128 Web Applications Specification

Version 1.0

128.1 Introduction

The Java EE Servlet model has provided the backbone of web based applications written in Java. Given the popularity of the Servlet model, it is desirable to provide a seamless experience for deploying existing and new web applications to Servlet containers operating on the OSGi service platform. Previously, the Http Service in the catalogue of OSGi compendium services was the only model specified in OSGi to support the Servlet programming model. However, the Http Service, as defined in that specification, is focused on the run time, as well as manual construction of the servlet context, and thus does not actually support the standard Servlet packaging and deployment model based on the Web Application Archive, or WAR format.

This specification defines the Web Application Bundle, which is a bundle that performs the same role as the WAR in Java EE. A WAB uses the OSGi life cycle and class/resource loading rules instead of the standard Java EE environment. WABs are normal bundles and can leverage the full set of features of the OSGi Service Platform.

Web applications can also be installed as traditional WARs through a manifest rewriting process. During the install, a WAR is transformed into a WAB. This specification was based on ideas developed in *PAX Web Extender* on page 429.

This Web Application Specification provides support for web applications written to the Servlet 2.5 specification, or later. Given that Java Server Pages, or JSPs, are an integral part of the Java EE web application framework, this specification also supports the JSP 2.1 specification or greater if present. This specification details how a web application packaged as a WAR may be installed into an OSGi Service Platform, as well as how this application may interact with, and obtain, OSGi services.

128.1.1 Essentials

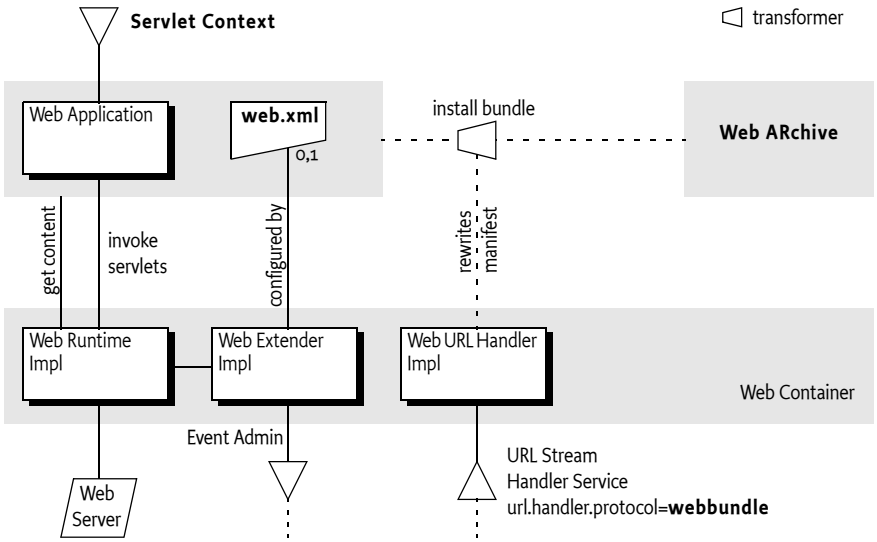
- *Extender* – Enable the configuration of components inside a bundle based on configuration data provided by the bundle developer.
- *Services* – Enable the use of OSGi services within a Web Application.
- *Deployment* – Define a mechanism to deploy Web Applications, both OSGi aware and non OSGi aware, in the OSGi environment.
- *WAR File Support* – Transparently enhance the contents of a WAR's manifest during installation to add any headers necessary to deploy a WAR as an OSGi bundle.

128.1.2 Entities

- *Web Container* – The implementation of this specification. Consists of a Web Extender, a Web URL Handler and a Servlet and Java Server Pages Web Runtime environment.
- *Web Application* – A program that has web accessible content. A Web Application is defined by *Java EE Web Applications on page 429*.
- *Web Application Archive (WAR)* – The Java EE standard resource format layout of a JAR file that contains a deployable Web Application.
- *Web Application Bundle* – A Web Application deployed as an OSGi bundle, also called a WAB.
- *WAB* – The acronym for a Web Application Bundle.
- *Web Extender* – An extender bundle that deploys the Web Application Bundle to the Web Runtime based on the Web Application Bundle's state.

