JSF2 Showcase Tutorial

Version 1.0

- 02 May 2013

Table of Contents

AIM OF THIS DOCUMENT

The aim of this document is to provide the implementation details of the Appverse Web JSF2 showcase application. The showcase application shows how to integrate JSF2 using RichFaces 4 as UI framework and Mojarra reference JSF implementation.

This is an example but you could integrate JSF2 with Appverse Web using the JSF2 implementation and UI component framework of your choice. In this regard, Appverse Web provides a set of recommendations or best practices showing what we think is the best way to integrate JSF2 with Appverse Web for Rich Application Interface development but, in any case, your code can be pure JSF2 without any restriction.

However, we have our own vision and our recommendations are based on this. The way we see JSF2 and UI component frameworks working together with Appverse Web back end technologies is as a mere MVC and UI component framework (replacing Spring MVC or Struts, for instance, and providing a rich set of UI components). This would allow a clear separation between presentation and pure back-end service layers.

We have based our showcase implementation on this idea and we will explain you this in detail.

We will show you how to tackle basic and important aspects that most of the applications require such as: i18N, data validation and errors management, security, logging, styling, etc.

WHO CAN READ THIS DOCUMENT?

Anyone interested in developing applications with Appverse Web and JSF2 is encouraged to read this document, especially developers. However, you need to be aware that this is a quite technical document. In order to be able to follow the explanation and have a deep understanding of the technical details a good knowledge in Java basics, web development, JSF2 and Appverse Web back-end devolpment is required.

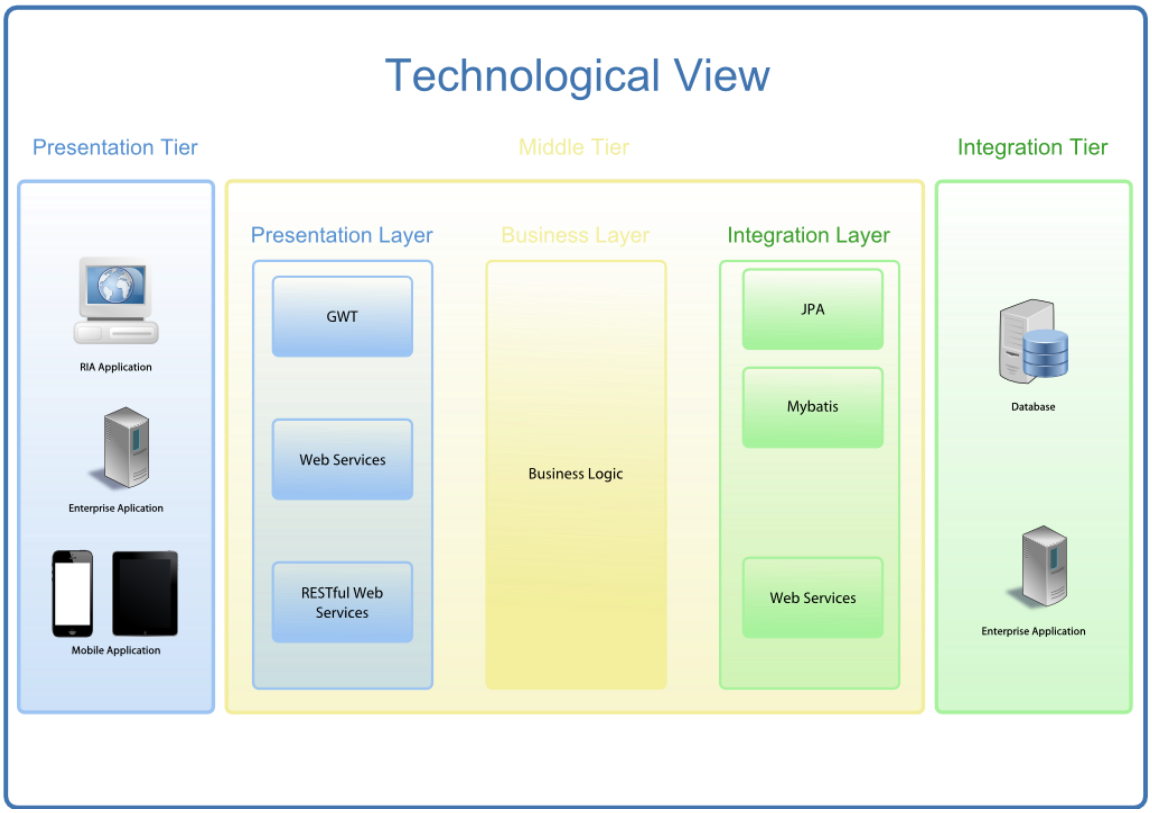
It is highly recommended to read the following Appverse Web documentation before starting this document. Some aspects will be just commented quickly as you can find further explanations here:

http://appverse.github.io/appverse-web/guide/index.htm

1.1. JSF 2 and RichFaces 4 as MVC and UI components framework

Appverse Web is built using a multi-tier architecture pattern, either using a three-tier architecture or being part of a more complex or simpler N-tier architecture.

For this showcase, we are going to focus on the three-tier architecture traditionally used for web development:



We will not give an extense explanation of N-tier architectures as this is out of the scope of this document. If you want further details regarding this matter and how Appverse Web supports this, please read the following section of the document:

(PENDING TO ADD UPDATED LINK TO ARCHITECTURE GUIDE SECTION HERE)

The point we would like to make with the previous picture is that JSF 2 fits perfectly in the Presentation Layer of the Middle Tier for RIA Applications. The Presentation Tier is responsible of physically data rendering, providing a visual representation of the data feed by the middle tier into presentation flavour specifics and reacts to the user interaction calling to the middle tier in order to perform transactions and build a response. From a logical point of view, the Presentation Layer is providing support to the Presentation Tier to achieve its goal.

JSF 2 is one of the possible flavours for the Presentation Layer. For instance, you could develop an application based on RIA paradigm using JSF2 (JSF2 flavoured) instead of GWT (GWT flavoured) depending on your requirements. You would need to take into account that JSF 2 and GWT are completely different technologies with its advantages and disadvantages.

As you can see in the picture above, the Business and Integration layers are not dependant on the technology used in the presentation layer. This allows us to use JSF2 as a mere implementation of the

presentation layer that delegates on the Business Layer (completely agnostic of the Presentation Layer) in order to retrieve application data.

1.2. Why JSF2 ?

It is out of the scope of this document to compare different web technologies or frameworks that allow you to develop rich internet applications.

We believe that depending on the requirements, JSF2 could be a perfect technology to develop a Rich Internet Application (RIA). It could be a good alternative to other technologies such as Google Web Toolkit (GWT). However, as it has been commented earlier, each technology has its advantages and disadvantages. For instance, GWT presents a pure client – server architecture with the client (user browser) being able to keep status and making GWT RPC calls or JSON based calls to the service layer in the server side. Thus, you can make most of your services stateless. On the other hand, the fact that GWT follows a client – server architecture obliges you to be aware of the data serialization process between server and client. Architectures based on JSF2 are completely different in the sense that it is not a client – server architecture. In this case, all the processing it is performed in the server side, like traditional servlets or JSP that return directly the HTML code to the browser. In this regard, JSF2 makes you to be aware about the fact that status has to be kept between HTTP requests – even though that JSF2 lifecycle is precisely designed to make this almost transparent for you. On the other hand, with JSF2 you will not experience serialization issues as all the processing is carried out in the same JVM. In summary, these are some things to take into account in order to choose JSF2 or GWT: history management required by the application, if the HTML code has to be crawlable by search engines, scalability, status to be kept by the application, richness of the UI components offered by the plain technology or third parties that you might use, new components development effort, developers skills and so on.

For RIA (and web development in general) we recommend you to develop your presentation layer using either Google Web Toolkit or JSF2.

We recommend you to use JSF 2 instead of traditional JSP or servlets developments (even using MVC frameworks such as Struts, Struts2 or Spring MVC) as JSF implements MVC itself and we value the following features as quite interesting:

- It 2 encourages consistent use of MVC throughout your application
- JSF 2 lifecycle makes keeping the state between HTTP requests very easy
- JSF 2 lifecycle makes data binding and validations easy
- It integrates well with JSR-303 validation specification
- JSF 2 supports AJAX very well. This is very important for RIA development
- It provides a set of HTML based GUI controls along with code to handle their events
- There are several free and payment third party GUI components libraries that offer rich components (grids, accordion, drop-down..) that support AJAX. This is very important for RIA development.

1.3. Why RichFaces 4?

You could use any JSF GUI components library enriching the plain JSF components in your developments. Deloping and maintaining JSF GUI components takes time and effort. For this reason, we recommend you to use third party libraries as long as the components fit your application needs. If not, you can always develop a new component or extend an existing one.

There are several JSF GUI components libraries both for free or for a fee and most of the times you can even combine several libraries in your project.

So, even though we have choosen RichFaces 4 for our showcase application, you could use another or even combine them. In case you decide to combine them you might need to struggle a bit more with styling.

We have choosen RichFaces 4 for our showcase because:

- It is supported by JBoss Community. This ensures there is a big community behind, developing the JSF framework, providing new components and giving support in forums
- It is licensed under "GNU LESSER GENERAL PUBLIC LICENSE"
- It offers a rich showcase of components supporting AJAX and client side JSR-303 validation which helps to develop rich Internet applications
- It offers a powerful skinning mechanism
- It is well documented

2.1. Application aspects covered by this showcase

The own showcase application shows a list with the different application aspects that it is covering:

- JSF2 and Spring 3.x integration
- Maven project layout, dependencies and plugins setup
- Appverse Web three layer architecture
- In memory database setup
- Maven plugin setup to generate database scripts from JPA annotated beans
- Spring security integration
- I18N
- Styling and Skinning
- JSR-303 field validation in client
- JSR-303 server side validation
- Errors management
- Remote pagination and sorting

The showcase application is based on standard Maven layout. This mean you can download the project from GitHub repository and set up the project in your favourite IDE.

https://github.com/Appverse/appverse-web.git

3.1. Configured servlet containers

For this showcase we have set up Maven configuration to be able to deploy and run the showcase in two servers:

- Embedded Jetty servlet container – Configured as a Maven plugin (no external setup is required)
- External Tomcat 7 servlet container with Maven deploy plugin setup (requires a external Tomcat 7 servlet container to be set up).

Note: Take into account that by default we specify Java version 1.6 in pom.xml to compile the source code. If your project requires 1.7 you can just change the maven compile plugin so that it uses Java 1.7 to compile in "source" and "target" arguments. On the hand, the Jetty Maven plugin we use requires Java 1.7. This implies you need Java 1.7 correctly setup to run the showcase in Jetty embedded servlet container. Take into account that it is not a problem at all that you have compiled the web application with Java 1.6 and you are running Jetty with Java 1.7 as Jetty is proposed just for local development environments. You should not have any problem to run this showcase.

3.1.1. Running the application in embedded Jetty servlet container (default Maven goal)

The Maven default goal of the project is set up as follows:

```
<defaultGoal>clean jetty:run-war</defaultGoal>
```

So, if you build your project, the target directory will be cleared, classes will be compiled and the web application will be running inside an embedded Jetty servlet container.

This is the easier setup and probably the best option during most of the development phase as Jetty is a very light servlet container and besides we have configured the Maven plugin so that changes in classes and files are automatically reloaded.

