MODULAR JAVA WEB APPLICATIONS

By

Simon Richard Kaegi

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of

the requirements for the degree of

MASTER OF COMPUTER SCIENCE

Ottawa-Carleton Institute for Computer Science
School of Computer Science

Carleton University
Ottawa, Ontario

August, 2007

Your file Votre référence
ISBN: 978-0-494-33680-9
Our file Notre référence
ISBN: 978-0-494-33680-9

Abstract

The Java Enterprise Edition (Java EE) has provided the industry with a standard suite of APIs and services for developing server-side applications in Java. Over several releases the Java EE standard has added many new APIs however has maintained the status quo in terms of packaging and modularity support. As Java EE applications increase in size and complexity the constraints imposed by the existing component model restrict utility.

In this thesis we examine problems related to building modular and evolvable server-side applications in Java. We use Eclipse's OSGi runtime as a basis for solving these problems and demonstrate integration in a Java EE Application Server environment. We show how this approach provides benefits in terms of support for functional decoupling of components and allows for improved extensibility and deployment when compared with the typical approach provided by the Java EE standard.

Acknowledgements

I'd like to thank my advisor, Dwight Deugo, for introducing me to Eclipse and coaching me through the research and writing processes.

The Equinox OSGi community at Eclipse provided an abundance of useful feedback and support for this work. In particular I'd like to thank Jeff McAffer, Pascal Rapicault, and Tom Watson for helping work through some of the tougher integration problems I encountered.

Finally to my family, thank-you for providing the time and support needed, especially in the final months of writing.

Table of Contents

List of Figures

Chapter 1: Introduction

Although Java had its earliest success on the client, and especially the browser with Java Applets, its mainstream adoption came a little later when it became widely used for building server applications. Perhaps the earliest server-side Java technology developed was for the web tier, in the form of Java Servlets. The first servlet engine was Sun's Java Web Server (JWS) built in 1995 [10]. JWS was well liked by the then nascent server-side community and its success prompted the creation of a number of other servlet engines and eventually led to standardization and the creation of the first Servlet Specification in 1997.

One of the main reasons servlets were so popular was that they simplified the creation of dynamic HTML web pages. Servlets act on the incoming HTTP requests at a fairly low-level and this makes them suitable for handling logic and state changes though direct HTML page creation is verbose and tedious. Shortly after introducing servlets, Sun added and later standardized JavaServer Pages (JSPs), a more script-like and HTML friendly technology built on-top of Servlets, which greatly simplified page creation.

With a useable web tier in place the servlet container market flourished with further revisions to the Servlet and JSP specifications providing improvements as needed. As Java on the server became more mature, both the size of problems addressed by applications and their subsequent demands for greater functionality (particularly in the middle and data tier) increased.

1.1 Context and Motivation

The Java Enterprise Edition (Java EE) [54] was created to help address the need for multi-tiered architectures on the server. In doing this it collects a number of standards related to messaging, database access, management, and componentization together to produce one verifiably consistent Java middleware platform. In addition to specifying programming APIs, this family of specifications has helped formalize deployment and packaging requirements critical in ensuring a degree of portability between server implementations.

Java EE has provided the necessary consistency to build an ecosystem where application server vendors can create complete platforms for hosting large enterprise-class applications. It's fair to say that Java EE technology has a dominant position in the Java middleware market.

One of the challenges facing the Java server community and vendors is related to the size and complexity of the Enterprise Platform. The Java EE family of specifications now covers some twenty-six technologies, and includes requirements for interoperability with CORBA, SOAP, and RMI stacks. The net result is that a Java Enterprise server has become a substantial and very complex piece of software.

The problems inherent in managing a platform containing hundreds of libraries with potentially conflicting version dependencies are a real challenge. To that end, many application server vendors have already, or are in the process of re-basing their component models on Java modularity technologies, like OSGi [6, 24, 28], that are specifically designed to deal with these sorts of issues.

1.2 Problem

Although these efforts by the server vendors are commendable and should ultimately lead to better product, their work does not address the same problems increasingly faced by their customers when building enterprise applications. The problem remains that Java EE applications are monolithic in terms of their deployment and mechanisms for resolving library interdependencies.

At the root of this problem is an overly simplistic class loader model. Class loaders are central to Java modularity support and, focusing on web applications, the Java Servlet specification provides a single class loader per deployable application archive. In effect a web application is one course-grained component where all contained libraries share the same namespace.

Although this assumption simplifies the interaction model between server and application, it limits the application's ability to use libraries with incompatible version dependencies. What's worse is that with no clear component model, incompatibilities might not be detected until after an application has entered production. The resulting errors can be very difficult to understand and the problem has been labeled "Jar Hell" [62] by the development community. Clearly these are the sorts of issues we'd rather avoid when building server-side applications.

A further consequence of per application deployment is that partial update without complete re-deployment of the application is not possible. This can be a real problem with large applications running where system administrators are trying to minimize downtime. Ideally the delivery of a small patch should not result in a complete re-install of the application.

1.3 Goal

The current Java web server infrastructure does not provide adequate modularity support to allow composition of user applications where:

- Individual components and their dependencies won't interfere with one another except where defined to.
- Augmenting the web content and components can be done at runtime without having to restart or redeploy the application.

In this thesis we propose a new style of web application that makes use of an integrated OSGi framework [40] to meet these requirements. Our goal is to first demonstrate the feasibility of this integration, and then show that by re-basing components in a web application around OSGi constructs, we're able to successfully manage library version incompatibilities and the dynamic updating of individual components.

1.4 Expected Contributions

The expected contributions for this thesis are as follows:

- An approach and implementation to allow the embedding of an OSGi framework inside a Java EE server.
- An implementation of the OSGi Http Service designed to bridge communication from an external servlet container into an embedded OSGi framework where the request can be serviced.
- Integration of a JavaServer Pages engine into an OSGi environment.

1.5 Usage Example

Our approach seems particularly applicable to large enterprise web applications where over time patching, formal version upgrades, or potentially the addition of new functional modules are required. As an example, take a browser based accounting suite that includes a domestic tax calculation module. Tax laws are always changing and it's not unusual for an expanding company to quite suddenly require support for international transactions. There might be useful analytics and forecasting add-on modules made available long after the original product was installed. A later version might provide an improved user interface or support for off-line use. These all require modification of the existing software and potentially could introduce dependency mismatches between the components that make up the application.

Applications using our approach have built-in support to both resolve these dependencies and if necessary can run older versions of the software concurrent with the latest version. For example, if the tax calculation module requires an older version of a math library than the analytics module, we can still run both. If a critical security problem is identified we can deploy a patch quickly at runtime to address the concern. Likewise, we can uninstall the patch if it's found to cause other problems. Enterprise applications are rarely static. For these types of applications our approach provides an alternate more dynamic model.

1.6 Outline

The next three chapters provide the background information for this thesis. In Chapter 2 we look at modularity support in Java, focusing on class loaders and delegation models. Chapter 3 examines the Servlet API and web applications and discusses the programming

and deployment model used. In Chapter 4 we look at how OSGi builds on class loaders to define modular components. OSGi defines an HTTP Service for building servlet-based applications and after introducing it we compare it with those features in classical servlet web applications.

The next section introduces our approach – "OSGi Web Applications". In Chapter 5 we take a deeper look at the problems inherent in our approach and further develop the set of requirements we'll need to address. Chapter 6 provides a technical overview of our approach and provides a detailed look at techniques employed to address the requirements.

Elements of the OSGi web application approach developed in this thesis are being used in commercial products and as part of the Eclipse IDE. In Chapter 7 we evaluate our results relative to our requirements and discuss insights drawn from our experience integrating with real world applications.

In the final chapter we evaluate our success relative to the goals set forth, re-iterate the contributions made, and discuss future work.

Chapter 2: Java Modularity and Class Loaders

A complex system can be managed by dividing it up into smaller pieces and looking at each one separately. When the complexity of one of the elements crosses a certain threshold, that complexity can be isolated by defining a separate abstraction that has a simple interface. The abstraction hides the complexity of the element; the interface indicates how the element interacts with the larger system.
- Design Rules: Volume1. The Power of Modularity [5]

Modularity is the concept of breaking a system up into its constituent modules, where the modules operate independently of one another but still can be combined together along constrained lines to do useful work. As the quote above indicates, the primary motivators for modularizing a system are to reduce internal complexity and more tightly control the interaction between constituents. A software component that is both internally cohesive and loosely coupled to its environment is seen as desirable and modules precisely embody these qualities.

In this chapter we examine the Java language's built-in support for modularity. Section 2.1 introduces the Java ARchive (JAR) file format, the primary unit of deployment used to deliver libraries of code. In Section 2.2 we look at Java packages and the "namespacing" support they provide. Class loaders are central to the modularity support provided by Java and Section 2.3 examines their role.

2.1 Java ARchive (JAR) Files

At the most basic level a JAR file [52] is nothing more than a ZIP compressed file that optionally contains a "META-INF" folder where additional information used by the Java runtime is placed. The JAR format was first used to allow the packaging of Java classes and resources together in a single file to ease deployment concerns. This was important for Java Applets since packaging several classes together prevented round-tripping for each individual class. The format has since been adopted by the Java EE family of specifications for Web ARchives (WAR), Enterprise ARchives (EAR), and Resource Adapter aRchives (RAR).

The "META-INF" folder can contain arbitrary files but for our purposes the most important is the "MANIFEST.MF" file or simply the manifest file. The manifest file typically contains entries that describe the contents of the JAR file.

As an example, here is the content of the manifest file that comes packaged with the Servlet 2.4 API in the Apache Tomcat 5.5 servlet container.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.6.5
Created-By: 1.4.2_11-b06 (Sun Microsystems Inc.)

Name: javax/servlet/
Specification-Title: Java API for Servlets
Specification-Version: 2.4
Specification-Vendor: Sun Microsystems, Inc.
Implementation-Title: javax.servlet
Implementation-Version: 2.4.public_draft
Implementation-Vendor: Apache Software Foundation
```

Figure 2-1: Sample MANIFEST.MF

The manifest format structurally consists of a top "main" section followed by a blank line and then individual "named" sections also separated by a blank line. On each line "name:value" pairs referred to as "attributes" are used to provide information to the running system. In our example, the "main" section provides the manifest version and

some build information, and the single "named" section provides information used to describe the contents of the "javax.servlet" package.

JAR files can be directly loaded and run inside a Java Runtime Environment (JRE) and, if used in this manner, two attributes that provide information useful for running and linking to other libraries come into play: Main-Class and Class-Path [51]. Main-Class is simply the name of the main application class that should be loaded and run. At run-time the JRE will perform its initialization, load the class, and run it by calling the "main" method passing in any arguments used. Class-Path is more interesting in the context of defining a module as it provides a mechanism for a JAR file to express a dependency on another library. This attribute is used to provide a list of relative URLs to other JAR files where classes can be loaded from. Perhaps the single largest weakness of the approach is the brittleness of expressing the dependency as a location instead of the actual constraint; the classes and resources contained in the JAR file. In addition, the Class-Path attribute is only defined when first creating the set of libraries to link with, and is subsequently not amenable to later run-time modification.

The JAR file format and manifest file provide a useful mechanism for assembling collections of Java resources along with related meta-information. As we will show in later chapters it can be extended to more fully support modularity concerns.

2.2 Packages and Namespaces

When you create a class or interface in Java perhaps the first thing you place at the top of your class file is the package declaration. For example

