

OSGi Application Development using GlassFish Server

Version 1.4

By: Sanjeeb Sahoo

Table of Contents

1 Introduction:	3
1.1 New to OSGi?	3
1.2 Relationship between GlassFish Server & OSGi	4
2 OSGi Applications in GlassFish Server	4
2.1 Application Programming Interfaces (APIs)	5
3 Hybrid Application Bundle	6
3.1 Types of Hybrid Application Bundles	6
3.2 Hybrid Application Bundle Packaging and Deployment	7
3.3 Benefits of Hybrid Application Bundles	7
4 Web Application Bundle (WAB)	8
4.1 Structure of a WAB	8
4.2 WAB Metadata	8
4.3 WAB Life Cycle	9
4.4 Using OSGi Service	10
4.5 Fragment Bundle and WAB	10
4.6 Static Resources and WAB	10
4.7 Web Fragment and WAB	11
4.8 Supported technologies	11
4.9 WAR to WAB Conversion	11
5 EJB Application Bundle	12
5.1 Structure of EJB Application Bundle	12
5.2 Required Metadata	12
5.3 EJB Bundle Life Cycle	12
5.4 Consuming OSGi Service	12
6 Publishing an EJB as an OSGi Service	13
6.1 Export-EJB Manifest	13
6.2 Benefits of EJB based OSGi Service	14
7 Type-safe Injection of OSGi Services into Java EE Component	14
7.1 CDI	14
7.2 GlassFish Server OSGi/CDI Extension	14
7.3 Example	15
8 JPA in OSGi Application	15
8.1 Standalone Persistence Unit	16
8.2 Enhancement of JPA Entities	16
9 Java EE OSGi Services	16
9.1 HTTP Service	17
9.2 Transaction Service	18
9.3 JDBC Service	19
9.4 JMS Resource Service	19
10 Tools	19
10.1 Build Tools	19
10.2 Development Tools	20

10.3 Deployment Tools	20
10.4 Runtime Tools	21
10.4.1 Felix Remote Shell	21
10.4.2 GlassFish Server OSGi Web Console	22
11 Advanced Features	22
11.1 Setting Up an OBR	22
11.2 Using Alternative OSGi Runtime	23
11.2.1 Equinox	23
11.3 Embedded GlassFish	23
12 Future Direction	24
13 Additional Resources	24
14 References	24
15 Appendices	25
15.1 Appendix A – GlassFish OSGi/CDI API Javadoc	25
15.2 Appendix B – Setting up GlassFish Server to use OSGi/Java EE features (Only applicable for GlassFish 3.1)	27
15.3 Appendix C - Samples	27
15.4 Appendix D - TODO	29
15.5 Appendix E - Revision History	29

1 Introduction:

GlassFish Server Open Source Edition (GlassFish Server in short) [1] is built using OSGi module system [2] to provide features such as modularity, extensibility, and embedded support. OSGi is not only used to implement these features, it is now available as a first-class feature to application developers to deploy service oriented, modular, dynamic, extensible enterprise Java applications in GlassFish Server. This document describes how users can leverage OSGi to build and deploy such applications. In a nutshell, the following features of GlassFish Server are described in this document:

- OSGi Applications running in GlassFish Server
- Hybrid (Java EE-enabled OSGi) application bundles
- Modular web applications
- Building enterprise quality OSGi service using EJB technology
- Shared/Standalone Persistence Unit
- Injection of OSGi Services in Java EE applications using CDI
- Consuming Java EE Services from OSGi applications.

All features described in this document are available in both web profile as well as full profile distribution of GlassFish Server Open Source Edition.

Enterprise Expert Group in OSGi has been very actively working on standardising use of Java EE technologies in OSGi applications. So, some of the above mentioned features are standards based.

1.1 New to OSGi?

As described in OSGi Alliance Home Page [2],

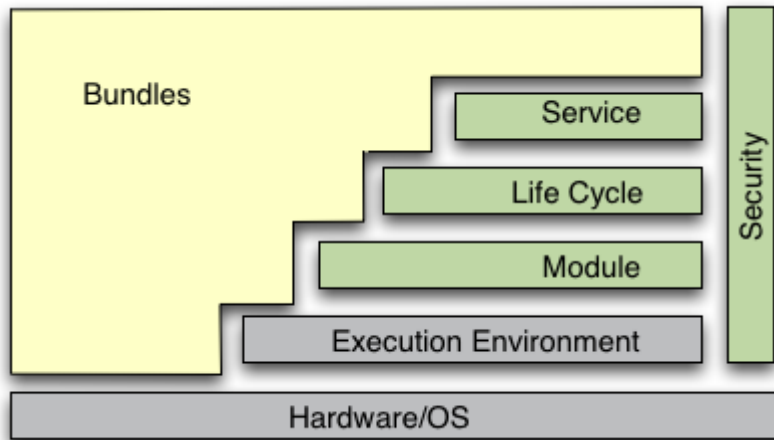
“OSGi technology provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software lifecycle.”

An OSGi framework is a module system and service platform for the Java programming language that implements a complete and dynamic component model, something that does not exist in standalone Java/VM environments. An OSGi framework forms the core of an OSGi technology based system. The framework provides a general-purpose, secure, and managed Java framework that supports the deployment of extensible and downloadable applications known as bundles or modules. The functionality of the Framework is divided in the following layers:

- **Module Layer:** Defines a modularization model, and is responsible for classloading and enforcing visibility rules.
- **Life Cycle Layer:** Provides an API to control life cycle of bundles. The life cycle is observable.
- **Service Layer:** Provides a programming model for defining and consuming services. Service consumers and suppliers can be completely decoupled.
- **Framework Services:** Framework also comes with a few very basic Services such as

Package Admin Service, Start Level Service, etc..

- **Security Layer:** Defines a secure packaging format as well as the runtime interaction with the Java security layer.



(Diagram #1: Various layers of OSGi System (Taken from OSGi Specification [3]))

OSGi application developers should consult OSGi Core framework specification [3] and many other useful materials available in the Internet to become familiar with OSGi.

1.2 Relationship between GlassFish Server & OSGi

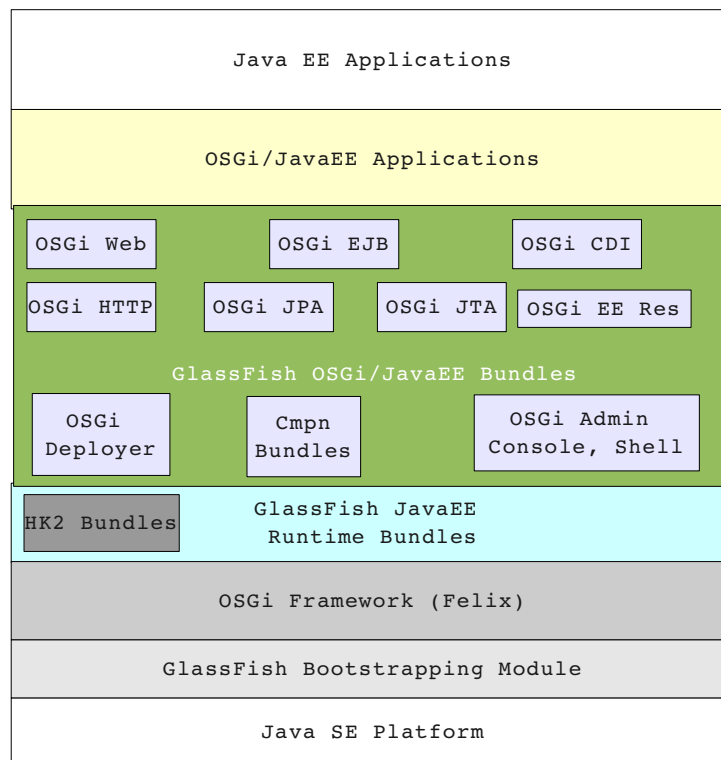
GlassFish Server runtime is implemented as a set of OSGi bundles. When GlassFish Server is started, first an OSGi runtime is bootstrapped and then these implementation bundles are deployed there. It is even possible to embed GlassFish Server inside an existing OSGi runtime. The default OSGi runtime started by GlassFish Server is Apache Felix [4], which is a fully compliant implementation of OSGi R4 version 4.2 Core Specification.