3.1.2. Running the application in an external Tomcat 7 servlet container

This option requires an external Tomcat 7 servlet container and configure a user and password to be able to do remote deployments using Maven plugins by the Tomcat Manager URL. How to do this is out of the scope of this document.

In order to build, deploy and run your application in a Tomcat 7 servlet container, please run the next Maven goals:

```
clean package tomcat7:deploy-only
```

or, if you prefer, replace the default goal with:

```
<defaultGoal>clean package tomcat7:deploy-only</defaultGoal>
```

Once Jetty or Tomcat servlet container has started please enter the following URL in your browser to access to the application:

http://localhost:8080/jsf2showcase

3.1.3. Running the application in a external Tomcat 6 servlet container

If you want to deploy the JSF2 showcase in a Tomcat 6 Servlet you will need to do some setup in the application deployment descriptor and in the Tomcat 6 /lib folder.

The reason is that Tomcat 6 includes EL 2.1 by default. If you want to use all the power of JSF2 you need EL 2.2 which gives you the possibility of using all JSF2 features.

3.1.3.1. web.xml settings

You need to configure the right expression factory adding this context param:

```
<context-param>
      <param-name>com.sun.faces.expressionFactory</param-name>
      <param-value>com.sun.el.ExpressionFactoryImpl</param-value>
</context-param>
```

3.1.3.2. Tomcat preparation

Tomcat6 requires some set-up, otherwise it will crash with a Exception on container startup:

1. Go to the tomcat home directory

2. Remove tomcat el-api.jar from /lib folder

3. copy the EL-2.2 jars into ./lib (el-api-2.2.jar, el-impl-2.2.jar)

4. Make sure that your WAR does not contain those two EL jars. You might need to modify the dependencies Maven scopes in your pom.xml.

4.1. Showcase Application structure

The showcase application follows a standard Maven layout.

In next sections we are going to review the structure for:

- Java packages (src/main/java)
- Application resources (src/main/resources)
- Java test packages (src/test/java)
- Test resources (src/test/resources)
- Webapp (src/main/webapp)

We will just comment the structure of the project quickly. We will not go in detail for every file.

Do not worry if you do not understand exactly what a configuration file is for. The important thing is that you are able to understand the project structure and we will go in detail in further sections.

4.1.1. Java packages

Following standard Maven layout, we place Java packages in src/main/java folder.

The table below shows the showcase application packages structure. This is the structure we recommend you to follow in your projects. This is the structure you would obtain if you used an Appverse Web archetype to generate a project.

All subpackages are included into main project package:

- Main project package: org.appverse.web.showcases.jsf2showcase

4.1.1.1. Backend and frontend subpackages

JSF2 runs completely in the server side like traditional JSP and servlet technology.

The AJAX support allows JSF2 components to make calls to the server without refreshing completely the page but in the end all the processing is performed on the server side and JSF2 returns the generated HTML code from the server side and the browser is able to refresh the DOM accordingly.

So, even though all the code runs in the server side, we distinguish between two main subpackages:

- backend: This package holds all classes related to the services layers, not directly connected to the GUI presentation. It contains all the classes that would be frontend (GUI framework) agnostic.
- frontend.jsf2: This package holds all classes directly connected to the GUI presentation. In this case we add jsf2 so that it is clear that the GUI front end is developed with JSF2.

The point of having this separation is that backend classes could be reusable in the event of migrating from one frontend technology to another or even reused in a multichannel or multifrontend application. The only layer that could be tied to a specific frontend is the Presentation Service layer (only if necessary). This is the key point that allows you to keep your business and integration layer completely agnostic and reusable regardless the frontend flavour.

4.1.1.1.1. Backend subpackages structure

Package	Subpackage	Description
[MAIN].backend.services	presentation	This package holds all presentation services interfaces.
	presentation.impl.live	This package holds all presentation services live implementations. Presentation Services are the only "backend" classes that might be dependant on frontend technology.
	business	This package holds all business services interfaces.
	business.impl.live	This package holds all business services live implementations. Business Services are Presentation Layer agnostic.
	integration	This package holds all integration services interfaces
	integration.impl.live	This package holds all integration services live implementations.
[MAIN].backend.model	presentation	This package holds all presentation model objects
	business	This package holds all business model objects
	integration	This package holds all integration model objects
[MAIN].backend.converters	p2b	This package holds all presentation to business converters. Converters work in both ways, converting model objects from presentation layer to business and the other way around.
	b2i	This package holds all business to integration converters. Converters work in two directions, converting business model objects from business layer to integration and the other way around.

Table 1 Backend subpackages structure

4.1.1.1.2. Frontend subpackages structure

Package	Subpackage	Description
[MAIN].frontend.jsf2	bean	This package holds all JSF2 managed beans. JSF2 managed beans acts as a controller, retrieving data model from presentation services, populating data model objects to forms and submitted data in forms to data model objects (data binding), validating data, handling error messages, etc, all this based on JSF2 lifecycle.
[MAIN].frontend.jsf2	helpers	This package holds helper classes, for instance ExceptionHandlerFactory and ExceptionHandler in order to have your custom exception management in the JSF2 layer. Feel free to create your own subpackages in 'helpers' folder. In this example there are just three classes but in real project it might make sense to group them in subpackages.

Table 2 Frontend subpackages structure

4.1.2. Application resources

Following standard Maven layout, we place resources in src/main/resources.

Let us review the resources folder structure:

Folder	Description	Contents
[ROOT FOLDER]	/src/main/resources root folder	Internacionalized application messages (messages_LANGUANGE_CODE.properties)
		JSR-303 internacionalized validation messages. By default JSR-303 requires them to be in this location (ValidationMessages_LANGUAGE_CODE.properties)
dozer	Contains Dozer mappings setup used for b2i and p2b converters.	b2i-bean-mappings.xml Business to integration dozer converters mappings.
		p2b-bean-mappings.xml Presentation to business dozer converters mappings
		common-bean-mappings.xml Common dozer converters setup

Folder	Description	Contents
Log4j	Log4j setup	log4j.properties
META-INF	Directory where the different providers require their setup to be stored	orm.xml JPA mapping file
		persistence.xml JPA file
		persistence-dll.xml File used by Appverse Web "appverse-web-tools-jpa-ddl-generator" Maven plugin. This plugin use this configuration file in order to generate the database Data Definition Language (DDL) scripts automatically from the annotated JPA entities.
		/skins Folder that holds RichFaces4 custom skins. File names must follow the pattern: [skin-Name].skin.properties.
properties	Application property files	db.properties Database connection info. The real values are over-ridden in this file by Maven depending on the current Maven profile. This makes distribution for different environments easier.
spring	Spring framework config files	application-config.xml Spring context configuration file. Includes specific database config and security files.
		HSQLDB_database-config.xml Default HSQLDB in-memory database out of the box setup.
		security-security.xml Spring security specific setup file
sql	Contains automatically generated Data Definition Language (DDL) scripts by Appverse Web "appverse-web-tools-jpa-ddl-generator" Maven plugin.	

Table 3 Resources folder structure

4.1.3. Java test packages

TODO: There are not test in the showcase yet

4.1.4. Test resources

TODO: There are not test in the showcase yet

4.1.5. Webapp folder

Following Maven standard layout, web application sources are stored in /src/main/webapp folder.

Let us review the structure:

Folder	Description	Contents
[ROOT FOLDER]	This is the standard Maven layout for web application sources.	Facelets without specific mapping in faces-config.
/errorPages	Error message facelets	Contains facelets to show error messages, both for HTTP errors automatically handled (defined in web.xml) and application internal errors (exceptions, for example).
/resources	Folder holding static content	/css Contain CSS
		/images Contain images
		/yaml In this showcase we have used YAML for basic flexible layout. This folder contains all static resources generated by YAML. Of course, it is optional to use it in your projects.
		/samplePages This folder contains facelets used in the showcase application as a kind of "live" tutorial showing basic JSF2 facelets development.
		/sourcePages This folder contains *.txt files containing a copy of the up-to-date source code of some components to show in the application as part of the "live" tutorial
		/template Contains the application template based on face-

Folder	Description	Contents
		lets templating system. In this case there is just one template as all the pages in the showcase have the same layout but you could create subfolders here and have templates for different "screen" layouts.
/WEB-INF	Standar WEB-INF directory for web applications	web.xml Deployment descriptor file
		faces-config.xml JSF2 configuration file

Table 4 Webapp folder

5.1. Maven project configuration

The JSF2 showcase application follows a standard Maven project layout.

Let us review the most important things to know.

5.1.1. Configured servlet containers

In previous sections we have explained you how to build, deploy and run the application either in an embedded Jetty container or external Tomcat 7 container.

Let us review the plugins configuration:

5.1.1.1. Java compilation version in Maven Compiler Plugin

By default, the project Java compilation version is 1.6. If you needed to change it, the maven-compiler-plugin allows you to specify the "source" and "target" compiler options:

```
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
        <configuration>
                <source>1.6</source>
                <target>1.6</target>
                <showWarnings>true</showWarnings>
                <showDeprecated>true</showDeprecated>
                <encoding>UTF-8</encoding>
        </configuration>
</plugin>
```

5.1.1.2. Tomcat 7 container

The Maven setup is:

```
<plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.1</version>
        <configuration>
                <username>${tomcat.manager.username}</username>
                <password>${tomcat.manager.password}</password>
                <server>tomcat</server>
                <path>/${app.context.root}</path>
                <url>${tomcat.manager.url}</url>
                <update>true</update>
        </configuration>
</plugin>
```

The setup is quite straightforward. The only thing you need to take into account is that you need the tomcat manager url, username and password to be properly defined in an active Maven profile, for instance:

```
<properties>

  <tomcat.manager.username>admin</tomcat.manager.username>

  <tomcat.manager.password>admin</tomcat.manager.password>

  <tomcat.manager.url>http://localhost:8080/manager/text/deploy</tomcat.manager.url>
```

```
</properties>
```

5.1.1.3. Jetty container

The Maven setup is pretty straightforward and in this case you do not have to do any extra configuration unless you want to modify the default one:

```
        <plugin>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-maven-plugin</artifactId>
                <configuration>
                        <webAppConfig>
                                <contextPath>/${app.context.root}</contextPath>
                        </webAppConfig>
                        <webAppSourceDirecto-
ry>${basedir}/src/main/webapp</webAppSourceDirectory>
                        <webXml>${basedir}/src/main/webapp/WEB-INF/web.xml</webXml>
                        <classesDirectory>target/appverse-web-showcases-jsf2-
${project.version}/WEB-INF/classes</classesDirectory>
                        <reload>automatic</reload>
                        <scanIntervalSeconds>2</scanIntervalSeconds>
                        <scanTargets>
                                <scanTarget>src/main</scanTarget>
                        </scanTargets>
                        <scanTargetPatterns>
                                <scanTargetPattern>
                                        <directory>src/main/resources</directory>
                                        <includes>
                                                <include>**/*.xml</include>
                                                <include>**/*.properties</include>
                                        </includes>
                                </scanTargetPattern>
                        </scanTargetPatterns>
                </configuration>
        </plugin>
```

Please note that Jetty is set up to automatically reload changes every 2 seconds. This is very useful for development.