```
package org.eclipse.equinox.http.servletbridge;
```

Packages have an important role in modularity in that they provides Java's namespace support and subsequently help prevent naming conflicts. By convention developers define their packages by using the reverse Internet domain name of their organization. In this way conflicts are avoided except possibly within the developer's own organization where they can be directly resolved.

Packages have a second role to play in terms of information hiding. Packages permit access to non-private fields and methods for classes and interfaces in the same package. This allows the grouping of related functionality that can share an internal API without having to expose those implementation details outside the package.

The access support provided by packages is simple and in many cases sufficient although it can become cumbersome as the number of privileged classes in a package grows. The problem in these situations is that sub-dividing the package will end up exposing internal API as there's currently no support for providing cross-package privileged access.

2.3 Class Loading

In Java, class loaders are the components responsible for the dynamic linking and loading of byte-code class definitions to create runtime Class objects. Liang and Bracha in their paper "Dynamic Class Loading in the Java Virtual Machine" [33] noted four main features of class loaders:

- **Lazy loading** – classes are not loaded until required.
- **Type-safe linkage** – The VM provide type-safety guarantees such that only one class definition is visible to a class loader hierarchy.

- **User-definable class loading policy** – Class loaders are first class objects and definable by the programmer to control all aspects of dynamic loading.
- **Multiple namespaces** – Each distinct type in Java is identified by its class name and the class loader that defined it. Subsequently, two types with same class name can be loaded into a single VM provided their class hierarchies are isolated.

All four features are important for modularity support. The first two features provide for an efficient and safe runtime environment, whereas the latter two provide the necessary flexibility to let us create sophisticated systems of inter-communicating components.

2.3.1 java.lang.ClassLoader

All class loaders are sub-classes of java.lang.ClassLoader but are otherwise regular Java objects. The ClassLoader class provides a series of methods that allow it to find, define, and ultimately load classes and make them available for other classes to link against and use. Classes can be loaded implicitly by the VM when needed or explicitly by using the ClassLoader directly [34]. Regardless of the approach taken it ultimately results in the ClassLoader's loadClass(String className) method being called.

User defined class loaders can be written by the programmer and as a result the implementation details of "loadClass" can vary [23]. Regardless, there are three general steps that take place when this method is called. [8, 37, 48, 61]

1. Determine if the associated class identified by name has previously been loaded by this class loader and if so return it immediately.

2. Using implementation dependent means find the associated class file bytes for the given class name.

3. Call the native defineClass method to have the Java Virtual Machine (JVM or Java VM) interpret the bytes and create a Class object and finally return it.

One particularly important point to understand is the relationship between a Class and its "defining" class loader. The type system in Java is based on uniquely identifying the pair <C, L> consisting of the class name (C) and the class loader (L) that ultimately provided the bytes and called defineClass. As a result, classes with the same name when "defined" in different class loaders are different types. This has important consequences in terms of modularity as it grants a class loader the ability to define its own class namespace or **class-space**.

At the object level each Class object contains a reference to its associated class loader; likewise each ClassLoader instance maintains a cache of classes defined internally to prevent redefinition and subsequent breaking of the type system. This two-way linking has a consequence in how class loaders are garbage collected in that even if unreferenced a class loader can not be reclaimed until all classes it has previously defined are also available for garbage collection.

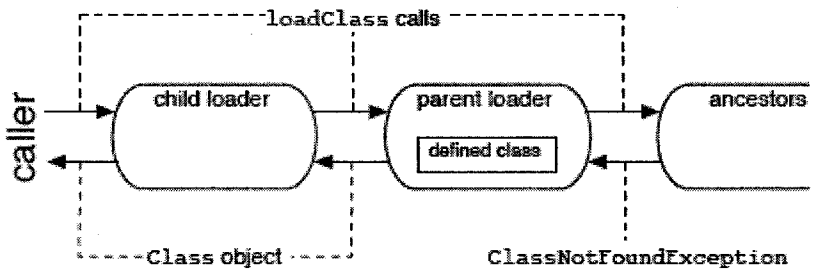
2.3.2 Standard Delegation

Our description of the mechanisms used in class loaders for determining if a class had been previously loaded was overly simplistic as there might be several class loaders involved. Class loaders have a parent class loader that's either explicitly set in their

constructor or by default implicitly set as the System Class Loader (which we'll get to in the next section).

There are many delegation models possible, but in what is referred to as "standard delegation" the parent class loader is always consulted first to see if it can load the class. If the parent class loader fails to load the class the child class loader is given the chance.

The class loader that instigated the class loading process is referred to as the "initiating" class loader and failing any class loader in its delegation chain finding the class it will throw a java.lang.ClassNotFoundException.



[61]

Figure 2-2: Class loader delegation

In addition to class loading delegation the chain of class loaders establishes the class visibility in that the child has a reference and can see the parent's class definitions but not vice-versa. One consequence of this arrangement is that code in a child class loader can sub-class from classes defined in the parent while still permitting the parent to manipulate the class instance through methods defined in the base class. This idiom is common and forms the basis of how Java EE servers interact with installed applications.

2.3.3 System Class Loader

The System Class Loader is the base class loader used to define the Java runtime. In practice it consists of three hierarchically organized class loaders that use standard delegation: the bootstrap class loader, extension class loader, and application class loader.

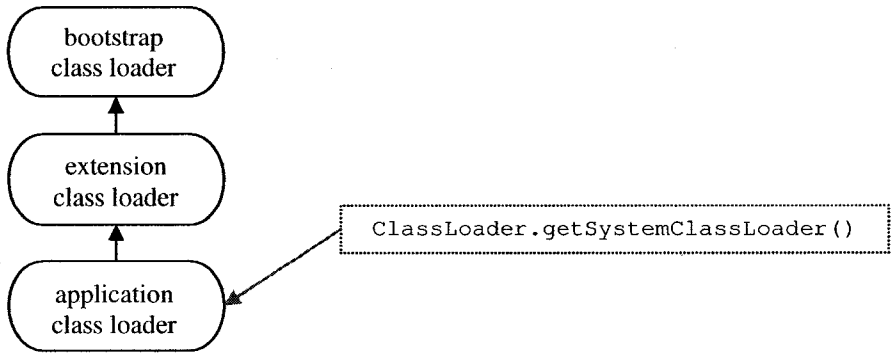


Figure 2-3: System Class Loader

Bootstrap Class Loader. At the root of the hierarchy we have the bootstrap or primordial class loader that holds the definition for the Java platform classes (e.g. java.lang.Object, java.lang.ClassLoader, etc.) as well as any VM specific classes. The boot class loader has the distinction that it is the only class loader able to define classes in the java.* family.

One consequence of not being able to redefine java.* classes is that static variables used in the bootstrap class loader create **VM singletons**. Notable examples of VM singletons that we will refer to later in this thesis include the **ContentHandlerFactory**, **Security Manager**, **System Properties** and **URLStreamHandlerFactory**.

Extension Class Loader. The extension class loader is designed to accept extensions to the platform. Typically this includes functionality like default security and cryptography providers as well as resources to support the localization of content.

Application Class Loader. The application class loader is defined by the Java "classpath" and normally contains the set of jar files (and in some cases directories of class files) required to execute the user's program. This class loader can be retrieved by calling ClassLoader.getSystemClassLoader() and as a result is also referred to as the "System class loader".

2.3.4 Context Class Loader

In a hierarchical parent-child class loader arrangement the child can load classes from the parent. The parent, however, does not have the same visibility into the child. For those situations where the parent wants to instantiate a class from a child or otherwise non-visible class loader, a reference to that class loader is required.

With a reference to the class loader, the parent can use:

- ClassLoader.loadClass("some.class") to fetch the associated Class object.
- Class.newInstance() to create an instance
- Cast the instance to a common mutually visible class or interface.

This idiom is in common use in Java [47] and to help support it all java.lang.Thread's hold a reference to a "context class loader" to allow use-cases where a class loader lower down in a hierarchy wishes to instantiate an otherwise not visible class. The Java Naming and Directory Interface (JNDI) [57] and Java API for XML Processing (JAXP) [53] both make use of the Thread Context Class Loader when instantiating naming contexts and parsers respectively.

It's important to note that the context class loader is not a concrete class loader implementation and is a per Thread mutable reference. To access this class loader java.lang.Thread provides the following public methods:

- `ClassLoader getContextClassLoader()`
- `void setContextClassLoader(ClassLoader cl)`

2.3.5 Hierarchical Delegation

Hierarchical delegation is the most common form of class loader delegation and is characterized by delegating to a single parent class loader. Standard or parent-first delegation is the most common form but is by no means the only possible model. Two other common hierarchical models are: Parent-Last, and Filtered Parent.

Parent-Last. Also called child-first delegation, this is really the opposite of parent-first delegation. The parent class loader is only consulted if the class is unavailable in the immediate class loader. This type of class loader is commonly used in servlet containers where the web application is configured to isolate itself from the implementation details of the server.

Filtered-Parent. This is essentially parent-first where the delegation is altogether filtered based on some criteria, typically the class name. This type of class loader is used in libraries that want to guarantee isolation of particular portions of their internals that are sensitive to version discrepancies.

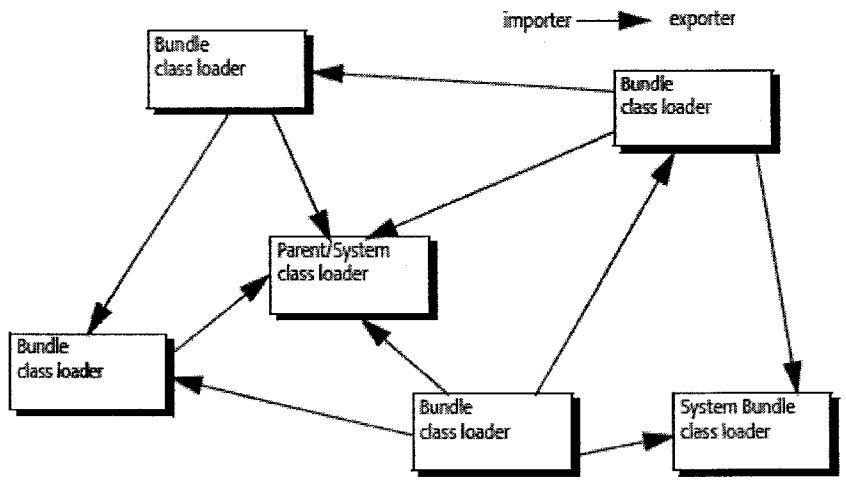
2.3.6 Non-Hierarchical Delegation

What distinguishes non-hierarchical delegation is that in addition to the parent the request to load a class might be delegated to another class loader. This effectively creates

a model with more than one parent. The resulting delegation model is still directed although with multiple parents it is no longer a simple tree and is a more general directed-acyclic graph.

Non-hierarchical class loading schemes allow for greater control of the dependency resolution process and have proven suitable for constructing modular component systems. Perhaps two of the best known non-hierarchical models are OSGi and the JBoss Unified Class Loader.

OSGi [40] is covered in greater detail in a later chapter. In a nutshell it provides a system where specialized JAR files are augmented with metadata to allow an implementation to resolve the dependencies and contributions between various components. The relationships between class loaders form a web by resolving import and export wires between these specialized JAR files or bundles.

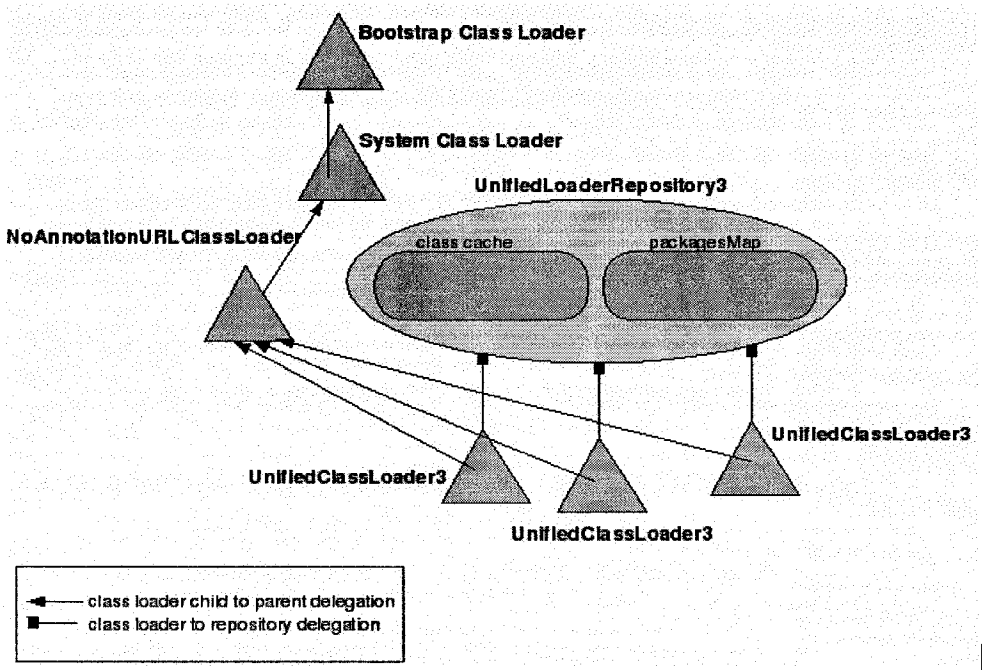


[40]

Figure 2-4: OSGi Class Loader Delegation

The **JBoss Unified Class Loader** [16] allows the building of class repositories such that components can be added and removed at run-time. The Unified Class Loader provides a flat namespace such that components in a particular repository have full

visibility of other participants in that repository. The stated benefit of this scheme is that it facilitates the sharing of common classes and allows for a more dynamic system where components can come and go.



[16]

Figure 2-5: JBoss Unified Class Loader

A discussion of non-hierarchical class loaders should mention issues with potential deadlock in cross dependency situations. In many VM implementations, notably Sun's, the class loader's mutex is locked by the VM while its loadClass method is called.

In some cases multiple class loaders are involved in the lookup and eventual loading of a Class. In this situation the complete chain of class loaders will become locked. In a classic hierarchical scheme this does not lead to deadlock as the class loaders are locked in a fixed hierarchical order. With non-hierarchical class loaders the ordering is not guaranteed and there exists the possibility of deadlock when two class loaders consult each other in their delegation models but have been locked by different threads prior to resolving the loadClass call.

Early implementations of the Unified Class Loader scheme were particularly vulnerable to the deadlock and JBoss identified this bug in the Sun VM. It now sits as the #1 bug on Sun's bug tracker list.

2.4 Summary

This chapter started by looking at the JAR file and the approach taken for packaging Java classes together. This format works well and seems amenable to future extension. We next looked at the packages declarations that provide namespaces and simple albeit limited support for internal API. Finally the chapter concluded with a look at class loaders and how the various delegation models can be used to support module isolation.

In the next chapter we look at Java web applications and the Servlet API and consider the application isolation support offered.

Chapter 3: Java Web Applications and the Servlet API

"Servlets were originally conceived of by James Gosling in 1995 but put aside for other interests. ... Incidentally, JSP was invented over a weekend by Anselm Baird-Smith ..."
- Jim Driscoll [10]

Servlet and JavaServer Pages (JSPs) are two of the most significant and successful technologies for developing server-side web applications in Java. These quotes from Jim Driscoll, one of the earlies servlet programmers, show the nonchalant manner in which Servlet and JavaServer Pages were developed in the early days of Java. More than ten years later both technologies have undergone significant change, yet the fundamental concepts are still present and remarkably the original API is still intact.

In this chapter we introduce Java web applications and examine the programming model, deployment, and support for modularity. Section 3.1 introduces the Servlet API and provides a basic vocabulary for the rest of the chapter. Section 3.2 presents JavaServer Pages (JSP) and Tag Libraries, a complimentary and standardized approach for creating web pages. Servlets and JSPs are deployed together in Web ARchive files. Section 3.3 discusses this format and other elements of the web application's deployment descriptor. Web applications are ultimately deployed into servlet containers and in Section 3.4 we examine the modularity qualities of this arrangement.

3.1 Servlet API Basics

The Servlet API [60] provides developers with a set of classes and interfaces that serve as the fundamental building blocks used for building Java based web components. A few of the most important classes and interfaces are:

- **Servlet.** Users create sub-classes of the Servlet class and use its methods to handle web requests. It would be unusual to directly implement the Servlet interface and in most cases users will extend the abstract **HttpServlet** class which simplifies interaction with HTTP methods.
- **ServletRequest.** The ServletRequest contains the incoming path and parameters as well the protocol headers and methods to access the message payload depending on the HTTP method.
- **ServletResponse.** The ServletResponse contain methods to set outgoing headers and write the message content.
- **ServletConfig.** The ServletConfig holds configuration parameters needed by the Servlet during initialization. In addition, the ServletConfig provides access to the ServletContext.
- **ServletContext.** The ServletContext provides methods to communicate with the servlet container as well as accessing information about the containing web application's configuration.

Of these classes the developer will only create custom Servlets. The lifecycle sequence that the servlet container uses to tie the user's servlet to these other classes is as follows:

1. The servlet container will create an instance of the user defined servlet. This single instance will be used for handling all incoming requests that match a defined URL pattern.

2. The servlet's **init()** method is called passing in a ServletConfig. The servlet can initialize resources like data connections or other resources used to render pages. The init() method will be called exactly once during the life of the servlet. If and only if the servlet has been successfully initialized does it becomes available to handle incoming requests.

3. When a matching incoming request arrives, the servlet container will call the servlet's **service()** method passing in a newly generated ServletRequest and ServletResponse object. The service() method can be called in parallel on separate threads once for each incoming request and subsequently it is critical that this method is thread-safe.

4. At some point the servlet container will decide that the servlet should be stopped. This may be because the container is shutting down or the web application is being uninstalled. The servlet container will first prevent the servlet from receiving new requests and then wait for all current service() requests to complete. Finally it will call the servlet's **destroy()** method to allow for the cleanup of any resources created during the servlet's lifetime.

The discussion so far has just covered the basic elements of the Servlet API. Three more elements of the API that deserve mentioning are: the RequestDispatcher, Filter, HttpSession and Listener.

- **RequestDispatcher**. The RequestDispatcher supports the use of a common application style where a servlet acting as a front controller performs initial logic processing and then forwarding the request on to a second servlet that constructs the page. The RequestDispatcher acts as the decoupling agent in support of a servlet specialized Model-View-Controller setup.

- **Filter.** A Filter allows the transformation and partial servicing of incoming servlet requests and outgoing requests. Filters are implemented by the developer and are frequently used to work transparently with servlets to support authentication and compression of outgoing responses.

- **HttpSession.** The HttpSession provides a mechanism that allows the servlet container to identify a set of request as belonging to a single user session. Typically the association is done by sending a special session cookie. On the server the HttpSession provides a way to store the associated state. Since state is stored on the server use of HttpSession can get complicated when multiple servers are involved. As a result of this usage pattern many Java EE servers provide built-in support for HttpSession clustering.

- **Listener.** Listener actually covers a broad set of classes however what they have in common is that they allow a web application to react to the various lifecycle events around the request, servlet, session, and web application. Custom listeners are sometimes written to handle logging and when special initialization and cleanup is required that must occur at times not well covered by the existing servlet lifecycle.

3.2 JavaServer Pages and Tag Libraries

JavaServer Pages (JSP) [58] is a complimentary technology to Servlets that provide a very low barrier to entry for creating dynamic HTML pages. The process of writing a JSP is very similar to creating a web page but adds the capability to insert small pieces of dynamic content written in Java called **scriptlets**.

```
<html>
<head><title>Hello World</title></head>
<body>
Hello World - <%=new java.util.Date()%>
</body>
</html>
```

Figure 3-1: HelloWorld JSP Example

This example simply prints Hello World and the current date and time. The characters "<%" and "%>" are special markers that indicate to a JSP compiler where special Java processing should be done. To indicate to the servlet container that a particular file is a JSP it is usually just a matter of ensuring the resource filename ends in ".jsp". At runtime the JSP compiler takes the .jsp file and compiles it into a Servlet, so a JSP is really just a simpler way of creating servlet based web pages.

In our example we did not interact with the actual request and other typical elements of the Servlet API. JSPs can access the same objects a regular Servlet might through implicitly declared variables. For example, the "request" variable is set to the ServletRequest and "response" is set to the "ServletResponse". This makes it very convenient for handling simple interactions however there are limits to how far scriptlets should be taken as JSPs do not provide the same code level abstractions as Java classes.

3.2.1 Tag Libraries

With scriptlets you can certainly call out to Java classes to reduce the complexity of your JSP however the now generally accepted approach is to use tag libraries. Tag libraries allow the definition of special XML elements that will be processed by the JSP compiler. The tag libraries themselves are written in Java and deployed alongside the JSP in the same web application. Updating the previous example to replace the use of scriptlets with tag libraries we get the following:

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<jsp:useBean id="date" class="java.util.Date" />
<html>
<head><title>Hello World</title></head>
<body>
Hello World - <c:out value="${date}" />
</body>
</html>
```

Figure 3-2: Hello world JSP Example using Tag Libraries

The "@ taglib" directive is used to declare the use of a tag library in a JSP. In this case we're declaring that the xml elements prefixed with "c" should use a popular set of tag libraries called the Java Standard Tag Library (JSTL) core. We later use the <c:out> tag to write out the date. The date we've now stored in Java bean by using the built in <jsp:useBean> tag.

Although this particular example is more verbose it no longer embeds Java code. This is useful in that the tags provide a degree of abstraction and allow developers who otherwise don't have Java expertise to use tag library collections like JSTL and the well known Struts framework to produce web pages.

3.3 Web Applications and Web ARchives (WAR)

Structurally, a web application is a set of libraries and web resource files packaged together in a directory structure as set out in the Servlet specification. A Web ARchive or WAR file is simply this directory structure ZIP compressed into a single archive to facilitate installation in a Java EE server.

3.3.1 Directory Structure

A web archive will generally contain web resources such as HTML files, images, and JSPs placed in directories logically organized for web clients to browse. The one exception in terms of visibility is the **WEB-INF** folder which is explicitly made inaccessible outside the web application. This folder's structure is as follows:

- **/WEB-INF/web.xml** – the deployment descriptor
- **/WEB-INF/classes** – folder that contains class files
- **/WEB-INF/lib** – folder that contains libraries in the form of JAR files

The **/WEB-INF** folder is also used for other application specific folders and resources. These resources are accessed using the **ServletContext**'s **getResource()** and **getResourcePaths()** methods.

3.3.2 Web Application Class Loader

The servlet engine will create a "web application class loader" consisting of the contents of **/WEB-INF/classes** and each JAR file in the **/WEB-INF/lib** folder. This class loader is used to load the servlets, JSPs, and other elements described in the deployment descriptor [43].

One of the fundamental reasons for running your application using a Java EE server container is to allow the sharing of the engine's resources. Subsequently, the web application class loader is constructed with a parent that has visibility to a "common" subset of the engine's classes. Here's a typical class loader arrangement.

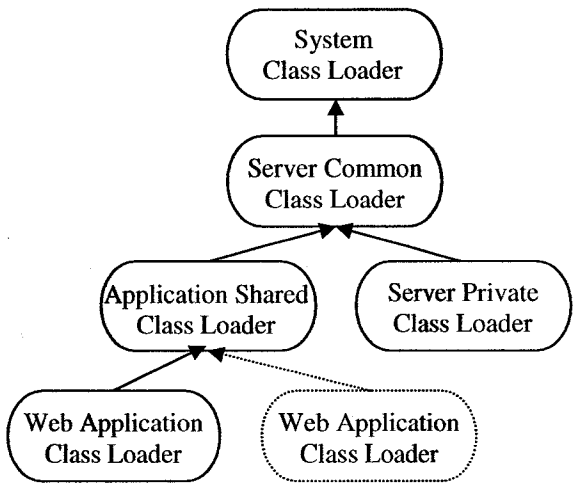


Figure 3-3: Web Application Class Loader Hierarchy

The **server common class loader** normally will hold classes fundamental to building a servlet application like the Servlet API, JSP Engine and API, as well as other key infrastructure. The **shared application class loader** is sometimes used to allow web applications to share resources.

One problem that has had an adverse affect on portability of a web application is the leakage of non-API packages from the servlet container's class loaders. Different implementations will expose different sets of classes and this can lead to class name collisions and unexpected behavior.

Two techniques used with moderate success to alleviate these sorts of problems are parent-last loaders and package renaming of the server's common classes. Both

approaches muffle the problem without really dealing with it and ultimately further illustrate the modularity difficulties in web applications

3.3.3 Web Application Deployment Descriptor

The web application deployment descriptor or simply web.xml contains the declarations for all the various elements of the application. This list of configuration elements includes:

- ServletContext initial parameters
- Servlet Definitions and Mappings
- Filter Definitions and Mappings
- Listeners
- Tag Library Definitions
- Welcome Files
- Error Handlers
- Security

The primary role of the deployment descriptor is to provide sufficient meta-information to describe how the servlet container should go about initializing the web application, handle errors and security. For the most part this involves instantiating the various classes detailed in the correct order using the web application class loader.

The deployment descriptor also provides information to correctly map servlets and filters to incoming requests. Here's a sample servlet definitions and set of mapping to illustrate the process and types of mapping rules.

```
<servlet>
  <servlet-name>Helloworld</servlet-name>
  <servlet-class>my.package.Helloworld</servlet-class>
</servlet>
…
<servlet-mapping>
  <servlet-name>Helloworld</servlet-name>
  <url-pattern>/hello</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>Helloworld</servlet-name>
  <url-pattern>/helloworld/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>Helloworld</servlet-name>
  <url-pattern>*.hello</url-pattern>
</servlet-mapping>
```

Figure 3-4: Sample Servlet Mapping Rules

An incoming request is ultimately matched against exactly one rule. The rules for matching are as follows:

- **/hello** is called an exact match rule as incoming request must exactly match it.
- **/helloworld/*** is called a path mapping and will also accept requests to /helloworld/1 for example. An exact match or longer matching path mapping will be used in preference to this rule.
- ***.hello** is called an extension mapping and in this case will map to any request that ends in ".hello" An extension mapping will be used except when there's an exact match.
- JSPs have special status in that there is usually an **implicit extension mapping** for *.jsp provided by the servlet engine that allows for an internal JSP Engine to handle the request. This implicit extension mapping can still be over-ridden in the web.xml with an explicit mapping entry.

3.4 Servlet Containers

To run your Servlets you deploy your web application into a managed environment called a Servlet Container or Servlet Engine. The servlet container provides the implementation that takes the incoming HTTP or HTTPS requests and translates them into a representation suitable for consumption by the Servlet API. A servlet container may be listening on multiple ports and is free to make changes to this set at runtime, subsequently a servlet applications should be cautious when making assumptions on how they can be communicated with externally.

3.4.1 Application Isolation

To provide cross-application isolation servlet containers use separate class loaders for each individual web application and provide separate private "temp" file areas. In addition prior to servicing an incoming request the servlet container will set the Thread's context class loader to be the matching web application's class loader.

Threads may be shared across web applications and subsequently the use of thread locals should be avoided. One potential problem that can occur is the reference maintained in a thread local variable will prevent the web application class loader from unloading properly as there are no guarantees a web application will see the Thread again and have a chance to cleanup

VM singletons pose a particular challenge for web application isolation. java.lang.System contains a set of properties that many libraries use for configuration. Servlet applications should avoid setting these at runtime as this may interfere with the configuration of other applications. System properties should instead be managed only by the servlet container.

3.5 Summary

In this chapter we've provided an introduction to many of the elements of Java web applications. The Servlet API provides the necessary building blocks for building web components in Java. With JavaServer Pages we have a script friendly approach for building web pages. Web applications are packaged and deployed using WAR files. The web.xml deployment descriptor is used to create the relationship between the web application and the servlet container. Finally we concluded the chapter by looking at the servlet container and the application isolation it affords.

In the next chapter we look at OSGi's support for modularity and servlet applications.

Chapter 4: OSGi Bundles, Framework and Services

"OSGi services are more like jumping beans; they have a life of their own."
-Peter Kriens [32]

OSGi is a technology for building dynamic software in Java. As the above quote indicates, in an OSGi system components and services are live and can come and go. The core specifications created by the OSGi Alliance [38, 39] support this functionality providing module, lifecycle, and service interaction layers in their architecture. This technology has evolved gradually over nearly ten years and with the adoption by Eclipse and more recent approval in the Java Community Process as JSR 291 it seems poised for mainstream acceptance.

In this chapter we introduce OSGi technology examining its support for modularity and for building server based applications. Section 4.1 introduces bundles, the unit of deployment and modularity in OSGi, and examines how they are specified. Section 4.2 examines the key features of an OSGi framework looking at the module, lifecycle, and service layers. In section 4.3 we examine those specified OSGi Services relevant to this thesis. In particular we look at the Http Service as it provides support for building servlet applications. There are many technologies related to OSGi and in section 4.4 we look at a few of the more important examples.

4.1 Bundles

OSGi [40] defines a unit of modularization and deployment called a bundle. As a deployment artifact a bundle is a JAR file that contains a collection of related classes and resources, but perhaps most significantly it makes use of the JAR's manifest file to provide contribution and dependency metadata. An OSGi framework makes use of this information to "resolve" dependencies and allow the sharing of classes between bundles in a controlled and well defined manner.