2 OSGi Applications in GlassFish Server

OSGi Applications are deployed as one or more OSGi bundle(s). GlassFish Server deployment and administration infrastructure have been enhanced to allow users to deploy and manage their OSGi bundles. GlassFish Server, being a fully compliant Java EE 6 container, not only provides the latest and greatest Java EE APIs but also exposes the Java EE component model and application framework to OSGi bundles. In other words, OSGi developers can now use Java EE component model like JSF, JSP, EJB, CDI, etc. That is useful because OSGi developers are able to leverage Java EE features while maintaining the benefits (and purity) of the OSGi service-based programming model. Use of any of these technologies is optional. Java EE platform services like the transaction service, http service, JMS, etc., are available as OSGi services, so OSGi bundles can use them via OSGi service registry as well.

GlassFish Server enables interaction between OSGi services and Java EE components. This interaction is bi-directional. OSGi services managed by OSGi framework can invoke Java EE components managed by Java EE container and vice versa. Application developers can declaratively export EJBs as OSGi services without having to write any OSGi code. That allows any pure OSGi component, which is running without the Java EE context, to discover the EJB and invoke it. That allows developers to write business components as EJBs so that they can take advantage of Java EE platform features such as declarative security, transaction management, dependency injection, etc., and yet allow them to be accessible to non-Java EE components. Similarly, Java EE components can locate OSGi services provided by non-Java EE OSGi bundles

and use them as well. GlassFish Server extends the platform default injection framework called Context and Dependency Injection (CDI) framework to make it a lot simpler for Java EE components to consume dynamic OSGi services in a type-safe manner.



(Diagram #2: GlassFish Process Constituents)

In this document, we classify OSGi bundles into two categories based on the features used by them:

- Plain vanilla OSGi bundles
These are bundles which do not contain any Java EE component in it.
- Java EE-enabled OSGi bundles (aka Hybrid Application Bundles)
These are OSGi bundles containing Java EE components in them. So, such a bundle is not only an OSGi bundle, but also a Java EE archive.

2.1 Application Programming Interfaces (APIs)

APIs available to OSGi applications are of two types:

- OSGi APIs:
As stated earlier, GlassFish Server comes with an OSGi R4, version 4.2 compliant framework, so all 4.2 core APIs are available. Moreover, GlassFish Server also comes bundled with a number of general purpose OSGi services which users can use from their bundle, such as:
 - OSGi Configuration Admin Service [7]
 - OSGi Event Admin Service [8]
 - OSGi Declarative Service [9]

GlassFish uses the implementations of these services from Apache Felix. Please refer to the Apache Felix documentation to learn more about these services.

- Java EE APIs:

A user-deployed OSGi bundle can not only use OSGi APIs, they can also use Java EE APIs. A point worth noting is that GlassFish Server 3.1.1 is a Java EE 6 compliant server, so users have access to the latest and greatest versions of these aforementioned APIs. Not all Java EE APIs are accessible to all kinds of OSGi bundles. For the purpose of this discussion, we can classify the APIs provided by a Java EE runtime into following three categories:

- **Class A: Components managed by the Java EE container**, such as EJB, CDI, Servlet, JSF, JAX-RS, etc. These are used to define components which are managed by the Java EE container. Since a vanilla OSGi bundle is not managed by the Java EE runtime, it can't leverage these APIs. An OSGi bundle has to become a hybrid application bundle to make use of these APIs. See [“Hybrid Application Bundles.”](#)
- **Class B: APIs to access underlying platform services**, such as JNDI, JTA, JDBC, JMS, etc. OSGi bundle developers can use these APIs as well. Typically one has to use JNDI to get access to the underlying service, but GlassFish actually makes the platform services available as OSGi services, so OSGi application developers can use OSGi Service APIs to access the services. Please refer to [“Java EE OSGi Services”](#) for more details.
- **Class C: Utilities like JAXB, JAXP, JPA, etc.**
Most of these APIs have a pluggability layer which allows an application to plug in a different implementation of the API. Typically the pluggability is achieved using the Java SE Service Provider mechanism. Typical implementations of these APIs rely on a Thread's context class loader to function correctly. However, the implementations of these APIs in GlassFish Server do not have such limitation – they are known to work when used from an OSGi bundle. So, an OSGi bundle deployed in GlassFish Server can safely use these APIs.

3 Hybrid Application Bundle

As defined earlier, a hybrid application bundle is an OSGi bundle as well as a Java EE module. At runtime, it has both an OSGi bundle context and a Java EE context. With hybrid applications, developers can continue to build standard and familiar enterprise application components, such as Java Servlets and EJBs, and take full advantage of:

- Features such as modularity/dependency management, service dynamism, and more provided by OSGi
- Infrastructure services such as transaction management, security, persistence, EIS access, bean validation, dependency injection, and more offered by Java EE.

3.1 Types of Hybrid Application Bundles

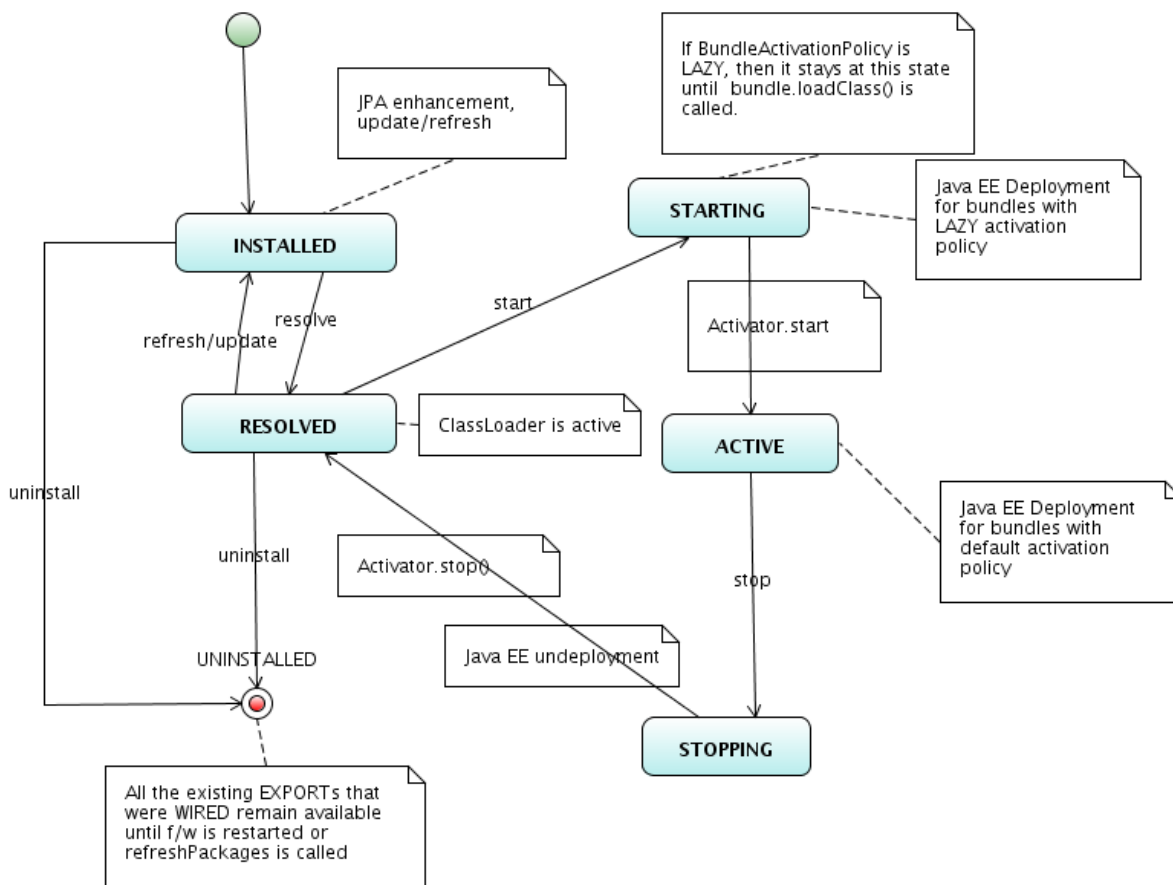
Currently, GlassFish Server supports web applications and EJB applications to be deployed as hybrid application bundles. Accordingly, we have two types of hybrid application bundles, viz:

- Web Application Bundle,
- EJB Application Bundle.