5.1.2. Appverse Web JPA DLL generator plugin

Appverse Web JPA DLL generator plugin generates DLL database scripts automatically when you build your project if you are using EclipseLink as a JPA provider for your choosen platform.

This tool is very useful especially for development when you are working with an in-memory database as this ensures that your database is always kept up-to-date when you make changes in your JPA annotated model objects.

The script is generated in the configured “ddlOutputDir” and by default when you use an in-memory database, this is used to initialize your database.

```
        <plugin>
                <groupId>org.appverse.web.tools.jpaddlgenerator</groupId>
                <artifactId>appverse-web-tools-jpa-ddl-generator</artifactId>
                <version>1.0.4-SNAPSHOT</version>
                <executions>
                        <execution>
                                <goals>
                                        <goal>generate-schema</goal>
                                </goals>
                                <phase>prepare-package</phase>
                                <configuration>
                                        <ddlOutputDir>target/appverse-web-showcases-jsf2-
${project.version}/WEB-INF/classes/sql</ddlOutputDir>
```

```
                        </configuration>
                    </execution>
                </executions>
                <dependencies>
                    <dependency>

        <groupId>org.appverse.web.framework.modules.backend.core.persistence</groupId>
                        <artifactId>appverse-web-modules-backend-core-
persistence</artifactId>
                        <version>1.0.4-SNAPSHOT</version>
                    </dependency>
                </dependencies>
        </plugin>
```

5.1.3. Native to ASCII maven plugin

This plugin converts files with characters in any supported character encoding to one with ASCII and/or Unicode escapes. We aply this plugin to messages and validations property file to ensure a correct encoding:

```
        <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>native2ascii-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <goals>
                            <goal>native2ascii</goal>
                        </goals>
                        <configuration>
                            <src>${basedir}/src/main/resources</src>
                            <dest>target/appverse-web-showcases-jsf2-
${project.version}/WEB-INF/classes</dest>
                            <encoding>UTF8</encoding>
                            <includes>
                                <include>**/*.properties</include>
                            </includes>
                        </configuration>
                    </execution>
                </executions>
        </plugin>
```

5.1.4. Appverse Web dependencies

In order to setup Appverse Web dependencies, we include the following code in "dependencyManagement" section:

```
        <dependencyManagement>
                <dependencies>
                    <!-- Appverse dependencies (BOM) -->
                    <dependency>
                        <groupId>org.appverse.web.framework.poms</groupId>
                        <artifactId>appverse-web-masterpom</artifactId>
                        <version>1.0.4-SNAPSHOT</version>
                        <type>pom</type>
                        <scope>import</scope>
                    </dependency>
                </dependencies>
                ….
        </dependencyManagement>
```

This "Bill of materials" setup adds default Appverse Web managed dependencies.

On the other hand, in the "dependencies" section we will need to add the Appverse Web dependencies that our project needs. Take into account that the fact that dependency versions are "managed" by a BOM ("bill of materials") avoids the need of specifyng the version of the dependencies to be in-

cluded as the BOM will determine the right one. Please, do not specify any version in Appverse Web dependencies in order to guarantee right depencendy setup.

In this case our Appverse Web dependencies are:

```
<dependencies>
…
        <dependency>
                <groupId>org.appverse.web.framework.modules.backend.core.api</groupId>
                <artifactId>appverse-web-modules-backend-core-api</artifactId>
                <scope>compile</scope>
                <exclusions>
                        <exclusion>
                                <artifactId>servlet-api</artifactId>
                                <groupId>javax.servlet</groupId>
                        </exclusion>
                </exclusions>
        </dependency>

        <dependency>
               <groupId>org.appverse.web.framework.modules.backend.core.persistence</groupId>
               <artifactId>appverse-web-modules-backend-core-persistence</artifactId>
              <scope>compile</scope>
        </dependency>

        <dependency>
                <groupId>org.appverse.web.framework.modules.backend.core.ws</groupId>
                <artifactId>appverse-web-modules-backend-core-ws</artifactId>
                <scope>compile</scope>
                <exclusions>
                        <exclusion>
                                <artifactId>servlet-api</artifactId>
                                <groupId>javax.servlet</groupId>
                        </exclusion>
                </exclusions>
        </dependency>
  …
</dependencies>
```

5.1.5. JSF2 – Richfaces 4 configuration dependencies

In order to setup Richfaces 4 dependencies we also need to add a "Bill of materials" in our Maven set-up.

This "Bill of materials" from Richfaces will manage the JSF2 dependencies versions to be included.

```
        <dependencyManagement>
                <dependencies>
                        <!-- JSF2 richfaces dependencies (BOM) -->
                        <dependency>
                                <groupId>org.richfaces</groupId>
                                <artifactId>richfaces-bom</artifactId>
                                <version>${org.richfaces.bom.version}</version>
                                <scope>import</scope>
                                <type>pom</type>
                        </dependency>
                </dependencies>
        </dependencyManagement>
```

In the "dependencies" section we will need to add the JSF2 dependencies that our project needs. Take into account that the fact that dependency versions are "managed" by a BOM ("bill of materials") avoids the need to specify the version of the dependencies to be included as the BOM will determine the right one. Please, do not specify any version in JSF2 dependencies in order to guarantee right dependency setup.

In this case our case JSF2 dependencies are:

```
<dependencies>
..
        <dependency>
                <groupId>org.richfaces.ui</groupId>
                <artifactId>richfaces-components-ui</artifactId>
        </dependency>
        <dependency>
                <groupId>org.richfaces.core</groupId>
                <artifactId>richfaces-core-impl</artifactId>
        </dependency>
        <dependency>
                <groupId>javax.faces</groupId>
                <artifactId>javax.faces-api</artifactId>
                <scope>provided</scope>
        </dependency>
        <dependency>
                <groupId>org.glassfish</groupId>
                <artifactId>javax.faces</artifactId>
                <scope>compile</scope>
        </dependency>
        <dependency>
                <groupId>javax.servlet</groupId>
                <artifactId>javax.servlet-api</artifactId>
                <scope>provided</scope>
        </dependency>
        <dependency>
                <groupId>javax.servlet.jsp</groupId>
                <artifactId>jsp-api</artifactId>
                <scope>provided</scope>
        </dependency>
        <dependency>
                <groupId>javax.el</groupId>
                <artifactId>el-api</artifactId>
                <scope>provided</scope>
        </dependency>
..
</dependencies>
```

5.1.6. EL 2.2 dependency

El 2.2 starts to be an individual part of the Java EE specification since Java EE version 6.

This means that Java EE full compliant application servers will provide both the specification and implementation of this EL version. In not full compliant Java EE version 6 application servers / servlets containers we need to make sure we are including this version as a dependency.

Take into account that in order to use certain high benefitial JSF2 features El 2.2 is required. For instance, EL expressions for actions with parameters will fail to be parsed for versions previous to EL 2.2, for instance:

```
<h:commandLink action="#{languageBean.changeLanguage('en')}">
```

For not full Java EE 6 application servers we will include the EL 2.2 reference implementation dependency as follows:

```
<dependencies>
..
        <dependency>
                <groupId>org.glassfish.web</groupId>
                <artifactId>el-impl</artifactId>
                <version>2.2</version>
        </dependency>
..
</dependencies>
```

5.1.7. HSLQ DB dependency

We include in the project POM a convenient HSQL DB setup that it is very useful in development and continuos integration test environments. Using an in-memory database in these environments make development environments lighter and running continuos integration tests faster.

This is the dependency inclusion:

```
<dependencies>
..
        <dependency>
                <groupId>org.hsqldb</groupId>
                <artifactId>hsqldb</artifactId>
                <scope>runtime</scope>
        </dependency>
..
</dependencies>
```

5.2. JSF2 and Spring contexts configuration

Let us review the basic setup in our web.xml showcase application file.

Setup for specific aspects such as skinning, automatic HTTP errors management, etc, will be explained in separate sections.

Please note that the order of some elements defined in web.xml matters. If you have any doubt in this regard, please have a look at the web.xml file included in the example directly.

5.2.1. Java Server Faces context setup

```
<!-- Necessary for Jetty server to work, for other servers / servlet containers it is not nec-
essary -->
<listener>
        <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
</listener>

<!-- Faces context initialization -->

<!--  RichFaces 4 Skinning setup -->
<context-param>
        <param-name>org.richfaces.skin</param-name>
        <param-value>#{skinBean.skin}</param-value>
</context-param>
<context-param>
        <param-name>org.richfaces.enableControlSkinning</param-name>
        <param-value>true</param-value>
</context-param>

<!-- Specifying project stage -->
<context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
</context-param>

<!-- Default sufix for pages containing JSF2 code -->
<context-param>
        <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
        <param-value>.xhtml</param-value>
</context-param>
<context-param>
        <param-name>javax.faces.FACELETS_VIEW_MAPPINGS</param-name>
        <param-value>*.xhtml</param-value>
```

```
</context-param>

<!-- Skip comments in JSF pages -->
<context-param>
        <param-name>javax.faces.SKIP_COMMENTS</param-name>
        <param-value>true</param-value>
</context-param>
<context-param>
        <param-name>javax.faces.FACELETS_SKIP_COMMENTS</param-name>
        <param-value>true</param-value>
</context-param>

<!-- Load custom tags libraries into JSF web application -->
<context-param>
        <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
        <param-value>/WEB-INF/rich-appverse.taglib.xml</param-value>
</context-param>

<!-- Faces Servlet mappings -->
<servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
<servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
```

5.2.2. Spring context setup

```
<!--  Log4j Setup -->
<listener>
        <listener-class>org.springframework.web.util.Log4jConfigListener</listener-class>
</listener>
<context-param>
        <param-name>log4jConfigLocation</param-name>
        <param-value>classpath:log4j/log4j.properties</param-value>
</context-param>

<!-- Spring listeners setup -->
<listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<listener>
        <listener-
class>org.springframework.web.context.request.RequestContextListener</listener-class>
</listener>
<listener>
        <listener-class>org.springframework.web.util.IntrospectorCleanupListener</listener-
class>
</listener>

<!-- Spring context config file include -->
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring/application-config.xml</param-value>
</context-param>

<!-- Spring security filters setup -->
<filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

As we have commented earlier, Appverse Web sees JSF2 as a MVC implementation and GUI development framework. Appverse Web is quite flexible: as it works with standard technologies it is not restrictive regarding technologies integration. However, we would like to provide some best practices and recommendations based on what we think it is better.

6.1. JSF2 – Spring integration options and recommendation

The fact that JSF2 context (FacesContext) and Spring context (ApplicationContext) are separately and they do not "see" each other by default, makes the way you integrate JSF2 with Spring an important decision. At the end on the day what you need to take care of is how you use Spring managed beans in JSF2 Managed beans.

Spring 2.5+ introduces SpringBeanFacesELResolver class that allows JSF2 pages to access beans managed by Spring (in Spring contexts) using EL (Expression Language) and allows JSF2 "Managed Beans" to have "managed properties" that are Spring beans using EL.

Another important aspect to take into account to choose an integration strategy is to take into account that JSF2 manages its beans inside its defined lifecycle and bean scopes. On the other hand, Spring provides its own bean scopes and they do not match all JSF2 scopes.

In the table below you can see the default scopes of JSF2 and Spring and their equivalence:

JSF2 scopes	Spring scopes
Request	Request
Session	Session
Application	Singleton
View	-
Flash	-
-	Prototype
-	Global Session
Custom Implementation	Custom Implementation

We will explain you the three options we have considered to use Spring Managed beans in JSF2 Managed bean and what is our recommendation.

6.1.1. Using pure JSF2 annotations in "Manages Beans" and Spring context for Services beans

This approach uses pure JSF2 for the GUI management with JSF2 "Managed Beans" in their own context whilst relying on the Spring context for Services. Services called from the JSF2 layer are managed by Spring.

SpringBeanFacesELResolver allows you to "inject" beans managed by Spring corresponding to Services as "managed properties".

This is the cleaner way if you want to have a completely separated JSF2 layer and Spring services layer and you do not want to mix them.

The most important advantatge is that you are using the default JSF2 lifecycle and so, you can use all the JSF2 scopes by default, without further development or maintenance.

Note: View scope (to keep the same bean while the application is not replacing a view with another) and Flash (to transfer data between a view to another) are really necessary to use the narrower scope possible and keep your applications as lighter as possible. For instance, use of Session scope is discouraged unless the use is really justified, especially in applications that are mainly stateless.

Example, with scope view:

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ManagedProperty;
import javax.faces.bean.ViewScoped;

import org.appverse.web.showcases.jsf2showcase.backend.services.presentation.UserServiceFacade;

@ManagedBean
@ViewScoped
public class UsersManagementBean implements Serializable {
……
    @ManagedProperty(value="#{userServiceFacade}")
    private UserServiceFacade userServiceFacade;

    public void setUserServiceFacade(UserServiceFacade userServiceFacade) {
        this.userServiceFacade = userServiceFacade;
    }
}
```