4.1.1 Bundle Manifest

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Http Services Servlet
Bundle-Vendor: Eclipse
Bundle-SymbolicName: org.eclipse.equinox.http.servlet
Bundle-Version: 1.0.0.v20070715
Bundle-Classpath: .
Bundle-Activator: org.eclipse.equinox.http.servlet.internal.Activator
Export-Package: org.eclipse.equinox.http.servlet;version="1.0.0"
Import-Package: javax.servlet;version="2.3",
 javax.servlet.http;version="2.3",
 org.osgi.framework;version="1.3.0",
 org.osgi.service.http;version="1.2.0"
Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0,
 J2SE-1.3
```

Figure 4-1: Bundle Manifest Example

All JAR file manifests are required to contain a Manifest-Version however OSGi adds a number of interesting new headers to describe the bundle.

- **Bundle-ManifestVersion.** This header is used to identify which version of the OSGi platform this bundle uses. A value of 1 should be used for version 3 and earlier, 2 for release 4 and later. The current release of OSGi is version 4.1.
- **Bundle-Name.** The Bundle-Name header provides a short readable name for human consumption. This is typically used in management consoles.

- **Bundle-Vendor.** The vendor or entity that provides this bundle sets this. In this example Eclipse is the provider.

- **Bundle-SymbolicName.** Together with **Bundle-Version** this header is used to uniquely identify a bundle inside an OSGi framework. The Bundle-SymbolicName is assigned by the developer and should be constructed based on the reverse domain name convention.

- **Bundle-Version.** This header is used to specify the version of the bundle. Versions can contain up to four parts, the first three being numbers and the fourth a qualifier that can include characters. If not present a version of 0.0.0 is assumed.

- **Bundle-Activator.** The Bundle-Activator head provides the name of a class that should be instantiated and then started and stopped in line with the bundle's lifecycle. We'll cover this in greater detail when we look at the lifecycle layer.

- **Bundle-Classpath.** This header is used to define those resource paths that should be used to create the bundle's class loader. If not specified the classpath is simply "." meaning the root directory of the bundle, however embedded jars and directory can be added. For example [`Bundle-Classpath: /jar/http.jar, .`] would create a bundle classpath containing the noted JAR file as well as the root directory of the bundle. Referencing JAR files outside of the bundle is not supported.

- **Export-Package.** A bundle uses the Export-Package header to express the comma separated set of Java packages it can provide to other bundles to help resolve dependencies. All packages are versioned and if not done so explicitly are assumed

to be 0.0.0. Details of the resolution process are provided when we look at the module layer.

- **Import-Package.** A bundle expresses dependencies with the Import-Package header and resolves its dependencies against other bundle's Export-Package declarations. For versioning, this header permits the use of constraints that permit specification of inclusive and exclusive ranges for the minimum and maximum acceptable version values. A bundle can declare a particular constraint as optional. More details are provided when we look at the module layer.

- **Bundle-RequiredExecutionEnvironment.** The Bundle-RequiredExecutionEnvironment header provides a list of execution environments that must be present in order for this bundle to operate. Typically these environments are upwards compatible mean the requirements for J2SE-1.3 are also met by a J2SE-1.4 environment.

A bundle will generally not contain all possible headers and this particular bundle is no exception. We'll briefly mention five other significant headers.

- **Dynamic-ImportPackage.** This header can be used to provide a set of import package constraints that are implicitly optional and only should be resolved and imported on demand. In addition this header supports the use of the '*' wildcard to allow matching a package prefix like "org.eclipse.*" or even all packages with '*'. Dynamic-ImportPackage should be used with caution, however it can help resolve problems when the classes that need to be loaded are not determined until runtime.

- **Required-Bundle.** This is similar to an Import-Package constraint however instead of being resolved by an Export-Package this header requires that a bundle with a matching Bundle-SymbolicName be present.

- **Bundle-NativeCode.** In addition to Java classes, bundles can be used to contain and run native code via the Java Native Interface (JNI). This header is used to provide the location of native code library in the bundle as well as additional attributes to describe for which platforms the native code should be used.

- **Bundle-Localization.** Bundles support the localization of the information contained in the manifest. For example the Bundle-Name can be localized by prefixing the value with "%", so for example "%bundleName" could be used. At runtime this headers value is used to find the location of properties file according to the Java Localization rules and then do the lookup based on the "%" prefixed name. If not present a default value of OSGI-INF/l10n/bundle is assumed.

4.1.2 System Bundle

In OSGi all Java artifacts are deployed as bundles and this includes the framework implementation itself. The bundle that contains the framework implementation is called the System Bundle and is by definition always installed and ready to use.

Since the System Bundle is the first bundle in a system it is not permitted to use Import-Package to resolve dependencies. Instead it must resolve all its dependencies either internally or else from the environment used to start the framework. In this respect the System Bundle is unique. It is the only bundle that has direct visibility of classes and

resources external to the framework. One of its key responsibilities is to export those non-java.* packages from the JRE and external environment necessary for operation.

4.1.3 Fragment Bundles

Fragments are bundles that are attached to another existing bundle and provide additional classes, resources to it. The content provided by a fragment is treated no differently than regular content however it does not over-write existing resources or classes. Two of the primary use-cases for fragments are to provide additional language files for use in localization and to deliver platform specific native code.

The manifest of a fragment bundle is permitted to import and export packages like a regular bundle however it must also contain a **Fragment-Host** header. This header is used to provide the Bundle-SymbolicName of the bundle to which this fragment should be attached. For example: `Fragment-Host: org.eclipse.equinox.http.servlet`

4.1.4 Extension bundles

The OSGi Release 4 specification added extension bundles as a means to provide optional pieces to the framework implementation and also additions to the boot class loader. Extension bundles are delivered as fragment bundles to the System Bundle. Like regular fragments the manifest of an extension bundle can contain Export-Package headers but must not contain any Import-Package or Require-Bundle headers since it must resolve with the same visibility restrictions as the system bundle. There are two types: **bootclasspath** and **framework** and these are declared in the Fragment-Host header using the "extension" attribute.

```
Fragment-Host: system.bundle; extension:=bootclasspath [or framework]
```

Boot class path extension bundles are meant to allow the delivery of resources that will be used to augment the JRE's boot class loader. The canonical example is the delivery of java.* resources like an implementation of the java.sql package for JSR 169. Boot class path extensions are extraordinarily powerful however they require tight control and integration with their target VM. Because of this tight coupling, boot class path extension bundles are an optional part of the specification and not reliably implemented by framework implementations.

Framework extension bundles provide the capability to augment the underlying framework implementation to provide optional features. For example, a framework extension bundle could be used to provide classes that alter certain aspects of how the framework performs class and resource loading. Similar to the "boot class path" variant this type of extension bundle has its resources directly appended to the framework's class path and its exported resources become accessible from the system bundle.

4.2 OSGi Framework

The OSGi framework provides a layered architecture that builds on the capabilities of the Java language and associated class libraries. These layers are:

- **Module (L1).** Provides the runtime support for bundle creation, dependency resolution, and class sharing.
- **Lifecycle (L2).** Manages the lifecycle of bundles including runtime install and uninstall operations.
- **Service (L3).** Provides an in process publish/find/bind sevice registry to promote loose coupling of bundles.

4.2.1 Module Layer (L1)

The module layer works with bundles and is chiefly concerned with using the metadata from the bundle's manifest to allow sharing of classes and resources while maintaining strong modularity guarantees. The primary mechanism used by the module layer to provide this support is the class loader.

Bundle Class Space

Each non-fragment bundle is associated with a class loader and it's through the delegation support provided by the OSGi framework for wiring dependencies that a bundles class-space is created. A bundles class-space is the product of:

- java.* resources (from the boot class loader)
- imported packages (static and dynamic)
- required bundles
- bundle-classpath (private packages)
- attached fragments

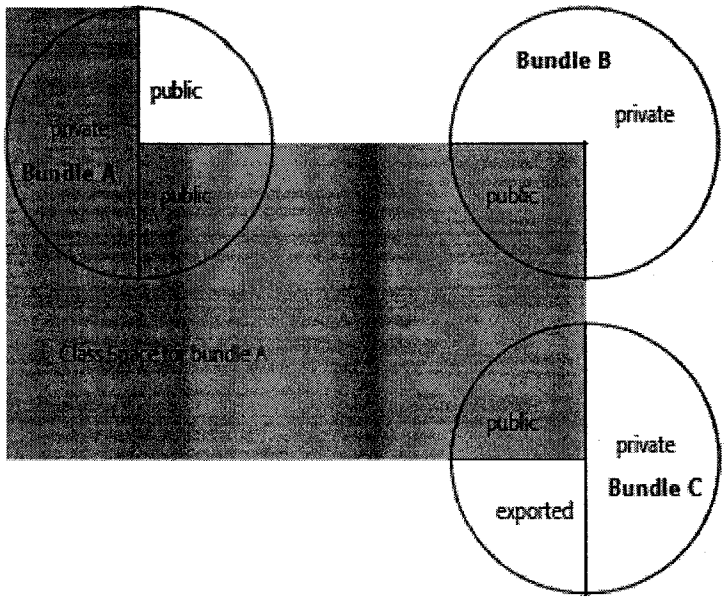


Figure 4-2: Bundle Class Space

A bundle does not have implicit visibility of resources from the System Class Loader and must explicitly import all resources that are not in the java.* package. This includes packages like javax.xml.parsers that in some environments come with the JRE. In these environments the System Bundle exports these packages on behalf of the runtime environment however bundles still have to import them.

Resolving

Before a bundle can be used or otherwise participate in the sharing of classes the framework must first ensure that all of its constraints and dependencies are resolved. Resolving is the process by which wires are created from importer to the exporter of a particular package.

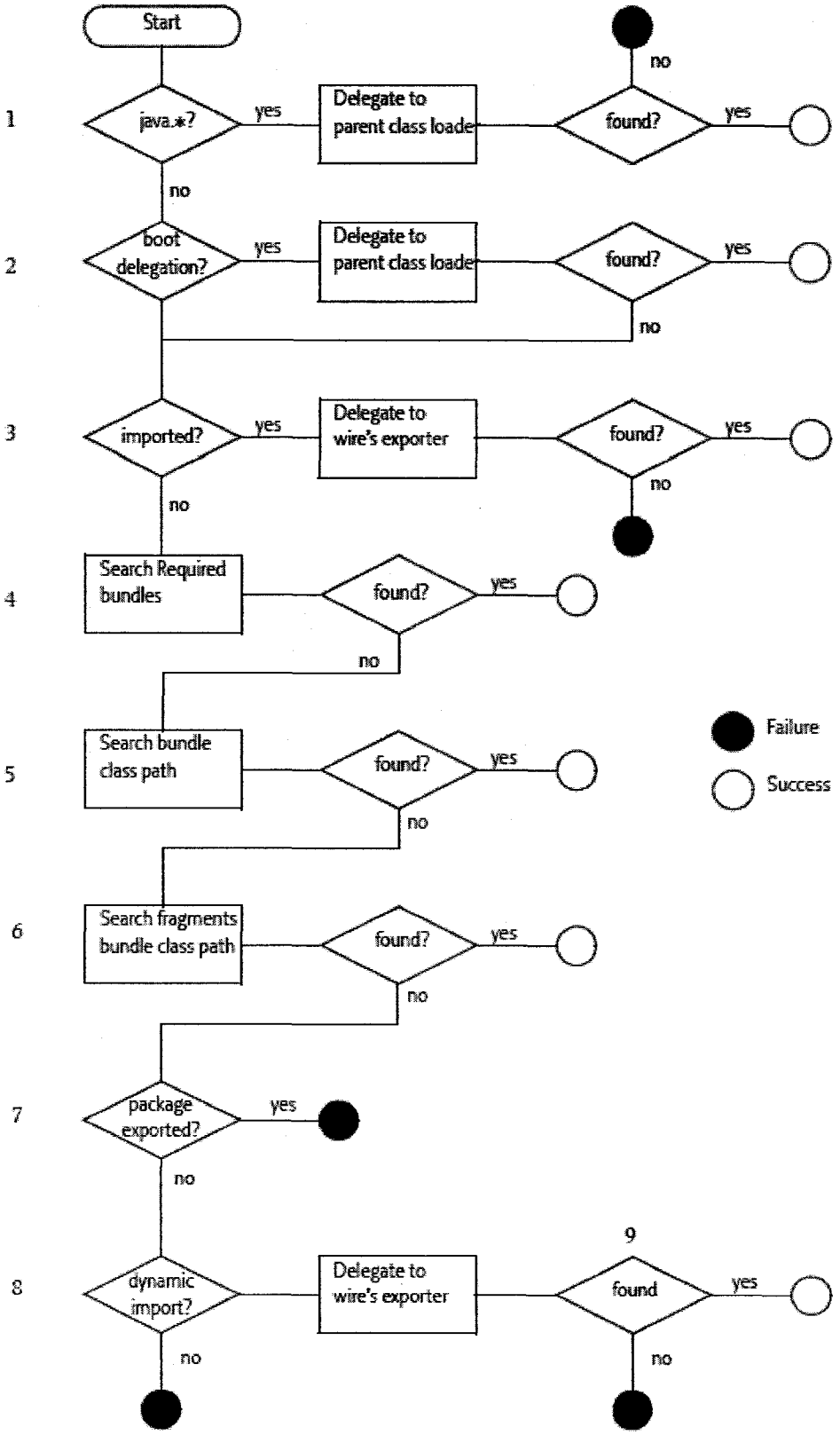
The first step in resolving a bundle is to first ensure that all of its fragments are attached. After this it is up to the framework to start the constraint solving process to match importers and exporters as well as validate availability of required bundles. A

bundle is deemed resolved if all mandatory imports are wired and required bundles present.

Runtime Class Loading

When a bundle's class loader is asked to load a class, there is strict ordering for how the resource should be looked up. Figure 4-3 provides a flowchart that runs through the process. Two concepts warrant definition before looking at this chart:

- **Parent class loader.** The OSGi specification does not define precisely which class loader is the parent however it's prudent for a bundle developer to make no assumptions beyond the contents of the boot class loader.
- **Boot delegation.** This refers to the set of packages above and beyond the java.* packages that should automatically be delegated to the parent class loader. By default this property (org.osgi.framework.bootdelegation) is empty.



[40]

Figure 4-3: Runtime Class Loading Flow Chart

Resource Loading

In addition to supporting resource access through the bundle's class loader, the Bundle class also provides public API to directly access the bundle's local resources.

```
java.net.URL getEntry(java.lang.String path)

java.util.Enumeration getEntryPaths(java.lang.String path)

