)` locate a statistic through the dotted name of the statistic. The dotted name of a statistic for an add-on component is determined from the registration of the event listener that defines the statistic as follows:

```
listener-parent-node.path-to-statistic.statistic-name
```

EXAMPLE 5–7  Registering an Event Listener

This example shows the code for registering the event listener `TxListener` for the add-on component that is represented by the class `TransactionManagerImpl`. The statistics that are defined in this listener are associated with the `web-container` configuration element. The listener is registered under the `server/applications` node. The path from this node to the statistics in the event listener is `tx/txapp`.

Code for the constructor of the `TxListener` class is beyond the scope of this example.

```
import org.glassfish.external.probe.provider.StatsProviderManager;
import org.glassfish.external.probe.provider.PluginPoint;

public class TransactionManagerImpl {
  ...
  StatsProviderManager.register("web-container", PluginPoint.APPLICATIONS,
                              "tx/txapp", new TxListener());
  ...
}
listener-parent-node
The node in the monitorable object tree under which the event listener that defines the statistic is registered. This node is passed in the invocation of the register method that registers the event listener. For more information, see “Registering an Event Listener” on page 78.

path-to-statistic
The path through the monitorable object tree from the node under which the event listener is registered down to the statistic in the event listener in which each slash is replaced with a period. This path is passed in the invocation of the register method that registers the event listener. For more information, see “Registering an Event Listener” on page 78.

statistic-name
The name of the statistic. This name is the value of the id element of the @ManagedAttribute annotation on the property that represents the statistic. For more information, see “Representing a Component’s Statistics in an Event Listener Class” on page 75.

For example, the dotted name of the txcount statistic that is defined in Example 5–5 and registered in Example 5–7 is as follows:

server.applications.tx.txapp.txcount

Example of Adding Monitoring Capabilities
This example shows a component that monitors the number of requests that a container receives. The following table provides a cross-reference to the listing of each class or interface in the example.

<table>
<thead>
<tr>
<th>Class or Interface</th>
<th>Listing</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModuleProbeProvider</td>
<td>Example 5–8</td>
</tr>
<tr>
<td>ModuleBootStrap</td>
<td>Example 5–9</td>
</tr>
<tr>
<td>ModuleStatsTelemetry</td>
<td>Example 5–10</td>
</tr>
<tr>
<td>Module</td>
<td>Example 5–11</td>
</tr>
<tr>
<td>ModuleMBean</td>
<td>Example 5–12</td>
</tr>
</tbody>
</table>

EXAMPLE 5–8  Event Provider Class

This example illustrates how to define an event provider as explained in “Defining an Event Provider by Writing a Java Class” on page 68.

The example shows the definition of the ModuleProbeProvider class. The event provider sends events when the request count is increased by 1 or decreased by 1.
This class defines the following methods:

- moduleCountIncrementEvent
- moduleCountDecrementEvent

The name of each method is also the name of the event type that is associated with the method.

A parameter that is named count is passed to each method.

```java
package org.glassfish.pluggability.monitoring;

import org.glassfish.external.probe.provider.annotations.Probe;
import org.glassfish.external.probe.provider.annotations.ProbeParam;
import org.glassfish.external.probe.provider.annotations.ProbeProvider;

/**
 * Monitoring count events
 * Provider interface for module specific probe events.
 *
 */
@ProbeProvider(moduleProviderName = "glassfish", moduleName = "mybeanmodule",
probeProviderName = "mybean")
public class ModuleProbeProvider {

    /**
     * Emits probe event whenever the count is incremented
     */
    @Probe(name = "moduleCountIncrementEvent")
    public void moduleCountIncrementEvent(@ProbeParam("count") Integer count) {
        // Method implementation
    }

    /**
     * Emits probe event whenever the count is decremented
     */
    @Probe(name = "moduleCountDecrementEvent")
    public void moduleCountDecrementEvent(@ProbeParam("count") Integer count) {
        // Method implementation
    }
}
```
EXAMPLE 5–9  Bootstrap Class

This example illustrates how to register an event listener as explained in “Registering an Event Listener” on page 78. The example shows the code for registering an instance of the listener class ModuleStatsTelemetry. This instance is added as a child of the server/applications node of the tree.

```java
package org.glassfish.pluggability.monitoring;

import org.jvnet.hk2.component.PostConstruct;
import org.jvnet.hk2.annotations.Service;
import org.jvnet.hk2.annotations.Scoped;
import org.jvnet.hk2.component.Singleton;
import org.glassfish.external.probe.provider.StatsProviderManager;
import org.glassfish.external.probe.provider.PluginPoint;

/**
 * Monitoring Count Example
 * Bootstrap object for registering probe provider and listener
 *
 */
@Service
@Scoped(Singleton.class)
public class ModuleBootStrap implements PostConstruct {

    @Override
    public void postConstruct() {
        try {
            StatsProviderManager.register(
                "web-container",
                PluginPoint.APPLICATIONS, "myapp", new ModuleStatsTelemetry());
        } catch (Exception e) {
            System.out.println("Caught exception in postconstruct");
            e.printStackTrace();
        }
    }
}

EXAMPLE 5–10  ListenerClass

This example shows how to perform the following tasks:

- “Creating Event Listeners” on page 75. The example shows the code of the ModuleStatsTelemetry listener class.
- “Representing a Component’s Statistics in an Event Listener Class” on page 75. The example shows the code for representing the countmbeancount statistic.
“Subscribing to Events From Event Provider Classes” on page 77. The example shows the code for subscribing to the following types of events from the count component:

- moduleCountIncrementEvent
- moduleCountDecrementEvent

```java
package org.glassfish.pluggability.monitoring;

import org.glassfish.external.statistics.CountStatistic;
import org.glassfish.external.statistics.impl.CountStatisticImpl;
import org.glassfish.external.probe.provider.annotations.ProbeListener;
import org.glassfish.external.probe.provider.annotations.ProbeParam;
import org.glassfish.gmbal.ManagedAttribute;
import org.glassfish.gmbal.ManagedObject;

/**
 * Monitoring counter example
 * Telemetry object which listens to probe events and updates
 * the monitoring stats
 *
 */
@ManagedObject
public class ModuleStatsTelemetry {
    private CountStatisticImpl countMBeanCount = new CountStatisticImpl("CountMBeanCount", "count", "Number of MBeans");

    @ManagedAttribute(id = "countMBeanCount")
    public CountStatistic getCountMBeanCount() {
        return countMBeanCount;
    }

    @ProbeListener("count:example:countapp:moduleCountIncrementEvent")
    public void moduleCountIncrementEvent(@ProbeParam("count") Integer count) {
        countMBeanCount.increment();
    }

    @ProbeListener("count:example:countapp:moduleCountDecrementEvent")
    public void moduleCountDecrementEvent(@ProbeParam("count") Integer count) {
        countMBeanCount.decrement();
    }
}
```
EXAMPLE 5–11  MBean Interface

This example defines the interface for a simple standard MBean that has methods to increase and decrease a counter by 1.

```java
package com.example.count.monitoring;

/**
 * Monitoring counter example
 * ModuleMBean interface
 */
public interface ModuleMBean {
    public Integer getCount();
    public void incrementCount();
    public void decrementCount();
}
```

EXAMPLE 5–12  MBean Implementation

This example illustrates how to send an event as explained in “Sending an Event” on page 73. The example shows code for sending events as follows:

- The `moduleCountIncrementEvent` method is invoked in the body of the `incrementCount` method.
- The `moduleCountDecrementEvent` method is invoked in the body of the `decrementCount` method.

The methods `incrementCount` and `decrementCount` are invoked by an entity that is beyond the scope of this example, for example, JConsole.

```java
package org.glassfish.pluggability.monitoring;

/**
 * Monitoring counter example
 * ModuleMBean implementation
 */
public class Module implements ModuleMBean {
    private int k = 0;
    private ModuleProbeProvider mpp = null;

    @Override
    public Integer getCount() {
        return k;
    }
```
EXAMPLE 5–12 MBean Implementation (Continued)

```java
    @Override
    public void incrementCount() {
        k++;
        if (mpp != null) {
            mpp.moduleCountIncrementEvent(k);
        }
    }

    @Override
    public void decrementCount() {
        k--;
        if (mpp != null) {
            mpp.moduleCountDecrementEvent(k);
        }
    }

    void setProbeProvider(ModuleProbeProvider mpp) {
        this.mpp = mpp;
    }
```
Adding Configuration Data for a Component

The configuration data of a component determines the characteristics and runtime behavior of a component. GlassFish Server provides interfaces to enable an add-on component to store its configuration data in the same way as other GlassFish Server components. These interfaces are similar to interfaces that are defined in Java Specification Request (JSR) 222: Java Architecture for XML Binding (JAXB) 2.0. By using these interfaces to store configuration data, you ensure that the add-on component is fully integrated with GlassFish Server. As a result, administrators can configure an add-on component in the same way as they can configure other GlassFish Server components.