- *Web URL Handler* – A URL handler which transforms a Web Application Archive (WAR) to conform to the OSGi specifications during installation by installing the WAR through a special URL so that it becomes a Web Application Bundle.
- *Web Runtime* – A Java Server Pages and Servlet environment, receiving the web requests and translating them to servlet calls, either from Web Application servlets or other classes.
- *Web Component* – A Servlet or Java Server Page (JSP).
- *Servlet* – An object implementing the Servlet interface; this is for the request handler model in the Servlet Specification.
- *Servlet Context* – The model representing the Web Application in the Servlet Specification.
- *Java Server Page (JSP)* – A declarative, template based model for generating content through Servlets that is optionally supported by the Web Runtime.
- *Context Path* – The URI path prefix of any content accessible in a Web Application.

Figure 128.1 Web Container Entities



128.1.3 Dependencies

The package dependencies for the clients of this specification are listed in Table 128.1.

Table 128.1 Dependency versions

Packages	Export Version	Client Import Range
javax.servlet	2.5	[2.5,3.0)
javax.servlet.http	2.5	[2.5,3.0)
javax.servlet.jsp.el	2.1	[2.1,3.0)
javax.servlet.jsp.jstl.core	1.2	[1.2,2.0)
javax.servlet.jsp.jstl.fmt	1.2	[1.2,2.0)
javax.servlet.jsp.jstl.sql	1.2	[1.2,2.0)
javax.servlet.jsp.jstl.tlv	1.2	[1.2,2.0)
javax.servlet.jsp.resources	2.1	[2.1,3.0)
javax.servlet.jsp.tagext	2.1	[2.1,3.0)
javax.servlet.jsp	2.1	[2.1,3.0)

JSP is optional for the Web Runtime.

128.1.4 Synopsis

The Web Application Specification is composed of a number of cooperating parts, which are implemented by a *Web Container*. A Web Container consists of:

- *Web Extender* – Responsible for deploying Web Application Bundles (WAB) to a Web Runtime,
- *Web Runtime* – Provides support for Servlet and optionally for JSPs, and
- *Web URL Handler* – Provides on-the-fly enhancements of non-OSGi aware Web ARchives (WAR) so that they can be installed as a WAB.

WABs are standard OSGi bundles with additional headers in the manifest that serve as deployment instructions to the Web Extender. WABs can also contain the Java EE defined `web.xml` descriptor in the `WEB-INF/` directory. When the Web Extender detects that a WAB is ready the Web Extender deploys the WAB to the Web Runtime using information contained in the `web.xml` descriptor and the appropriate manifest headers. The Bundle Context of the WAB is made available as a Servlet Context attribute. From that point, the Web Runtime will use the information in the WAB to serve content to any requests. Both dynamic as well as static content can be provided.

The Web URL Handler allows the deployment of an unmodified WAR as a WAB into the OSGi framework. This Web URL Handler provides a URL stream handler with the `webbundle:` scheme. Installing a WAR with this scheme allows the Web URL Handler to interpose itself as a filter on the input stream of the contents of the WAR, transforming the contents of the WAR into a WAB. The Web URL Handler rewrites the manifest by adding necessary headers to turn the WAR into a valid WAB. Additional headers can be added to the manifest that serve as instructions to the Web Extender.

After a WAB has been deployed to the Web Runtime, the Web Application can interact with the OSGi framework via the provided Bundle Context. The Servlet Context associated with this WAB follows the same life cycle as the WAB. That is, when the underlying Web Application Bundle is started, the Web Application is deployed to the Web Runtime. When the underlying Web Application Bundle is stopped because of a failure or other reason, the Web Application is undeployed from the Web Runtime.

128.2 Web Container

A Web Container is the implementation of this specification. It consists of the following parts:

- *Web Extender* – Detects Web Application Bundles (WAB) and tracks their life cycle. Ready WABs are deployed to the Web Runtime.
- *Web Runtime* – A runtime environment for a Web Application that supports the [4] *Servlet 2.5 specification* and [5] *JSP 2.1 specification* or later. The Web Runtime receives web requests and calls the appropriate methods on servlets. Servlets can be implemented by classes or Java Server Pages.
- *Web URL Handler* – A URL stream handler providing the `webbundle:` scheme. This scheme can be used to install WARs in an OSGi Service Platform. The Web URL Handler will then automatically add the required OSGi manifest headers.

The extender, runtime, and handler can all be implemented in the same or different bundles and use unspecified mechanisms to communicate. This specification uses the defined names of the sub-parts as the actor; the term Web Container is the general name for this collection of actors.