6.1.2. Using pure Spring annotations in “Manages Beans” so they are managed by Spring context

This approach would be perfect if Spring bean scopes and JSF2 were aligned, but they are not as you have seen in the table above.

If you go this way, you could only use the Spring scopes with a clear JSF2 scope match by default.

If you want to use View and Flash JSF2 scopes, you will need to implement custom Spring scopes. This could be a good solution but it implies to develop at least this two scopes in Spring and maintain them according to Spring and JSF2 versions changes. This is error prone and it has a maintenance cost.

The advantatge would be that you would use Spring annotations for everything and default Spring DI in “managed beans”:

Example, with scope request:

```
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Scope;

import org.appverse.web.showcases.jsf2showcase.backend.services.presentation.UserServiceFacade;

@Component
@Scope("request")
public class UsersManagementBean implements Serializable {
…..

       @Autowired
       private UserServiceFacade userServiceFacade;
```

```
        public void setUserServiceFacade(UserServiceFacade userServiceFacade) {
              this.userServiceFacade = userServiceFacade;
        }
}
```

6.1.3. Mixing both solutions: using JSF2 @ManagedBean and @ManagedProperty annotations mixed with Spring annotations

Hundreds of forum threads have been opened in the attempt of mixing the two previous solutions. However, we discourage projects to implement it even if this in theory might work and some developers propose this as a full working solution (allowing you to use all JSF2 Scopes).

Other developers consider that this solution represents bad practices because instead of using Spring provided way to do the integration (by using SpringBeanFacesELResolver to inject properties to beans using EL) what you are doing is taking the best of the previous two approaches mixing Spring and JSF2 annotation.

Appverse Web development team shares this last opinion. As we have commented the way we see that JSF2 fits in Appverse Web is as an MVC implementation and GUI component library. We use it just as a presentation layer. Our service layer is based on Spring and it could be reusable among different frontend technologies. We recommend you to make this separation clear and not to mix Spring and JSF2 annotations in your managed beans.

6.1.4. Summary: recommended option

In the previous section we have explained why we discourage mixing JSF2 and Spring annotation in managed beans.

The second approach (using pure Spring annotations in managed beans) has the advantatge of being able to use full Spring annotation and DI, but from our point of view, this does not overweight the maintenance cost and risk of developing custom Spring scopes just to save a bit more code (having to add a setter for the services to be injected).

So, our recommendation is to use pure JSF2 in presentation layer using “official” provided way to integrate Spring with JSF2, injecting Spring services as managed properties using EL expressions. This showcase application implementation is based on this choice.

7.1. Database setup

Please find below the details explaining how to set up the database connection.

7.1.1. HSQLDB In-memory database

By default, the project comes with a HSQLDB in-memory database setup out of the box. As we have explained earlier in this document, this database will be very useful especially for early phases of development (at least until you have a first stable version of your JPA model) and for continuos integration tests.

See the import of the file containing specific database configuration imported in Spring application-context.xml file:

```
        <import resource="HSQLDB_database-config.xml" />
```

Let us see HSQLDB_database-config.xml file content:

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:p="http://www.springframework.org/schema/p"
xmlns:jdbc="http://www.springframework.org/schema/jdbc"
           xsi:schemaLocation=" http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
           http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd "
           default-autowire="byName">

      <bean id="jpaVendorAdapter"
                class="org.springframework.orm.jpa.vendor.EclipseLinkJpaVendorAdapter"
                    p:databasePlatform="org.eclipse.persistence.platform.database.HSQLPlatform"
                    p:showSql="true" />
      <!-- ============================================================================== -->
      <!-- Data Source                                                                    -->
      <!-- ============================================================================== -->

           <jdbc:embedded-database id="dataSource">
               <jdbc:script location="classpath:/sql/HSQLPlatform-schema-create.sql"/>
               <jdbc:script location="classpath:/sql/HSQLPlatform-dml-create.sql"/>
       </jdbc:embedded-database>

</beans>
```

Please note that jpaVendorAdapter contains the EclipseLink JPA provider setup. In this case the database platform is the corresponding to HSQLDB.

The Spring jdbc:embedded-database tag contains in-memory database setup. It specifies the DDL script to create the database (HSQLPlatform-schema-create.sql) and a DML script where you can initialize the database data.

Please remember that HSQLPlatform-schema-create.sql is autogenerated every time you build the project according to the annotations in your JPA integration model objects. This ensures that your database is always up-to-date at least in development environments using in-memory database. You can use the autogenerated script for production-like environments as well.

7.1.2. Other databases

Using an in-memory database is optional, even though we think is a good practice to use it in development and automated test environments (project health / continuos integration).

Of course, you will need a different setup for production-like environments. Let us see how to setup the database in those cases.

Imagine you have the in-memory database setup we have just review for development environments and you have a MySQL database setup for production like environments.

We recommend you to have another database setup file in /src/main/resources/spring. For instance MYSQL_database-config.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:p="http://www.springframework.org/schema/p"
xmlns:jdbc="http://www.springframework.org/schema/jdbc"
           xsi:schemaLocation=" http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd "
           default-autowire="byName">

    <!-- ========================================================================  -->
    <!-- JPA ORM                                                                    -->
    <!-- ========================================================================  -->
    <bean id="jpaVendorAdapter"
                class="org.springframework.orm.jpa.vendor.EclipseLinkJpaVendorAdapter"
                p:databasePlatform="org.eclipse.persistence.platform.database.MySQLPlatform"
                p:showSql="true" />

    <!-- ======================================================================== -->
    <!-- Data Source                                                              -->
    <!-- ======================================================================== -->
    <bean id="dataSource"
          class="org.apache.commons.dbcp.BasicDataSource"
          p:driverClassName="${db.driverClassName}" p:url="${db.url}"
p:username="${db.username}" p:password="${db.password}"
                p:initialSize="${db.poolInitialSize}" p:maxActive="${db.poolMaxActive}"
p:minIdle="${db.poolMinIdle}" p:maxIdle="${db.poolMaxIdle}"
                p:testOnBorrow="${db.testOnBorrow}"  p:testWhileIdle="${db.testWhileIdle}"
p:timeBetweenEvictionRunsMillis="${db.timeBetweenEvictionRunsMillis}"
                p:validationQuery="${db.validationQuery}"/>

</beans>
```

Note that in this case the jpaVendorAdapter setup contains the MySQL platform.

In this case the datasource setup (as a Spring bean) contains the different parameter required for the datasource setup. In this example they appear as variables (for instance ${db.url}) because this allows your to define different setups for different environments (for instance integration database, UAT database, production database) and use Maven profiles and filters in order to replace the right values automatically for the desired environment. See next section explaining how to do this in detail.

Please, take into account that this is a showcase example. For production-like environments it will be very common that you want to register your datasources in the JNDI registry. Of course, this option is possible and recommended in most of the cases. However, how to do this is out of the scope of this document.

7.1.3. Using Maven filters and profiles for different environments database setup

Explaining in detail all the possibilities that Maven offers is out of the scope of this document. However, we would like to give you some recommendations based on a Maven very interesting feature. Of course, feel free to adapt this recommendation to your needs.

It is very common during software development lifecycle to have different environments. Typically environments associated to development stages are: development environment, integration environment, UAT environment, production environment and continuous integration environment, but you might have other environments according to the needs of your project.

It is common as well that most of the environments are production-like but using a different database. Continuos integration environments typically use in-memory databases whilst production-like use different databases managed by different DBMS.

In this scenario it is highly desirable to have an automated system to build the project with the correct setup according to the environment. This will save you the time of doing the changes manually and makes the possibility of human mistakes smaller.

In this showcase we have added to example profiles that use different database configuration:

- 'development' profile : default in-memory development profile
- 'integration' profile : example 'integration' environment profile (production like with MySQLDatabase)

In order to deal with both profiles we add a Maven profile to select a different property file containing the database configuration according to the profile in our POM.xml:

```
        <filters>
                <filter>src/main/filters/${env}.properties</filter>
        </filters>
```

The Maven variable ${env} takes a different value in each profile definition:

```
<profiles>
        <!-- local development profile example -->
        <profile>
                <id>development</id>
                <activation>
                        <activeByDefault>true</activeByDefault>
                </activation>
                <dependencies>
                        <dependency>
                                <groupId>org.hsqldb</groupId>
                                <artifactId>hsqldb</artifactId>
                                <scope>runtime</scope>
                        </dependency>
                </dependencies>
                <properties>
                        <env>development</env>
                </properties>
        </profile>
        <!-- integration (production-like) profile example -->
        <profile>
                <id>integration</id>
                <activation>
                        <activeByDefault>false</activeByDefault>
                </activation>
                <dependencies>
                        <dependency>
                                <groupId>mysql</groupId>
                                <artifactId>mysql-connector-java</artifactId>
                        </dependency>
                </dependencies>
```

```
                <build>
                        <plugins>
                                <plugin>
                                        <groupId>
                                             org.appverse.web.tools.jpaddlgenerator
                                        </groupId>
                                        <artifactId>appverse-web-tools-jpa-ddl-
generator</artifactId>
                                       <configuration>
                                               <targetDbPlat-
form>MySQLPlatform</targetDbPlatform>
                                       </configuration>
                                </plugin>
                        </plugins>
                </build>
                <properties>
                        <env>integration</env>
                </properties>
        </profile>
```

You can active a profile by default or you can specify a profile when you run a Maven goal either using Maven command line or your Maven IDE integration.