The following topics are addressed here:

- “How GlassFish Server Stores Configuration Data” on page 87
- “Defining an Element” on page 88
- “Defining an Attribute of an Element” on page 89
- “Defining a Subelement” on page 91
- “Validating Configuration Data” on page 92
- “Initializing a Component’s Configuration Data” on page 94
- “Creating a Transaction to Update Configuration Data” on page 97
- “Dotted Names of Configuration Attributes” on page 98
- “Examples of Adding Configuration Data for a Component” on page 99

How GlassFish Server Stores Configuration Data

GlassFish Server stores the configuration data for a domain in a single configuration file that is named domain.xml. This file is an extensible markup language (XML) instance that contains a hierarchy of elements to represent a domain’s configuration. The content model of this XML instance is not defined in a document type definition (DTD) or an XML schema. Instead, the content model is derived from Java language interfaces with appropriate annotations. You use these annotations to add configuration data for a component as explained in the sections that follow.
Defining an Element

For detailed information about the domain.xml file, see Oracle GlassFish Server 3.0.1 Domain File Format Reference.

Defining an Element

An element represents an item of configuration data. For example, to represent the configuration data for a network listener, GlassFish Server defines the network-listener element.

Define an element for each item of configuration data that you are adding.

▼ To Define an Element

1 Define a Java language interface to represent the element.
Define one interface for each element. Do not represent multiple elements in a single interface.
The name that you give to the interface determines name of the element as follows:
   ▪ A change from lowercase to uppercase in the interface name is transformed to the hyphen (-) separator character.
   ▪ The element name is all lowercase.
   For example, to define an interface to represent the wombat-container-config element, give the name WombatContainerConfig to the interface.

2 Specify the parent of the element.
To specify the parent, extend the interface that identifies the parent as shown in the following table.

<table>
<thead>
<tr>
<th>Parent Element</th>
<th>Interface to Extend</th>
</tr>
</thead>
<tbody>
<tr>
<td>config</td>
<td>org.glassfish.api.admin.config.Container</td>
</tr>
<tr>
<td>applications</td>
<td>org.glassfish.api.admin.config.ApplicationName</td>
</tr>
<tr>
<td>Another element that you are defining</td>
<td>org.jvnet.hk2.config.ConfigBeanProxy</td>
</tr>
</tbody>
</table>

3 Annotate the declaration of the interface with the org.jvnet.hk2.config.Configured annotation.

Example 6–1 Declaration of an Interface That Defines an Element

This example shows the declaration of the WombatContainerConfig interface that represents the wombat-container-config element. The parent of this element is the config element.
import org.jvnet.hk2.config.Configured;

import org.glassfish.api.admin.config.Container;

@Configured
public interface WombatContainerConfig extends Container {

...}

More Information  How Interfaces That Are Annotated With @Configured Are Implemented

You are not required to implement any interfaces that you annotate with the @Configured annotation. GlassFish Server implements these interfaces by using the Dom class. GlassFish Server creates a Java Platform, Standard Edition (Java SE) proxy for each Dom object to implement the interface.

Defining an Attribute of an Element

The attributes of an element describe the characteristics of the element. For example, the port attribute of the network-listener element identifies the port number on which the listener listens. For a description of all the attributes of the network-listener element, see “network-listener” in Oracle GlassFish Server 3.0.1 Domain File Format Reference.

Representing an Attribute of an Element

Represent each attribute of an element as the property of a pair of JavaBeans specification getter and setter methods of the interface that defines the element. The component for which the configuration data is being defined can then access the attribute through the getter method. The setter method enables the attribute to be updated.

Specifying the Data Type of an Attribute

The data type of an attribute is the return type of the getter method that is associated with the attribute. To enable the attribute take properties in the form $(property-name) as values, specify the data type as String.

Identifying an Attribute of an Element

To identify an attribute of an element, annotate the declaration of the getter method that is associated with the attribute with the org.jvnet.hk2.config.Attribute annotation.
To specify the properties of the attribute, use the elements of the `@Attribute` annotation as explained in the sections that follow.

**Specifying the Name of an Attribute**

To specify the name of an attribute, set the `value` element of the `@Attribute` annotation to a string that specifies the name. If you do not set this element, the name is derived from the name of the property as follows:

- A change from lowercase to uppercase in the interface name is transformed to the hyphen (`-`) separator character.
- The element name is all lowercase.

For example, if the getter method `getNumberOfInstances` is defined for the property `NumberOfInstances` to represent an attribute, the name of the attribute is `number-of-instances`.

**Specifying the Default Value of an Attribute**

The default value of an attribute is the value that is applied if the attribute is omitted when the element is written to the domain configuration file.

To specify the default value of an attribute, set the `defaultValue` element of the `@Attribute` annotation to a string that contains the default value. If you do not set this element, the parameter has no default value.

**Specifying Whether an Attribute Is Required or Optional**

Whether an attribute is required or optional determines how GlassFish Server responds if the parameter is omitted when the element is written to the domain configuration file:

- If the attribute is required, an error occurs.
- If the attribute is optional, the element is written successfully to the domain configuration file.

To specify whether an attribute is required or optional, set the `required` element of the `@Attribute` annotation as follows:

- If the attribute is required, set the `required` element to `true`.
- If the attribute is optional, set the `required` element to `false`. This value is the default.
Example of Defining an Attribute of an Element

This example defines the attribute number-of-instances. To enable the attribute take properties in the form $\{property-name\}$ as values, the data type of this attribute is String.

```java
import org.jvnet.hk2.config.Attribute;
...
@Attribute
public String getNumberOfInstances();
public void setNumberOfInstances(String instances) throws PropertyVetoException;
```

Defining a Subelement

A subelement represents a containment or ownership relationship. For example, GlassFish Server defines the network-listeners element to contain the configuration data for individual network listeners. The configuration data for an individual network listener is represented by the network-listener element, which is a subelement of network-listeners element.

To Define a Subelement

1. Define an interface to represent the subelement.
   For more information, see "Defining an Element" on page 88.
   The interface that represents the subelement must extend the org.jvnet.hk2.config.ConfigBeanProxy interface.

2. In the interface that defines the parent element, identify the subelement to its parent element.
   a. Represent the subelement as the property of a JavaBeans specification getter or setter method.
   b. Annotate the declaration of the getter or setter method that is associated with the subelement with the org.jvnet.hk2.config.Element annotation.

Example 6–3 Declaring an Interface to Represent a Subelement

This example shows the declaration of the WombatElement interface to represent the wombat-element element.
Validating Configuration Data

Validating configuration data ensures that attribute values that are being set or updated do not violate any constraints that you impose on the data. For example, you might require that an attribute that represents a name is not null, or an integer that represents a port number is within the range of available port numbers. Any attempt to set or update an attribute value that fails validation fails. Any validations that you specify for an attribute are performed when the attribute is initialized and every time the attribute is changed.

To standardize the validation of configuration data, GlassFish Server uses JSR 303: Bean Validation for validating configuration data. JSR 303 defines a metadata model and API for the validation of JavaBeans components.

To validate an attribute of an element, annotate the attribute’s getter method with the annotation in the javax.validation.constraints package that performs the validation that you require. The following table lists commonly used annotations for validating GlassFish Server configuration data. For the complete list of annotations, see the javax.validation.constraints package summary.
Validating Configuration Data

<table>
<thead>
<tr>
<th>TABLE 6–1</th>
<th>Commonly Used Annotations for Validating GlassFish Server Configuration Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation</td>
<td>Annotation</td>
</tr>
<tr>
<td>Not null</td>
<td>javax.validation.constraints.NotNull</td>
</tr>
<tr>
<td>Null</td>
<td>javax.validation.constraints.Null</td>
</tr>
<tr>
<td>Minimum value</td>
<td>javax.validation.constraints.Min</td>
</tr>
<tr>
<td></td>
<td>Set the value element of this annotation to the minimum allowed value.</td>
</tr>
<tr>
<td>Maximum value</td>
<td>javax.validation.constraints.Max</td>
</tr>
<tr>
<td></td>
<td>Set the value element of this annotation to the maximum allowed value.</td>
</tr>
<tr>
<td>Regular expression matching</td>
<td>javax.validation.constraints.Pattern</td>
</tr>
<tr>
<td></td>
<td>Set the regexp element of this annotation to the regular expression that is to be matched.</td>
</tr>
</tbody>
</table>

EXAMPLE 6–5  Specifying a Range of Valid Values for an Integer

This example specifies that the attribute rotation-interval-in-minutes must be a positive integer.

```java
... import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
...
@Min(value=1)
@Max(value=Integer.MAX_VALUE)
String getRotationIntervalInMinutes();
...```

EXAMPLE 6–6  Specifying Regular Expression Matching

This example specifies that the attribute classname must contain only non-whitespace characters.

```java
import javax.validation.constraints.Pattern;
...
@Pattern(regexp="^[\\S]*$")
String getClassname();
...```
Initializing a Component's Configuration Data

To ensure that a component's configuration data is added to the domain.xml file when the component is first instantiated, you must initialize the component's configuration data.

