Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Master Thesis

Arnold Kemoli

Design and Implementation of a Dynamic Component based Web Application Framework

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Arnold Kemoli

Design and Implementation of a Dynamic Component based Web Application Framework

Master thesis based on the examination and study regulations for the
Master of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Supervising examiner : Prof. Dr. Hans-Jürgen Hotop
Second examiner : Prof. Dr. Henning Dierks

Arnold Kemoli

Title of the Master Thesis

Design and Implementation of a Dynamic Component based Web Application Framework

Keywords

OSGi, JSF, Facelets, Jetty, Equinox, Modular Web Applications, Spring DM

Abstract

This thesis describes the development of a web based dynamic modular application framework for hosting JSF based web applications. The framework makes it possible to add or remove functional modules from an application during runtime without having to temporarily shut down the framework.

End of text

Arnold Kemoli

Thema der Masterarbeit

Design und Implementierung eines dynamischen komponentenbasierten Web Application Frameworks.

Stichworte

OSGi, JSF, Facelets, Jetty, Equinox, Modular Web Applications, Spring DM

Kurzzusammenfassung

Diese Arbeit beschreibt die Entwicklung eines web-basierten dynamischen modularen Application Frameworks für JSF-basierte Web Anwendungen. Das Framework ermöglicht das dynamische Hinzufügen und Entfernen von Funktionsmodulen einer Application zur Laufzeit, ohne dass das Framework dazu heruntergefahren werden muss.

Ende des Textes

Table of Contents

Abbreviations 1

1. Introduction 2

1.1. Objectives 3

1.2. Thesis Outline 3

2. *Clintweb* Framework Overview 5

2.1. Web container (Tomcat layer) 5

2.2. *JSF* Layer 6

2.3. *Facelets* Layer 8

2.4. Applications Layer 9

2.5. Challenges in *Clintweb* 10

3. Modular Web Applications 11

3.1. Modular Application Design Principles 11

3.2. Introduction to *OSGi* 13

3.2.1. Why choose *OSGi*? 16

3.2.2. *OSGi* Web Implementations 18

3.3. Web Application Deployment Topologies 24

3.3.1. Web container in *OSGi* 25

3.3.2. *OSGi* in Web container 28

3.4. Session State Management 30

3.5. Summary 31

4. Requirements Analysis 34

4.1. Framework Requirements 34

4.2. User Requirements 38

4.3. Administrator Requirements 39

5. System Design 42

5.1. Framework Architecture 43

5.2. Application Architecture 43

5.3. User Requests Processing 46

5.4. Resolving *Facelet* Resources 53

5.5. Application Session Listeners 54

5.6. *JSF* Functionality 55

5.7. Session State Preservation 60

5.8. Client and Administrator Test Application Design 65
5.9. Design Constraints 66
6. System Implementation 68
6.1. Development Tools 68
6.2. Framework and Application Architecture 69
6.3. Session State Management 86
7. Testing and Evaluation 91
7.1. Framework Functionality Tests 91
7.2. Performance Tests 97
7.3. Summary 100
8. Conclusion 102
8.1. Recommendations 103
8.2. Outlook 104
9. References 106
10. Appendix 109

List of Figures

Figure 2.1: Clintweb framework……5

Figure 2.2: Navigation Rules example……7

Figure 3.1: Module size……12

Figure 3.2: Module ability to reuse……13

Figure 3.3: OSGi architecture……14

Figure 3.4: Bundle lifecycle……15

Figure 3.5: Tightly coupled JARs……16

Figure 3.6: Eclipse architecture……18

Figure 3.7: Plug-ins in Eclipse……20

Figure 3.8: Extension point processing……21

Figure 3.9: Spring DMK topology……26

Figure 3.10: Jetty web server……28

Figure 3.11: Equinox in Tomcat……29

Figure 4.1: Resource navigation……36

Figure 4.2: Administrator use-case diagram……39

Figure 4.3: Module swapping sequence diagram……40

Figure 5.1: System overview……42

Figure 5.2: Framework architecture……43

Figure 5.3: Application required bundles……44

Figure 5.4: Jetty's functional parts……44

Figure 5.5: New bundle format……45

Figure 5.6: User request forwarding sequence diagram (bundle perspective)……46

Figure 5.7: Servlet extension point……47

Figure 5.8: Application Servlet classes……48

Figure 5.9: HttpContext extension point……50

Figure 5.10: HttpContext class diagram……50

Figure 5.11: User request forwarding class diagram……52

Figure 5.12: Resource resolver extension point……53

Figure 5.13: Resource locator interface……54

Figure 5.14: Session listeners class diagram……55

Figure 5.15: Manage bean processing class diagram……56

Figure 5.16: Manage beans extension point……57

Figure 5.17: Loading Managed Beans class diagram……57

Figure 5.18: Navigation Rules extension point……58

Figure 5.19: Tag library classes……59

Figure 5.20: Object serialization sequence diagram……61

Figure 5.21: Serialization class diagram……62

Figure 5.22: Managed Bean restoration sequence diagram……63

Figure 5.23: De-serializer classes……64

Figure 5.24: Test application UI layout……65

Figure 5.25: Administrator application layout……65

Figure 5.26: Cyclic dependencies……66

Figure 6.1: Application architecture……70

Figure 6.2: FacesServletAdapter registration……71

Figure 6.3: ResourceHttpServlet registration……72

Figure 6.4: FacesHttpContext registration……73

Figure 6.5: FacesHttpContext registration……73

Figure 6.6: Loading tag libraries……74

Figure 6.7: Namespaces for UI components……75

Figure 6.8: JSF Resource resolver declaration……75

Figure 6.9: Resolving facelets URL……76

Figure 6.10: Resource locator extension point……76

Figure 6.11: Resolving Managed Bean objects……79

Figure 6.12: Managed Bean extension point……80

Figure 6.13: Resolving custom components……83

Figure 6.14: Session class registration class diagram……84

Figure 6.15: Session listener extension poin……85

Figure 6.16: De-serializing Managed Beans……87

Figure 6.17: Resolving Managed Beans……..88

Figure 6.18: OSGiObjectInputStream resolveClass function……89

Figure 7.1: Application front page version 1……90

Figure 7.2: Storing values in Managed Beans……92

Figure 7.3: Custom component……93

Figure 7.4: Administrator application……94

Figure 7.5: Home page version 2……95

Figure 7.6: User input data page version 2……96

Figure 7.7: Configuration page……96

Figure 7.8: Machine specifications……98

Figure 7.9: Test case 1……98

Figure 7.10: Test case 2 line graph……99

Figure 7.11: Test case 3 line graph……100

Abbreviations

API	Application Programming Interface
AJAX	Asynchronous JavaScript
CODA	Component Oriented Development and Assembly
CSS	Cascading Style Sheets
e.g.	Example given
HTTP	Hyper Text Transfer Protocol
IDE	Integrated Development Environment
JAR	Java Archive
JRE	Java Runtime Environment
JSF	Java Server Faces
JSP	Java Server Pages
JVM	Java Virtual Machine
MVC	Model View Controller
OSGi	Open Source Gateway Initiative
UEL	Unified Expression Language
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WAR	Web Archive
WTP	Web Tools Platform
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language

1. Introduction

Nowadays web applications are essential tools for companies to easily reach customers and provide internal services to their workforce. Companies have opted to employ solutions for hosting large and complex web based applications to be able to provide services to users without having to install an application on every computer (web browsers are installed in all work stations). As a company based in the telecommunication market, *Clintworld GmbH* specializes in offering tariff management and optimization advice to mobile service providers. Users are able to access *Clintworld's* services via a framework called *Clintweb*. *Clintweb* is a Java web based framework which hosts a range of applications offering different kinds of services to users. Users just require a web browser and a network/internet connection to access these applications at any place and any time.

Clintweb has been successful in offering stable services to users. However the dynamic developments in the telecommunication industry have led to challenges affecting applications on *Clintweb*. The challenges in *Clintweb* come from the fact that users very often request the addition of features to applications. Such changes need to be in cooperated into applications as soon as they are required because they may be critical to requesting user. Changes may include creation of new web pages or the addition of extra text fields for inputting information on a certain web page. Changes may also come from the developer's side in the form of optimization of a process in an application. In such scenarios, *Clintweb's* architecture requires an application to be stopped and then changes can be applied. Thereafter the application can be redeployed. This means that for an undefined amount of time, an application will be offline and users will not have access to it. Since *Clintweb* is used by clients across the world, users can be online at all times of a day. It is therefore difficult to find a time slot where no user is online in order to carry out maintenance of an application. On the developer side, a whole application has to be re-packaged and redeployed only to accommodate the required changes. Developers have to put in vast amounts of time in this process and users lose access to an application when it is needed.

The above identified problems led to the search of solutions for allowing applications on *Clintweb* to be serviced whilst keeping them online and accessible to users. Therefore users will never have to be inconvenienced when maintenance is taking place, leading to a better user experience while using applications on *Clintweb*. The popular idea of solving such a problem is by distributing an application into different components, with each component representing some functionality. The main challenge lies in making an application fully dynamic, where changes can be made to it during runtime without losing user inputted information.

1.1. Objectives

The objective of this thesis is derived from the above mentioned problems affecting *Clintweb*. It is the aim of this thesis to research and develop a web application framework[1] which can support servicing of applications during runtime without affecting user experience. Instead of researching on methods which update single Java classes on the fly, this thesis will focus on methods in which a web application is split into different functional components which can be individually manipulated during runtime. Research will be done on solutions which support modular web application[2] architectures, where applications are deployed as a collection of dynamic modules. After identifying possible solutions, considerations will be made on which solution would be the most optimal for *Clintweb*. This decision will be based on the migration efforts from the current *Clintweb* framework to the new system and the scalability of the new system. Thereafter the new framework will be designed and created; the framework will demonstrate its support for *Clintweb's* important features[3]. Finally this thesis will discuss the benefits and drawbacks of the new system, and provide recommendations[4] on whether the current *Clintweb* framework should be migrated into the new framework.

The research done by this thesis will assist web developers in the deployment stages of web applications. It will lead to the elimination of application downtimes during the deployment and maintenance stages of an application. By splitting applications into modules, it will be easier for developers to isolate errors when they occur and quickly solve them.

1.2. Thesis Outline

This thesis starts by discussing the current *Clintweb* framework and its building blocks in Chapter 2. The challenges of the *Clintweb* framework are identified in order to emphasis the need for a framework which supports dynamic modular web applications. Thereafter, Chapter 3 presents an

[1] A Web Application Framework is a software that provides services to a web based application. Such services may include: session management, provide libraries for database access etc. See: http://docforge.com/wiki/Web_application_framework and http://en.wikipedia.org/wiki/Web_application_framework [Accessed June 2010]

[2] See chapter 3

[3] See chapter 2

[4] See section 8.1

insight on technologies which support the deployment of dynamic web based modular applications. A description of how these technologies function is presented and comparisons are drawn between them in order to identify the best technology for realizing the desired framework.

Chapter 4 lists the requirements of the framework to be developed and its applications. It further identifies the user and administrator requirements that have to be fulfilled by the new framework. For each requirement, a potential solution based on the information presented in chapter 3 is outlined. At the end of this chapter a solution for creating the new framework is selected.

Chapter 5 discusses the design of a web based modular framework which fulfills the requirements stated in chapter 4. These design principles aim to address the problems of the *Clintweb* framework that were stated in chapter 2. Chapter 6 will thereafter discuss how the design specifications in chapter 5 were implemented.

Chapter 7 will then present a test application aimed at demonstrating the capability of the new framework to support modular applications and fulfilling the requirements stated in chapter 4. This chapter will include screen shots of the test application demonstrating its functionality.

Finally chapter 8 states a brief summary of the objectives of the thesis and how they were accomplished. It further states the recommendations on how the new framework can be adopted by *Clintweb* and future work that can be done on the new framework.

2. *Clintweb* Framework Overview

This chapter discusses the building blocks of the *Clintweb* framework. It is necessary to discuss the features of *Clintweb* because they will be included in the design of the new framework which will be developed during this thesis. *Figure 2.1* illustrates the layers of the *Clintweb* framework:

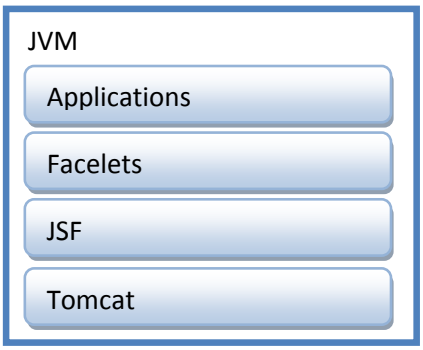


Figure 2.1: Clintweb framework

2.1. Web container (Tomcat layer)

Web container functionality is based on `Servlets` and JSP technologies. `Servlets` are Java classes which run in Java based web application servers[5] and are used for handling client requests. `Servlets` provide dynamic response to a client request and also manage state information[6] on top of the stateless HTTP. According to Java SUN specification[7], JSP is a technology which provides a simplified way for generating dynamic web content. JSP provides a means for Java code to be integrated into static web markup content, with the resulting page being compiled and executed on the server to deliver an HTML or XML document which can be viewed on a standard browser.

Tomcat is a `Servlet` container which is the official reference implementation of Java `Servlets` and Java Server Pages technologies[8]. Tomcat is composed of three parts (Catalina, Coyote and Jasper) which implement different functionalities. Catalina is the `Servlet` container which is an implementation of the SUN Microsystems specification for `Servlet` and JSP. It handles the

[5] An Application server is a software which executes procedures (programs, routines, scripts) for supporting the construction of applications. See: http://en.wikipedia.org/wiki/Application_server [Accessed 20 June 2010]
[6] Servlet and JSP, Available at: http://www.apl.jhu.edu/~hall/Java/Servlet-Tutorial/ [Accessed 21 June 2010]
[7] JSP Technology, Available at: http://Java.sun.com/products/jsp/ [Accessed 21 June 2010]
[8] Tomcat Overview by *Wellhouse* Consultant, Available at: http://www.wellho.net/downloads/A651.pdf [Accessed 21 June 2010]

management of user sessions and services client requests and responses. Coyote is the HTTP connector which listens for connections on a defined port on the server. It additionally receives HTTP requests and forwards them to the web server for processing. Jasper is the JSP engine which is responsible for compiling a JSP page into a `Servlet`.

Applications in Tomcat are deployed as WAR (Web Application Archive) files, which are files consisting of a packaged web application.

2.2. *JSF* Layer

On an abstract level, *JSF* is an implementation of the Model View Controller[9] (MVC) architecture. This architecture separates application concerns to UI, business logic and their connector which allows communication between the UI and business logic. *JSF* is a server-side UI component framework for Java based web applications. The framework provides the following main features[10]:

- JavaBean[11] management
- Standard UI components defined by *JSF* tag libraries
- Page navigation specification
- User Input validation
- Event handling

JSF framework provides the wiring of web applications using the above stated features. A typical *JSF* application contains a deployment descriptor (a *web.xml* file), which sets the properties[12] a *JSF* application will adopt during runtime. Furthermore, a *JSF* application contains a configuration file (*faces-config.xml*) which defines *Managed Beans*, *Navigation Rules*, Custom components and other application components, as explained below:

[9] MVC, Available at: http://www.oracle.com/technetwork/java/mvc-140477.html and http://en.wikipedia.org/wiki/Model_view_controller [Accessed 21 June 2010]
[10] *JSF KickStart*, Available at: http://www.exadel.com/tutorial/*JSF*/*JSF*tutorial-kickstart.html [Accessed 21 June 2010]
[11] JavaBean is a portable, platform-independent component model written in the Java programming language, see: http://download.oracle.com/javase/tutorial/javabeans/index.html [Accessed 21 June 2010]
[12] These properties include Java Session listeners (see section 5.5) and Servlets (see section 2.1)

2.2.1. *Managed Beans*

Managed Beans are JavaBeans which are be used by *JSF* applications to store the state of an application. They contain attributes and functions which are referenced by *JSF* components within a *JSF* page using Expression Language (EL) syntax. For a *Managed Bean* to be instantiated in *JSF*, the following information is required:

- *Managed Bean* name: The reference name assigned to a bean which *JSF* components will use to access a bean's attributes and call its functions.
- *Managed Bean* class: The class implementation of the *Managed Bean*.
- *Managed Bean* scope: The lifecycle span of the *Managed Bean*. This determines if a bean is valid during request or during the span of a session.

