Eclipse GlassFish Add-On Component Development Guide, Release 7

Eclipse GlassFish

Add-On Component Development Guide

Release 7

Contributed 2018 - 2024

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

This document is for software developers who are developing add-on components for Eclipse GlassFish. This document assumes that the developers are working with a distribution of Eclipse GlassFish. 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.

Preface

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

This document is for software developers who are developing add-on components for Eclipse GlassFish. This document assumes that the developers are working with a distribution of Eclipse GlassFish. 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 Eclipse GlassFish (Eclipse GlassFish) documentation set.

Eclipse GlassFish 7 is developed through the GlassFish project open-source community at https://github.com/eclipse-ee4j/glassfish. The GlassFish project provides a structured process for developing the Eclipse GlassFish platform that makes the new features of the Jakarta EE platform available faster, while maintaining the most important feature of Jakarta EE: compatibility. It enables Java developers to access the Eclipse GlassFish source code and to contribute to the development of the Eclipse GlassFish.

The following topics are addressed here:

Eclipse GlassFish Documentation Set

The Eclipse GlassFish documentation set describes deployment planning and system installation. For an introduction to Eclipse GlassFish, refer to the books in the order in which they are listed in the following table.

Book Title	Description
Release Notes	Provides late-breaking information about the software and the documentation and includes a comprehensive, table-based summary of the supported hardware, operating system, Java Development Kit (JDK), and database drivers.
Quick Start Guide	Explains how to get started with the Eclipse GlassFish product.
Installation Guide	Explains how to install the software and its components.

Book Title	Description
Upgrade Guide	Explains how to upgrade to the latest version of Eclipse GlassFish. This guide also describes differences between adjacent product releases and configuration options that can result in incompatibility with the product specifications.
Deployment Planning Guide	Explains how to build a production deployment of Eclipse GlassFish that meets the requirements of your system and enterprise.
Administration Guide	Explains how to configure, monitor, and manage Eclipse GlassFish 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.
Security Guide	Provides instructions for configuring and administering Eclipse GlassFish security.
Application Deployment Guide	Explains how to assemble and deploy applications to the Eclipse GlassFish and provides information about deployment descriptors.
Application Development Guide	Explains how to create and implement Java Platform, Enterprise Edition (Jakarta EE platform) applications that are intended to run on the Eclipse GlassFish. These applications follow the open Java standards model for Jakarta EE components and application programmer interfaces (APIs). This guide provides information about developer tools, security, and debugging.
Add-On Component Development Guide	Explains how to use published interfaces of Eclipse GlassFish to develop add-on components for Eclipse GlassFish. This document explains how to perform only those tasks that ensure that the add-on component is suitable for Eclipse GlassFish.
Embedded Server Guide	Explains how to run applications in embedded Eclipse GlassFish and to develop applications in which Eclipse GlassFish is embedded.
High Availability Administration Guide	Explains how to configure Eclipse GlassFish to provide higher availability and scalability through failover and load balancing.
Performance Tuning Guide	Explains how to optimize the performance of Eclipse GlassFish.
Troubleshooting Guide	Describes common problems that you might encounter when using Eclipse GlassFish and explains how to solve them.
Error Message Reference	Describes error messages that you might encounter when using Eclipse GlassFish.
Reference Manual	Provides reference information in man page format for Eclipse GlassFish administration commands, utility commands, and related concepts.
Message Queue Release Notes	Describes new features, compatibility issues, and existing bugs for Open Message Queue.

Book Title	Description
Message Queue Technical Overview	Provides an introduction to the technology, concepts, architecture, capabilities, and features of the Message Queue messaging service.
Message Queue Administration Guide	Explains how to set up and manage a Message Queue messaging system.
Message Queue Developer's Guide for JMX Clients	Describes the application programming interface in Message Queue for programmatically configuring and monitoring Message Queue resources in conformance with the Java Management Extensions (JMX).
Message Queue Developer's Guide for Java Clients	Provides information about concepts and procedures for developing Java messaging applications (Java clients) that work with Eclipse GlassFish.
Message Queue Developer's Guide for C Clients	Provides programming and reference information for developers working with Message Queue who want to use the C language binding to the Message Queue messaging service to send, receive, and process Message Queue messages.

Related Documentation

The following tutorials explain how to develop Jakarta EE applications:

- Your First Cup: An Introduction to the Jakarta EE Platform. For beginning Jakarta EE programmers, this short tutorial 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.
- The Jakarta EE Tutorial. This comprehensive tutorial explains how to use Jakarta EE platform technologies and APIs to develop Jakarta EE applications.

Javadoc tool reference documentation for packages that are provided with Eclipse GlassFish is available as follows.

- The Jakarta EE specifications and API specification is located at https://jakarta.ee/specifications/.
- The API specification for Eclipse GlassFish 7, including Jakarta EE platform packages and nonplatform packages that are specific to the Eclipse GlassFish product, is located at https://glassfish.org/docs/.

For information about creating enterprise applications in the NetBeans Integrated Development Environment (IDE), see the NetBeans Documentation, Training & Support page.

For information about the Derby database for use with the Eclipse GlassFish, see the Derby page.

The Jakarta EE Samples project is a collection of sample applications that demonstrate a broad range of Jakarta EE technologies. The Jakarta EE Samples are bundled with the Jakarta EE Software Development Kit (SDK) and are also available from the repository (`https://github.com/eclipse-ee4j/glassfish-samples`).

Typographic Conventions

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

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name% su`
		
		`Password:`

AaBbCc123	A placeholder to be replaced with a real name or value	The command to remove a file is `rm` filename.
AaBbCc123	Book titles, new terms, and terms to be emphasized (note that some emphasized items appear bold online)	Read Chapter 6 in the User's Guide. A cache is a copy that is stored locally. Do not save the file.

Symbol Conventions

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

Symbol	Description	Example	Meaning
`[ ]`	Contains optional arguments and command options.	`ls [-l]`	The `-l` option is not required.
`{ | }`	Contains a set of choices for a required command option.	`-d {y|n}`	The `-d` option requires that you use either the `y` argument or the `n` argument.
`${ }`	Indicates a variable reference.	`${com.sun.javaRoot}`	References the value of the `com.sun.javaRoot` variable.
`-`	Joins simultaneous multiple keystrokes.	Control-A	Press the Control key while you press the A key.
`+`	Joins consecutive multiple keystrokes.	Ctrl+A+N	Press the Control key, release it, and then press the subsequent keys.
`>`	Indicates menu item selection in a graphical user interface.	File > New > Templates	From the File menu, choose New. From the New submenu, choose Templates.

Default Paths and File Names

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

Placeholder	Description	Default Value
as-install	Represents the base installation directory for Eclipse GlassFish. In configuration files, as-install is represented as follows: `${com.sun.aas.installRoot}`	• Installations on the Oracle Solaris operating system, Linux operating system, and Mac OS operating system: user's-home-directory`/glassfish7/glassfish` • Installations on the Windows operating system: SystemDrive`:\glassfish7\glassfish`
as-install-parent	Represents the parent of the base installation directory for Eclipse GlassFish.	• Installations on the Oracle Solaris operating system, Linux operating system, and Mac operating system: user's-home-directory`/glassfish7` • Installations on the Windows operating system: SystemDrive`:\glassfish7`
domain-root-dir	Represents the directory in which a domain is created by default.	as-install`/domains/`
domain-dir	Represents the directory in which a domain's configuration is stored. In configuration files, domain-dir is represented as follows: `${com.sun.aas.instanceRoot}`	domain-root-dir/domain-name
instance-dir	Represents the directory for a server instance.	domain-dir/instance-name

Introduction to the Development Environment for Eclipse GlassFish Add-On Components

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

The following topics are addressed here:

Eclipse GlassFish Modular Architecture and Add-On Components

Eclipse GlassFish has a modular architecture in which the features of Eclipse GlassFish 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 Eclipse GlassFish 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 Eclipse GlassFish enables you to extend the basic functionality of Eclipse GlassFish 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 Eclipse GlassFish in the same way as components that are supplied in Eclipse GlassFish distributions.

You can create and offer new or updated add-on components at any time. Eclipse GlassFish administrators can install add-on components and update or remove installed components after Eclipse GlassFish is installed. For more information, see "Extending and Updating Eclipse GlassFish"

in Eclipse GlassFish Administration Guide.

OSGi Alliance Module Management Subsystem

To enable components to be added when required, Eclipse GlassFish provides a lightweight and extensible kernel that uses the module management subsystem from the OSGi Alliance (`http://www.osgi.org/`). Any Eclipse GlassFish component that plugs in to this kernel must be implemented as an OSGi bundle. To enable an add-on component to plug in to the Eclipse GlassFish kernel in the same way as other components, package the component as an OSGi bundle. For more information, see Packaging an Add-On Component.