Initializing a component's configuration data involves the following tasks:

- “To Define a Component's Initial Configuration Data” on page 94
- “To Write a Component's Initial Configuration Data to the domain.xml File” on page 94

▼ To Define a Component's Initial Configuration Data

1. Create a plain-text file that contains an XML fragment to represent the configuration data.
   - Ensure that each XML element accurately represents the interface that is defined for the element.
   - Ensure that any subelements that you are initializing are correctly nested.
   - Set attributes of the elements to their required initial values.

2. When you package the component, include the file that contains the XML fragment in the component's JAR file.

   Example 6–7 XML Data Fragment

   This example shows the XML data fragment for adding the wombat-container-config element to the domain.xml file. The wombat-container-config element contains the subelement wombat-element. The attributes of wombat-element are initialized as follows:
   - The foo attribute is set to something.
   - The bar attribute is set to anything.

   `<wombat-container-config>
     <wombat-element foo="something" bar="anything"/>
   </wombat-container-config>

▼ To Write a Component's Initial Configuration Data to the domain.xml File

Add code to write the component's initial configuration data in the class that represents your add-on component. If your add-on component is a container, add this code to the sniffer class. For more information about adding a container, see Chapter 7, “Adding Container Capabilities.”
1 Set an optional dependency on an instance of the class that represents the XML element that you are adding.
   
a. Initialize the instance variable to null.
   
   If the element is not present in the domain.xml file when the add-on component is initialized, the instance variable remains null.

b. Annotate the declaration of the instance variable with the
   org.jvnet.hk2.annotations.Inject annotation.

c. Set the optional element of the @Inject annotation to true.

2 Set a dependency on an instance of the following classes:
   
   ■ org.glassfish.api.admin.config.ConfigParser
     The ConfigParser class provides methods to parse an XML fragment and to write the fragment to the correct location in the domain.xml file.
   
   ■ org.jvnet.hk2.component.Habitat

3 Invoke the parseContainerConfig method of the ConfigParser object only if the instance is null.
   
   If your add-on component is a container, invoke this method within the implementation of the setup method the sniffer class. When the container is first instantiated, GlassFish Server invokes the setup method.

   The test that the instance is null is required to ensure that the configuration data is added only if the data is not already present in the domain.xml file.

   In the invocation of the parseContainerConfig method, pass the following items as parameters:

   ■ The Habitat object on which you set a dependency
   ■ The URL to the file that contains the XML fragment that represents the configuration data

Example 6–8 Writing a Component’s Initial Configuration Data to the domain.xml File

This example writes the XML fragment in the file init.xml to the domain.xml file. The fragment is written only if the domain.xml file does not contain the wombat-container-config-element.

The wombat-container-config element is represented by the WombatContainerConfig interface. An optional dependency is set on an instance of a class that implements WombatContainerConfig.
import org.glassfish.api.admin.config.ConfigParser;
import org.glassfish.examples.extension.config.WombatContainerConfig;

import org.jvnet.hk2.annotations.Inject;
import org.jvnet.hk2.component.Habitat;
import com.sun.enterprise.module.Module;
import java.util.logging.Logger;

import java.io.IOException;
import java.lang.annotation.Annotation;
import java.lang.reflect.Array;
import java.net.URL;

@Inject(optional=true)
WombatContainerConfig config=null;

@Inject
ConfigParser configParser;

@Inject
Habitat habitat;

public Module[] setup(String containerHome, Logger logger) throws IOException {
    if (config==null) {
        URL url = this.getClass().getClassLoader().getResource("init.xml");
        if (url!=null) {
            configParser.parseContainerConfig(habitat, url, WombatContainerConfig.class);
        }
    }
    return null;
}

Example 6–9  domain.xml File After Initialization

This example shows the domain.xml file after the setup method was invoked to add the
wombat-container-config element under the config element.

<domain...>
...
<configs>
    <config name="server-config">
        <wombat-container-config number-of-instances="5">
            <wombat-element foo="something" bar="anything" />
        </wombat-container-config>
    </config>
Creating a Transaction to Update Configuration Data

Creating a transaction to update configuration data enables the data to be updated without the need to specify a dotted name in the `set(1)` subcommand. You can make the transaction available to system administrators in the following ways:

- By adding an `asadmin(1M)` subcommand. If you are adding an `asadmin` subcommand, include the code for the transaction in the body of the subcommand’s `execute` method. For more information, see Chapter 4, “Extending the asadmin Utility.”
- By extending the Administration Console. For more information, see Chapter 3, “Extending the Administration Console.”

▼ To Create a Transaction to Update Configuration Data

Any transaction that you create to modify configuration data must use a configuration change transaction to ensure that the change is atomic, consistent, isolated, and durable (ACID).

1. **Set a dependency on the configuration object to update.**
2. **Define a method to invoke to perform the transaction.**
   a. **Use the generic `SimpleConfigCode` interface to define the method that is to be invoked on a single configuration object, namely:** `SingleConfigCode<T extends ConfigBeanProxy>()`.
   b. **In the body of this method, implement the `run` method of the `SingleConfigCode<T extends ConfigBeanProxy>` interface.**
   c. **In the body of the `run` method, invoke the setter methods that are defined for the attributes that you are setting.**
      These setter methods are defined in the interface that represents the element whose elements you are setting.
3. **Invoke the static method** `org.jvnet.hk2.config.ConfigSupport.ConfigSupport.apply`. In the invocation, pass the following information as parameters to the method:
   a. The code of the method that you defined in Step 2
   b. The configuration object to update, on which you set the dependency in Step 1
Creating a Transaction to Update Configuration Data

This example shows code in the execute method of an asadmin subcommand for updating the number-of-instances element of wombat-container-config element.

```java
import org.glassfish.api.Param;
...
import org.jvnet.hk2.annotations.Inject;
import org.jvnet.hk2.config.Transactions;
import org.jvnet.hk2.config.ConfigSupport;
import org.jvnet.hk2.config.SingleConfigCode;
import org.jvnet.hk2.config.TransactionFailure;
...
@Param
String instances;

@Inject
WombatContainerConfig config;

public void execute(AdminCommandContext adminCommandContext) {
    try {
        ConfigSupport.apply(new SingleConfigCode<WombatContainerConfig>() {
            public Object run(WombatContainerConfig wombatContainerConfig)
             throws PropertyVetoException, TransactionFailure {
                wombatContainerConfig.setNumberOfInstances(instances);
                return null;
            }
        }, config);
    } catch(TransactionFailure e) {
    }
}
...
```

Dotted Names of Configuration Attributes

The GlassFish Server administrative commands `get(1)`, `list(1)`, and `set(1)` locate a configuration attribute through the dotted name of the attribute. The dotted name of an attribute of a configuration element is as follows:

```
cfgs.config.server-config.element-name[subelement-name...].attribute-name
```

**element-name**
The name of an element that contains a subelement or the attribute.

**subelement-name**
The name of a subelement, if any.
attribute-name

The name of the attribute.

For example, the dotted name of the foo attribute of the wombat-element element is as follows:

```
cfg.config.server-config.wombat-container-config.wombat-element.foo
```

**Examples of Adding Configuration Data for a Component**

This example shows the interfaces that define the configuration data for the Greeter Container component. The data is comprised of the following elements:

- A parent element, which is shown in Example 6–11
- A subelement that is contained by the parent element, which is shown in Example 6–12

This example also shows an XML data fragment for initializing an element. See Example 6–13.

Code for the Greeter Container component is shown in “Example of Adding Container Capabilities” on page 110.

Code for an asadmin subcommand that updates the configuration data in this example is shown in Example 4–7.

**EXAMPLE 6–11 Parent Element Definition**

This example shows the definition of the greeter-container-config element. The attributes of the greeter-container-config element are as follows:

- number-of-instances, which must be in the range 1–10.
- language, which must contain only non-whitespace characters.
- style, which must contain only non-whitespace characters.

The greeter-element element is identified as a subelement of the greeter-container-config element. The definition of the greeter-element element is shown in Example 6–12.

```java
package org.glassfish.examples.extension.greeter.config;

import org.jvnet.hk2.config.Configured;
import org.jvnet.hk2.config.Attribute;
import org.jvnet.hk2.config.Element;
import org.glassfish.api.admin.config.Container;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Min;
import javax.validation.constraints.Max;
import java.beans.PropertyVetoException;
```
EXAMPLE 6–11  Parent Element Definition  
(Continued)

@Configured
public interface GreeterContainerConfig extends Container {

    @Attribute
    @Min(value=1)
    @Max (value=10)
    public String getNumberOfInstances();
    public void setNumberOfInstances(String instances) throws PropertyVetoException;

    @Attribute
    @Pattern(regexp = "^[\S]*$")
    public String getLanguage();
    public void setLanguage(String language) throws PropertyVetoException;

    @Attribute
    @Pattern(regexp = "^[\S]*$")
    public String getStyle();
    public void setStyle(String style) throws PropertyVetoException;

    @Element
    public GreeterElement getElement();
    public void setElement(GreeterElement element) throws PropertyVetoException;
}

EXAMPLE 6–12  Subelement Definition

This example shows the definition of the greeter-element element, which is identified as a
subelement of the greeter-container-config element in Example 6–11. The only attribute of
the greeter-element element is greeter-port, which must be in the range 1030–1050.

package org.glassfish.examples.extension.greeter.config;

import org.jvnet.hk2.config.ConfigBeanProxy;
import org.jvnet.hk2.config.Configured;
import org.jvnet.hk2.config.Attribute;
import javax.validation.constraints.Min;
import javax.validation.constraints.Max;
import java.beans.PropertyVetoException;

@Configured
EXAMPLE 6–12 Subelement Definition  (Continued)

public interface GreeterElement extends ConfigBeanProxy {

   @Attribute
   @Min(value=1030)
   @Max (value=1050)
   public String getGreeterPort();
   public void setGreeterPort(String greeterport) throws PropertyVetoException;

}

EXAMPLE 6–13 XML Data Fragment for Initializing the greeter-container-config Element

This example shows the XML data fragment for adding the greeter-container-config element to the domain.xml file. The greeter-container-config element contains the subelement greeter-element.