With the enhancements in Java EE 6, one can actually use a subset of EJB technology called EJB-lite in a web application itself. It won't be grossly incorrect to imagine a WAB's functionalities to be a superset of those provided by an EJB Application Bundle. We will talk more about these two types in following sections. Web Application Bundle is governed by a standard from OSGi Alliance.

3.2 Hybrid Application Bundle Packaging and Deployment

A hybrid application bundle is packaged as an OSGi bundle. Existing Java EE archive formats like the Web Archive (WAR) and EJB archive that are used for packaging standalone Java EE modules fit nicely with the OSGi bundle format. So, the application developer has to only add OSGi metadata in their Java EE archive to convert it into an OSGi bundle. When a hybrid application bundle is deployed in the OSGi runtime, GlassFish Server observes its life cycle and does necessary deployment/undeployment in/from Java EE container using the well known “extender pattern [11].” The life cycle of a hybrid application bundle is shown below:



(Diagram 3: Hybrid Application Bundle Life Cycle)

3.3 Benefits of Hybrid Application Bundles

Enterprise applications typically need transactional, secured access to data stores, messaging systems and other such enterprise information systems, have to cater to a wide variety of clients like web browsers and desktop applications, etc. Java EE makes development of such applications easier with a rich set of APIs and frameworks. It also provides a scalable, reliable and easy to administer runtime to host such applications.

OSGi platform complements these features with modularity. It enables applications to be separated into smaller, reusable modules with a well defined and robust dependency specification. A module explicitly specifies its capabilities and requirements. This explicit dependency specification encourages developers to visualize dependencies among their modules and help them make their modules highly cohesive and less coupled. OSGi module system is dynamic – it allows modules to be added and removed at runtime. OSGi has very good support for versioning – it supports package

versioning as well module versioning. In fact, it allows multiple versions of the same package to co-exist in the same runtime thus allowing greater flexibility to deployer. The service layer of OSGi platform encourages a more service oriented approach to build a system. The service oriented approach and dynamic module system used together allow a system to be more agile during development as well as runtime. It makes them better suited to run in an **Platform-as-a-Service (PaaS)** environment.

You no longer have to choose one of the two platforms. A hybrid approach like OSGi enabling your Java EE applications allows new capabilities to applications hitherto unavailable to applications built using just one of the two platforms.

4 Web Application Bundle (WAB)

When a web application is packaged and deployed as an OSGi bundle, it is called a Web Application Bundle (WAB). WAB support in GlassFish is based on “Web Applications Specification,” that's part of “**Enterprise OSGi Service Platform Enterprise Specification, Release 4, Version 4.2 [3].**”

4.1 Structure of a WAB

The Java Servlet Specification defines the structure of a web application [12]. It also defines a jar file-based archive file format called Web Archive (WAR) which is a portable way to package a web application. WAR is an exact replica of the hierarchical structure of web applications as defined in the Java Servlet Specification. On the other hand, a WAB is packaged as an OSGi bundle, so there is a possibility that WAB is not packaged like a WAR. When there is a mismatch in the structure, the OSGi Web Container of GlassFish Server will map the WAB to the hierarchical structure of Web Application using the following rules:

- *the root of the WAB corresponds to docroot of the Web Application.*
- *every jar in Bundle-ClassPath of the WAB is treated like a jar in WEB-INF/lib/.*
- *every directory except “.” in Bundle-ClassPath of the WAB is treated like WEB-INF/classes/.*
- *Bundle-ClassPath entry of type “.” is treated as if the entire WAB is a jar in WEB-INF/lib/.*
- *Bundle-ClassPath automatically includes the Bundle-ClassPath entries of any attached fragment bundles.*

The simplest way to avoid knowing these mapping rules is to avoid the problem in the first place. More over, there are many packaging tools and development time tools that understand WAR structure. So, it is strongly recommended that you package the WAB exactly like a WAR with additional OSGi metadata only.

4.2 WAB Metadata

In addition to the standard OSGi metadata, the main attributes of META-INF/MANIFEST.MF of the WAB must have an additional attribute called **Web-ContextPath**. The Web-ContextPath attribute specifies the value of the Context Path of the Web Application.

Since the root of a WAB is mapped to the docroot of the web application, it should not be used in the Bundle-ClassPath. More over, WEB-INF/classes/ should be specified ahead of WEB-INF/lib/ in the Bundle-ClassPath in order to be compliant with search order used for traditional WAR files.

Example: Assuming the WAB is structured as shown below:

```
foo.war/  
    index.html  
    foo.jsp  
    WEB-INF/classes/  
        foo/BarServlet.class  
    WEB-INF/lib/lib1.jar  
    WEB-INF/lib/lib2.jar
```

the OSGi metadata for the WAB as specified in META-INF/MANIFEST.MF of the WAB is shown below:

MANIFEST.MF:

```
Manifest-Version: 1.0  
Bundle-ManifestVersion: 2  
Bundle-SymbolicName: com.acme.foo  
Bundle-Version: 1.0  
Bundle-Name: Foo Web Application Bundle Version 1.0  
Import-Package: javax.servlet; javax.servlet.http  
Bundle-ClassPath: WEB-INF/classes, WEB-INF/lib/lib1.jar, WEB-INF/lib/lib2.jar  
Web-ContextPath: /foo
```

4.3 WAB Life Cycle

Refer to the life cycle as depicted in diagram #1. When the WAB is active, it gets deployed in the web container. When the WAB is stopped, it gets undeployed from web container. While deployment takes place asynchronously, undeployment happens synchronously. When a WAB is deployed in the web container, the associated ServletContext is registered in the OSGi service registry with following service property:

```
objectClass = javax.servlet.ServletContext  
osgi.web.contextpath = <Context Path of the web application>  
osgi.web.symbolicname = Bundle-SymbolicName of the WAB  
osgi.web.version = Bundle-Version of the WAB
```

Similarly, the BundleContext of the WAB is stored in an attribute called “**osgi-bundlecontext**” in the ServletContext, so any code that has access to ServletContext can access the BundleContext like this:

```
BundleContext.class.cast(  
    servletContext.getAttribute(“osgi-bundlecontext”));
```

The OSGi Web Container publishes events asynchronously using OSGi Event Admin service during deployment and undeployment of the WAB in the web runtime. For more details like event topics and event metadata, please refer to the OSGi Web Container specification.

4.4 Using OSGi Service

Since WAB has a valid Bundle-Context, it can consume OSGi services. Although developers are free to use any OSGi API to locate OSGi services, GlassFish Server makes it very easy for WAB users to use OSGi services by virtue of extending Context and Dependency Injection (CDI) framework. Given below is an example of injection of an OSGi Service into a Servlet:

```
@WebServlet
public class MyServlet extends HttpServlet {
    @Inject @OSGiService(dynamic=true)
    FooService fooService;
}
```

To learn more about this feature, please refer to the section [“Type-safe Injection of OSGi Services into Java EE Components.”](#)

4.5 Fragment Bundle and WAB

A WAB can't be a fragment bundle, but it can act as a host for other fragment bundles. A fragment bundle does not have its own class loader. All the fragments attached to a host share the class loader of the host bundle. The OSGi specification has well-defined rules that govern how a fragment bundle's Bundle-ClassPath is merged with host bundle's Bundle-ClassPath to come up with the effective Bundle-ClassPath. While mapping the content of a WAB to Web Application, effective Bundle-ClassPath is used, which means Bundle-ClassPath of all the attached fragments are automatically considered. Although fragments are not a great example of modularity, they can be used to provide additional content as discussed below.

4.6 Static Resources and WAB

Prior to Servlet 3.0 specification, the WEB-INF/ directory was only used to contain dynamic content like servlets, filters, utility classes, JSP tag libraries, configuration files, etc. That has changed in Servlet 3.0. One of the new features of Servlet 3.0 specification is that jar files from WEB-INF/lib/ directory are now allowed to contain static resources and JSPs. A jar file in WEB-INF/lib/ can package some static resources (say html files) and JSPs in META-INF/resources/ directory and they will become part of the Web Application's document root.

Jar file entries from the Bundle-ClassPath of a WAB are treated as if they are present in WEB-INF/lib/. It means this new feature of Servlet 3.0 specification is applicable to any jar file in Bundle-ClassPath as well. As a result, a jar from Bundle-ClassPath can now host static content and JSPs as long as the static content is packaged under META-INF/resources/ directory of the jar.