The default OSGi module management subsystem in Eclipse GlassFish is the Apache Felix OSGi framework (`http://felix.apache.org`). However, the Eclipse GlassFish kernel uses only the OSGi Service Platform Release 4 (`http://www.osgi.org/Release4/HomePage`) API. Therefore, Eclipse GlassFish 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) (`https://github.com/eclipse-ee4j/glassfish-hk2`) is the lightweight and extensible kernel of Eclipse GlassFish. HK2 consists of the following technologies:

- Module subsystem. The HK2 module subsystem provides isolation between components of the Eclipse GlassFish. 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. Eclipse GlassFish discovers these components automatically and dynamically. HK2 components use injection of dependencies to express dependencies on other components. Eclipse GlassFish provides two-way mappings between the services of an HK2 component and OSGi services.

For more information, see 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 Eclipse GlassFish distributions, the component must meet the following requirements:

- If the component generates management data, configuration data, or monitoring data, it must provide that data to other Eclipse GlassFish components in the same way as other Eclipse GlassFish components.
- If the component generates management data, configuration data, or monitoring data, it must provide that data to users through Eclipse GlassFish 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
- Extending the Administration Console
- Extending the `asadmin` Utility
- Adding Monitoring Capabilities
- Adding Configuration Data for a Component
- Adding Container Capabilities
- Creating a Session Persistence Module
- Packaging and Delivering an Add-On Component

Writing HK2 Components

The Hundred-Kilobyte Kernel (HK2) is the lightweight and extensible kernel of Eclipse GlassFish. To interact with Eclipse GlassFish, 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.

For more information, see Writing HK2 Components.

Extending the Administration Console

The Administration Console is a browser-based tool for administering Eclipse GlassFish. 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.

For more information, see Extending the Administration Console.

Extending the `asadmin` Utility

The `asadmin` utility is a command-line tool for configuring and administering Eclipse GlassFish. Extending the `asadmin` utility enables you to provide administrative interfaces for an add-on component that are consistent with the interfaces of other Eclipse GlassFish 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.

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

For more information, see Adding Monitoring Capabilities.

Adding Configuration Data for a Component

The configuration data of a component determines the characteristics and runtime behavior of a component. Eclipse GlassFish provides interfaces to enable an add-on component to store its configuration data in the same way as other Eclipse GlassFish components. These interfaces are similar to interfaces that are defined in Jakarta XML Binding 3.0 (`https://jakarta.ee/specifications/xml-binding/3.0/`). By using these interfaces to store configuration data, you ensure that the add-on component is fully integrated with Eclipse GlassFish. As a result, administrators can configure an add-on component in the same way as they can configure other Eclipse GlassFish components.

For more information, see Adding Configuration Data for a Component.

Adding Container Capabilities

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

For more information, see Adding Container Capabilities.

Creating a Session Persistence Module

Eclipse GlassFish enables you to create a session persistence module in the web container for high availability-related functionality by implementing the `PersistenceStrategyBuilder` interface . Using the `PersistenceStrategyBuilder` interface in an HK2 service makes the session manager extensible because you can implement a new persistence type without having to modify the web container code.

For information about other high-availability, session persistence solutions, see "Configuring High Availability Session Persistence and Failover" in Eclipse GlassFish High Availability Administration Guide.

For more information, see Creating a Session Persistence Module.

Packaging and Delivering an Add-On Component

Packaging an add-on component enables the component to interact with the Eclipse GlassFish kernel in the same way as other components. Integrating a component with Eclipse GlassFish enables Eclipse GlassFish to discover the component at runtime.

For more information, see Packaging, Integrating, and Delivering an Add-On Component.

Writing HK2 Components

The Hundred-Kilobyte Kernel (HK2) is the lightweight and extensible kernel of Eclipse GlassFish. To interact with Eclipse GlassFish, 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

The Hundred-Kilobyte Kernel (HK2) provides a module system and component model for building complex software systems. HK2 forms the core of Eclipse GlassFish'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.annotations.Contract` annotation.

```
@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.

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

For more information, see `Service` (http://hk2.java.net/auto-depends/apidocs/org/jvnet/hk2/annotations/Service.html).

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).

```
@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:

```
@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 `Habitat` instance. The simplest way to use the `Habitat` instance is through a `getComponent(Class<T> contract)` call:

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

More APIs are available at `Habitat` (http://hk2.java.net/auto-depends/apidocs/org/jvnet/hk2/component/Habitat.html).

HK2 Lifecycle Interfaces

Components can attach behaviors to their construction and destruction events by implementing the `org.jvnet.hk2.component.PostConstruct` (http://hk2.java.net/auto-depends/apidocs/org/jvnet/hk2/component/PostConstruct.html) interface, the `org.jvnet.hk2.component.PreDestroy` (http://hk2.java.net/auto-depends/apidocs/org/jvnet/hk2/component/PreDestroy.html) 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 `PostContruct` and `PreDestroy`

```
@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.

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` (http://hk2.java.net/auto-depends/apidocs/org/jvnet/hk2/annotations/Inject.html) 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:

```
@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;
```

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`.

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

Iterable<Startup> startups;
startups = habitat.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

Eclipse GlassFish 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 Eclipse GlassFish service, add the `org.jvnet.hk2.annotations.Service` annotation at the class-definition level of your Java programming language class.

```
@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.glassfish.hk2.scopes.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>org.glassfish.hk2</groupId>
    <artifactId>hk2-maven-plugin</artifactId>
    <configuration>
        <includes>
            <include>com/example/**</include>
        </includes>
    </configuration>
</plugin>
```

Example 2-5 Example of `META-INF/services` File Generation

This example shows how to use `@Contract` (http://hk2.java.net/auto-depends/apidocs/org/jvnet/hk2/annotations/Contract.html) and `@Service` (http://hk2.java.net/auto-depends/apidocs/org/jvnet/hk2/annotations/Service.html) and the resulting `META-INF/services` files.

The interfaces and classes in this example are as follows:

```
package com.example.wallaby.annotations;
@Contract
public interface Startup {...}
```

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

package com.example.wallaby;
@Service
public class MyService implements Startup, RandomContract, PropertyChangeListener {
    ...
}
```

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

```
com.example.wallaby.annotations.Startup
com.example.wombat.RandomContract
```

Extending the Administration Console

The Administration Console is a browser-based tool for administering Eclipse GlassFish. 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 Jakarta Server 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.

The following topics are addressed here:

- Administration Console Architecture
- About Administration Console Templates
- About Integration Points
- Specifying the ID of an Add-On Component
- Adding Functionality to the Administration Console
- Adding Internationalization Support
- Changing the Theme or Brand of the Administration Console
- Creating an Integration Point Type

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 Eclipse GlassFish 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 Eclipse GlassFish 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 theAdministration 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 and HK2 Component Model.

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.

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.

About Administration Console Templates

Eclipse GlassFish includes a set of templates that make it easier to create Jakarta Server Faces pages for your add-on component. These templates use Templating for Jakarta Server Faces Technology, which is also known as JSFTemplating.

Examples of JSFTemplating technology can be found in the following sections of this chapter:

- Creating a Jakarta Server Faces Page for Your Node
- Creating Jakarta Server Faces Pages for Your Tabs
- Creating a Jakarta Server Faces Page for Your Task
- Creating a Jakarta Server Faces Page for Your Task Group
- Creating a Jakarta Server Faces Page for Your Page Content
- Adding a Page to the Administration Console

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
- Jakarta Server 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 Jakarta Server 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 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:

```
<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 for an example.

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

```
<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/`.

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

- Adding Tabs to a Page
- Adding a Task to the Common Tasks Page
- Adding a Task Group to the Common Tasks Page
- Adding Content to a Page
- Adding a Page to the Administration Console

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 Eclipse GlassFish 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

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.

```
        <integration-point
```

```
                id="samplenode"
                parentid="tree"
                type="org.glassfish.admingui:treeNode"
                priority="200"
                content="sampleNode.jsf"
        />
```

This example specifies the following values in addition to the `parentId`:

- 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 Jakarta Server Faces Page for Your Node

A Jakarta Server Faces page for a tree node uses the tag `sun:treeNode`. This tag provides all the capabilities of the Project Woodstock tag `webuijsf:treeNode`.

Example 3-3 Example Jakarta Server Faces Page for a Tree Node

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

```
<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 example, most of the integration points use a very simple Jakarta Server 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). 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 Jakarta Server 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).