128.3 Web Application Bundle

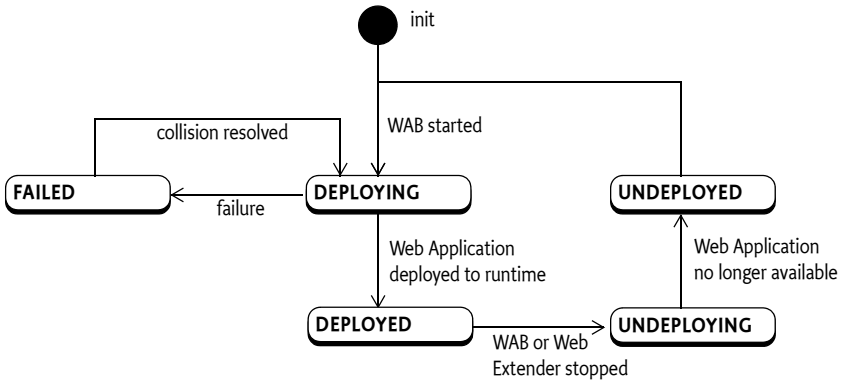
Bundles are the deployment and management entities under OSGi. A *Web Application Bundle* (WAB) is deployed as an OSGi bundle in an OSGi framework, where each WAB provides a single *Web Application.* A Web Application can make use of the [4] *Servlet 2.5 specification* and [5] *JSP 2.1 specification* programming models, or later, to provide content for the web.

A WAB is defined as a normal OSGi bundle that contains web accessible content, both static and dynamic. There are no restrictions on bundles. A Web Application can be packaged as a WAB during application development, or it can be transparently created at bundle install time from a standard Web Application aRchive (WAR) via transformation by the Web URL Handler, see *Web URL Handler* on page 424.

A WAB is a valid OSGi bundle and as such must fully describe its dependencies and exports (if any). As Web Applications are modularized further into multiple bundles (and not deployed as WAR files only) it is possible that a WAB can have dependencies on other bundles.

A WAB may be installed into the framework using the BundleContext.installBundle methods. Once installed, a WAB's life cycle is managed just like any other bundle in the framework. This life cycle is tracked by the Web Extender who will then deploy the Web Application to the Web Runtime when the WAB is ready and will undeploy it when the WAB is no longer ready. This state is depicted in Figure 128.2.

Figure 128.2 State diagram Web Application



128.3.1 WAB Definition

A WAB is differentiated from non Web Application bundles through the specification of the additional manifest header:

```
Web-ContextPath ::= path
```

The Web-ContextPath header specifies the value of the *Context Path* of the Web Application. All web accessible content of the Web Application is available on the web server relative to this Context Path. For example, if the context path is /sales, then the URL would be something like: http://www.acme.com/sales. The Context Path must always begin with a forward slash ('/').

The Web Extender must not recognize a bundle as a Web Application unless the Web-ContextPath header is present in its manifest and the header value is a valid path for the bundle.

A WAB can optionally contain a web.xml resource to specify additional configuration. This web.xml must be found with the Bundle findEntries method at the path:

```
WEB-INF/web.xml
```

The findEntries method includes fragments, allowing the web.xml to be provided by a fragment. The Web Extender must fully support a web.xml descriptor that specifies Servlets, Filters, or Listeners whose classes are required by the WAB.

128.3.2 Starting the Web Application Bundle

A WAB's Web Application must be *deployed* while the WAB is *ready*. Deployed means that the Web Application is available for web requests. Once deployed, a WAB can serve its web content on the given Context Path. Ready is when the WAB:

- Is in the ACTIVE state, or
- Has a lazy activation policy and is in the STARTING state.

The Web Extender should ensure that serving static content from the WAB does not activate the WAB when it has a lazy activation policy.

To deploy the WAB, the Web Extender must initiate the deploying of the Web Application into a Web Runtime. This is outlined in the following steps:

1 Wait for the WAB to become ready. The following steps can take place asynchronously with the starting of the WAB.
2 Post an org/osgi/service/web/DEPLOYING event. See *Events* on page 426.
3 Validate that the Web-ContextPath manifest header does not match the Context Path of any other currently deployed web application. If the Context Path value is already in use by another Web Application, then the Web Application must not be deployed, and the deployment fails, see *Failure* on page 421. The Web Extender should log the collision. If the prior Web Application with the same Context Path is undeployed later, this Web Application should be considered as a candidate, see *Stopping the Web Application Bundle* on page 423.
4 The Web Runtime processes deployment information by processing the web.xml descriptor, if present. The Web Container must perform the necessary initialization of Web Components in the WAB as described in the [4] *Servlet 2.5 specification*. This involves the following sub-steps in the given order:
   - Create a Servlet Context for the Web Application.
   - Instantiate configured Servlet event listeners.
   - Instantiate configured application filter instances etc.