This new feature can further be extended to fragment bundles. As stated earlier, a fragment bundle's Bundle-ClassPath is merged with host WAB's Bundle-ClassPath to come up with the effective Bundle-ClassPath. So, any fragment bundle can also contain static resources and JSPs as long as they are packaged correctly. This simple yet powerful technique actually allows us to break our WAB into multiple bundles and develop/deploy them independently.

Example:

```
host.war/
    one.html
    two.jsp
    WEB-INF/classes/
    META-INF/
```

MANIFEST.MF

MANIFEST.MF contains:

Bundle-SymbolicName: host

Web-ContextPath: /webapp

Bundle-ClassPath: WEB-INF/classes

Bundle-ManifestVersion: 2

fragment.jar/

 META-INF/

 resources/

 bar/

 three.html

 four.jsp

 MANIFEST.MF

MANIFEST.MF contains:

Bundle-SymbolicName: fragment

Fragment-Host: host

Bundle-ManifestVersion: 2

When the host.war WAB is only deployed, /webapp/one.html and /webapp/two.jsp are only accessible. But, if fragment.jar is also deployed, not only /webapp/foo.html and /webapp/one.jsp but also /webapp/bar/three.html and /webapp/bar/four.jsp are accessible.

Please note bundle fragment is an OSGi R4 concept, so fragment bundle must have Bundle-ManifestVersion header set to 2 to indicate it as an R4 bundle, otherwise you may find unexpected results.

4.7 Web Fragment and WAB

With introduction of component defining annotations in Servlet 3.0 specification, the need for the web.xml has been reduced dramatically, yet sometimes it is necessary. Earlier, web.xml used to be a monolithic file packaged in WEB-INF/ directory of the Web Application. Servlet 3.0 introduces something called web-fragment.xml. It can be thought as a logical partitioning of the web.xml. As the name suggests, it is a fragment of web.xml. Each such fragment can be packaged inside a jar file in WEB-INF/lib. There can be many such web fragments in a Web Application and there even exists a mechanism to order them when conflicting instructions are present. So far, we have not discussed anything which makes it any more interesting than for the vanilla WAR file.

In case of a WAB, web fragments need not be packaged inside the WAB itself. Web fragments can be part of fragment bundles which can then attach to the WAB using the Fragment-Host attribute. As you can see, this allows a much greater degree of modularity than what is offered in the case of a traditional WAR.

4.8 Supported technologies

All Java EE technologies that can be used from regular a WAR can also be used from a WAB. This includes but not limited to Servlet 3.0, JSP 2.1, JSF 2.0, JPA 2.0, CDI 1.0, EJB 3.1, JAX-RS, resource injection, etc.

4.9 WAR to WAB Conversion

Web applications can also be installed as traditional WARs through a manifest rewriting process. OSGi Web Applications specification defines a protocol scheme called **webbundle** which can be used to install plain vanilla WAR files into the OSGi runtime. The corresponding URL handler transforms the input WAR to a WAB by adding necessary manifest entries. After the transformation, the WAR behaves like a WAB. The transformation process can be customized by use of various query parameters in the URL. For a detailed discussion on this, please refer to the OSGi Enterprise Specification. Here is an example of how one can use this in GlassFish:

```
install webbundle:http://acme.com:80/repo?webapp=foo.war?Web-ContextPath=/foo
```

5 EJB Application Bundle

The second kind of hybrid application bundle that is currently supported is the EJB Application Bundle. When an EJB Jar is packaged with additional OSGi metadata and deployed as an OSGi bundle it is called an EJB Application Bundle.

5.1 Structure of EJB Application Bundle

At this point, we only support packaging the OSGi bundle as a simple jar file with required OSGi metadata as you would have done for an ejb-jar.

5.2 Required Metadata

An EJB Application Bundle must have a manifest metadata called **Export-EJB** in order to be considered as an EJB Bundle. For syntax of Export-EJB header, please refer to the section [“Publishing EJB as OSGi Service.”](#)

Example of an EJB Application Bundle with metadata is given below:

```
myEjb.jar/
```

```
com/acme/Foo
```

```
com/acme/impl/FooEJB
```

```
META-INF/MANIFEST.MF
```

```
MANIFEST.MF:
```

```
Manifest-Version: 1.0
```

```
Bundle-ManifestVersion: 2
```

```
Bundle-SymbolicName: com.acme.foo EJB bundle
```

```
Bundle-Version: 1.0.0.BETA
```

```
Bundle-Name: com.acme.foo EJB bundle version 1.0.0.BETA
```

```
Export-EJB: ALL
```

```
Export-Package: com.acme; version=1.0
```

```
Import-Package: javax.ejb, com.acme; version=[1.0, 1.1)
```

5.3 EJB Bundle Life Cycle

The EJB Bundle life cycle is same as that of the WAB.

5.4 Consuming OSGi Service

Since an EJB Bundle has a valid Bundle-Context, it can consume OSGi services. Although

developers are free to use any API to locate OSGi services, GlassFish Server makes it very easy for users to use OSGi services by virtue of it extending Context and Dependency Injection (CDI) framework. Below is an example of the injection of OSGi Service into an EJB:

Example:

```
@Stateless
public class MyEJB {
    @Inject @OSGiService(dynamic=true)
    Foo foo;
    ...
}
```

To learn more about this feature, please refer to the section [“Type-safe Injection of OSGi Services into Java EE Components.”](#)

6 Publishing an EJB as an OSGi Service

Enterprise business services typically require transactional access to EIS systems such as database, message oriented middleware, etc. which are integrated with the security policies of the underlying system. Unfortunately there is no easy way to build services with such characteristics in the OSGi world, whereas there exist well-established APIs to do these in the Java EE platform. GlassFish Server provides a mechanism for a hybrid component model wherein an OSGi service is implemented as an EJB component and therefore able to use the aforementioned APIs. Once the EJB component manifests itself as an OSGi service, OSGi clients are agnostic to the implementation details of the service. This mechanism not only allows the use of Java EE infrastructure services and APIs in OSGi service implementations, but it also broadens the scope of usage of an EJB. In the future, this functionality could theoretically be extended to other component models like Managed Beans, CDI beans, etc.

OSGi Services are of two types: Singleton and Singleton-per-Bundle. In the former case, there is one service implementation instance shared between all the clients, and in the latter case there is one service implementation instance shared between all service consumers belonging to a bundle. On the other hand, the EJB specification defines a richer scoping model for EJB components such as Singleton, Stateless, and Stateful. As there is no natural support for defining the scope of an OSGi service, Stateful beans are not currently mapped as OSGi services. Using manifest metadata, an application developer can declaratively specify the list of EJBs that need to be exported as OSGi services. For all selected stateless and Singleton EJBs, the EJB extender bundle registers a proxy object in the service registry with additional service metadata as specified in the "Export-EJB" manifest header entry.

Each OSGi service can be associated with one or more service interfaces during service registration. When the EJB Extender bundle registers the proxy, it registers all the component interfaces of the EJB as the OSGi service interfaces. OSGi service lookup methods allow additional selection criteria which are used by the OSGi runtime during the service selection process. Once a suitable proxy is selected, it locates the delegate (underlying EJB instance) by performing a JNDI lookup using the portable JNDI name of the EJB to forward the method invocations. If an OSGi client begins a transaction using OSGi/JTA API, then the underlying EJB method invocation happens in the context of the same transaction. The EJB component developer can control the transactional behavior of their component through the transactional attribute as specified in the EJB specification.

6.1 Export-EJB Manifest

"Export-EJB" header has the following syntax

```
Export-EJB ::= ALL | NONE | (EJB-List)
EJB-List ::= (EJB-Desc ( ' , ' EJB-Desc )*)
EJB-Desc ::= (EJB Name)
```

Examples:

1. Following header will cause all applicable EJBs in the bundle to be exported as OSGi services

```
Export-EJB: ALL
```

2. The following header will cause none of the EJBs to be exported:

```
Export-EJB: NONE
```

3. The following header will cause EJBs with names com.acme.FooEJB and com.acme.BarEJB to be registered in OSGi Service Registry :

```
Export-EJB: com.acme.FooEJB, com.acme.BarEJB
```

6.2 Benefits of EJB based OSGi Service

As previously stated, EJB technologies make it a lot easier to build OSGi services which meet the needs of enterprise applications. An EJB which is exposed as an OSGi service is no longer restricted to be accessed from EJB clients only – it is accessible from OSGi applications as well.