The attributes of greeter-container-config are initialized as follows:

- The number-of-instances attribute is set to 5.
- The language attribute is set to norsk.
- The style element is set to formal.

The greeter-port attribute of the greeter-element element is set to 1040.

<greeter-container-config number-of-instances="5" language="norsk" style="formal">
   <greeter-element greeter-port="1040"/>
</greeter-container-config>

The definition of the greeter-container-config element is shown in Example 6–11. The definition of the greeter-element element is shown in Example 6–12.
Adding Container Capabilities

Applications run on GlassFish Server in containers. GlassFish Server enables you to create containers that extend or replace the existing containers of GlassFish Server. Adding container capabilities enables you to run new types of applications and to deploy new archive types in GlassFish Server.

The following topics are addressed here:

- “Creating a Container Implementation” on page 103
- “Adding an Archive Type” on page 106
- “Creating Connector Modules” on page 108
- “Example of Adding Container Capabilities” on page 110

Creating a Container Implementation

To implement a container that extends or replaces a service in GlassFish Server, you must create a Java programming language class that includes the following characteristics:

- It is annotated with the org.jvnet.hk2.annotations.Service annotation.
- It implements the org.glassfish.api.container.Container interface.

Marking the Class with the @Service Annotation

Add a com.jvnet.hk2.annotations.Service annotation at the class definition level to identify your class as a service implementation.

@Service
public class MyContainer implements Container {
    ...
}

...
To avoid potential name collisions with other containers, use the fully qualified class name of your container class in the @Service annotation’s name element:

```java
package com.example.containers;
...
@Service(name="com.example.containers.MyContainer")
public class MyContainer implements Container {
  ...
}
```

**Implementing the Container Interface**

The `org.glassfish.api.container.Container` interface is the contract that defines a container implementation. Classes that implement `Container` can extend or replace the functionality in GlassFish Server by allowing applications to be deployed and run within the GlassFish Server runtime.

The `Container` interface consists of two methods, `getDeployer` and `getName`. The `getDeployer` method returns an implementation class of the `org.glassfish.api.deployment.Deployer` interface capable of managing applications that run within this container. The `getName` method returns a human-readable name for the container, and is typically used to display messages belonging to the container.

The `Deployer` interface defines the contract for managing a particular application that runs in the container. It consists of the following methods:

- `getMetaData`
  - Retrieves the metadata used by the `Deployer` instance, and returns an `org.glassfish.api.deployment.MetaData` object.

- `loadMetaData`
  - Loads the metadata associated with an application.

- `prepare`
  - Prepares the application to run in GlassFish Server.

- `load`
  - Loads a previously prepared application to the container.

- `unload`
  - Unloads or stops a previously loaded application.

- `clean`
  - Removes any artifacts generated by an application during the `prepare` phase.

The `DeploymentContext` is the usual context object passed around deployer instances during deployment.
EXAMPLE 7–1  Example Implementation of Container

This example shows a Java programming language class that implements the Container interface and is capable of extending the functionality of GlassFish Server.

```java
package com.example.containers;

@Service(name="com.example.containers.MyContainer")
public class MyContainer implements Container {

    public String getName() {
        return "MyContainer";
    }

    public Class<? extends org.glassfish.api.deployment.Deployer> getDeployer() {
        return MyDeployer.class;
    }
}
```

EXAMPLE 7–2  Example Implementation of Deployer

```java
package com.example.containers;

@Service
public class MyDeployer {

    public MetaData getMetaData() {
        return new MetaData(...);
    }

    public <V> V loadMetaData(Class<V> type, DeploymentContext dc) {
        ...
    }

    public boolean prepare(DeploymentContext dc) {
        // performs any actions needed to allow the application to run,
        // such as generating artifacts
        ...
    }

    public MyApplication load(MyContainer container, DeploymentContext dc) {
        // creates a new instance of an application
        MyApplication myApp = new MyApplication (...);
        ...
        // returns the application instance
        return myApp;
    }

    public void unload(MyApplication myApp, DeploymentContext dc) {
        ...
    }
```
Adding an Archive Type

An archive type is an abstraction of the archive file format. An archive type can be implemented as a plain JAR file, as a directory layout, or a custom type. By default, GlassFish Server recognizes JAR-based and directory-based archive types. A new container might require a new archive type.

There are two sub-interfaces of the `org.glassfish.api.deployment.archive.Archive` interface, `org.glassfish.api.deployment.archive.ReadableArchive` and `org.glassfish.api.deployment.archive.WritableArchive`. Typically developers of new archive types will provide separate implementations of `ReadableArchive` and `WritableArchive`, or a single implementation that implements both `ReadableArchive` and `WritableArchive`.

Implementations of the `ReadableArchive` interface provide read access to an archive type. `ReadableArchive` defines the following methods:

- `getEntry(String name)`: Returns a `java.io.InputStream` for the specified entry name, or null if the entry doesn’t exist.
- `exists(String name)`: Returns a boolean value indicating whether the specified entry name exists.
- `getEntrySize(String name)`: Returns the size of the specified entry as a `long` value.
- `open(URI uri)`: Returns an archive for the given `java.net.URI`.
- `getSubArchive(String name)`: Returns an instance of `ReadableArchive` for the specified sub-archive contained within the parent archive, or null if no such archive exists.
- `exists()`: Returns a boolean value indicating whether this archive exists.
delete()
   Deletes the archive, and returns a boolean value indicating whether the archive has been
   successfully deleted.

renameTo(String name)
   Renames the archive to the specified name, and returns a boolean value indicating whether
   the archive has been successfully renamed.

Implementations of the WritableArchive interface provide write access to the archive type.
WritableArchive defines the following methods:

create(URI uri)
   Creates a new archive with the given path, specified as a java.net.URI.

closeEntry(WritableArchive subArchive)
   Closes the specified sub-archive contained within the parent archive.

closeEntry()
   Closes the current entry.

createSubArchive(String name)
   Creates a new sub-archive in the parent archive with the specified name, and returns it as a
   WritableArchive instance.

putNextEntry(String name)
   Creates a new entry in the archive with the specified name, and returns it as a
   java.io.OutputStream.

**Implementing the ArchiveHandler Interface**

An archive handler is responsible for handling the particular layout of an archive. Java EE
defines a set of archives (WAR, JAR, and RAR, for example), and each of these archives has an
ArchiveHandler instance associated with the archive type.

Each layout should have one handler associated with it. There is no extension point support at
this level; the archive handler’s responsibility is to give access to the classes and resources
packaged in the archive, and it should not contain any container-specific code. The
java.lang.ClassLoader returned by the handler is used by all the containers in which the
application will be deployed.

ArchiveHandler defines the following methods:

getArchiveType()
   Returns the name of the archive type as a String. Typically, this is the archive extension,
   such as jar or war.

getDefaultApplicationName(ReadableArchive archive)
   Returns the default name of the specified archive as a String. Typically this default name is
   the name part of the URI location of the archive.
Creating Connector Modules

Connector modules are small add-on modules that consist of application “sniffers” that associate application types with containers that can run the application type. GlassFish Server connector modules are separate from the associated add-on module that delivers the container implementation to allow GlassFish Server to dynamically install and configure containers on demand.

When a deployment request is received by the GlassFish Server runtime:

1. The current Sniffer implementations are used to determine the application type.
2. Once an application type is found, the runtime looks for a running container associated with that application type. If no running container is found, the runtime attempts to install and configure the container associated with the application type as defined by the Sniffer implementation.
3. The Deployer interface is used to prepare and load the implementation.

Associating File Types With Containers by Using the Sniffer Interface

Containers do not necessarily need to be installed on the local machine for GlassFish Server to recognize the container’s application type. GlassFish Server uses a “sniffer” concept to study the artifacts in a deployment request and to choose the associated container that handles the application type that the user is trying to deploy. To create this association, create a Java programming language class that implements the org.glassfish.api.container.Sniffer interface. This implementation can be as simple as looking for a specific file in the application’s archive (such as the presence of WEB-INF/web.xml), or as complicated as running an annotation
scanner to determine an XML-less archive (such as enterprise bean annotations in a JAR file). A Sniffer implementation must be as small as possible and must not load any of the container’s runtime classes.