2.2.2. *Navigation Rules*

JSF supports the definition of navigation paths between pages within a web application. These navigation paths are called *Navigation Rules* in *JSF* and are defined in the *faces-config.xml* file. *Figure 2.2* shows an example of a *Navigation Rule*:

```
<navigation-rule>
    <from-view-id>/start.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>stop</from-outcome>
      <to-view-id>/stop.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Figure 2.2: Navigation Rules example

The `<from-view-id>` tag defines the source page URI[13] (the page where the link is prompted). The `<navigation-case>` defines the destination page of the *Navigation Rule* depending on the value in the `<from-outcome>` tag. The `<from-outcome>` defines a String value that prompts the *JSF* framework to navigate to the page declared in the `<to-view-id>` tag. Each *Navigation*

[13] URI is a string of characters which identify a resource, see: http://labs.apache.org/webarch/uri/rfc/rfc3986.html [Accessed 20 June 2010]

Rule can have more than one `<navigation-case>`, which navigate to various pages depending on the `<from-outcome>` and `<to-view-id>` attributes.

A *Navigation Rule* can be invoked as an output action of a component (for example, a button). Such an action will prompt *JSF* to navigate to a page defined in a *Navigation Rule*.

2.2.3. Custom Components

In *JSF*, a component is a group of interacting classes that provide a reusable web based UI code. A component is made up of three classes which co-operate together. The renderer class is responsible for creating a client-side representation of a UI component and it serves as an interface for receiving user input[14]. The renderer usually generates HTML code to represent the UI component and it transforms the values in the HTML form posts into values it can understand. The second class is the `UIComponent`[15] subclass. This class is controls the component behavior on the server side. The last class, known as the tag library class, declares how the custom component is referenced within a *JSF* page. The tag library class combines the reference name of the custom component with its UI component class and its renderer class.

2.3. *Facelets* Layer

Facelets is the view definition framework which is a page declaration language developed for *JSF*[16]. *Facelets* has become the standard presentation technology for *JSF*; previously *JSF* was only designed to work with JSPs. *Facelets* was adopted by *JSF* because JSP had limitations whereas it was not able to support new features provided in the recent versions of *JSF*. *Facelets* provides a language for constructing *JSF* views using XHTML files. A view is the object tree in memory,

[14] *JSF* custom components, Available at: http://today.Java.net/pub/a/today/2004/07/16/*JSF*custom.html [Accessed 20 June 2010]
[15] UIComponent specification, Available at: http://java.sun.com/javaee/javaserverfaces/1.2/docs/api/javax/faces/component/UIComponent.html [Accessed 16 August 2010]
[16] Oracle *Facelets,* Available at: http://download.oracle.com/docs/cd/E17410_01/Javaee/6/tutorial/doc/gijtu.html [Accessed 20 June 2010]

created from parsing an XHTML page. In order to apply *Facelets* to *JSF*, a `ViewHandler`[17] has to be defined. A `ViewHandler` is a *JSF* plug-in that handles the *Render Response* and *Restore View* phases of the *JSF* request-processing life cycle[18]. A page that is rendered by the `ViewHandler` is referred to as a 'view'. A view is associated with a view ID, which is a unique identifier referring to an XHTML page.

Facelets can additionally process Unified Expression Language (UEL) statements, which are used to reference *JSF Managed Beans* properties.

Note that from chapter 3 onwards, the term *'facelets'* will refer to XHTML pages containing *JSF* components and '*JSF-Facelets'* will refer to the libraries providing the *facelets* functionalities.

2.4. Applications Layer

This layer refers to applications which are deployable on the entire framework. *Clintweb's* applications utilize Tomcat as their web container and *JSF* combined with *Facelets* framework to provide functionality to users.

The applications are composed of small sized libraries (JARs[19]) where each library implements unique functionality. For example, a single library may be responsible for handling user management related processes. In order to support *JSF* and *Facelets* functionality, each library contains a *JSF faces-config.xml*[20] file which declares *Navigation Rules*, *Managed Beans* and Custom Components contained within them. For deployment, all libraries associated with an application are packed into a WAR[21] file and deployed in Tomcat web container[22] as a web application.

[17] ViewHandler specification, Available at: http://download.oracle.com/docs/cd/E17824_01/dsc_docs/docs/jscreator/apis/jsf/javax/faces/application/ViewHandler.html [Accessed 20 July 2010]
[18] IBM, *Facelets fits like a glove*: Available at: http://www.ibm.com/developerworks/java/library/j-facelets/ [Accessed 20 June 2010]
[19] A JAR file is a file format which packs multiple files into a single archive file. Typically a JAR contains the class files and auxiliary resources associated with java applications. See: http://java.sun.com/developer/Books/javaprogramming/JAR/basics/ [Accessed 20 June 2010]
[20] See section 2.2
[21] See section 2.1
[22] See section 2.1

2.5. Challenges in *Clintweb*

Applications in *Clintweb* have performed effectively in providing their services to users. However, these applications do provide some areas where improvements are necessary. For instance, users very often require new features to be added to applications. Since the applications are launched as a single WAR file containing JARs (libraries), during maintenance or while adding new features, the modified libraries have to be recompiled and packed into a new WAR file. The running application has to be stopped and the new updated WAR file is deployed in its place. This causes an application to be offline for a certain amount of time which inconveniences users who would like to access it at that time. For critical changes, the updating of an application is done immediately otherwise updates have to wait until the next application release date. Another issue is that, although applications in *Clintweb* are based on a modular architecture, where functionality is separated into different JARs, these applications do not take full advantage of this modularity. They are still confined to behaving like normal web application where all functionalities are packed into one file. Therefore application parts cannot be manipulated during runtime.

A solution is needed to convert *Clintweb* into a framework which supports the updating of applications during runtime. Such a solution will improve user experience when interacting with the framework because they will not be inconvenienced during application updates. The approach taken by this thesis is to find means in which applications can be split into different modules, therefore allowing them to be independently deployed and manipulated. It is the intention of this thesis to investigate the available technologies which can provide an environment for launching such dynamic modular applications and at the same time provide supports the features offered by *Clintweb*.

3. Modular Web Applications

The aim of this research is to investigate how the *Clintweb* framework cited in chapter 2 can be converted to a framework that supports dynamic modular web applications. This chapter will give a brief introduction of the general meaning of dynamic modular web applications. It will thereafter outline the available popular technologies used for hosting and managing these types of applications. Comparisons will be made between these technologies in order to determine the most efficient and effective technologies that can be used in designing a new framework for applications in *Clintweb*.

3.1. Modular Application Design Principles

In software engineering, modularity is a design technique where an application is divided into smaller functional parts. Each functional part is referred to as a module, which adds unique functionality to an application. Modules have a clean separation of functionality between them whereas the functionality of a module does not interfere with the functionality of other modules in an application. The partitioning of application functionality is important because it ensures the plug-in nature of a modular application, where modules are loosely coupled and can be plugged in and out (in other words, modules can be added to and removed from an application). The following rules apply to modular applications[23]:

- Modules should not be allowed to directly reference other modules in an application. The communication between modules should be done via shared services[24], shared resources[25] or other loosely coupled communication techniques in order to maintain a clean separation of functionality between modules.
- Modules should not manage their own dependencies[26]. Dependencies should be managed externally by the runtime environment which hosts the modular applications.

[23] Modularity by Microsoft msdn, Available at: http://msdn.microsoft.com/en-us/library/ff648404.aspx [Accessed 20 June 2010] and http://en.wikipedia.org/wiki/Modular_programming [Accessed 20 June 2010]

[24] Shared services refer to a common interface which is used to define the communication between modules whilst maintaining the independence of modules.

[25] Shared resources can be in a form of a web service or a database, which modules can access to communicate with other modules

[26] Dependencies in a modular application refer to how modules can be dependent on other modules. For example, Module A may require services from Module B, therefore making Module A dependent on Module B.

- Modules should be dynamic by allowing them to be added and removed from applications during runtime. This feature is known as *Hot Swapping*[27] which refers to plugging-in or plugging-out of modules during runtime without having to stop an application

The size of modules plays a big role in application design. Modules should neither be too small nor too large. The size of the modules affects the ease of application maintenance[28], as shown in the chart below:

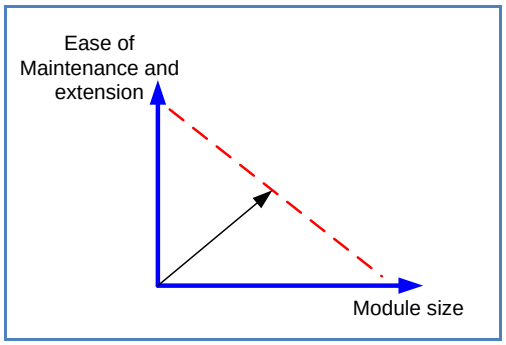


Figure 3.1: Module size

As seen in the *Figure 3.1*, the increase in module size leads to a decrease in ease of maintenance and extension. Large modules are difficult to maintain and extend because they contain a huge amount of an application's functionality. This makes it difficult to separate a module from an application without having a large impact on an application's overall functionality. However, making modules too small increases module dependencies; which contradicts an important requirement stating that modules should have minimal dependencies in order to keep them loosely coupled. The center arrow in *Figure 3.1* suggests the optimal value, where modules are not too large or too small. By optimizing modules sizes, it becomes easier to identify impacts of change within a modular application and perform maintenance on modules.

Modules in applications should be reusable. The reusability property is directly dependent on the module size, as shown in the chart below:

[27] Hot swapping, Available at: http://en.wikipedia.org/wiki/Hot_swapping [Accessed 20 June 2010]
[28] Modularity patterns, Available at: http://techdistrict.kirkk.com/2009/08/05/modularity-patterns/ [Accessed 20 June 2010]

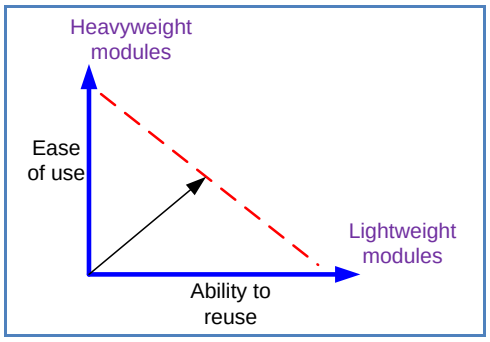


Figure 3.2: Module ability to reuse[29]

As shown in *Figure 3.2*, the ability to reuse modules increases for small sized modules because they can be easily replaced. However, heavyweight modules have a high ease of use but they are difficult to reuse. Large modules require more effort to decouple from an application, because they contain a large portion of an application's logic. Therefore, it is a requirement for modules to be of optimal size (the middle arrow in *Figure 3.2*) in order to fulfill the reusability requirement.

As discusses in this section, there are many considerations which have to be kept in mind when designing modular applications. Modules have to follow the above stated rules in order to allow *Hot Swapping*. In this thesis the term swapping-in and swapping-out will be used to refer to adding and removing modules from an application.

The next section discusses a framework specification that is designed to support modular applications which follow the module characteristics discussed in this section.

3.2. Introduction to *OSGi*

OSGi (Open Source Gateway initiative) is a Java based framework specification allowing small, reusable and collaborative Java modules to run collectively on a single Java virtual machine[30]. *OSGi* is an open source universal middleware which provides a service oriented, component-based environment for applications and offers ways to manage an application's components lifecycles. *OSGi* has gained popularity and has been widely acceptanced by many web

[29] Modularity patterns, Available at: http://techdistrict.kirkk.com/2009/08/05/modularity-patterns/ [Accessed 20 June 2010]

[30] *OSGi* and Equinox: Creating Highly Modular Java™ Systems – chapter 2 (*OSGi* concepts) [2] and *OSGi* alliance, Available at: http://www.*OSGi*.org/Main/HomePage [Accessed 10 June 2010]

frameworks, which include: Apache Felix[31], Equinox[32] and Knopflerfish[33]. The *OSGi* specification is composed of a set of services represented as layers that an *OSGi* implementation container (e.g. Apache Felix) must implement. *Figure 3.3* shows the layers of the *OSGi* framework:

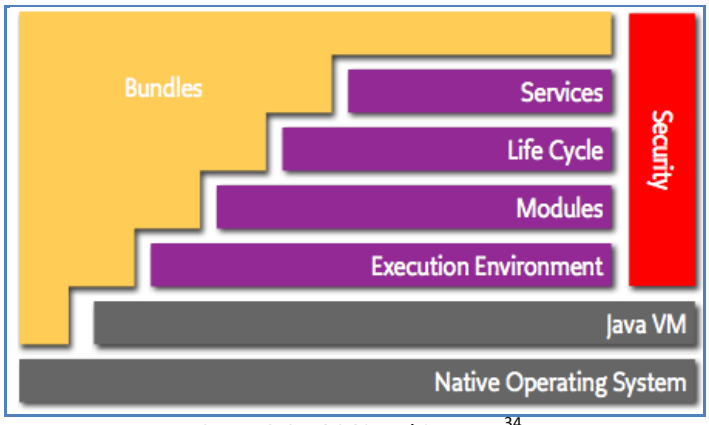


Figure 3.3: OSGi architecture[34]

According to *Figure 3.3*, *Bundles* are application modules which follow the module specifications discussed in section 3.1. Bundles are in form of JARs[35] containing identity information and dependencies declarations, which are described in a manifest file located inside a bundle. The service layer provides a means for bundles to communicate with each other via a specified service. A service involves an interface offered by a bundle which other bundles can access to provide their custom implementation of the interface. Services create a clean separation of code specification and implementation; they additionally allow bundles which offer a service to execute processes defined in other bundles which provide an implementation of a specified service.

The Life cycle layer provides control over the lifecycles of application bundles. Bundles lifecycles can be dynamically changed over the lifetime of an application. Bundles can be in any of the following states (*Figure 3.4*) during runtime:

[31] See section 3.2.2.2
[32] See section 3.2.2.1
[33] See section 3.2.2.3
[34] *OSGi* Alliance, *OSGi* architecture, Available at: http://www.*OSGi*.org/About/WhatIs*OSGi* [Accessed 20 June 2010]
[35] JAR: see section 2.4

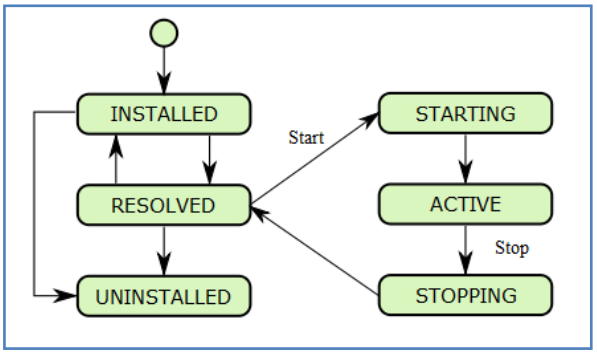


Figure 3.4: Bundle lifecycle[36]

A bundle's state can be changed to any of the above depicted states using simple *OSGi* commands.

The modules layer defines how packages belonging to bundles can be imported and exported. Java packages can be selectively exported, allowing other bundles to import and utilize classes in the exported packages. Bundles may also have dependencies between each other. A bundle requiring classes or functionality provided by another bundle will be dependent on the bundle hosting the classes/functionality. Therefore the dependent bundle can only be activated when the bundles it depends on are active. No circular dependencies between bundles in *OSGi* are allowed. A bundle's dependencies, exported and imported packages are defined in a manifest file (*MANIFEST.MF file*) which is located in the base directory of a bundle.

The security layer handles the security aspects of *OSGi*. Each bundle has its own memory space, and there are strict security rules governing access of a bundle's memory spaces. A bundle is not allowed to directly access resources and classes in another bundle. This ensures bundle security and enforces the loosely coupled property of bundles. The Execution environment layer defines the methods and classes that are provided by the *OSGi* platform[37].

OSGi implementations function by creating a bundle context, which is the environment of an active *OSGi* based application where bundles are deployed. Within the bundle context, a bundle can be started; in this case, the *OSGi* framework must resolve a bundle's dependencies, by checking if its dependent bundles are active. A bundle can be stopped or uninstalled, however bundles dependent on the stopped bundle will still be able to reference its classes which are specified in its exported packages. The bundles which reference the stopped bundle have to be updated in order to update their imported packages. When an update is performed, *OSGi*

[36] *OSGi* - bundle lifecycle, Available at: http://en.wikipedia.org/wiki/*OSGi* and http://www.osgi.org/javadoc/r4v42/org/osgi/framework/Bundle.html [Accessed 20 June 2010]
[37] *OSGi* Alliance – *OSGi* platform: Available at: http://www.*OSGi*.org/About/WhatIs*OSGi* [Accessed 20 June 2010]

basically re-wires the application bundles references again. When a bundle is uninstalled, it is completely removed from the framework's context and it will no longer be available to an application. Hence to make it available, it must be re-installed. When a bundle is stopped, it remains in the framework's context but it is no longer *ACTIVE*; it remains in a *RESOLVED* state.

In *OSGi*, each bundle has its own class loader which is instantiated during bundle start up. The class loader is responsible for resolving a bundle's classes and imported package stated in its manifest file. The bundle class loaders are not visible to each other, therefore a class loader is only able to locate and instantiate classes located within its host bundle.

3.2.1. Why choose *OSGi*?

OSGi is preferable over the conventional implementation of applications which stack JARs[38] together. Conventional applications which use JARs have tight coupling between JARs without any formal structure[39]. This is because when all JARs are in the same class path, they reference classes within other JARs without any restriction (unless methods are set to private), which leads towards a tightly coupled structure, as shown below:

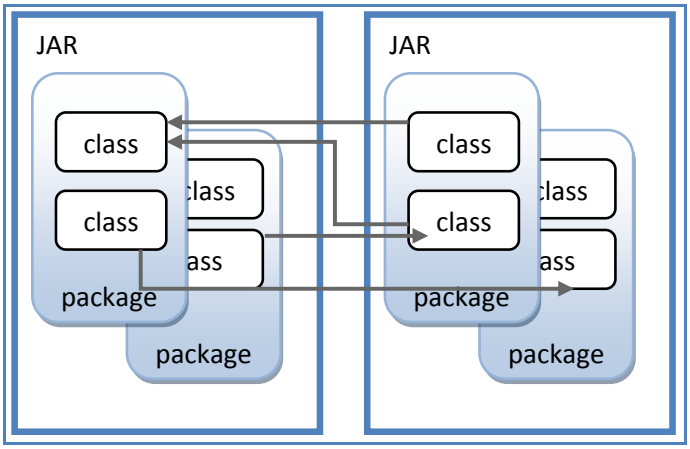


Figure 3.5: Tightly coupled JARs[40]

As seen in *Figure 3.5*, in a normal JARs scenario, there are no definitions of dependencies and each class has access to classes located in other JARs. In *OSGi*, packages in a bundle are hidden from other bundles unless explicitly exported. Bundles have to explicitly import an exported

[38] See section 2.4
[39] *OSGi* alliance, *about OSGi,* Available at: http://www.*OSGi*.org/About/Technology and '*OSGi* and Equinox: Creating Highly Modular Java™ Systems' chapter 2
[40] *OSGi* and Equinox – Creating Highly Modular Java Systems – chapter 2

package for it to have access to the classes within the package. This preserves the loosely coupled property of bundles in *OSGi*. In modular applications, modules must be loosely coupled so that managing of individual modules can be realized. If application modules are tightly coupled, it is difficult to do module replacement without having to stop a running application.

OSGi additionally uses services to enforce dynamic collaboration between bundles. According to the *OSGi* service specification, bundles providing services are not aware of the bundles using their services. Bundles using a service need only to know the service interface which is provided as part of the service registration to the *OSGi* service layer. Such a feature is not found in conventional applications consisting of JARs.

Based on the features offered by *OSGi*, it is a suitable candidate for implementing a modular web application which offers control over the lifecycle each module. As much as *OSGi* provides the desired functionality required by a modular application, it does have a major disadvantage. In exceptional cases, *OSGi* can lead to redundant code in an application environment. This scenario can happen when a bundle is uninstalled, and a new one is installed, to replace the uninstalled bundle. All bundles importing packages from the uninstalled bundle must be updated in order to import the packages of the new bundle. If this update is not done, the class loader of the package importing bundle still reference the old imported package which it loaded from the uninstalled bundle during application start up. During an update, a bundle's class loader disposes old imported packages and imports packages afresh; therefore the new versions of exported packages will be imported by a bundle. A system administrator must initiate the update process for all bundles referencing a bundle that has modified exported packages[41], otherwise bundles will be referencing the old set of packages and the correct packages will not be used. Therefore there will be two sets of exported packages, one from the uninstalled bundle and the other from the newly installed bundle. This is the redundancy which has to be avoided when changes are made to an application, by calling the UPDATE command on all bundles which are dependent on the uninstalled bundle. It is important to have a clear outline of the structure of an application which contains a listing of bundles and their dependencies, therefore making it easy to identify area of impact within an application when changes are applied on a specific bundle.

[41] *OSGi* Alliance*, Importance of Importing and Exporting*, Available at: http://www.*OSGi*.org/blog/2007_04_01_archive.html [Accessed 20 July 2010]

3.2.2. *OSGi* Web Implementations

OSGi was originally intended for running in embedded devices and home service gateways, but developers have used the *OSGi* specification to create web frameworks which are based on its principles of modularity, component orientation and service orientation[42]. This section lists discusses three popular *OSGi* based web application frameworks.

3.2.2.1. Equinox

Equinox is the module runtime at the core of the Eclipse IDE which implements the *OSGi* specification features[43]. According to Eclipse org[44], Equinox is a plug-in system that allows the implementation of applications consisting on a set of bundles using common services and infrastructure. The following sections discuss how Eclipse implements the Equinox framework in order to get an insight on how Equinox functions.

3.2.2.1.1. Eclipse Plug-in Architecture

A prime example of how a modular application can be built is Eclipse IDE, where the application is built on a collection of plug-ins which are resolved by a plug-in loader during start-up. A plug-in in Eclipse is an *OSGi* bundle containing a manifest file where dependencies, exported and imported packages are defined. *Figure 3.6* shows the structure of plug-ins in eclipse:

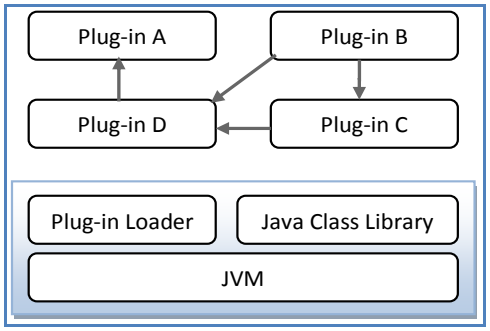


Figure 3.6: Eclipse architecture[45]

[42] Apache Felix, Available at: http://felix.apache.org/site/index.html [Accessed 20 June 2010]
[43] Java World *OSGi* tutorials, Available at: http://www.Javaworld.com/Javaworld/jw-03-2008/jw-03-*OSGi*1.html [Accessed 20 June 2010]
[44] Equinox Home, Available at: http://www.eclipse.org/Equinox/ [Accessed 20 June 2010]
[45] 'Eclipse plug-ins (Third edition - 2008) by Eric Clayberg and Dan Rubel – Chapter 3: Eclipse infrastructure

The direction of the arrows in the above figure represents dependencies between plug-ins. In an eclipse installation, a plug-in folder is created in the root path of the eclipse application. All plug-ins are stored in this folder. When eclipse starts, it searches the plug-in folder for available plug-ins and loads them. Each plug-in contains a manifest file which tells eclipse what is needed prior to activation of a plug-in. The plug-in manifest file also contains entries of the bundle name, unique identifier, version, `Activator` class and provider[46]. The bundle name is a qualified name given to a plug-in which allows the developer to recognize a plug-in. The plug-in identifier uniquely identifies the plug-in in the runtime environment. In eclipse, the plug-in identifiers are based on the Java package naming convention, where plug-ins which have related functionality have identifiers starting with the same name sequence (for example: `org.eclipse.equinox.registry` and `org.eclipse.equinox.preference`). The plug-in version is used to differentiate multiple versions of a plug-in. The version usually contains three numbers: the major version, the minor version and the service level (e.g. 1.2.0).
The plug-in `Activator` class is an optional entry for referencing a class which is invoked when a plug-in is started or stopped. Activators are useful for registering events and services during plug-in's startup. The `Activator` must implement `start()` and `stop()` functions which are respectively called when starting or stopping a plug-in.

i. Extension points

Equinox extended *OSGi*'s specification by adding an Extension Registry layer on top of the *OSGi* service layer. The Equinox Extension Registry is a mechanism for supporting inter-bundle collaboration. This mechanism allows bundles to open themselves for extension or configuration by declaring an extension point[47]. A bundle is in essence telling other bundles that if they offer it certain information, it will perform a specific task. Bundles may contribute information to an extension point in the form of extensions. The main advantage of the extension registry is that a plug-in is not aware of the plug-ins connecting to its extension point.

Extension points are declared in a plugin.xml file located in the base directory of a bundle (each bundle has a plugin.xml file). As bundles are resolved by Equinox, their extensions and extension points are loaded into the Extension registry, therefore making them available to other bundles

[46] Similar ideas are also stated in the Eclipse Org, *Notes on the Eclipse plug-in Architecture* article, Available at: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html [Accessed 10 June 2010]
[47] '*OSGi* and Equinox – Creating Highly Modular Java Systems' chapter 2 – *section 2.7.2.*

during runtime[48]. The *Figure 3.7* shows how plug-ins connect with each other using extension points and extensions:

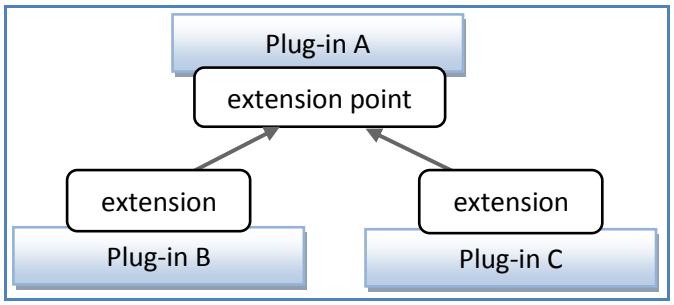


Figure 3.7: Plug-ins in Eclipse

In the figure above, plug-in A has defined an extension point where plug-in C and plug-in B can extend (the arrows in the above figure also depict dependencies between the plug-ins). For plug-ins to extend an extension point, they must define an extension in their plug-in.xml file by referencing the extension point which they are extending. This extension must explicitly define the ID of the extension point it is extending. An extension point defines its own XML vocabulary in an XML schema[49], which specifies the information an extending plug-in has to supply when contributing to an extension point. Depending on the extension point requirements, the XML schema can be defined to request bundles to provide a resource name (image, file), Boolean value, String or a Java class. If a Java class is to be defined in an extension point, its interface or super class must be specified. This allows the host plug-in (the plug-in which defines the extension point) to recognize the classes supplied at its extension points and call functions within them. When defining an extension point, the ID is necessary in order to allow other plug-ins to identify an extension point. The ID must be unique within an application; this helps in creating a clearly organized set of extensions and extension points.

The extension registry feature of Equinox can be visualized as follows: the plug-in which provides an extension point creates a contract and plug-ins willing to extend this extension point have to fulfill the requirements of the contract. Otherwise, if extending plug-ins violate the contract, the extension will not function. The host plug-in of an extension point has the responsibility of defining how an application should deal with faulty extensions. The best option is to ignore faulty extension.

[48] '*OSGi* and Equinox – Creating Highly Modular Java Systems' chapter 2 – *section 2.7.2.*
[49] An XML schema describes the structure of an XML document, see: http://www.w3schools.com/schema/default.asp [Accessed 20 June 2010]

The declaration of extensions and extension points is at the surface of the eclipse architecture. The processing of the extensions is done programmatically as specified in *Figure 3.8*:

```
IConfigurationElement[] extensionsArray = Platform.getExtensionRegistry()
    .getConfigurationElementsFor("extension point ID");