7 Type-safe Injection of OSGi Services into Java EE Component

7.1 CDI

The Contexts and Dependency Injection (CDI) specification (JSR-299) defines a set of complementary services that help improve the structure of application code. CDI provides an enhanced lifecycle and interaction model over existing Java component types, including managed beans and Enterprise Java Beans. The CDI services provide:

- an improved lifecycle for stateful objects, bound to well-defined contexts
- a type-safe approach to dependency injection
- an SPI for developing portable extensions to the container.

The CDI framework can be extended in a portable manner through extensions. A CDI portable extension integrates with the container and:

- * can provide its own beans, interceptors and decorators to the container
- * can inject dependencies into its own objects using the dependency injection service

The Java EE Platform allows Java EE components to be injected with their dependencies through standard dependency injection (`@Resource` etc) and CDI-style injection (`@Inject`) APIs. There is no easy way for a Java EE component to

- Easily express dependencies to OSGi Service
- A type-safe mechanism to state such dependencies

- Discover and bind to OSGi Services which takes advantage of OSGi Service Dynamism.

7.2 GlassFish Server OSGi/CDI Extension

GlassFish Server comes with a CDI extension that enables Java EE components (such as Servlets, EJBs, etc) that are part of hybrid application bundles to express a type-safe dependency on an OSGi Service using CDI APIs. Note that an OSGi Service can be provided by any OSGi Bundle without any knowledge of Java EE/CDI and they are allowed to be injected transparently in a type-safe manner into a Java EE component.

A custom CDI Qualifier[1], `@org.glassfish.osgi CDI.OSGiService`, is used by the component to represent dependency on an OSGi Service. The Qualifier has additional metadata to customize the service discovery and injection behavior. The following `@OSGiService` attributes are currently available:

- **dynamic:** Dynamically obtain a service reference (true/false)
- **waitTimeout:** Waits for specified duration for a service to appear in the OSGi service registry
- **serviceCriteria:** An LDAP filter query used for service selection

7.3 Example

Bundle B0 defines a Service contract called `com.acme.Foo` and exports `com.acme` package for use by other bundles. Bundle B1 in turn provides a Service Implementation, `FooImpl`, of that interface `com.acme.Foo`. It then registers this Service `FooImpl` with the OSGi service registry with `com.acme.Foo` as the Service interface.

Bundle B2 is a hybrid application bundle that imports `com.acme` package. It has a component called `BarServlet` that expresses a dependency to `com.acme.Foo` by adding a field/setter method and qualifies that Injection point with `@OSGiService`. So for instance, `BarServlet` could look like:

```
@Servlet
public void BarServlet extends HttpServlet{
    @Inject @OSGiService(dynamic=true)
    private com.acme.Foo f;
}
```

8 JPA in OSGi Application

It is a very common requirement for business applications to deal with data stores and Java Persistence API is the standard API in this problem domain. This API can be used by *managed* as well as *non-managed* code. An OSGi bundle's activator is non-managed code, where as a Java EE component that's part of a hybrid application bundle is managed code. Except for the bootstrapping part, the rest of the API remains same in either case.

The primary interface for the application developer is `EntityManager` which can be produced by an `EntityManagerFactory`. `EntityManagerFactory` is a runtime representation of the *Persistence Unit* which is defined in an XML file called `META-INF/persistence.xml`. An application can define more than one persistence units in an application using `persistence.xml`.

In a managed environment, either both an `EntityManagerFactory` or an `EntityManager` can be obtained from JNDI or be injected into Java EE components. These are commonly referred

to as “Java EE” bootstrapping APIs. In a non-managed environment, the user has to call `Persistence.createEntityManagerFactory()` method to get a handle to `EntityManagerFactory`. This is commonly referred to as “Java SE” bootstrapping API. Although the Java SE bootstrapping API can be called by a managed component, it is not recommended because of better alternatives available to such code.

The Java SE bootstrapping API relies on Thread's context class loader to be set properly for two reasons:

- to discover persistence units and JPA domain classes,
- to discover Persistence Provider.

While it is easy to work around the first one by just setting the context class loader to the bundle class loader before making the call to the `Persistence.createEntityManagerFactory`, the latter is hard to fix, as the bundle does not typically have a direct dependency on Persistence Provider. Fortunately, GlassFish Server can influence the resolution of persistence providers using a custom implementation of JPA SPI called

`javax.persistence.spi.PersistenceProviderResolver`. This custom resolver can be enabled by setting the following property to true in

`$GLASSFISH_HOME/glassfish/osgi/felix/conf/config.properties`:

```
org.glassfish.osgi.jpa.extension.useHybridPersistenceProviderResolver=true
```

It is by default set to false in that file.

If you are using 3.1.2 or above, then the config file path is

`$GLASSFISH_HOME/glassfish/config/osgi.properties`, but you don't have to edit this file, as the following `asadmin` command will set a system property that can override the value:

```
asadmin create-jvm-options --target server-config  
-Dorg.glassfish.osgi.jpa.extension.useHybridPersistenceProviderResolver=true
```

An entity manager factory created by using `Persistence.createEntityManagerFactory()` method has to be explicitly configured about the server environment so that the persistence provider can talk to appropriate transaction manager, validation provider, etc. While using EclipseLink as a persistence provider, this can be specified by using the following property in `persistence.xml` file or while by passing this in the properties object while invoking `createEntityManagerFactory` method:

```
eclipselink.target-server=SunAS9
```

8.1 Standalone Persistence Unit

In a standard Java EE application, the scope of a persistence unit is limited to the deployment unit, i.e., the `ear/jar/war`. If there are multiple applications that share the same domain model stored in the same datastore, then there is a separate copy of the domain classes and `persistence.xml` packaged into each of those applications. This affects the size and performance of the runtime, because the “second level cache,” which is more-or-less the norm in every persistence provider is scoped to a persistence unit.

In GlassFish, one can package JPA domain classes in a separate OSGi bundle, deploy that bundle alone, export an `EntityManagerFactory` as an OSGi service from such a bundle such that other OSGi bundles can reference the `EntityManagerFactory`. This leads to improved modularity/reuse, better performance because of shared 2nd level cache. Refer to the examples to see it being used.

8.2 Enhancement of JPA Entities

Certain persistence provider like EclipseLink rely on byte code enhancement (or weaving) to provide lazy loading support for one-to-one and many-to-one type relationships. There is no

standard way to enhance classes loaded from bundle space in an OSGi environment. The solution GlassFish Server implements is to statically enhance JPA domain classes using PersistenceProvider library when the corresponding JPA bundle is installed or updated in the system. This functionality is currently only provided with EclipseLink. For this functionality to work, user must either specify the following element in their persistence-unit:

```
<exclude-unlisted-class>false</exclude-unlisted-classes>
```

or

enumerate all the JPA domain classes in their persistence-unit using

```
<class>entity class name</class>
```

9 Java EE OSGi Services

Java EE components (like an EJB or Servlet) can lookup Java EE platform services using JNDI names in the Java EE naming context associated. Such code can rely on Java EE container to inject the required services as well. Unfortunately, neither of them works when the code runs outside a Java EE context. Examples of such code is BundleActivator of an OSGi bundle. For such code to access Java EE platform services, GlassFish makes available key services and resources of underlying Java EE platform as OSGi services. Thus, an OSGi bundle deployed in GlassFish can access these services using OSGi Service look up APIs or using white board pattern. Given below are the list of services of underlying Java EE platform that are available as OSGi services:

- HTTP Service
- Transaction Service
- JDBC Data Source Service
- JMS Resource Service

The table below describes the services, their interfaces and associated service properties that can be used during service selection.