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 Eclipse GlassFish 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 Eclipse GlassFish page of the Administration Console:

```
        <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 Eclipse GlassFish 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 Jakarta Server 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 Eclipse GlassFish page of the Administration Console:

```
        <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 Jakarta Server Faces pages that display the tabs.

Creating Jakarta Server Faces Pages for Your Tabs

A Jakarta Server Faces page for a tab uses the tag `sun:tab`. This tag provides all the capabilities of the Project Woodstock tag `webuijsf:tab`.

Example 3-6 Example Jakarta Server Faces Page for a Tab

In the example, the `sampleTab.jsf` file has 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>
```

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

In your own Jakarta Server 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.

`text`

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 Jakarta Server 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 Jakarta Server Faces file that corresponds to that tab:

```
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 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:

```
        <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 Jakarta Server Faces Page for Your Task

A Jakarta Server Faces page for a task uses the tag `sun:commonTask`. This tag provides all the capabilities of the Project Woodstock tag `webuijsf:commonTask`.

Example 3-8 Example Jakarta Server Faces Page for a Task

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

```
<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>
```

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 Jakarta Server 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:

```
        <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 Jakarta Server Faces Page for Your Task Group

A Jakarta Server Faces page for a task group uses the tag `sun:commonTasksGroup`. This tag provides all the capabilities of the Project Woodstock tag `webuijsf:commonTasksGroup`.

Example 3-10 Example Jakarta Server Faces Page for a Task Group

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

```
<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:tree:configuration:sampleConfigNode',

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

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 Jakarta Server Faces Page for Your Page Content

A Jakarta Server 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 `webuijsf:property`.

Example 3-12 Example Jakarta Server Faces Page for a Resource Page Item

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

```
<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 Jakarta Server 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:

```
<!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:hidden id="helpkey" value="" />

</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:

```
<!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:

```
        <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.

`org.glassfish.admingui:loginimage`

Specifies 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.

`org.glassfish.admingui:loginform`

Specifies the name and location of the include file containing the customized login form code. This code also contains the login background image used for the login page for the Administration Console.

`org.glassfish.admingui:versioninfo`

Specifies the name and location of the include file containing the branding image that will be integrated with the content of the version popup window.

Example 3-15 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.

```
        <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://github.com/eclipse-ee4j/glassfish-woodstock`). 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 Jakarta Server 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.

Extending the asadmin Utility

The asadmin utility is a command-line tool for configuring and administering Eclipse GlassFish. Extending the asadmin utility enables you to provide administrative interfaces for an add-on component that are consistent with the interfaces of other Eclipse GlassFish 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 Eclipse GlassFish

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

- Location independence. Administration subcommands can be loaded from any add-on component that is known to Eclipse GlassFish.
- Extensibility. Administrative subcommands that are available to Eclipse GlassFish 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 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
- Specifying the Name of an `asadmin` Subcommand
- Ensuring That an `AdminCommand` Implementation Is Stateless
- Example of Adding an `asadmin` Subcommand

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 Eclipse GlassFish runtime.

  For information about how to make a component known to the Eclipse GlassFish runtime, see Integrating an Add-On Component With Eclipse GlassFish.

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.

Subcommand names are case-sensitive.

Subcommands that are supplied in Eclipse GlassFish distributions typically create, delete, and list objects of a particular type. For consistency with the names of subcommands that are supplied in Eclipse GlassFish 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, Eclipse GlassFish 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 Eclipse GlassFish distributions, see the Eclipse GlassFish 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.

```
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;

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

Adding Parameters to an `asadmin` Subcommand

The parameters of an `asadmin` subcommand are the options and operands of the subcommand.

- Options control how the `asadmin` utility performs a subcommand.
- Operands are the objects on which a subcommand acts. For example, the operand of the `start-domain` subcommand is the domain that is to be started.

The following topics are addressed here:

Representing a Parameter of an `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 Eclipse GlassFish to perform the following operations at runtime on the parameter:

- Validation. The Eclipse GlassFish determines whether all required parameters are specified and returns an error if any required parameter is omitted.
- Injection. Before the subcommand runs, the Eclipse GlassFish injects each parameter into the required field or method before the subcommand is run.
- Usage message generation. The Eclipse GlassFish 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 Eclipse GlassFish 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`

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 Eclipse GlassFish 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. You can also compute the default value dynamically by extending the `ParamDefaultCalculator` class and setting the `defaultCalculator` element of the `@Param` annotation to this class. If these elements are not set, 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`.

Specifying Whether a Parameter Can Be Used Multiple Times on the Command ==== Line

By default, each parameter can be used once on the command line. To use the parameter multiple times, set the `multiple` element of the `@Param` annotation to `true`. The type of the annotated parameter must be an array.

Example of Adding Parameters to an `asadmin` Subcommand

Example 4-2 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.

Name	Represented As	Acceptable Values	Default Value	Optional or Required	Short Name	Option or Operand
`--originator`	A field that is named `originator`	Any character string	None defined	Required	None	Option
`--description`	A field that is named `mycontainerDescription`	Any character string	None defined	Optional	None	Option
`--enabled`	A field that is named `enabled`	`true` or `false`	`false`	Optional	None	Option
`--containername`	A field that is named `containername`	Any character string	None defined	Required	None	Operand

```
...
import org.glassfish.api.Param;
...
{
//…
    @Param
    String originator;

    @Param(name="description", optional=true)
    //…
    String mycontainerDescription

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

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

Making `asadmin` Subcommands Cluster-Aware

The Eclipse GlassFish `asadmin` command framework provides support for making `asadmin` subcommands work properly in a clustered environment or with standalone server instances. A command that changes a configuration is first executed on the domain administration server (DAS) and then executed on each of the server instances affected by the change. Annotations provided by the framework determine the instances on which the command should be replicated and executed.

Commands that do not change a configuration need not be executed on the DAS at all, but only on the necessary instances. The framework provides support for collecting the output from the instances and sending a report back to the user.

Subcommands in a multi-instance environment can accept a `--target` option to specify the cluster or instance on which the command acts. From within the command, the `Target` utility allows the command to determine information about where it is running. For some commands, it may be desirable to have a main command that runs on the DAS and supplemental preprocessing or postprocessing commands that run on the instances.

The following topics are addressed here:

- Specifying Allowed Targets
- The `Target` Utility
- Specifying `asadmin` Subcommand Execution
- Subcommand Preprocessing and Postprocessing
- Running a Command from Another Command

Specifying Allowed Targets

When you define a `--target` option by using the `@Param` annotation in the `org.glassfish.api` package, possible targets are as follows:

- `domain` — The entire domain
- `server` — The domain administration server, or DAS
- cluster — A homogeneous set of server instances that function as a unit
- standalone instance — A server instance that isn't part of a cluster
- clustered instance — A server instance that is part of a cluster
- config — A configuration for a cluster or standalone server instance

These possible targets are represented by the following `CommandTarget` elements of the `@TargetType` annotation in the `org.glassfish.config.support` package:

- `CommandTarget.DOMAIN`
- `CommandTarget.DAS`
- `CommandTarget.CLUSTER`
- `CommandTarget.STANDALONE_SERVER`
- `CommandTarget.CLUSTERED_INSTANCE`
- `CommandTarget.CONFIG`

By default, the allowed targets are `server` (the DAS), standalone server instances, clusters, and configurations. Not specifying a `@TargetType` annotation is equivalent to specifying the following `@TargetType` annotation:

```
@TargetType(CommandTarget.DAS,CommandTarget.STANDALONE_SERVER,CommandTarget.CLUSTER,Co
mmandTarget.CONFIG)
```

Subcommands that support other combinations of targets must specify `@TargetType` annotations. For example, the `create-http-lb` subcommand supports only standalone server instance and cluster targets. Its `@TargetType` annotation is as follows:

```
@TargetType(CommandTarget.STANDALONE_SERVER,CommandTarget.CLUSTER)
```

Most subcommands do not act on server instances that are part of a cluster. This ensures that all server instances in a cluster remain synchronized. Thus, the `CommandTarget.CLUSTERED_INSTANCE` element of the `@TargetType` annotation is rarely used.

An example exception is the `enable` subcommand. To perform a rolling upgrade of an application deployed to a cluster, you must be able to enable the new application (which automatically disables the old) on one clustered instance at a time. The `@TargetType` annotation for the `enable` subcommand is as follows, all on one line:

```
@TargetType(CommandTarget.DAS,CommandTarget.STANDALONE_INSTANCE,CommandTarget.CLUSTER,
CommandTarget.CLUSTERED_INSTANCE)
```

Note that the `CommandTarget.CLUSTERED_INSTANCE` element is specified.