With this setup you can have different property files with the database setup per environment in the directory src/main/filters. For instance:

- src/main/filters/development.properties
- src/main/files/integration.properties

The structure of the property files is as follows. There you would fill in the database connection required parameters:

```
# Database Connection Properties
env.db.url=
env.db.username=
env.db.password=

env.db.poolInitialSize=2
env.db.poolMaxSize=10
env.db.poolMinIdle=2
env.db.poolMaxIdle=4
```

7.2. Authentication and authorization

Authorization and authentication is implemented with Spring Security.

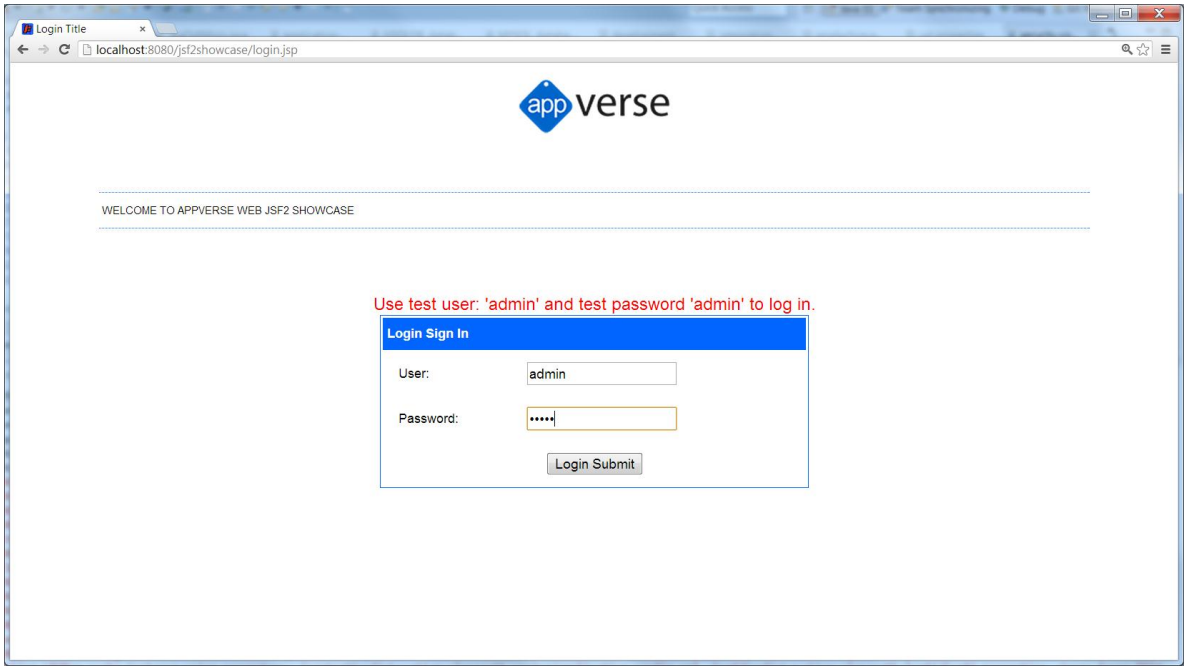
This is a showcase so the authentication is simple: based form authentication with an authentication manager delegating in an in-memory authentication provider. The in-memory authentication provider holds authentication and authorization data for every user: user name, password and list of authorities. This allows you to implement both your active and reactive security in your application based on this information.

Something great about Spring security is that you can replace an authentication manager setup easily replacing an authentication provider by another. This would allow you easily implement your own authentication provider (for instance LDAP directory or a database) to retrieve users authentication and authorization credentials. The big advantage here is that Spring Security treats security as an aspect. This implies that replacing authentication providers and changing Spring security configuration a bit is enough in order to change the way users are authenticated/authorized. Active and reactive security checks will remain unchanged in your application source code.

Let us see how authentication and authorization are implemented in this showcase.

7.2.1. Authentication form

Let us start having a look at the login form. Form based authentication requires the users to introduce their user names and passwords to log in the application. In our case, the login screen looks like:



In this example if the users are not logged in, they will be automatically redirected to the login page.

Our login page is based on Spring base form authentication. As long as the form action is "`j_spring_security_check`" and it contains "`j_username`" and " `j_password`" Spring will be able to authenticate and retrieve the list of authorities for this user.

As this showcase should be kept simple, we have implemented the login form with a JSP page. This is the only JSP you will find in the application as once logged in the application everything will be JSF2 facelets. In a real application we recommend you to develop the login page as a facelet.

You can have a look at src/main/webapp/login.jsp to see the implementation details.

7.2.2. Spring security setup

Spring security config file is located at: /src/main/resources/spring/security-config.xml.

Let us review its configuration.

The http section below specifies security to apply to different URLs. Note that all URLs will require "ROLE_USER" with a few exceptions: one is the own login page that has anonymous access. Otherwise it will not possible to access to the own login page. The images shown in the login page have anonymous authentication as well.

Please note that the 'form-login' section specifies the login page to show in case a user is not authenticated and the URL to redirect in case of failure which - in our case - just includes a parameter that we use to know that we have to show a notification in the login page.

Finally, you can see how to specify the URL to go after the user logs out. In our case when the user logs out is redirected to the login page.

```
  <http auto-config='true'>
    <intercept-url pattern="/login.jsp*" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
    <intercept-url pattern="/favicon.ico" access="IS_AUTHENTICATED_ANONYMOUSLY" />
    <intercept-url pattern="/logo_admin.png" access="IS_AUTHENTICATED_ANONYMOUSLY" />
    <intercept-url pattern="/**" access="ROLE_USER" />
    <form-login login-page='/login.jsp' authentication-failure-url="/login.jsp?error=failed" />
    <logout logout-success-url="/login.jsp"/>
  </http>
```

`global-method-security` annotation allows you to apply default security to all beans matching the corresponding pattern. In this example, for instance, we have three differents patterns matching every service layer separately and by default all services require the user to have the role "ROLE_USER" at least.

```
  <global-method-security secured-annotations="enabled" pre-post-annotations="enabled">
          <protect-pointcut

            expression="execution(*
org.appverse.web.showcases.jsf2showcase.backend.services.presentation..*.*(..)))"
            access="ROLE_USER"/>
          <protect-pointcut

            expression="execution(*
org.appverse.web.showcases.jsf2showcase.backend.services.business..*.*(..)))"
            access="ROLE_USER"/>
          <protect-pointcut

            expression="execution(*
org.appverse.web.showcases.jsf2showcase.backend.services.integration..*.*(..)))"
            access="ROLE_USER"/>
  </global-method-security>
```

Due to your application requirements, you might be interested in securing a method with specific roles. You can do this, directly in your services interfaces like in this example:

```
public interface UserServiceFacade extends IPresentationService {

        @PreAuthorize("hasRole('ROLE_USER')")
        UserVO loadUser(long userId) throws Exception;

        @PreAuthorize("hasRole('ROLE_USER')")
        List<UserVO> loadUsers() throws Exception;

        @PreAuthorize("hasRole('ROLE_USER')")
        long saveUser(UserVO userVO) throws Exception;

        @PreAuthorize("hasRole('ROLE_USER')")
        PresentationPaginatedResult<UserVO> loadUsers(
                        PresentationPaginatedDataFilter config) throws Exception;

        @PreAuthorize("hasRole('ROLE_USER')")
        void deleteUser(final UserVO userVO) throws Exception;
}
```

In this example, every method requires the user to have the role "ROLE_USER" in order to be invoked. In this example it does not makes much sense as the global-method-security default setup requires the same rol. It is just an example. Take into account that you could do a specific method more restrictive just adding more condicitions in the @PreAuthorize annotation.

Let us see the configuration required in order to add users and credentials to our in-memory authentication provider. These kind of providers are normally used for testing and showcases:

```
<!--  Authentication manager with test (in-memory) authentication provider  -->
<authentication-manager alias="authenticationManager">
  <authentication-provider>
    <user-service>
      <user name="admin" password="admin" authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
</authentication-manager>
```

7.3. Internacionalization

How to implement Internacionalization is completely self-explained in the corresponding menu option. You can access directly to "Internacionalization" menu option or go directly accessing this URL:

http://localhost:8080/jsf2showcase/internationalization.jsf

7.4. Errors and exception handling

In this section we show different error and exception handling scenarios.

7.4.1. Automatic HTTP Error handling

HTTP errors can be automatically handled by the web application including the right setup in the web.xml file.

This showcase is self-explained in the own showcase application. In order to see how to setup automatic HTTP error handling you can click the menu option:

Exception Handling -> Error Pages -> Automatic HTTP error handling

Or directly go to the showcase page:

http://localhost:8080/jsf2showcase/errGeneratorErrCod.jsf

Here you will find an explanation and you will be able to try the automatic HTTP Error handling with two different HTTP errors that are deliberately caused.

7.4.2. Application exception handler implementation

Both checked and unchecked application exceptions need to be handled so that the user can see exception information in a controlled way according to the project requirements. A use case example would be showing the exception technical information to the user so they can contact to the IT team.

We recommend every application to implement its own centralized exception handler according to the project requirements. In this showcase we show you an example of how to do it.

This showcase is self-explained in the own showcase application. In order to see how develop your own exception handler click the menu option:

Exception Handling -> Application exception handler implementation

Or directly go to the showcase page:

http://localhost:8080/jsf2showcase/errGeneratorCode.jsf

If you want to see a working example of the application exception handler, both for checked and unchecked exceptions, have a look at the following sections.

7.4.3. Unchecked exceptions

Unchecked exceptions are handled by the application exception handler.

You can see this working with two different deliberately caused coding errors clicking the menu option: *Exception Handling -> Unchecked Exceptions*

Or accessing directly to the showcase page:

http://localhost:8080/jsf2showcase/errGeneratorUnch.jsf

7.4.4. Checked exceptions

Checked exceptions are handled by the application exception handler.

You can see this working with two different deliberately caused coding errors clicking the menu option: *Exception Handling -> Checked Exceptions*

Or accessing directly to the showcase page:

http://localhost:8080/jsf2showcase/errGeneratorChck.jsf

7.5. Validations

7.5.1. JSR-303 (Bean Validation) Integration

Java Community Process (JCP) defines JSR-303 specification which, according to JCP, "defines a meta-data model and API for JavaBeanTM validation based on annotations, with overrides and extended meta-data through the use of XML validation descriptors".

You can read specification details here: http://jcp.org/en/jsr/detail?id=303

The fact that you can annotate beans with validation metadata implies that you only need to annotate the bean properly and everytime the bean is validated it will be in the same way regardless of the process invoking the validation method. Besides, this is a standard specification that most of well proof and known frameworks support. This makes your solutions easier to migrate if needed. For these reasons, Appverse Web recommends using JSR-303 validations. However, Appverse Web does not set any technical restriction on how to implement validations since you could use proprietary validation mechanisms (JSF2 custom validators mechanism, for instance).

In particular, JSF2 supports JSR-303 validation.

7.5.1.1. Client and server double validation

The fact that Spring and Richfaces provide integration support for JSR-303 gives you the chance to implement a safe double-validation system.

Richfaces enrich JSF2 validation features adding the possibility of field validation in the client side. This is very useful because if the data is not correct no request is sent to the server. Besides server side validation when the user submits the data is supported.

We recommend you to implement a double validation system for security reasons.

Client validation is great because improves performance and makes the UI richer. In the event of an error, the user sees the error at the moment and there is no need to do an HTTP request to the server.