java.util.Enumeration findEntries(java.lang.String path,

    java.lang.String filePattern, boolean recurse)
```

These methods returns paths and bundle entry URLs to resources in the bundle. The bundle entry URLs returned might use implementation specific schemes, but are specified to support relative access to other resources in the bundle.

4.2.2 Lifecycle Layer (L2)

The Lifecycle Layer manages all aspects of the lifecycle of a bundle including installation, activation, deactivation, and un-installation. Once installed in an OSGi framework each bundle is represented by a **Bundle** object. The Bundle object provides methods to **start()**, **stop()**, **update()**, and **uninstall()** the bundle. Installation is performed from another a bundle using the **install()** method on the **BundleContext**. We discuss the BundleContext later in this section when we look at activation.

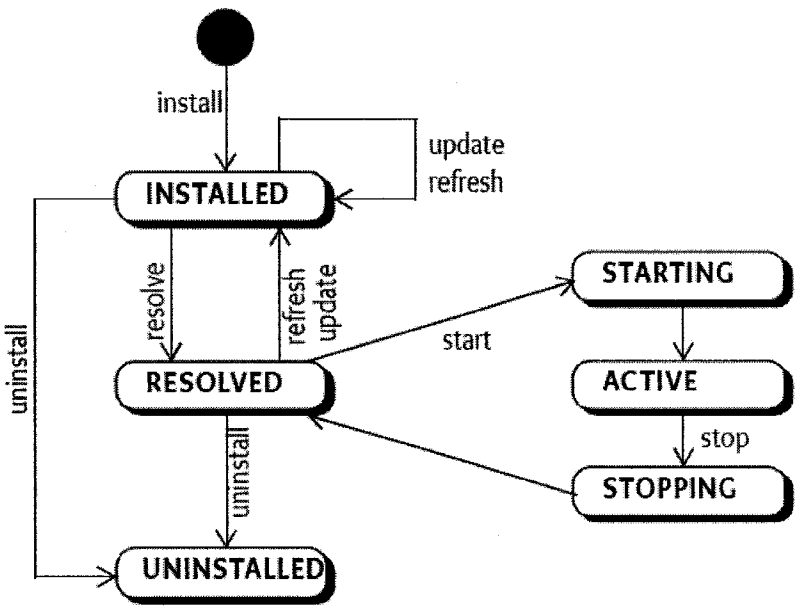
Bundle State

A bundle can be in any of the following states:

- **INSTALLED**. Installed and ready to be resolved
- **RESOLVED**. All constraints met and ready to be started

- **STARTING.** In the process of being started but not yet complete. This occurs when the **Bundle-Activator** has been instantiated and started but the call has not returned yet.
- **ACTIVE.** The bundle has been successfully started and is running
- **STOPPING.** In the process of being stopped but not yet complete. The **Bundle-Activator** has been stopped but the call has not yet returned.
- **UNINSTALLED.** The bundle has been uninstalled and once dependencies are refreshed the associated resources of this bundle can be re-claimed.

We show the bundle state diagram along with the triggering events in Figure 4-4.



[40]

Figure 4-4: Bundle State Diagram

Installation

Installation is the process by which a Bundle object is created and made available to the platform. A key step in the process is to validate that the bundle meets the format

requirements of the platform and that the Bundle-SymbolicName and Bundle-Version pair provide a unique identity for the bundle.

Resolving

Bundle resolution has already been covered in more detail in the module layer however it's worthwhile to add that the framework provides the control of when to start the resolution process. This is useful when installing large sets of bundles at one time that have interdependencies and should be resolved as a group.

Updating

The framework provides explicit support for updating or refreshing the contents of a bundle. This is very similar to installation however any internal state the bundle has recorded is maintained.

Activation

The relationship between the Framework and all bundles in the system is captured by the BundleContext object. This object also captures the identity and capabilities of a bundle and subsequently should not be handed to code in other bundles without caution. During activation a BundleContext object is created and passed into the bundle's BundleActivator.start() method. The activation process is completed when this call returns.

Deactivation

During deactivation the stop() method is called on the bundle's BundleActivator if present. After this call has returned the associated BundleContext object is no longer valid and will throw exceptions if used.

Uninstallation

When a bundle is uninstalled it is no longer available for other bundles to resolve against. Any existing bundles using it will maintain links to packages in this bundle until the framework explicitly refreshes those dependencies or re-resolves the importing bundle.

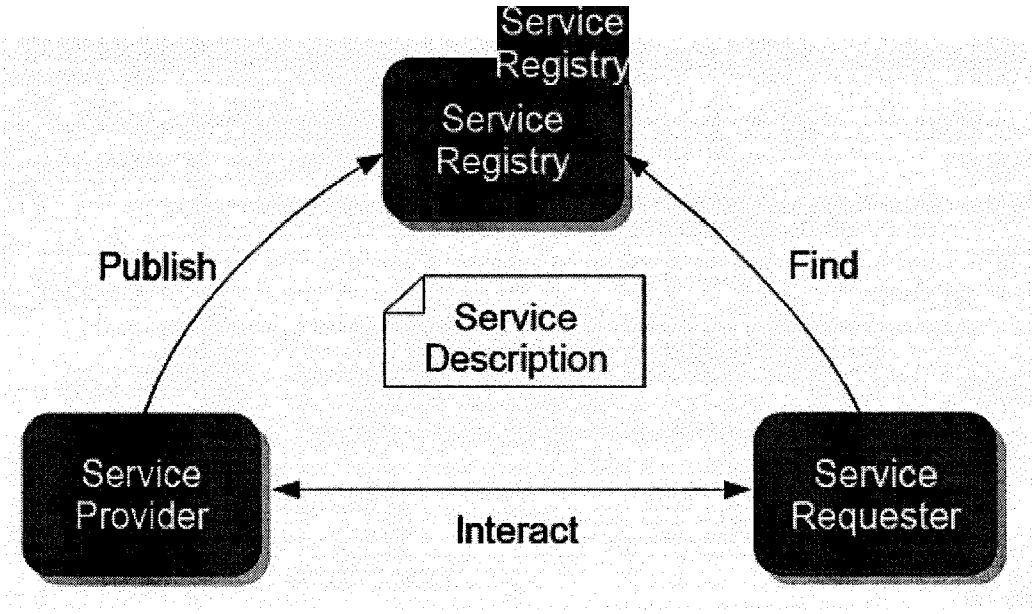
4.2.3 Service Layer (L3)

The service layer provides registration facilities for service implementers to register their service implementations, and likewise service users a mechanism to find and bind an implementation to a service interface. Bundles can interact with services by using methods from their BundleContext. This implies that only bundles that are active or being activated may register or use services.

Service Registry

The service registry provides a service model that uses publish, find, and bind concepts. A service is **published** for a particular class or interface name along with any additional properties that further describe it and help a user determine if it's appropriate.

A service using bundle **finds** the service by requesting a **ServiceReference** that matches published criteria. A ServiceReference represents the service and provides access to the underlying properties and provides a point of indirection until the service is actually required. The final **bind** step resolves this indirection and occurs when the bundle calls BundleContext.getService() passing in the ServiceReference from the previous step. The service instance can then be used to interact with the service provider.



[22]

Figure 4-5: Service Registry Model

Service Events and the ServiceTracker

OSGi is a dynamic system where a service may become unregistered and direct use of a service object invalid. To support correct use of services the framework will publish events that can be listened for when services are registered and unregistered.

The management of these events is complex and hard to get correct. Although not an explicit part of the service layer, the OSGi specifies the **ServiceTracker** class as an abstraction over the event listening. When interacting with services use of ServiceTracker is an OSGi best practice since it dramatically simplifies many of the threading issues involved in dynamic service access and use.

4.3 OSGi Services

The OSGi Core and Service Compendium specify some twenty-five services covering matters integral to running a managed platform including logging, configuration, provisioning, and monitoring. Many of these services are out of scope and for purposes of

this thesis we will direct coverage to those services and specific elements that are directly relevant.

4.3.1 Conditional Permission Admin

The OSGi framework leverages the existing Java Security model to enforce code level permissions. The Conditional Permission Admin specification provides support for managing these permissions at runtime. This is necessary in an environment like OSGi because the permissions cannot be statically determined when arbitrary bundles can come and go without restarting the Java VM.

The Conditional Permission Admin specification performs this permission management by making use of Java's Access Control Context and Security Manager. In addition to the regular permission checks performed by the Java Security Model, this specification introduces the concept of a delayed permission which requires control of the Security Manager. It is the requirement to use the Security Manger that is relevant to this thesis. The SecurityManager is a VM singleton and the consequences of its use will be examined more fully in a later chapter when we look at our approach.

4.3.2 URL Handlers Service

The standard mechanisms in Java do not permit dynamic extension of the built-in URL support. The URL Handlers Service adds support for dynamically adding and removing new URL schemes and ContentHandlers at runtime.

The integration is seamless so that existing code can continue to use regular URL creation mechanisms. In order to support this integration the URL Handlers Service is

required to control of the static URLStreamHandlerFactory and ContentHandlerFactory in the URL and URLConnection classes respectively.

These two static factories are VM singletons and similar to Conditional Permission Admin's control of the Security Manger this has consequences related to the work being done in this thesis.

4.3.3 Http Service

From the outset OSGi technology was positioned for building server applications and to support the use-cases for building servlet-based applications the Http Service was developed. The Http Service was developed along side the Servlet 2.1 specification and provides basic support for the registration of servlets and resources.

One of the key differences is that this registration support is dynamic meaning servlets and resources can be both registered and unregistered at run-time. This also implies dynamic control of the Http Services URI space.

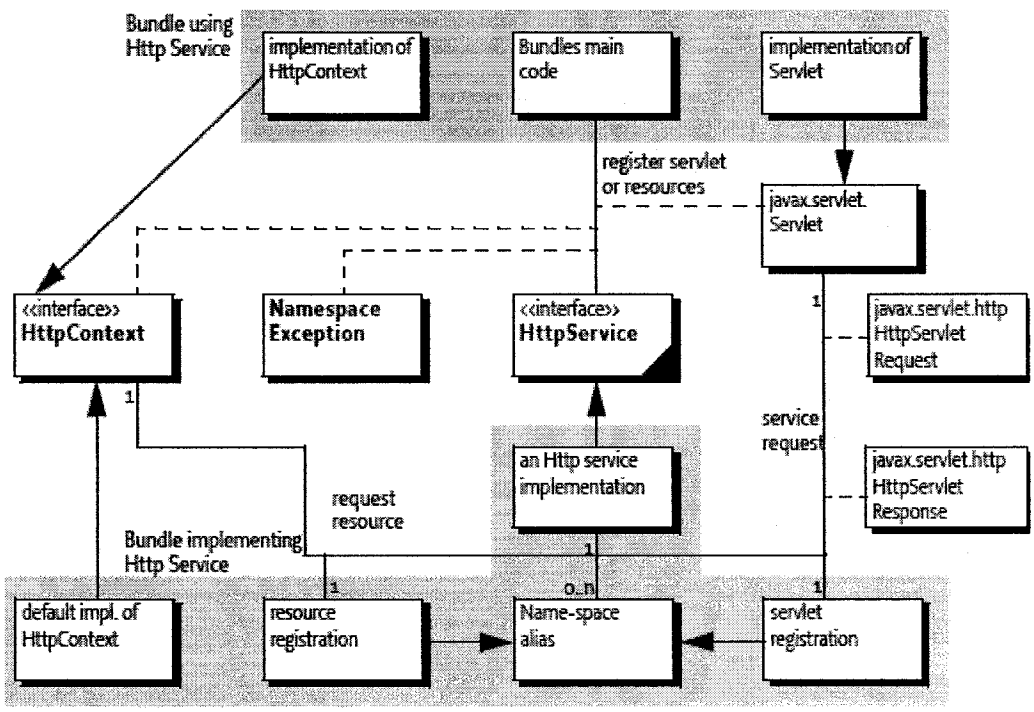
The Http Service builds on the Servlet API and is relatively small introducing three new classes.

- **HttpContext.** The HttpContext class is very similar to a ServletContext and provides similar access to resources, mime-type resolution, and authentication. One key difference when compared with the ServletContext is that the user is expected to sub-class and develop custom implementations of this class. It provides the following methods:
    - **getMimeType (String mimeType)**
    - **getResource (String path)**

- **handleSecurity (HttpServletRequest request, HttpServletRequest response)**

- **HttpService.** The HttpService class provides the service interface registered in the service registry and provides the following methods:
  - **createDefaultHttpContext ()**
  - **registerServlet (String alias, Servlet s, Dictionary initparams, HttpContext c)**
  - **registerResource (String alias, String path, HttpContext c)**
  - **unregister (String alias)**

- **NamespaceException.** This exception is thrown if there is an attempt to register more than one servlet or resource with a particular URI.



[41]

Figure 4-6: Http Service Overview

Default HttpContext

If when registering a servlet or resource the HttpContext is not provided a default is provided. This default HttpContext can also be created by using the **createDefaultHttpContext** method on the HttpService.

The default implementation will override the HttpContext methods as follows:

- **getResources** will return resources by calling getResouces on the registering bundles class loader.
- **getMimeType** will return null allowing the underlying servlet containers default mime type handling to be used
- **handleSecurity** will return true meaning the request may proceed.

Registering Servlets and Resources

Servlets and resources are registered with an alias which is roughly equivalent to a url-mapping when compared to the Servlet API. The alias must start with a "/" and exactly one registration per alias is supported.

The mapping of alias to incoming requests is done by finding the registration with the longest matching prefix. There is no special treatment for the "*" wildcard. The longest prefix matching system obviates this particular need since effectively all aliases are registered with a "*" wildcard at the end. The Servlet APIs concept of extension mappings like "*.jsp" is not currently supported.

Servlet registrations interact with the Servlet API as opposed to the Http Service. During the registration process the Http Service will use the registration parameters to

create the equivalent ServletConfig and ServletContext objects from the "initparams" dictionary and HttpContext object and call the Servlet's **init()** method.

Resource registrations provide a "path" parameter to allow the registering of directories of resources. The "path" parameter is used as a base directory to look up resources using the Httpcontext's getResource method. For example, a request for "/hello.html" might be translated into HttpContext.getResource("/resources/hello.html") if the "path" parameter registered was "/resources".

Unregistering Servlets and Resources

The Http Service provides a single **unregister** method since both servlet and resource registrations share the same URI space. If the alias unregistered is for a servlet registration the Http Service implementation will call **destroy()** on the servlet originally registered.

4.4 Related Work

OSGi received a huge boost in terms of number of users and exposure when Eclipse adopted it as its underlying component architecture. In the R4 release it incorporated a number of new ideas from the earlier module system in Eclipse, ModJava [9], and ModuleLoader [19, 21]. More recently the OSGi 4.1 release was approved as part of the Java Community Process (JCP) [26] as JSR 291 [39].

A major focus of the Java modularity community's effort is now on Java 7 as two new Java Specification Requests (JSR 277 and JSR 294) have been proposed for addition.

4.4.1 JSR 277: Java Module System

JSR 277 [55, 56] was created to bring consistency to the packaging, versioning, and deployment mechanisms in the Java Standard Edition. In particular it aims to add:

- A distribution format called a JAva Module (JAM) that's based on the JAR format but adds additional metadata.
- Versioning support and means to resolve dependencies between components
- A metadata aware repository that allows discovery and retrieval of modules.

The first two items directly overlap with OSGi. If anything it appears that OSGi provides richer support for metadata and dependency resolution. Support for a metadata aware repository on the other-hand is not currently provided OSGi and might be useful if there is a means to share modules between OSGi and JSR-277.

4.4.2 JSR 294: Improved Modularity Support in the Java Programming Language

One of the chief aims of JSR 294 [50] is to resolve the issues around Java packages and package private access. The problem is that there is no direct support for producing cross-package public and private API. In order to support this functionality this JSR proposes adding direct support into the language with the concept of "superpackages". "Superpackages" would allow you to create a second level of package membership and permit access to members. The current plan is to build this functionality in conjunction with the Java Modules provided by JSR 277.

This is functionally very similar to OSGi bundles providing Package-Export statements for those packages that are part of the bundles' public API. Language level

support would be beneficial here as internal bundle communications between packages must still use public methods.

4.5 Summary

In this chapter we've provided coverage of many OSGi concepts relevant to this thesis. OSGi provides a unit of deployment and modularity called a bundle. We've shown how bundles participate in the layered architecture that makes up an OSGi framework. The modules and lifecycle support provide a robust foundation for the services layer. Services are central to how OSGi exposes functionality and we've covered how they're supported in the framework. We've look at a few relevant service specifications including the Http Service used for creating servlet applications. Finally we looked at related work in Java modularity occurring as part of the Java Community Process and how it relates to OSGi.

With this chapter we complete the background discussion of the thesis. In the next chapter we look at the design and approach taken for supporting OSGi Web Applications.

Chapter 5: OSGi Web Applications: Design

Our main goal in this thesis is to develop an approach for building web applications from strongly modular components that support modification and reconfiguration at runtime. Web applications as defined by the Servlet specification do not meet these requirements. The modularity support in OSGi would, if it could be successfully embedded and used in Java EE environment. The challenge then is to demonstrate that this integration is both possible and provides a viable platform for building dynamic web applications.

In this chapter we discuss our design and highlight the decisions that shaped our implementation. Section 5.1 presents the design and discusses the three major components of our solution. In Section 5.2 we discuss decisions that shaped our design and implementation. Finally, section 5.3 examines additional criteria for ensuring applicability in real world applications.

5.1 Design Overview

Our design embeds and launches an OSGi framework implementation from inside of a web application. The web application registers a single servlet that acts as a front controller for the whole solution and delegates all servlet level interaction to handlers

running inside the OSGi framework. The name given to describe this arrangement is the **Servletbridge**.

The Servletbridge consists of three major components:

- **Bridge Servlet.** The Bridge Servlet provides the integration with the web application and manages deployment and launching of the embedded OSGi framework. At runtime it forwards servlet requests to the internal Http Service.
- **OSGi Framework.** The OSGi Framework provides the basis for modular components in our design and isolates us from the servlet engine's implementation details.
- **Http Service.** Our requirements call for the support of dynamic addition and removal of web resources. This capability is provided by a custom implementation of the OSGi Http Service specification designed to integrate with the Bridge Servlet infrastructure.

During runtime operation, servlet requests pass through the bridge servlet to the Http Service implementation where a registered servlet or resource handles the request. Before we can get to the point where this delegation is performed an initialization phase has to occur first. Figure 5-1 shows the initialization process.

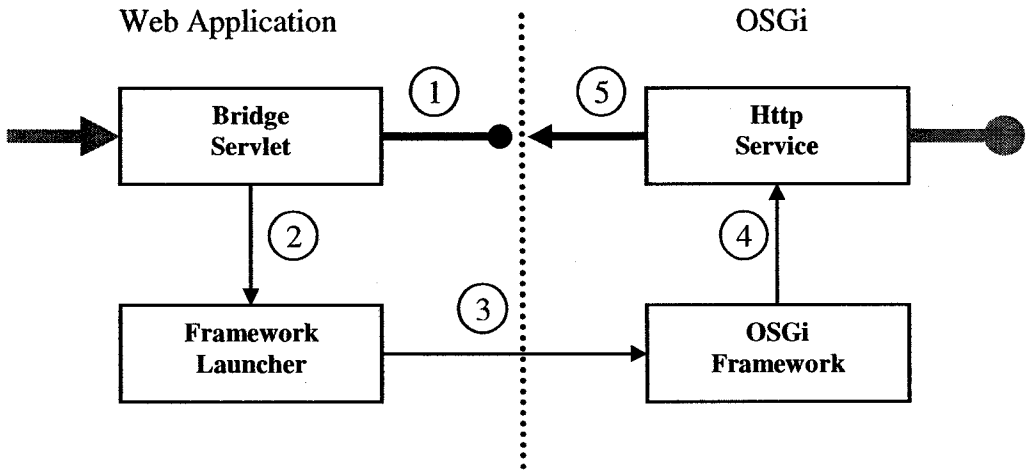


Figure 5-1: Servletbridge Initialization Process.

1. The Bridge Servlet creates a registration point where another servlet can register to handle requests. This registration point is accessed using "static" methods and later used by the Http Service implementation.

2. The Bridge Servlet creates and initializes the Framework Launcher. The Framework Launcher extracts configuration information from the "web.xml" and web application environment for configuring the OSGi environment.

3. The Framework Launcher first deploys the OSGi framework and then starts it.

4. The OSGi framework installs, resolves, and starts the specialized implementation of the Http Service.

5. The HTTP Service implementation registers a special "proxy" servlet with the Bridge Servlet. Future requests pass through this "proxy" where they are mapped to servlets and resources registered with the Http Service.

Termination is triggered when the Bridge Servlet asks the Framework Launcher to stop the OSGi Framework. This might occur because the web application is being stopped or uninstalled. Stopping the OSGi Framework will first cause all contained bundles to be stopped which in turn will cause the "proxy" servlet to be unregistered. Once the framework is shutdown the Framework Launcher discards the instance allowing it to be garbage collected and resources reclaimed.

5.2 Design Decisions

Although the design for the Servletbridge is relatively simple, when made concrete and placed in a real Java EE environment complexities arise. In this section we cover design decisions that impacted our implementation.

5.2.1 Servlet Container Environment

The Bridge Servlet operates within a web application which in turn runs inside a servlet container. This environment placed a number of constraints on our design and subsequently some decision making was required. Our main areas of concern were portability, multiple web applications, lifecycle, and security.

Portability

Web applications compliant with the servlet specification are meant to be portable from one application server to another. What makes this requirement complex is that classes from the server's implementation can leak into and conflict with those of the web application. To counter this problem, our design uses a parent-last class loader to isolate the OSGi framework classes. We use OSGi framework extension bundles to selectively

control which classes from the servlet container are made available to the broader set of bundles.

Another major difference between servlet container implementations is the version of the Servlet API. The most recent release is 2.5 however there are still many servers and applications that use version 2.3. The Servletbridge itself only requires 2.3 though an internal OSGi servlet application may require 2.4. Our design supports applications using Servlet API version 2.3 or higher by dynamically calculating and exporting the correct version from the web container into the OSGi environment.

Finally, many servers provide private API or proprietary elements in the web deployment descriptor. Our design does not make use of server-specific functionality.

Multiple Web Applications

Servlet containers allow you to install multiple web applications in one server. In addition, multiple instances of the same web application might be installed in a single container. This requires our design ensure that it does not interfere with the server's use of VM singletons. In addition where shared state visible to multiple web applications is used we're careful to use thread synchronization to avoid conflicts.

Lifecycle

Web applications have well defined lifecycle where Java EE servers provide controls to deploy, start, stop, and uninstall web applications. The Servletbridge is careful to honor these lifecycle requirements and transfers the same events down to the OSGi framework. When stopped the OSGi framework is careful to "join" and stop all threads

that it uses internally and remove any references that might prevent it from being garbage collected.

Security

In secure server environments where a Java Security Manager is used a policy file is used to grant or deny code access to protected functionality. An OSGi framework provides support for managing code level authorization internally however it needs to be bootstrapped appropriately. The Servletbridge implementation is designed with this authorization support in mind and is the only code in our solution requiring special privileges.

5.2.2 OSGi Framework Implementation

The integration of an OSGi framework in a web application environment is complicated by the fact that it's unclear who "owns" the process and subsequently controls the VM singletons. Prior to our work there had been very little investigation into running an OSGi framework where this control could not be assumed. It was clear that our design would require considerable modification of an OSGi framework implementation.

At the time the three major open source implementations were Equinox [11], Oscar (now Felix) [1], and Knopflerfish [30]. From a purely technical standpoint, any of these implementations would have been a reasonable choice to base our design on. What tipped the balance were primarily social considerations. Equinox is the OSGi implementation used as the underlying runtime for Eclipse [20]. We had previous experience interacting with the broader Eclipse community and coincidentally several of the Equinox project's

developers worked locally. Recognizing that our changes would likely require interaction with the core developers, we decided that Equinox made the most sense.

Eclipse and Equinox

Eclipse is probably best known as a Java development environment [46] however from the moment it was released to the open source community it has been evolving into a more general platform for building tools. One significant part of that evolution was the re-basing of the component model on OSGi technology that was done in the Eclipse 3.0 release. The project at Eclipse created to manage the transition and eventually house the new technology was Equinox.

One feature that has captured the attention of many developers is Eclipses's built-in support for **extensions** and the ability to create **plug-ins** to add new functionality. Both technologies are integral parts of the work done on the Equinox project; a plug-in is after all just an OSGi bundle and the extension registry is one of the project's most heavily used OSGi services.

Extension Registry

The extension registry provides a declarative means for a bundle to express a point of extensibility called an "extension-point" in a special file at the root of their bundle named "plugin.xml". Similarly, a consumer provides an "extension" in their "plugin.xml" to add new functionality. Many developers find this approach simpler to use than the OSGi service model. Use of the extension registry is part of our solution and discussed later in this chapter.

5.2.3 Http Service

Our design called for an API to support servlet application creation with dynamic characteristics and the OSGi Http Service provides basic support for this requirement. Unfortunately, there were no implementations that could integrate with an external web application. This meant we had to develop our own implementation. This proved beneficial as there were a number of integration challenges we were able to tackle more easily with this control.

Extended Servlet API Support

The OSGi HttpService was developed during the Servlet 2.1 [59] timeframe and subsequently does not currently provide support for a few critical portions of the Servlet API that are in common use. In particular there is no support for the following Servlet API concepts:

- ServletContext.getResourcePaths()
- Servlet Name access
- Extension Mappings (e.g. *.jsp)

These concepts are critical for providing a basic level of integration support to allow Web UI frameworks to operate in an OSGi environment. Our implementation specifically provides support for these concepts.

Our coverage leaves the following concepts unsupported.

- ServletContext initial parameters
- Filters

- Listeners

These unimplemented elements of the Servlet API are important in some situations. We decided they could be added in a later revision as they did not impede progress for developing our main concepts.

Declarative Support

In regular web applications developers register servlets by adding entries in the web.xml. When using the OSGi Http Service this same registration is instead done in code. In many cases a declarative approach is simpler and easier to understand. To support this usage style our design makes use of the extension registry and adds extension-points to allow XML based registration similar to web.xml's servlet declaration and mapping.

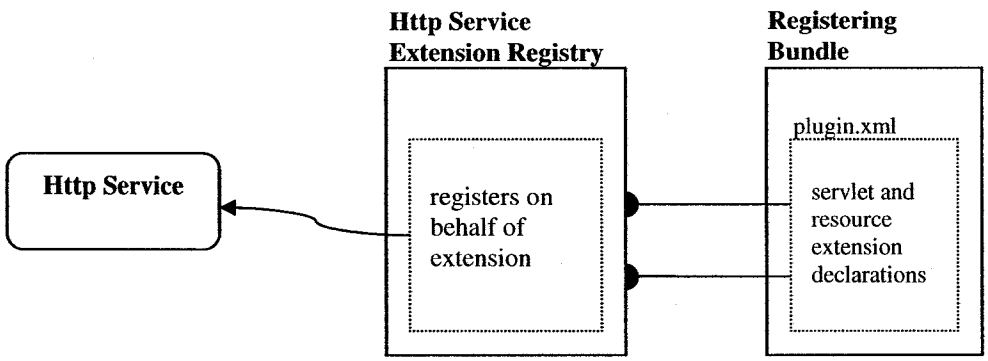


Figure 5-2: Http Service Declarative Support

Bundles can declare "servlet" and "resource" extensions in their "plugin.xml" file without having to write any OSGi specific code. Our implementation registers the declared servlets and resources with the Http Service "on behalf" of the bundle.

JavaServer Pages Support

JavaServer Pages are one of the most common approaches for building web pages in servlet applications. In addition many of the most popular Web UI frameworks leverage JSP and contribute extensive tag libraries to facilitate page construction. JSP support was essential for the success of our work however support as provided by servlet containers is fundamentally at odds with our integration.

Typically JSPs are compiled and run using the web application class loader. Our implementation embeds a JSP Engine in the OSGi framework and provides support for building an OSGi friendly class loader for compilation and runtime support. This was a significant undertaking and is covered separately from the Http Service implementation in the approach chapter.

5.3 Related Work

There is a number of related efforts underway specifically focused on integration of OSGi with server-side environments. Two of the most interesting are OSGi Application Servers and OSGi / Spring integration. We also discuss earlier work integrating pre-OSGi Eclipse with environments including Java EE.

5.3.1 OSGi Application Servers

A number of the application server vendors have announced that they have already or are in the process of adopting OSGi as the basis of their internal runtime. The list includes some of the largest players in the market (Websphere[24], Weblogic[6], JBoss[28]) and is likely to create interesting work at the OSGi Alliance's Enterprise Expert Group in support of their integrations.

The largest difference between application server integration and our work is that we are focused on providing support to user applications. As the OSGi based application servers mature there should be interesting overlap of the approaches.

5.3.2 Spring / OSGi Integration

Spring is one of the most popular application frameworks for building enterprise applications in Java. At its heart is an Inversion-Of-Control (IOC) framework that allows the injection of configuration information into Spring components at runtime. Recently the Spring community has announced their intention to adopt OSGi in their next release [25].

This effort is most certainly complimentary as there has already been work integrating the Spring work with the Servletbridge.

5.3.3 Eclipse Plug-in Based Applications and J2EE Components

Although based on Eclipse's earlier plug-in modularity as opposed to OSGi, this paper by Martin Buechi [7] describes integration of Eclipse in a number of environments including a web application environment. In particular this paper discusses concerns and an approach for avoiding problems with the URLStreamHandlerFactory VM singleton.

Our approaches are ultimately different however this paper provides some interesting coverage of the integration issues of embedding a modularity framework in other environments

5.4 Summary

In this chapter we provided an overview of the design of our solution, the Servletbridge. We examined key decisions related to our integration with the web

application environment, our selection of Equinox as our OSGi implementation as well as the custom Http Service implementation developed.

In the next chapter we take a more detailed look at our approach.

Chapter 6: OSGi Web Applications: Approach

OSGi Web Applications bring together the efforts from two communities with conflicting views on who controls the underlying platform. The integration of systems is rarely easy and our design is no exception.

In this chapter we take a detailed look at the key components of our approach. The Servletbridge provides the entry point to our solution. In Section 6.1 we take a closer look at how it's packaged in a web application and the techniques it uses to isolate its embedded OSGi framework from the servlet container implementation. Both OSGi frameworks and servlet containers want to assume control of Java's VM singletons. In section 6.2 we discuss how we resolved these issues. Section 6.3 examines our custom Http Service implementation and its capabilities. Finally, in section 6.4 we look at our approach for supporting JavaServer Pages in an OSGi environment.

6.1 Servletbridge

The Servletbridge includes the set of components that reside in the web application as well as the OSGi bundle that registers back with the Bridge Servlet. Together these components are responsible for both launching and providing the Servlet API communication channel to the embedded OSGi framework. In this section we discuss

how the solution is packaged in a WAR file, as well as details on launching, communication, and techniques used to provide isolation from the servlet container.

6.1.1 Web Application Structure and Deployment

The Servletbridge is designed to be packaged in a WAR file with a specific structure as follows:

```
WEB-INF/
    ➢ web.xml
    ➢ lib/servletbridge.jar
    ➢ eclipse/
        ➢ launch.ini
        ➢ configuration/
        ➢ plugins/
```

Figure 6-1: Servletbridge web application structure

The above structure is meant to closely resemble an Eclipse Rich Client Platform (RCP) [14, 36] application with the **/eclipse** directory holding the application containing components more suitable for server side interaction. The **plugins/** directory is used to hold the OSGi bundles including **org.eclipse.osgi**, the Equinox frameworks system bundle. Other file contents are:

- **web.xml** contains a servlet mapping assigning all requests to the Bridge Servlet
- **servletbridge.jar** contains the implementation classes
- **launch.ini** contains the framework property values

Equinox requires full permissions to write files in the **configuration/** directory and at runtime may deploy new bundles to the **plugins/** directory. Web applications can not portably or safely write in these folders as, strictly speaking, there is no guarantee that the

contents of the WAR file are unzipped and running off the file system. For example, it's conceivable that the WAR file may be running from a database.

As a result of this requirement the Servletbridge has a deploy phase where it copies the contents of the **eclipse/** directory to the web applications private "context temp" directory. Although temporary, in practice this location is kept until the web application is undeployed. As a precaution the Servletbridge implementation will verify the presence of files and re-copy if necessary each time the web application is started.

6.1.2 Framework Class Loader

The OSGi framework is not packaged in the web application lib or classes folder and subsequently is not available through the web application class loader. The Servletbridge implementation will instead construct a custom class loader that helps isolate the framework implementation from potential conflicting classes in the servlet container implementation and also provide the necessary permissions required internally.

The framework class loader is a specialized URLClassLoader that contains just the location of the system bundle JAR file deployed as described in the previous section. For Equinox this is the "org.eclipse.osgi" bundle.

To promote isolation this class loader uses "parent-last" delegation. By using "parent-last" delegation the framework class loader ensures that all classes in the system bundle are used in preference to those offered by the server. This has become a more important requirement recently, with application servers like Websphere 6.1 [24] also embedding the Equinox framework.

To support environments where a SecurityManager is installed, the framework class loader is provided with "All Permissions". An OSGi framework creates class loaders that are secured using the Condition Permission Admin specification. The system bundle itself must be run as trusted code.

6.1.3 Starting and Stopping the Framework

Equinox provides a class called **EclipseStarter** that is used to configure and launch Equinox. It's worth noting that the Servletbridge does not interact directly with the EclipseStarter class and instead uses Java Reflection to both create an instance using the framework class loader and call methods.

EclipseStarter provides a **startup()** and **shutdown()** which allow control of the Equinox framework's lifecycle. The Servletbridge uses these methods to remain synchronized with the web application lifecycle.

Prior to calling the startup method EclipseStarter has to be provided with configuration properties including a few file locations where the framework can store state as well as the set of bundles to initially install and start. These properties are typically set using System Properties in client side applications however this approach does not work in a server-side application since control of these properties cannot be assumed. To support the server-side environment our approach has added a **setInitialPropertiesMethod** to EclipseStarter that allows the passing of a map of name/value pairs use to populate the Framework Properties.

6.1.4 Setting up the Communication Channel

There are two sides to a communication channel: sender and receiver. In our solution the Servletbridge sends servlet requests to a recipient inside the OSGi framework where the request is ultimately handled. Both sender and receiver have to have a common message format and for that we use the classes from the Servlet API.

Setting up this channel of communication requires the use of a static registration point since the web application class loader does not have class-level visibility of the intended recipient inside the OSGi framework and cannot simply instantiate an instance. The **BridgeServlet** provides static **registerServletDelegate** and **unregisterServletDelegate** methods to allow the OSGi portion of the Servletbridge to register a proxy servlet. Once registered, future servlet requests are then delegated to the proxy where they can be handled by the OSGi Http Service implementation.

6.1.5 Exposing Classes from the Framework Class Loader

One requirement of this arrangement is that the OSGi bundle that hooks back into the BridgeServlet must have class level visibility of both the registration point and the Servlet API.

OSGi provides a number of mechanisms for exposing class resources from the parent or framework class loader to bundles inside the framework. In this section we'll discuss an implicit approach that uses boot delegation and a more explicit approach that uses framework extension bundles. Both approaches are useful in different situations when using the Servletbridge.

Boot Delegation

When a class is loaded from a bundle, the first step is to determine if it should be loaded from the "parent class loader". This is always the case for those classes in the java.* packages however the list of packages can be extended by using the boot delegation property (**org.osgi.framework.bootdelegation**).

Although this is a powerful technique, a drawback of using this approach is that one loses package versioning and other metadata that might normally be exposed in an Export-Package entry. It is still a useful approach when handling indirect class dependencies on classes from the JRE; it's not unusual to boot delegate sun.* and com.sun.* on Sun VMs.

In server environment boot delegation can be useful to provide visibility of server specific implementations of standard API. For example, some application servers override the default JAXP (Java API for XML Parsing) implementation. Bundles that use this API will typically only depend on the API and boot delegation provides a means to access the relevant implementation classes without having to modify the dependencies in the bundle manifest.

In addition to standard boot delegation the Servletbridge supports an Equinox-specific variation that delays the lookup on the parent class loader until after the regular bundle has been checked.

Framework Extension Bundles

The preferred approach for exposing explicit dependencies is to use framework extension bundles. These bundles are fragments of the system bundle and have full

visibility of the framework class loader. Framework extension bundles like other bundles can contain class files and other resources. However, it is the visibility of the framework class loader that allows for an interesting technique.

The Servletbridge creates a framework extension bundle that only contains a manifest and no other classes or resources.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Version: 1.0.0
Bundle-Name: Servletbridge Extension Bundle
Bundle-SymbolicName: org.eclipse.equinox.servletbridge.extensionbundle
Fragment-Host: system.bundle; extension:=framework
Export-Package: org.eclipse.equinox.servletbridge; version=1.0,
 javax.servlet; version=2.4,
 javax.servlet.http; version=2.4,
 javax.servlet.resources; version=2.4
```

Figure 6-2: Servletbridge Extension Bundle Manifest

This technique uses the visibility of the framework class loader to provide Export-Package statements for the Servlet API and Servletbridge implementation and allow other bundles to use and resolve against. In particular the Servletbridge's Http Service implementation can resolve against these exports and register with the BridgeServlet.

When compared with boot delegation, what's powerful about this approach is that resources exported in this fashion participate in the regular OSGi resolving process and are exposed only to those bundles that explicitly require them.

6.2 Embedding Equinox / OSGi

OSGi was historically used in set-top boxes and other scenarios where there was very tight control of the system. It's only more recently, in particular with the advent of Eclipse's adoption, that OSGi has been used for desktop and other more mainstream applications. One consequence of this history is that OSGi makes some fairly strong assumptions about the control of a Java process.

As discussed earlier, every Java type is identified by a pair consisting of the class name and the class loader that defined it. This allows more than one class definition to be active for every class with one exception; classes in the java.* packages can only be defined by the bootstrap class loader. This situation leads to true VM singletons wherever static fields are used in the Java class library.

VM Singletons can obviously cause great difficulty for integration when two systems both assume control. This unfortunately is the case for our design where we integrate Equinox inside of an application server. When running an OSGi framework inside a web application the problem is exacerbated as there might potentially be many instances active – one per web-app. Couple this problem with the fact that application servers are beginning to adopt OSGi frameworks that in turn might have conflicting use of the VM singletons and we came to the realization that when running Equinox inside a web application we can make few assumptions on control of VM singletons and instead must adapt to the parent container's expectations.

The most obvious example of a VM singleton is the java.lang.System object which among other things holds the references to:

- System properties
- the security manager

Two other VM singletons relevant to our integration are the URLStreamHandlerFactory and ContentHandlerFactory used in Java's support for URLs.

6.2.1 System and BundleContext properties

System properties have long been used as a configuration approach both in the VM and in application code. The challenge is that they effectively provide a flat namespace and can result in property name collisions when more than one component assumes control.

Our goal was to allow multiple instances of Equinox to run in the same VM and there were three problems with System property usage in an OSGi framework that need to be addressed.

- First, the framework itself was using System properties to configure itself. We needed an alternative that provided the same functionality but per framework instance.
- Second, we wanted use a property API and an approach that was non-proprietary. An effort was underway to ensure that the core bundles from Equinox could run on other OSGi frameworks and likewise accept bundles originally targeting other implementations. Our work could not impact this.
- Third, we had to take backwards compatibility into consideration. Equinox is the back-bone of Eclipse and we had to ensure that we weren't going to break existing use of System properties in the many pre-existing Eclipse plugins.

From this set of requirements the FrameworkProperties class was developed. The FrameworkProperties class is a per-framework singleton that provides the various property manipulation operations available via the System Object. This class is only visible inside Equinox's core system bundle (e.g. org.eclipse.osgi).

Other bundles access property information through the BundleContext's getProperty method. One caveat to access through the BundleContext is that properties cannot be "set", only retrieved.

Finally, to support backwards compatible access to System properties the FrameworkProperties class can operate in one of two modes controlled ironically enough by a System property during class initialization.

- In per-VM mode the FrameworkProperties class will simply wrap the System properties object.
- In per-framework mode a snapshot of the current System properties is used to create an initial set of properties, however after that these properties are independent.

With the FrameworkProperties class in place the challenge was to then make these modifications on a very large code base.

The code changes were themselves fairly simple and generally took the form of translating "System.getProperty("x.y.z")" to FrameworkProperties.getProperty("x.y.z"). In addition we had to take into consideration less obvious use of System Properties like for Boolean.getBoolean("x.y.z") which despite outwards appearances is actually creating a Boolean from the system property "x.y.z". All told there were several hundred of these little changes that had to be made. The changes were tested, submitted and eventually committed in early spring 2006 in time for the Eclipse 3.2 release.

6.2.2 SecurityManager and the Conditional Permissions Service

In many server-side environments the application server infrastructure will provide and set a custom Security Manager. This is done to improve the manageability of the server platform as a whole and to allow better control of the various permissions assigned to contained applications. This unfortunately conflicts with the OSGi Conditional Permission Admin specification, which for full compliance requires a Framework specific Security Manager to be set.

With Java support for code-level authorization, secured code will perform "permission" checks using the SecurityManager. Normally when SecurityManager.checkPermission is called the call is delegated to the AccessController. The Condition Permission Admin specification requires a special SecurityManager that will allow certain portions of a check to be deferred until after the AccessController.checkPermission call returns. This allows behavior that might typically be triggered during an access check to occur later. Without the Framework SecurityManager in place we lose the ability to defer permission checks and instead must handle all checks as they're encountered.

The basic security Condition objects provided by the Conditional Permission Admin specification do not require this deferred behavior. Furthermore, the typical use-cases for deferred Condition checking most often occur to prevent unnecessary user interaction, like prompting the user for login information, until the last moment.

This use-case is less applicable in server environments and as a result we decided to not override the existing Security Manager and instead operate at a lowered functionality level. This approach was then validated with Equinox to ensure feasibility.

6.2.3 Default Locale

Equinox provides support for setting a default locale and encoding. In a server environment it's important to not override the default locale. Similarly overriding the file encoding might result in problems. We chose to do nothing more than provide advice in this matter as these are normal restrictions for server-side applications.

6.2.4 URLs, Factories, and the URL Handler Service

URL protocol and content type processing are handled by two of the lesser known VM singletons:

- java.net.URL's URLStreamHandlerFactory
- java.net.URLConnection's ContentHandlerFactory.

The URLStreamHandlerFactory works with the URL class to produce the correct URLStreamHandler based on the protocol used. For example, when you create a URL object in Java [`URL eclipse = new URL("http://www.eclipse.org");`], the URLStreamHandlerFactory will provide a suitable URLStreamHandler for interpreting the protocol scheme and later retrieving and interacting with the protocol end-point.

The ContentHandlerFactory serves a related task when retrieving content from a URL and translating the incoming data stream into Java objects. The content-type is handed to the ContentHandlerFactory which returns a ContentHandler appropriate for the MIME type.

The default configuration mechanisms in Java for these two factories are static and fundamentally at odds with the way an OSGi system would like to operate. The OSGi URL Handler Service provides the ability to dynamically add and remove

URLStreamHandler and ContentHandler associations. This requires control of the aforementioned singletons and is particularly critical because many of the OSGi frameworks are implemented with dependencies on the ability to create custom framework URL types for handling bundle resources.

The URL Handler Service is a required service of the OSGi Core and without the URLStreamHandlerFactory and ContentHandlerFactory set most framework implementations will not operate.

The challenge with our integration is that virtually all application servers also set these singletons so we're left with an awkward situation where two technologies are at odds with one another. A further complication is that our approach calls for running an OSGi framework per web application. We need to take into consideration support for multiple active frameworks as well as lifecycle aspects involved when a framework instance is started and shutdown.

Initialization

The URL and URLConnection classes will only permit these two factories to be set exactly once. Our approach requires us to first retrieve the current factories and if present wrap them with a specialized factory of our own design. We then use Java reflection to "reset" the factory instance as well as any cached "handler" state. Finally we set our newly created factory instances.

Termination

When the last instance of the OSGi framework is shutdown we carefully reverse the process. Again we use Java reflection to "reset" the factory and cached handlers and then

set the two factories with their original values. Our specialized factory instances are now available for garbage collection along with the framework that provided their class definition. This whole process occurs while holding the relevant mutex so is thread-safe.

Multiplexing Handler Creation

The Equinox implementations of the URLStreamHandlerFactory and ContentHandlerFactory have built in support for multiplexing URL and Content handler creation requests internally and to other frameworks depending on where the request was initiated.

There are two pieces to our solution:

- A protocol for new framework instances to register and unregister with an existing factory.
- An approach for determining caller context

Protocol

When an instance of the Equinox framework is started it will try to install a URLStreamHandlerFactory using the URL.setURLStreamHandlerFactory method. This call will fail if a factory has all ready been set.

The next step is to determine if the set factory is capable of supporting multiplexing. The check is performed by seeing if the "class name" of the factory matches the Equinox factory implementation. If there is no match we perform initialization as described earlier. If, however, the current factory does support multiplexing we use reflection to "register" the new factory with the current multiplexing factory. It should be added that

all factories that are registered are the same factory class and subsequently can potentially support multiplexing.

Likewise once the framework is shutdown a similar process occurs in reverse. The first check done is to determine if the framework's factory instance is also the set factory. If the factories are different then we use reflection to call "unregister" and remove the factory from future consideration.

One additional complication that can occur is if the framework's factory is the currently set VM singleton. If there are other multiplexing factories registered we select one and register all other multiplexing factories with it and then set the singleton in a manner consistent with the original initialization process. If no other factories are registered we follow the termination process.