   The Web Runtime is required to complete instantiation of listeners prior to the start of execution of the first request into the Web Application by the Web Runtime. Attribute changes to the Servlet Context and Http Session objects can occur concurrently. The Servlet Container is not required to synchronize the resulting notifications to attribute listener classes. Listener classes that maintain state are responsible for the integrity of the data and should handle this case explicitly.
   If event listeners or filters are used in the web.xml, then the Web Runtime will load the corresponding classes from the bundle activating the bundle if it was lazily started. Such a configuration will therefore not act lazily.
5 Publish the Servlet Context as a service with identifying service properties, see *Publishing the Servlet Context* on page 421.
6 Post an org/osgi/service/web/DEPLOYED event to indicate that the web application is now available. See *Events* on page 426.

If at any moment before the org/osgi/service/web/DEPLOYED event is published the deployment of the WAB fails, then the WAB deployment fails, see *Failure* on page 421.

128.3.3 Failure

Any validation failures must prevent the Web Application from being accessible via HTTP, and must result in a org/osgi/service/web/FAILED event being posted. See *Events* on page 426. The situation after the failure must be as if the WAB was never deployed.

128.3.4 Publishing the Servlet Context

To help management agents with tracking the state of Web Applications, the Web Extender must register the Servlet Context of the WAB as a service, using the Bundle Context of the WAB. The Servlet Context service must be registered with the service properties listed in Table 128.2.

Table 128.2 Servlet Context Service Properties

Property Name	Type	Description
osgi.web.symbolicname	String	The symbolic name for the Web Application Bundle
osgi.web.version	String	The version of the Web Application Bundle. If no Bundle-Version is specified in the manifest then this property must not be set.
osgi.web.contextpath	String	The Context Path from which the WAB's content will be served.

128.3.5 Static Content

A deployed WAB provides content on requests from the web. For certain access paths, this can serve content from the resources of the web application: this is called *static content.* A Web Runtime must use the Servlet Context resource access methods to service static content, the resource loading strategy for these methods is based on the findEntries method, see *Resource Lookup* on page 428. For confidentiality reasons, a Web Runtime must not return any static content for paths that start with one of the following prefixes:

```
WEB-INF/
OSGI-INF/
META-INF/
OSGI-OPT/
```

These *protected directories* are intended to shield code content used for dynamic content generation from accidentally being served over the web, which is a potential attack route. In the servlet specification, the WEB-INF/ directory in the WAR is protected in such a way. However, this protection is not complete. A dependent JAR can actually be placed outside the WEB-INF directory that can then be served as static content. The same is true for a WAB. Though the protected directories must never be served over the web, there are no other checks required to verify that no content can be served that is also available from the Bundle class path.

It is the responsibility of the author of the WAB to ensure that confidential information remains confidential by placing it in one of the protected directories. WAB bundles should be constructed in such a way that they do not accidentally expose code or confidential information. The simplest way to achieve this is to follow the WAR model where code is placed in the WEB-INF/classes directory and this directory is placed on the Bundle's class path as the first entry. For example:

```
Bundle-ClassPath: WEB-INF/classes, WEB-INF/lib/acme.jar
```

128.3.6 Dynamic Content

Dynamic content is content that uses code to generate the content, for example a servlet. This code must be loaded from the bundle with the Bundle loadClass method, following all the Bundle class path rules.

Unlike a WAR, a WAB is not constrained to package classes and code resources in the WEB-INF/classes/ directory or dependent JARs in WEB-INF/lib/ only. These entries can be packaged in any way that's valid for an OSGi bundle as long as such directories and JARs are part of bundle class path as set with the Bundle-ClassPath header and any attached fragments. JARs that are specified in the Bundle-ClassPath header are treated like JARs in the WEB-INF/lib/ directory of the Servlet specification. Similarly, any directory that is part of the Bundle-ClassPath header is treated like WEB-INF/classes/ directory of the Servlet specification.

Like WARs, code content that is placed outside the protected directories can be served up to clients as static content.