Server side validation is necessary because data is validated in the server and it ensures that if the data associated to the request has been modified by any mean it will not pass the server validatons.

A good practice is to apply validaton in your service layer methods. Just as an example, in this showcase we have set a Spring AOP advice in order to automatically validate objects in all presentation methods that start with "save".

7.5.1.2. RichFaces client side validation

Richfaces JSR-303 integration features are really well explained in Richfaces documentation, so if you want to know more, we encourage you to have a look:

http://docs.jboss.org/richfaces/latest_4_2_X/Component_Reference/en-US/html/chap-Component_Reference-Validation.html

7.5.1.2.1. Spring server side validation

Spring provides integration support for JSR-303. In order to include a "Validator" bean, just add this to your spring configuration (in our example in the file application-config.xml):

```
<bean id="validator"
        class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean" />
```

Then you are able to use the injected "validator" bean to validate your beans in the service layer.

7.5.1.2.2. Spring server side validation as an aspect (AOP)

Appverse Web provides a ValidationAdvice and ValidationManager that you can associate to a particular pointcut using Spring AOP in order to apply validation as an aspect.

For instance, imagine that for security reasons you would like all your presentation methods delegating on business layer to save data to validate beans before saving them. If all your presentation services methods that save data start with "save" you can define a pointcut matching this pattern so that automatically validate the data. This is what we have implemented in this showcase as an example. The setup in the application-config.xml file is:

```
        <bean id="validationManager"
class="org.appverse.web.framework.backend.api.aop.managers.impl.live.ValidationManagerImpl" />


        <aop:config>
                <aop:pointcut id="allPresentationSaveMethodsCalls" expression="execution(*
org.appverse.web.showcases.jsf2showcase.backend.services.presentation..save*(..))" />

                <aop:advisor advice-ref="validationAdvice" pointcut-
ref="allPresentationSaveMethodsCalls" />
        </aop:config>
```

The showcase includes two examples showing JSR-303:

- Simple bean validation: showing how to validate individual bean fields. You can access the showcase option through the menu options: Validation -> Simple Validations or directly going to the showcase page:

  http://localhost:8080/jsf2showcase/simpleValidation.jsf

- Cross field validation: showing how to validate relationships between bean fields. You can access the showcase option through the menu options: Validation -> Cross-field Validations or directly going to the showcase page:

  http://localhost:8080/jsf2showcase/crossFieldValidation.jsf

Both examples use the same VO JSR-303 UserVO.java bean.

7.6. Presentation and services layers integration

In order to show an end-to-end example, the JSF2 showcase includes an example of a RichFaces grid component that shows data stored in database using remote pagination. Even thought it is a simple example it will help you to understand how to develop the different components in every layer from UI development and presentation layer to business and integration layer.

In order to see the example live, please go through the menu options: Components -> Grid or go directly to the showcase page:

http://localhost:8080/jsf2showcase/userManagement.jsf

Let us review the different layers and components implementation.

We will start the explanation from the front-end UI and we will move to the services layer (presentation, business and integration in this order).

7.6.1. Facelets as UI View

The facelet including the template for this screen (userManagement.xhtml) looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:ui="http://java.sun.com/jsf/facelets"
        xmlns:a4j="http://richfaces.org/a4j"
        xmlns:rich="http://richfaces.org/rich"
        xmlns:fn="http://java.sun.com/jsp/jstl/functions"
        xmlns:rich-jsf2showcase="http://appverse-web-showcases-jsf2/facelets">

<ui:composition template="template/template.xhtml">
        <ui:define name="body">

                <rich-jsf2showcase:userGrid />

        </ui:define>
</ui:composition>
</html>
```

As you can see, the facelets use the same template used for the rest of application views.

We are using a facelet “userGrid” that is included inside an application facelet taglib we have defined. This allows us to reuse facelets as components in diffent screens. Please note that in this particular example, the `userManagement.xhtml` view is quite simple as it is just an example but in your views

you might have different sections that can be clearly identified and separated so that your views are easier to understand, to maintain and you can reuse some sections in different screens if necessary.

We show you the application facelets taglib definition in /WEB-INF/rich-jsf2showcase.taglib.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE facelet-taglib PUBLIC
  "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
  "http://java.sun.com/dtd/facelet-taglib_1_0.dtd">
<facelet-taglib>
        <namespace>http://appverse-web-showcases-jsf2/facelets</namespace>
        <tag>
                <tag-name>userGrid</tag-name>
                <source>../userGrid.xhtml</source>
        </tag>
</facelet-taglib>
```

Let us see the `userGrid.xhtml` code. Basically, this facelet is in charge for building the part of the UI corresponding to the user grid, supporting remote pagination. Richfaces offers the "extended-DataTable" component for this purpose. It is a very flexible component in terms of configuration. Please have a look at Richfaces documentation and showcase if you are interested in learning more about this component:

http://showcase.richfaces.org/richfaces/component-sample.jsf?demo=extendedDataTable&skin=blueSky

The `userGrid.xhtml` code looks like:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:ui="http://java.sun.com/jsf/facelets"
        xmlns:a4j="http://richfaces.org/a4j"
        xmlns:rich="http://richfaces.org/rich">

        <h:outputStylesheet library="css" name="default.css" />

        <h:form>

                <div class="title_head" style="margin-bottom: 1em;">
                        <h:outputText value="#{msg['pagingGrid.title']}" />
                </div>

                <rich:extendedDataTable value="#{userManagementBean.userVOList}"
                        sortMode="multi" var="user" id="table" selectionMode="none"
                        clientRows="#{usersSorterBean.clientRows}" iterationStatusVar="it"
                        sortPriority="#{usersSorterBean.sortPriorities}" cellpadding="0"
                        cellspacing="0" style="height:278px; width:791px; text-align:center"
                        rowClasses="odd-row, even-row">

                        <f:facet name="noData">
                No data found
                </f:facet>

                        <f:facet name="header">
                                <h:outputText value="#{msg['pagingGrid.tableTitle']}" />
                        </f:facet>
                        <rich:column width="131px" sortBy="#{user.id}"
                                sortOrder="#{usersSorterBean.sortsOrders['id']}" sort-
Type="custom">
                                <f:facet name="header">
                                        <a4j:commandLink execute="@this" value="User Id" ren-
der="table"
                                                action="#{usersSorterBean.sort}">
                                                <f:param name="sortProperty" value="id" />
```

```
                                        </a4j:commandLink>
                                        <h:graphicImage style="position:absolute;"
name="down_icon.gif"
                                                library="images"
                                                ren-
dered="#{usersSorterBean.sortsOrders['id']=='descending'}" />
                                        <h:graphicImage style="position:absolute;"
name="up_icon.gif"
                                                library="images"
                                                ren-
dered="#{usersSorterBean.sortsOrders['id']=='ascending'}" />
                                        <br />
                                </f:facet>
                                <h:outputText value="#{user.id}" />
                        </rich:column>
                        <rich:column width="131px" sortBy="#{user.name}"
                                sortOrder="#{usersSorterBean.sortsOrders['name']}" sort-
Type="custom">
                                <f:facet name="header">
                                        <a4j:commandLink execute="@this" value="Name" ren-
der="table"
                                                action="#{usersSorterBean.sort}">
                                                <f:param name="sortProperty" value="name" />
                                        </a4j:commandLink>
                                        <h:graphicImage style="position:absolute;"
name="down_icon.gif"
                                                library="images"
                                                ren-
dered="#{usersSorterBean.sortsOrders['name']=='descending'}" />
                                        <h:graphicImage style="position:absolute;"
name="up_icon.gif"
                                                library="images"
                                                ren-
dered="#{usersSorterBean.sortsOrders['name']=='ascending'}" />
                                        <br />
                                </f:facet>
                                <h:outputText value="#{user.name}" />
                        </rich:column>
                        <rich:column width="131px" sortBy="#{user.lastName}"
                                sortOrder="#{usersSorterBean.sortsOrders['lastName']}"
                                sortType="custom">
                                <f:facet name="header">
                                        <a4j:commandLink execute="@this" value="Last Name"
render="table"
                                                action="#{usersSorterBean.sort}">
                                                <f:param name="sortProperty" value="lastName"
/>
                                        </a4j:commandLink>
                                        <h:graphicImage style="position:absolute;"
name="down_icon.gif"
                                                library="images"
                                                ren-
dered="#{usersSorterBean.sortsOrders['lastName']=='descending'}" />
                                        <h:graphicImage style="position:absolute;"
name="up_icon.gif"
                                                library="images"
                                                ren-
dered="#{usersSorterBean.sortsOrders['lastName']=='ascending'}" />
                                        <br />
                                </f:facet>
                                <h:outputText value="#{user.lastName}" />
                        </rich:column>
                        <rich:column width="131px" sortBy="#{user.email}"
                                sortOrder="#{usersSorterBean.sortsOrders['email']}"
                                sortType="custom">
                                <f:facet name="header">
                                        <a4j:commandLink execute="@this" value="Email" ren-
der="table"
                                                action="#{usersSorterBean.sort}">
                                                <f:param name="sortProperty" value="email" />
                                        </a4j:commandLink>
                                        <h:graphicImage style="position:absolute;"
name="down_icon.gif"
                                                library="images"
                                                ren-
```

```
dered="#{usersSorterBean.sortsOrders['email']=='descending'}" />
                                        <h:graphicImage style="position:absolute;"
name="up_icon.gif"
                                                library="images"
                                                ren-
dered="#{usersSorterBean.sortsOrders['email']=='ascending'}" />
                                        <br />
                                </f:facet>
                                <h:outputText value="#{user.email}" />
                        </rich:column>
                        <rich:column width="131px" sortBy="#{user.password}"
                                sortOrder="#{usersSorterBean.sortsOrders['password']}"
                                sortType="custom">
                                <f:facet name="header">
                                        <a4j:commandLink execute="@this" value="Password" ren-
der="table"
                                                action="#{usersSorterBean.sort}">
                                                <f:param name="sortProperty" value="password"
/>
                                        </a4j:commandLink>
                                        <h:graphicImage style="position:absolute;"
name="down_icon.gif"
                                                library="images"
                                                ren-
dered="#{usersSorterBean.sortsOrders['password']=='descending'}" />
                                        <h:graphicImage style="position:absolute;"
name="up_icon.gif"
                                                library="images"
                                                ren-
dered="#{usersSorterBean.sortsOrders['password']=='ascending'}" />
                                        <br />
                                </f:facet>
                                <h:outputText value="#{user.password}" />
                        </rich:column>
                        <rich:column width="51px" sortBy="#{user.active}"
                                sortOrder="#{usersSorterBean.sortsOrders['active']}"
                                sortType="custom">
                                <f:facet name="header">
                                        <h:outputText value="Active" />
                                </f:facet>
                                <f:param name="sortProperty" value="active" />
                                <h:selectBooleanCheckbox value="#{user.active}" />
                        </rich:column>
                        <rich:column width="63px">
                                <f:facet name="header">
                                        <h:outputText value="Actions" />
                                </f:facet>

                                <h:graphicImage name="edit.png" library="images"
                                        title="#{msg['pagingGrid.tooltip.edit']}" />
                                <h:graphicImage name="delete.png" library="images"
                                        title="#{msg['pagingGrid.tooltip.delete']}" />