A simple version of a Sniffer implementation uses the handles method to check the existence of a file in the archive that denotes the application type (as WEB-INF/web.xml denotes a web application). Once a Sniffer implementation has detected that it can handle the deployment request artifact, GlassFish Server calls the setUp method. The setUp method is responsible for setting up the container, which can involve one or more of the following actions:

- Downloading the container’s runtime (the first time that a container is used)
- Installing the container’s runtime (the first time that a container is used)
- Setting up one or more repositories to access the runtime’s classes (these are implementations of the HK2 com.sun.enterprise.module.Repository interface, such as the com.sun.enterprise.module.impl.DirectoryBasedRepository class)

The setUp method returns an array of the com.sun.enterprise.module.Module objects required by the container.

The Sniffer interface defines the following methods:

handles(ReadableArchive source, ClassLoader loader)
Returns a boolean value indicating whether this Sniffer implementation can handle the specified archive.

getURLPatterns()
Returns a String array containing all URL patterns to apply against the request URL. If a pattern matches, the service method of the associated container is invoked.

getAnnotationTypes()
Returns a list of annotation types recognized by this Sniffer implementation. If an application archive contains one of the returned annotation types, the deployment process invokes the container’s deployers as if the handles method had returned true.

getModuleType()
Returns the module type associated with this Sniffer implementation as a String.

setUp(String containerHome, Logger logger)
Sets up the container libraries so that any dependent bundles from the connector JAR file will be made available to the HK2 runtime. The setUp method returns an array of com.sun.enterprise.module.Module classes, which are definitions of container implementations. GlassFish Server can then load these modules so that it can create an instance of the container’s Deployer or Container implementations when it needs to. The module is locked as long as at least one module is loaded in the associated container.

tearDown()
Removes a container and all associated modules in the HK2 modules subsystem.
getContainerNames()  
Returns a String array containing the Container implementations that this Sniffer implementation enables.

isUserVisible()  
Returns a boolean value indicating whether this Sniffer implementation should be visible to end-users.

getDeploymentConfigurations(final ReadableArchive source)  
Returns a Map<String, String> of deployment configuration names to configurations from this Sniffer implementation for the specified application (the archive source). The names are created by GlassFish Server; the configurations are the names of the files that contain configuration information (for example, WEB-INF/web.xml and possibly WEB-INF/sun-web.xml for a web application). If the getDeploymentConfigurations method encounters errors while searching or reading the specified archive source, it throws a java.io.IOException.

Making Sniffer Implementations Available to the GlassFish Server

Package Sniffer implementation code into modules and install the modules in the as-install/modules directory. GlassFish Server will automatically discover these modules. If an administrator installs connector modules that contain Sniffer implementations while GlassFish Server is running, GlassFish Server will pick them up at the next deployment request.

Example of Adding Container Capabilities

This example shows a custom container and a web client of the container. The example is comprised of the following code:

- Code for the container, which is shown in "Container Component Code" on page 110
- Code for a web client of the container, which is shown in "Web Client Code" on page 117

Code that defines the configuration data for the container component is shown in "Examples of Adding Configuration Data for a Component" on page 99.

Code for an asadmin subcommand that updates the configuration data in this example is shown in Example 4–7.

Container Component Code

The container component code is comprised of the classes and interfaces that are listed in the following table. The table also provides a cross-reference to the listing of each class or interface.
### Annotation to Denote a Container's Component

This example shows the code for defining a component of the Greeter container.

```java
package org.glassfish.examples.extension.greeter;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

/**
 * Simple annotation to denote Greeter's component
 */
@Retention(java.lang.annotation.RetentionPolicy.RUNTIME)
public @interface Greeter {
  /**
   * Name to uniquely identify different greeters
   * @return a good greeter name
   */
  public String name();
}
```

### Application Container Class

This example shows the Java language class GreeterAppContainer, which implements the ApplicationContainer interface.

```java
package org.glassfish.examples.extension.greeter;

import org.glassfish.api.deployment.ApplicationContainer;
import org.glassfish.api.deployment.ApplicationContext;
import org.glassfish.api.deployment.archive.ReadableArchive;
import java.util.List;
import java.util.ArrayList;
```
public class GreeterAppContainer implements ApplicationContainer {

    final GreeterContainer ctr;
    final List<Class> componentClasses = new ArrayList<Class>();

    public GreeterAppContainer(GreeterContainer ctr) {
        this.ctr = ctr;
    }

    void addComponent(Class componentClass) {
        componentClasses.add(componentClass);
    }

    public Object getDescriptor() {
        return null;
    }

    public boolean start(ApplicationContext startupContext) throws Exception {
        for (Class componentClass : componentClasses) {
            try {
                Object component = componentClass.newInstance();
                Greeter greeter = (Greeter)
                    componentClass.getAnnotation(Greeter.class);
                ctr.habitat.addComponent(greeter.name(), component);
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
        }
        return true;
    }

    public boolean stop(ApplicationContext stopContext) {
        for (Class componentClass : componentClasses) {
            ctr.habitat.removeAllByType(componentClass);
        }
        return true;
    }

    public boolean suspend() {
        return false;
    }

    public boolean resume() throws Exception {
        return false;
    }
}
EXAMPLE 7-4  Application Container Class  (Continued)

    public ClassLoader getClassLoader() {
        return null;
    }

EXAMPLE 7-5  Container Class

This example shows the Java language class GreeterContainer, which implements the Container interface.

    package org.glassfish.examples.extension.greeter;

    import org.glassfish.api.container.Container;
    import org.glassfish.api.deployment.Deployer;
    import org.jvnet.hk2.annotations.Service;
    import org.jvnet.hk2.annotations.Inject;
    import org.jvnet.hk2.component.Habitat;

    @Service(name="org.glassfish.examples.extension.GreeterContainer")
    public class GreeterContainer implements Container {
        @Inject
        Habitat habitat;

        public Class<? extends Deployer> getDeployer() {
            return GreeterDeployer.class;
        }

        public String getName() {
            return "greeter";
        }
    }

EXAMPLE 7-6  Deployer Class

This example shows the Java language class GreeterDeployer, which implements the Deployer interface.

    package org.glassfish.examples.extension.greeter;

    import org.glassfish.api.deployment.Deployer;
    import org.glassfish.api.deployment.MetaData;
    import org.glassfish.api.deployment.DeploymentContext;
import org.glassfish.api.deployment.ApplicationContainer;
import org.glassfish.api.deployment.archive.ReadableArchive;
import org.glassfish.api.container.Container;
import org.jvnet.hk2.annotations.Service;
import java.util.Enumeration;

@Service
public class GreeterDeployer
        implements Deployer<GreeterContainer, GreeterAppContainer> {

    public MetaData getMetaData() {
        return null;
    }

    public <V> V loadMetaData(Class<V> type, DeploymentContext context) {
        return null;
    }

    public boolean prepare(DeploymentContext context) {
        return false;
    }

    public GreeterAppContainer load(
            GreeterContainer container, DeploymentContext context) {

        GreeterAppContainer appCtr = new GreeterAppContainer(container);
        ClassLoader cl = context.getClassLoader();

        ReadableArchive ra = context.getOriginalSource();
        Enumeration<String> entries = ra.entries();
        while (entries.hasMoreElements()) {
            String entry = entries.nextElement();
            if (entry.endsWith(".class")) {
                String className = entryToClass(entry);
                try {
                    Class componentClass = cl.loadClass(className);
                    // ensure it is one of our component
                    if (componentClass.isAnnotationPresent(Greeter.class)) {
                        appCtr.addComponent(componentClass);
                    }
                } catch (Exception e) {
                    throw new RuntimeException(e);
                }
            }
        }
        return appCtr;
    }
}
EXAMPLE 7–6  Deployer Class  (Continued)

    }  
    return appCtr;  
}

public void unload(GreeterAppContainer appContainer, DeploymentContext context) {
}

public void clean(DeploymentContext context) {
}

private String entryToClass(String entry) {
    String str = entry.substring("WEB-INF/classes/".length(), entry.length()-6);  
    return str.replaceAll("/", ".");
}

EXAMPLE 7–7  Sniffer Class

This example shows the Java language class GreeterSniffer, which implements the Sniffer interface.

package org.glassfish.examples.extension.greeter;

import org.glassfish.api.container.Sniffer;
import org.glassfish.api.deployment.archive.ReadableArchive;
import org.glassfish.api.admin.config.ConfigParser;
import org.glassfish.examples.extension.greeter.config.GreeterContainerConfig;
import org.jvnet.hk2.annotations.Service;
import org.jvnet.hk2.annotations.Inject;
import org.jvnet.hk2.component.Habitat;
import com.sun.enterprise.module.Module;
import java.util.logging.Logger;
import java.util.Map;
import java.io.IOException;
import java.lang.annotation.Annotation;
import java.lang.reflect.Array;
import java.net.URL;