128.3.7 Content Serving Example

This example consists of a WAB with the following contents:

```
acme.jar:
   Bundle-ClassPath: WEB-INF/classes, LIB/bar.jar
   Web-ContextPath: /acme

   WEB-INF/lib/foo.jar
   LIB/bar.jar
   index.html
   favicon.ico
```

The content of the embedded JARs foo.jar and bar.jar is:

```
foo.jar:                    bar.jar:
   META-INF/foo.tld            META-INF/bar.tld
   foo/FooTag.class            bar/BarTag.class
```

Assuming there are no special rules in place then the following lists specifies the result of a number of web requests for static content:

```
/acme/index.html           acme.wab:index.html
/acme/favicon.ico          acme.wab:favicon.ico
/acme/WEB-INF/lib/foo.jar  not found because protected directory
/acme/LIB/bar.jar          acme.wab:LIB/bar.jar  (code, but not not protected)
```

In this example, the tag classes in bar.jar must be found (if JSP is supported) but the tag classes in foo.jar must not because foo.jar is not part of the bundle class path.

128.3.8 Stopping the Web Application Bundle

A web application is stopped by stopping the corresponding WAB. In response to a WAB STOPPING event, the Web Extender must *undeploy* the corresponding Web Application from the Servlet Container and clean up any resources. This undeploying must occur synchronously with the WAB's stopping event. This will involve the following steps:

1 An org/osgi/service/web/UNDEPLOYING event is posted to signal that a Web Application will be removed. See *Events* on page 426.
2 Unregister the corresponding Servlet Context service
3 The Web Runtime must stop serving content from the Web Application.
4 The Web Runtime must clean up any Web Application specific resources as per servlet 2.5 specification.
5 Emit an org/osgi/service/web/UNDEPLOYED event. See *Events* on page 426.
6 It is possible that there are one or more *colliding* WABs because they had the same Context Path as this stopped WAB. If such colliding WABs exists then the Web Extender must attempt to deploy the colliding WAB with the lowest bundle id.

Any failure during undeploying should be logged but must not stop the cleaning up of resources and notification of (other) listeners as well as handling any collisions.

128.3.9 Uninstalling the Web Application Bundle

A web application can be uninstalled by uninstalling the corresponding WAB. The WAB will be uninstalled from the OSGi framework.

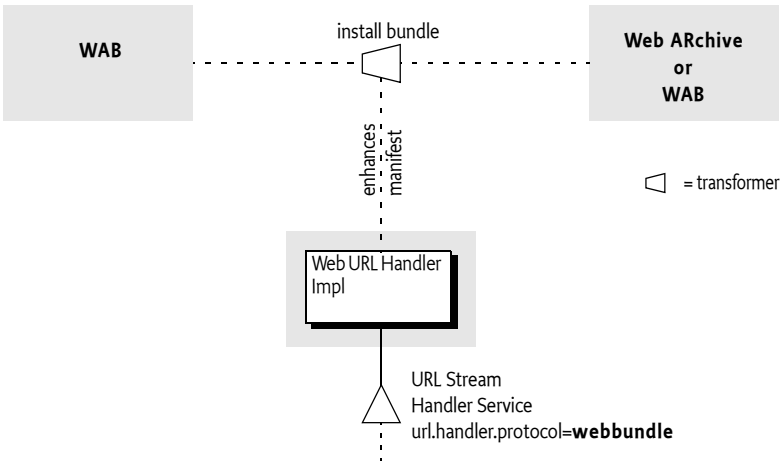
128.3.10 Stopping of the Web Extender

When the Web Extender is stopped all deployed WABs are undeployed as described in *Stopping the Web Application Bundle* on page 423. Although the WAB is undeployed it remains in the ACTIVE state. When the Web Extender leaves the STOPPING state all WABs will have been undeployed.

128.4 Web URL Handler

The Web URL Handler acts as a filter on the Input Stream of an install operation. It receives the WAB or WAR and it then generates a JAR that conforms to the WAB specification by rewriting the manifest resource. This process is depicted in Figure 128.3 *Web URL Handler*.

Figure 128.3 Web URL Handler



When the Web Container bundle is installed it must provide the `webbundle:` scheme to the URL class. The Web URL Handler has two primary responsibilities:

- *WAB* – If the source is already a bundle then only the Web-ContextPath can be set or overwritten.
- *WAR* – If the source is a WAR (that is, it must not contain any OSGi defined headers) then convert the WAR into a WAB.

The Web URL Handler can take parameters from the query arguments of the install URL, see 128.4.3 *URL Parameters*.

The URL handler must validate query parameters, and ensure that the manifest rewriting results in valid OSGi headers. Any validation failures must result in Bundle Exception being thrown and the bundle install must fail.