The target name specified in the command line is injected into the subcommand implementation if the following annotation is present:

```
@Param(optional=true,defaultValue=SystemPropertyConstants.DEFAULT_SERVER_INSTANCE_NAME
)
String target;
```

The `Target` Utility

The `Target` utility is a service, present in the `internal-api` module, `org.glassfish.internal.api` package, which a command implementation can obtain by using the following annotation:

```
@Inject Target targetUtil;
```

You can use this utility to avoid writing boiler plate code for actions such as getting the list of server instances for a cluster or checking if a server instance is part of a cluster. For example, here is an example of using the utility to obtain the configuration for a target cluster or server instance:

```
Config c = targetUtil.getConfig(target);
```

The `Target` utility is packaged in the as-install`/modules/internal-api.jar` file. Its methods are documented with comments.

Specifying `asadmin` Subcommand Execution

By default, all `asadmin` subcommands are automatically replicated and run on the DAS and all Eclipse GlassFish instances specified in the `--target` option. To run a subcommand only on the DAS, use the following `@ExecuteOn` annotation in the `org.glassfish.api.admin` package:

```
@ExecuteOn(RuntimeType.DAS)
```

The `stop-domain` subcommand and subcommands that list information are examples of subcommands that execute only on the DAS.

To run a subcommand only on applicable server instances, use the following `@ExecuteOn` annotation:

```
@ExecuteOn(RuntimeType.INSTANCE)
```

Not specifying an `@ExecuteOn` annotation is equivalent to specifying the following `@ExecuteOn` annotation:

```
@ExecuteOn(RuntimeType.DAS,RuntimeType.INSTANCE)
```

In addition to `RuntimeType`, you can specify the following additional elements with the `@ExecuteOn` annotation:

- `ifFailure` — By default, if errors occur during execution of a subcommand on a server instance, command execution is considered to have failed and further execution is stopped. However, you can choose to ignore the failure or warn the user rather than stopping further command execution. Specify the `ifFailure` element and set it to `FailurePolicy.Ignore` or `FailurePolicy.Warn`. For example:

  ```
  @ExecuteOn(value={RuntimeType.DAS}, ifFailure=FailurePolicy.Warn)
  ```

- `ifOffline` — By default, if a server instance is found to be offline during the command replication process, command execution is considered to have failed and further execution is stopped. However, you can choose to ignore the failure or warn the user rather than stopping further command execution. Specify the `ifOffline` element and set it to `FailurePolicy.Ignore` or `FailurePolicy.Warn`. For example:

  ```
  @ExecuteOn(value={RuntimeType.DAS}, ifOffline=FailurePolicy.Ignore)
  ```

Subcommand Preprocessing and Postprocessing

Some `asadmin` subcommands may require preprocessing or postprocessing. For example, after an application is deployed to the DAS, references are created in all applicable server instances, which synchronize with the DAS. As another example, Message Queue or load balancer settings may have to be reconfigured whenever a server instance is added to a cluster.

For such cases, the command replication framework provides the `@Supplemental` annotation (in the `org.glassfish.api.admin` package). An implementation must use the `value` element of the `@Supplemental` annotation to express the supplemented command. This value is the name of the command as defined by the supplemented command's `@Service` annotation (in the `org.jvnet.hk2.annotations` package).

For example, the `deploy` subcommand requires postprocessing. The deployment command implementation looks like this:

```
@Service(name="deploy")
@ExecuteOn(RuntimeType.DAS)
public DeployCommand implements AdminCommand {
// Do Actual Deployment
}
```

A supplemental command that is run after every successful deployment looks like this:

```
@Service(name="DeploymentSupplementalCommand")
@Supplemental("deploy")
@ExecuteOn(RuntimeType.INSTANCE)
public DeploymentSupplementalCommand implements AdminCommand {
// Do logic that happens after deployment has been done
}
```

As another example, a subcommand to create a local server instance might look like this:

```
@Service(name = "create-local-instance")
@Scoped(PerLookup.class)
public final class CreateLocalInstanceCommand implements AdminCommand {
// Do local instance creation
}
```

A supplemental command to change Message Queue or load balancer settings after local instance creation might look like this:

```
@Service(name="CreateLocalInstanceSupplementalCommand")
@Supplemental("create-local-instance")
public CreateLocalInstanceSupplementalCommand implements AdminCommand {
// Change MQ/LB properties here
```

```
}
```

A supplemental command implements AdminCommand, thus it can use the `@Param` annotation and expect the corresponding `asadmin` command parameters to be injected at runtime. The parameter values available for injection are the same ones provided for the original command with which the supplemental command is associated. For example, the `DeploymentSupplementalCommand` has access to the parameter values available to the `DeployCommand` invocation.

An `asadmin` subcommand can be supplemented with multiple supplemental commands. In this case, all supplemental commands are run after completion of the main command but without any guarantee of the order in which they run.

To specify that a supplemental command is run before the main command, set the `on` element of the `@Supplemental` annotation to `Supplemental.Timing.Before`. For example:

```
@Supplemental(value="mycommand", on=Supplemental.Timing.Before)
```

Supplemental commands can use the `@ExecuteOn` annotation as described in Specifying `asadmin` Subcommand Execution.

Running a Command from Another Command

An `asadmin` subcommand or supplemental command might need to run another subcommand. For example, a subcommand running on the DAS might need to run a different subcommand on one or more server instances. Such invocations might use the `ClusterExecutor` class (in the `org.glassfish.api.admin` package), which accepts a `ParameterMap`, to pass parameters and their values to the invoked command.

The `ParameterMapExtractor` utility is a service, present in the `common-util` module, `org.glassfish.common.util.admin` package, which creates a new `ParameterMap` populated using the parameters and values of another `AdminCommand` that has already been injected.

To list parameter names you want excluded from the `ParameterMap`, pass the following:

```
Set<String>
```

This is optional.

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.

Target	Format
Subcommand or parameter with the `@I18n` annotation	`subcommand-name.command.resource-name`
Subcommand without the `@I18n` annotation	`subcommand-name.command`
Parameter without the `@I18n` annotation	`subcommand-name.command.param-name`

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.

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.

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 `--containername` parameter is associated with the message `The container name`. No internationalization support is provided for this message.

The addition of the parameters `originator`, `description`, `enabled` and `containername` to the subcommand is shown in Example 4-2.

```
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 `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
```

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.

```
    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 Eclipse GlassFish CLI

The brand in the Eclipse GlassFish 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` subcommand displays
- In each entry in the `server.log` file

If you are incorporating Eclipse GlassFish into a new product with an external vendor's own brand

name, change the brand in the Eclipse GlassFish CLI.

To change the brand in the Eclipse GlassFish 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.abbreviation=...
product.version=...
...
```

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=Eclipse GlassFish`

Specifies the full product name without any release information.

`product.name.abbreviation=GF`

Specifies an abbreviated form of the product name without any release information.

`product.version=2023.02.24`

Returns the product version. It should follow standard form `major.minor.patch[-suffix]`

`product.build.git.commit=ec1ce24934f0bb174333a9886b58a0d2f9e52b44`

Specifies the git commit, used for the build.

Example 4-4 `glassfish-version.properties` File for Changing the Brand in the Eclipse GlassFish CLI

This example shows the content of the `glassfish-version.properties` for defining the product name and release information of Eclipse GlassFish 7.0.3-SNAPSHOT. The abbreviated product name is `GF`.

```
product.name=Eclipse GlassFish
product.name.short=GF
product.version=7.0.3-SNAPSHOT
product.build.git.commit=93176e2555176091c8522e43d1d32a0a30652d4a

domain.template.defaultJarFileName=appserver-domain.jar
admin.command.name=asadmin
```

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

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

    @Param(primary=true)
    String containername;

    /**
     * Executes the subcommand with the subcommand parameters passed as Properties
     * where the keys are the paramter names and the values the parameter values
     * @param context information
     */
    public void execute(AdminCommandContext context) {
        // domain and originator are not null
```

```
        // mycontainerDescription can be null.
    }
}
```

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 Eclipse GlassFish.

The usage statement for this subcommand is as follows:

```
asadmin list-runtime-environment{runtime|os}
```

```
package com.example.env.cli;

import org.glassfish.api.admin.AdminCommand;
import org.glassfish.api.admin.AdminCommandContext;
import org.glassfish.api.ActionReport;
import org.glassfish.api.I18n;
import org.glassfish.api.ActionReport.ExitCode;
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;

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.

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.

Option	Acceptable Values	Default value	Optional or Required
`--instances`	An integer in the range 1-10	5	Required
`--language`	`english`, `norsk`, or `francais`	`norsk`	Optional
`--style`	`formal`, `casual`, or `expansive`	`formal`	Optional

Code for the container component is shown in Example of Adding Container Capabilities.