for (IConfigurationElement extension : extensionsArray) {
  String attrbute = extension.getAttribute("attribute name");
  Object obj = extension.createExecutableExtension("class");
}
```

Figure 3.8: Extension point processing

The information collected from an extension point is stored in objects which are of type `IConfigurationElement` interface. The `getConfigurationElementsFor` method reads extensions from a specified extension point which is reference using an ID. Using a for-loop, each extension object is accessed and its attributes are extracted using the `getAttribute` method by providing the attribute name as a String parameter. To extract a class from an extension, the `createExecutableExtension` method is used. Extension points in the new framework were processed using the above depicted code.

As seen above the implementation of classes from different plug-ins can be executed without the host plug-in having any knowledge of the extension provider plug-in. This just adds more emphasis on the modularity of this type of architecture. For plug-ins to use an extension point of another plug-in (host plug-in) there must be a dependency between the plug-ins.

The Equinox plug-in architecture would be ideal for a modular application, where communication between bundles can be done via extension points. Applications on the current version of *Clintweb*[50] utilize the extension point mechanism to allow libraries to be extended by other libraries. *Clintweb* does not use the Equinox extension registry classes but custom created classes which serve to do the same task as the Equinox extension registry classes. Each application library has a *plugin.xml* file which may declare extensions and/or extension points. However, applications in *Clintweb* function in a single class loader environment; therefore classes referenced by an extension point are easily resolved because they are visible to the class loader. The *Clintweb* extension registry feature cannot function in an *OSGi* based modular application environment because of it contains multiple class loaders. Classes declared by an extension point will not be resolvable if they are located in a bundle other than the extension point host bundle. The

[50] See chapter 2

Clintweb extension registry feature must be fully replaced by the Equinox's extension registry in order to be able to use extension point in an *OSGi* based modular application.

3.2.2.2. Apache Felix

Apache Felix implements the *OSGi*[51] service platform and other *OSGi* related technologies under the Apache license. Apache Felix has sub projects which specifically handle the functionality of the *OSGi* platform. The following are some of the sub projects according to the Apache Felix functionality documentation:

- Configuration Admin[52]: A service for management of bundle configuration properties.
- Dependency manager: A service for managing bundle dependencies during runtime. This project additionally manages the services and package dependencies between bundles. Unlike other *OSGi* implementations, service dependencies are specified in a declarative way in order to make monitoring and managing services easier.
- Apache Felix HTTP Service[53]: An implementation of the standard *OSGi* HTTP Service specification, which provides a simplified means of registration of `Servlets` and resources to a `Servlet` container.
- Apache Felix web console[54]: A tool for inspecting and managing *OSGi* framework instances using a standard browser.

Apache Felix extends the *OSGi* specifications with unique additional features which make it easier to create modular applications. For example, other *OSGi* implementations (e.g. Equinox) do not provide the dependency management feature of Apache Felix which manages services and packages. However, Equinox has the advantage over Apache Felix because it provides the extension registry feature, which eases communication between bundles. Equinox is not shipped with a web administration console like Apache Felix, but there are available external applications which can be used to manage the Equinox framework (for example: *mBs Prosyst Console*[55]).

[51] See section 3.2
[52] Apache Felix sub projects, Available at: http://felix.apache.org/site/index.html [Accessed 20 June 2010]
[53] Apache Felix HTTP service, Available at: http://felix.apache.org/site/apache-felix-http-service.html [Accessed 20 June 2010]
[54] Apache Felix Web Console, Available at: http://felix.apache.org/site/apache-felix-web-console.html [Accessed 20 June 2010]
[55] Prosyst, see: http://www.prosyst.com/ [Accessed 20 June 2010]

3.2.2.3. Knopflerfish

Knopflerfish is an implementation of the *OSGi* specification described in section 3.2. It contains the following components:

- Components defined by *OSGi*: This includes the base *OSGi* framework which enables bundles to be managed (started, stopped, uninstalled etc.) and the HTTP service for providing a web server where `Servlets` can be published.
- Knopflerfish Components: This includes a desktop application and a console for remotely managing bundles.

Knopflerfish can be used to deploy modular web applications however it only provides few additional features on top of the *OSGi* specification in comparison to Equinox and Apache Felix. Knopflerfish has the ability to support web applications due to its HTTP service, which supports the registration of `Servlets`. The Knopflerfish desktop application is advantageous in allowing the remote management of applications; it however does not entail a web based remote application management tool. It is preferable to have a web based management tool, which is more convenient in managing applications because no application installations are required.

3.2.3. Summary

Modular applications can be developed using the above stated *OSGi* implementations. Equinox[56] has an advantage over the other implementations because it is light weight (consumes the least memory) and its bundles are small sized. Equinox is more scalable in running many applications on the same machine. Performance is boosted under Equinox as compared to using the other frameworks (Apache Felix and Knopflerfish) because system memory is efficiently utilized due to Equinox's minimal memory requirements. Furthermore, Equinox introduces a unique way of building and deploying applications called *Component Oriented Development and Assembly* (CODA)[57]. CODA is advantageous because it gives developers more flexibility in assembling and customizing their applications. Developers can select components from different component producers, customize them to meet specific requirements and finally use them to create individual solutions[58]. CODA will allow templates for *Clintweb* applications to be created, so that

[56] See section 3.2.2.1
[57] CODA, Available at: http://www.eclipse.org/equinox-portal/whitepaper/20080310_equinox.php [Accessed 20 June 2010]
[58] Equinox advantages, Available at: http://java.sys-con.com/node/520844 [Accessed 20 June 2010]

developers will not have to always start from scratch when creating applications for the new framework. This will ensure application development is done quicker and with least amount of configurations.

Equinox also provides the Extension Registry[59] feature which is helpful for inter-bundle communication. Equinox is compatible with many web frameworks for hosting web applications as compared to Apache Felix and Knopflerfish; it can be easily placed into a web server by applying minimal configurations. Lastly, the Eclipse IDE demonstrates how a stable modular application can function using Equinox; new plug-ins can be deployed during runtime and they will be activated without an application restart. Therefore, Equinox is the preferred *OSGi* implementation in comparison to Apache Felix and Knopflerfish.

It is possible to switch between different *OSGi* implementations but this process requires configurations depending on the frameworks involved. For example, in order to deploy Equinox bundles in Knopflerfish, the Knopflerfish framework must be configured to support the Equinox's specific features like the Extension Registry. Such configurations include the addition of specific libraries.

3.3. Web Application Deployment Topologies

After investigating the different types of *OSGi* implementations[60], the next step is to discuss deployment topologies of modular web applications which are based on the *OSGi* implementations discussed in section 3.2.2. There are two types of deployment topologies which will be discussed in this section, the *web container in OSGi* and *OSGi in web container* deployments. Furthermore, functional implementations of each topology will be discussed and comparisons between them will be made, in order to decide which topology is suitable for the objectives of this thesis.

[59] See section 3.2.2.1
[60] See section 3.2.2

3.3.1. Web container in *OSGi*

This deployment architecture packages a web server into the *OSGi* based framework[61]. There are several frameworks which use this architecture to create an environment for deploying dynamic modular web applications[62]. This section discusses three frameworks which have this type of deployment profile.

3.3.1.1. *Spring DM*

Spring DM (Dynamic Modules) is an *OSGi* based framework where *Spring* powered bundles can be deployed[63]. *Spring DM* dynamically manages application modules using *OSGi*. To convert ordinary bundles to *Spring* powered bundles, a `/Spring` folder must be added under a bundle's META-INF[64] directory. This folder must contain the necessary configuration XML files required by the *Spring* framework for instantiation of a bundle and its properties (e.g. Services).

The core of the *Spring DM* framework is the `org.springframework.osgi.bundle.extender` bundle; it handles the instantiation and management of application bundles during runtime. During start-up the extender bundle attempts to load bundles under a specified directory by first checking if they are *Spring* powered. The extender bundle thereafter loads a bundle's configuration files from its META-INF/*Spring* directory and creates an application context for that specific bundle. The extender also checks if a bundle has registered services in its *Spring* configuration files. A bundle's `Activator` class[65] must also be registered in a configuration file. In a conventional *OSGi* bundle, the `Activator` class implements the `BundleActivator` interface to provide a bundle's start and stop methods. In *Spring DM*, the bundle `Activator` class does not have to implement an *OSGi* specific interface, but the functions of the `Activator` class have to be registered in a configuration XML file as a bean. *OSGi* services have to also be declared in a configuration file as beans. The *Spring DM* framework passes the bean information to the *OSGi* runtime during bundle start-up.

---

[61] See section 3.2

[62] See chapter 3

[63] Introduction to Spring Dynamic Modules by JavaWorld, Available at: http://www.Javaworld.com/Javaworld/jw-04-2008/jw-04-*OSGi*2.html?page=2 [Accessed 22 June 2010] and '*Spring* Dynamic Modules in Action' by Arnaud Cogoluegnes and Thierry Templier

[64] The META-INF directory stores the manifest file of a bundle. It is located in the base directory of a bundle. See section 3.2

[65] The Activator class manages the lifecycle of a bundle and it is instantiated when a bundle is started and stopped. Available at: http://fusesource.com/docs/esb/4.1/*OSGi*/activator.html [Accessed 22 June 2010]

Spring DM can provide a suitable environment for a modular application. It runs on an *OSGi* framework implementation (e.g. Apache Felix, Equinox) which allows it to deploy and manage *Spring DM* modular applications. However, bundle properties (`Activator` class and Services) have to be declared in *Spring* configuration XMLs. If these configurations are not included in a bundle then it cannot function in the *Spring* environment. Extra efforts are required in order to make application modules compliant with *Spring DM*.

3.3.1.2. *Spring DMK*

Another deployment topology of *Spring DM* is shown below:

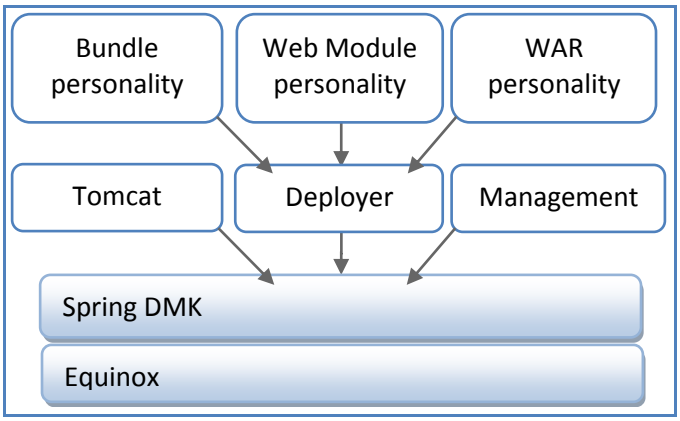


Figure 3.9: Spring DMK topology [66]

The *Spring DMK* (Dynamic Module Kernel) is the *OSGi* based kernel which takes advantage of the modularity and versioning of *OSGi* and extends its capabilities by adding more functionality to the DM server. Equinox[67] is the *OSGi* implementation used in the kernel. This gives provisions for modules to be deployed in *Spring DM* and managed like in an ordinary *OSGi* environment. On top of the kernel, there is a deployer which converts different deployment profiles into bundles and runs them in Equinox. This allows, for applications in WAR[68], bundle and web module formats to be deployable in the *Spring DM* framework. Additionally, this *Spring DMK* deployment uses the

[66] Spring DM Server by Springsource, Available at: http://static.*Spring*source.org/s2-dmserver/2.0.x/programmer-guide/html/ch02s02.html [Accessed 22 June 2010] and http://static.*Spring*source.org/s2-dmserver/2.0.x/programmer-guide/html/ch02s03.html [Accessed 22 June 2010]
[67] See section 3.2.2.1
[68] See section 2.1

Tomcat web server to provide web container functionalities to applications. *Spring DMK* supports three deployment formats[69]:

- Standard WAR – which the deployer converts into a bundle and launches it on tomcat. In this setting, the WAR file does not have to be modified to run in the *Spring DM* kernel. The problem with this deployment is that it does not consider the modularity of an application. Applications in this setting are deployed as a single WAR file.
- Shared Libraries – In this deployment the DM server contains a manifest file declaring dependencies and imported/exported packages. The dependency declarations in the *Spring* manifest file are *OSGi* compliant, meaning that the *Spring DM* kernel can resolve dependency classes when required. This deployment format is ideal for modular application in WAR file formats.
- Shared services WAR – In this format, interfaces can be programmed which provide services to bundles running within the same application.

Spring DM can be used as a platform for deploying a modular application. It offers a web server that manages deployment of modules as bundles and it provides communication between bundles during runtime. An application can be kept in WAR format and still be deployed. However, this topology requires configurations to be done on web applications; WAR files have to be made *OSGi* compliant by having to manually declare dependencies and package imports/exports within them.

3.3.1.3. *Equinox - Jetty*

This deployment profile launches the Jetty web server in an Equinox framework. Jetty is a light weight HTTP server and `Servlet`[70] container which can be deployed as a bundle in Equinox[71]. Jetty is a fully featured web server for static and dynamic content[72]. Jetty combines server and

[69] Deployment Architecture *by* Springsource, Available at: http://static.*Spring*source.org/s2-dmserver/2.0.x/programmer-guide/html/ch03.html [Accessed 22 June 2010]
[70] See section 2.1
[71] See section 3.2.2.1
[72] Jetty - Available at: http://Java-source.net/open-source/web-servers/jetty [Accessed 20 June 2010]

container solutions to run within the same process, without interconnection overheads and complications. Jetty has a simple structure which is illustrated in the figure below[73]:

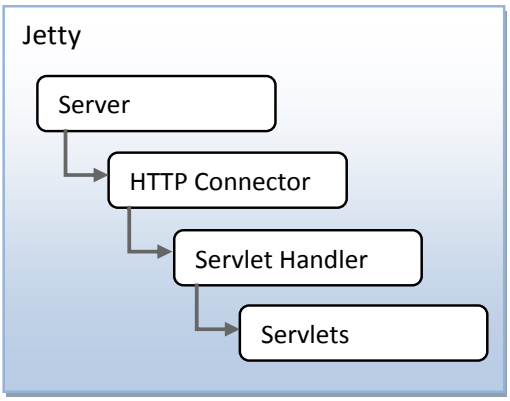


Figure 3.10: Jetty web server

An instance of the server is created during start-up. A server can have multiple connectors which declare properties like the access port of the application, the HTTP request header size and timeout length. The connector passes HTTP requests to the `Servlet` handler which is used to examine requests and responses. The `Servlet` handler can then forward requests to `Servlets` for processing.

Equinox - Jetty is capable of running a modular application. Equinox is used to control the Jetty web server bundle and other application bundles. The advantage of this deployment is that all components including the web server are managed as bundles, and they have full access to Equinox's functionality. The extension point[74] functionality provided by Equinox allows for properties like `Servlets`, to be registered and retrieved from any bundle within the framework. *Equinox - Jetty* is easy to install and requires minimal configuration efforts for bundles to be deployed within the framework as compared to the Spring DM architectures where bundles must be packed with *Spring* configuration files.

3.3.2. *OSGi* in Web container

In this topology, *OSGi* is embedded within a web container. The web container has to instantiate *OSGi* and thereafter application components can be started. The web container has to forward user requests to the *OSGi* container using a bridge `Servlet` (further explained in section 3.3.2.1).

[73] Embedding Jetty, Available at: http://wiki.eclipse.org/Jetty/Tutorial/Embedding_Jetty [Accessed 20 June 2010]
[74] See section 3.2.2.1

3.3.2.1. Equinox in Tomcat

In an alternative deployment topology, Equinox can be deployed into a Tomcat container[75]. Other *OSGI* implementations (e.g. Apache Felix) can be used in this topology but they require more configuration efforts than Equinox. Equinox provides a simple `Servlet` bridge between itself and a web container as shown in the figure below[76]:

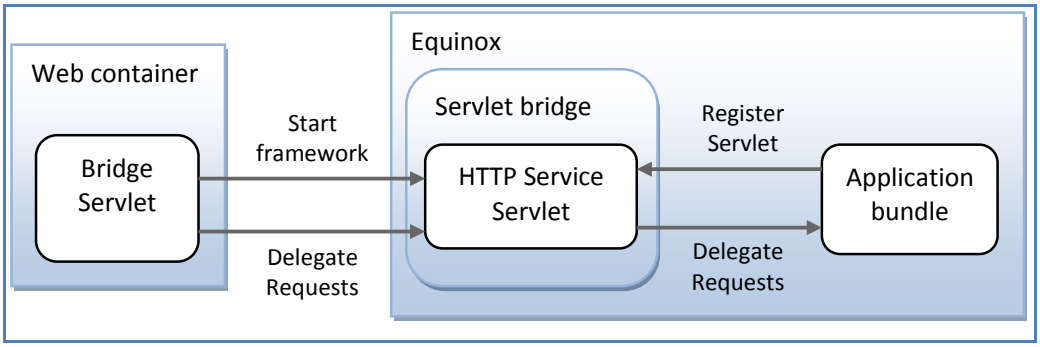


Figure 3.11: Equinox in Tomcat

The Bridge `Servlet` receives HTTP requests from the web container (Tomcat) and forwards it to `Servlets` registered on its `Servlet` registry extension point. On the web container side, there is bridge `Servlet`, in the form of a WAR[77] file (called *bridge.war*). The bridge `Servlet` is responsible for:

- Embedding and launching Equinox within the tomcat container
- Tunneling `Servlet` requests from Tomcat to the Equinox runtime's `Servlet` bridge

Tomcat instantiates the *bridge.war* it in order to launch Equinox. Thereafter the Equinox console will be displayed on the Tomcat console. *OSGi* commands can be inputted in the Tomcat console to control application component. Client requests going to the Equinox based application will initially be received by the Tomcat web container and then tunneled to Equinox via the bridge `Servlet` component. In order to support JSP in this architecture, additional configurations have to be done on the Tomcat container. Since all application modules are controlled by Equinox, the Tomcat JSP compiler cannot be used to service JSPs. Instead an external JSP compiler like Jasper has to be used[78].

[75] See section 2.1
[76] Mit *OSGi* Webanwendung entwickeln – Was geht, was nicht? - 2009 P. Roßbach, G. Wütherich, M. Lippert
[77] See section 2.1
[78] Embedded *OSGi*, Available at: http://techdistrict.kirkk.com/2009/02/16/embedding-*OSGi*-in-tomcat/ [Accessed 20 June 2010]

An example of this type of deployment is the eclipse BIRT [79](Business Intelligence and Reporting Tools) project. BIRT is an open source Eclipse-based reporting system that can be integrated into a Java application to create and deploy reports. BIRT is constructed as a composite of components (Eclipse plug-ins) which offer different functionality. These components can be dynamically deployed into a Tomcat web container as WAR files. BIRT has however suffered from memory related problems[80]. Many users have complained of memory leaks and high memory usage occurring in BIRT applications which causes the Tomcat server to crash. In this configuration, object references are not properly cleaned when objects are destroyed which therefore leads to memory leaks. The object references problem is a result of a configuration problem between Tomcat and Equinox. The Tomcat web server may reference an object which was already destroyed by the Equinox framework, which leads to crashes the Tomcat server. The architecture stated in *Figure 3.11* is not stable as compared to the other topologies (Web server in *OSGi*) discussed in this chapter.

The *Equinox in Tomcat* approach is successful in creating an environment for a modular application but it does require vast amount of configurations. Depending on the operating system, Tomcat has to be configured in order to allow Equinox to function in the Tomcat console[81]. Furthermore, the Equinox runtime is dependent on Tomcat, so if a problem occurs in the web container, Equinox will be affected. Furthermore, the BIRT implementation of this architecture demonstrates that this architecture is not stable because it is prone to serious memory problems.

3.4. Session State Management

One of the most important issues in a dynamic modular application is the preservation of session objects when *Hot Swapping*[82] occurs. The term 'session objects', refers to the objects which hold user inputted information during the span of a session. In an ideal case, when modules are dynamically swapped in and out, their session objects should be restored in order for users to not

[79] Eclipse Org, BIRT project, Available at: http://www.eclipse.org/birt/phoenix/ [Accessed 28 July 2010]

[80] Memory leaks are caused by referencing objects that are no longer in use. This happens when references to destroyed objects are not cleared. See: http://java.sys-con.com/node/1071319 for Java memory problems [Accessed 20 June 2010]

[81] Embedded *OSGi*, Available at: http://techdistrict.kirkk.com/2009/02/16/embedding-*OSGi*-in-tomcat/ [Accessed 20 June 2010]

[82] See section 3.1

have to input information again. Some web containers are able to persistently store session objects (e.g. Tomcat) by serializing them on to the disk. So that when a new application module is swapped in, the session objects can be restored into it. However some web containers, like Jetty, do not have this functionality. In an *OSGi* environment, there are no in-built mechanisms for saving session objects after a bundle swap occurs. The session objects must be manually serialized into an application's memory or on the disk. Writing objects to the disk requires more time as compared to storing them in the application's memory. This is because the hardware I/O operation is slower compared to accessing RAM memory[83].

There are tools which can be used to preserve session objects when module swaps occur. *XStream*[84] serializes objects into XMLs, which can later be parsed into Java objects and placed back into a running application. The disadvantage of storing session objects as XMLs is that, depending on the structure of the Java object being serialized, the XML file can increase in memory storage size. The alternative method of preserving session objects is using the Java `Serializable`[85] API, which converts objects into byte streams. In this case, serialization of an object is enabled by a class implementing the `java.io.Serializable` interface. Classes that do not implement this interface will not have any of their state serialized. The `Serializable` API also manages class versioning of serialized object especially during migration between different class versions. This is necessary when a serialized object with an old data structure has to be converted to an object with a different data structure during de-serialization. Java Serialization is relatively stable when fields are added to or removed from a class. It is easier to place the serialization code into a class as compared to the Object-XML converter code.

3.5. Summary

The discussed modular web application deployment architectures (*Spring DM* and *Equinox - Jetty*[86]) provide functional solutions for deploying modular applications. They only differ in setup configurations. *Spring DM* solutions require vast configurations to be done on application bundles

[83] Java I/O performance, Available at: http://java.sun.com/developer/technicalArticles/Programming/PerfTuning/ [Accessed 10 May 2010]
[84] XStream object serializer, Available at: http://xstream.codehaus.org/ [Accessed 10 May 2010]
[85] Serialization at Java SUN, Available at: http://Java.sun.com/developer/technicalArticles/Programming/serialization/, http://www.tutorialspoint.com/Java/Java_serialization.htm and Java World serialization at: http://www.Javaworld.com/community/node/2915 [Accessed 10 May 2010]
[86] See section 3.3.1

in order to make them compatible. In *Spring DM* and *Spring DMK* architectures, bundles have to be packed with *Spring* configuration XMLs to make them *Spring* powered. The process of having to create configuration files for each bundle is tedious. In the `Servlet` bridge configuration, an external JSP compiler has to be configured into the framework and the bundle profile may have to be changed to a WAR file instead of a JAR. It is therefore recommended to use the *Equinox - Jetty* approach because it does not require vast amount of configurations. In Equinox, a bundle just has to fulfill the *OSGi* bundle specification and it can thereafter be deployed in to the framework. The Jetty web server is deployed as an ordinary bundle in Equinox, and its lifecycle can be controlled like any other bundle. Additionally, Equinox provides the Extension Registry functionality which allows creation of extensions and extension points. This feature will be advantageous in allowing bundles to offer functionality to other application bundles while still maintaining application modularity. Using extension points, bundles can be able to exchange resources and execute code located within other bundles.

Based on the analysis presented in this chapter, it is best to deploy a web server in an *OSGi* container than vice-versa because it is stable and bundle deployment formats can remain as the standard JAR format. In terms of application web servers, Jetty is a preferable web server as compared to frameworks offering other web servers. According to a *Jetty vs. Tomcat comparative analysis study*[87], Jetty has better scalability than the Tomcat web server when there are many client connections. Additionally, Jetty has a small memory footprint compared to other web servers, therefore less memory and CPU cache is used by the Jetty web server. The *Jetty vs. Tomcat comparative analysis article* further states that Jetty has better performance when serving static content because it uses advanced memory mapped file buffers combined with NIO[88] to instruct the operating system to send file contents at maximum DMA[89] speed without entering user memory space or the JVM. Furthermore, Jetty can handle HTTP session security by providing authentication and user access control for applications. Jetty is therefore the memory efficient and stable choice for the new framework in comparison to other web servers.

This chapter outlined that the best solution for hosting dynamic modular web application is the *Equinox - Jetty* framework. However, further considerations have to be made before a framework

[87] *Jetty vs. Tomcat, A comparative analysis*, Available at: http://www.webtide.com/choose/jetty.jsp [Accessed 10 May 2010]

[88] NIO: Non-blocking I/O is a form of input/output processing that allows other processing to continue before transmission is finished. See: http://en.wikipedia.org/wiki/Asynchronous_I/O and http://msdn.microsoft.com/en-us/library/aa365683%28VS.85%29.aspx [Accessed 10 May 2010]

[89] Direct Memory Access: A feature of modern computers that allows certain hardware subsystems within a computer to access system memory for reading and/or writing independently of the CPU. See: http://en.wikipedia.org/wiki/Direct_memory_access and http://www.pcguide.com/ref/mbsys/res/dma/index.htm [Accessed 20 June 2010]

can be selected for this thesis. In order for applications be fully dynamic, the `Serializable` API has to be built into the web framework (*Spring DM* or *Equinox - Jetty*) so that user session state information can be preserved and restored during bundle swaps. Furthermore, *JSF* has to be made functional in the web framework. *JSF* was not designed for *OSGi* based application environments. Since *OSGi* implementations use multiple class loaders, it becomes problematic for *JSF* to resolves its components because it is designed to function under a single class loader environment. The next chapter will outline the possible ways in which *JSF* can be configured to function in the *Equinox - Jetty* and *Spring DM* frameworks.

4. Requirements Analysis

Chapter 2 discussed *Clintweb's* architecture and the main challenges it faces. It was noted that *Clintweb* is to be migrated to a framework which supports deployment of dynamic modular web applications in order to allow maintenance of applications during runtime. Chapter 3 discussed the available technologies which can be used to realize the desired framework. This chapter continues to identify the requirements related to the new *Clintweb* framework and its applications. These requirements include the features which the new framework must offer to applications. The system user and administrator requirements will also be discussed in this chapter.

4.1. Framework Requirements

This section discusses the requirements which are to be fulfilled in relation to the current *Clintweb* framework[90]. These requirements include features offered by the current *Clintweb* that must also be present in the new framework.

4.1.1. Application Requirements

Applications in the new framework will have to be distributed into independent modules. The modules should follow the module specification outlined in section 3.1, where each module contains unique functionality and can communicate with other modules via predefined interfaces and/or services. The modules should be dynamic by allowing them to be swapped in and out of an application during runtime (This feature is referred to as hot swapping).

This modular architecture can be realized using the *OSGi* framework[91]. Applications in *Clintweb* can be split into sets of *OSGi* bundles. In a *Spring DM*[92] implementation, bundles will have to be extended to contain *Spring* configuration XML files to make them compatible with the *Spring DM* framework. In the Equinox framework, bundles do not need any extra configurations to make

[90] See chapter 2
[91] See section 3.2
[92] See section 3.3.1

them deployable. Equinox additionally provides bundles with the extension registry feature[93], which allows them to have another means of exchanging resources and providing functionality between each other.

It is a requirement that when a module is removed from an application and replaced with another module, the objects containing user inputted information in the swapped-out module should not be lost. The data should be present in the swapped-in module so that a user will not have to input his data again. The only method of fulfilling this requirement is that the objects storing user information in the swapped-out module have to be temporarily saved at a predefined location and then restored when the replacement module is swapped-in. These objects can either be saved in an application's memory or on the disk, so that when the new module is swapped in, the saved objects can be restored into it. As discussed in section 3.4, the `Serializable` API can be used to preserve session objects. In comparison to other object preservation techniques, storing objects as XMLs using *XStream*[94] requires more Java code and additional libraries to be packed into an application to support this functionality. The `Serializable` API is easy to build into an application and it is very stable in preserving and recovering Java objects.

4.1.2. Resource Requirements

Each module should be able to independently contribute resources to an application. When a module is deployed into an application, its resources should be resolvable upon user request. Furthermore, when a module is removed from an application, its resources should no longer be available to the application. Modules should be able to reference resources in other modules. Therefore, linking of resources in different modules should be made possible as shown in the figure below:

[93] See section 3.2.2.1
[94] See section 3.4

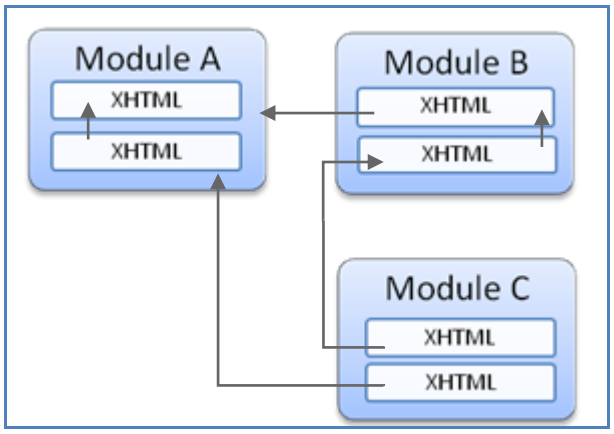


Figure 4.1: Resource navigation

Resources are in the form of XHTML files as depicted in the figure above. A resource should be able to declare a reference name to a resource located in another module without knowing of its location and what information it contains. As seen in the above diagram, *Module C* can declare a reference to a resource in *Module A* without having knowledge of where the resource is located. The process of resolving referenced resources should be handled by a central bundle. This requirement should be implemented in a way that modules should still maintain the loosely coupled property. It should also be noted that navigation between resources in different bundles should be clearly outlined in order to avoid circular dependencies between modules. Circular dependency occurs if *Module A* references a resource in *Module C* and *Module C* also references a resource in *Module A*. If *Module C* is swapped out, then *Module A* will have a null reference to *Module C* (because *Module C* has been swapped out). It is therefore a requirement that resources should only reference resources within their host module or modules it is dependent on. For example in *figure 4.1 Module B* should only be allowed to reference resources in *Module A* and itself.

The resources requirements can be solved using *OSGi*, where bundles can contain resources which are accessible during the span of their lifecycles[95]. When a bundle is uninstalled, its resources will no longer be available to an application. However, bundles cannot be able to directly access a resource in another bundle. This is because communication between bundles can only be conducted via an interface or an *OSGi* service. The Equinox framework's extension registry feature supports registration of resources under an extension point. This feature allows bundles to reference resources in other bundles. In *Spring DM*[96], resolving of resources is done by the

[95] See section 3.2
[96] See section 3.3.1

framework; they do not have to be explicitly registered in order for them to be resolved like in the Equinox framework.

4.1.3. *JSF* Requirements

JSF[97] components must be resolvable. As discussed in section 2.2, *Clintweb*[98] uses the *JSF* framework to implement some of its functionalities. Each library in *Clintweb* provides unique *JSF* components like *Managed Beans*, *Navigation Rules* and custom UI components to an application. *JSF* components are declared in a *faces-config* XML files which are contained in *Clintweb's* application libraries. Since all application libraries are packed in a single WAR[99] file under Tomcat's *webApp* directory, Tomcat is able to access all *faces-config* XML files (in all libraries) and instantiate their declared components. In *OSGi*, *JSF* functionality cannot be split into individual bundles using the *faces-config* files. Each bundle has its own class loader, therefore all *faces-config* files will be loaded under different class loaders and they will not be visible to the *JSF* system. *JSF* cannot resolve these files in all application bundles because it is not designed to function in an environment with multiple class loaders. The following solutions can be used to solve the problem of resolving JSF components in a modular application

- In the *Equinox - Jetty* environment[100], the *JSF* components can be externalized to extension points where they can be registered. Bundles can provide their *JSF* components as extensions on extension points, therefore allowing them to be resolvable by *JSF* in the Equinox framework.

- In a *Spring DM* environment[101], the *JSF* components must be registered as *Spring* Beans in XML configuration files. Each bundle must contain these configuration files which declare the components located inside them.

In this case, the configuration efforts of registering *JSF* components in *Spring DM* and Equinox are the same. The main disadvantage with *Spring DM* is that, each *JSF* component class will have to be defined in an XML file. If a bundle has many *JSF* components, the number of XML files required will increase.

[97] See section 2.2
[98] See section 2
[99] See section 2.1
[100] See section 3.3.1.3
[101] See section 3.3.1

The usage of *JSF* components by resources (*facelets*[102]) in a modular application must be governed by rules. Resources should only be allowed to reference *JSF* components in their host modules and in modules which their host module is dependent on. This will allow JSF components to be consistent with module dependencies in an application, therefore avoiding null references caused by resources referencing *JSF* components located in modules that are not visible to the resource's host module.

All application modules will require access to *JSF* and *JSF-Facelets* libraries. Considerations need to be made on where these libraries will be stored without having any file redundancies. It is not memory efficient to allow each module in an application to store its own libraries. This will lead to an increase in the size of modules and a redundancy of the library files. Therefore the libraries must be stored in a centralized point where modules can be able to access their functionality. The most efficient way to fulfill this requirement in OSGi, is by assigning the *JSF* libraries to a single bundle in an application which will export the library's classes in order for other bundles to import and utilize them.

4.2. User Requirements

The user should be able to request and view resources in an application on the framework. Additionally, during the occurrence of a module swap, the user should not notice that the module swap process has been initiated. The user's interaction with the application should not be suspended but it should continue as if the application was running normally. During module swap, slight delays on the client side can be tolerated but it should not take more than a few seconds. An application will have to delay a user's response until the module swapping[103] process is done, thereafter the application can continue processing a user's request.

[102] See section 2.3
[103] See section 3.1

4.3. Administrator Requirements

The application will require an administrator to control its modules as shown in the use case diagram below:

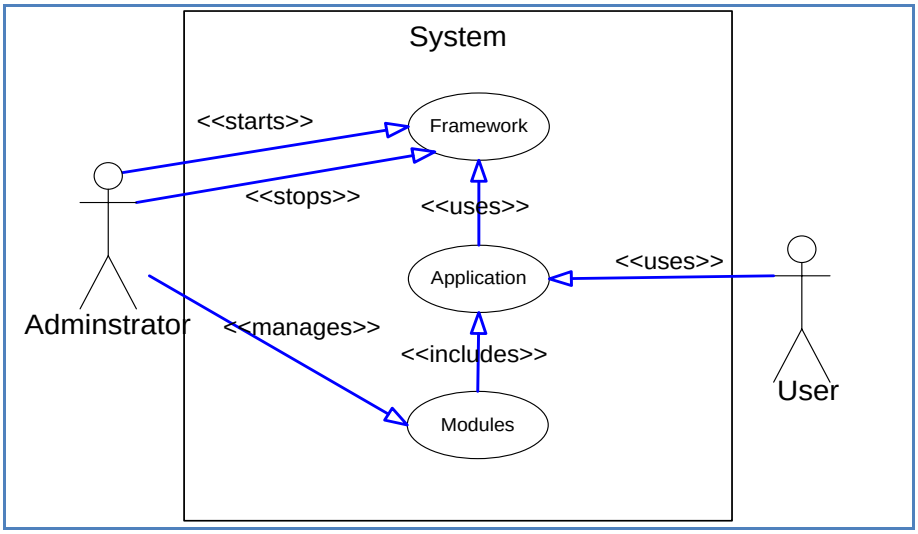


Figure 4.2: Administrator use-case diagram

The administrator should have a platform where he can manage the modules of an application. The management of the modules includes:

- Add and remove application modules
- Update modules

The management of modules can be done by a desktop application or a web based application. Modules of same name but different versions should be swappable. This swapping process should be done as fast as possible so as to avoid delays in processing user requests. Scenarios may occur such as, a user requests a resource in a module that has just been swapped out but is also available in a module that is to be swapped in. The application has to wait for the module containing the resource to be swapped in, in order to deliver the requested resource to the requesting user. Therefore the faster the execution of the swapping process, the less inconvenienced the user will be. The updating of modules is vital to applications because during updates, the inter-module references need to be updated.

The user and administrator processes described in *Figure 4.2* should occur concurrently. The following sequence diagram (*Figure 4.3*) depicts a scenario where modules are being swapped while a user interacts with an application:

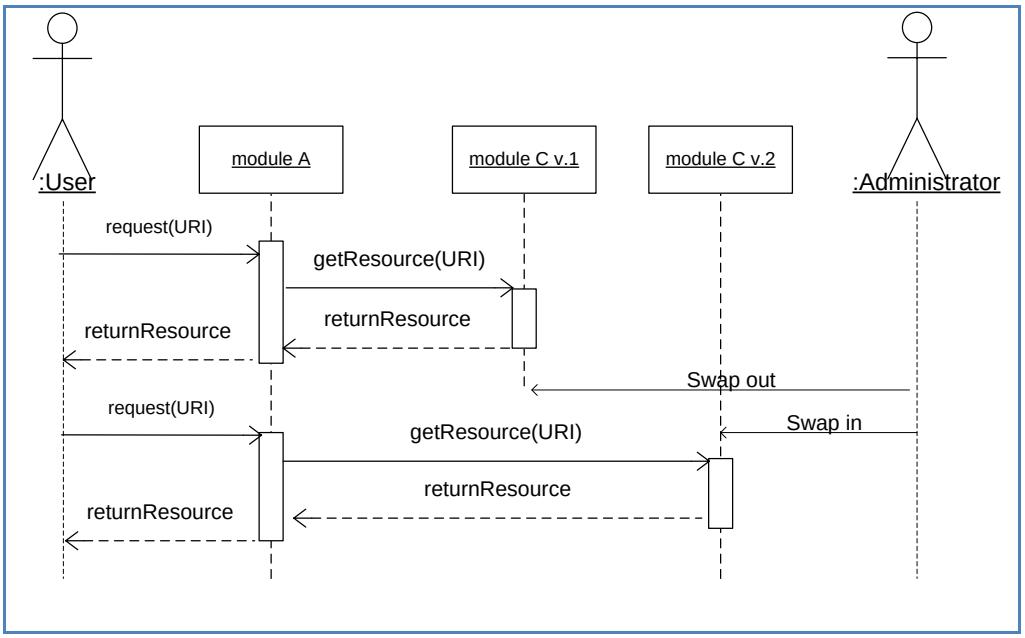


Figure 4.3: Module swapping sequence diagram

In the beginning, all application required modules should be available, in the above case, module A and C (Either version 1 or version 2 should be available but not necessarily both). The application should allow only the activation of one of the versions of module C during runtime. In *OSGi* the bundle with the highest version number is usually activated first. According to *Figure 4.3*, the user sends a resource request to the application which is received by module A. Module A then retrieves the requested resource from module C v.1. The administrator thereafter removes module C v.1 and replaces it with module C v.2, which can be considered as an extended version of module C v.1. Next time the user requests the resource that was requested earlier, it will be retrieved from module C v.2. If a module swap occurs during a resource request, the response will have to be delayed until the module containing the resource is swapped into the running application. The module swapping delay should be kept as low as possible so that users will not have to incur long delays while waiting for the application to respond to their requests.

4.4. Summary

This chapter discussed the requirements which the new framework must fulfill. According to the analysis conducted in this chapter, both *Spring DM* and *Equinox - Jetty*[104] can fulfill the stated requirements. However, based on the discussion in section 3.5, the *Equinox - Jetty* framework has more benefits than other frameworks. Furthermore, *JSF*[105] functionality can be easily integrated into the *Equinox - Jetty* framework. *JSF* in *Spring DM* requires vast amount of configurations in order to use *JSF* components (these configurations refer to the creation of *Spring* XML files). Bundles will have to be packed with additional XML files declaring *JSF* components which will increase the size of a bundle. Migration of applications in the current *Clintweb* framework to *Spring DM* will require more configuration efforts and code changes as compared to *Equinox - Jetty*. Therefore, the new framework design will be based on the *Equinox - Jetty* framework.

[104] See section 3.3.1
[105] See section 2.2

5. System Design

This chapter discusses the conceptual design principles of a framework that supports the deployment of dynamic modular applications. The term '*framework*', refers to the tools and technologies which will support the deployment of modular applications, whilst '*application*' refers to the collection of modules which will utilize the framework's services to provide functionality to users. It is the intention of this chapter to outline how the technologies presented in chapter 3 will be applied to fulfill the requirements stated in chapter 4.

The overall structure of the system is shown below:

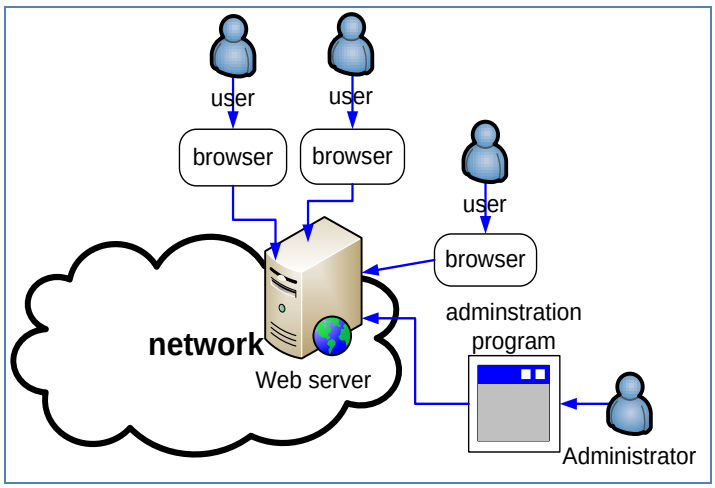


Figure 5.1: System overview

The web server will host the framework where dynamic modular applications will be deployed. Users will access applications on the web server using a standard browser. Applications will be managed by an administrator via a Client based desktop application.

The following section will outline the framework architecture for hosting modular applications. Thereafter the design of an application which fulfills the requirements stated in section 4.1 will be discussed.

5.1. Framework Architecture

As outlined in section 4.4, the new framework is based on *Equinox - Jetty* framework. *Figure 5.2* outlines the structure of the new framework:

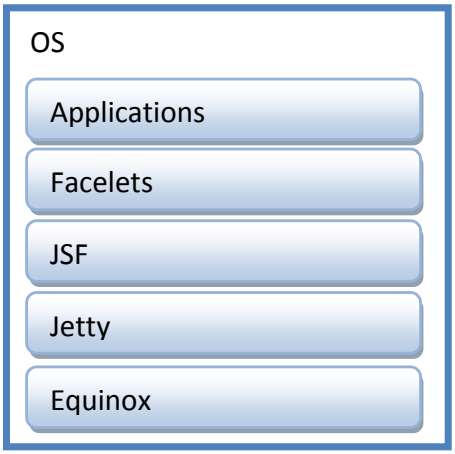


Figure 5.2: Framework architecture

Applications will be represented by a set of bundles which are deployed on Equinox[106]. The Equinox layer will provide the functionality of controlling the lifecycle of each bundle in the application layer. The Jetty web server[107] will be deployed as a bundle. It will serve to manage user sessions and security for applications. The *JSF* layer[108] will provide an environment where application bundles containing *JSF* components can be deployed. The *Facelets*[109] layer will be applied on to *JSF* in order for applications to support *JSF-Facelets* functionality.

5.2. Application Architecture

This section describes the design specifications of applications which will be deployed in the application layer of the framework described in section 5.1.

[106] See section 3.3.2.1
[107] See section 3.3.1.3
[108] See section 2.2
[109] See section 2.3

An application will consist of a set of bundles which follow the *OSGi* specification[110]. The following bundles will be mandatory for applications:

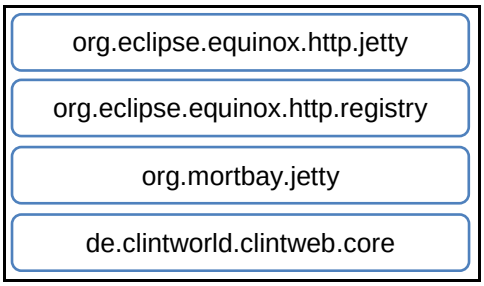


Figure 5.3: application required bundles

The `org.eclipse.Equinox.http.jetty` (referred to as *Equinox - Jetty* bundle) bundle is an Equinox bundle which instantiates the Jetty web server located in the `org.mortbay.jetty` bundle. The `org.mortbay.jetty` bundle contains the implementation classes of the Jetty web server which are split into the following three functional parts:

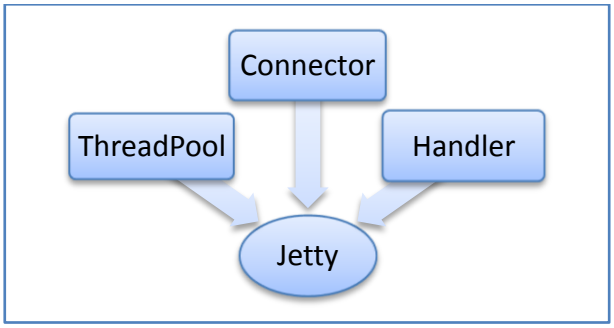


Figure 5.4: Jetty's functional parts