/**<p>
 * @author Jerome Dochez
 */
@Service(name="greeter")
public class GreeterSniffer implements Sniffer {

    @Inject(optional=true)
    GreeterContainerConfig config=null;

    @Inject
    ConfigParser configParser;

    @Inject
    Habitat habitat;

    public boolean handles(ReadableArchive source, ClassLoader loader) {
        return false;
    }

    public String[] getURLPatterns() {
        return new String[0];
    }

    public Class<? extends Annotation>[] getAnnotationTypes() {
        Class<? extends Annotation>[] a = (Class<? extends Annotation>[]) Array.newInstance(Class.class, 1);
        a[0] = Greeter.class;
        return a;
    }

    public String getModuleType() {
        return "greeter";
    }

    public Module[] setup(String containerHome, Logger logger) throws IOException {
        if (config==null) {
            URL url = this.getClass().getClassLoader().getResource("init.xml");
            if (url!=null) {
                configParser.parseContainerConfig(habitat, url, GreeterContainerConfig.class);
            }
        }
        return null;
    }

    public void tearDown() {
    }

    public String[] getContainersNames() {
        String[] c = { GreeterContainer.class.getName() };
    }
}
Sniffer Class  (Continued)

return c;
}

public boolean isUserVisible() {
    return true;
}

public Map<String, String> getDeploymentConfigurations (ReadableArchive source) throws IOException {
    return null;
}

public String[] getIncompatibleSnifferTypes() {
    return new String[0];
}
}

Web Client Code

The web client code is comprised of the classes and resources that are listed in the following table. The table also provides a cross-reference to the listing of each class or resource.

<table>
<thead>
<tr>
<th>Class or Resource</th>
<th>Listing</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld</td>
<td>Example 7-8</td>
</tr>
<tr>
<td>SimpleGreeter</td>
<td>Example 7-9</td>
</tr>
<tr>
<td>Deployment descriptor</td>
<td>Example 7-10</td>
</tr>
</tbody>
</table>

Container Client Class

import components.SimpleGreeter;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.annotation.WebServlet;
import javax.servlet;
import javax.annotation.Resource;

@WebServlet(urlPatterns={"/hello"})
public class HelloWorld extends HttpServlet {

Example of Adding Container Capabilities
EXAMPLE 7–8 Container Client Class  (Continued)

@Resource(name="Simple")
SimpleGreeter greeter;

public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws IOException, ServletException {

    PrintWriter pw = res.getWriter();
    try {
        pw.println("Injected service is " + greeter);
        if (greeter!=null) {
            pw.println("SimpleService says " + greeter.saySomething());
            pw.println("<br>");
        }
    } catch(Exception e) {
        e.printStackTrace();
    }
}

EXAMPLE 7–9 Component for Container Client

package components;

import org.glassfish.examples.extension.greeter.Greeter;

@Greeter(name="simple")
public class SimpleGreeter {

    public String saySomething() {
        return "Bonjour";
    }
}

EXAMPLE 7–10 Deployment Descriptor for Container Client

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javae..." Line break added for readability
</web-app>
Packaging, Integrating, and Delivering an Add-On Component

Packaging an add-on component enables the component to interact with the GlassFish Server kernel in the same way as other components. Integrating a component with GlassFish Server enables GlassFish Server to discover the component at runtime. If an add-on component is an extension or update to existing installations of GlassFish Server, deliver the component through Update Tool.

The following topics are addressed here:
- “Packaging an Add-On Component” on page 119
- “Integrating an Add-On Component With GlassFish Server” on page 120
- “Delivering an Add-On Component Through Update Tool” on page 120

Packaging an Add-On Component

To enable an add-on component to plug in to the GlassFish Server kernel in the same way as other components, package the component as an OSGi bundle.

A bundle is the unit of deployment in the OSGi module management subsystem. To package a component as an OSGi bundle, package the component’s constituent files in a Java archive (JAR) file with appropriate manifest entries. The manifest entries provide information about the component that is required to enable the component to be plugged into the GlassFish Server kernel, such as:
- Name
- Version
- Dependencies
- Capabilities
Integrating an Add-On Component With GlassFish Server

Integrating an add-on component with GlassFish Server enables GlassFish Server to discover the component at runtime. To integrate an add-on component with GlassFish Server, ensure that the JAR file that contains the component is copied to or installed in the `as-install/modules/` directory.

Delivering an Add-On Component Through Update Tool

If an add-on component is an extension or update to existing installations of GlassFish Server, deliver the component through Update Tool. To deliver an add-on component through Update Tool, create an Image Packaging System (IPS) package to contain the component and add the package to a suitable IPS package repository.

For information about how to create IPS packages, see the IPS best practices document (http://wikis.sun.com/display/IpsBestPractices/).
This appendix provides reference information about integration points, which are described in Chapter 3, "Extending the Administration Console." For complete information about integration points, see http://wiki.glassfish.java.net/Wiki.jsp?page=V3IntegrationPoint.

Define an integration point for each user interface feature in the console-config.xml file for your add-on component.

The following topics are addressed here:

- "Integration Point Attributes" on page 121
- "org.glassfish.admingui:navNode Integration Point" on page 122
- "org.glassfish.admingui:rightPanel Integration Point" on page 123
- "org.glassfish.admingui:rightPanelTitle Integration Point" on page 124
- "org.glassfish.admingui:serverInstTab Integration Point" on page 124
- "org.glassfish.admingui:commonTask Integration Point" on page 125
- "org.glassfish.admingui:configuration Integration Point" on page 125
- "org.glassfish.admingui:resources Integration Point" on page 126
- "org.glassfish.admingui:customtheme Integration Point" on page 126
- "org.glassfish.admingui:masthead Integration Point" on page 127
- "org.glassfish.admingui:loginimage Integration Point" on page 127
- "org.glassfish.admingui:loginform Integration Point" on page 128
- "org.glassfish.admingui:versioninfo Integration Point" on page 128

Integration Point Attributes

For each integration-point element, specify the following attributes. Each attribute takes a string value.

id
An identifier for the integration point. The remaining sections of this appendix do not provide details about specifying this attribute.
**parentId**
The ID of the integration point’s parent.

**type**
The type of the integration point.

**priority**
A numeric value that specifies the relative ordering of integration points with the same parentId. A lower number specifies a higher priority (for example, 100 represents a higher priority than 400). You may need to experiment in order to place the integration point where you want it. This attribute is optional.

**content**
A relative path to the JavaServer Faces page that contains the content to be integrated. Typically, the file contains a JavaServer Faces code fragment that is incorporated into a page. The code fragment often specifies a link to another JavaServer Faces page that appears when a user clicks the link.

---

**org.glassfish.admingui:navNode Integration Point**

Use an `org.glassfish.admingui:navNode` integration point to insert a node in the Administration Console navigation tree. Specify the attributes and their content as follows.

**type**
`org.glassfish.admingui:navNode`, the left-hand navigation tree

**parentId**
The id value of the navNode that is the parent for this node. The parentId can be any of the following:

- **tree**
  The root node of the entire navigation tree. Use this value to place your node at the top level of the tree. You can then use the id of this node to create additional nodes beneath it.

  **registration**
  The Registration node

  **applicationServer**
  The GlassFish Server node

  **applications**
  The Applications node

  **resources**
  The Resources node

  **configuration**
  The Configuration node
webContainer
   The Web Container node under the Configuration node

httpService
   The HTTP Service node under the Configuration node

Note – The webContainer and httpService nodes are available only if you installed the web
container module for the Administration Console (the console-web-gui.jar OSGi
bundle).

priority
   A numeric value that specifies the relative ordering of the node on the tree, whether at the
top level or under another node.

content
   A relative path to the JavaServer Faces page that contains the content to be integrated, or a
URL to an external resource that returns the appropriate data structure for inclusion.

For an example, see Example 3–2.

org.glassfish.admingui:rightPanel Integration Point

Use an org.glassfish.admingui:rightPanel integration point to specify content for the
right frame of the Administration Console. Specify the attributes and their content as follows.

type
   org.glassfish.admingui:rightPanel

parentId
   None.

priority
   A numeric value that specifies the relative ordering. If multiple plug-ins specify content for
the right frame, the one with greater priority will take precedence.

content
   A path relative to the root of the plug-in JAR file to a file containing the content for the right
panel. Alternatively, it may contain a full URL which will deliver the content for the right
panel.
org.glassfish.admingui:rightPanelTitle Integration Point

Use an org.glassfish.admingui:rightPanel integration point to specify the title for the right frame of the Administration Console. Specify the attributes and their content as follows.

- **type**
  - org.glassfish.admingui:rightPanelTitle

- **parentId**
  - None.

- **priority**
  - A numeric value that specifies the relative ordering. If multiple plug-ins specify content for the right frame, the one with greater priority will take precedence.

- **content**
  - Specifies the title to display at the top of the right panel.

org.glassfish.admingui:serverInstTab Integration Point

Use an org.glassfish.admingui:serverInstTab integration point to place an additional tab on the GlassFish Server page of the Administration Console. Specify the attributes and their content as follows.