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

```
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) {
        }

    }
}
```

Implementing Create, Delete, and List Commands Using Annotations

Many `asadmin` subcommands simply create, delete, or list objects in the configuration. Such code is repetitive to write and error prone. To simplify the writing of these `asadmin` commands, Eclipse GlassFish supports annotations that can create, delete, and list configuration objects from a command invocation. Unless attributes or properties are set to non-default values or extra actions are required, no writing of code is needed.

The following topics are addressed here:

Command Patterns

Create command pattern. The most basic create commands are implemented in the following pattern:

1. Retrieve the parent configuration object instance to which the child will be added. For example, the parent could be a `Clusters` object and the child a `Cluster` object.
2. Start a transaction on the parent instance.
3. Create the child configuration object instance.
4. Set the attributes and properties of the newly created child instance.
5. Add the child to the parent using one of the following accessor methods:

   ```
   void setChild(ChildType child)
   ```

   Used when there can be zero or one children of a single type associated with one parent instance.

   ```
   List<ChildType> getChildren()
   ```

   Used when there can be zero or more children of a single type associated with one parent

instance. You cannot retrieve a set of children of the same type from the same parent using two different accessor methods.

6. Commit the transaction.

A generic create command implementation can do most of these tasks if the following information is provided:

- A way to resolve the identity of the parent instance.
- The type of the child instance.
- A mapping between command options and child attributes.
- The accessor method for adding the child to the parent.

Delete command pattern. The most basic delete commands are implemented in the following pattern:

1. Retrieve the configuration object instance to be deleted.
2. Start a transaction on the parent instance.
3. Delete the child by removing it from the list or calling `setXXX(null)`.
4. Commit the transaction.

A generic delete command implementation can do most of these tasks if the following information is provided:

- A way to resolve the identity of the child instance.
- The accessor method for deleting the child.

List command pattern. The most basic list commands simply retrieve all configuration object instances of a given type.

Resolvers

A resolver retrieves a configuration object instance of a particular type. For a create command, it retrieves the parent of the object to be created. For a delete command, it retrieves the object to be deleted. A resolver implements the CrudResolver interface:

```
package org.glassfish.config.support;

/**
 * A config resolver is responsible for finding the target object of a specified
 * type on which a creation command invocation will be processed.
 *
 * Implementation of these interfaces can be injected with the command invocation
 * parameters in order to determine which object should be returned
 */
@Contract
public interface CrudResolver {
```

```
    /**
     * Retrieves the existing configuration object a command invocation is
     * intented to mutate.
     * @param context the command invocation context
     * @param type the type of the expected instance
     * @return the instance or null if not found
     */
    <T extends ConfigBeanProxy> T resolve(AdminCommandContext context, Class<T> type);
}
```

Given an `AdminCommandContext`, plus injection with the `asadmin` command line parameters (or any other HK2 services if necessary), the resolver should be able to determine the particular configuration object on which to act.

The following resolvers are provided in the `org.glassfish.config.support` package:

- `TargetBasedResolver` — Uses the `--target` option and the expected return type to retrieve the configuration object instance.
- `TargetAndNameBasedResolver` — Uses the `--target` option to look up a `Config` object and a name to retrieve one of the `Config` object's children.
- `TypeAndNameResolver` — Uses the requested type and `asadmin` command name operand to find the configuration object instance. This is useful for a configuration that uses the `@Index` annotation, which registers instances under names.
- `TypeResolver` — Uses the requested type to find the configuration object instance. This is the default resolver.

The `@Create` Annotation

By placing the `org.glassfish.config.support.Create` annotation on a method, you provide the following information:

- The `value` element of the `@Create` annotation is the name of the `asadmin` subcommand that creates the configuration object.
- The method's class is the type of the parent.
- The method's return type or parameter type is the type of the child.
- The method is the accessor method that adds a child of the specified type to the parent.

The only additional information needed is the resolver to use.

The following example specifies a `create-cluster` subcommand:

```
@Configured
public interface Clusters extends ConfigBeanProxy, Injectable {

    /**
     * Return the list of clusters currently configured
```

```
     *
     * @return list of {@link Cluster }
     */
    @Element
    @Create(value="create-cluster")
    public List<Cluster> getCluster();
}
```

Because there is only one instance of the parent type, `Clusters`, in the configuration, this example uses the default resolver to retrieve it. Therefore, no resolver needs to be specified.

The `@Delete` Annotation

By placing the `org.glassfish.config.support.Delete` annotation on a method, you provide the following information:

- The `value` element of the `@Delete` annotation is the name of the `asadmin` subcommand that deletes the configuration object.
- The method's class is the type of the parent.
- The method's return type or parameter type is the type of the child.
- The method is the accessor method that deletes a child of the specified type from the parent.

The only additional information needed is the resolver to use.

The following example specifies a `delete-cluster` subcommand:

```
@Configured
public interface Clusters extends ConfigBeanProxy, Injectable {

     /**
     * Return the list of clusters currently configured
     *
     * @return list of {@link Cluster }
     */
    @Element
    @Delete(value="delete-cluster", resolver=TypeAndNameResolver.class)
    public List<Cluster> getCluster();
}
```

The `TypeAndNameResolver` uses the child type and the name operand passed through the command line to retrieve the specific cluster instance to be deleted.

The `@Listing` Annotation

By placing the `org.glassfish.config.support.Listing` annotation on a method, you provide the following information:

- The `value` element of the `@Listing` annotation is the name of the `asadmin` subcommand that lists

the configuration objects.

- The method's class is the type of the parent.
- The method's return type is the type of the children to be listed.
- The method is always the following accessor method:

```
List<ChildType> getChildren()
```

The default resolver retrieves all of the children of the specified type. Therefore, no resolver needs to be specified for a list command.

The following example specifies a `list-clusters` subcommand:

```
@Configured
public interface Clusters extends ConfigBeanProxy, Injectable {

     /**
      * Return the list of clusters currently configured
      *
      * @return list of {@link Cluster }
      */
    @Element
    @Listing(value="list-clusters")
    public List<Cluster> getCluster();
}
```

Create Command Decorators

Most create commands must do more than create a single configuration object instance with default attribute values. For example, most create commands allow the user to specify non-default attribute values through command options. For another example, the `create-cluster` subcommand creates children of the `Cluster` object and copies a referenced `Config` object. A creation decorator provides the code necessary to perform such additional operations.

The interface that a creation decorator must implement is as follows:

```
@Scoped(PerLookup.class)
public interface CreationDecorator<T extends ConfigBeanProxy> {

    /**
     * The element instance has been created and added to the parent, it can be
     * customized. This method is called within a
     * {@link org.jvnet.hk2.config.Transaction}
     * and instance is therefore a writeable view on the configuration component.
     *
     * @param context administration command context
     * @param instance newly created configuration element
```

```
     * @throws TransactionFailure if the transaction should be rollbacked
     * @throws PropertyVetoException if one of the listener of <T> is throwing
     * a veto exception
     */
    public void decorate(AdminCommandContext context, T instance)
        throws TransactionFailure, PropertyVetoException;

    /**
     * Default implementation of a decorator that does nothing.
     */
    @Service
    public class NoDecoration implements CreationDecorator<ConfigBeanProxy> {
        @Override
        public void decorate(AdminCommandContext context, ConfigBeanProxy instance)
            throws TransactionFailure, PropertyVetoException {
            // do nothing
        }
    }
}
```

The CreationDecorator interface is in the `org.glassfish.config.support` package.

A `@Create` annotation specifies a creation decorator using a `decorator` element. For example:

```
@Configured
public interface Clusters extends ConfigBeanProxy, Injectable {

     /**
      * Return the list of clusters currently configured
      *
      * @return list of {@link Cluster }
      */
    @Element
    @Create(value="create-cluster", decorator=Cluster.Decorator.class)
    public List<Cluster> getCluster();
}
```

The `@Create` annotation is on a method of the parent class. However, the referenced creation decorator class is associated with the child class. For example:

```
@Configured
public interface Cluster extends ConfigBeanProxy, ... {

    ...

    @Service
    @Scoped(PerLookup.class)
    class Decorator implements CreationDecorator<Cluster> {
```

```
        @Param(name="config", optional=true)
        String configRef=null;

        @Inject
        Domain domain;

        @Override
        public void decorate(AdminCommandContext context, final Cluster instance)
                throws TransactionFailure, PropertyVetoException {

        ...

        }
    }
}
```

The decorator class can optionally be an inner class of the child class. You can inject command options using the `@Param` annotation. You can also inject HK2 services or configuration instances.