The *Connector* represents classes which accept HTTP connections for receiving client requests and forwards them to the `Handler` classes. A `Handler` receives an HTTP request and either services it or passes it on to other registered handlers like `Servlets`. The `ThreadPool` is a container of threads which are used by the handlers to service requests. The `ThreadPool` has a limit on the number of threads which can be assigned to a handler class. Usually when the maximum number of threads have been already assigned to handler, then no more client requests can be accepted. The maximum number of `Threads` can be manually configured to suit the framework's host server machine capabilities.

The `org.eclipse.equinox.http.registry` bundle (referred to as the Equinox registry bundle) is an Equinox bundle providing extension points for registering `Servlets`, `Filter` and

[110] See section 3.2

`HttpContext` class implementations (These classes will be explained in section 5.3). All bundles in *Figure 5.3* apart from the `de.clintworld.clintweb.core` bundle (referred to as the core bundle) are *Equinox - Jetty* implementation bundles. The `de.clintworld.clintweb.core` bundle will be the core of an application. It will provide the following functionalities to applications:

- Processing of user requests
- Registering of session events
- Resolving of *facelet*[111] resources in all application bundles
- Registration of *JSF Managed Beans*, *Navigation Rules* and *Custom tag libraries*
- Exporting of *JSF* classes to other application bundles
- Preservation and restoration of session objects after a bundle swap[112] occurs

Other application bundles will set their dependencies on the core bundle in order access the above listed functionalities.

The core bundle and other application bundles will have the following format:

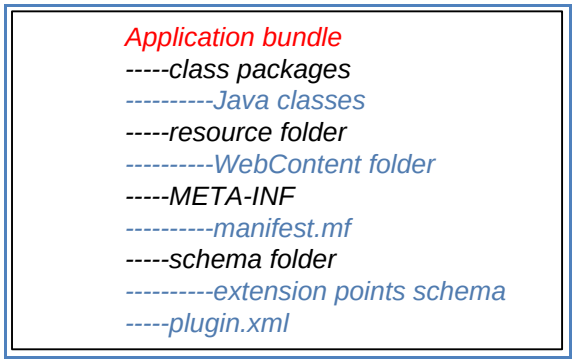


Figure 5.5: New bundle format

The *class packages* will contain bundle classes stored under various packages. Packages will group classes providing specific functionalities. The *resource folder* will contain the static resources which include *facelets*, images and cascading style sheets. The *META-INF* folder will contain the bundle's manifest file which describes a bundle's dependencies, import and export packages. The Equinox runtime will use the manifest file to resolve a bundle's dependencies when a bundle is

[111] See section 2.3
[112] See section 3.1

started. The *plugin.xml* file defines a bundle's extensions and extension points[113]. The *schema folder* stores the XML schemas[114] which provide XML vocabulary for extension points. The core bundle will contain a *faces-config.xml*[115] file where the application property resolver class[116] will be defined. No other application bundle will contain a *faces-config.xml* file.

5.3. User Requests Processing

The core bundle will be responsible for processing user requests and generating responses. The sequence diagram in *Figure 5.6* shows how requests will be routed to the core bundle as soon they are dispatched by a user:

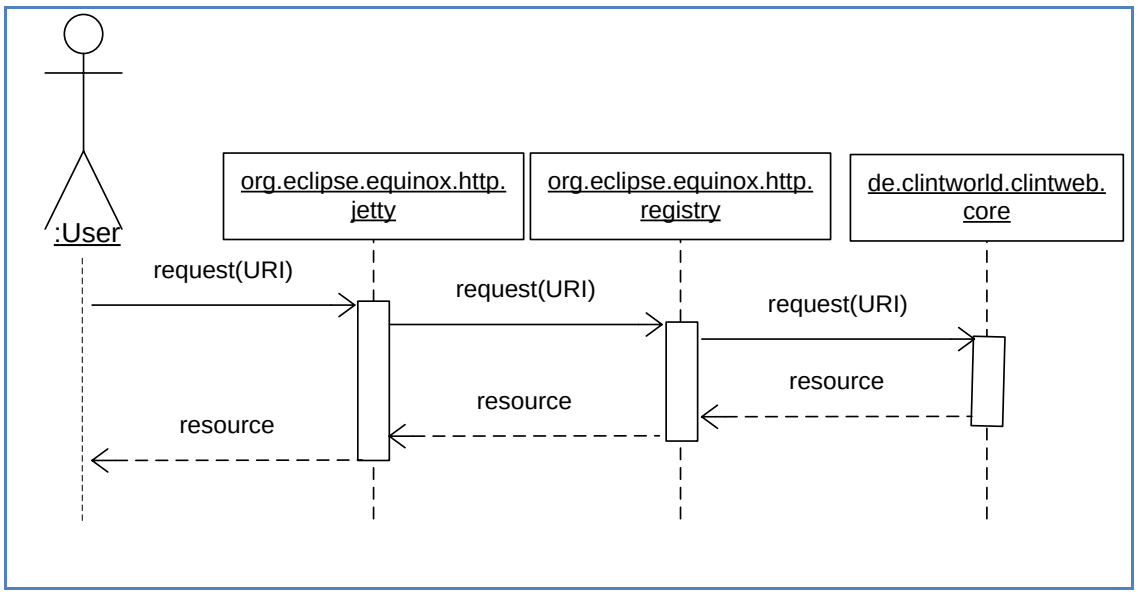


Figure 5.6: User request forwarding sequence diagram (bundle perspective)

The *Equinox - Jetty*[117] bundle will receive user requests for resources with a specified URI. The requests will be forwarded to the Equinox registry bundle which will forward them to handler classes provided by the core bundle. The core bundle will resolve the requested resource and deliver it to the user as a response. For the above delegation of user requests to function, the core bundle will have to set its dependencies on the *Equinox - Jetty* and Equinox registry

[113] See section 3.2.2.1
[114] See section 3.2.2.1
[115] See section 2.2
[116] See section 5.6.1
[117] See section 3.3.1.3

bundles[118]. Therefore, these two bundles will have to be active before the core bundle can be activated.

In order to process requests, a Java web server requires a `Servlet`[119] class which receives requests and generates responses. `Servlets` utilize an `HttpContext` object to locate and load requested resources to a client response object. In order to filter requests (e.g. for security purposes), a `Filter` class may be implemented which will intercept requests before they are processed by a `Servlet`. According to the Java SUN specification[120], a `Filter` is an object that performs filtering tasks on a resource request and/or response. Filters may be in the form of authentication filters, encryption filters, etc. `Filter` functionality will be available in the new framework but a `Filter` class will not be implemented in this thesis. The `Filter` specification in the new framework can be viewed in appendix A.

5.3.1. Servlet and HttpContext Class Registration

The core bundle will provide two `Servlet` class implementations for processing user requests. These `Servlet` classes will be registered to the Equinox registry bundle[121] for them to receive user requests. The registry bundle provides a `Servlet` extension point which follows the HTTP Service specification[122]. The core bundle will provide the following information to the `Servlet` extension point:

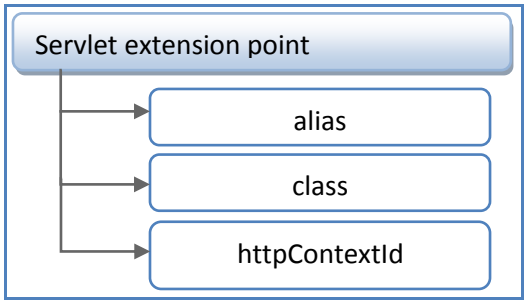


Figure 5.7: `Servlet` extension point

[118] See section 5.2
[119] See section 2.1
[120] Java SUN, Filter specification, Available at: http://Java.sun.com/products/servlet/2.3/Javadoc/Javax/servlet/Filter.html [Accessed 20 June 2010]
[121] See section 5.2
[122] *OSGi* Alliance, HTTP Service specification, Available at: http://www.*OSGi*.org/Javadoc/r4v42/org/*OSGi*/service/http/HttpService.html [Accessed 20 June 2010]

The *alias* attribute refers to the request URI pattern which will be handled by the declared `Servlet` class. A request for a resource with a URI which matches the *alias* parameter value will be forwarded to the specified `Servlet` class. The *class* attribute is the `Servlet` class implementation which must be an instance of `javax.servlet.Servlet`[123]. Finally, the *httpcontextId* is a unique identifier which is mapped to an `HttpContext` object (`HttpContext` is explained later in this section). This object is assigned to the declared `Servlet` class in order to locate and load requested resources as streams.

The Equinox registry bundle contains a `ServletManager` class which is responsible for instantiating `Servlet` objects registered as extensions on the `Servlet` extension point[124]. Jetty passes user requests to the registry bundle's `ServletManager` class, which matches a request URI to the *alias* attribute declared in `Servlet` extensions. If a match is found, the class in the extension is extracted and the request is forwarded to its `service()` method, as shown in the following class diagram (*Figure 5.8*):

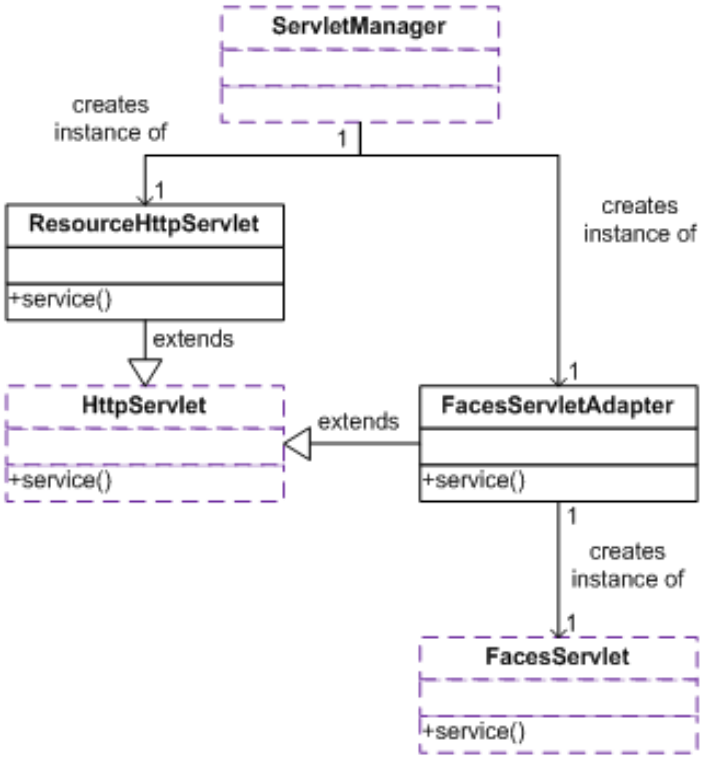


Figure 5.8: Application `Servlet` classes

[123] Java SUN, Servlet specification, Available at: http://Java.sun.com/products/Servlet/2.1/api/Javax.Servlet.Servlet.html [Accessed 20 June 2010]
[124] See section 3.2.2.1

The classes with dashed line borders are Java classes contributed by bundles or libraries; the other classes are will be created in this thesis. The `FacesServletAdapter` and `ResourceHttpServlet` classes will be located in the core bundle. The `FacesServletAdapter` will receive requests for *facelets*[125] and forward them to a `FacesServlet` object for processing. The `FacesServlet` cannot process requests for other file types because it is specifically designed for processing requests for *facelets*. Requests for other file types (e.g. images, CSS, pdf) will be forwarded to the `ResourceHttpServlet` class.

`Servlet` classes are associated with `HttpContext`[126] objects which allow them to share a `ServletContext` object. According to Jakarta Apache[127], a `ServletContext` is a class which defines a set of methods that a `Servlet` uses to communicate with its `Servlet` container (in the *Equinox - Jetty* configuration[128], Jetty provides the `Servlet` container), for example to retrieve the MIME type of a requested file. The `HttpContext` objects are important for `Servlet` classes for the following reasons[129]:

- `ServletContext` *sharing policy*: In traditional Java based web applications, `Servlets` in the same web application had the same `ServletContext`. In the Equinox framework[130], different `ServletContexts` can be assigned to different `Servlets`, so that not all `Servlets` in an application share the same `ServletContext`.

- *Resource retrieving logic*: An `HttpContext` class implementation defines logic for retrieving resources. For example, if resources are stored under a specific application path, the `HttpContext` class can be programmed to search for resources in a particular path.

- *Security validation*: The `handleSecurity()` method of the `HttpContext` class can be implemented to provide request authentication and resource authorization logic. This is useful in cases when requests do not pass security requirements, then the response object can be filled with error description messages and request processing can be stopped.

---

[125] See section 2.3

[126] *OSGi* Org, *HttpContext specification*, Available at: http://www.*OSGi*.org/Javadoc/r4v42/org/*OSGi*/service/http/HttpContext.html [Accessed 20 June 2010]

[127] Jakarta Apache, Available at: http://jakarta.apache-korea.org/cactus/api/framework-13/Javax/Servlet/ServletContext.html [Accessed 20 June 2010]

[128] See section 3.3.1.3

[129] Dynamic Java Org, *HTTP Service specification explained*, Available at: http://www.dynamicJava.org/articles/*OSGi*-compendium/http-service [Accessed 20 June 2010]

[130] See section 3.2.2.1

It must also be noted that if an `HttpContext` object is not specified, then a default one is provided by Equinox HTTP Service. As previously stated, `HttpContext` classes can be registered under an extension point provided by the Equinox registry bundle. The extension point expects the core bundle to provide the following information:

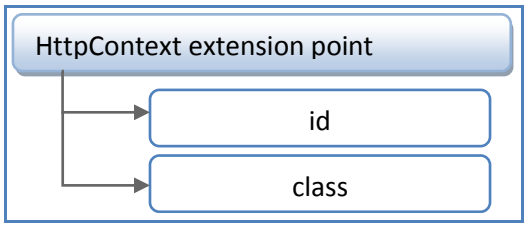


Figure 5.9: `HttpContext` extension point

The *Id* attribute is a unique identifier referring to the specified `HttpContext` class. The *class* attribute refers to a class which implements the `org.osgi.service.http.HttpContext` interface. The core bundle will provide an implementation of the `HttpContext` as specified in the following diagram (*Figure 5.10*):

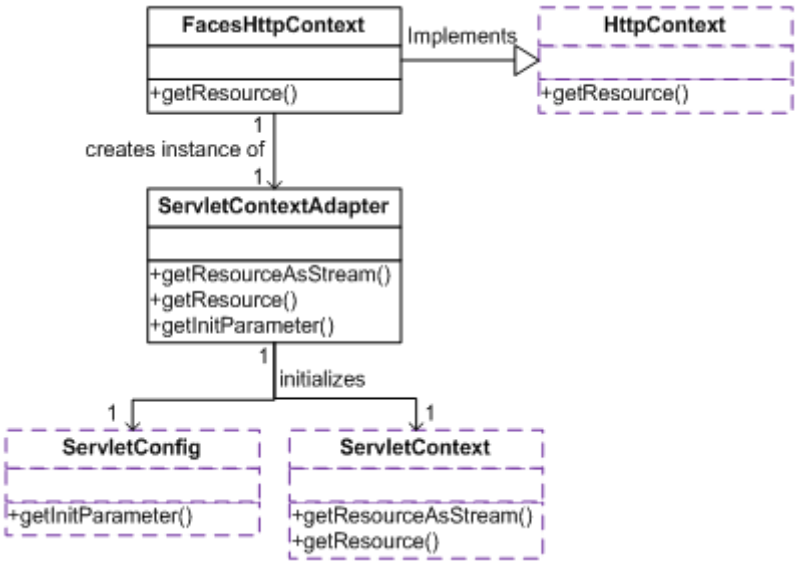


Figure 5.10: HttpContext class diagram

The `FacesHttpContext` will be assigned to the two stated `Servlet` class implementations (`FacesServletAdapter` and `ResourceHttpServlet`). These `Servlet` classes will call the `FacesHttpContext` object to resolve the URL of requested resources. The `FacesHttpContext` will delegate the task of retrieving resources to the `ServletContextAdapter`, which will use the `ServletContext` to load resources as streams and forward them to the request handling

`Servlet` classes. The `ServletConfig`[131] object is required for the web server[132] to pass information to a registered `Servlet` during initialization. It should be noted that the `ServletConfig`, `ServletContext` and `HttpContext` classes are Java classes contributed by bundles or libraries.

5.3.2. Handling User Requests

The `FacesServletAdapter` will receive requests for *facelets*[133] and load the requested *facelet* view[134] by:

- building a component tree of *JSF* UI components declared on the requested *facelet* page
- applying the appropriate values to the *JSF* components
- rendering a response to the client

Figure 5.11 shows the classes involved in carrying out the above discussed process:

[131] Jakarta Apache, Available at: http://jakarta.apache-korea.org/cactus/api/framework-13/Javax/Servlet/ServletConfig.html [Accessed 20 June 2010]
[132] See section 2.1
[133] See section 2.3
[134] See section 2.3

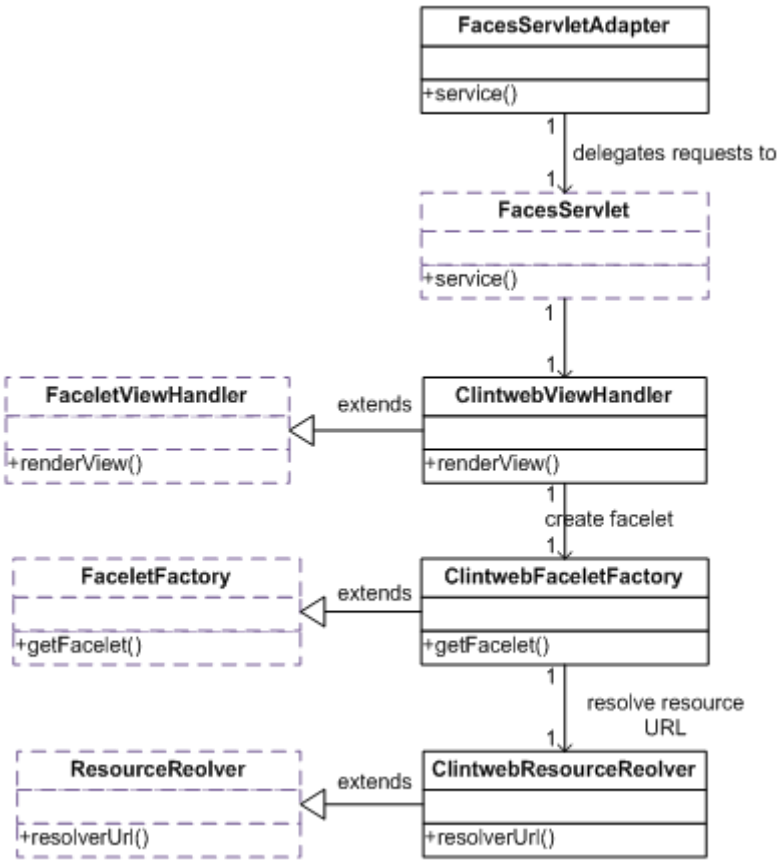


Figure 5.11: User request forwarding class diagram

The `FacesServlet` will delegate the task of rendering a *facelet* view to the `ClintwebViewHandler`[135]. The `ClintwebViewHandler` is responsible for rendering a response to a request by calling a `ClintwebFaceletFactory`[136] object which generates the view for a given view ID (the view ID is the URI of a *facelet*). The `ClintwebFaceletFactory` will attempt to associate the requested view to a *facelet* using the `ClintwebResourceResolver`, which will resolve URLs of *facelets* located in application bundles. The URL (*Universal Resource Locator*) is a URI that specifies where a resource is located whilst the URI (*Universal Resource Identifier*) specifies the ID of a resource

[135] FaceletViewHandler is an extension of the ViewHandler class, which is the pluggablity mechanism for allowing applications using the *JSF* specification to provide their own handling of the activities in the *Render Response* and *Restore View* phases of the request processing lifecycle. See: http://www.docjar.com/docs/api/Javax/faces/application/ViewHandler.html [Accessed 20 July 2010]

[136] The *Clintweb*FaceletFactory extends FaceletFactory, see specification: http://www.docjar.com/docs/api/com/sun/*Facelets*/FaceletFactory.html [Accessed 20 July 2010]

5.4. Resolving *Facelet* Resources

Users will interact with *facelets* in order to receive services from an application. To include *JSF-Facelets*[137] into JSF in *Equinox - Jetty*[138], the `Servlet` extension[139] which registers the `FacesServletAdapter` class will be configured with *JSF-Facelets* initialization parameters. These parameters will associate *JSF* views (the view is the page that is rendered to the user) with XHTML files.

The process of resolving resource URLs in an application will be done by the `ClintwebResourceResolver` class. Applications will consist of more than one bundle; *facelet* resources in all application bundles will have to be resolvable by the core bundle during runtime. The core bundle cannot directly reference resources in other bundles; resource access between bundles can only be done via specified interfaces. Therefore an extension point[140] for resolving resource URLs[141] in application bundles will be implemented in the core bundle. This extension point will require the following information from bundles willing to make their resources available to an application:

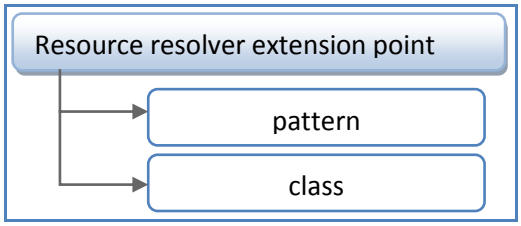


Figure 5.12: Resource resolver extension point

The *pattern* attribute refers to the unique URI associated with a *facelet* resource. The *class* attribute is an implementation of an interface provided by the core bundle[142] which specifies a function for resolving a URL for a requested resource, as shown below:

[137] See section 2.3
[138] See section 3.3.1.3
[139] See section 5.3.1
[140] See section 3.2.2.1
[141] See section 5.35.3.2
[142] See section 5.2

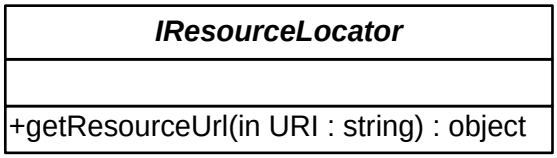


Figure 5.13:Resource locator interface

Bundles will provide implementations of the above interface, where the declared method will specify logic for locating resources and generating their URLs. The method will receive a URI as an input parameter and it will output an object of type URL, which will contain a valid URL of the resource matching the URI. The core bundle will use the custom implementations of the above stated interface provided by application bundles to resolve a requested resource's URL.

By using extension points for the resolving resources process, an application is able to adapt to dynamic changes within the runtime environment. New *facelets* will be registered at the resource extension point; therefore making it possible for the core bundle to resolve resources upon request. Bundles will be able to register multiple resources (*facelets*). However, the resource's URI must be unique within an application. This can be ensured by using a unique path prefix for *facelets*, e.g. */db/* for database related pages.

5.5. Application Session Listeners

According to Java SUN specification[143], a session listener is a class which set to be notified when changes occur to active sessions (e.g. Session started or destroyed). A listener must be registered to the web container in order to receive session event notifications. Session listeners are required by applications when certain processes have to be conducted after a session event occurs. In the new framework, session listeners are required in the session objects serialization process stated in section 5.7. The core bundle will contain one session listener class implementation with the following specification:

[143] Java SUN, HttpSessionListener, Available at: http://Java.sun.com/j2ee/sdk_1.3/techdocs/api/Javax/Servlet/http/HttpSessionListener.html [Accessed 20 July 2010]

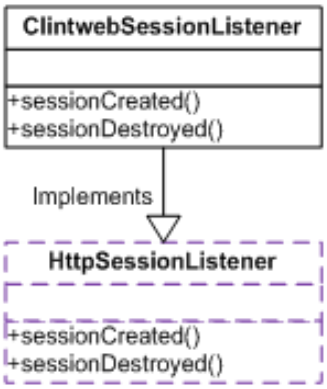


Figure 5.14: Session listeners class diagram

The `ClintwebSessionListener` will forward session events to listener classes registered on a listener class extension point[144]. This extension point will only contain one attribute, which specifies a class that implements the `HttpSessionListener` interface. Bundles which require a session event to initiate a specified process will have to provide a listener class implementation on the listener extension point.

5.6. *JSF* Functionality

JSF[145] is required to generate the UI components on an application's frontend. The *JSF* and *JSF-Facelets* libraries will be stored in the core bundle, making the core bundle the access point of application bundles to *JSF* functionality. Since the core bundle will be handling most of the *JSF* related processing, it is not efficient to store multiple instances of *JSF* libraries in other bundles. The core bundle will export some *JSF* packages to allow other bundles to utilize the *JSF* functionality.

In this new architecture, the rendering and processing of *JSF* components will be centralized to the core bundle. The core bundle will collect and instantiate *JSF* related components (e.g. *Managed Beans*, *Navigation Rules*, *Custom tag libraries*) from other bundles using extension points. Bundles in an application will use the extension points provided by the core bundle to register their own *JSF* components.

[144] See section 3.2.2.1
[145] See section 2.2 and 2.3

5.6.1. *Managed Beans* registration

JSF contains a resolver class, which is responsible for receiving *Managed Beans* requests and creating *Managed Bean* objects by calling a *JSF* `RuntimeConfig` object. The `RuntimeConfig` class handles the creation of *Managed Beans* and *Navigation Rules* from the *faces-config* XML. In order to externalize the registration of *JSF* components to an extension point, a custom implementation of the *Managed Bean* resolver class and `RuntimeConfig` has to be created. In *OSGi* based frameworks, the default *JSF* mechanism would not find *Managed Bean* classes in application bundles other than the core bundle, because the core bundle's class loader cannot directly access classes in other bundles[146]. Therefore a class called, `ClintwebManagedBeanResolver` (the *Bean* resolver class implementation) will be implemented, which will receive *Managed Bean* requests and delegate the creation of *Managed Bean* objects to a class called `OSGiRuntimeConfig` as shown in the class diagram below:

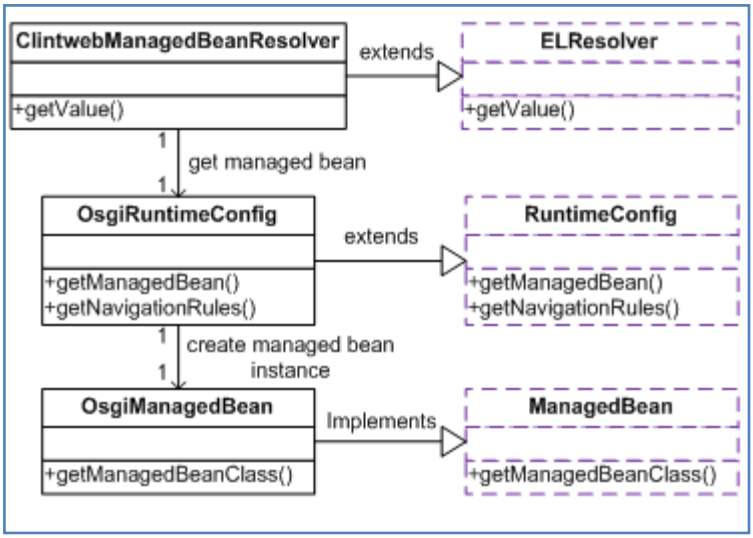


Figure 5.15: Managed bean processing class diagram

Note that the `ClintwebManagedBeanResolver` object is called only when a *facelet* containing a reference to a *Managed Bean* is requested. The `OSGiRuntimeConfig` extends the `RuntimeConfig` class, which will contain the logic of resolving *Managed Beans* and *Navigation Rules* from extension points. The `OSGiRuntimeConfig` will create *Managed Bean* objects which are instances of `OSGiManagedBean`. The `OSGiManagedBean` class will be able to load *Managed*

[146] See section 3.2

Bean classes from their host bundles. The *Managed Bean* extension point will have the following specification:

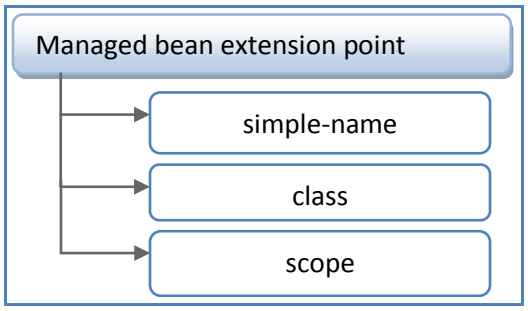


Figure 5.16: Managed beans extension point

The *simple-name* of the *Managed Bean* is the EL syntax *facelets* use to call a *Managed Bean's* properties. The *scope* property is of type String and it is for declaration of the lifecycle of a *Managed Bean*. The *class* attribute refers to a *Managed Bean* class implementation. Each bundle will be able to register multiple unique *Managed Beans*.

In order to fully externalize the *Managed Bean* functionality to an extension point, additional *JSF* classes have to be customized. In a normal *JSF* setting, a *Managed Bean* class instance is loaded using a `DefaultLifecycleProviderFactory`[147] object when it is requested for the first time in an application's lifecycle. In order to load *Managed Bean* classes from application bundles in an *OSGi* environment, a custom implementation of the `DefaultLifecycleProviderFactory` will be created according to the following class diagram (*Figure 5.17*):

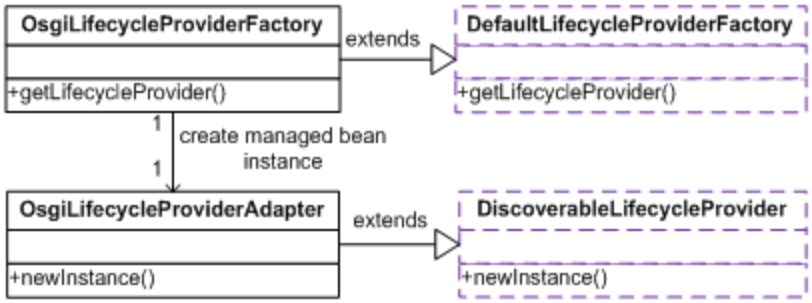


Figure 5.17: Loading Managed Beans class diagram

[147] Apache MyFaces, DefaultLifeCycleProviderFactory specification, Available at: http://myfaces.apache.org/core12/myfaces-impl/apidocs/org/apache/myfaces/config/annotation/DefaultLifecycleProviderFactory.html [Accessed 16 June 2010]

The `OSGiLifecycleProviderFactory` will create an instance of the `OSGiLifecycleProviderAdapter` which creates an instance of a *Managed Bean* class registered in an extension point.

5.6.2. *Navigation Rules* registration

Navigation Rules[148] are important for managing navigation between resources in a *JSF* based web application. The new framework will support the registration of *Navigation Rules* by application bundles via an extension point provided by the core bundle. The core bundle will instantiate and process *Navigation Rules*. *Navigation Rules* will be split into two parts, the *from-view* and the *navigation case*. The *from-view* property of type String is the URI of the source page of the *Navigation Rule* (the page where the navigation link will be placed). The *navigation case* is the target page of the *Navigation Rule* (the page the *Navigation Rule* will lead to). *Figure 5.18* shows the specification of the *Navigation Rules* extension point:

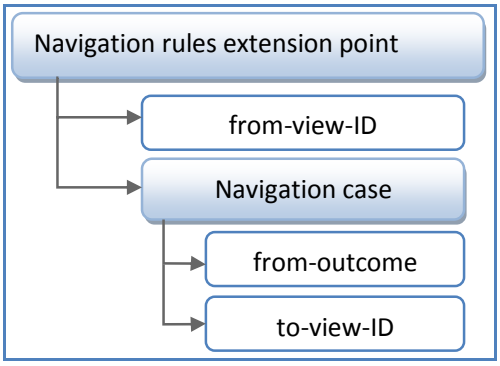


Figure 5.18: Navigation Rules extension point

The *from-view-ID* is an optional property; *Navigation Rules* will only be required to specify navigation cases. A *Navigation Rule* will allow the declaration of more than one navigation case. The *from-outcome* and *to-view-id* are both of type String.

The `OSGiRuntimeConfig`[149] class is responsible for instantiating *Navigation Rules* registered on the *Navigation Rule* extension point. The `OSGiRuntimeConfig` class will create *JSF* `NavigationRule` objects for each provided extension and for each `NavigationRule` object, *JSF* `NavigationCase` objects will be created based on the information provided at the extension

[148] See section 2.2
[149] See Figure 5.15*: Managed bean processing class diagram*

point. Thereafter all created `NavigationRule` objects will be stored in a list for the application to access them when they are requested during the lifetime of an application.

Navigation Rules can be activated by returning a String at the end of a *Managed Bean* function or by specifying a value of type String as an action output of a *JSF* UI component. The returned String value will be detected by *JSF* which will prompt execution of a *Navigation Rule* based on the String value.

5.6.3. Custom tag libraries

Custom tag libraries will allow custom *JSF* UI components to be used on *facelets*. In order to support custom tag libraries, the core bundle will contain an XML file, where the qualified name of a tag library class will be defined. The `ClintwebTagLibrary` class will contain tag names of custom UI components which can be used on *facelets*. For *facelets* to reference custom components, `ClintwebTagLibrary` class will define a namespace which *facelets* must declare for them to use the custom components defined in the `ClintwebTagLibrary`. The specification of the `ClintwebTagLibrary` is shown below:

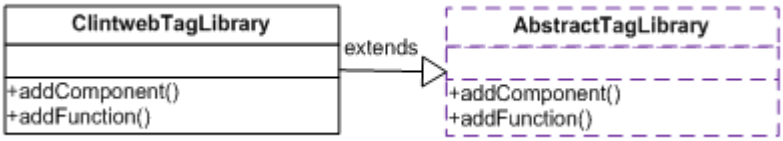


Figure 5.19: Tag library classes

The `ClintwebTagLibrary` will derive its functions from *JSF*'s `AbstractTagLibrary`[150] class. The `addComponent()` method will add a custom UI component, its renderer class and a reference tag name to the tag library. UI Components require a renderer class in order for them to be rendered on a *facelet*. A renderer class is responsible for encoding and decoding components. Encoding refers to displaying the component so that it is visible to the user and decoding means translation of the user's input to a component value[151]. The `addFunction()` method will add a method specified by a class and a function reference name to the tag library. The method will define the parameter type which should be supplied when it is called.

[150] See AbstractTagLibrary specification, Available at: http://myfaces.apache.org/core20/myfaces-impl/apidocs/org/apache/myfaces/view/*Facelets*/tag/AbstractTagLibrary.html [Accessed 20 July 2010]
[151] RoseIndia, *JSF* Renderers, Available at: http://www.roseindia.net/*JSF*/*JSF*renderers.shtml [Accessed on 20 July 2010]

The core bundle will contain a tag library extension point where application bundles will register tag library classes which reference custom UI components located within the respective bundles. The registered tag library classes will have to implement the `com.sun.facelets.tag.TagLibrary` interface. The tag library classes will define tag names which reference UI components located in the bundles extending the tag library extension point. The usage of custom components will be governed by the following rules:

- A bundle's custom components will only be valid on *facelets* belonging to the custom component's host bundle and its child bundles. So that when a component's host bundle is uninstalled, other bundles that are not dependent on the components host bundle will not be affected because they will not be utilizing its components.

- The core bundle is allowed to provide custom components which are valid application wide because all application bundles will have dependencies on the core bundle and it will always be active during the lifecycle of an application.

In order for bundles to register their custom UI components to *JSF*, an extension point for registering the component classes will be implemented. This extension point will define an attribute named 'class' which requests extending bundles to supply a class instance of `javax.faces.component.UIComponent`. Therefore when a custom component located outside the core bundle is referenced by a *facelet*, JSF will search for the component on the component extension point and instantiate it.

5.7. Session State Preservation

In *JSF*[152], the state of an application is stored in its *Managed Beans*. When a bundle is swapped out, its *Managed Beans* have to be preserved and restored when another version of the swapped out bundle is deployed in an application. The core bundle will handle the preservation of *Managed Beans*. In *OSGi* implementations, when a bundle is updated or uninstalled, the user inputted data contained in the bundle is lost and cannot be recovered unless programming logic for preserving the bundle state is implemented. In order to preserve a bundle's state, the core

[152] See section 2.2

bundle must detect bundles initiating *OSGi UPDATE* or *UNINSTALL* events[153] and then persistently preserve their *Managed Beans*. The *Managed Bean* preservation process is split into two parts:

- Saving a *Managed Bean*'s state
- Restoring a *Managed Bean* into an application

5.7.1. Saving a *Managed Bean*'s State

The recommended method for preserving Java objects is by serialization. In Java, serialization of an object is enabled by a class implementing the `Java.io.Serializable` interface. This interface allows an object's state to be stored as a byte stream.

Before serialization can be done, the core bundle will have to detect the bundles whose *Managed Beans* have to be serialized. In order for this functionality to be implemented, a bundle event listener is required to notify the core bundle of events occurring in an application environment. *OSGi* provides a bundle event listener interface which when implemented, receives all bundle events when they occur. The core bundle will contain a bundle event listener class which will be notified when an *OSGi* event of type *UNINSTALL* or *UPDATE.* The core bundle will thereafter extract the *Managed Beans* belonging to the event initiator bundle and preserve their state as shown in the following sequence diagram (*Figure 5.20*):

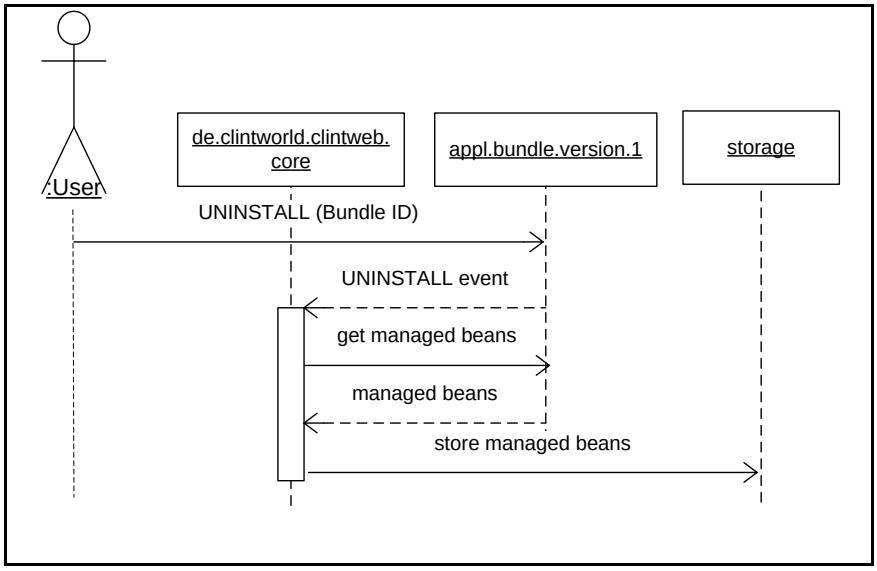


Figure 5.20: Object serialization sequence diagram

[153] See section 3.2

The 'storage' object in the above figure refers to either the disk or the application memory (RAM). The storage destination will be set on a configuration page. To further refine the above sequence diagram to a class level, *Figure 5.21* shows how the *core* bundle will handle the process of *Managed Bean* serialization:

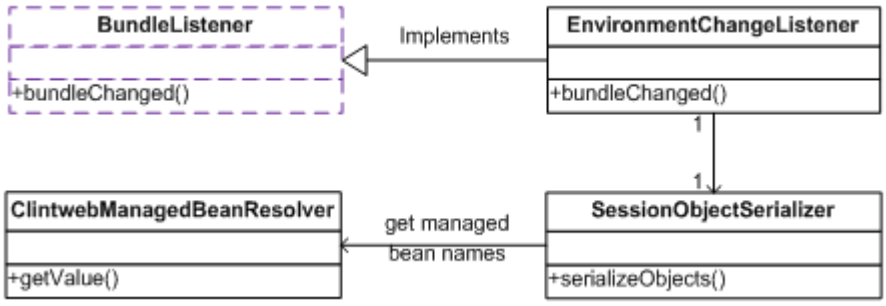


Figure 5.21: Serialization class diagram

The `EnvironmentChangeListener` class will listen for bundle events and forward the qualified name of a bundle which requires serialization of its *Managed Beans* to the `SessionObjectSerializer` class. This class will request a list of *Managed Bean simple-names* belonging to the bundle which caused the *UNINSTALL* or *UPDATE* event from the `ClintwebManagedBeanResolver`. Thereafter the `SessionObjectSerializer` will retrieve the *Managed Bean* objects for each active user from the application session map[154], serialize them and store them in the application memory or write them onto the disk. The serialized *Managed Beans* will thereafter be deleted from an application's session map[155]. On the application memory, the serialized *Managed Beans* will be stored in a hash map, where the user session ID will be the key, and an additional hash map will be the value. The hash map which is referenced by a session ID, will map a *Managed Bean simple-name* (key) to a *Managed Bean* object (value). The other option is to store the serialized objects on the disk under a specified path. The base path (the folder where the session objects will be stored) will be hard coded into the core bundle and it will be configurable during application runtime using a configuration page. On the file system, serialized objects will be stored in folders named according to active user sessions IDs. The serialized *Managed Bean* objects will be stored in session ID folders according to the session ID which they belong to. The serialized objects will have .SER extension and they will be named according to their *Managed Bean simple-name* property.

[154] A session map is an object which maps session IDs to objects belonging to a session ID. In web applications each user is assigned a session ID and the objects created during the lifetime of a session are stored in the session map under the user session ID.
[155] See section 5.7.1

5.7.2. *Managed Bean* Restoration

After a bundle is replaced with another bundle of a different version, the serialized session *Managed Beans* will be restored from their storage location and placed back into a running application, in accordance to the following sequence diagram (*Figure 5.22*):

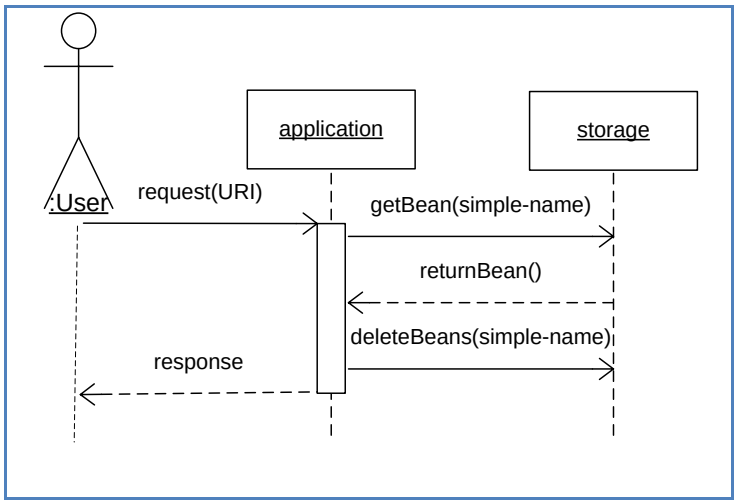


Figure 5.22: Managed Bean restoration sequence diagram

According to the sequence diagram above, serialized *Managed Beans* will only be restored when a request for a *facelet*[156] page which references a *Managed Bean* is received by the application. During the serialization process, a *Managed Bean* will be deleted from the application session map after it is serialized. The next time a *Managed Bean* is requested, *JSF* will not be able to find it in the session map. Therefore a new instance of a *Managed Bean* will have to be created once again. The `OSGiLifecycleProviderAdapter` will be called by *JSF* to create a new instance of a *Managed Bean* from the bean extension point. The `OSGiLifecycleProviderAdapter` will initially check if the requested *Managed Bean* has been previously serialized by calling the class responsible for de-seriliazation, as shown in the following diagram (*Figure 5.23*):

[156] See section 2.3

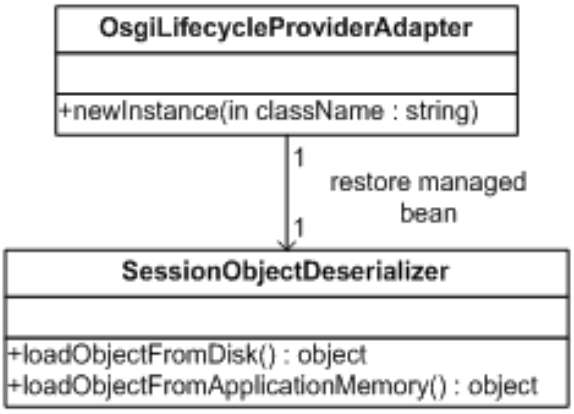


Figure 5.23: De-serializer classes

The de-serializer will receive the *simple-name* of the *Managed Bean* object. Using the session ID of the user where the *Managed Bean* request originated from and the *simple-name* property of requested *Managed Bean*, the de-serializer will attempt to retrieve the serialized *Managed Bean* from its storage location (the disk or application memory). If it is found, it will be de-serialized and then forwarded to *JSF* as the new *Managed Bean* class containing its original session variable values. Otherwise if no serialized *Managed Bean* is found then a new instance of the *Managed Bean* will be created from the *Managed Bean* extension point and forwarded to *JSF*. Note that the `java.io.Serializable` interface[157] will manage the migration of *Managed Bean* classes between different versions. This is necessary when a serialized *Managed Bean* with an old data structure has to be converted to a *Managed Bean* object with a different data structure during de-serialization.

It should also be noted that during a bundle swap, the user should not notice changes taking place in an application. An application will not be temporarily suspended but its client response will be delayed until the bundle swap[158] process is complete (when a swapped out bundle is replaced by a new bundle).

[157] See section 3.4
[158] See section 3.1

5.8. Client and Administrator Test Application Design

A test application which demonstrates the design specifications discussed this chapter will be created. The application will contain two bundles in addition to the bundles outlined in section 5.2. One of the two bundles will have two versions with different implementations. Versions of the same bundle will be replaceable during runtime. Each bundle will offer *facelets* (XHTML files containing JSF components) which will demonstrate *JSF*[159] functionality according to the design specifications in section 5.4. The layout of the *facelets* resources will be as follows:

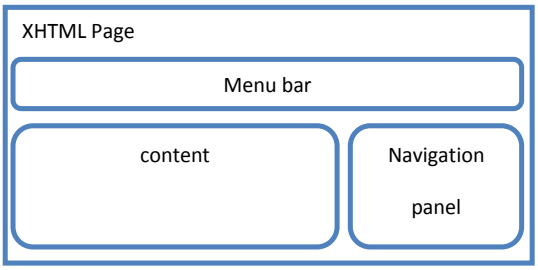


Figure 5.24: Test application UI layout

The menu bar will contain links to home page of the application, and to a properties page contained in the core bundle. The content part will consist of text, images, *JSF* custom components, *Navigation Rules* components and information contained in the application's *Managed Beans*. The navigation panel will contain links to pages in the application. The navigation panel will only be available on *facelets* in one bundle version.

There already exist applications for managing an Equinox framework (e.g. *Prosyst mB-SDK web admin tool*[160]), which can later be used for management of the framework. For test purposes, an administrator application will be created which will provide the basic administrator requirements defined in section 4.3. The application will have the following layout:

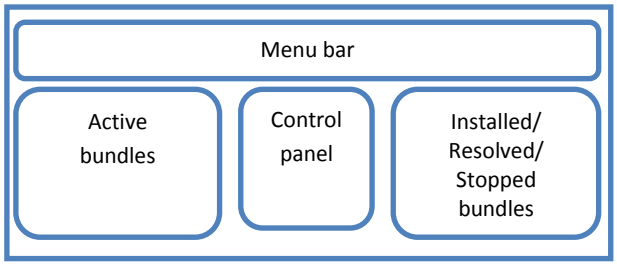


Figure 5.25: Administrator application layout

[159] See section 2.2
[160] *Prosyst mB-SDK*. Available at: http://dz.prosyst.com/devzone/Home/ [Accessed 28 July 2010]

The menu bar will contain options for starting/stopping the application. The Active bundles panel will display the qualified name and version of active bundles in the application. The control panel will provide buttons for starting, stopping, updating and swapping bundles. The panel on the right side of *Figure 5.25* will display bundles in the application context which are not active but installed. This administrator application will connect to the Equinox framework[161] via the Java socket connection[162]. The framework will be running on a server which will be listening for connections from the administrator application. All administrator requests will be forwarded to the server and the server will notify the administrator application on the current state of the application bundles.

5.9. Design Constraints

In this deployment architecture, the core bundle is the center of an application. It was set to be the central provider for services for all application bundles so that each bundle will only have to set its dependencies on one bundle in order to access the essential framework functionality. The core bundle must always be active and any changes to its runtime state will affect all other application bundles. Failure of operation of the core bundle could lead to unprocessed user requests, a collapse of the framework's *JSF* functionality and other application bundles will also not be resolvable. Therefore the core bundle must be made stable and reliable.

No cyclic dependencies can be set between bundles. Therefore there can be no cyclic usage of extension points[163], as shown in the figure below:

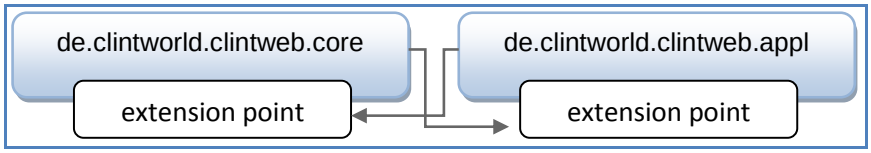


Figure 5.26: Cyclic dependencies

[161] See section 3.3.1.3
[162] Java Sockets, Available at: http://www.javaworld.com/jw-12-1996/jw-12-sockets.html [Accessed 10 August 2010]
[163] See section 3.2.2.1

Bundles cannot extend extension points and use services bi-directionally. For a bundle to use an extension point or a service, it must set the bundle providing the extension point and service as a dependency. The *OSGi* specification states that two bundles are not allowed to have dependencies on each other. If two bundles depend on each other, then there is no way of determining which bundle should be resolved first. This emphasizes the fact that developers should clearly plan dependencies between bundles so as to avoid complications.

Another limitation is that when bundles are swapped or uninstalled, Equinox does not clean up after the bundles. For example, if a database connection is opened by a bundle and then the bundle is swapped out, Equinox will not close the database connection. Such problems can be severe to the framework. They can cause memory leaks which can cripple the server. Therefore bundles must be created while keeping in mind that they must clean up after themselves. A bundle's `Activator` class contains a `stop()` method where the cleaning up code can be placed, so that when a bundle is stopped, the clean up process can be executed.

In term of migration of the current *Clintweb* framework[164] to the new framework, some of the current *Clintweb* application classes will have to be changed to adapt to this architecture. For example, JSF components will have to be externalized to extension points. Extension points will also have to be implemented in some classes in order to take full advantage of application modularity. These changes are a requirement which must be fulfilled in order for current *Clintweb* applications to function in the new framework.

The design principles described in this chapter are the preferred solution for *Clintweb* because they provide minimal migration efforts as compared to other solutions and at the same time they assure the fulfillment of the requirements stated in chapter 4.

The next chapter discusses how the design principles stated in this chapter were realized.

[164] See chapter 2

6. System Implementation

This chapter discusses how the design specifications stated in the previous chapter were realized. The required development software and tools will be listed followed by a detailed explanation of how they were used in fulfilling the requirements stated in chapter 4.

6.1. Development Tools

For development, the *Eclipse Java EE IDE for Web Developers* was used which requires a minimum of Java JRE version 5. This version of Eclipse contains the necessary tools for building Java based web applications. It furthermore provides a graphical HTML/JSP/*JSF* editor, database management tools and support for popular application servers[165]. The mentioned Eclipse version needs to be equipped with the Web Tools Platform[166] (WTP). The WTP contains a *JSF* Tools project which provides an extensible tooling infrastructure for building *JSF*-based, web-enabled applications. The *JSF* project is required because the new framework must support *JSF* functionality.

The mentioned version of Eclipse provides an environment for developing Equinox[167] based plug-ins (bundles in Equinox) which are created from an Eclipse plug-in project.

For setting up the *Equinox - Jetty*[168] architecture, the following bundles were downloaded from Eclipse Org Equinox site[169]:

- *org.eclipse.equinox.http.jetty* version 1.1.0: Creates a bridge from Equinox to Jetty
- *org.eclispse.equinox.http.registry* version 1.1.0: Enables Equinox to registers web descriptor components to the web server
- *org.mortbay.jetty* version 5.1.11: The Jetty web server

[165] Eclipse org, Eclipse IDE for Java developers, Available at: http://www.eclipse.org/downloads/moreinfo/jee.php [Accessed 16 June 2010]
[166] Web Tools Platform, Available at: http://www.eclipse.org/projects/project_summary.php?projectid=webtools [Accessed 16 June 2010]
[167] See section 3.2.2.1
[168] See section 3.3.1.3
[169] Equinox – Jetty bundles, Available at: http://www.eclipse.org/Equinox/server/http_in_Equinox.php [Accessed 16 June 2010]

The versions of the above listed bundles must be compatible with each other because different versions have different implementations which may be incompatible with one another. The versions stated above are fully compatible and can be imported into an Eclipse workspace.

For setting up the *JSF* and *JSF-Facelets*[170] environment, the following libraries were utilized[171]:

- *myfaces-api-1.2.2.jar*: Implementation of the *JSF* classes defined in the *JSF* specification
- *myfaces-impl-1.2.2.jar*: the "invisible" classes that are needed for a *JSF* implementation but are not part of the public API.
- *JSF-Facelets.jar*: Implementation of the *JSF-Facelets* classes
- *tomahawk-1.1.1.jar*: Implementation of *JSF* components and utilities that can be used with any *JSF* implementation.
- *tomahawk-facelets-taglibs.jar*: contains *JSF* tag libraries.

For setting up the runtime configuration for Equinox, additional bundles must be included in the runtime environment. A list of these bundles has been specified in the appendix B. Most of the required bundles are available in the Eclipse IDE but few of them were found by extensively searching the internet.

6.2. Framework and Application Architecture

Figure 6.1 shows the structure of the developed framework and test application. The arrows in the figure outline the dependencies between bundles. The arrow direction means '*requires*'; for example, the `org.eclipse.equinox.http.jetty` requires the `org.mortbay.jetty` for it to be deployed.

[170] See section 2.2 and 2.3

[171] Apache MyFaces, Available at: http://myfaces.apache.org/core20/index.html [Accessed 16 June 2010]

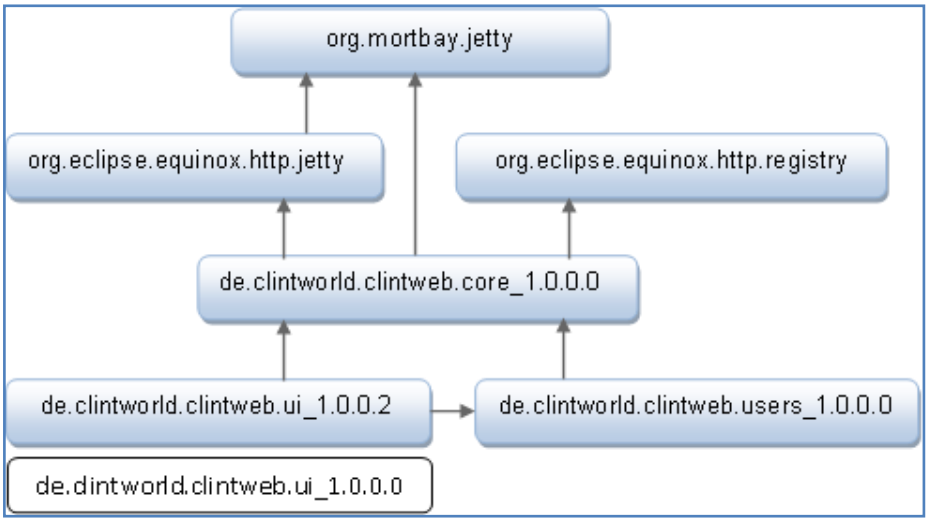


Figure 6.1: Application architecture

The framework contains the above depicted bundles and the test application is represented by the *UI* bundle. The *UI* bundle has two versions which are distinguishable by their minor version number. There are implementation differences between the different bundle versions. The `1.0.0.2` bundle versions contain more *facelets* and extended *Managed Bean*[172] classes as compared to the `1.0.0.0` versions. When Equinox is started, the bundle with the highest version number is automatically activated and smaller bundle versions remain in the *OSGi INSTALLED*[173] state. The *core* bundle and *Equinox* bundles will remain active for the lifetime of the application because they are required for the application to function and other bundles have their dependencies set on them. As seen in figure above, the *UI* and *Users* bundle are dependent on the core bundle and the *UI* is also dependent on the *User* bundle. The *UI* bundle imports packages from the *Users* bundle, therefore when the *Users* bundle is updated; the *UI* imported packages must also be updated.

The core bundle is dependent on the Jetty bundle because it provides it with the necessary classes for accessing user session information. The *Equinox registry* bundle provides the core bundle with the mechanism for registration of `Servlet`[174] class implementations which enables the core bundle to receive user requests.

[172] See section 112.2
[173] See section 3.2
[174] See section 5.3

6.2.1. Servlet classes initialization

As stated in section 5.3.1, the *core* bundle provides two `Servlet`[175] class implementations which are registered as extensions[176] on the Equinox registry bundle `Servlet` extension point. When processing requests, the Equinox registry bundle contains a `ServletManager`[177] class which forwards requests to the `service()` method of either one of the two `Servlet` classes registered as extensions. The figure below shows how the `FacesServletAdapter` class was registered on the `Servlet` extension point:

```
<extension point="org.eclipse.equinox.http.registry.servlets">
   <servlet
         alias="/*.jsf"
         class="de.clintworld.clintweb.servlets.FacesServletAdapter"
         httpcontextId="de.clintworld.clintweb.facesHttpContext"
         load-on-startup="false">
```

Figure 6.2: FacesServletAdapter registration

The `point` attribute specifies the ID of the `Servlet` extension point which is based in the Equinox registry bundle. The `alias` attribute was set to '`/*.jsf`' so that requests for any resource with a `.jsf` extension are forwarded to the `FacesServletAdapter` class which is specified in the `class` clause. The `load-on-startup` property declares if the `Servlet` class should be loaded when the web server is started. By setting it to `false`, the `Servlet` is only loaded when a request is received by the application. The `httpcontextId` refers to the `FacesHttpContext` object which is used by the `FacesServletAdapter` to resolve *JSF* tag libraries[178] and the *faces-config.xml* file[179] in the core bundle.

When the application is started, the `FacesServletAdapter` calls an `initialize()` function which is required to perform the following initialization tasks:

- Replacement of *JSF*'s `DefaultLifecycleProviderFactory`[180] in the system properties with the `OSGiLifecycleProviderFactory` class, which is responsible for creating new *Managed Bean* instances when they are requested.

[175] See section 2.1
[176] See section 3.2.2.1
[177] See section 5.3.1
[178] See section 5.6.3
[179] See section 2.2
[180] See section 5.6.1

- Creation of a `StartupServletContextListener`[181] object which initializes the *JSF* system.
- Creation of a `ServletConfig`[182] object which is required by the Jetty `Servlet` container to pass initialization parameters to a `Servlet` during initialization.
- Creation of a `ServletContextAdapter` object which is required by the application `Servlets` to load requested resources into a `Servlet` response object as streams.

Note that the above process is only conducted once in an application's lifecycle, during start up.

Figure 6.3 shows the extension for the above stated extension point which registers the `ResourceHttpServlet`[183] for servicing image, style sheets or any other none XHTML file types requests. The current *Clintweb* framework uses a `DefaultServlet` class to resolve requests for non XHTML file types. The `DefaultServlet` functions only in Tomcat's Catalina web server and it therefore cannot be implemented in *Equinox - Jetty*.