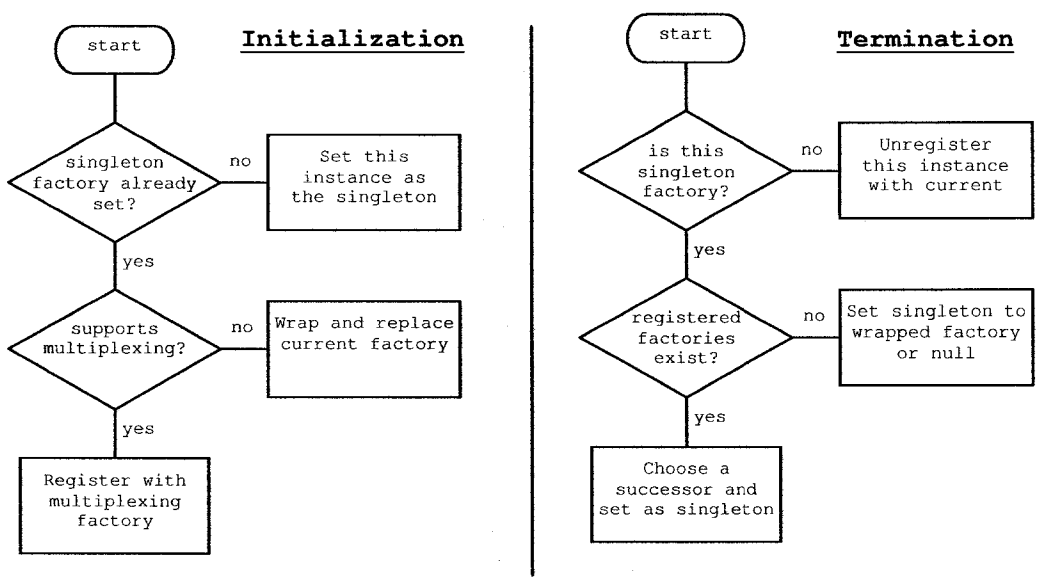


Figure 6-3: URLStreamHandlerFactory Initialization and Termination

Determining Caller Context

In determining caller context we're trying to figure out which registered factory instance and subsequently which instance of the OSGi framework to delegate to. In order to do this successfully we need two things: context information and a way to determine if the context applies to a particular framework instance.

SecurityManagers provides a "getClassContext" method that returns the set of Class objects on the current execution stack. In our factories we access this method in a private instance of Security Manager customized to allow use of this normally protected method.

Each framework instance can determine if a particular Class originated internally by using the OSGi Package Admin Service. This service provides a getBundle method that takes a Class as a parameter. Using this method a framework can determine if it originated a call based on whether or not a bundle is returned.

In order to support multiplexing, the registered VM singleton factory snapshots the class stack and starting from the top gives each framework the chance to claim context. If a framework claims context the request is delegated to that factory instance. Otherwise the process continues at each level of the stack until either a framework claims ownership or we reach the bottom of the stack. If the bottom is reached the request is delegated to the factory wrapped during the initialization process.

Portability

One significant challenge with this approach is the required use of Java Reflection to replace the factories and reset the handler cache. One factor that makes this approach

significantly less risky is that the URL and URLConnection classes have been around since Java 1.0 and have changed very little since that time.

Portability has been validated with the major Java class libraries used on the server including the Sun JDK, IBM J9, and BEA JRocket. Code analysis shows few code differences and the same basic synchronization patterns. The implementation is very cautious and forgiving and likely to work with any implementation. In addition it is maintained as part of the Eclipse project.

6.3 Http Service Implementation

The most common approach for an Http Service implementation is to integrate tightly with a specific servlet container. Our approach has been different in that we can integrate with many servlet containers through the registration of a single "proxy servlet". Additional features of our implementation include more up to date support for the Servlet API as well as a declarative style of servlet registration similar to what's used in a web application deployment descriptor. To facilitate porting exisiting web applications to the OSGi environment it's critical that the Http Service move towards parity with the current Servlet API. Our implementation is a step in that direction and aims to influence future versions of the specification.

6.3.1 Proxy Servlet

The ProxyServlet is a specialized servlet that when registered in a Servlet container will register and expose an OSGi Http Service inside the framework. In the context of the Servletbridge an instance of the ProxyServlet is what is registered with the BridgeServlet.

To the servlet engine it appears that all requests are handled by the ProxyServlet however it really is just a conduit to those servlets and resources registered in the Http Service it provides. Serving as a conduit forces our implementation to take on an increased role in mapping requests. Apart from path mapping our implementation is careful to delegate all other aspects of request handling back to the servlet container.

6.3.2 Path Translation

The path elements of a ServletRequest consist of:

- context path
- servlet path
- path info

, where a typical request they will look something like:

```
http://localhost:8080[/contextpath][/servletpath][/some/path_info]
```

A servlet registered in the Http Service expects the paths to be handled as set out in the Servlet specification. Since the ProxyServlet delegates to the final recipient in the Http Service it ends up having to take on part of the role of a servlet container and must adjust the path elements to correctly target the final servlet.

What this amounts to is a path translation as follows:

- new_contextpath = contextpath + servletpath
- new_servletpath = portion of path_info registered by the servlet
- new_path_info = remaining portion of path_info

For our original example and a servlet registered with "/some" we get:

- contextpath = [/contextpath/servletpath]
- servletpath = [/some]
- path_info = [/path_info]

Our implementation does not do internal request dispatching and instead leverages the support in the servlet container. Much in the way our implementation looks after servlet paths it must also handle the path translation issues when the RequestDispatcher is involved.

6.3.3 ServletContext and getResourcePaths

With a typical servlet application the ServletContext's resources come from the web application directory structure. For an OSGi Http Service application, resources come from the HttpContext object. Unfortunately there is not parity in terms of resource access methods as "**getResourcePaths**" was added to the ServletContext class in the Servlet 2.3 release of the specification.

Many Java web UI frameworks and even JSP engine implementations make use of the "getResourcePaths" method to discover configuration files at runtime. Our design extends the Http Service in this area and uses Java reflection to call this method if provided on the HttpContext.

6.3.4 Servlet Name

Servlet name support is not provided by the Servlet registration process in the Http Service. Some Web UI frameworks (Struts for example) use the servlet name to partition configuration information and depend on calls to **ServletConfig.getServletName** to return a reasonable value.

When registering a servlet, a dictionary of initial parameters is passed in. Our implementation will look for a "**servlet-name**" parameter and if present use its value in response to getServletName.

6.3.5 Extension Mappings

A feature in the web mappings permitted under the Servlet API is the concept of a file extension mapping like *.jsp or *.do. These mappings are used to simplify resource co-location so that for example JSPs can easily co-exist with image files in the same directory. The Http Service unfortunately does not support this style of URL mapping.

To support file-extension mapping while remaining in the spirit of Http Service a compromise style of mapping was created where a file-extension could be added as a more exact refinement of the longest prefix-match algorithm. For example, /a/b/*.jsp would be a better match for /a/b/my.jsp then simply /a/b.

6.3.6 Declarative Mapping

When a web application is deployed into a servlet container it must also provide a deployment descriptor that will describe among other things how to map incoming requests to a Servlet class.

When using the OSGi HttpService this same registration is typically done in code. Although equivalent, many developers prefer a declarative approach. Our implementation provides declarative support for use of the Http Service through the Eclipse extension registry. The extension-points provided are:

- httpcontexts
- servlets

- resources

These extension points provide a one-to-one mapping of the Http Service. The syntax used in the extensions is very similar to that used in web.xml, with a few exceptions like the mapping of "servlet-name" as an init-param instead of as a top-level element.

web.xml servlet declaration	**servlets extension**
<servlet> <servlet-name>test servlet</servlet-name> <servlet-class>my.TestServlet</servlet-class> <init-param> <param-name>testParam</param-name> <param-value>test param value</param-value> </init-param> </servlet> <servlet-mapping> <servlet-name>testservlet</servlet-name> <url-pattern>/test</url-pattern> </servlet-mapping>	<servlet alias="/test" class="my.TestServlet"> <init-param name="servlet-name" value="testservlet"> </init-param> <init-param name="testParam" value="test param value"> </init-param> </servlet>

Figure 6-4: Comparing web.xml and servlets extension syntax

6.4 JavaServer Pages Support

The JavaServer Pages engines provided by the application server vendors vary between one another and in most cases don't provide public API. Even if there was consistency between the engines and a well documented API it's unclear how we would provide the necessary class level visibility into our OSGi environment necessary for compilation. Instead, the approach we took for JSP support was to embed our own engine.

We selected the Jasper 2 engine from Tomcat [2] in particular because it had a history of being embedded in other application servers. Another important consideration was that Jasper 2 uses the "Eclipse Compiler for Java" (ECJ). A key feature that greatly facilitates integration with an OSGi environment is that ECJ allows compilation context to be

passed in the form of a class loader. "javac", the compiler that comes with Sun's JDK, for comparison, requires a classpath which it uses to create its own class loader for compilation context.

Jasper provides a special Servlet, the JspServlet, just for the sort of integration we required. In a regular web application JspServlet uses the ServletContext made available during initialization for resource lookup of JSPs and tag library discovery. For compilation and runtime execution JspServlet uses the thread context class loader.

Our implementation wraps Jasper's JSPServlet and hides all internal details of providing the appropriate ServletContext and handling interaction with the context class loader. A user can register our wrapped JSPServlet with the Http Service just like any other servlet. In addition we provide similar support for JSPs declared using the "servlets" extension-point.

6.4.1 Resource Lookup

Producing a suitable ServletContext for JSP resource lookup was greatly simplified by the fact that we already had this infrastructure in place for the HttpService. However, one additional requirement is placed on our implementation in order to support tag library discovery.

The tag library discovery process relies on using the "getResourcePaths" method on the ServletContext. This method cannot be mapped to a similar call on the HttpContext object. As discussed earlier our HttpService implementation will opportunistically use reflection to call this method however there is no guarantee that the method will be

present. In an effort to provide consistency with respect to resource lookup our implementation of the ServletContext will first try to find resource paths in the bundle.

6.4.2 JSP Class Loader

Jasper places a number of requirements on the context class loader set during compilation and runtime. This class loader must:

- be an instance of URLClassLoader
- have visibility of the Jasper engine
- have visibility of those classes referenced in JSPs or tag libraries

In Equinox the normally set context class loader is called **ContextFinder**. ContextFinder's "context" comes from the code that made the request for classes and resources. A call to ContextFinder.loadClass is mapped to the "caller's" ClassLoader.loadClass. This approach uses technique's that are sensitive to the distance on the stack between ContextFinder and whatever code is using it to load classes or resources and is unfortunately not amenable to being wrapped. To retain consistency our implementation mimics the lookup qualities of this class loader in **JSPContextFinder**.

To support these requirements we create a **JSPClassLoader** which is a subclass of URLClassLoader parented by a chain of class loaders consisting of:

- an instance of JSPContextFinder
- class loader for the JSP bundle
- class loader for the Jasper engine bundle

A bundle does not provide direct access to its underlying class loader however does provide similar access to classes and resources. Our implementation uses a special wrapper class called **BundleProxyClassLoader** (see Appendix A) to provide the mapping from bundle to class loader.

The URLs for the URLClassLoader come from any JAR files contained on the JSP bundle's Bundle-Classpath manifest header. These JAR files are searched as part of the tag library discovery process.

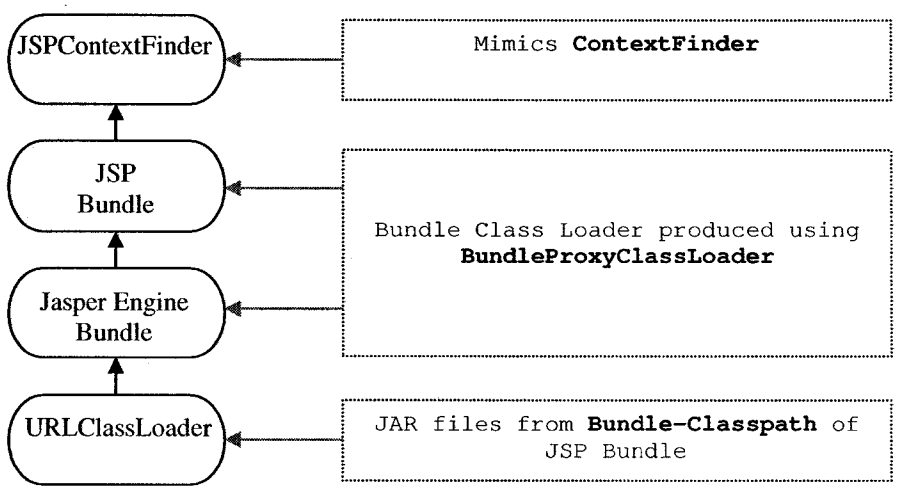


Figure 6-5: JSPClassLoader delegation hierarchy

6.4.3 Wrapped JspServlet and Extension Registry Support

With the fully constructed ServletContext and JSPClassLoader details ironed out our JspServlet implementation is ready to use with the Http Service. One difference with this servlet compared with others is that it is OSGi specialized and requires a Bundle as a parameter in its constructor. This is normally not a problem since use of the Http Service already requires a BundleContext which provides this. For example:

```
Servlet myServlet = new JspServlet(bundleContext.getBundle());
```

This requirement would however normally prevent its use with the "servlets" extension-point. Instead "servlets" extension should use:

```
class="org.eclipse.equinox.jsp.jasper.registry.JSPFactory"
```

6.5 Summary

In this chapter we've examined our approach in detail. The Servletbridge provides the entry point to our solution and we've looked at details concerning how we package a web application and the process of launching Equinox. The Bridge Servlet works together with our Http Service Implementation to establish a servlet based communications channel. We've looked at a number of issues around VM singletons and how we resolve difficulties in the Equinox framework. Our Http Service implementation is constructed to support registration with the BridgeServlet and improves on the support for the Servlet API. In addition it supports a declarative approach for servlet and resource registration using the Eclipse Extension registry. Finally, we looked at our support for JavaServer Pages in OSGi and how it integrates with the Http Service and our extension registry support.

In the next chapter we evaluate our approach and discuss relevant experience.

Chapter 7: Evaluation and Experience

The Servletbridge and its related components were developed at the Eclipse Foundation [13] as part of the Equinox project [11]. In 2005 shortly after the Eclipse 3.1 release there were a number of server-side OSGi discussions in the equinox news groups that culminated in the formation of a "server-side" incubator project [12, 29]. The initial work done in the incubator enabled Equinox to be embedded in a servlet container and was included as part of Eclipse 3.2. The remaining components including the Http Service implementation and JavaServer Pages support graduated from the incubator in spring 2007 and are now used in the Eclipse 3.3 SDK.

In this chapter we evaluate our solution relative to our goals and expectations. In section 7.1 we discuss our testing approach, environment, and related tools. Section 7.2 demonstrates the launching and integration of the Equinox OSGi framework from a web application. Section 7.3 demonstrates the dynamic addition and removal of web components to a live system. Section 7.4 demonstrates our support for multiple versions of components. In section 7.5 we examine our support for different application servers and Java runtimes. One of our contributions is JavaServer Pages support. In section 7.6 we show how our support was validated. The Servletbridge and its related components are used in several commercial and open source products. In section 7.7 we provide a brief overview of these products and their use.

7.1 Evaluation Environment

The Servletbridge is packaged as a WAR file and so naturally our validation environment requires a servlet container and a Java runtime. In addition to our components for the Servletbridge we add an OSGi management console to let us inspect framework status and manage the set of bundles and services installed.

7.1.1 Servlet Engine and Servletbridge WAR

To evaluate our implementation we create an environment that mimics a typical servlet engine setup. Our environment consists of:

- Microsoft Windows XP SP2
- Sun Microsystems Java Runtime Environment (JRE) 5.0
- Apache Tomcat version 5.5

The choice of operating system is arbitrary. The use of Sun's JRE and Apache Tomcat is a fairly common and reasonable setup both for developers and production environments. Tomcat 5.5 is particularly appropriate as it is the reference implementation for the Servlet 2.4 specification.

The Equinox server-side project distributes a WAR file that contains the Equinox OSGi framework as well the components developed for this thesis. This file is delivered at fixed intervals to coincide with the major releases of the Eclipse platform. In our environment we use the Servletbridge WAR file for the Eclipse 3.3 release.

To this WAR file we add one additional bundle, the Knopflerfish 2.0 Http Console, to help us demonstrate the dynamic capabilities of our solution.

7.1.2 Http Console

The Knopflerfish project [29], in addition to providing an OSGi framework implementation, offers a variety of useful bundles and service implementations. One of these bundles is the "Http Console" [30], which provides a simple management web UI for OSGi frameworks.

This console is an integral part of our test environment as it lets us examine the state of our running system and in particular manage the set of bundles installed.

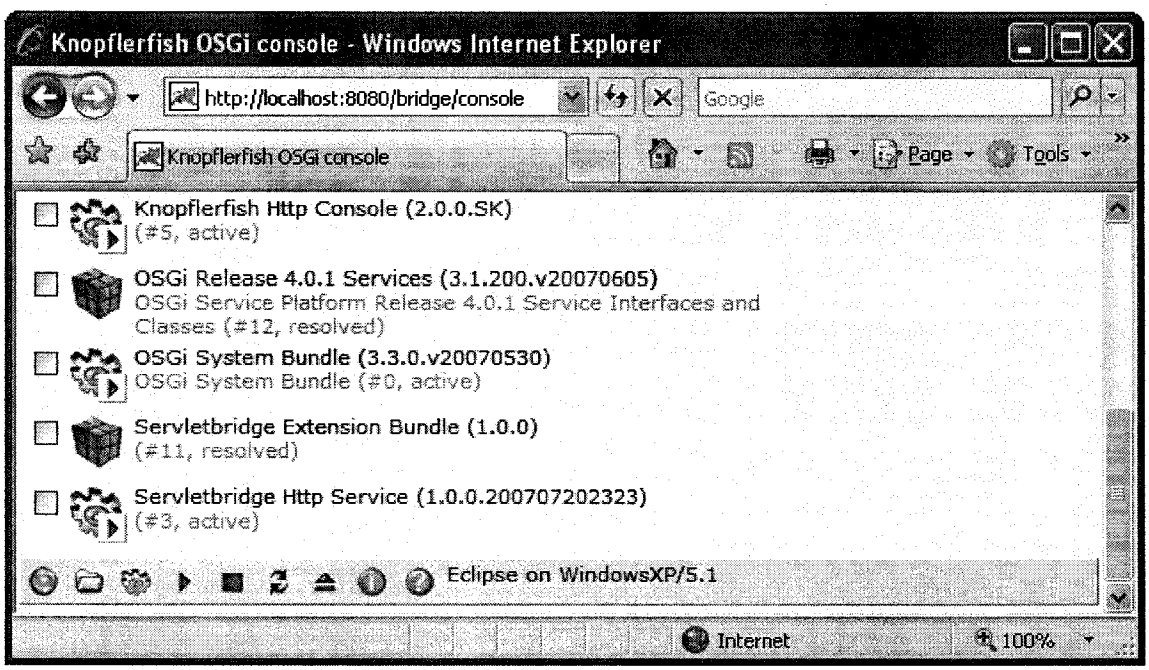


Figure 7-1: Knopflerfish Http Console

The console lists the bundles installed in the OSGi framework along with their current state. In this case the three bundles represented with a "cog" are started. The console allows selecting bundles and then performing operation like starting, stopping, uninstalling. In addition the console permits installation of new bundles, functionality we use in the next section.