Delete Command Decorators

Some delete commands must do more than delete a single configuration object instance. For example, the `delete-cluster` subcommand deletes the referenced `Config` object if no other `Cluster` or `Instance` objects reference it. A deletion decorator provides the code necessary to perform such additional operations.

The interface that a deletion decorator must implement is as follows:

```
/**
 * A decorator for acting upon a configuration element deletion.
 *
 * @param <T> the deleted element parent type
 * @param <U> the deleted element
 */
@Scoped(PerLookup.class)
public interface DeletionDecorator<T extends ConfigBeanProxy,
    U extends ConfigBeanProxy> {

    /**
     * notification of a configuration element of type U deletion.
     *
     * Note that this notification is called within the boundaries of the
     * configuration transaction, therefore the parent instance is a
     * writable copy and further changes to the parent can be made without
     * enrolling it inside a transaction.
     *
     * @param context the command context to lead to the element deletion
     * @param parent the parent instance the element was removed from
     * @param child the deleted instance
```

```
     */
    public void decorate(AdminCommandContext context, T parent, U child);
}
```

The DeletionDecorator interface is in the `org.glassfish.config.support` package.

A `@Delete` annotation specifies a deletion decorator using a `decorator` element. For example:

```
@Configured
public interface Clusters extends ConfigBeanProxy, Injectable {

     /**
      * Return the list of clusters currently configured
      *
      * @return list of {@link Cluster }
      */
    @Element
    @Delete(value="delete-cluster", resolver= TypeAndNameResolver.class,
        decorator=Cluster.DeleteDecorator.class)
    public List<Cluster> getCluster();
}
```

The `@Delete` annotation is on a method of the parent class. However, the referenced deletion decorator class is associated with the child class. For example:

```
@Configured
public interface Cluster extends ConfigBeanProxy, ... {

    ..
    @Service
    @Scoped(PerLookup.class)
    class DeleteDecorator implements DeletionDecorator<Clusters, Cluster> {
        ....
    }
}
```

The decorator class can optionally be an inner class of the child class. You can inject command options using the `@Param` annotation. You can also inject HK2 services or configuration instances.

Specifying Command Execution

Commands specified with the `@Create`, `@Delete`, and `@Listing` annotations can use the `@ExecuteOn` annotation. The `@ExecuteOn` annotation specifies whether the command runs on the DAS, on server instances, or both (the default). For more information, see Specifying `asadmin` Subcommand Execution.

To add an `@ExecuteOn` annotation to a `@Create` or `@Delete` annotation, use the `cluster` element. For example:

```
@Create(value="create-instance", resolver=TypeResolver.class,
        decorator=Server.CreateDecorator.class,
        cluster=@org.glassfish.api.admin.ExecuteOn(value=RuntimeType.DAS))
```

Using Multiple Command Annotations

You can specify multiple command annotations on the same method. The following example combines create, delete, and list commands for clusters:

```
@Configured
public interface Clusters extends ConfigBeanProxy, Injectable {

     /**
      * Return the list of clusters currently configured
      *
      * @return list of {@link Cluster }
      */
    @Element
    @Create(value="create-cluster", decorator=Cluster.Decorator.class)
    @Delete(value="delete-cluster", resolver= TypeAndNameResolver.class,
        decorator=Cluster.DeleteDecorator.class)
    @Listing(value="list-clusters")
    public List<Cluster> getCluster();
}
```

You can also specify multiple create or delete command annotations for the same configuration object type using the `@Creates` or `@Deletes` annotation (both in the `org.glassfish.config.support` package). For example:

```
@Element
@Creates(
   @Create(value="create-something", decorator=CreateSomething.Decorator)
   @Create(value="create-something-else", decorator=CreateSomethingElse.Decorator)
   List<Something> getSomethings();
)
```

These commands create configuration object instances of the same type. Differences in the decorators and resolvers can produce differences in the options each command takes. The `@Param` annotated attributes of the created type define a superset of options for both commands.

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

The following topics are addressed here:

- Defining Statistics That Are to Be Monitored
- Updating the Monitorable Object Tree
- Dotted Names and REST URLs for an Add-On Component's Statistics
- Example of Adding Monitoring Capabilities

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 Eclipse GlassFish, 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
- Sending an Event

Defining an Event Provider

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

Eclipse GlassFish 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 Eclipse GlassFish, 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 Eclipse GlassFish, `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 Eclipse GlassFish, 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 Eclipse GlassFish, 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.
- 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.

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.

```
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 Eclipse GlassFish, 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 Eclipse GlassFish, `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` attribute is not required to be unique. In Eclipse GlassFish, 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 Eclipse GlassFish, 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 Eclipse GlassFish 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
  ```

  The `class-list` is 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
  ```

  The `path-list` is 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.

```
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 Eclipse GlassFish, 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.

```
...
public class TransactionManagerImpl {
...
     public TransactionManagerImpl() {
         TxManager txProvider = new TxManager();
         ...
     }
    ...
    public void begin() {
        String txId = createTransactionId();
        ....
        txProvider.onTxBegin(txId); //emit
      }
...
}
```

Updating the Monitorable Object Tree

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

Because the tree is dynamic, the tree changes as components of the Eclipse GlassFish 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 Eclipse GlassFish administrative commands `get`, `list`, and `set`. 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 Eclipse GlassFish 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.

Updating the monitorable object tree involves the following tasks:

- Creating Event Listeners
- Representing a Component's Statistics in an Event Listener Class
- Subscribing to Events From Event Provider Classes
- Registering an Event Listener

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.gmbal.ManagedObject` 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. However,Eclipse GlassFish provides utility classes to perform some common operations such as `count`, `avg`, and `sum`.
- The listener must have the same restrictions as a Jakarta 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.

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.

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.

```
...
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");
```

```
...
    @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 and Defining an Event Provider by Writing an XML Fragment.

module-name

The module for which the event provider is defined. This parameter must match be 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 and Defining an Event Provider by Writing an XML Fragment.

probe-provider-name

The name of the event provider. This parameter must match be 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 and Defining an Event Provider by Writing an XML Fragment.

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.

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.

```
...
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 {
    ...;    @ProbeListner("examplecomponent:transaction:manager:begin")
    public void begin(
    @ProbeParam("{txId}")
    String txId) {
      txCount.increment();
    }
  }
```

Listening for Changes to Values That are Not Part of the Target Method Definition

Event listeners can express their interest in certain predefined values that are not part of the target method definition. For example, $\{gf.appname}, $\{gf.modulename} etc. are some of the computed params that are available to the clients, these values are computed/evaluated only on demand and provided by the event infrastructure.

Getting Information About a Event Provider

`ProbeProviderInfo` contains details about individual event types in an event provider class.

```
public interface ProbeProviderInfo {

      public String getModuleName();

      public String getProviderName();

      public String getApplicationName();

      public String getProbeName();

      public String[] getParamterNames();

      public Class getParamterTypes();
```

```
}
```

Listening for Events From Classes That Are Not Event Providers

gfProbes infrastructure allows clients to monitor glassfish even in the absence of provider classes. This is done by allowing clients to receive callbacks when a java methods are entered / exited. Note that while this approach allows a client to monitor legacy code, it may not always be possible to receive "high-level" events.

For example, while it is easy to monitor (through gfProbes) when TransactionManagerImpl.begin() entered / exited, the client cannot determine the transaction ID in this case.

```
public class TxMonitor {
     @MethodEntry("tx:com.sun.tx.TxMgrImpl::onTxBegin")
     public void onTx(String tId) {
       count++;
     }
  }
```

Monitoring Method Entry

The @MethodEntry annotation must be used by the client to receive callback when the target method is entered. The client method argument types and count must match the target methods parameter types/count.

Monitoring Method Exit

The @MethodExit annotation must be used by the client to receive callback when the target method is exited. The client method argument types and count must match the target methods parameter types/count. The first parameter in the client method should match the return type of the target method (only if the target method has a non void return type)

Monitoring Exceptions

The @OnException annotation must be used by the client to receive callback when the target method exits because of an exception. The client method argument types and count must match the target methods parameter types/count. (This restriction might be removed later). The first parameter in the client method should be of type Throwable

Registering an Event Listener

Registering an event listener enables the listener to receive callbacks from the Eclipse GlassFish 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 Eclipse GlassFish 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:

- The name of the configuration element with which all statistics in the event listener are to be associated. System administrators use this element for enabling or disabling monitoring for the event listener.
- The node in the monitorable object tree under which the event listener is to be registered. To specify the node, pass one of the following constants of the `org.glassfish.external.probe.provider.PluginPointPluginPoint` enumeration:
  - To register the listener under the `server/applications` node, pass the `APPLICATIONS` constant.
  - To register the listener under the `server` node, pass the `SERVER` constant.
- The path through the monitorable object tree from the node under which the event listener is registered down to the statistics in the event listener. The nodes in this path are separated by the slash (`/`) character.
- The listener object that you are registering.

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());
...
}
```

Dotted Names and REST URLs for an Add-On Component's Statistics

The Eclipse GlassFish administrative subcommands `get`, `list`, and `set` 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
```

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.

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.

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.

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
```

The formats of the URL to a REST resource that represents a statistic is as follows:

```
http://host:port/monitoring/domain/path
```

host

The host where the DAS is running.

port

The HTTP port or HTTPS port for administration.

path

The path to the statistic. The path is the dotted name of the attribute in which each dot (`.`) is replaced with a slash (`/`).