Once a WAB is generated and installed, its life cycle is managed just like any other bundle in the framework.

128.4.1 URL Scheme

The Web URL Handler's scheme is defined to be:

```
scheme     ::= 'webbundle:" embedded '?' web-params
embedded   ::= <embedded URL according to RFC 1738>
web-params ::= ( web-param ( '&' web-param )* )?
web-param  ::= <key> '=' <value>
```

The `web-param` <key> and <value> as well as the <embedded url> must follow [7] *Uniform Resource Locators, RFC 1738* for their escaping and character set rules.A Web URL must further follow all the rules of a URL. Whitespaces are not allowed between terms.

An example for a `webbundle:` URL:

```
webbundle:http://www.acme.com:8021/sales.war?
```

Any URL scheme understood by the framework can be embedded, such as an http:, or file: URL. Some forms of embedded URL also contain URL query parameters and this must be supported. The embedded URL most be encoded as a standard URL. That is, the control characters like colon (':'), slash ('/'), percent ('%'), and ampersand ('?') must not be encoded. Thus the value returned from the getPath method may contain a query part. Any implementation must take care to preserve both the query parameters for the embedded URL, and for the complete webbundle: URL. A question mark must always follow the embedded URL to simplify this processing. The following example shows an HTTP URL with some query parameters:

```
webbundle:http://www.acme.com/sales?id=123?Bundle-SymoblicName=com.example
```

128.4.2 URL Parsing

The URL object for a webbundle: URL must return the following values for the given methods:

- getProtocol – webbundle
- getPath – The complete embedded URL
- getQuery – The parameters for processing of the manifest.

For the following example:

```
webbundle:http://acme.com/repo?war=example.war?Bundle-SymoblicName=com.example
```

The aforementioned methods must return:

- getProtocol – webbundle
- getPath – http://acme.com/repo?war=example.war
- getQuery – Bundle-SymoblicName=com.example

128.4.3 URL Parameters

All the parameters in the webbundle: URL are optional except for the Web-ContextPath parameter. The parameter names are case insensitive, but their values must be treated as case sensitive. Table 128.3 describes the parameters that must be supported by any webbundle: URL Stream handler. A Web URL Handler is allowed to support additional parameters.

Table 128.3 Web bundle URL Parameters

Parameter Name	Description
Bundle-SymbolicName	The desired symbolic name for the resulting WAB.
Bundle-Version	The version of the resulting WAB. The value of this parameter must follow the OSGi versioning syntax.
Bundle-ManifestVersion	The desired bundle manifest version. Currently, the only valid value for this parameter is 2.
Import-Package	A list of packages that the war file depends on.
Web-ContextPath	The Context Path from which the Servlet Container should serve content from the resulting WAB. This is the only valid parameter when the input JAR is already a bundle. This parameter must be specified.

128.4.4 WAB Modification

The Web URL Handler can set or modify the Web-ContextPath of a WAB if the input source is already a bundle. It must be considered as a bundle when any of the OSGi defined headers listed in Table 128.3 is present in the bundle.

For WAB Modification, the Web URL Handler must only support the Web-ContextPath parameter and it must not modify any existing headers other than the Web-ContextPath. Any other parameter given must result in a Bundle Exception.

128.4.5 WAR Manifest Processing

The Web URL Handler is designed to support the transparent deployment of Java EE Web ARchives (WAR). Such WARs are ignorant of the requirements of the underlying OSGi service platform that hosts the Web Runtime. These WARs are not proper OSGi bundles because they do not contain the necessary metadata in the manifest. For example, a WAR without a Bundle-ManifestVersion, Import-Package, and other headers cannot operate in an OSGi service platform.

The Web URL Handler implementation copies the contents of the embedded URL to the output and rewrites the manifest headers based on the given parameters. The result must be a WAB.

Any parameters specified must be treated as manifest headers for the web. The following manifest headers must be set to the following values if not specified:

- Bundle-ManifestVersion – Must be set to 2.
- Bundle-SymbolicName – Generated in an implementation specific way.
- Bundle-ClassPath – Must consist of:
  - WEB-INF/classes/
  - All JARs from the WEB-INF/lib directory in the WAR. The order of these embedded JARs is unspecified.
  - If these JARs declare dependencies in their manifest on other JARs in the bundle, then these jars must also be appended to the Bundle-ClassPath header. The process of detecting JAR dependencies must be performed recursively as indicated in the Servlet Specification.
- Web-ContextPath – The Web-ContextPath must be specified as a parameter. This Context Path should start with a leading slash ('/'). The Web URL handler must add the preceding slash it if it is not present.