Table #1: Java EE Service to OSGi Service Mapping

Underlying Service	Service Interface	Service Properties	Comments
HTTP Service	org.osgi.service.http.HttpService [10]	VirtualServer=<Virtual Server Name>	Each HTTP Service corresponds to one virtual server in GlassFish.
Transaction Service	javax.transaction.UserTransaction, javax.transaction.TransactionManager, javax.transaction.TransactionSynchronizationRegistry	None	Use UserTransaction interface for transaction demarcation.
JDBC Data Source	javax.sql.DataSource	jndi-name = <JNDI name of the data source> osgi.jdbc.driver.class = <Fully Qualified Class	User can create/delete JDBC datasources using GlassFish Administration API and they are synchronously

		Name of the Driver class>	registered/unregistered from OSGi service registry.
JMS Queue	javax.jms.Queue	jndi-name = <JNDI name of the Queue>	
JMS Topic	javax.jms.Topic	jndi-name = <JNDI name of the Queue>	
JMS Connection Factory	javax.jms.QueueConnectionFactory or javax.jms.TopicConnectionFactory or javax.jms.ConnectionFactory	Jndi-name = <JNDI name of the connection factory>	Actual service interface depends on what is selected by user while creating the connection factory.

9.1 HTTP Service

The GlassFish Server web container is made available as a service for OSGi users who do not use a WAB to develop web applications. This service is made available using the standard OSGi/HTTP service specification [10], which is a very light API that predates the concept of a Web Application as we know today. This simple API allows users to register servlets and static resources dynamically and draw a boundary around them in the form of a `HttpContext`. This simple API can be used to build very feature rich web application. The Felix Web Console [13] is a good example of this.

This particular functionality is provided by a separate GlassFish Server OSGi bundle which is not by default available in a GlassFish installation. One has to download the bundle from GlassFish Server update center or from maven.

The GlassFish Server web container has a notion of a virtual server. The hierarchy in GlassFish Server is:

GlassFish Web Container has one or more virtual server(s).

A virtual server has one or more web application deployed in it. Each web application has a distinct context path. Each virtual server has a set of HTTP listeners. Each HTTP listener listens on a particular port. When multiple virtual servers are present, one of them is treated as the “default virtual server.” Every virtual server comes configured with a default web application. The default web application is used to serve static content from docroot of GlassFish Server. This default web application uses “/” as the context path.

A web application contains one or more wrappers (Servlets).

Each virtual server is mapped to an `org.osgi.services.http.HttpService` instance. When there are multiple virtual servers present, obviously there will be multiple `HttpServices` registered in the service registry as well. In order to distinguish one service from another, each service is registered with a service property called “**VirtualServer**” whose value is the name of the virtual server. Moreover, the service corresponding to “default virtual server” has the highest ranking, so when someone just does a service lookup of `HttpService` without any additional criteria, they get the service corresponding to the default virtual server – which is the desired behavior. In a typical GlassFish Server installation, the default virtual server is configured to listen on port 8080 and 8181 for http and https protocol respectively.

Since the default web application uses the context path “/,” every resource and servlet registered using `registerResource()` and `registerServlet()` methods of `HttpService` is made available under a context path `/osgi` in the virtual server. The context path `/osgi` can be changed to some other value by setting an appropriate value in OSGi configuration property or system property called

```
org.glassfish.osgihttp.ContextPath.
```

For example, HelloWorldServlet will be available at

```
http://localhost:8080/osgi/helloworld
```

when the following code is executed:

```
HttpService httpService = getHttpService(); // Obtain HttpService
httpService.registerServlet(httpService.registerServlet("/helloworld", new HelloWorldServlet(),
null, ctx);
```

9.2 Transaction Service

Java Transaction API (JTA) defines three interfaces to interact with transaction management system, viz: `UserTransaction`, `TransactionManager` and `TransactionSynchronizationRegistry`. They all belong to package `javax.transaction`. But for the first one, the rest two are intended for system level code, e.g., a persistence provider. As the name suggests, `UserTransaction` is the entity that application programmers should use to control transactions. All these objects of the underlying JTA layer is made available in OSGi service registry with following service interfaces respectively:

- `javax.transaction.UserTransaction`
- `javax.transaction.TransactionManager`
- `javax.transaction.TransactionSynchronisationRegistry`

There is no additional service property associated with them. Although `UserTransaction` appears to be a singleton, in reality any call to it gets rerouted to the the actual transaction associated with the calling thread.

As you know, code which runs in the context of Java EE component gets hold of `UserTransaction` typically by doing a JNDI lookup in the component naming context or using injection as shown below:

```
(UserTransaction)(new InitialContext().lookup("java:comp/UserTransaction"));
```

or

```
@Resource UserTransaction utx;
```

When some code (let's consider an OSGi Bundle Activator) which does not have Java EE component context wants to get hold of `UserTransaction` or any of the other JTA artifacts, then they can lookup service registry. Given below is an example of such code:

```
BundleContext context;
ServiceReference txRef =
    context.getServiceReference(UserTransaction.class.getName());
UserTransaction utx = (UserTransaction);
context.getService(txRef);
```

Refer to OSGi/JTA sample for further details.

9.3 JDBC Service

Any JDBC `DataSource` created in GlassFish is automatically made available as an OSGi Service, so OSGi bundles can track availability of JDBC data source using `ServiceTracking` facility of OSGi platform. Please consult GlassFish administration guide to learn how to administer JDBC resources in GlassFish. The life of the OSGi service matches that of the underlying data source created in

GlassFish. As specified in table #1, the OSGi service is registered with objectClass = "javax.sql.DataSource" and service property called jndi-name which contains the JNDI name of the datasource. Sample code that looks up a Data Source service is shown below:

```
@Inject
@OSGiService(true, "(jndi-name=jdbc/MyDS)")
private DataSource ds;
```

9.4 JMS Resource Service

Like JDBC data sources, JMS administered objects like topics, queues and connection factories are also automatically made available as OSGi services. Table #1 details the service details. With this, any OSGi bundle can use the underlying JMS system as well.

10 Tools

10.1 Build Tools

bnd [19] is the de-facto tool used by OSGi users. It can just be used to generate the manifest or it can be used to package an OSGi bundle as well. It can be used in Ant and maven environment as well. The maven plugin of bnd, known as "maven-bundle-plugin [20]," is developed in Felix project and is very flexible. In our samples, we use maven-bundle-plugin and we recommend certain configurations of the plugin to make your life easier.

10.2 Development Tools

Hybrid Applications are a new kind of applications. Even though it may appear that many IDEs don't support development of such applications, it's actually a myth. Most IDEs are well designed and the functions they provide are additive in nature. Our recommendation is to use maven based Java EE project in those IDEs so that you can use maven-bundle-plugin to generate the necessary OSGi metadata and use the rest of the IDE wizards to do the Java EE tasks. We have tutorials available at [17] showing how to do this in NetBeans and Eclipse.

10.3 Deployment Tools

These are the various ways one can deploy OSGi bundles in GlassFish. A brief overview of various OSGi bundle deployment options is given below. Please refer to GlassFish Administration Guide [6] for more details.

- Autodeploy directory:

GlassFish Server monitors the `${domain_dir}/autodeploy/bundles/` directory for OSGi bundles, so you can actually use File operations like cp, rm to install/update, uninstall OSGi bundles. You can actually install, update, uninstall multiple bundles at a time using autodeploy directory. Moreover, one can configure GlassFish to watch as many directories as they like. This functionality in GlassFish is provided by Apache Felix FileInstall [5] module, so you can find much more detailed documentation there.

- CLI:

```
asadmin deploy -type osgi <path to bundle jar>
```

- Admin Console:

While deploying OSGi bundles, select type as “other” and then select “osgi” in the available options.

- Felix Remote Shell:

GlassFish Server includes Felix remote shell which can be used to administer OSGi runtime.

- OSGi Web Console:

The GlassFish Server Administration Console has an additional plugin that enables user to administer the OSGi runtime as well.

- OBR:

It is possible to configure GlassFish Server to use an OBR repository.

Since GlassFish is extensible, application developers can actually deploy their own or third-party OSGi management bundle in the system if they wish to do so.

10.4 Runtime Tools

GlassFish Server provides a couple of tools to user to help them manage and debug the OSGi runtime. They are explained below. OSGi being an extensible platform, you can install your own favorite tool to do the needful as well. Currently, these tools are only supposed to be used in development environment, as they lack are not integrated with GlassFish's Server authentication and authorization policy.