For example, the URL the REST resource for the `txcount` statistic that is defined in Example 5-5 and

registered in Example 5-7 is as follows:

```
http://localhost:4848/monitoring/domain/server/applications/tx/txapp/txcount
```

In this example, the DAS is running on the local host and the HTTP port for administration is 4848.

Adding a Type to the `monitor` Command

To add a type to the `monitor` command, implement the `MonitorContract` interface.

An implementation of the `MonitorContract` interface is an HK2 service that provides monitoring data to the `monitor` command.

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.

Class or Interface	Listing
`ModuleProbeProvider`	Example 5-8
`ModuleBootStrap`	Example 5-9
`ModuleStatsTelemetry`	Example 5-10
`Module`	Example 5-11
`ModuleMBean`	Example 5-12

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.

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.

```
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) {
    }

    /**
     * Emits probe event whenever the count is decremented
     */
    @Probe(name = "moduleCountDecrementEvent")
    public void moduleCountDecrementEvent(
            @ProbeParam("count") Integer count) {
    }
}
```

Example 5-9 Bootstrap Class

This example illustrates how to register an event listener as explained in Registering an Event Listener. 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.

```
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 Listener Class

This example shows how to perform the following tasks:

- Creating Event Listeners. The example shows the code of the `ModuleStatsTelemetry` listener class.
- Representing a Component's Statistics in an Event Listener Class. The example shows the code for representing the `countmbeancount` statistic.
- Subscribing to Events From Event Provider Classes. The example shows the code for subscribing to the following types of events from the `count` component:
  - `moduleCountIncrementEvent`
  - `moduleCountDecrementEvent`

```
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
 * Telemtry 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.

```
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. 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.

```
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;
    }

    @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. Eclipse GlassFish provides interfaces to enable an add-on component to store its configuration data in the same way as other Eclipse GlassFish components. These interfaces are similar to interfaces that are defined in [Jakarta XML Binding](https://jakarta.ee/specifications/xml-binding/). By using these interfaces to store configuration data, you ensure that the add-on component is fully integrated with Eclipse GlassFish. As a result, administrators can configure an add-on component in the same way as they can configure other Eclipse GlassFish components.

The following topics are addressed here:

How Eclipse GlassFish Stores Configuration Data

Eclipse GlassFish 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

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

Parent Element	Interface to Extend
`config`	`org.glassfish.api.admin.config.Container`
`applications`	`org.glassfish.api.admin.config.ApplicationName`
Another element that you are defining	`org.jvnet.hk2.config.ConfigBeanProxy`

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 {
...
}
```

How Interfaces That Are Annotated With @Configured Are Implemented

You are not required to implement any interfaces that you annotate with the `@Configured` annotation. Eclipse GlassFish implements these interfaces by using the `Dom` class. Eclipse GlassFish creates a 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.

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 Eclipse GlassFish 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

Example 6-2 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`.

```
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, Eclipse GlassFish 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. 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.
3. Represent the subelement as the property of a JavaBeans specification getter or setter method.
4. 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.

```
...
import org.jvnet.hk2.config.ConfigBeanProxy;
import org.jvnet.hk2.config.Configured;
...
@Configured
public interface WombatElement extends ConfigBeanProxy {
...
}
...
```

Example 6-4 Identifying a Subelement to its Parent Element

This example identifies the `wombat-element` element as a subelement.

```
...
import org.jvnet.hk2.config.Element;
...
import java.beans.PropertyVetoException;
...
@Element
    public WombatElement getElement();
    public void setElement(WombatElement element) throws PropertyVetoException;
...
```

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, Eclipse GlassFish uses Jakarta Bean Validation (`https://jakarta.ee/specifications/bean-validation/`) for validating configuration data. Jakarta Bean Validation 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 `jakarta.validation.constraints` package that performs the validation that you require. The following table lists commonly used annotations for validating Eclipse GlassFish configuration data. For the complete list of annotations, see the `jakarta.validation.constraints` package summary (`https://jakarta.ee/specifications/bean-validation/3.0/apidocs/jakarta/validation/constraints/package-summary.html`).

Table 6-1 Commonly Used Annotations for Validating Eclipse GlassFish Configuration Data

Validation	Annotation
Not null	`jakarta.validation.constraints.NotNull`
Null	`jakarta.validation.constraints.Null`
Minimum value	`jakarta.validation.constraints.Min` Set the `value` element of this annotation to the minimum allowed value.
Maximum value	`jakarta.validation.constraints.Max` Set the `value` element of this annotation to the maximum allowed value.

Validation	Annotation
Regular expression matching	`jakarta.validation.constraints.Pattern` Set the `regexp` element of this annotation to the regular expression that is to be matched.

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.

```
...
import jakarta.validation.constraints.Max;
import jakarta.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.

```
import jakarta.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
- To Write a Component's Initial Configuration Data to the `domain.xml` File

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 Adding Container Capabilities.

1. Set an optional dependency on an instance of the class that represents the XML element that you are adding.
2. 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`.

3. Annotate the declaration of the instance variable with the `org.jvnet.hk2.annotations.Inject` annotation.
4. Set the `optional` element of the `@Inject` annotation to `true`.
5. 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`
6. 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, Eclipse GlassFish 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>
      <http-service>
...
</domain>
```

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` subcommand. You can make the transaction available to system administrators in the following ways:

- By adding an `asadmin` 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 Extending the `asadmin` Utility.
- By extending the Administration Console. For more information, see 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.
3. Use the generic `SimpleConfigCode` interface to define the method that is to be invoked on a single configuration object, namely: `SingleConfigCode<T extends ConfigBeanProxy>()`.
4. In the body of this method, implement the `run` method of the `SingleConfigCode<T extends ConfigBeanProxy>` interface.
5. 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.
6. Invoke the static method `org.jvnet.hk2.config.ConfigSupport.ConfigSupport.apply`. In the invocation, pass the following information as parameters to the method:
   - The code of the method that you defined in Step 2.

- The configuration object to update, on which you set the dependency in Step 1.

Example 6-10 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.

```
...
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 and REST URLs of Configuration Attributes

The Eclipse GlassFish administrative subcommands `get`, `list`, and `set` locate a configuration attribute through the dotted name of the attribute. The dotted name of an attribute of a configuration element is as follows:

```
configs.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:

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

The formats of the URL to a REST resource that represent an attribute of a configuration element is as follows:

```
http://host:port/management/domain/path
```

host

The host where the DAS is running.

port

The HTTP port or HTTPS port for administration.

path

The path to the attribute. The path is the dotted name of the attribute in which each dot (`.`) is replaced with a slash (`/`).

For example, the URL to the REST resource for the `foo` attribute of the `wombat-element` element is as follows:

```
http://localhost:4848/management/domain/configs/config/server-config/wombat-container-config/wombat-element/foo
```

In this example, the DAS is running on the local host and the HTTP port for administration is 4848.

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.

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.

```
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 jakarta.validation.constraints.Pattern;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.Max;

import java.beans.PropertyVetoException;

@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 jakarta.validation.constraints.Min;
import jakarta.validation.constraints.Max;

import java.beans.PropertyVetoException;

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

The following topics are addressed here:

- Creating a `Container` Implementation
- Adding an Archive Type
- Creating Connector Modules
- Example of Adding Container Capabilities

Creating a `Container` Implementation

To implement a container that extends or replaces a service in Eclipse GlassFish, 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.

You should also run the HK2 Inhabitants Generator utility on your class files, which adds classes marked with the `@Service` annotation to the `META-INF/hk2-locator/default` file in your JAR file. For more information about the HK2 Inhabitants Generator, see the HK2 Inhabitants Generator page.

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:

```
package com.example.containers;
...

@Service @jakarta.inject.Named("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 Eclipse GlassFish by allowing applications to be deployed and run within the Eclipse GlassFish 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 Eclipse GlassFish.

`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 Eclipse GlassFish.