The Web URL Handler is responsible for managing the import dependencies of the WAR. Implementations are free to handle the import dependencies in an implementation defined way. They can augment the Import-Package header with byte-code analysis information, add a fixed set of clauses, and/or use the Dynamic-ImportPackage header as last resort.

Any other manifest headers defined as a parameter or WAR manifest header not described in this section must be copied to the WAB manifest by the Web URL Handler. Such an header must not be modified.

128.4.6 Signed WAR files

When a signed WAR file is installed using the Web URL Handler, then the manifest rewriting process invalidates the signatures in the bundle. The OSGi specification requires fully signed bundles for security reasons, security resources in partially signed bundles are ignored.

If the use of the signing metadata is required, the WAR must be converted to a WAB during development and then signed. In this case, the Web URL Handler cannot be used. If the Web URL Handler is presented with a signed WAR, the manifest name sections that contain the resource's check sums must be stripped out by the URL stream handler. Any signer files (*.SF and their corresponding DSA/RSA signature files) must also be removed.

128.5 Events

The Web Extender must track all WABs in the OSGi service platform in which the Web Extender is installed. The Web Extender must post Event Admin events, which is asynchronous, at crucial points in its processing. The topic of the event must be one of the following values:

- org/osgi/service/web/DEPLOYING – The Web Extender has accepted a WAB and started the process of deploying a Web Application.
- org/osgi/service/web/DEPLOYED – The Web Extender has finished deploying a Web Application, and the Web Application is now available for web requests on its Context Path.

- org/osgi/service/web/UNDEPLOYING – The web extender started undeploying the Web Application in response to its corresponding WAB being stopped or the Web Extender is stopped.
- org/osgi/service/web/UNDEPLOYED – The Web Extender has undeployed the Web Application. The application is no longer available for web requests.
- org/osgi/service/web/FAILED – The Web Extender has failed to deploy the Web Application, this event can be fired after the DEPLOYING event has fired and indicates that no DEPLOYED event will be fired.

For each event topic above, the following properties must be published:

- bundle.symbolicName – (String) The bundle symbolic name of the WAB.
- bundle.id – (Long) The bundle id of the WAB.
- bundle – (Bundle) The Bundle object of the WAB.
- bundle.version – (Version) The version of the WAB.
- context.path – (String) The Context Path of the Web Application.
- timestamp – (Long) The time when the event occurred
- extender.bundle – (Bundle) The Bundle object of the Web Extender Bundle
- extender.bundle.id – (Long) The id of the Web Extender Bundle.
- extender.bundle.symbolicName – (String) The symbolic name of the Web Extender Bundle.
- extender.bundle.version – (Version) The version of the Web Extender Bundle.

In addition, the org/osgi/service/web/FAILED event must also have the following property:

- exception – (Throwable) If an exception caused the failure, an exception detailing the error that occurred during the deployment of the WAB.
- collision – (String) If a name collision occurred, the Web-ContextPath that had a collision
- collision.bundles – (Long) If a name collision occurred, a list of bundle ids that all have the same value for the Web-ContextPath manifest header.

128.6 Interacting with the OSGi Environment

128.6.1 Bundle Context Access

In order to properly integrate in an OSGi environment, a Web Application can access the OSGi service registry for publishing its services, accessing services provided by other bundles, and listening to bundle and service events to track the life cycle of these artifacts. This requires access to the Bundle Context of the WAB.

The Web Extender must make the Bundle Context of the corresponding WAB available to the Web Application via the Servlet Context osgi-bundlecontext attribute. A Servlet can obtain a Bundle Context as follows:

```
BundleContext ctxt = (BundleContext)
   servletContext.getAttribute("osgi-bundlecontext");
```

128.6.2 Other Component Models

Web Applications sometimes need to inter-operate with services provided by other component models, such as a Declarative Services or Blueprint. Per the Servlet specification, the Servlet Container owns the life cycle of a Servlet; the life cycle of the Servlet must be subordinate to the life cycle of the Servlet Context, which is only dependent on the life cycle of the WAB. Interactions between different bundles are facilitated by the OSGi service registry. This interaction can be managed in several ways:

- A Servlet can obtain a Bundle Context from the Servlet Context for performing service registry operations.
- Via the JNDI Specification and the osgi:service JNDI namespace. The OSGi JNDI specification describes how OSGi services can be made available via the JNDI URL Context. It defines an

osgi:service namespace and leverages URL Context factories to facilitate JNDI integration with the OSGi service registry.