                        </rich:column>
                </rich:extendedDataTable>

                <a4j:commandButton execute="@this"
                        value="#{msg['pagingGrid.resetSorting']}"
                        action="#{usersSorterBean.reset}" render="table" />

        </h:form>
</ui:composition>
```

7.6.2. Managed Beans as a controllers

As it has been explained earlier, we recommend using Managed Beans as pure controllers. In this particular example, let us review `UserManagementBean` implementation. Basically, in this example its responsability is to delegate to the service layer (presentation service) in order to retrieve data according to the pagination parameters (number of rows to show in the grid, sorting parameters, etc) that the view (grid) handles and that are hold by the `UserSorterBean` Managed bean.

See the code below:

```
@ManagedBean
@ViewScoped
public class UserManagementBean implements Serializable {

	private static final long serialVersionUID = -6237417487105926855L;

	private Logger logger = LoggerFactory.getLogger(this.getClass());

	@ManagedProperty(value = "#{usersSorterBean}")
	private UsersSorterBean usersSorterBean;

	@ManagedProperty(value = "#{userServiceFacade}")
	private UserServiceFacade userServiceFacade;

	private List<UserVO> userVOList = null;

	public UserServiceFacade getUserServiceFacade() {
		return userServiceFacade;
	}

	public UsersSorterBean getUsersSorterBean() {
		return usersSorterBean;
	}

	public List<UserVO> getUserVOList() {

		logger.debug("getUserVOList");

		try {
			usersSorterBean.getSortPriorities();

			PresentationPaginatedDataFilter presentationPaginatedDataFilter = new
PresentationPaginatedDataFilter();
			for (String columnToSort : usersSorterBean.getSortPriorities()) {
				SortOrder sortOrder = usersSorterBean.getSortsOrders().get(
						columnToSort);
				presentationPaginatedDataFilter
						.addSortingColumn(
								columnToSort,
								sortOr-
der.equals(SortOrder.ascending) ? PresentationDataFilter.ASC
                                                                : Presenta-
tionDataFilter.DESC);
			}

			// Current function
			PresentationPaginatedResult<UserVO> paginatedResult = userServiceFacade
					.loadUsers(presentationPaginatedDataFilter);
			userVOList = paginatedResult.getData();
		} catch (Throwable e) {
			// TODO: Review real error handling this is a minimal exception han-
dling mechanism in JSF2
			FacesMessage fm = new FacesMessage(FacesMessage.SEVERITY_ERROR,
					"ERROR [" + e.getMessage() + "]", e.getMessage());
			FacesContext.getCurrentInstance().addMessage(null, fm);
		}
		return userVOList;
	}


	public void setUserServiceFacade(UserServiceFacade userServiceFacade) {
		this.userServiceFacade = userServiceFacade;
	}

	public void setUsersSorterBean(UsersSorterBean usersSorterBean) {
		this.usersSorterBean = usersSorterBean;
	}

	public void setUserVOList(List<UserVO> userVOList) {
		this.userVOList = userVOList;
	}
```

```
	@PostConstruct
	public void initIt() {
		logger.info("*** BEAN POST CONSTRUCTION: " + this.getClass().toString());
	}

	@PreDestroy
	public void cleanUp() {
		logger.info("*** BEAN POST DESTRUCTION: " + this.getClass().toString());
	}
}
```

7.6.3. Presentation service layer and model

The Presentation Service layer is the entry point to the service layer. It isolates the business layer from the presentation technology used for front end implementation. That is why we recommend you to keep this layer between you Managed Beans and your business layer and keep managed beans acting as a controllers and as simple as possible.

Let us see presentation service code interface and implementation.

7.6.3.1. UserServiceFacade Interface

All presentation services implement an interface so that the callers are not affected for changes in the implementation.

Note that in this example we are adding Spring security annotations in order to apply access security to methods in the interface. Adding the annotation in the interface method signature ensures that security will be always applied consistently regardless the implementation.

Please, see the code below:

```
package org.appverse.web.showcases.jsf2showcase.backend.services.presentation;

import java.util.List;

import
org.appverse.web.framework.backend.api.model.presentation.PresentationPaginatedDataFilter;
import org.appverse.web.framework.backend.api.model.presentation.PresentationPaginatedResult;
import org.appverse.web.framework.backend.api.services.presentation.IPresentationService;
import org.appverse.web.showcases.jsf2showcase.backend.model.presentation.UserVO;
import org.springframework.security.access.prepost.PreAuthorize;


public interface UserServiceFacade extends IPresentationService{

	@PreAuthorize("hasRole('ROLE_USER')")
	UserVO loadUser(long userId) throws Exception;

	@PreAuthorize("hasRole('ROLE_USER')")
	List<UserVO> loadUsers() throws Exception;

	@PreAuthorize("hasRole('ROLE_USER')")
	long saveUser(UserVO userVO) throws Exception;

	@PreAuthorize("hasRole('ROLE_USER')")
	PresentationPaginatedResult<UserVO> loadUsers(
			PresentationPaginatedDataFilter config) throws Exception;

	@PreAuthorize("hasRole('ROLE_USER')")
	void deleteUser(final UserVO userVO) throws Exception;
```

7.6.3.2. UserServiceFacade Implementation

As we would expect, all methods defined in the UserServiceFacade interface are implemented by UserServiceFacade implementation.

Note that this service delegates to the subsequent services layer (business) by means of the corresponding interface. As every service layer has its own model, conversion from one model to another is required before delegating to the business service. Take into account that in this case services and models from subsequent layers are quite similar because is an example but in a more complex scenario presentation and business services and their respective models would have different degrees of granularity. Having different object models in each layer ensures layers decoupling.

Please, see the code below:

```
package org.appverse.web.showcases.jsf2showcase.backend.services.presentation.impl.live;

import java.util.List;

import
org.appverse.web.framework.backend.api.converters.p2b.PaginatedDataFilterP2BBeanConverter;
import org.appverse.web.framework.backend.api.helpers.log.AutowiredLogger;
import org.appverse.web.framework.backend.api.model.business.BusinessPaginatedDataFilter;
import
org.appverse.web.framework.backend.api.model.presentation.PresentationPaginatedDataFilter;
import org.appverse.web.framework.backend.api.model.presentation.PresentationPaginatedResult;
import
org.appverse.web.framework.backend.api.services.presentation.AbstractPresentationService;
import org.appverse.web.showcases.jsf2showcase.backend.model.presentation.UserVO;
import org.appverse.web.showcases.jsf2showcase.backend.converters.p2b.UserP2BBeanConverter;
import org.appverse.web.showcases.jsf2showcase.backend.model.business.User;
import org.appverse.web.showcases.jsf2showcase.backend.services.business.UserService;
import org.appverse.web.showcases.jsf2showcase.backend.services.presentation.UserServiceFacade;
import org.slf4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("userServiceFacade")
public class UserServiceFacadeImpl extends AbstractPresentationService
                implements UserServiceFacade {

        @AutowiredLogger
        private static Logger logger;

        @Autowired
        private UserService userService;
        @Autowired
        private UserP2BBeanConverter userP2BBeanConverter;

        @Autowired
        private PaginatedDataFilterP2BBeanConverter paginatedDataFilterP2BBeanConverter;

        public UserServiceFacadeImpl() {
        }

        @Override
        public UserVO loadUser(final long userId) throws Exception {
                final User user = userService.loadUser(userId);
                return userP2BBeanConverter.convert(user);
        }

        @Override
        public List<UserVO> loadUsers() throws Exception {
                final List<User> users = userService.loadUsers();

                final List<UserVO> usersVO = userP2BBeanConverter
                                .convertBusinessList(users);
                return usersVO;
        }

        @Override
```

```
        public PresentationPaginatedResult<UserVO> loadUsers(
                        final PresentationPaginatedDataFilter config) throws Exception {

                final BusinessPaginatedDataFilter businessDatafilter = paginatedData-
FilterP2BBeanConverter
                                .convert(config);

                // Get the total number of rows first
                final int total = userService.countUsers(businessDatafilter);

                // Get the rows
                final List<User> users = userService.loadUsers(businessDatafilter);

                final List<UserVO> usersVO = userP2BBeanConverter
                                .convertBusinessList(users);

                return new PresentationPaginatedResult<UserVO>(usersVO, total,
                                config.getOffset());
        }

        @Override
        public long saveUser(final UserVO userVO) throws Exception {
                final User user = userP2BBeanConverter.convert(userVO);
                return userService.saveUser(user);
        }

        @Override
        public void deleteUser(final UserVO userVO) throws Exception {
                final User user = userP2BBeanConverter.convert(userVO);
                userService.deleteUser(user);
        }
}
```

7.6.3.3. Presentation model (VO bean)

Let us see the code corresponding to the presentation model used in this example (`UserVO.java`). All presentation Java Beans class names end up with "VO" as a convention and extend Appverse Web presentation bean hierarchy.

The most striking thing in `UserVO.java` is that this is the JSR-303 annotated Java Bean that allows client and server side consistent validation as we have explained earlier and, of course, follow the requeriments to be a Java Bean.

Please, see the code below:

```
package org.appverse.web.showcases.jsf2showcase.backend.model.presentation;

import java.util.Date;

import javax.validation.constraints.AssertTrue;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import
org.appverse.web.framework.backend.api.model.presentation.AbstractPresentationAuditedBean;

public class UserVO extends AbstractPresentationAuditedBean {

        private static final long serialVersionUID = 1L;

        private long id;

        @Size(min = 4, max = 10, message = "{validator.username.length}")
        @NotNull(message = "{validator.username.required}")
        private String name;

        @Size(min = 4, max = 10, message = "{validator.lastname.length}")
```

```
	@NotNull(message = "{validator.lastname.required}")
	private String lastName;

	@Size(min = 10, max = 20, message = "{validator.email.length}")
	@NotNull(message = "{validator.email.required}")
	private String email;

	@Size(min = 4, max = 40, message = "{validator.password.length}")
	@NotNull(message = "{validator.password.required}")
	private String password;

	private String signup;

	@NotNull
	private boolean active = true;

	@NotNull
	private Date initDate;

	@NotNull
	private Date endDate;

	@AssertTrue(message = "{validator.inidnidate.invalid}")
	public boolean isCheckDates() {
		if (endDate.before(initDate)) {
			return false;
		}
		return true;
	}

	public Date getEndDate() {
		return endDate;
	}

	public void setEndDate(Date endDate) {
		this.endDate = endDate;
	}

	public Date getInitDate() {
		return initDate;
	}

	public void setInitDate(Date initDate) {
		this.initDate = initDate;
	}

	public UserVO() {
		super();
	}

	public String getEmail() {
		return email;
	}

	public long getId() {
		return id;
	}

	public String getLastName() {
		return lastName;
	}

	public String getName() {
		return name;
	}

	public String getPassword() {
		return password;
	}

	public String getSignup() {
		return signup;
	}

	public boolean isActive() {
```

```
                return active;
        }

        public void setActive(final boolean active) {
                this.active = active;
        }

        public void setEmail(final String email) {
                this.email = email;
        }

        public void setId(final long id) {
                this.id = id;
        }

        public void setLastName(final String lastName) {
                this.lastName = lastName;
        }

        public void setName(final String name) {
                this.name = name;
        }

        public void setPassword(final String password) {
                this.password = password;
        }

        public void setSignup(final String signup) {
                this.signup = signup;
        }
}
```

7.6.4. Business Service layer and model

The Business Service layer provides business services that are independent of the presentation technology.

7.6.4.1. UserService Interface

All presentation services implement an interface so that the callers are not affected by changes in the implementation.

See the code below:

```
package org.appverse.web.showcases.jsf2showcase.backend.services.business;

import java.util.List;

import org.appverse.web.framework.backend.api.model.business.BusinessPaginatedDataFilter;
import org.appverse.web.showcases.jsf2showcase.backend.model.business.User;
import org.springframework.security.access.prepost.PreAuthorize;

public interface UserService {

        int countUsers(BusinessPaginatedDataFilter filter) throws Exception;


        User loadUser(long pk) throws Exception;


        List<User> loadUsers() throws Exception;


        List<User> loadUsers(BusinessPaginatedDataFilter config) throws Exception;


        long saveUser(User user) throws Exception;
```

```
        void deleteUser(final User user) throws Exception;
}
```

7.6.4.2. UserService Implementation

As we would expect, all methods defined in the UserService interface are implemented by UserService implementation.

Note that this service delegates to the subsequent services layer (integration) by means of the corresponding interface. As every service layer has its own model, conversion from one model to another is required before delegating to the integration service.

Please, see the code below:

```
package org.appverse.web.showcases.jsf2showcase.backend.services.business.impl.live;

import java.util.List;

import org.appverse.web.framework.backend.api.converters.ConversionType;
import
org.appverse.web.framework.backend.api.converters.b2i.PaginatedDataFilterB2IBeanConverter;
import org.appverse.web.framework.backend.api.model.business.BusinessPaginatedDataFilter;
import org.appverse.web.framework.backend.api.model.integration.IntegrationPaginatedDataFilter;
import org.appverse.web.framework.backend.api.services.business.AbstractBusinessService;
import org.appverse.web.showcases.jsf2showcase.backend.converters.b2i.UserB2IBeanConverter;
import org.appverse.web.showcases.jsf2showcase.backend.model.business.User;
import org.appverse.web.showcases.jsf2showcase.backend.model.integration.UserDTO;
import org.appverse.web.showcases.jsf2showcase.backend.services.business.UserService;
import org.appverse.web.showcases.jsf2showcase.backend.services.integration.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;


@Service("userService")
public class UserServiceImpl extends AbstractBusinessService implements
                UserService {

        @Autowired
        private UserRepository userRepository;

        @Autowired
        private UserB2IBeanConverter userB2IBeanConverter;

        @Autowired
        private PaginatedDataFilterB2IBeanConverter paginatedDataFilterB2IBeanConverter;

        @Override
        public int countUsers(final BusinessPaginatedDataFilter filter)
                        throws Exception {
                final IntegrationPaginatedDataFilter integrationDataFilter = paginatedData-
FilterB2IBeanConverter
                                .convert(filter);
                return userRepository.count(integrationDataFilter);
        }

        @Override
        public User loadUser(final long pk) throws Exception {
                // Note that default conversion type (not specified) is
                // ConversionType.Complete. This will convert all collections
                // using the proper corresponding mapping
                final UserDTO userDTO = userRepository.retrieve(pk);
                final User user = userB2IBeanConverter
                                .convert(userDTO);
                return user;
        }

        @Override
        public void deleteUser(final User user) throws Exception {
```

```
                final UserDTO userDTO = userRepository.retrieve(user.getId());
                userRepository.delete(userDTO);
        }

        @Override
        public List<User> loadUsers()
                        throws Exception {
                // Note that ConversitonType.WithoutDependencies will not convert
                // collections using the corresponding mapping
                List<UserDTO> userList = userRepository.retrieveList();
                return userB2IBeanConverter.convertIntegrationList(userList);
        }

        @Override
        public List<User> loadUsers(
                        final BusinessPaginatedDataFilter config) throws Exception {
                final IntegrationPaginatedDataFilter integrationDataFilter = paginatedData-
FilterB2IBeanConverter
                                .convert(config);

                final List<UserDTO> userList = userRepository
                                .retrieveList(integrationDataFilter);

                return userB2IBeanConverter.convertIntegrationList(userList);
        }

        @Override
        public long saveUser(
                        final User user)
                        throws Exception {
                UserDTO userDTO;

                if (user.getId() != 0L) {
                        // As it is an existing user we retrieve the entity manager managed
                        // object
                        userDTO = userRepository.retrieve(user.getId());
                        userB2IBeanConverter.convert(user, userDTO,
                                        ConversionType.WithoutDependencies);
                } else {
                        // We are creating a new DTO (not managed by the entity manager yet)
                        userDTO = userB2IBeanConverter.convert(user);
                }
                return userRepository.persist(userDTO);
        }
}
```

7.6.4.3. Business model

Let us see the code corresponding to the business model used in this example (`User.java`). Business Java Beans class names do not add any prefix (like "VO" and "DTO" for presentation and integration layer) and they extend Appverse Web business bean hierarchy. Of course, business model objects follow the requeriments to be a Java Bean.

Please, see the code below:

```
package org.appverse.web.showcases.jsf2showcase.backend.model.business;

import org.appverse.web.framework.backend.api.model.business.AbstractBusinessAuditedBean;

public class User extends AbstractBusinessAuditedBean {

        private static final long serialVersionUID = 92981844901681239L;

        private long id;
        private String name;
        private String lastName;
        private String email;
        private String password;

        private boolean active = true;
```

```
        public String getEmail() {
                return email;
        }

        public long getId() {
                return id;
        }

        public String getLastName() {
                return lastName;
        }

        public String getName() {
                return name;
        }

        public String getPassword() {
                return password;
        }

        public boolean isActive() {
                return active;
        }

        public void setActive(final boolean active) {
                this.active = active;
        }

        public void setEmail(final String email) {
                this.email = email;
        }

        public void setId(final long id) {
                this.id = id;
        }

        public void setLastName(final String lastName) {
                this.lastName = lastName;
        }

        public void setName(final String name) {
                this.name = name;
        }

        public void setPassword(final String password) {
                this.password = password;
        }
}
```

7.6.5. Integration service layer and model

The Integration Service layer is the entry point to the services layer. It is in charge for data processing: retrieval, storage or exchange with database and systems. Take into account that in the integration services layer you could have integration with different sorts of databases, JMS queues and other systems by means web services, for instance.

Let us see presentation service code interface and implementation.

7.6.5.1. UserRepository Interface

All integration services implement an interface so that the callers are not affected by changes in the implementation. Interfaces hide the underlying integration technology (Database – JPA, Web Services, JMS, etc) in this case.

Let us see the source code:

```
Import
org.appverse.web.framework.backend.persistence.services.integration.IJPAPersistenceService;
import org.appverse.web.showcases.jsf2showcase.backend.model.integration.UserDTO;
import org.springframework.security.access.prepost.PreAuthorize;

public interface UserRepository extends IJPAPersistenceService<UserDTO> {

        @PreAuthorize("hasRole('ROLE_USER')")
        UserDTO loadUserByUsername(final String username) throws Exception;
}
```

7.6.5.2. UserRepository Implementation

As we would expect, all methods defined in the `UserService` interface are implemented by `UserService` implementation. All integration JPA services extend `JPAPersistenService` which allows basic relational database operations and support for basic filtering and remote sorting and pagination. However, we recommend you to restrict its usage to these scenarios as in any case it was developed thinking on it as a replacement for custom queries.

```
package org.appverse.web.showcases.jsf2showcase.backend.services.integration.impl.live;

import java.util.List;

import org.appverse.web.framework.backend.api.helpers.log.AutowiredLogger;
import
org.appverse.web.framework.backend.persistence.services.integration.helpers.QueryJpaCallback;
import
org.appverse.web.framework.backend.persistence.services.integration.impl.live.JPAPersistenceSer
vice;
import org.appverse.web.showcases.jsf2showcase.backend.model.integration.UserDTO;
import org.appverse.web.showcases.jsf2showcase.backend.services.integration.UserRepository;
import org.slf4j.Logger;
import org.springframework.stereotype.Repository;

@Repository("userRepository")
public class UserRepositoryImpl extends JPAPersistenceService<UserDTO>
                implements UserRepository {

        @AutowiredLogger
        private static Logger logger;

        @Override
        public UserDTO loadUserByUsername(final String username) throws Exception {
                final StringBuilder queryString = new StringBuilder();
                queryString.append("select user from UserDTO user where user.email='")
                                .append(username).append("'");
                final QueryJpaCallback<UserDTO> query = new QueryJpaCallback<UserDTO>(
                                queryString.toString(), false);
                List<UserDTO> list = retrieveList(query);

                if (list != null && list.size() > 0) {
                        return list.get(0);
                } else
                        return null;
        }
}
```

In this example, the service handles a relational database through JPA. You can see the implementation of the `loadUserByUsername` method that retrieves user information by means a custom query. Please note that in this example you do not find the `retrieveList` method that retrieves the list of users. The reason is that `JPAPersistenceService` automatically provides the implementation for the method with the signature:

```
public List<User> loadUsers(final BusinessPaginatedDataFilter config) throws
Exception;
```

This is a plain method that provides the full list of users that we use to show in the grid providing support for remote sorting and pagination by means a `BusinessPaginatedDataFilter`.

7.6.5.3. JPA annotated integration model

As we have commented earlier, the integration model objects in this example are JPA annotated beans. You can see the code below:

```
package org.appverse.web.showcases.jsf2showcase.backend.model.integration;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.TableGenerator;
import javax.persistence.Version;

import
org.appverse.web.framework.backend.persistence.model.integration.AbstractIntegrationAuditedJPAB
ean;

@Entity
@Table(name = "USER")
public class UserDTO extends AbstractIntegrationAuditedJPABean implements
                Serializable {

        private static final long serialVersionUID = 1L;

        private String name;
        private String lastName;
        private String email;
        private String password;

        private boolean active = true;


        public UserDTO() {
        }

        @Column(nullable = false, unique = true, length = 40)
        public String getEmail() {
                return email;
        }

        @Id
        @TableGenerator(name = "USER_GEN", table = "SEQUENCE", pkColumnName = "SEQ_NAME",
pkColumnValue = "USER_SEQ", valueColumnName = "SEQ_COUNT", initialValue = 1, allocationSize =
1)
        @GeneratedValue(strategy = GenerationType.TABLE, generator = "USER_GEN")
        public long getId() {
                return id;
        }

        @Column(nullable = false, length = 40)
        public String getLastName() {
                return lastName;
        }

        @Column(nullable = false, length = 40)
        public String getName() {
                return name;
        }

        @Column(nullable = false, length = 40)
        public String getPassword() {
                return password;
        }
```

```
        @Override
        @Version
        public long getVersion() {
                return version;
        }

        @Column(nullable = false)
        public boolean isActive() {
                return active;
        }

        public void setActive(final boolean active) {
                this.active = active;
        }

        public void setEmail(final String email) {
                this.email = email;
        }

        public void setLastName(final String lastName) {
                this.lastName = lastName;
        }

        public void setName(final String name) {
                this.name = name;
        }

        public void setPassword(final String password) {
                this.password = password;
        }
}
```