```
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`

```
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) {
        // stops and removes the application
        ...
    }

    public void clean (DeploymentContext dc) {
        // cleans up any artifacts generated during prepare()
        ...
    }
}
```

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, Eclipse GlassFish 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. Jakarta 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.

`handles(ReadableArchive archive)`

Returns a `boolean` value indicating whether this implementation of `ArchiveHandler` can work with the specified archive.

`getClassLoader(DeploymentContext dc)`

Returns a `java.lang.ClassLoader` capable of loading all classes from the archive passed in by the `DeploymentContext` instance. Typically the `ClassLoader` will load classes in the scratch directory area, returned by `DeploymentContext.getScratchDir()`, as stubs and other artifacts are generated in the scratch directory.

`expand(ReadableArchive source, WritableArchive target)`

Prepares the `ReadableArchive`'s source archive for loading into the container in a format the container accepts. Such preparation could be to expand a compressed archive, or possibly

nothing at all if the source archive format is already in a state that the container can handle. This method returns the archive as an instance of `WritableArchive`.

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. Eclipse GlassFish connector modules are separate from the associated add-on module that delivers the container implementation to allow Eclipse GlassFish to dynamically install and configure containers on demand.

When a deployment request is received by the Eclipse GlassFish 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 Eclipse GlassFish to recognize the container's application type. Eclipse GlassFish 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, Eclipse GlassFish 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. Eclipse GlassFish 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 Eclipse GlassFish; 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 Eclipse GlassFish

Package `Sniffer` implementation code into modules and install the modules in the as-install`/modules`

directory. Eclipse GlassFish will automatically discover these modules. If an administrator installs connector modules that contain`Sniffer` implementations while Eclipse GlassFish is running, Eclipse GlassFish 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
- Code for a web client of the container, which is shown in Web Client Code

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

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.

Class or Interface	Listing
`Greeter`	Example 7-3
`GreeterContainer`	Example 7-4
`GreeterContainer`	Example 7-5
`GreeterDeployer`	Example 7-6
`GreeterSniffer`	Example 7-7

Example 7-3 Annotation to Denote a Container's Component

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

```
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();
}
```

Example 7-4 Application Container Class

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

```
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;
    }

    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;
    }

    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;

/**
 * @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() };
        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.

Class or Resource	Listing
HelloWorld	Example 7-8 +
SimpleGreeter	Example 7-9 +
Deployment descriptor	Example 7-10 +

Example 7-8 Container Client Class

```
import components.SimpleGreeter;

import java.io.IOException;
import java.io.PrintWriter;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import jakarta.annotation.Resource;


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

    @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="3.1"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
</web-app>
```

Creating a Session Persistence Module

Eclipse GlassFish enables you to create a session persistence module in the web container for high availability-related functionality by implementing the `PersistenceStrategyBuilder` interface . Using the `PersistenceStrategyBuilder` interface in an HK2 service makes the session manager extensible because you can implement a new persistence type without having to modify the web container code.

For information about other high-availability, session persistence solutions, see "Configuring High Availability Session Persistence and Failover" in Eclipse GlassFish High Availability Administration Guide.

Implementing the `PersistenceStrategyBuilder` Interface

You can implement the `PersistenceStrategyBuilder` interface by creating a new web session manager type.

```
package com.sun.enterprise.web;

import com.sun.enterprise.deployment.runtime.web.SessionManager;
import org.apache.catalina.Context;
import org.jvnet.hk2.annotations.Contract;

@Contract
public interface PersistenceStrategyBuilder {

    public void initializePersistenceStrategy(
            Context ctx,
            SessionManager smBean,
            ServerConfigLookup serverConfigLookup);

    public void setPersistenceFrequency(String persistenceFrequency);

    public void setPersistenceScope(String persistenceScope);

    public void setPassedInPersistenceType(String persistenceType);
}
```

Here is an example of how to implement the `PersistenceStrategyBuilder` interface by creating a new web session manager and setting a store for it:

```
@Service(name="xyz")
public class XYZStrategyBuilder implements PersistenceStrategyBuilder {

    private String persistenceFrequency = null;
    private String persistenceScope = null;
    private String persistenceType = null;
```

```
    public void init(StandardContext ctx, SessionManager sessionManager,
      ServerConfigLookup serverConfigLookup) {
        // add listeners, valves, etc. to the ctx
        // Set the manager and store
    }

    public void setPersistenceFrequency(String persistenceFrequency) {
        this.persistenceFrequency = persistenceFrequency;
    }

    public void setPersistenceScope(String persistenceScope) {
        this.persistenceScope = persistenceScope;
    }

    public void setPassedInPersistenceType(String persistenceType) {
        this.passedInPersistenceType = persistenceType;
    }
}
```

If a `Manager` is provided, then it will be used in Eclipse GlassFish.

If a backing store is required, it is the responsibility of the `Manager` to make sure that the `findSession` method correctly uses the `Store` that the `Manager` provides.

Example 8-1 Implementing `PersistenceStrategyBuilder` With a Custom Web Session Manager

This example defines a session manager type that is named `MyHASolution`.

```
@Service(name="MyHASolution")
public class MyHASolutionStrategyBuilder implements PersistenceStrategyBuilder {

    private String persistenceFrequency = null;
    private String persistenceScope = null;
    private String persistenceType = null;

    public void init(StandardContext ctx, SessionManager sessionManager,
      ServerConfigLookup serverConfigLookup) {
        // add listeners, valves, etc. to the ctx
        // Set the manager and store
        MyManager myManager = new MyManager(persistenceType, persistenceFrequency);
        // (You could also make this a service and look it up in the habitat.
        // For simplicity we are just doing a new implementation of the class here.)
        MyStore store = new MyStore();
        myManager.setStore(store);
        ctx.setManager(myManager);
    }

    public void setPersistenceFrequency(String persistenceFrequency) {
        this.persistenceFrequency = persistenceFrequency;
```

```
    }

    public void setPersistenceScope(String persistenceScope) {
        this.persistenceScope = persistenceScope;
    }

    public void setPassedInPersistenceType(String persistenceType) {
        this.passedInPersistenceType = persistenceType;

    }

}
```

Example 8-2 Session Manager Configuration in the `glassfish-web.xml` File

This example sets the `persistence-type` attribute of the `session-manager` element of `glassfish-web.xml` to `myHASolution`

Based on the `domain.xml` and `glassfish-web.xml` settings, the web container looks up the appropriate `PersistenceStrategyBuilder` interface in the Habitat and uses it.

```
       <glassfish-web-app>
         <session-config>
           <session-manager persistence-type="myHASolution"/>
         <session-config>
       <glassfish-web-app>
```

Packaging, Integrating, and Delivering an Add-On Component

Packaging an add-on component enables the component to interact with the Eclipse GlassFish kernel in the same way as other components. Integrating a component with Eclipse GlassFish enables Eclipse GlassFish to discover the component at runtime.

The following topics are addressed here:

- Packaging an Add-On Component
- Integrating an Add-On Component With Eclipse GlassFish

Packaging an Add-On Component

To enable an add-on component to plug in to the Eclipse GlassFish 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 Eclipse GlassFish kernel, such as:

- Name
- Version
- Dependencies
- Capabilities

Integrating an Add-On Component With Eclipse GlassFish

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

Integration Point Reference

This appendix provides reference information about integration points, which are described in Extending the Administration Console. 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

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 Jakarta Server Faces page that contains the content to be integrated. Typically, the file contains a Jakarta Server Faces code fragment that is incorporated into a page. The code fragment often specifies a link to another Jakarta Server 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 Eclipse GlassFish 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

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 Jakarta Server 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 Eclipse GlassFish 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 Eclipse GlassFish. 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 Jakarta Server Faces page that contains the content to be integrated. When you use this integration point, your Jakarta Server 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 Jakarta Server 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`

`parentId`

The `id` value of the property sheet for the Configuration page. This value is `propSheetSection`, the section that contains the property definitions for the Configuration page.

`priority`

A numeric value that specifies the relative ordering of the item on the Configuration page.

`content`

A relative path to the Jakarta Server Faces page that contains the content to be integrated.

`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 Jakarta Server 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 `parentId` 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.

`content`

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 Jakarta Server 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`

`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 Jakarta Server Faces page.

For an example, see Example 3-15.

`org.glassfish.admingui:loginform` Integration Point

Use an `org.glassfish.admingui:loginform` integration point to specify the name and location of the include file containing the customized login form code. This code also contains the login background image used for the login page for 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:loginform`

`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 Jakarta Server 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 Jakarta Server Faces page.

For an example, see Example 3-15.