Per this specification, it is not possible to make the Servlet life cycle dependent on the availability of specific services. Any synchronization and service dependency management must therefore be done by the Web Application itself.

128.6.3 Resource Lookup

The getResource and getResourceAsStream methods of the ServletContext interface are used to access resources in the web application. For a WAB, these resources must be found according to the findEntries method, this method includes fragments. For the getResource and getResourceAsStream method, if multiple resources are found, then the first one must be used.

The getResourcePaths method must map to the Bundle getEntryPaths method, its return type is a Set and can not handle multiples. However, the paths from the getEntryPaths method are relative while the methods of the getResourcePaths must be absolute.

For example, assume the following manifest for a bundle:

```
Bundle-ClassPath: localized, WEB-INF
...
```

This WAB has an attached fragment acme-de.jar with the following content:

```
META-INF/MANIFEST.MF
localized/logo.png
```

The getResource method for localized/logo.png uses the findEntries method to find a resource in the directory /localized and the resource logo.png. Assuming the host bundle has no localized/ directory, the Web Runtime must serve the logo.png resource from the acme-de.jar.

128.6.4 Resource Injection and Annotations

The Web Application web.xml descriptor can specify the metadata-complete attribute on the web-app element. This attribute defines whether the web.xml descriptor is *complete*, or whether the classes in the bundle should be examined for deployment annotations. If the metadata-completeattribute is set to true, the Web Runtime must ignore any servlet annotations present in the class files of the Web Application. Otherwise, if the metadata-complete attribute is not specified, or is set to false, the container should process the class files of the Web Application for annotations, if supported.

A WAB can make use of the annotations defined by [8] *JSR 250 Common Annotations for the Java Platform* if supported by the Web Extender. Such a WAB must import the packages the annotations are contained in. A Web Extender that does not support the use of JSR 250 annotations must not process a WAB that imports the annotations package.

128.6.5 JavaServer Pages Support

Java Server Pages (JSP) is a rendering technology for template based web page construction. This specification supports [5] *JSP 2.1 specification* if available with the Web Runtime. The servlet element in a web.xml descriptor is used to describe both types of Web Components. JSP components are defined implicitly in the web.xml descriptor through the use of an implicit .jsp extension mapping, or explicitly through the use of a jsp-group element.

128.6.6 Compilation

A Web Runtime compiles a JSP page into a Servlet, either during the deployment phase, or at the time of request processing, and dispatches the request to an instance of such a dynamically created class. Often times, the compilation task is delegated to a separate JSP compiler that will be responsible for identifying the necessary tag libraries, and generating the corresponding Servlet. The container then proceeds to load the dynamically generated class, creates an instance and dispatches the servlet request to that instance.

Supporting in-line compilation of a JSP inside a bundle will require that the Web Runtime maintains a private area where it can store such compiled classes. The Web Runtime can leverage its private bundle storage area. The Web Runtime can construct a special class loader to load generated JSP classes such that classes from the bundle class path are visible to newly compiled JSP classes.

The JSP specification does not describe how JSP pages are dynamically compiled or reloaded. Various Web Runtime implementations handle the aspects in proprietary ways. This specification does not bring forward any explicit requirements for supporting dynamic aspects of JSP pages.

128.7 Security

The security aspects of this specification are defined by the *Servlet 2.5 specification* on page 429.

128.8 References

[1] *OSGi Core Specifications*
http://www.osgi.org/Specifications/HomePage

[2] *Jave Enterprise Edition Release 5*
http://java.sun.com/javaee/technologies/javaee5.jsp
Java 1.5.0 Packages

[3] *Java EE Web Applications*
http://java.sun.com/javaee/technologies/webapps/

[4] *Servlet 2.5 specification*
http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index.html

[5] *JSP 2.1 specification*
http://jcp.org/aboutJava/communityprocess/final/jsr245/index.html

[6] *PAX Web Extender*
http://wiki.ops4j.org/display/ops4j/Pax+Web+Extender

[7] *Uniform Resource Locators, RFC 1738*
http://www.ietf.org/rfc/rfc1738.txt

[8] *JSR 250 Common Annotations for the Java Platform*
http://jcp.org/aboutJava/communityprocess/pfd/jsr250/index.html