```
<servlet
      alias="/clintwebStyles"
      class="de.clintworld.clintweb.servlets.ResourceHttpServlet"
      httpcontextId="de.clintworld.clintweb.facesHttpContext"
      load-on-startup="false">
</servlet>
```

Figure 6.3: ResourceHttpServlet registration

According to the `alias` attribute stated in *Figure 6.3*, requests which are under the `"/ClintwebStyles"` path are serviced by the `Servlet` class declared under the `class` attribute. The resources in the specified path are images and CSS style sheets. It should be noted that, if resources are to be stored outside the path specified in the `alias` attribute then an additional `Servlet` extension is required for mapping the external resources file types to a `Servlet` class; otherwise the resources will not be resolved.

When a request for an image or style sheet is received, the `ResourceHttpServlet` uses the `FacesHttpContext` class specified in the `httpcontextId` attribute to resolve the requested

[181] StartupServletContextListener specification, Available at: http://myfaces.apache.org/core11/myfaces-impl/apidocs/org/apache/myfaces/webapp/StartupServletContextListener.html [Accessed 16 June 2010]
[182] See section 5.3.1
[183] See section 5.3.1

resources. The `FacesHttpContext` class is specified under an extension point provided by the Equinox registry bundle, as shown below:

```
<extension
      point="org.eclipse.equinox.http.registry.httpcontexts">
   <httpcontext
         class="de.clintworld.clintweb.contexts.FacesHttpContext"
         id="de.clintworld.clintweb.facesHttpContext">
   </httpcontext>
</extension>
```

Figure 6.4: FacesHttpContext registration

The `id` is used to associate the above mentioned `Servlet` classes to the specified `httpcontext` class.

6.2.2. Applying *JSF-Facelets* to the Framework

To support *JSF-Facelets*[184] in the new framework, the `FacesServletAdapter` was configured to allow XHTML files to be used in *JSF*. The `FacesServletAdapter` extension on the `Servlet` extension point contains the following initialization parameters for *JSF-Facelets* configuration:

```
<init-param
      name="Javax.faces.DEFAULT_SUFFIX"
      value=".xhtml">
</init-param>
```

Figure 6.5: FacesHttpContext registration

The `init-param` clause refers to the declaration of the initialization parameter. The `name` clause refers to the property name and the `value` clause refers to the value assigned to the specified property. The above parameter tells JSF to assume a prefix of `.xhtml`, which the *Facelets* renderer can interpret[185]. When users request to view a *facelet* on a browser, the extension of the requested resource must be inputted as `.jsf` instead of `.xhtml`. *JSF* will associate the `.jsf`

[184] See section 2.2 and 2.3.
[185] JSF facelets, Available at: http://www.ibm.com/developerworks/java/library/j-facelets/ [Accessed 20 June 2010]

request with an `.xhtml` file using the `ClintwebViewHandler`[186] (the `ViewHandler` class implementation) which was registered to the application in the core bundle's *faces-config*.xml file.

In order for *JSF* tag libraries to be used on *facelets*, they have to be loaded into the *facelets* compiler. Normally the tag libraries are stored in the *JSF* JARs[187] outlined in section 6.1, however in the Equinox[188] environment, these tag libraries cannot be found because the core bundle's class loader does not search for these resources in the *JSF* tag libraries. Therefore, the tag library XML files were extracted from the *JSF* libraries and stored in the *META-INF* (under the *WebContent* package) folder of the core bundle. The `ClintwebViewHandler` directs the *facelet* compiler to load the tag libraries from this folder as shown in the code snippet below:

```
// Locate and load the JSF tag libs
String[] taglibraryNames = ClintwebTaglibsRegister.getClintwebTagLibs();
ClassLoader classLoader = Thread.currentThread().getContextClassLoader();

for (String taglibraryName : taglibraryNames) {
  URL resource = classLoader.getResource("META-INF/" +
      taglibraryName + ".taglib.xml");

  if (resource != null) {
    try {
      compiler.addTagLibrary(TagLibraryConfig.create(resource));
```

Figure 6.6: Loading tag libraries

As seen in *Figure 6.6*, the list of tag library names in the *META-INF* folder of the core bundle is initially generated. The bundle class loader (referred to as `classLoader` in the above figure) is used to retrieve the URL of each tag library and thereafter the tag library is added to the *facelets* compiler. The `TagLibraryConfig` class creates a `TagLibrary`[189] class instance from a tag library XML file. Note that the tag library files are searched for only in the *META-INF* directory of the core bundle. Therefore, for tag library XMLs to be found, they have to be stored under the *META-INF* directory.

After setting the above configurations, for *facelets* to use tags referenced by the available tag libraries, they must declare the namespace belonging each tag library in their `<html` tags, as shown in the figure below:

[186] See section 5.3.1
[187] JAR explained, see section 2.4
[188] See section 3.2.2.1
[189] DocJar TagLibrary Interface, Available at: http://www.docjar.com/docs/api/com/sun/*Facelets*/tag/TagLibrary.html [Accessed 22 June 2010]

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:t="http://myfaces.apache.org/tomahawk"
      xmlns:cw="http://www.clintworld.de/clintweb">
```

Figure 6.7: Namespaces for UI components

The namespace contains a prefix, which is used to reference the objects within the specified tag library. A component can be referenced in the following way:

```
<cw:selectDate/>
```

Where the `cw` prefix is a reference to the namespace of the tag library and `selectDate` is a component in the referenced tag library. Components can only be referenced via their namespaces, therefore all application *facelets* must contain the above listed namespace declarations in order to utilize JSF components.

6.2.3. Resolving *Facelets* resources

The task of resolving *facelet*[190] resources is handled by the `ClintwebResourceResolver`[191] class. This class was specified as the resource resolver in the `FacesServletAdapter` extension's initialization parameters, as shown below:

```
<init-param
      name="facelets.RESOURCE_RESOLVER"
      value="de.clintworld.clintweb.ClintwebResourceResolver">
</init-param>
```

Figure 6.8: JSF Resource resolver declaration

The above declaration sets the *facelet* resolver class to `ClintwebResourceResolver`, which is called by the `ClintwebViewHandler` class when *facelets* are requested. The `ClintwebResourceResolver` is programmed to resolve *facelets* in all application bundles via an extension point[192] defined in the core bundle. The resource resolver extension point requests extending bundles to provide the URI of their *facelets* and a class which implements logic for

[190] See section 2.3
[191] See section 5.4
[192] See section 3.2.2.1

resolving the *facelet* URLs. The supplied class must be an implementation of the core bundle's `IResourceLocator` interface as described in section 5.4. This interface defines one method called `getResourceUrl()` which contains an input parameter specifying a resource's URI. The `getResourceUrl` method is implemented as shown in the following figure (*Figure 6.9*):

```
@Override
public URL getResourceUrl(String path) {
// Search the bundle for the requested resource
  try {
    return getClass().getClassLoader().getResource("WebContent"+path);
  }
  catch (Exception e) {
    return null;
```

Figure 6.9: Resolving facelets URL

The class loader is used to retrieve the URL of the requested resource by searching for it in the *WebContent* folder of the resource host bundle. If the resource is found, a URL pointing to the requested resource will be returned to the `ClintwebResourceResolver`. Otherwise if no resource matches the requested URI, then a `null` is returned. As seen in *Figure 6.9*, resources are only searched for in specified paths. In the new framework, resources are located in a *Webcontent* folder.

In order for the `ClintwebResourceResolver` to resolve URLs, it must iterate through all extensions in the resource extension point until it finds an extension where the requested resource's URI matches the `pattern` attribute of the extension, as shown in the figure below:

```
IConfigurationElement[] config = Platform.getExtensionRegistry()
.getConfigurationElementsFor(ApplicationConstants.RESOURCE_EXTENSION);
for (IConfigurationElement e : config) {
  // get the extension name
  if(e.getName().equals("resource")){
    // compare the requested URI with the pattern attribute of the extension point
    // the pattern attribute refers to the resource URI
    if(requested_uri.endsWith(e.getAttribute("pattern"))){
      // if a match is found, then extract IResourceLocator object
      Object o = e.createExecutableExtension("class");
      // return the IResourceLocator instance
      return ((IResourceLocator) o).getResourceUrl(requested_uri);
```

Figure 6.10: Resource locator extension point

The above code extracts the `pattern` attribute in the registered extension, which corresponds to a resource name, and compares it to the requested URI. When a match is found, the

`IResourceLocator` class implementation is extracted from the extension and the requested URI is passed to its `getResourceUrl()` method. The output of this procedure call is a valid URL for the requested resource.

JSF creates a `Facelet` object for each requested resource matching a URI. These objects are cached in the `ClintwebFaceletFactory`[193]. During a resource request, the `ClintwebFaceletFactory` will initially check if a `Facelet` object matching the requested URI exists in its cache. If the `Facelet` object is found, it is then forwarded to the user; otherwise, *JSF* will resolve the requested resource from the resource extension point[194] and create a new `Facelet` object for it. The problem with a cache is that, when *facelets* are updated or removed from an application, *JSF* will instead search for a requested resource in the cache. The updated *facelets* which are available on the resource extension point will not be referenced and *facelets* belonging to bundles which are no longer in the application will still be available. Therefore, a scheme of clearing the cache was implemented. This scheme is executed in the following steps:

- When a new bundle is uninstalled or activated, the `Activator` class of the core bundle has a static Boolean flag which is set to true.
- When a request for a *facelet* is received by the core bundle, the `ClintwebFaceletFactory` class instance checks the value of the Boolean flag in the above step. If the flag is set to true, then the *facelet* cache is cleared and the Boolean flag is set to false.

The `ClintwebFaceletFactory` references a `DefaultFaceletFactory` object which stores the cache of *facelets*. To clear the cache, the `ClintwebFaceletFactory` must be re-initialized in order to clear the cached *facelets*. When this re-initialization process is done, the *facelet* resource resolver will not be able to find requested resources in the empty cache; it will have to load the resources from the resource extension point. This technique keeps the resource resolving functionality dynamic and responsive to changes in an application's environment.

The `ResourceHttpServlet`[195] is used to resolve non *facelet* resources (non XHTML resources) by initially calling the `FacesHttpContext` object to resolve the MIME type of a resource and then loading it into the client response object. In order to resolve images and other non XHTML files in other application bundles, the files (images/pdf/CSS) must be registered on the resource extension point, otherwise, they will not be resolved. The `FacesHttpContext` will initially

[193] See section 5.3.2
[194] See section 5.4
[195] See section 5.3

search for the resources in the core bundle, and if they are not found, it will attempt to resolve them using the resource extension point.

6.2.4. Resolving *JSF* Components

This section discusses how the design specifications of *JSF* features stated in section 5.6 were implemented in the new framework. The *JSF* features include, *Managed Beans*, *Navigation Rules* and custom components.

6.2.4.1. *Managed Beans*

The `ClintwebManagedBeanResolver` class is responsible for resolving *Managed Bean* classes. To set this class as the application bean resolver, the following entry was declared under the application settings of the core bundle's *faces-config.xml* file:

```
<el-resolver>de.clintworld.Clintweb.ClintwebManagedBeanResolver</el-resolver>
```

When a *Managed Bean* is requested, *JSF* delegates the task of resolving a *Managed Bean* class to the above specified resolver class. *Figure 6.11* depicts how the *Managed Beans* requests are resolved:

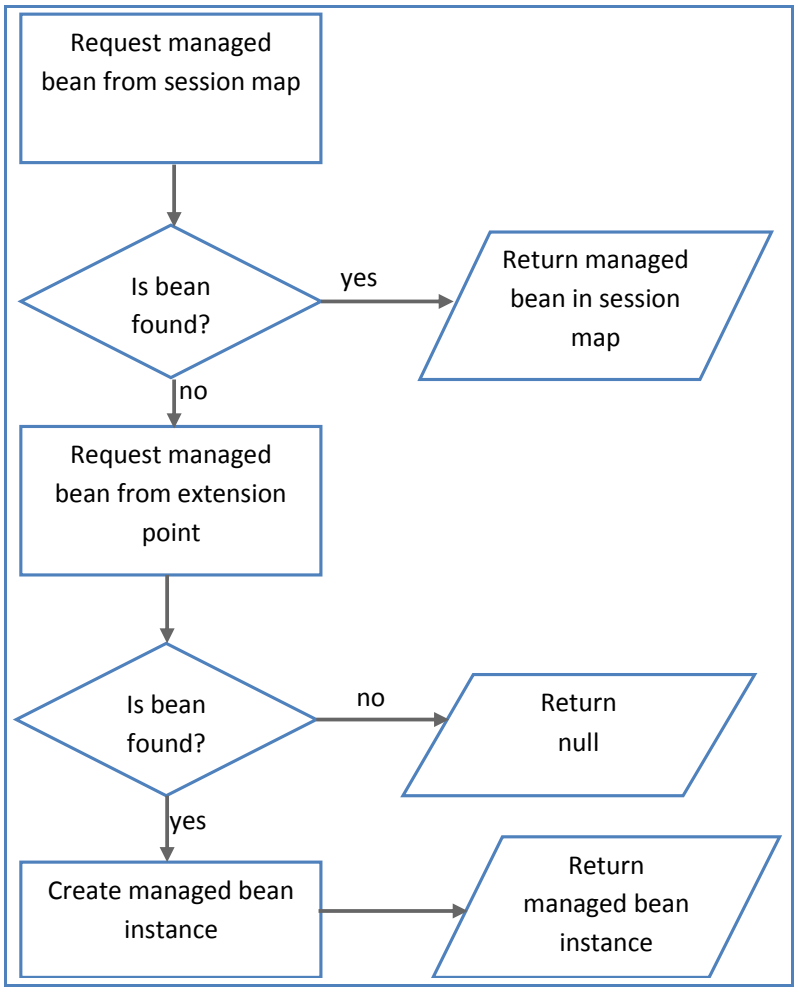


Figure 6.11: Resolving Managed Bean objects

The `ClintwebManagedBeanResolver` checks the session map[196] for the requested *Managed Bean* object. If the *Managed Bean* is found in the session map, it is forwarded to *JSF*. Otherwise, if the *Managed Bean* object is not in the session map, the resolver delegates the task of creating the *Managed Bean* to the `OSGiRuntimeConfig`[197]. The `OSGiRuntimeConfig` class instance receives a String parameter denoting the *simple-name* of the requested *Managed Bean* and it creates a *Managed Bean* from an extension point[198] as shown in the code below:

[196] See section 5.7.1
[197] See section 5.6.1
[198] See section 3.2.2.1

```
// Read managed beans from extension point
IConfigurationElement[] config = Platform.getExtensionRegistry()
    .getConfigurationElementsFor(ApplicationConstants.BEAN_EXTENSION_POINT);
for (IConfigurationElement e : config) {
  if(e.getName().equals("bean")){
    // get managed bean name
    String beanName = e.getAttribute("simpleName");
    if(name.equals(beanName)){
      // get managed bean scope
      String scope = e.getAttribute("scope");
      // get the managed bean class
      String className = e.getAttribute("class");
      // get the host bundle of the managed bean
      bundleChangedName = e.getNamespaceIdentifier();
      // create the OsgiManagedBean instance using the above
      // extracted information
      return new OsgiManagedBean(beanName, scope, className,
          Platform.getBundle(e.getNamespaceIdentifier()));
    }
```

Figure 6.12: Managed Bean extension point

The *Managed Bean*'s *scope*, *class name* and *simple-name* are extracted from the extension and using this information, a new instance of `OSGiManagedBean` is created. The bundle instance which hosts a *Managed Bean* is required because the `OSGiManagedBean` implements a mandatory `getClass()` function for loading a *Managed Bean* class from a bundle instance. *Managed Bean* classes are loaded from their host bundles otherwise they cannot be found. The created *Managed Bean* is thereafter passed to the `ClintwebManagedBeanResolver`[199] which passes control of processing *Managed Bean* properties to *JSF*.

The algorithm for creating and resolving *Managed Beans* is dynamic. *Managed Beans* belonging to newly installed bundles can be instantiated by the `ClintwebManagedBeanResolver` and made available to an application. Furthermore, when a bundle is uninstalled, its *Managed Beans* are deleted from the application session map so that they will no longer be available to a running application.

6.2.4.2. *Navigation Rules*

Just like the *Managed Beans*, *Navigation Rules*[200] are created by the `OSGiRuntimeConfig` class. When the first *Navigation Rule* request is received, the `OSGiRuntimeConfig` reads the

[199] See section 5.6.1
[200] See section 2.2.2

extensions available at the *Navigation Rule* extension point; instantiates a `NavigationRule` object for each registered extension and stores it in a list. From then onwards, *JSF* will be requesting *Navigation Rules* from the created list of `NavigationRule` objects.

The registration of *Navigation Rules* is dynamic depending on the behavior of application bundles. New bundles which contain new *Navigation Rules* may be installed during the lifetime of an application; therefore their *Navigation Rules* must be detected and in-cooperated into a running application. A procedure for handling this situation was programmed as follows:

- When a new bundle is updated or uninstalled, the `Activator`[201] class instance of the core bundle[202] receives a *bundle-changed* event. This class contains a static Boolean `resetFaceletCache` flag which is set to true when a bundle changed event occurs.
- When a *facelet* request is received, the `FacesServletAdapter` class checks if the `resetFaceletCache` flag in the above step is set to true, and if so, the framework assumes that there has been a change in a bundle's state therefore the *Navigation Rules* have to be updated. The next time a *Navigation Rule* is requested, the current *Navigation Rules* list is cleared and the *Navigation Rules* extensions[203] will be instantiated again to create new *Navigation Rules* objects.

The *Navigation Rule* resolving process can adapt to application changes; the only disadvantage of this method is that all *Navigation Rules* are deleted when a bundle is uninstalled or updated. For example, if a bundle containing one *Navigation Rule* is activated; all application *Navigation Rules* in the application must be reloaded.

6.2.4.3. Custom JSF UI Components

As discussed in section 5.6.3, in order for the new framework to support custom components, a tag library XML file (`clintweb.taglib.xml`) which defines the qualified name of a tag library class (`ClintwebTagLibrary`) was created. This XML file (located in the *META-INF* folder of the core bundle) was loaded into the *facelets*[204] compiler to allow *facelets* to reference components defined in the `ClintwebTagLibrary` class. In order to specify custom components in the `ClintwebTagLibrary` class, the following method was utilized:

[201] See section 3.2
[202] See section 5.2
[203] See section 5.6.2
[204] See section 2.3

```
addComponent("cwLabel", ClintwebLabelComponent.class.getName(),
                "Javax.faces.render.Renderer");
```

The `addComponent()` method is inherited from *JSF*'s `com.sun.facelets.tag.AbstractTagLibrary` class; it requires the reference tag name of the UI component, followed by the name of component class and the renderer class name as input parameters of type String.

The core bundle contains an extension point for bundles to register `TagLibrary` class implementations. A `TagLibrary` is a class which associates a collection of tags with a namespace. Bundles can provide an implementation of the `TagLibrary` which declares tag names associated with custom UI components that are located within the respective bundles. When a request for a custom component located outside the core bundle is received, the `ClintwebTagLibrary` initially searches the tag library extension point for a registered `TagLibrary` class which is associated with the requested component's namespace. Upon finding the required `TagLibrary` class implementation, the `ClintwebTagLibrary` creates a `TagHandler` object that will be responsible for the rendering of the custom component belonging to the specified namespace. From then on, when the custom component is referenced during the lifecycle of an application, the created `TagHandler` class instance will resolve the component and its values.