- **type**
  - org.glassfish.admingui:serverInstTab

- **parentId**
  - The id value of the tab set that is the parent for this tab. For a top-level tab on this page, this value is serverInstTabs, the tab set that contains the general information property pages for GlassFish Server.

  For a sub-tab, the value is the id value for the parent tab.

- **priority**
  - A numeric value that specifies the relative ordering of the tab on the page, whether at the top level or under another tab.

- **content**
  - A relative path to the JavaServer Faces page that contains the content to be integrated.

  When you use this integration point, your JavaServer Faces page must call the setSessionAttribute handler for the command event to set the session variable of the serverInstTabs tab set to the id value of your tab. For example, the file may have the following content:
<sun:tab id="sampleTab" immediate="true" text="Sample First Tab">
  <!command
    setSessionAttribute(key="serverInstTabs" value="sampleTab");
    gf.redirect(page="#{request.contextPath}/page/tabPage.jsf?name=Sample%20First%20Tab");
  />
</sun:tab>

The id of the sun:tab custom tag must be the same as the value argument of the setSessionAttribute handler.

For examples, see Example 3–4 and Example 3–5.

**org.glassfish.admingui:commonTask Integration Point**

Use an org.glassfish.admingui:commonTask integration point to place a new task or task group on the Common Tasks page of the Administration Console. Specify the attributes and their content as follows.

- **type**
  - org.glassfish.admingui:commonTask

- **parentId**
  - If you are adding a task group, the id value of the Common Tasks page, which is commonTasksSection.
  - If you are adding a single task, the id value of the task group that is the parent for this tab, such as deployment (for the Deployment group).

- **priority**
  - A numeric value that specifies the relative ordering of the tab on the page, whether at the top level or under another tab.

- **content**
  - A relative path to the JavaServer Faces page that contains the content to be integrated.

For examples, see Example 3–7 and Example 3–9.

**org.glassfish.admingui:configuration Integration Point**

Use an org.glassfish.admingui:configuration integration point to add a component to the Configuration page of the Administration Console. Typically, you add a link to the property sheet section of this page. Specify the attributes and their content as follows.

- **type**
  - org.glassfish.admingui:configuration
org.glassfish.admingui:resources Integration Point

Use an org.glassfish.admingui:resources integration point to add a component to the Resources page of the Administration Console. Typically, you add a link to the property sheet section of this page. Specify the attributes and their content as follows.

type
org.glassfish.admingui:resources

parentId
The id value of the property sheet for the Resources page. This value is propSheetSection, the section that contains the property definitions for the Resources page.

priority
A numeric value that specifies the relative ordering of the item on the Resources page.

content
A relative path to the JavaServer Faces page that contains the content to be integrated.

For an example, see Example 3–11.

org.glassfish.admingui:customtheme Integration Point

Use an org.glassfish.admingui:customtheme integration point to add your own branding to the Administration Console. Specify the attributes and their content as follows. Do not specify a parent Id attribute for this integration point.

type
org.glassfish.admingui:customtheme

priority
A numeric value that specifies the relative ordering of the item in comparison to other themes. This value must be between 1 and 100. The theme with the smallest number is used first.
The name of the properties file that contains the key/value pairs that will be used to access your theme JAR file. You must specify the following keys:

```
com.sun.webui.theme.DEFAULT_THEME
```
Specifies the theme name for the theme that this application may depend on.

```
com.sun.webui.theme.DEFAULT_THEME_VERSION
```
Specifies the theme version this application may depend on.

For example, the properties file for the default Administration Console brand contains the following:

```
com.sun.webui.theme.DEFAULT_THEME=suntheme
com.sun.webui.theme.DEFAULT_THEME_VERSION=4.3
```

For an example, see Example 3–14.

**org.glassfish.admingui:masthead** Integration Point

Use an **org.glassfish.admingui:masthead** integration point to specify the name and location of the include masthead file, which can be customized with a branding image. This include file will be integrated on the masthead of the Administration Console. Specify the attributes and their content as follows. Do not specify a parentId attribute for this integration point.

**type**

```
org.glassfish.admingui:masthead
```

**priority**

A numeric value that specifies the relative ordering of the item in comparison to other items of this type. This value must be between 1 and 100. The theme with the smallest number is used first.

**content**

A file that contains the content, typically a file that is included in a JavaServer Faces page.

For an example, see Example 3–15.

**org.glassfish.admingui:loginimage** Integration Point

Use an **org.glassfish.admingui:loginimage** integration point to specify the name and location of the include file containing the branding login image code that will be integrated with the login page of the Administration Console. Specify the attributes and their content as follows. Do not specify a parentId attribute for this integration point.

**type**

```
org.glassfish.admingui:loginimage
```
**org.glassfish.admingui:loginform Integration Point**

parentId
None; a login image does not have a parent ID.

priority
A numeric value that specifies the relative ordering of the item in comparison to other items of this type. This value must be between 1 and 100. The theme with the smallest number is used first.

content
A file that contains the content, typically a file that is included in a JavaServer Faces page.

For an example, see Example 3–15.

**org.glassfish.admingui:versioninfo Integration Point**

Use an org.glassfish.admingui:versioninfo integration point to specify the name and location of the include file containing the branding image that will be integrated with the content of the version popup window. Specify the attributes and their content as follows. Do not specify a parentId attribute for this integration point.

type
org.glassfish.admingui:versioninfo

priority
A numeric value that specifies the relative ordering of the item in comparison to other items of this type. This value must be between 1 and 100. The theme with the smallest number is used first.

content
A file that contains the content, typically a file that is included in a JavaServer Faces page.

For an example, see Example 3–15.
priority
A numeric value that specifies the relative ordering of the item in comparison to other items of this type. This value must be between 1 and 100. The theme with the smallest number is used first.

content
A file that contains the content, typically a file that is included in a JavaServer Faces page.

For an example, see Example 3–15.
Index

A
abbrev_product_name keyword, 60
acceptableValues element, @Param annotation, 54
add-on components
delivering, 120
integrating, 120
overview, 15-16
packaging, 119
specifying ID values, 32-33
AdminCommand interface, 50
execute method, 59
AdminCommandContext class, 59
Administration Console
adding content to pages, 42-44
adding functionality to, 33-44
adding internationalization support, 45
adding nodes to navigation tree, 34-36
adding pages to, 44
adding tabs and tab sets to pages, 36-39
adding task groups to Common Tasks page, 40-42
adding tasks to Common Tasks page, 39-40
architecture, 30-31
changing theme or brand of, 45-47
extending, 29-48
annotations
@Param, 53
@Attribute, 89-90
@Configured, 88
@Element, 91
@I18n, 56
@ManagedAttribute, 75
@ManagedObject, 75
annotations (Continued)
@Max, 93
@Min, 93
@Pattern, 93
@Probe, 69
@ProbeListener, 77-78
@ProbeName, 69
@ProbeParam, 69, 77
@ProbeProvider, 68-70
@Scoped, 51
@Service, 50
Apache Felix OSGi framework, 16
Apache Maven, See Maven
ApplicationName interface, 88
apply method, 97
Archive interface, 106-108
archive types, See containers: archive types
ArchiveHandler interface, 107-108
asadmin subcommand
branding, 59-61
context, 59
asadmin subcommands
adding, 50
default parameter values, 55
error messages, 56-59
internationalization, 56
naming, 50
operands, 52-56
options, 52-56
parameters, 52-56
running, 59
strings, 56-59
asadmin subcommands (Continued)
text, 56-59
validation of parameters, 54
@Attribute annotation, 89-90
attributes, defining, 89-91
AverageRangeStatisticImpl class, 76

B
BoundaryStatisticImpl class, 76
BoundedRangeStatisticImpl class, 76
brand of Administration Console, changing, 45-47
branding, asadmin subcommand, 59-61
BrandingVersion.properties file, 59-61
build_id keyword, 60

C
callbacks, 78-79
class element, 71
class loaders, 107-108
classes
AdminCommandContext, 59
AverageRangeStatisticImpl, 76
BoundaryStatisticImpl, 76
BoundedRangeStatisticImpl, 76
ConfigParser, 95
ConfigSupport, 97
CountStatisticImpl, 76
ProbeClientMediator, 78-79
RangeStatisticImpl, 76
stateless, 51
StatisticImpl, 76
StringStatisticImpl, 76
TimeStatisticImpl, 76
CLI (command-line interface), branding, 59-61
command-line interface (CLI), branding, 59-61
ComponentManager class, 23
components, instantiating, 23
ConfigBeanProxy interface, 88
ConfigParser class, 95
ConfigSupport class, 97
configuration data
attributes, 89-91
dotted names, 98-99
initializing, 94-97
storage of, 87-88
updating, 97-98
validating, 92-93
XML elements, 88-89
XML representation, 87-88
XML subelements, 91-92
@Configured annotation, 88
connector modules, See containers: connector modules
Console Add-On Component Service, 30-31
console-config.xml file, 32, 121-129
console-config element, 32-33
integration-point element, 32-33
core providers, 30-31
implementing, 30-31
ContainerProvider interface, 30-31
Container interface, 88, 103-106
containers
archive types, 106-108, 108-110, 110
collector modules, 108-110
developing, 103-118
examples, 105
implementing, 104-106
loading, 108-110, 110
naming, 103-104
containment relationships, representation as
subelements, 91
content attribute, integration-point
element, 121-122
default values, asadmin subcommand parameters, 55
defaultValue element
@Attribute annotation, 90
@Param annotation, 55
delivering, add-on components, 120
Deployer interface, 104-106, 108-110
deployment, examples, 105-106
Dom class, 89
domain.xml file
description, 87-88
updating, 97-98
writing initial data to, 94-97
dotted names
configuration data, 98-99
statistics, 79-80