7.2 Embedding OSGi in a Servlet Container

With the WAR file prepared from the previous setup steps we deploy the web application and launch Tomcat:

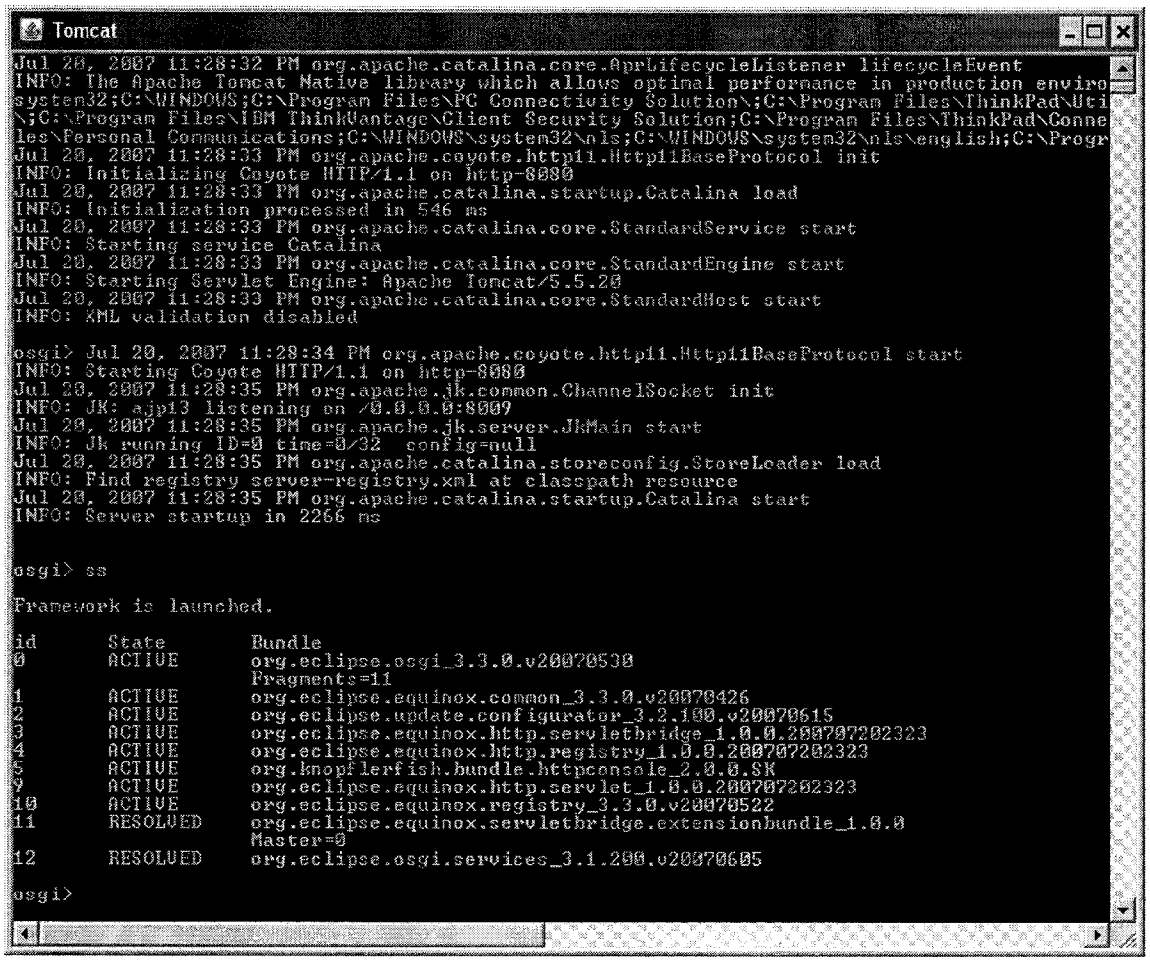


Figure 7-2: Launching the Servletbridge

Although it might not look like much, we've just successfully embedded and started an OSGi framework inside of a servlet container. The "osgi>" command prompt seen is Equinox's command line management console. In this particular case we've executed "ss" (short status) to display the list of bundles installed along with their current state.

This example is purely illustrative as in a production environment you would generally disable this console unless trying to debug a problem. By default this console

runs off of the process' standard input/output however it can also be configured to listen on a telnet port. Using telnet is more common when running against a full Java EE server especially when local access is not readily available.

7.3 Dynamic Installation and Uninstallation

One of the key benefits of OSGi is that it provides support for dynamically modifying the software running in a system. To illustrate how this applies in our solution we show how a bundle containing a simple servlet can be installed and later uninstalled at runtime. We also show that our Http Service implementation is able to react to the addition and removal of resources to its URI space.

7.3.1 Installation

To demonstrate we first install the test.servlet bundle by first providing the URL where it's located and then performing the install.

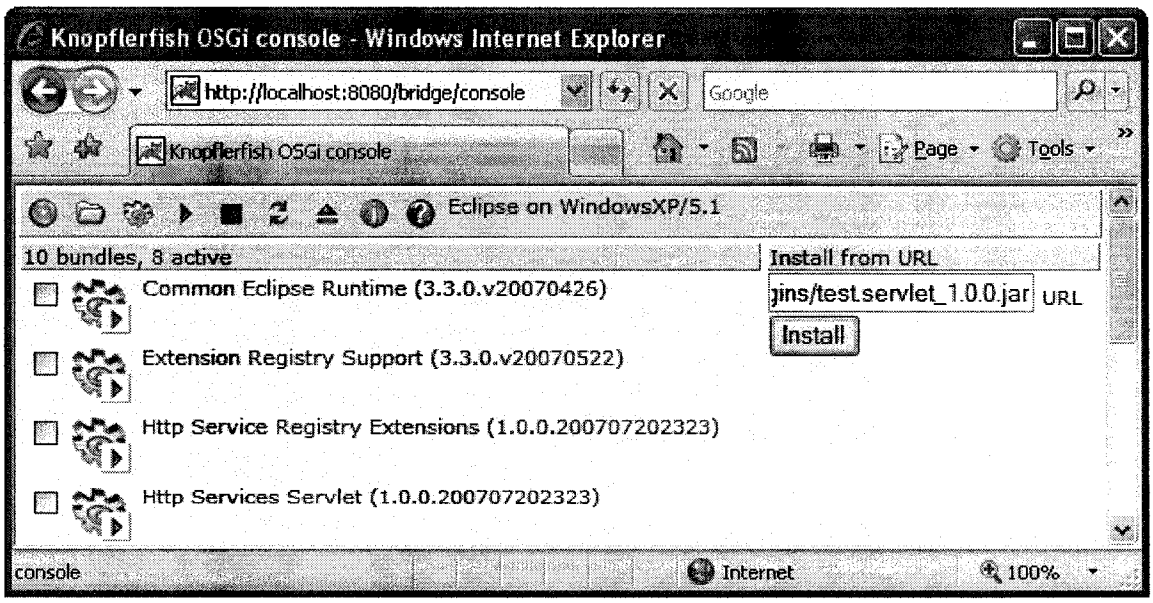


Figure 7-3: Installing a bundle

The result is that our test.servlet bundle is installed in the OSGi framework. The servlet content is made available only once the bundle has been activated. This is done by selecting the bundle's checkbox and using the "start" or "play" icon (fourth from the left).

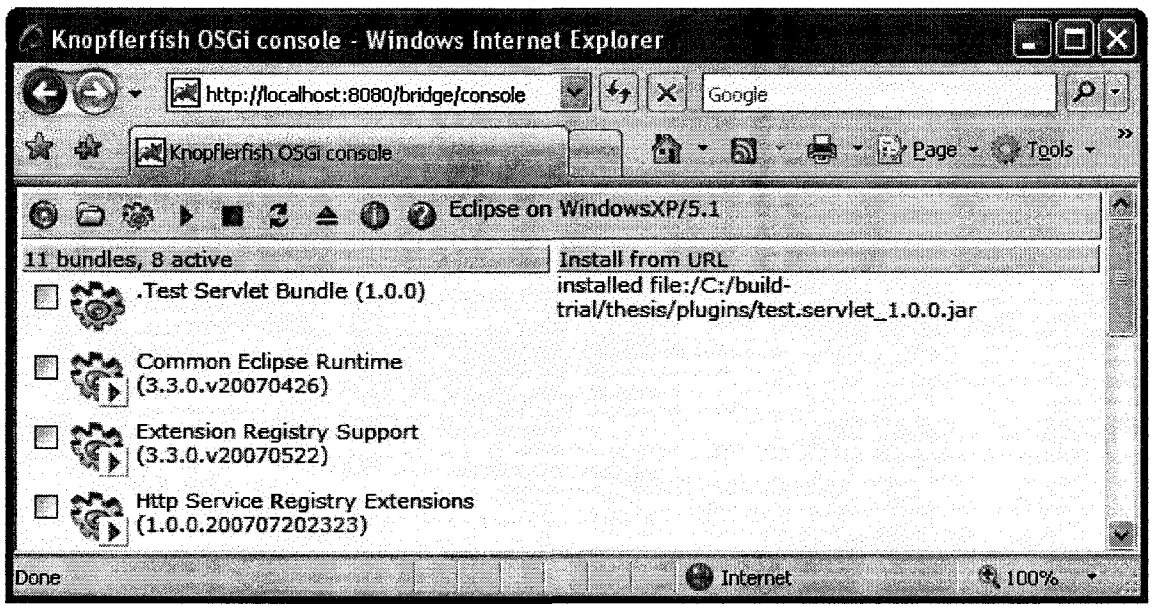


Figure 7-4: After installing a bundle

Our final step is to show our test servlet's simple web content. What is more relevant is that the web content had been dynamically added long after the web application was deployed.

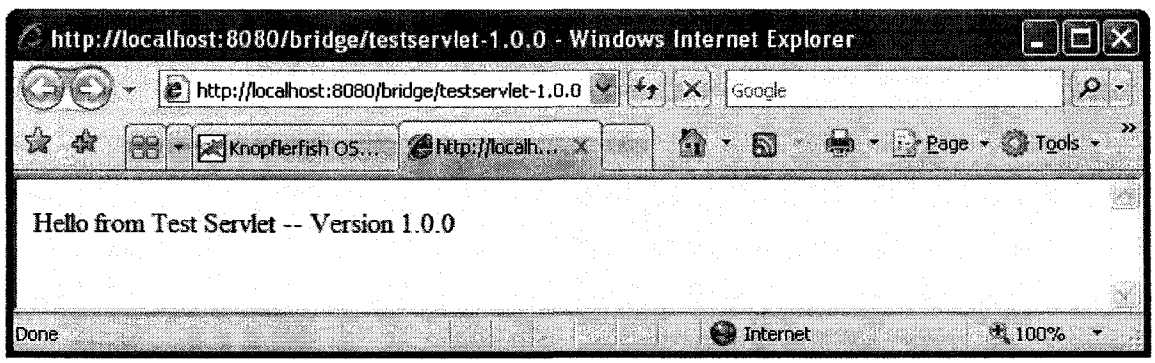


Figure 7-5: Dynamically added Test Servlet

7.3.2 Uninstallation

The uninstallation of the bundle is performed by selecting our Test Servlet Bundle and then using the "uninstall" or "eject" icon (seventh from the left). The result is that the bundle count has been reduced from eleven to ten, and our Test Servlet Bundle is uninstalled.

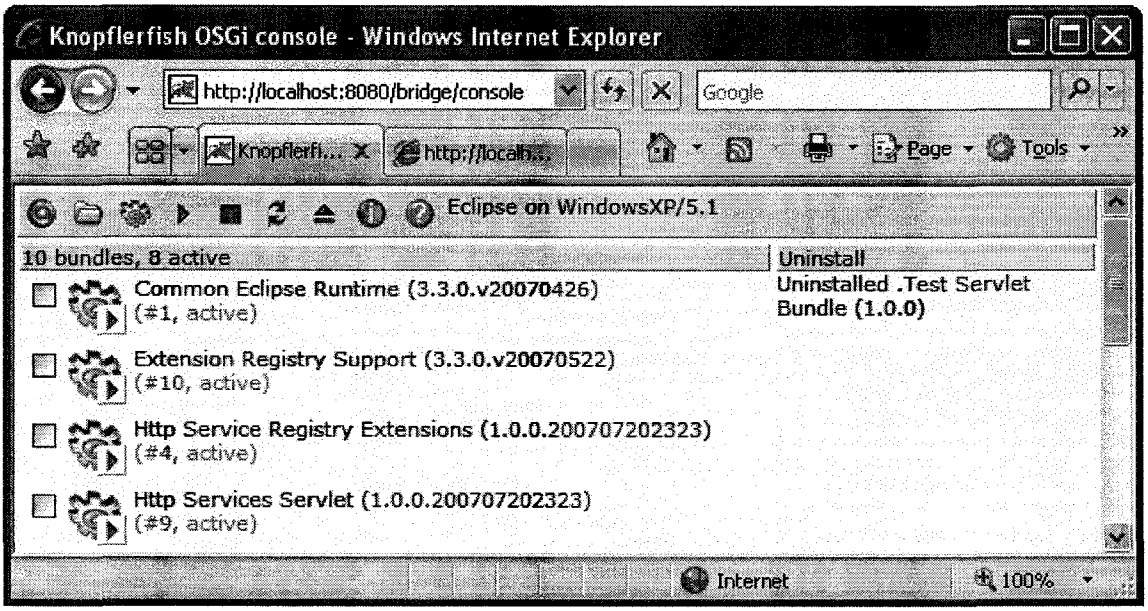


Figure 7-6: Uninstalling a bundle

Verifying that our web content has also been removed we get a 404 the HTTP Status code used when there is no resource present at the URL.

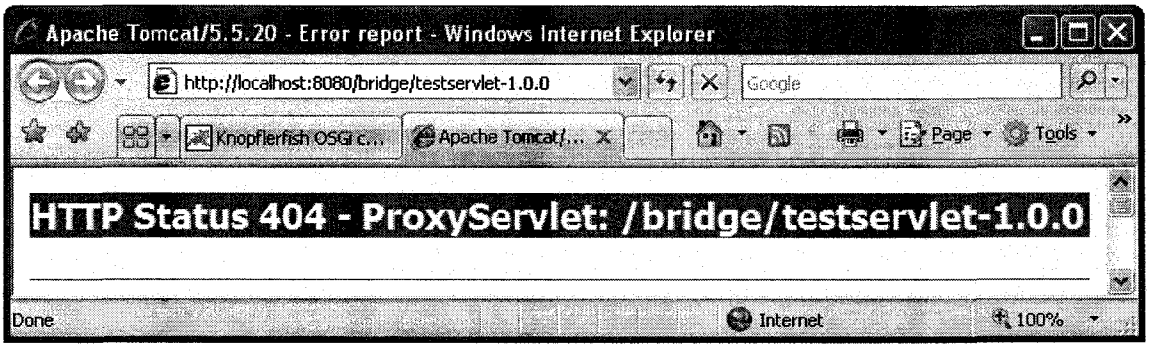


Figure 7-7: Verifying removal of web content

7.4 Multiple Version Support

OSGi provides strong isolation between the classes contained in bundles and allows the use of metadata to provide versioning information. In this section we demonstrate how this capability can be used to support web components that have dependencies on different versions of a particular library.

To illustrate this we augment our test.servlet to depend on classes from another bundle called test.library. To demonstrate support for multiple versions we create two versioned test.servlet and test.library pairs of bundles such that:

- test.servlet (1.0.0) depends on test.library (1.0.0)
- test.servlet (2.0.0) depends on test.library (2.0.0)

To provide visual feedback the test.servlet has been further altered to use test.library to generate part of the web content and indicate versioning information.

7.4.1 Scenario

To start the scenario we install both versions of the bundles, and as before we will also "start" the bundles.

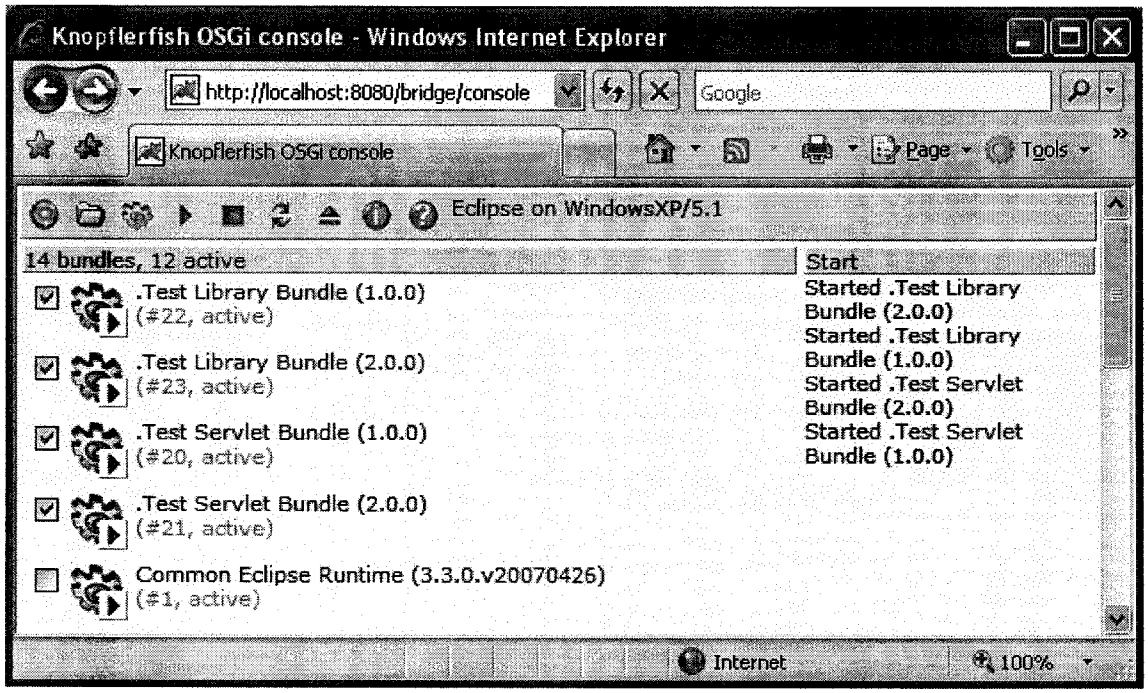


Figure 7-8: Supporting multiple versions of bundles

Figure 7-8 shows that all four bundles have resolved and started correctly. The two servlet bundles register with the Http Service in their Activators, this despite having different dependencies. In an OSGi environment they can share the Http Service interface and other common resources, and only their test.library dependency is wired differently.

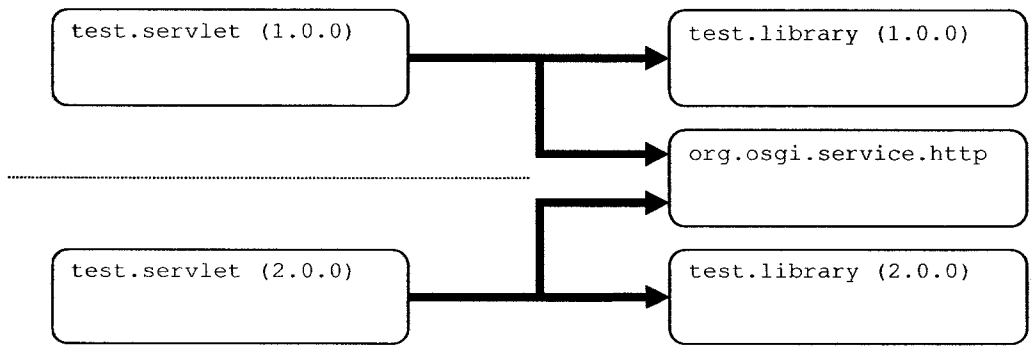


Figure 7-9: Dependencies for two versions of Test Servlet

7.4.2 Result

With our bundles installed and started we can verify that they work as expected, with the test.servlet and test.library bundle pairs wired together independent of one another.

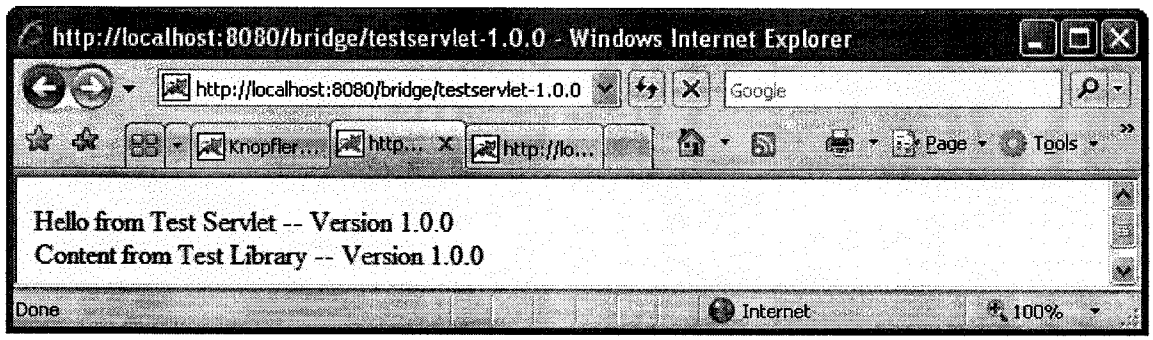


Figure 7-10: Test Servlet and Library 1.0.0

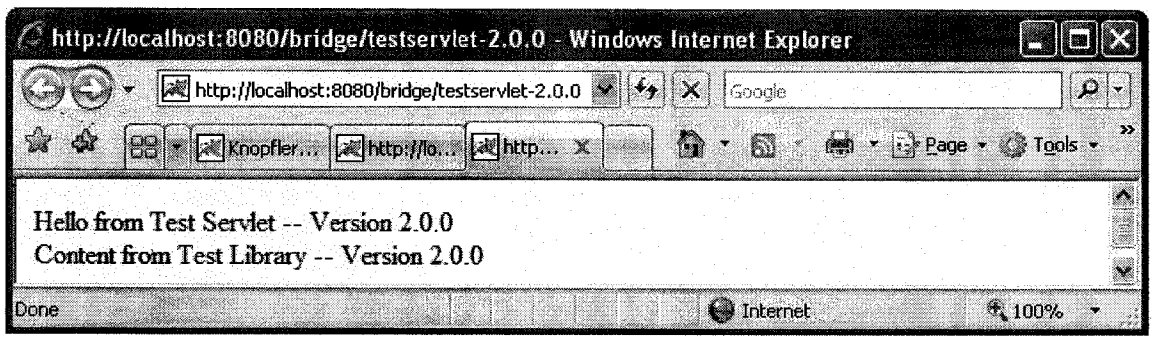


Figure 7-11: Test Servlet and Library 2.0.0

From figure 7-10 and 7-11 we can see that our two versions of test.servlet were indeed wired to their like versioned test.content pair. Although this scenario was very simple, the implications are significant and mean that we can for example support multiple versions of tag libraries or even web UI frameworks in the same web application.