The `ApplicationImpl` class is used by JSF to create `UIComponents` class instances. This class is only able to resolve `UIComponent` classes located in the core bundle because it uses the core bundle's class loader to resolve the classes. Since component classes defined in other application bundles are out of scope for the core bundle's class loader, the `ApplicationImpl` will throw an exception when a component located in another application bundle is requested. Therefore, in order to resolve `UIComponent` classes defined in other application bundles (excluding the core bundle), an extension point for registering custom `UIComponent` classes was implemented. The extension point allowed bundles to register their UI component classes as extensions. Furthermore, a custom implementation of the JSF `ApplicationFactory`[205] called `ClintwebFacesApplicationFactory` was created. The `ApplicationFactory` creates and

[205] `ApplicationFactory`, see: http://java.sun.com/javaee/javaserverfaces/1.2/docs/api/index.html [Accessed on 20 August 2010]

returns instances of a JSF `Application`[206]. JSF applications must contain at least a default implementation of the `ApplicationFactory`. During start-up the `ClintwebFacesApplicationFactory` creates an instance of the `ClintwebFacesApplication` by calling the `getApplication()` function as shown in the following class diagram:

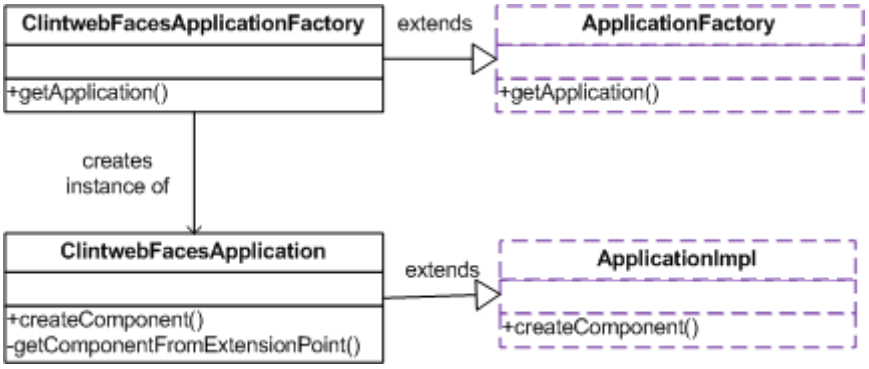


Figure 6.13: Resolving custom components

The `ClintwebFacesApplication` extends the `org.apache.myfaces.application.ApplicationImpl` class, which is the default implementation of the JSF `Application` class. In order to resolve custom `UIComponent` classes located in application bundles, the `ClintwebFacesApplication` must override the `createComponent()` function of the `ApplicationImpl` class. The `createComponent()` function receives requests for creating UI components and it calls the `getComponentFromExtensionPoint()` function to search for the requested component class in the component extension point. If the requested UI component is registered on the extension point, an instance of the component is created and passed to the JSF system. Otherwise, if the requested component is not found in the extension point, the request is forward to the `createComponent()` function of the super class (`ApplicationImpl`). JSF thereafter attempts to search for the requested UI component in the core bundle.

To set the `ClintwebFacesApplicationFactory` as the `ApplicationFactory`, the following code was added to the `start()` function of the core bundle's `Activator` class.

```
FactoryFinder.setFactory(FactoryFinder.APPLICATION_FACTORY,
        ClintwebFacesApplicationFactory.class.getName());
// To ensure that the factory will never change again during runtime,
// even if other factories are registered.
FactoryFinder.getFactory(FactoryFinder.APPLICATION_FACTORY);
```

[206] `Application` specification, see: http://java.sun.com/javaee/javaserverfaces/1.2/docs/api/index.html [Accessed on 20 August 2010]

The above code ensures that the `ApplicationFactory` class will always be set to `ClintwebFacesApplicationFactory`.

6.2.5. Session listeners

To register a session listener class to the Jetty web server[207], the core bundle contains `RegisterSessionListeners`[208] class, which accesses the Jetty session manager during application start up in order to register the `ClintwebSessionListener` class to the session manager as shown in the following class diagram (*Figure 6.14*):

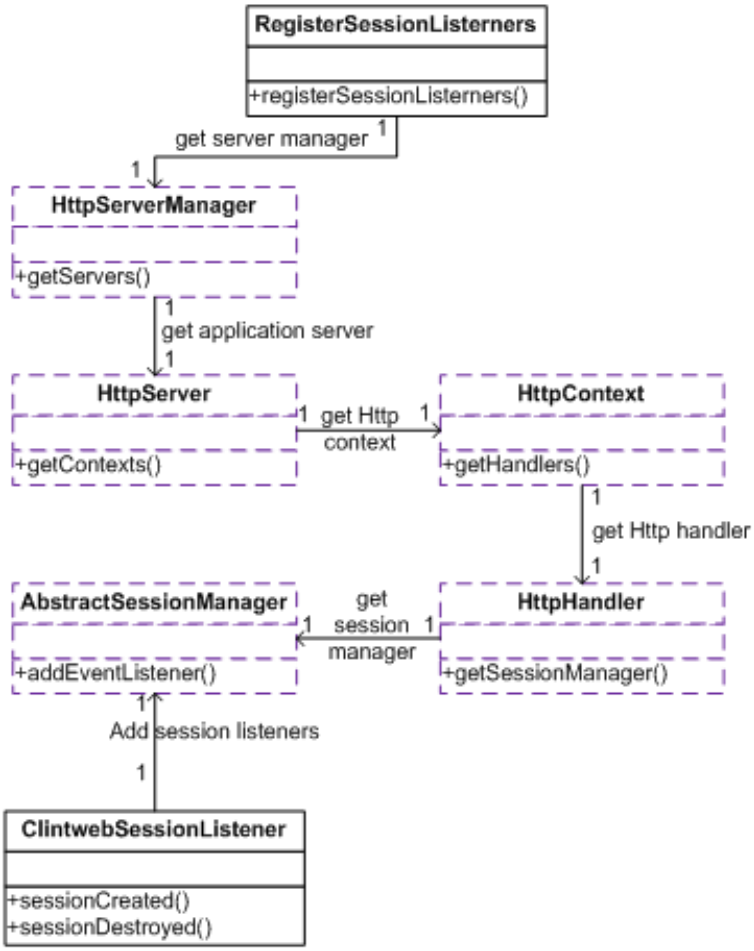


Figure 6.14: Session class registration class diagram

The `RegisterSessionListeners` class delegates the task of registering the `ClintwebSessionListener` listeners to Jetty's `AbstractSessionManager` as shown in

[207] See section 3.3.1.3
[208] See section 5.5

Figure 6.14. The following points explain the uses of each class mentioned in the above class diagram:

- The `HttpServerManager` object is provided by the `org.eclipse.equinox.http.jetty` bundle as an *OSGi* service[209] that is referenced using a `ServiceReference`[210] object. The `HttpServerManager` contains a list of active servers which are instances of `HttpServer`[211].

- An application can have one `HttpServer` instance, which references an `HttpContext` object in order to get the web server's `HttpHandler` object. The `HttpHandler` class is for handling requests from the `HttpServer`.

- The `HttpHandler` has access to the `AbstractSessionManager` class where event listener classes can be registered using a `addEventListener()` method.

The `RegisterSessionListeners` class is called by the `ClintwebViewHandler`[212] during application initialization. The core bundle contains a session listener class extension point[213] which the `ClintwebSessionListener` uses to notify other bundles of session events. The received session events are passed on to the registered listener class implementations provided as extensions, as shown below:

```
// read all registered extensions
IConfigurationElement[] config = Platform.getExtensionRegistry()
    .getConfigurationElementsFor(ApplicationConstants.
        HTTP_LISTENERS_EXTENSION);
for (IConfigurationElement e : config) {
  if (e.getName().equals("sessionListener")) {
    try {
      // extract the session listener class
      HttpSessionListener l = (HttpSessionListener)e
          .createExecutableExtension("class");
      // pass the session event to the extracted class
      l.sessionCreated(sessionEvent);
    }
```

Figure 6.15: Session listener extension point

[209] See section 3.2
[210] *OSGi* org, ServiceReference interface, Available at: http://www.*OSGi*.org/Javadoc/r4v42/org/*OSGi*/framework/ServiceReference.html [Accessed 20 July 2010]
[211] HttpServer specification, Available at: http://api.dpml.net/jetty/5.1.6/org/mortbay/http/HttpServer.html [Accessed 20 July 2010]
[212] See section 5.3.2
[213] See section 5.5

The code above shows how a listener class provided as an extension is accessed. The listener class is casted to an `HttpSessionListener` object. Thereafter a method in the listener class is called (`sessionCreated()`) and the `HttpSessionEvent` object is passed on to it. Therefore newly installed bundles containing session listener classes can receive session events from the core bundle via the session extension point.

6.3. Session State Management

This section discusses how session objects are preserved during bundle swaps. The preservation process is done by serialization of *Managed Beans*[214] belonging to a bundle because the *Managed Beans* hold the state of a *JSF* application.

6.3.1. *Managed Bean* Serialization

Managed Bean serialization only occurs when the core bundle receives a bundle *UPDATE* or *UNINSTALL* event from the Equinox framework[215]. To receive notification of these events, an `EnvironmentChangeListener`[216] class was implemented which contains a `bundleChanged()` method for receiving bundle events. This method captures bundle events that are of type *OSGi UPDATE* or *UNINSTALL* and stores the qualified name of the bundle which initiated the events in a list. Thereafter a `SessionObjectSerializer`[217] object is created to serialize the *Managed Beans* belonging to the bundle which caused the events.

The `SessionObjectSerializer` class accesses Jetty's `AbstractSessionManager` class in order to retrieve the application session map. The session map is required to get *Managed Bean* objects belonging to an uninstalled or updated bundle so that they can be serialized. The *Managed Beans* are stored in the session map where the *Managed Bean* reference name is the key and the *Managed Bean* object is the value. The `SessionObjectSerializer` additionally calls the `ClintwebManagedBeanResolver`[218] class instance to retrieve a list of *Managed Bean*

[214] See section 5.7
[215] See section 3.2.2.1
[216] See section 5.7.1
[217] See section 5.7.1
[218] See section 5.6.1

names which belong to the bundle which caused the UNINSTALL/UPDATE event. This list is used to reference the *Managed Bean* objects in the session map. For each active session, the required *Managed Beans* objects are serialized into byte streams and saved in a storage location (RAM or disk). After a *Managed Bean* is serialized, its entry in the session map is removed. This conserves application memory and reduces redundancy of data.

The serialization process is a batch process, meaning that *Managed Beans* belonging to all active user sessions are serialized at once. The *Managed Bean* preservation task is time costly. During this process, user requests received by the framework are delayed until bundle swaps are successfully completed.

6.3.2. *Managed Bean* De-serialization

When a bundle is swapped out, its *Managed Beans* are serialized and removed from the application session map. Shortly after, another version of the swapped out bundle is installed and the attribute values of the serialized *Managed Beans* are supposed to be restored into the swapped in bundle's *Managed Beans*. The core bundle[219] handles the restoration of the serialized of *Managed Beans*. When a *Managed Bean* is not found in the session map, *JSF* calls the `OSGiLifecycleProviderAdapter`[220] to create a new instance of the requested *Managed Bean* class. When a new *Managed Bean* instance is requested, the application initially checks if a version of the *Managed Bean* belonging to the requesting user has been previously serialized. This process is done by the `SessionObjectDeserializer`[221] class as shown in the following figure:

```
String beanClassName = e.getAttribute("class");
if(beanClassName.equals(className)) {
  // Extract bean from serialized object
  SessionObjectDeserializer sod = new SessionObjectDeserializer();
  Object obj = sod.loadObject(e.getAttribute("simpleName"),
      Platform.getBundle(e.getNamespaceIdentifier()));
  if(obj != null)
    return obj;
  else
    return e.createExecutableExtension("class");
```

Figure 6.16: De-serializing Managed Beans

[219] See section 5.2
[220] See section 5.6.1
[221] See section 5.7.2

The `loadObject()` function belonging to the `SessionObjectDeserializer`, takes the requested bean *simpleName* (provided by the *Managed Bean* extension point) as an input parameter and uses it to search for a serialized *Managed Bean* which matches the specified *simpleName* parameter. *Figure 6.17* describes the process flow of de-serializing a *Managed Bean*:

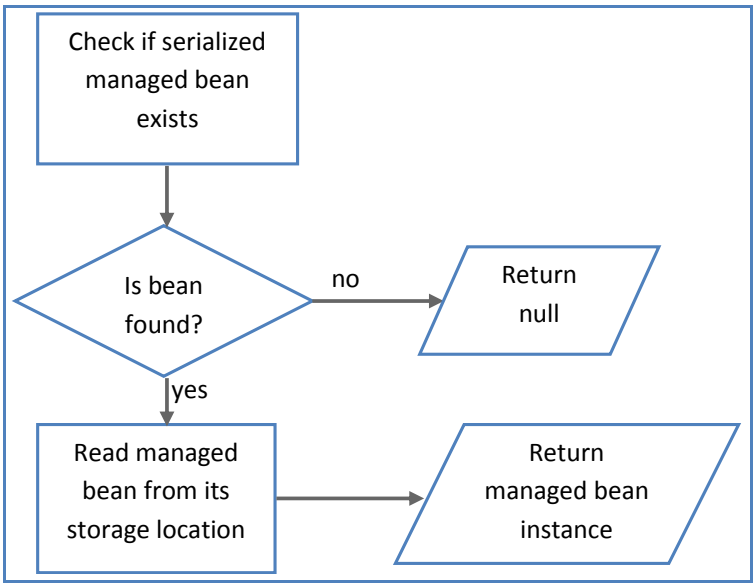


Figure 6.17: Resolving Managed Beans

The `SessionObjectDeserializer` first obtains the session ID[222] and the name of the bundle which hosts the referenced *Managed Bean* class. Using the session ID, the `SessionObjectDeserializer` checks if the requested serialized *Managed Bean* is in the storage location. If it is found, the requested *Managed Bean* is read into an `Object` using an `OSGiObjectInputStream` class instance. The `OSGiObjectInputStream` is an extension of the standard Java `ObjectInputStream` class. The `ObjectInputStream` could not be used in de-serializing a *Managed Bean* because it requires the *Managed Bean* class to be located in the core bundle where it can be associated with the serialized object; otherwise it throws a `ClassNotFoundException` exception. The `OSGiObjectInputStream` class uses the following code to associate serialized objects with their classes located in their host bundles:

```
protected Class<?> resolveClass(ObjectStreamClass desc)
ClassNotFoundException {
  String name = desc.getName();
  try {
    return bundle.loadClass(name);
  }
  catch (ClassNotFoundException ex) {   }
```

Figure 6.18: OSGiObjectInputStream resolveClass function

[222] The session ID refers to a unique identifier assigned to active session by the web container

The `loadClass` method takes the bean class name as an input parameter and gets the requested *Managed Bean* class from its host bundle. After the *Managed Bean* class is resolved by the `OSGiObjectInputStream`, it is stored into an object and its originating serialized object is deleted. In the case where the serialized *Managed Bean* is to be casted into an object with a different class structure (different class version), the de-serializer maps the values of variables which are present in both class versions to the new object and ignores the variables which are not in both class structures. If the recovered *Managed Bean* object (denoted as '`obj`' in *Figure 6.16*) is `null`, then no *Managed Bean* was found in the storage location, therefore a new *Managed Bean* can be created from the *Managed Bean* extension point using the `createExecutableExtension` function. Otherwise, if the object is not equal to `null`, the recovered *Managed Bean* will be forwarded to *JSF*. Note that the recovered beans will contain the variables values it previously had when the swapped out bundle was active.

Another consideration that had to be made was how serialized objects belonging to a user should be handled when a user session abruptly ends. According to the above described serialization process, the serialized data is only deleted when a *Managed Bean* request is received by an application. The main complication lies in, what should happen when a serialized *Managed Bean* was never requested before a user's session was abruptly ended. It is not efficient to allow the server to continue storing session objects that are no longer valid within an application. To address this problem, a procedure was programmed where a session listener object listens for *session-destroyed* events and as soon as they occur, it extracts the *session-id* of the user who initiated the *session-destroyed* event. The application thereafter deletes all preserved session objects belonging to the user with the extracted session ID. If a user's network connection breaks down abruptly but the browser remains open, his session can be later resumed and no session information will have been lost during the break down. The user's browser application must not be closed because it stores the session information which is required to reconnect to the framework and resume the old session. Since the state of an application in *JSF* is stored on the server, it is certain that session information cannot be lost when connectivity problems occur on the client side.

6.4. System Constraints

Class versioning has to be taken into consideration when swapping bundles. In different bundle versions, *Managed Bean* classes may change in structure and content. Considerations had to be made on the rules governing how old *Managed Bean* classes can be converted to new *Managed Bean* classes when a bundle swap occurs. In order to successfully recover a serialized *Managed Bean*, the new *Managed Bean* class must have the same qualified class name as the serialized *Managed Bean* object. The new *Managed Bean* may contain additional attributes and methods; these attributes will not affect the casting of the restored *Managed Bean* to the new *Managed Bean*. Static and transient variables will not be restored because the serialization API does not support them.

Another constraint is that, when changes are made to a bundle's exported classes then all dependent bundles must be updated. Changes are not taken automatically by the dependent bundles. If the dependent bundles are not updated, they will continue referencing the old exported packages stored in their caches. It is therefore important to have a clear application outline, denoting the bundle dependencies and exported packages, so that when updates are made affected bundles can also be updated.

7. Testing and Evaluation

After the framework and test application were created, they had to be tested to check if they meet the requirements stated in chapter 4. The application was tested to demonstrate its functionality and performance. The functionality refers to the dynamics of application bundles[223] and *JSF* features. The performance refers to the preservation of session objects when bundles are hot swapped[224].

7.1. Framework Functionality Tests

These functionality tests were carried out for sixteen users. These tests were meant to demonstrate that users can be able to use applications even when changes are being made to application components.

7.1.1. Resolving resources

Figure 7.1 shows how the front page provided by the `de.clintworld.clintweb.ui` bundle[225] version `1.0.0.2`. The layout of the page was created using CSS and images which are rendered by the `ResourceServlet` as stated in section 5.3. The entire *facelet*[226] shown in *Figure 7.1* was rendered by the `FacesServletAdapter`.

It was stated in the requirements[227] that *facelets* should be able to reference other *facelets* within their host bundles and also in other application bundles. *Figure 7.1* shows the application main page, the Home tab navigates to a *facelet* located in the `de.clintworld.clintweb.ui` bundle and the config tab is linked to a *facelet* located in the `de.clintworld.clintweb.core` bundle. Upon clicking the tabs, the user was directed to the requested page located within their respective bundles.

[223] See section 3.2
[224] See section 3.1
[225] See section 6.2
[226] See section 2.3
[227] See section 4.1.2

Figure 7.1: Application front page version 1

It can therefore be concluded that *facelet* resources can be resolved in the new framework regardless of where they are located within an application. Furthermore, resources like images and style sheets (CSS) are also resolved by the framework.

7.1.2. JSF functionality

In order to test the functionality of *Managed Beans*[228], the *facelets* of the `de.clintworld.clintweb.ui` bundle was made to reference *Managed Bean* properties. User inputted information on *JSF* UI components (e.g. Radio button and text boxes) and the inputted values were stored in *Managed Beans*. *Figure 7.2* shows a *facelet* in the `de.clintworld.clintweb.ui` bundle which contains text boxes where users can input text which is stored in a *Managed Bean* object on the server side.

[228] See section 2.2

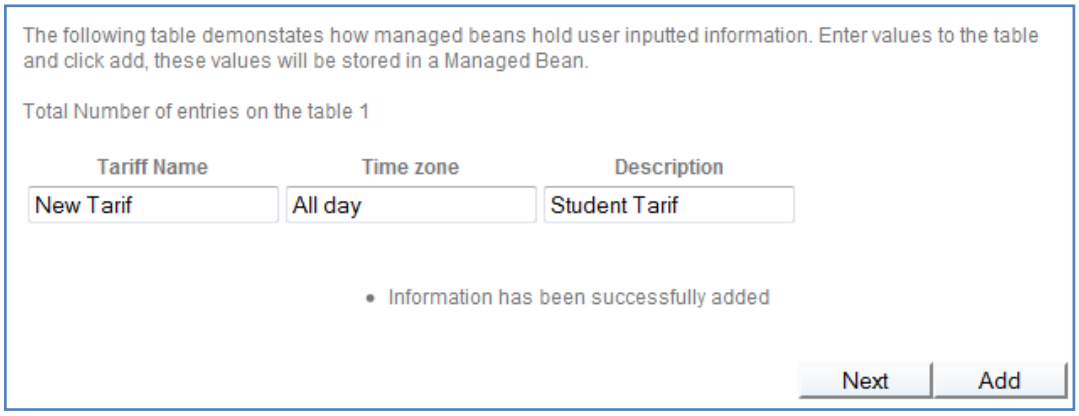


Figure 7.2: Storing values in Managed Beans

Text was inputted on the text boxes shown above and by clicking on the Add button; the text was stored in a *Managed Bean*. The stored information was displayed on another *facelet* which can be navigated to by clicking on the Next button.

For testing the *Navigation Rules* functionality, the home page contains a link which prompts a *Navigation Rule* to be executed. When this link was clicked, the application was able to successfully navigate to the *facelets* specified by the *Navigation Rule*.

Custom components[229] were also implemented in the application. A radio button component (*Figure 7.3*) was created and declared on a *facelet* to display values stored in a *Managed Bean*.

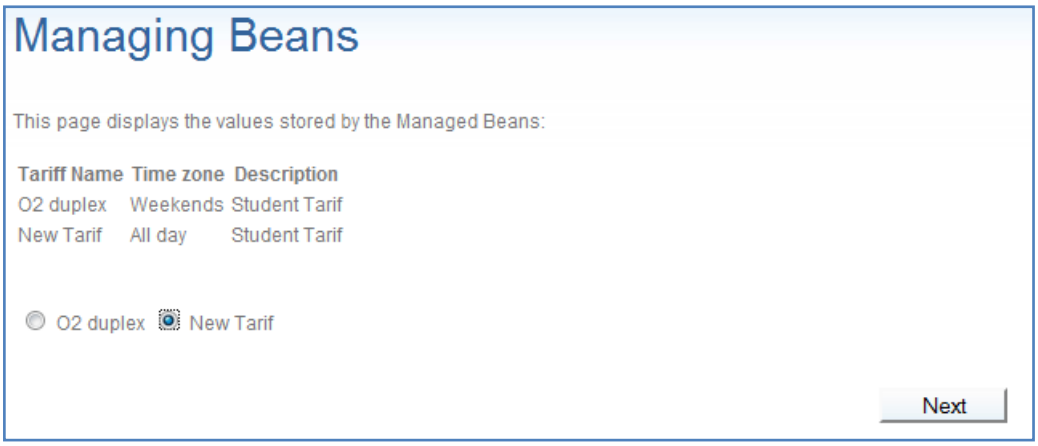


Figure 7.3: Custom component

As seen in *Figure 7.3*, the values inputted in the *facelet* displayed on *Figure 7.2* were used by the custom radio button component. Users were able to select one of the values displayed by the Radio buttons component. By clicking the Next button, the selected radio button value was

[229] See section 6.2.4.3

stored in a *Managed Bean*. The framework was additionally able to resolve custom UI components located in application bundles other than the core bundle.

According to the tests conducted in this section, it can be concluded that JSF components are fully functional in the new framework. Application bundles can provide *Managed Beans*, *Navigation Rules* and *Custom Components* which can be resolved by the framework.

7.1.3. Dynamic Bundles

Figure 7.4 shows the UI of the administrator software[230] which was created for managing application bundles within the Equinox - Jetty Framework[231]. The administrator functions are displayed in the middle panel of the application window.

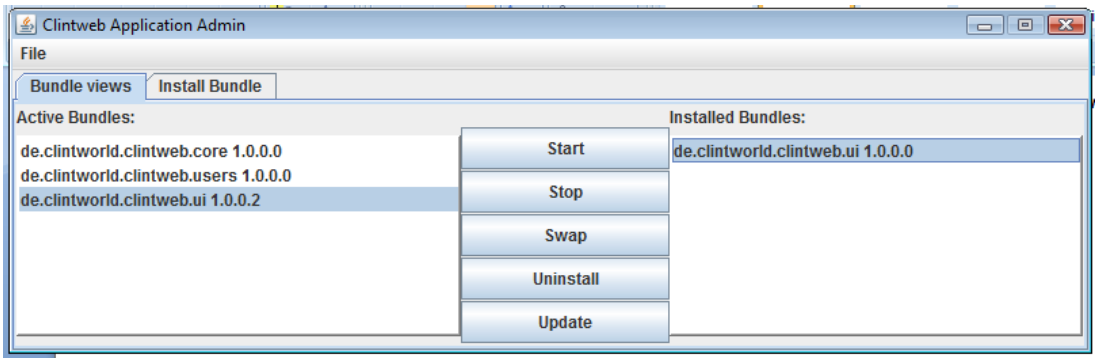


Figure 7.4: Administrator application

When the framework is started, the active bundles are displayed on the list on the left panel of *Figure 7.4*. The panel on the right side specifies bundles which are INSTALLED, STOPPED or RESOLVED. The Swap button is for hot swapping[232] bundles which are of the same name but different versions. In order to test the swapping of bundles feature, the `de.clintworld.clintweb.ui` bundle version `1.0.0.2` was swapped out and the bundle version `1.0.0.1` was swapped in. When bundles are being swapped, user requests are delayed until the swapping process is done. In this test, the bundle swap process was quick; therefore active users were not able to notice any delays in the application response. They noticed the

[230] See section 5.8
[231] See section 3.2.2.1
[232] See section 3.1

immediate effects[233] of the bundle swap, which was represented by the change in the *facelets* layout and style.

When the framework was started, if the user requested the application home page, the *facelet* in *Figure 7.1* would be displayed. However after the bundle swapping process, upon requesting the home page, the following *facelet* was displayed:

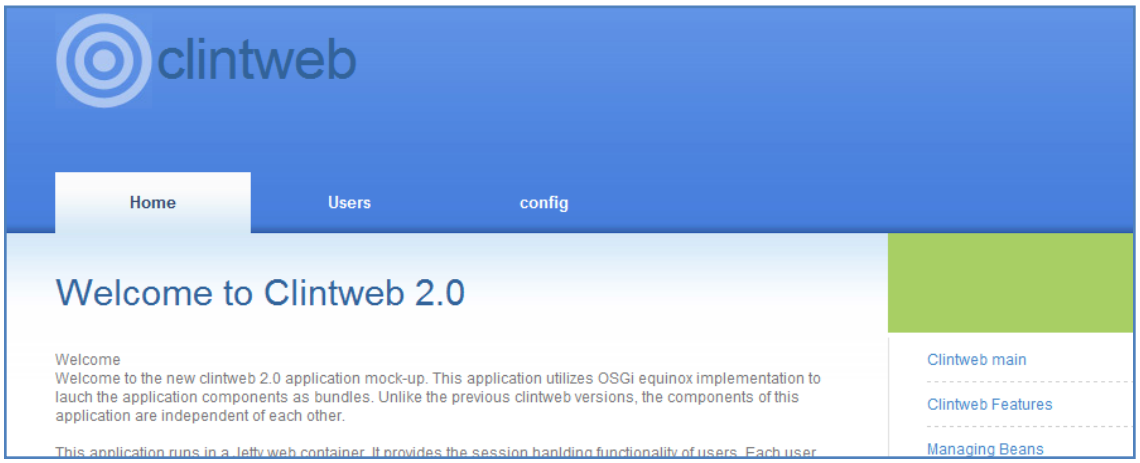


Figure 7.5: Home page version 2

Figure 7.5 shows that the above *facelet* is different from the one shown in *Figure 7.1*. The text has changed and there is an additional panel on the right side of the *facelet*. The *facelet* in *Figure 7.1* belonged to the `de.clintworld.clintweb.ui` version `1.0.0.2` which was replaced by the bundle version `1.0.0.1`. The *facelets* in both bundle versions have the same name (view ID[234]) but their contents are different as shown in *Figure 7.5*.

After the bundle swap, the user inputted information contained in the *Managed Beans* belonging to the swapped out bundle was restored into the swapped in bundle. The name of the *facelet* shown in *Figure 7.6* is the same as the *facelet* shown in *Figure 7.3*; however the contents in the *facelets* are different:

[233] The term "immediate effects" means that the effects were seen right after clicking on a button or refreshing the page. It does not mean "automatically" (via Javascript)
[234] See section 2.3

Figure 7.6: User input data page version 2

It can be seen in *Figure 7.6*, that the information in *Figure 7.3* has been restored in the newly installed bundle. Furthermore, the table on *Figure 7.6* contains a new column called *'provider'* which is not available in the table in *Figure 7.3*. The addition of this new column corresponds to the fact that the data structure of the new *Managed Bean* class is different from the data structure of the *Managed Bean* class contained in the bundle version `1.0.0.2`. The new *Bean* class contains more functionality as compared to the *Bean* in the bundle version `1.0.0.2`.