10.4.1 Felix Remote Shell

GlassFish Server comes preconfigured with Apache Felix Gogo, which is an implementation of OSGi RFC 147, and is a proposed standard shell for OSGi environments. The shell is more advanced and supports expected shell features (e.g., scripting, piping, variables) as well as more advanced features (e.g., closures, dynamic method invocation). This shell is accessible via telnet protocol and the default port is 6666. This shell is turned off by default in production builds. To enable it, Felix Remote Shell bundle needs to be activated. This bundle is already installed as part of GlassFish runtime, but not activated by default. Steps to activate the shell for various GlassFish releases are given below:

For GlassFish 3.1.2 and above: We have improved the configuration to leverage OSGi start level service to enable the shell. By default the final start level of the framework is set to 2 using a property called `glassfish.osgi.start.level.final` in `glassfish/config/osgi.properties`. You can either change it to 3 to enable the shell or override its value using a system property with same name. e.g., the following `asadmin` command can be used to override its value:

```
asadmin create-jvm-options --target server-config -Dglassfish.osgi.start.level.final=3
```

For GlassFish 3.1.1: Please add **org.apache.felix.shell.remote** to the list of bundles mentioned in the property called **optional.bundles** in `glassfish/osgi/felix/config/config.properties` file

For GlassFish 3.1: Please refer to [section #15.2](#) for instructions to enable the shell.

The shell is only available on loopback interface (i.e., 127.0.0.1). So, one can connect to it by running:

```
telnet localhost 6666
```

```
->
```

Some of the typical commands and their usage is given below:

Command Name	Description	Examples
help	Show help messages for a command	help -> Show available commands help inspect-> Shows description of inspect command
lb	List Bundles	lb lb -l grep foo.jar lb foo
bundle	Show details about a bundle	bundle 0 -> Shows details about (bundle 1) location ->
inspect	Inspect capabilities and requirements of a bundle	inspect p c 1 -> Show packages exported by bundle 1 inspect p r 1 -> Show packages imported by bundle 1 inspect s c 1 -> Show services registered by bundle 1 inspect s r 1 -> Show services referenced by bundle 1
which	determines from where a bundle loads a class	which 1 com.acme.Foo -> Shows from where bundle 1 loads com.acme.Foo class
install	Install a bundle from a URL	install file:/tmp/a.jar install http://acme.com/bar.jar
start	Start a bundle	start 7 start file:/tmp/a.jar -> Install from the URL and start it start -t 7 -> Start bundle 7 transiently
stop	Stop a bundle	stop 7
uninstall	Uninstall a bundle	uninstall 7
update	Update a bundle	update 7

This is a very very powerful shell. To explore its functionalities, please refer to [15] and [16].

10.4.2 GlassFish Server OSGi Web Console

GlassFish Server OSGi Web Console is a customized version of the Apache Felix Web Console. This is by default not configured in the runtime. There are two ways one can install the console and its dependencies as shown below:

1. Install the update center package called “Glassfish OSGi Admin Console” using GlassFish Server Admin Console or the `pkg` command line utility.
2. Download the following packages from maven and unzip them on top of your GlassFish Server installation:

<http://maven.glassfish.org/content/groups/glassfish/org/glassfish/packager/glassfish-osgi-http/3.1.1/glassfish-osgi-http-3.1.1.zip>

<http://maven.glassfish.org/content/groups/glassfish/org/glassfish/packager/glassfish-osgi-gui/3.1.1/glassfish-osgi-gui-3.1.1.zip>

While unzipping, make sure the jars are unzipped to autostart/ directory. There is no need to restart GlassFish Server, the changes will take effect automatically. You can now access the web console either directly going to <http://localhost:8080/osgi/system/console/> or via by visting OSGi Console link in GlassFish Admin Console, which is available at <http://localhost:4848/>. GlassFish OSGi Web Console uses its own security domain, so upon promoted for user name and password, use admin/admin.

You may find further information at [13] useful. If you are using GlassFish Server 3.1, replace 3.1.1 as 3.1 in the above URLs.

11 Advanced Features

11.1 Setting Up an OBR

To be documented

11.2 Using Alternative OSGi Runtime

11.2.1 Equinox

Although GlassFish Server comes with Felix pre-installed, it's pretty easy to make it run on Equinox and other platforms. Given below are the simple steps...

1. Download Equinox or if you have it, use that.
2. `cp org.eclipse.osgi_3.3.2.R33x_v20080105.jar $GLASSFISH_HOME/osgi/equinox/`
(replace 3.3.2.R33x_v20080105 by the actual version found in jar name in your equinox installation. It changes from version to version.)
3. Start GlassFish Server, but while starting let it know that you want to run on Equinox, else by default it uses Felix. To do this, you have couple of choices:
 - You can set an environment variable called GlassFish_Platform as Equinox.
 - You can set a system property called GlassFish_Platform as Equinox.

Why do we have these two options? The system property is handy when you are starting using "java -jar" command. The environment variable is useful when you are starting using the classic way, i.e., "asadmin start-domain." Putting them in practice: (I am using Bash shell in the following example)

1. In the following example, we set the option once in the environment and every subsequent use of "asadmin start-domain" starts GlassFish Server on Equinox.

```
export GlassFish_Platform=Equinox
asadmin start-domain
```

2. In the example below, asadmin start-domain is used to start GlassFish, but you are able to specify the enviornment variable on the same command.

```
GlassFish_Platform=Equinox asadmin start-domain
```

3. If you are used to "java -jar" style of starting GlassFish (a new thing in GlassFish v3), then do this:

```
java -DGlassFish_Platform=Equinox $GlassFish_HOME/modules/glassfish.jar
```

The Equinox config file is located in
\$GLASSFISH_HOME/osgi/equinox/configuration/config.ini

11.3 Embedded GlassFish

It is quite easy to embed GlassFish Server in an OSGi environment. One has to download either the web profile or full profile distribution of GlassFish Server and install it. Assuming you have downloaded and installed either of the aforementioned glassfish distributions in /tmp, then there are two ways to embed GlassFish in your OSGi environment:

Option #1: Use GlassFish bundle to control life cycle of GlassFish.

Your code can look like this:

```
BundleContext bundleContext;  
Bundle gfBundle =  
bundleContext.install("file:///tmp/glassfish3/glassfish/modules/glassfish.jar");  
gfBundle.start(); // Starts GlassFish  
gfBundle.stop(); // Stops GlassFish
```

When you install glassfish.jar with a location string as shown above, the activator in GlassFish can deduce the GlassFish installation location as well as GlassFish domain directory location. If you chose to use a different location string while installing GlassFish bundle, then you can specify the installation location and domain directory using following OSGi configuration properties:

com.sun.aas.installRoot -> where is glassfish installed? e.g., /tmp/glassfish3/glassfish/

com.sun.aas.instanceRoot -> where glassfish domain directory is located? e.g.,
/tmp/glassfish3/glassfish/domains/domain1

12 Future Direction

We will explore managed bean and CDI bean as basis for OSGi services. We welcome your suggestions and feedbacks. You can file bugs, RFEs in our bug tracking system. We will also allow EAR files to be deployed as a set of OSGi bundles to provide a more isolated runtime.

13 Additional Resources

We try to consolidate presentation material, blogs, etc. in a central place in our publicly accessible Wiki page [17]. The GlassFish community has an outstanding track record of providing timely responses to user's queries. If you need help, please use our forum available at [18]. Users who have purchased support contract for Oracle GlassFish Server can consult their support agreement to know whom to contact.