E
@Element annotation, 91
elements, XML, 70-72
elements, XML
attributes, 89-91
defining, 88-89
error messages, asadmin subcommands, 56-59
event listeners, creating, 75
event providers, defining, 68-73
events
defining, 68-73
listeners, 75, 78-79
receiving, 77-78
sending, 73-74
statistics monitoring, 67
subscribing, 77-78
examples
containers, 105
deployers, 105-106
execute method, AdminCommand interface, 59
extensible markup language (XML), 70-72
fragment for initializing configuration, 94
representation of configuration data as, 87-88
@Extract annotation, 25
extraction, 25

F
Felix OSGi framework, 16
fields, representation of subcommand parameters
as, 52
files, BrandingVersion.properties, 59-61

G
getConfiguration method, 30-31

H
Habitat, class, 25
HK2, scopes, 22-23
HK2 (Hundred-Kilobyte Kernel)
architecture, 21
overview, 16
services, 22
Hundred-Kilobyte Kernel (HK2)
architecture, 21, 24-26
extraction, 25
injection, 24-25
instantiating, 23
instantiation, 26
inversion of control, 24-26
lifecycle, 23-24
overview, 16
runtime, 22-24
services, 22

I
@I18n annotation, 56
id attribute, integration-point element, 121-122
id element, @ManagedAttribute annotation, 75
Image Packaging System (IPS), 120
initializing, configuration data, 94-97
@Inject annotation, 24-25, 26
injection, 24-25
instantiation, 26
integrating, add-on components, 120
integration-point element, 32-33
attributes, 33-44, 121-122
integration points, 30-31, 32
attributes, 121-122
creating types, 47-48
integration points (Continued)
org.glassfish.admingui:commonTask, 39-40, 
40-42, 125
org.glassfish.admingui:configuration, 42-44, 125-126
org.glassfish.admingui:customtheme, 45-47, 126-127
org.glassfish.admingui:loginform, 128
org.glassfish.admingui:loginimage, 127-128
org.glassfish.admingui:navNode, 122-123
org.glassfish.admingui:resources, 42-44, 126
org.glassfish.admingui:rightPanel, 123
org.glassfish.admingui:rightPanelTitle, 124
org.glassfish.admingui:serverInstTab, 36-39, 124-125
org.glassfish.admingui:treeNode, 34-36
org.glassfish.admingui:versioninfo, 128-129
reference, 121-129
interfaces
AdminCommand, 50, 59
ApplicationName, 88
ConfigBeanProxy, 88
Container, 88
SimpleConfigCode, 97
internationalization
asadmin subcommands, 56
providing for add-on components, 45
IPS (Image Packaging System), 120

J
JSFTemplating project, templates, 31
JSFTemplating tags
sun:commonTask, 40
sun:commonTasksGroup, 41-42
sun:property, 43-44
sun:tab, 38-39
sun:treeNode, 35-36

L
lifecycle interfaces, 23-24
listeners
creating, 75
registering, 78-79
long form, option names, 54

M
major_version keyword, 60
@ManagedAttribute annotation, 75
@ManagedObject annotation, 75
Maven, 27-28
@Max annotation, 93
method element, 71
methods
AdminCommand, 59
apply, 97
parseContainerConfig, 95
registerListener, 78-79
run, 97
@Min annotation, 93
minor_version keyword, 60
modular architecture, GlassFish Server, 15-16
moduleName element, @ProbeProvider
annotation, 68-70
moduleProviderName element, @ProbeProvider
annotation, 68-70
monitorable objects
adding to tree, 78-79
overview, 74-79
monitoring
adding to components, 67-85
dotted names, 79-80

N
name element
@Param annotation, 54
@Probe annotation, 69
@Service annotation, 50
names
asadmin subcommands, 50
attributes, 90
navigation nodes, adding to Administration Console, 34-36
@NotNull annotation, 93
annotations
  @NotNull, 93
  @Null, 93
  @Null annotation, 93

operands, asadmin subcommands, 52-56
optional element, @Param annotation, 55
options
  asadmin subcommands, 52-56
  long names, 54
  short names, 54
org.glassfish.admingui:commonTask integration point type, 39-40, 40-42, 125
org.glassfish.admingui:configuration integration point type, 42-44, 125-126
org.glassfish.admingui:customtheme integration point type, 45-47, 126-127
org.glassfish.admingui:loginform integration point type, 128
org.glassfish.admingui:loginimage integration point type, 127-128
org.glassfish.admingui:masthead integration point type, 127
org.glassfish.admingui:navNode integration point type, 122-123
org.glassfish.admingui:resources integration point type, 42-44, 126
org.glassfish.admingui:rightPanel integration point type, 123
org.glassfish.admingui:rightPanelTitle integration point type, 124
org.glassfish.admingui:serverInstTab integration point type, 36-39, 124-125
org.glassfish.admingui:treeNode integration point type, 34-36
org.glassfish.admingui:versioninfo integration point type, 128-129
OSGi Alliance, 16
overloaded methods, 69

overview
  add-on components, 15-16
  extensibility, 15-16
  ownership relationships, representation as subelements, 91

P
packaging
  add-on components, 119
  event providers, 72-73
pages, adding to Administration Console, 44
pages of Administration Console, adding content to, 42-44
@Param annotation, 53
parameters
  asadmin subcommands, 52-56
  default values, 55
  events, 69, 77
  validation of, 54
parentId attribute, integration-point element, 121-122
parseContainerConfig method, 95
@Pattern annotation, 93
plug-ins, See add-on components
PostConstruct interface, 23-24
PreDestroy interface, 23-24
primary element, @Param annotation, 53
priority attribute, integration-point element, 121-122
@Probe annotation, 69
probe element, 71
probe-param element, 71
probe-provider element, 70, 71
ProbeClientMediator class, 78-79
@ProbeListener annotation, 77-78
@ProbeName annotation, 69
@ProbeParam annotation, 69, 77
@ProbeProvider annotation, 68-70
probeProviderName element, @ProbeProvider annotation, 68-70
product name, defining, 59-61
product_name keyword, 60

Index

135
properties, representation of subcommand parameters as, 52
proxies, Java SE, 89

R
RangeStatisticImpl class, 76
ReadableArchive interface, 106-108
receiving, events, 77-78
regexp element, @Pattern annotation, 93
registering, event listeners, 78-79
registerListener method, 78-79
regular expressions, 93
release information, defining, 59-61
required element, @Attribute annotation, 90
return-param element, 72
run method, 97
running, asadmin subcommands, 59

S
@Scoped annotation, 22-23
@Scoped annotation, 51
sending, events, 73-74
@Service annotation, 22, 27-28, 103-106, 105
@Service annotation, 50
setter methods, subcommand parameters and, 52
short form, option names, 54
shortName element, @Param annotation, 54
signature element, 71
SimpleConfigCode interface, 97
singleton, 22-23
Sniffer interface, 108-110, 110
sniffers, 108-110, 110
stateless classes, 51
StatisticImpl class, 76
statistics
adding to components, 67-85
dotted names, 79-80
strings, asadmin subcommands, 56-59
StringStatisticImpl class, 76
subelements, XML, defining, 91-92
subscribing, to events, 77-78
sun:commonTask tag, 40
sun:commonTasksGroup tag, 41-42
sun:property tag, 43-44
sun:tab tag, 38-39
sun:treeNode tag, 35-36

T
tabs and tab sets, adding to Administration Console, 36-39
task groups, adding to Administration Console, 40-42
tasks, adding to Administration Console, 39-40
Templating for JavaServer Faces Technology, See JSF Templating project
text, asadmin subcommands, 56-59
theme of Administration Console, changing, 45-47
TimeStatisticImpl class, 76
transactions, updates to configuration data, 97-98
tree
adding objects to, 78-79
monitorable objects, 74-79
type attribute, integration-point element, 121-122

U
Update Tool, 120
updating, configuration data, 97-98

V
validating, configuration data, 92-93
validation, asadmin subcommand parameters, 54
value element
(Attribute annotation, 90
@Max annotation, 93
@Min annotation, 93
@ProbeParam annotation, 69, 77
version_prefix keyword, 60
version_suffix keyword, 60
W
WritableArchive interface, 106-108

X
XML (extensible markup language), 70-72
  fragment for initializing configuration, 94
  representation of configuration data as, 87-88