In order to preserve the *Managed Beans* belonging to a bundle. The core bundle contains a configuration page where the user can specify whether the *Managed Beans* should be preserved on the disk or on the application memory (RAM). The following figure shows the configuration page:

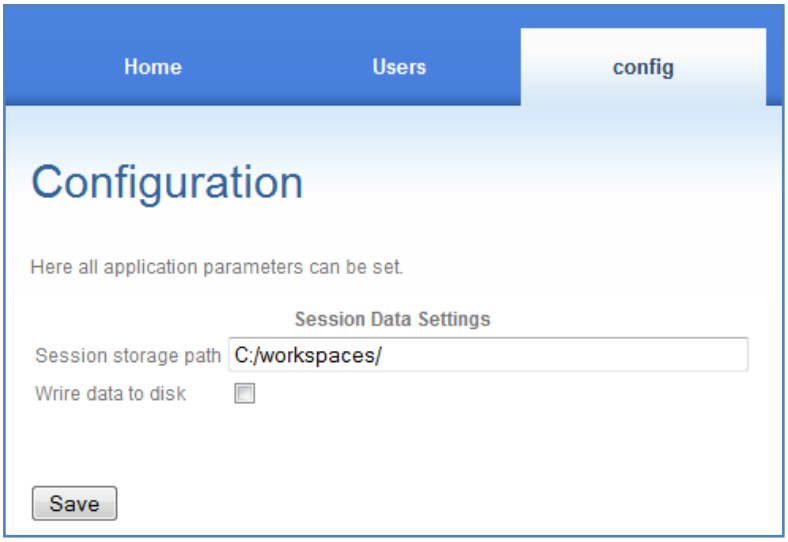


Figure 7.7: Configuration page

The check box in *Figure 7.7* specifies whether session objects are stored on the disk under a specified directory or application memory.

According to the tests conducted in this section, it was concluded that bundles can be swapped in and out of an application without disrupting a user's interaction with the application. *Managed Beans* in swapped out bundles which contain user data are always restored in a swapped in bundle. The *Managed Bean* restoration process is successfully able to adapt to changes in the class structure of a *Managed Bean* during the *Bean* restoration process.

7.2. Performance Tests

When bundles swaps occur, the *Managed Beans* belonging to a bundle can either be stored on disk or application memory (RAM)[235]. Tests were conducted to investigate the duration of writing *Bean* objects to RAM and disk in order to determine which storage method is quicker. The following test cases were conducted:

- Serialization of many small sized (size refers to memory size) *Beans*
- Serialization of many large sized *Beans*

It must be noted that the serialization process on the disk consists of creating a folder for each active session and then writing the serialized *Managed Beans* belonging to the active session into the created folder. It was not necessary to evaluate the duration of the restoration process[236] of *Managed Beans* from the disk or RAM because the process does not occur concurrently for all active sessions. *Managed Beans* are only restored from the disk/RAM when individual users request a page that references them. In these tests, a *Managed Bean* represents a user.

For the tests a machine with the following specifications was used:

[235] See section 6.3
[236] See section 6.3.2

Operating System	Windows Vista Home edition
RAM	2GB
Processor	Intel core 2 Duo -2.2 GHz

Figure 7.8: Machine specifications

7.2.1. Test case 1

For the following test case, 100 *Managed Beans* were serialized to the disk and RAM. The size of the *Beans* varied. The following results were collected from this test case:

# of *Managed Beans*	**Size of *Managed Bean* (KB)**	**Serialize to disk time (ms)**	**Serialize to RAM time (ms)**
100	2.18	457	73
100	18.7	1878	418
100	197	16280	3104

Figure 7.9: Test case 1

As seen in *Figure 7.9*, the increase in *Managed Bean* size does not necessarily mean a linear increase in serialization time. For small sized *Beans*, the serialization process to the disk is very quick. It can be seen that as the object size increases, the disk serialization time increases to a high value, meaning that users would have to wait for up to 16 seconds for the application to be responsive. On the other hand, serialization to the RAM is faster for all *Bean* sizes, therefore users would not have to wait for long periods of time during serialization.

7.2.2. Test case 2

For the following test case the number of serialized *Beans* was gradually increased while keeping the size the same (2.06 MB). The following chart shows the results collected in this test case:

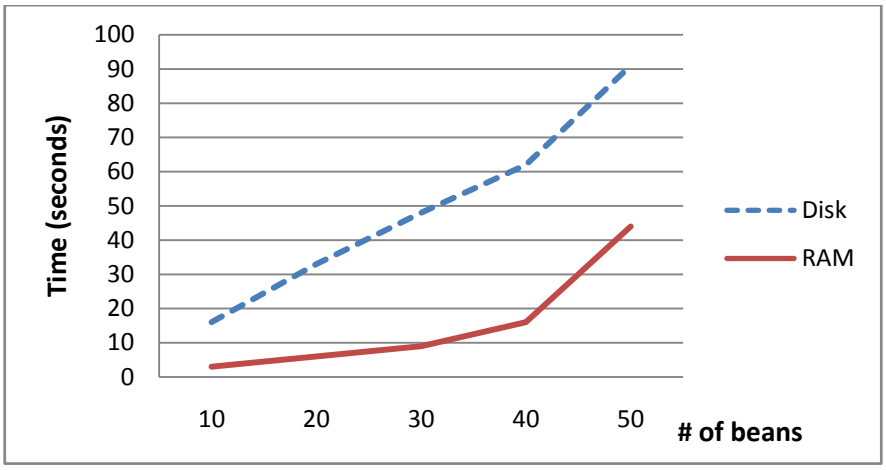


Figure 7.10: Test case 2 line graph

Due to RAM memory limitations of the test machine, it was not possible to conduct tests for serialization of more than 50 *Beans* to the RAM that are 2.06 MB in size. According to the results collected from this test case, serialization of *Beans* to the RAM is the fastest method. The time increase after 40 beans on the RAM is because the RAM of the test machine has been used up and the system is using virtual memory on the disk. Since virtual memory is located on the disk, it takes longer to access it which leads to an increase in serialization time delay. The time delay for serialization to the disk exceeds the delay requirements for the framework, it was therefore not necessary to consider the serialization time on disk for more than 50 beans. In both cases, the application will be delayed for long periods of time, but serialization to RAM will be more time efficient. Based on the test results, it can also be concluded that the serialization of few large sized *Beans* generates long delays as compared to many small sized beans.

7.2.3. Test case 3

For this test case, the number of *Beans* was incremented while keeping their memory size the same (4.16 MB). The following chart shows the results from this test case:

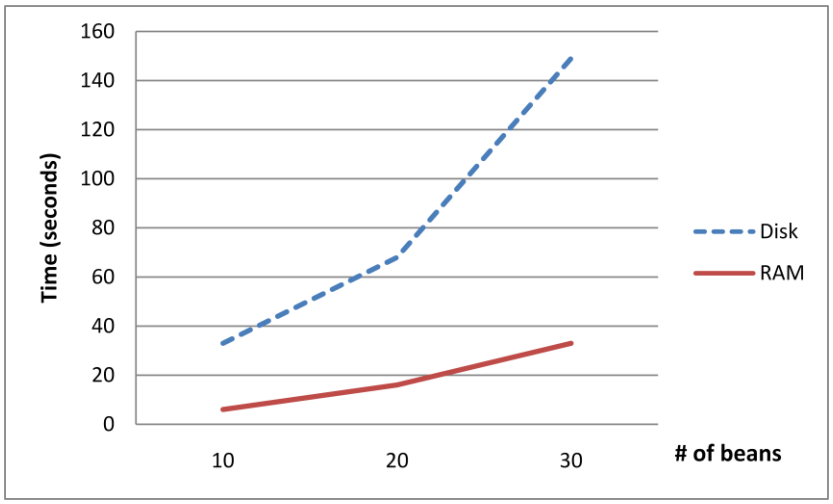


Figure 7.11: Test case 3 line graph

Due to memory limitations of the test machine, this test case could only be conducted for up to 30 *Beans*. According to the results in *Figure 7.11*, serialization to the RAM is much faster than serialization to the disk. The line in *Figure 7.11* which represents the serialization on disk, shows as the number of large sized *Beans* increase, the serialization time delay is very high. It is not acceptable to allow users to wait for up to 150 seconds during serialization. In the case of serializing to the RAM, the time delay is attributed to the fact that the RAM was depleted and the system had to utilize virtual memory, which requires more time to serialize the *Beans* to.

The results from this test case conclude that for large sized *Beans*, more time is required for serialization regardless of their quantity. The tests results show that the serialization on RAM is the most time efficient solution, however, for many large sized *Beans*, it can result in long delays depending on the machine's memory capacity as seen in *Figure 7.11*.

7.3. Summary

The functionality test results in section 7.1 clearly conclude that the developed framework meets the framework and application requirements stated in chapter 4. Bundles can be added and removed from an application without having to shutdown the framework. *Facelet* resources in application bundles are resolvable. The *JSF* and *Facelets* feature are also fully supported in the new framework, which make it possible to deploy JSF based applications.

According to the performance tests, it can be concluded, for many small sized *Beans*, the process of serialization on disk and RAM is fast. However, as the *Bean* size increases, the serialization process becomes slower even when there are a few beans to be serialized, as observed in test case 7.2.3. For many large sized *Beans*, the delay is noticeable to users. *Figure 7.10* shows that it takes 40 seconds to serialize 50 large sized *Beans*. Such a delay is not optimal for users but the time delay is much faster than serializing to disk. The delay in serialization to the RAM occurs because as the number of serialized beans increases, the RAM in the test machine is depleted. Therefore the system must use virtual memory on the disk to store the serialized beans, which causes delays because of the I/O operations involved. On a server machine with more RAM and higher processing power, the serialization time will be much faster because the framework will have more memory at its disposal. This serialization time on disk by far exceeds the requirements expectations stated in section 4.2. Users cannot be expected to wait for minutes while application changes are being made. The serialization delays on disk are caused by the I/O operations when writing to disk.

In general, the serialization performance values are expected to improve when running the framework on a server machine because servers have greater processing speeds as compared to the test machine used in this thesis. According to the tests conducted in this chapter, beans should be serialized to the RAM in order to minimize user delays. It must be noted that serialization delays cannot be completely avoided; in order to preserve user *Managed Bean*s, a slight compromise in time delay has to be made. If the size of *Managed Beans* and active users increases, the serialization process time delay will increase.

8. Conclusion

The objective of this thesis was to design and implement a dynamic component based web application framework which was required to support features offered by the current *Clintweb* framework[237]. In order to implement the desired framework, various technologies were investigated and the differences between them were outlined so as to select the most suitable technology to use for the new framework. It was therefore concluded that the *Equinox - Jetty* framework[238] was the best technology for creating a framework for hosting modular web applications because it is memory efficient, scalable and required the least configuration efforts as compared to other frameworks like, *Spring DM* and the `Servlet` Bridge configuration[239].

Using *Equinox - Jetty*, a framework which supports the deployment of *JSF*[240] based dynamic modular web applications[241] was successfully designed and realized. Its dynamic module handling capabilities were demonstrated in chapter 7, where modules in the form of *OSGi* bundles[242] were swapped[243] in and out of an application during runtime. However user session state information contained in bundles is lost when a bundle is swapped out of an application. Therefore a solution was required to preserve a user's session state when a bundle swap occurs, so that user information in a swapped out bundle can be transferred to a newly swapped in bundle. This problem was solved using the `Serializable` API[244], which demonstrated (in chapter 7) its capabilities of successfully preserving and restoring user session state during bundle swaps. According to the tests conducted in chapter 7, it was recommended that user session state should be preserved in an application's memory (RAM) so that the preservation and restoration process can be done faster as compared to storing the session state on disk. Writing the information to the disk incurred long delays, which as a result contradicts an important requirement stating that the bundle swapping process should be done as fast as possible in order not to delay user requests for too long.

The developed framework will benefit web application developers, by allowing them to add and removes features from an application without incurring any downtimes. Minor application bugs can be corrected immediately as soon as they are discovered and new features can be added to

[237] See chapter 2
[238] See section 3.3.1.3
[239] See section 3.3
[240] See section 2.2
[241] See chapter 3
[242] See section 3.2
[243] See section 3.1
[244] See section 3.4

applications, without having to wait for major application release dates. Furthermore, when errors occur in an application, the bundle which is the source of the error can always be replaced by a more stable bundle version. This will ensure that an application remains fully functional while bugs in other bundle versions are being fixed. User experience when using an application on the developed framework will be drastically improved, because users will not be inconvenienced when application maintenance is being done.

8.1. Recommendations

Due to the above stated advantages of the new framework, it is therefore recommended that applications on the current *Clintweb* framework be migrated to the new framework. In terms of migration efforts, there are several changes which will have to be made to applications in order to take full advantage of the new framework's functionality. *JSF* components will have to be declared on extension points[245] instead of the face-config.xml[246] file because the Equinox framework is not able to resolve faces-config.xml files located in multiple application bundles. Application bundles will have to declare their *facelet*[247] resources on an extension point in order for them to be resolved. According to the *OSGi* specification, bundles are not able to directly reference resources in other bundles. Using an extension point, bundles are able to register their resources making it possible for them to be resolved when they are referenced by other bundles. Other file types located outside the core bundle (e.g. images, cascading style sheets) must also be registered to an extension point or else they will not be resolvable.

The Jetty web server will replace the Tomcat web container[248] because it is more scalable and lightweight. Therefore application classes which utilize Tomcat's functionality to access session information will have to be changed to use Jetty. `Servlet`[249] class implementations in *Clintweb* will have to be replaced by the ones provided by the new framework.

In order to manage applications in the *Equinox - Jetty* framework, it is recommended that the *ProSyst*[250] *Web Administrator console* be used. This console is easy to install, secure and it offers

[245] See section 3.2.2.1
[246] See section 2.2
[247] See section 5.4
[248] See section 3.3.1.3 and section 2.1
[249] See section 2.1
[250] Prosyst mToolkit, Available at: http://www.prosyst.com/index.php/de/html/content/99/Other-FOU-|-Products-|-*OSGi*-SDK/ [Accessed 2 August 2010]

remote access to an Equinox framework, where a user can easily control the framework. Users just require a standard browser to use the web administrator. Another alternative is to use the Apache Felix web console, which can be configured to run on the Equinox framework. The Apache Felix console[251] is easy to use and it allows administration of Equinox based bundles. Note that if no remote console is used for framework administration, then the framework can only be controlled from its host server machine.

8.2. Outlook

The framework developed during this thesis is a fundamental framework for hosting *JSF* based dynamic modular web applications. The framework can be extended to support the registration of more *JSF* components on extension points (for example, UI components render classes).

Rich Faces or ICE Faces can be integrated into the new framework to enable the integration of AJAX capabilities in JSF based web applications. AJAX enables interactive and dynamic interfaces of a web page. Resources can be dynamically retrieved from the server side without interfering with the page displayed on the client frontend. AJAX can benefit the framework by allowing information on pages to be dynamically updated from the sever side.

Server load balancing must also be researched in order to efficiently manage applications on a server. Processes within an application can be allocated to multiple processors on a server in order to conduct them faster. Another server related issue which can be further developed is data integrity. In case problems such as power failure occur on the server, the user inputted data must have a back up, so that there is no loss of sensitive user information. A solution to this problem may be that the backup data could be serialized to the disk by a thread running in the background. If the user session ends, then the data on disk can be deleted in order to efficiently use system memory.

The development of the current *Clintweb*[252] version is done on the Eclipse IDE, where application projects are launched on a Tomcat web server and their build process is managed by Apache

[251] Apache Felix Web console, Available at: http://www.osgilook.com/2009/07/31/monitor-your-osgi-container-with-the-apache-felix-web-console/ [Accessed 20 July 2010]
[252] See chapter 2

Maven[253]. Research on developing and launching of the new framework in the Eclipse IDE can be done, so that Eclipse can continue to be used as the development tool for the new framework. Furthermore using IAM[254], Apache Maven can also be integrated into the Eclipse IDE[255], so that the new framework can utilize it to simplify the application build process.

Finally, according to the design specification of the new framework, the core bundle[256] must always be active during the lifecycle of an application. If the core bundle is to be updated, the entire framework has to be shut down. The core bundle can be further developed to enable it to be serviced without having to shutdown the entire framework and disrupting active sessions.

[253] Apache Maven, Available at: http://maven.apache.org/ [Accessed 10 September 2010]
[254] IAM – Integration for Apache Maven, Available at: http://www.eclipse.org/iam/ [accessed 10 September 2010]
[255] Eclipse IDE, Available at: http://www.eclipse.org/ [Accessed 10 September 2010]
[256] See section 6.2

9. References

[1] Andreas Grabner, 13 August 2009. *Java Memory Problems*, Available at: *http://java.sys-con.com/node/1071319* [Accessed 20 June 2010]

[2] Apache Felix. *Welcome to Apache Felix*, Available at*:* *http://felix.apache.org/site/index.html* [Accessed 20 June 2010]

[3] Apache Maven Project, *Introduction to the Dependency Mechanism*, Available at: *http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html* [Accessed 2 August 2010]

[4] Apache Software Foundation, *Apache MyFaces*, Available at: *http://myfaces.apache.org/core20/index.html* [Accessed 16 June 2010]

[5] Arnaud Cogoluegnes, Thierry Templier & Andy Piper, August 2010. *Spring Dynamic Modules in Action*

[6] Azad Bolour, Bolour Computing, 3 July 2003. *Notes on the Eclipse Plug-in Architecture*, Available at: *http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html* [Accessed 10 June 2010]

[7] BEA WebLogic Server™, 28 June 2006. *Developing Web Applications for WebLogic Server Version 8.1 Revised*, Available at: *http://download.oracle.com/docs/cd/E13222_01/wls/docs81/pdf/webapp.pdf* [Accessed 22 June 2010]

[8] Bill Dudney, 16 July 2004. *Creating custom components*, Available at: *http://today.java.net/pub/a/today/2004/07/16/jsfcustom.html* [Accessed 20 June 2010]

[9] Eclipse News Desk, 17 March 2008. *Eclipse Announces New Runtime Initiative Around Equinox*, Available at: http://java.sys-con.com/node/520844 [Accessed 20 June 2010]

[10] Eclipse Org, *BIRT project*, Available at: *http://www.eclipse.org/birt/phoenix/* [Accessed 28 July 2010]

[11] Eclipse org, *Eclipse IDE for Java EE developers*, Available at: *http://www.eclipse.org/downloads/moreinfo/jee.php* [Accessed 16 June 2010]

[12] Eclipse org, *Embedding an HTTP server in Equinox,* Available at: *http://www.eclipse.org/Equinox/server/http_in_Equinox.php* [Accessed 16 June 2010]

[13] Eclipse org, *Web Tools Platform Project*, Available at: *http://www.eclipse.org/projects/project_summary.php?projectid=webtools* [Accessed 16 June 2010]

[14] Eclipse Org, *Whitepaper: Component Oriented Development and Assembly with Equinox*, Available at: *http://www.eclipse.org/equinox-portal/whitepaper/20080310_equinox.php* [Accessed 20 June 2010]

[15] Equinox, Available at: *http://www.eclipse.org/Equinox/* [Accessed 20 June 2010]

[16] Eric Clayber & Dan Rubel, 22 December 2008. *Eclipse: Building Commercial-Quality Plug-ins (Eclipse (Addison-Wesley)) 3rd Revised edition (REV)*

[17] Eric Jendrock, Ian Evans, Devika Gollapudi, Kim Haase and Chinmayee Srivathsa, June 2010. *The Java EE 6 Tutorial*, Available at: *http://download.oracle.com/docs/cd/E17410_01/Javaee/6/tutorial/doc/gijtu.html* [Accessed 20 June 2010]

[18] Eric Jendrock, Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin & Kim Haase, June 2010. *The Java EE 5 Tutorial For Sun Java System Application Server 9.1*, Available at: *http://download.oracle.com/javaee/5/tutorial/doc/index.html*), [Accessed 22 June 2010]

[19] Exadel Inc. 2005. *JSF KickStart: A Simple JavaServer Faces Application*, Available at: *http://www.exadel.com/tutorial/jsf/jsftutorial-kickstart.html* [Accessed 21 June 2010]

[20] Glen McCluskey, March 1999. *Tuning Java I/O Performance*, Available at: *http://java.sun.com/developer/technicalArticles/Programming/PerfTuning/* [Accessed 10 May 2010]

[21] Graham J. Ellis, Well House Consultant Ltd, *Tomcat Overview*, Available at: *http://www.wellho.net/downloads/A651.pdf* [Accessed 21 June 2010]

[22] Greg Wilkins, May 2008. *Jetty vs. Tomcat, A comparative analysis*, Available at: *http://www.webtide.com/choose/jetty.jsp* [Accessed 10 May 2010]

[23] *Java serialization*, Available at: *http://www.tutorialspoint.com/java/java_serialization.htm* [Accessed 10 May 2010]

[24] Jeff McAffer, Paul Vanderlei & Simon Archer, 14 February 2010. *OSGi and Equinox: Creating Highly Modular Java Systems* (Eclipse (Addison-Wesley))

[25] Kirk, TechDistrict, 5 August 2009. *Modularity patterns*, Available at: *http://techdistrict.kirkk.com/2009/08/05/modularity-patterns/* [Accessed 20 June 2010]

[26] Microsoft Developer Network, October 2009. *Modularity*, Available *at: http://msdn.microsoft.com/en-us/library/ff648404.aspx* [Accessed 20 June 2010]

[27] Oracle – Sun Developer Network*, JavaServer Pages Technologies*, Available at: *http://java.sun.com/products/jsp/* [Accessed June 2010]

[28] *OSGi* alliance, Available at: http://www.*osgi.*org/Main/HomePage [Accessed 10 June 2010]

[29] Peter Roßbach (Systemarchitekt), Gerd Wütherich (Freier Softwarearchitekt) & Martin Lippert (akquinet it-agile GmbH), 22. April 2009. *Mit OSGi Webanwendungen entwickeln – Was geht, was nicht?* Available at: *http://www.wuetherich.com/public/ruhrjug-2009-05/mit-OSGi-webanwendungen-entwickeln.pdf* [Accessed 10 April 2010]

[30] *Prosyst mB-SDK*, Available at: *http://dz.prosyst.com/devzone/Home/* [Accessed 28 July 2010]

[31] Qusay H. Mahmoud, 12 November 1996. *Sockets programming in Java: A tutorial*, Available at: *http://www.javaworld.com/jw-12-1996/jw-12-sockets.html* [Accessed 10 August 2010]

[32] Ramnivas Laddad, Colin Yates, Sam Brannen, Rob Harrop, Christian Dupuis & Andy Wilkinson, 2008. *SpringSource dm Server™ Programmer Guide*, Available at: *http://static.Springsource.org/s2-dmserver/2.0.x/programmer-guide/html/index.html*, [Accessed 22 June 2010]

[33] Rick Hightower (ArcMind Inc.), 21 February 2006. *Facelets fits like a glove*, Available at: *http://www.ibm.com/developerworks/java/library/j-facelets/* [Accessed 20 June 2010]

[34] RoseIndia, *JSF* Renderers, Available at: http://www.roseindia.net/*JSF*/*JSF*renderers.shtml [Accessed on 20 July 2010]

[35] Sathiskumar Palaniappan, 5 July 2009. *Java serialization algorithm revealed*, Available at: *http://www.javaworld.com/community/node/2915* [Accessed 10 May 2010]

[36] *Servlets and javaServer Pages (JSP) 1.0: A Tutorial*: Available at: *http://www.apl.jhu.edu/~hall/Java/Servlet-Tutorial/* [Accessed 21 June 2010]

[37] Sunil Patil, 22 April 2008. *Hello, OSGi, Part 2: Introduction to Spring Dynamic Modules*, Available at: *http://www.Javaworld.com/Javaworld/jw-04-2008/jw-04-OSGi2.html?page=2* [Accessed 22 June 2010]

[38] Tim Berners-Lee (World Wide Web Consortium), Roy T. Fielding (Day Software) and Larry Masinter (Adobe Systems Incorporated), January 2005. *Uniform Resource Locator (URI): Generic Syntax*, *Available at: http://labs.apache.org/webarch/uri/rfc/rfc3986.html* [Accessed June 2010]

[39] Todd Greanier, July 2000. *Discover the secrets of the Java Serialization API*, Available at: *http://Java.sun.com/developer/technicalArticles/Programming/serialization/* [Accessed 10 May 2010]

[40] Valery Abu-Eid, 11 October 2008. *HTTP Service specification explained by example*, Available at: *http://www.dynamicJava.org/articles/OSGi-compendium/http-service* [Accessed 20 June 2010]

[41] Wikipedia, articles: *Web Application Framework*, *Application server*, *Modular Programming*, *NIO*, *Direct Memory Access*, *MVC*, *Hot Swapping*, *OSGi*, *Embedding Jetty*, *URI* and *URL*, Available at: *http://www.wikipedia.org/* [Accessed 21 June 2010]

[42] w3schools, *XML Schema Tutorial*, Available at: *http://www.w3schools.com/schema/default.asp* [Accessed 20 June 2010]

[43] XStream, Available at: *http://xstream.codehaus.org/* [Accessed 10 May 2010]

10. Appendix

CD-Contend

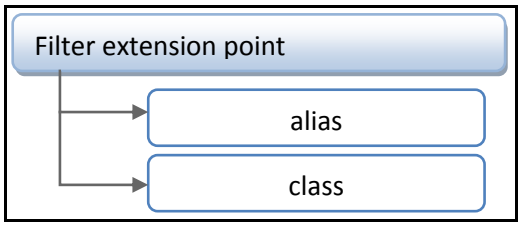
The following contents can be found on the included CD:

1. The complete master thesis in PDF and DOC format
2. Project Source Code
3. References (Web, pdf and slides)

The above listed CD has been deposited with Prof. Dr. Hans-Jürgen Hotop.

A. Filter extension specification

According to the Java SUN specification[257], a filter is an object that performs filtering tasks on a resource request and/or response. Filters may be in the form of authentication filters, encryption filters, etc. The filter extension point provided by the Equinox registry bundle requires the following information:



Listings 1: Filter extension point

The *alias* attribute refers to the URI pattern of requests that should be sent to the specified `Filter` class. The `Filter` class must be an implementation of the `javax.servlet.Filter` interface.

[257] Java SUN, Filter interface specification: Available at: http://Java.sun.com/products/servlet/2.3/Javadoc/Javax/servlet/Filter.html [Accessed 20 June 2010]

B. Deployment required bundles

```
- commons-discovery-0.2.jar
- org.apache.commons.collections_3.2.0.v200803061811.jar
- org.apache.xml.resolver_1.2.0.v200902170519.jar
- javax.xml_1.3.4.v200902170245.jar
- org.eclipse.equinox.common_3.5.1.R35x_v20090807-1100.jar
- org.eclipse.equinox.registry_3.4.100.v20090520-1800.jar
- javax.servlet_2.5.0.v200806031605.jar
- javax.servlet.jsp_2.0.0.v200806031607.jar
- org.eclipse.core.runtime.compatibility.registry_3.2.200.v2009042
  91800.jar
- org.eclipse.core.jobs_3.4.100.v20090429-1800.jar
- org.apache.xml.serializer_2.7.1.v200902170519.jar
- org.eclipse.equinox.app_1.2.0.v20090520-1800.jar
- org.apache.commons.logging_1.0.4.v200904062259.jar
- org.eclipse.equinox.preferences_3.2.300.v20090520-1800.jar
- org.eclipse.core.contenttype_3.4.1.R35x_v20090826-0451.jar
- org.eclipse.osgi.services_3.2.0.v20090520-1800.jar
- org.eclipse.equinox.http.servlet_1.0.200.v20090520-1800.jar
- org.eclipse.core.runtime_3.5.0.v20090525.jar
- org.apache.xerces_2.9.0.v200909240008.jar
- org.apache.commons.el_1.0.0.v200806031608.jar
- org.eclipse.persistence.jpa.equinox.weaving_1.1.3.v20091002-
  r5404.jar
- javax.transaction_1.1.1.v201002111330.jar
```

Listings 2: Equinox required bundles

Acknowledgements

I would like to thank Prof. Hans-Jürgen Hotop for his support during the writing of this thesis. I extend my gratitude to my supervisor and colleagues at Clintworld GmbH, Florian Albrecht who contributed many ideas to this research, Christian von Leesen for his support and all other colleagues who contributed to this thesis. I am grateful to Sahil Jawa, Zhonglei Zou, Kulathat Teanjaung and Durga Rajamani for their assistance during my master studies.

I would also like to thank my parents for their moral and financial support throughout my studies. Finally I thank God for the never ending motivation and inspiration I received from the bible, “And we know that all things work together for good to them that love God, to them who are the called according to His purpose” - Romans 8:28.

I declare within the meaning of section 25(4) of the Examination and Study Regulations of the International Degree Course Information Engineering that this Master Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

Hamburg, November 3, 2010 Arnold Kemoli