7.5 Validating Application Server and Virtual Machine Support

A goal of this thesis was to establish the feasibility of this framework. As part of the validation we have tested the full suite of Http Service and JSP support bundles successfully on the following application server platforms:

- Apache Tomcat (4.1, 5.5)

- BEA Weblogic (8.1, 9.2)
- IBM Websphere (6.0, 6.1)
- JBoss AS (4.0)
- Mortbay Jetty (5.1, 6.0, 6.1)
- Oracle OC4J 10g (10.1.3)

There were very few integration problems or significant differences between application servers for our work. It is likely that the Servletbridge will be portable with little if any effort with any Servlet Container providing Servlet 2.4 or 2.5 API support.

In all cases we performed this validation with either Sun Microsystems JRE version 5.0 or where applicable the Java runtime provided with the application server. Apache Tomcat 4.1 was also validated with Sun JRE 1.4. This implies our integration approach has been validated with the following Java VMs and associated class libraries:

- BEA JRockit (1.4, 5.0)
- IBM J9 (1.4, 5.0)
- Sun Microsystems JRE (1.4, 5.0)

Initial testing with Sun's JRE 6.0 shows no incompatibilities and there is little reason to believe there will be problems with the Java 6 release of J9 or JRockit. Though not formally validated our approach appears to be compatible at the code level with two other open source JVMs currently in development: GCJ, and Apache Harmony.

7.6 Validating JavaServer Pages Support

Many Java Web UI Frameworks leverage JavaServer Pages and in particular its tag library support. One of our contributions was to provide support for JSPs inside an OSGi framework and so to validate we performed integrations with the example suites provided by a few projects. In particular we chose:

- **Apache Tomcat's JSP Examples.** These examples are part of the Tomcat distribution and are used to demonstrate the usage of both the JSP 1.2 and 2.0 API. For our purposes it provided good coverage of the JSP API.[2]
- **Java Standard Tag Library (JSTL) Examples.** Developed as part of the standardization process at the JCP this set of examples provided excellent tag library coverage.[4, 49]
- **Struts Example.** "Struts" is perhaps the best know Java Web UI framework and exercises many elements of the Servlet API, JSP API and tag library support. The Struts examples are a suite that is packaged by the Struts team to illustrate how the framework might be used. For our purposes this suite was helpful in validating Struts support but is also provided a good overall test for many different aspects of our solution.[3]

7.7 Use in Commercial and Open Source Products

The work on the Servletbridge was still being "incubated" inside the Equinox project during the Eclipse 3.2 time frame. Despite this status other teams inside Eclipse started incorporating it in their products and supporting its refinement [17]. With the eventual

graduation of the work [12, 35] in Eclipse 3.3 there has been increasing interest at Eclipse and more recently use in commercial products.

7.7.1 Eclipse BIRT (Business Intelligence and Reporting Tool)

BIRT [15] is a project at Eclipse that provides a server-based solution for reporting and analysis. The reporting server components run inside a Java EE server as a web application. BIRT makes use of a modified version of the Servletbridge's framework launcher as well as the Equinox / Application Server integration. Actuate, the project's largest sponsor, sells an enhanced commercial version of BIRT as well as commercial support.

7.7.2 Eclipse User Assistance / Help

During Eclipse 3.3, the Eclipse SDK's User Assistance and Help components switched to using the server-side components from Equinox. In particular the Http Service implementation and JSP Support is being used. Proof of concept work has been done to demonstrate integration with the Servletbridge and is likely to be available in future releases of the platform. This will fill a gap and allow Eclipse Help applications to be hosted from existing application servers.

IBM's Infocenter is the web technology help system used by all IBM products. Infocenter leverages the work done in Eclipse User Assistance / Help so stands to benefit from the integration.

7.7.3 IBM Rational Jazz

In June 2006 IBM Rational announced the development of Jazz [26], a platform to better enable team development scenarios. The platform is built on top of Eclipse and its

component model so is subsequently built around OSGi. Jazz consists of both client and server elements, and what's significant is that the same component model is used on both sides. The Jazz Server infrastructure leverages the Servletbridge and Equinox / application server integration to provide this support.

7.8 Summary

In this chapter we've presented an evaluation of our work. and discussed how others are using it. We've demonstrated integration and launching of an OSGi framework inside of a servlet container, support for dynamically adding and removing components at runtime, as well as our support for multiple versions of a component. Our solution has been tested on a wide variety of application servers and Java runtimes. We've shown how we tested our JSP support. Finally we've given three examples where the Servletbridge and its related components are being used in other open source and commercial products.

Chapter 8: Conclusion

In this chapter we first reexamine our motivations and confirm that we have satisfied our goals. We then enumerate the contributions of this thesis and end by discussing areas for future work.

8.1 Confirmation of Approach

The primary goal of this thesis is to provide an approach for building web applications in Java where the individual components can be dynamically added, reconfigured, or removed. In this system, each component is an independent module tied to other components only to satisfy explicit dependencies. The web application can be composed from these pieces with confidence that they won't interfere with one another except where defined to.

Recognizing that for the foreseeable future Java EE servers will likely be the infrastructure made available to run these web applications we propose integrating whatever approach we come up with into this environment. That approach is OSGi and represents the state of the art when it comes to the use of Java class loaders for providing modularity support.

The integration of OSGi with a Java EE environment is challenging as both technologies want to take control of the Java runtime. Using the Equinox implementation we have shown how an OSGi framework can be altered to allow it to run in this

environment while still maintaining the strong isolation guarantees necessary for modularity.

Our solution, the Servletbridge, creates a communication channel restricted to the Servlet API that allows OSGi based web components to handle the requests. Unfortunately OSGi support for servlet applications is not as mature as what's provided by modern servlet engines. Where prudent we have provided updated support to the OSGi Http Service for both the Servlet API and JavaServer Pages.

To validate our approach we have shown how Equinox can be launched from a web application running inside the Tomcat servlet engine. We then demonstrated the addition and removal of web components at runtime as well as the support OSGi provides for running multiple versions of the same component.

To further re-assure our approach is sound we provide the set of Java EE application server environments and Java runtimes where we have confirmed correct behavior. We then present a set of existing example applications we have integrated with to validate our coverage of the Servlet API and JavaServer Pages technology. Finally we offer a few examples where other open source projects and commercial products are using our solution successfully.

8.2 Contributions

Our contributions are as follow:

- The Servletbridge, an approach that allows the use of OSGi to build web applications with existing Java EE servers.

- Modifications to the Eclipse Equinox OSGi implementation to provide support for embedding inside Java EE environments.
- An implementation of the OSGi Http Service designed to work as part of the Servletbridge approach.
- Improvements to the OSGi Http Service to allow support for more recent versions of the Servlet API.
- Integration of the Jasper JSP Engine in an OSGi environment and support for its use with the Http Service.

8.3 Future Work

Being able to embed an OSGi framework inside of a Java EE environment opens up a number of interesting avenues for future work. Our approach looked at providing support for OSGi server applications that use the Servlet API and JavaServer Pages. There are many other Java EE APIs that could also be considered. Examining how Enterprise Java Beans (EJB), Java Messaging Service (JMS), and Java Naming and Directory Interface (JNDI) might be made to better integrate with an OSGi environment look like particularly interesting avenues of investigation.

The OSGi Http Service provides very basic support for the Servlet API. Our work improved the situation but still left a portion of the Servlet API unsupported. Providing more complete support for the ServletContext as well as Filters and the various listeners would be an incremental but valuable contribution.

Finally, the technique we used to multiplex VM singletons required by the OSGi URL Handler Service may be useful with other singletons. Multiplexing the Security Manager

would be of significant value as this would allow for a fully compliant implementation of the Conditional Permission Admin specification when running in our environment.

Appendix A: BundleProxyClassLoader Recipe

In an OSGi framework the Bundle object provides methods to load classes and get resources similar to a class loader, however it not an instance of ClassLoader.

In this thesis the recipe is used when constructing a class loader to pass to the Jasper JavaServer Pages engine. There are many third party libraries that make use of class loaders to dynamically load resources. Many pre-date the use of OSGi and for integration in those situations the BundleProxyClassLoader recipe is particularly helpful.

```
import java.io.IOException;
import java.net.URL;
import java.util.Enumeration;

import org.osgi.framework.Bundle;

public class BundleProxyClassLoader extends ClassLoader {
	private Bundle bundle;
	private ClassLoader parent;

	public BundleProxyClassLoader(Bundle bundle) {
		this.bundle = bundle;
	}

	public BundleProxyClassLoader(Bundle bundle, ClassLoader parent) {
		super(parent);
		this.parent = parent;
	    this.bundle = bundle;
	}

	// Note: Both ClassLoader.getResources(...) and bundle.getResources(...)
consult
	// the boot classloader. As a result,
BundleProxyClassLoader.getResources(...)
	// might return duplicate results from the boot classloader. Prior to
Java 5
	// Classloader.getResources was marked final. If your target environment
requires
	// at least Java 5 you can prevent the occurence of duplicate boot
classloader
```

```
        // resources by overriding ClassLoader.getResources(...) instead of
        // ClassLoader.findResources(...).
        public Enumeration findResources(String name) throws IOException {
            return bundle.getResources(name);
        }

        public URL findResource(String name) {
            return bundle.getResource(name);
        }

        public Class findClass(String name) throws ClassNotFoundException {
            return bundle.loadClass(name);
        }

        public URL getResource(String name) {
              return (parent == null) ? findResource(name) :
super.getResource(name);
        }

        protected Class loadClass(String name, boolean resolve) throws
ClassNotFoundException {
              Class clazz = (parent == null) ? findClass(name) :
super.loadClass(name, false);
              if (resolve)
                    super.resolveClass(clazz);

              return clazz;
        }
}
```

Appendix B: Runtime Overhead

Although not a goal of this thesis we wanted to establish an overhead baseline as this could conceivably affect the viability of our solution.

Memory and Disk Space

OSGi technology has targeted the embedded hardware and mobile market and as a result the frameworks have tended to be small and efficient. With that said, there is some space overhead associated with loading and running under an OSGi framework. To measure we compared the running of a trivial "Hello world" type program with and without OSGi framework. This approach was only meant to establish basic numbers.

- Disk Space: 800K
- Memory: 3900K
    - Heap: 800K
    - Code Cache: 500K
    - Perm Gen: 2600K

In particular, the values for increased memory is likely larger than it likely would be in a real server application. A good proportion of the classes loaded into the "Code Cache" and "Perm Gen" comes from the JRE and in larger applications these classes would be

required anyway. Nonetheless, even an overhead of 4MB is acceptable in an environment where it's not unusual to have 4GB or more memory.

Performance

The Servletbridge introduces some performance overhead in that all requests are funneled through it. The following list provides a rough guide of where additional overhead is involved for each request:

- setting and unsetting the context class loader
- filtering the ServletRequest to adjust path information
- mapping the request to an appropriate handler

Although not rigorously measured in running basic tests with a simple "Hello World" Servlet there was no appreciable difference in terms of response time. This result is not surprising given that there is already very significant overhead in the Servlet containers to handle request servicing and dispatching. Despite this overhead in the container most non-static applications will spend the vast majority of time in the web application logic.

References

[1] Apache Felix. Apache Foundation. http://felix.apache.org

[2] Apache Tomcat. Apache Foundation. http://tomcat.apache.org

[3] Apache Struts. Apache Foundation. http://struts.apache.org

[4] Apache Jakarta Taglibs. Apache Foundation. http://jakarta.apache.org/taglibs/index.html

[5] C. Baldwin, K. Clark. *Design Rules: Volume 1. The Power of Modularity*. MIT Press, 2000.

[6] BEA. BEA microService Architecture. http://www.bea.com/framework.jsp?CNT=msa.jsp&FP=/content/, 2007.

[7] M. Buechi. "Eclipse Plugin-Based Applications and J2EE Components". ChoiceMaker Technologies whitepaper. 2003.

[8] B. Christudas. "Internals of Java Class Loading". http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html, 2005.

[9] J. Corwin, D. Bacon, D. Grove, C, Murthy. "MJ: A Rational Module System for Java and its Applications". OOPSLA, 2003.

[10] J. Driscoll. "Servlet History". http://weblogs.java.net/blog/driscoll/archive/2005/12/servlet_history_1.html, 2005.

[11] Eclipse Equinox OSGi Framework. Eclipse Foundation. http://www.eclipse.org/equinox

[12] Eclipse Equinox Server Work Area. Eclipse Foundation. http://www.eclipse.org/equinox/server

[13] Eclipse.org. Eclipse Foundation. http://www.eclipse.org

[14] Eclipse Rich Client Platform. Eclipse Foundation. http://www.eclipse.org/rcp

[15] Eclipse BIRT. Eclipse Foundation. http://www.eclipse.org/birt

[16] M. Fleury, F. Reverbel. "The JBoss Extensible Server". *Proceedings of the International Middleware Conference: 344-373*, 2003.

[17] W. Gehner. "Server-side Eclipse". http://www.infonoia.com/en/server_side_eclipse.jsp, 2005.

[18] J. Gosling, B. Joy, G. Steele, G. Bracha. *The Java Language Specification, Third Edition*. Prentice Hall, 2005.

[19] O. Gruber, R. Hall. "A Java Framework for Building and Integrating Runtime Module Systems". OTM Conferences 2006.

[20] O. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, T. Watson. "The Eclipse 3.0 platform: Adopting OSGi technology". *IBM Systems Journal 44(2): 289-300*, 2005.

[21] R. Hall. "A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks". *Lecture Notes in Computer Science Volume 3083: 81-96*. Springer, Berlin 2004.

[22] R. Hall. "OSGi R4 Service Platform: Java Modularity and Beyond". akquinet fws, Berlin 2007.

[23] S. Halloway. *Component Development for the Java Platform*. Addison Wesley. 2001.

[24] IBM. "IBM WebSphere Application Server v6.1 Componentization Overview". http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp?topic=/com.ibm.iea.was_v6/was/6.1/Architecture/WASv61_Componentization/player.html, 2006.

[25] Interface 21. "Spring-OSGi". http://www.springframework.org/osgi, 2006.

[26] Java Community Process. http://jcp.org

[27] Jazz.com. IBM Rational. http://www.jazz.com

[28] JBoss. "JBoss OSGi integration project". http://jira.jboss.com/jira/browse/JBOSGI, 2007

[29] S. Kaegi. "Equinox Incubator - Server-side Proposal". http://www.eclipse.org/equinox/server/proposal.php, 2005.

[30] Knopflerfish OSGi Framework. Knopflerfish. http://www.knopflerfish.org

[31] Knopflerfish Http Console. Knopflerfish. https://www.knopflerfish.org/svn/knopflerfish.org/trunk/osgi/bundles/http/httpconsole/readme.html, 2004.

[32] P. Kriens. "Spring and OSGi: Jumping Beans". *OSGi Blog* http://www.osgi.org/blog/2007/01/spring-and-osgi-jumping-beans.html, 2007.

[33] S. Liang, G. Bracha. "Dynamic Class Loading in the Java Virtual Machine". In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications: 36-44*, 1998.

[34] T. Lindholm, F. Yellin. *The Java Virtual Machine Specification, Second Edition.* Prentice Hall, 1999.

[35] J. McAffer, S. Kaegi. "Eclipse, Equinox and OSGi". *TheServerSide.com* http://www.theserverside.com/tt/articles/article.tss?l=EclipseEquinoxOSGi, 2007.

[36] J. McAffer, J. Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications.* Addison Wesley Professional, 2005.

[37] T. Neward. "Understanding Class.forName()". *JavaGeeks.com whitepaper*, 2001.

[38] OSGi Alliance. "About the OSGi Service Platform, Technical Whitepaper". http://www.osgi.org/documents/collateral/OSGiTechnicalWhitePaper.pdf, 2007.

[39] OSGi Alliance. Dynamic Component Support for Java SE (JSR 291). Java Community Process, 2007.

[40] OSGi Alliance, OSGi Service Platform Core Specification Release 4, Version 4.1. http://www.osgi.org 2007.

[41] OSGi Alliance, OSGi Service Platform Service Compendium Release 4, Version4.1. http://www.osgi.org 2007.

[42] C. Perez. "Six Operators of Modularity". http://www.manageability.org/blog/stuff/six-operators-of-modularity, April 2004.

[43] B. Peterson, "Understanding J2EE Application Server Class Loading Architectures". *The ServerSide.com*, 2002.

[44] J. Ponzo, L. Hasson, J. George, G. Thomas, O. Gruber, R. Konuru, A. Purakayastha, R. Johnson, J. Colson, R. Pollak. "On demand Web-client technologies". *IBM Systems Journal 43(2): 297-315*, 2004.

[45] J. Ponzo, O. Gruber. "Integrating Web technologies in Eclipse". *IBM Systems Journal 44(2): 279-288*, 2005.

[46] J. des Rivieres, J. Wiegand. "Eclipse: A platform for integrating development tools". *IBM Systems Journal 43(2): 371-383*, 2004.

[47] V. Roubtsov. "Find a way out of the ClassLoader maze". http://www.javaworld.com/javaworld/javaqa/2003-06/01-qa-0606-load.html, 2003.

[48] A. Schaefer. "Inside Class Loaders". http://www.onjava.com/pub/a/onjava/2003/11/12/classloader.html, 2003.

[49] Sun Microsystems. A Standard Tag Library for JavaServer Pages (JSR 52). Java Community Process, 2003.

[50] Sun Microsystems. Improved Modularity Support in the Java Programming Language (JSR 294). Java Community Process, 2006.

[51] Sun Microsystems. Java 2 Standard Edition Extension Mechanism Architecture. http://java.sun.com/j2se/1.4.2/docs/guide/extensions/spec.html, 1999.

[52] Sun Microsystems. Java 2 Standard Edition JAR File Specification. http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html, 1999.

[53] Sun Microsystems. Java API for XML Processing (JAXP) 1.3. (JSR 206). Java Community Process. 2006.

[54] Sun Microsystems. Java Enterprise Edition. http://java.sun.com/javaee/technologies/

[55] Sun Microsystems. Java Module System (JSR 277). Java Community Process, 2006.

[56] Sun Microsystems. JSR-277: Java Module System, Early Draft. Java Community Process, 2006.

[57] Sun Microsystems. Java Naming and Directory Interface (JNDI). http://java.sun.com/products/jndi/, 2003.

[58] Sun Microsystems. JavaServer Pages 2.0 Specification (JSR 152). Java Community Process 2003.

[59] Sun Microsystems. Java Servlet 2.1 Specification. http://java.sun.com/products/servlet/2.1/, 1998.

[60] Sun Microsystems. Java Servlet 2.5 Specification. (JSR 154). Java Community Process. 2006.

[61] M. Warres. "Class Loading Issues in Java RMI and Jini Network Technology". *SMLI TR-2006-149*. Sun Microsystems, Sun Labs. 2006.

[62] Wikipedia. Java Classloader. http://en.wikipedia.org/wiki/Classloader, 2007.