14 References

[1] GlassFish Server:

<http://GlassFish.org>

[2] OSGi Module System:

<http://www.osgi.org>

[3] OSGi Specifications:

<http://www.osgi.org/Specifications/HomePage>

[4] Apache Felix OSGi implementation:

<http://felix.apache.org>

[5] Apache Felix FileInstall Bundle:

<http://felix.apache.org/site/apache-felix-file-install.html>

[6] GlassFish Server Open Source Edition Administration Guide:

TBD

[7] Felix Configuration Admin Service

<http://felix.apache.org/site/apache-felix-config-admin.html>

[8] Felix Service Component Runtime

<http://felix.apache.org/site/apache-felix-service-component-runtime.html>

[9] Felix Event Admin Service:

<http://felix.apache.org/site/apache-felix-event-admin.html>

[10] OSGi/HTTP Service:

<http://www.osgi.org/javadoc/r4v42/org/osgi/service/http/HttpService.html>

[11] Extender Pattern in OSGi:

<http://www.osgi.org/blog/2007/02/osgi-extender-model.html>

[12] Web Application Deployment Hierarchy

Section 10.4 of Java Servlet Specification version 3.0

[13] Felix Web Console:

<http://felix.apache.org/site/apache-felix-web-console.html>

[14] CDI Qualifier:

<http://download.oracle.com/javaee/6/api/javax/inject/Qualifier.html>

[15] Gogo Shell Language:

<http://www.packtpub.com/article/apache-felix-gogo>

[16] Apache Felix Gogo Shell:

<http://felix.apache.org/site/apache-felix-gogo.html>

[17] GlassFish OSGi Wiki:

<http://wikis.sun.com/display/GlassFish/Osgi>

[18] GlassFish forum:

<http://www.java.net/forums/glassfish/glassfish>

[19] bnd:

<http://www.aqute.biz/Code/Bnd>

[20] maven-bundle-plugin:

<http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html>

[21] Glassfish JIRA

<http://java.net/jira/browse/GLASSFISH>

15 Appendices

15.1 Appendix A – GlassFish OSGi/CDI API Javadoc

Package: org.glassfish.osgi CDI

Version: 1.0

OSGiService Annotation

```
@Qualifier
@Target(value={TYPE, METHOD, PARAMETER, FIELD})
@Retention(value=RUNTIME)
public @interface OSGiService
```

A CDI (JSR-299) Qualifier that indicates a reference to a Service in the OSGi service registry that needs to be injected into a Bean/Java EE Component. A Java EE component developer uses this annotation to indicate that the injection point needs to be injected with an OSGi service and can also provide additional meta-data to aid in service discovery. If this qualifier annotates an injection point, the `OSGiServiceExtension` discovers and instantiates the service implementing the service interface type of the injection point, and makes it available for injection to that injection point.

dynamic

```
public abstract boolean dynamic
```

Determines if the OSGi service that is to be injected refers to a dynamic instance of the service or is statically bound to the service implementation discovered at the time of injection. If the value of this annotation element is true, a proxy to the service interface is returned to the client. When the service is used, an active instance of the service at that point in time is used. If a service instance that was obtained earlier has gone away (deregistered by the service provider or stopped), then a new instance of the service is obtained from the OSGi service registry. This is ideal for stateless and/or idempotent services or service implementations whose lifecycle may be shorter than the client's lifecycle. If the value of this annotation element is false, an instance of the service is obtained from the service registry at the time of injection and provided to the client. If the service implementation provider deregisters the obtained service or the service instance is stopped, no attempt is made to get another instance of the service and a `ServiceUnavailableException` is thrown on method invocation. This is ideal for stateful or contextual services and for references to service implementations whose lifecycle is well-known and is known to be greater than the lifecycle of the client.

Default:

false

serviceCriteria

```
public abstract java.lang.String serviceCriteria
```

Service discovery criteria. The string provided must match the Filter syntax specified in the OSGi Core Specification.

Default:

""

waitTimeout

public abstract int **waitTimeout**

Waits, for the specified milliseconds, for at least one service that matches the criteria specified to be available in the OSGi Service registry. 0 indicates indefinite wait. -1 indicates that the service is returned immediately if available or a null is returned if not available.

Default:

-1

ServiceUnavailableException

public class **ServiceUnavailableException** extends
org.osgi.framework.ServiceException

This exception is thrown to indicate that the service is unavailable. If an `OSGiService` service reference is marked as dynamic, an attempt is made to get a reference to the service in the OSGi Service Registry when the service is used, and then the method is invoked on the newly obtained service. If the service cannot be discovered or a reference obtained, the `ServiceUnavailableException` is thrown.

15.2 Appendix B – Setting up GlassFish Server to use OSGi/Java EE features (Only applicable for GlassFish 3.1)

All the features described in this document are available in both web profile as well as full profile distribution of GlassFish Server. The features are automatically available in 3.1.1 or trunk builds of GlassFish, but if you are using GlassFish 3.1 FCS build, they have to be explicitly enabled by adding **org.apache.felix.shell.remote** and **org.apache.felix.fileinstall** to the list of bundles mentioned in system property called **org.glassfish.additionalOSGiBundlesToStart** in domain.xml. There are a few ways to do this:

a) Using asadmin command:

```
asadmin delete-jvm-options --target server-config \  
-Dorg.glassfish.additionalOSGiBundlesToStart=\  
"org.apache.felix.shell,\  
org.apache.felix.gogo.runtime,\  
org.apache.felix.gogo.shell,\  
org.apache.felix.gogo.command"
```

```
asadmin create-jvm-options --target server-config \  
-Dorg.glassfish.additionalOSGiBundlesToStart=\  
"org.apache.felix.shell,\  
org.apache.felix.gogo.runtime,\  
org.apache.felix.gogo.shell,\  
org.apache.felix.shell.remote,\  
org.apache.felix.fileinstall"
```

Repeat the above commands by replacing server-config with default-config.

b) Using Admin Console:

You can also use GlassFish Admin Console to update the JVM Properties.

c) You can also make the changes in domain.xml by directly editing it. Look for occurrences of -Dorg.glassfish.additionalOSGiBundlesToStart .

The above steps are not required if you are using 3.1.1 or trunk build.

15.3 Appendix C - Samples

A number of sample applications are available in our workspace. You can checkout and build as shown below:

```
svn co http://svn.java.net/svn/glassfish~svn/trunk/fightertfish/sample
```

```
mvn install
```

Use JDK 6 and Maven 2.2.1 while running the above commands.

Now you can deploy each of the sample to GlassFish Server and explore it by yourself. You can also browse the sample workspace by visiting <http://java.net/projects/glassfish/sources/svn/show/trunk/fightertfish/sample>. The table below gives a short summary of the available samples as of now and the key features they demonstrate:

Sample Name	Short Summary	Highlight
sample.parent-pom	Parent POM – Extended by all other modules. Provides a framework to make development as simple as possible.	Development/Build Framework
sample.uas.api	A Simple OSGi bundle shows how to export/import package	Getting Familiar with OSGi Bundles
sample.uas.simpleservice	A Simple OSGi Service Implementation	Getting Familiar with OSGi Services
sample.uas.entities	An OSGi bundle containing JPA model classes. Its activator publishes an EMF Service.	Shared/Standalone Persistence Unit. EntityManagerFactory as an OSGi Service
sample.uas.ejbservice	An EJB based implementation of a service – this bundle embeds JPA classes	EJB as a Service Transactional, secured, persistence aware, multi-threaded OSGi service
sample.uas.ejbservice2	An EJB based implementation of a service – this bundle consumes the EMF service – so it does not bundle JPA model classes	Shared/Standalone Persistence Unit @Inject @OSGiService
sample.uas.simplewab	A Simple Web Application	WAB, @Inject @OSGiService

sample.uas.simplewabfragment	OSGi Fragment Bundle as Web Fragment	Modularity of web applications, Web Fragments in WAB
sample.osgijdbc.helloworld	A simple bundle demonstrating how to use JDBC from OSGi	JDBC DataSource as an OSGi Service
sample.osgijta.helloworld	A simple bundle demonstrating how to use JTA from OSGi	UserTransaction and TransactionSynchronisationRegistry as an OSGi Service
sample.osgihttp.helloworld	A simple OSGi bundle demonstrating use of OSGi/HTTP Service from a SCR component (user has to deploy osgi-http.jar available in maven or update centre to run this sample)	OSGi/HTTP Service of GlassFish. Also uses Declarative Service (aka SCR)

15.4 Appendix D - TODO

This is not a list of features to be implemented in the product. This is where we keep track of items to be added in this document.

- Add detailed use case for each feature.

15.5 Appendix E - Revision History

Version	Date	Comment	Author
0.1	3 Feb 2011	Initial Revision	Sanjeeb Sahoo
0.2	11 Feb 2011	Incorporated feedback from Siva	Sanjeeb Sahoo
0.5	Mar 2011	Incorporated feedback from John C, Richard Hall.	Sanjeeb Sahoo
1.0	May 2011	Fixed formatting, added internal and external links, added pictures, more feedback from Siva.	Sanjeeb Sahoo
1.1	July 2011	Added a note about eclipselink target server property	Sanjeeb Sahoo
1.2	30 July 2011	Updated with GlassFish Server 3.1.1	Sanjeeb Sahoo
1.3	28 Sep 2011	Updated with path for osgi config file and new way to enable the shell	Sanjeeb Sahoo

1.4	10 Feb 2012	Updated Bundle- ManifestVersion in fragments	Sanjeeb Sahoo
-----	-------------	--	---------------