

Web Anwendungen Entwickeln mit – JSF, Spring und Tomcat –

Rainer Schuler

Ulm, AOI-Systeme
Schulung, Beratung, Entwicklung

Januar 2011

Übersicht

1 System Architecture

Übersicht

1 System Architecture

2 User Interface

Übersicht

- 1 System Architecture
- 2 User Interface
- 3 Application and Domain Layer

Übersicht

- 1 System Architecture
- 2 User Interface
- 3 Application and Domain Layer
- 4 Infrastructure

Übersicht

- 1 System Architecture
 - Domain Driven Design
 - Two Tier Architecture
 - Inversion of Control
- 2 User Interface
- 3 Application and Domain Layer
- 4 Infrastructure

Domain Driven Design

Necessary aspects of the domain are described by a model which can be used to solve the tasks (business logic, services) related to the domain.

Building Blocks

- Entity, an object with an identity (e.g. seat in a concert hall)
- Value Object, object with attributes (e.g. dollar bill)
- Aggregate, aggregate root (e.g. house)
- Service, operations not belonging to an object
- Repository, specialized repository objects retrieve the domain objects
- Factory, factory objects create domain objects

Application Architecture

A typical enterprise application architecture consists of the following four conceptual layers:

- ① User Interface (Presentation Layer): Responsible for presenting information to the user and interpreting user commands.
- ② Application Layer: Coordinates the application activity. It holds the state (progress) of the application tasks.
- ③ Domain Layer: This layer contains information about the business domain. The business objects are held here.
- ④ Infrastructure Layer: This layer acts as a supporting library for all the other layers. It provides communication between layers.

Application and Domain

Application Layer

- It holds the state of the services (application tasks).
- It does not hold the state of business objects
- Business logic is delegated to the domain layer.

Domain Layer

- Implements business logic.
- The save, retrieve or modify operations of the business objects are delegated to the infrastructure layer.

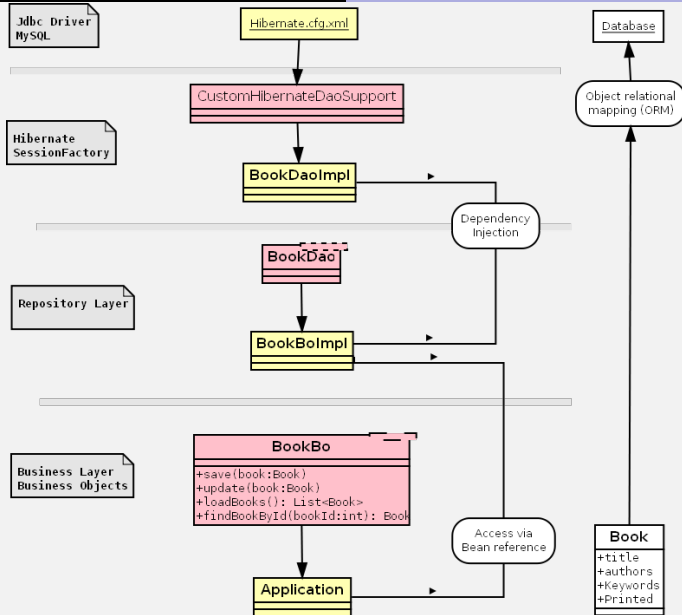
User Interface and Infrastructure

User Interface

- Displays and updates (domain) Value Objects.
- User requests are delegated to the application layer.

Infrastructure

- The repository layer retrieves business objects from the entities (DAOs) stored in a database.
- It does the object relational mapping of the DAOs to tables (of a relational database).
- It stores tables in a (relational) database (MySQL, HSQLDB).



Two Tier Architecture

Container

A container is a selfcontained piece of software which is run independently of the application. We will be (explicitly) using two container in our application.

- ❶ Tomcat, running the application (using JSF/Facelets).
 - ❷ Spring, running the database (using Hibernate) and access control (using spring-security).
- Resources are registered/injected using descriptions in xml-files and/or by annotations in the Java-classes.
 - The application can request/supply resources held by the container.

Tomcat container

A page request (or PPR) is delegated to a servlet deployed with the application. The servlet has access to the resources of the container (external context).

External context

A Java class (Bean) exposes its constructor (methods, fields) to the container. On request an instance of the class is created, requested resources are injected, and the class is put into the context.

- application-context: only one instance.
- session-context: one instance for each (user-) session.
- request scoped: one (stateless) instance for each request.

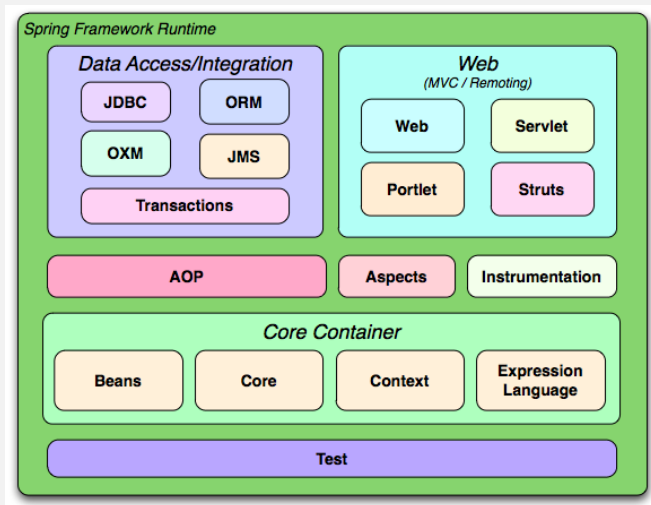
A Bean is a Java-class following certain conventions about method naming, construction, and behavior, i.e., it is serializable, it has a default constructor, and its properties have getters and setters.

Spring container

Resources (Beans) are created and held in an application context (as singletons or prototype).

Context

- Spring context can be described in the deployment descriptor (deployed with the application) or as `XmlApplicationContext` (build during run-time).
- Spring is able to auto scan, detect and instantiate (application) beans or components.
- Resources (Beans, Java-classes) are accessible through Dependency Injection.
- The application (Java-class, Bean) can access resources via Bean reference to pre-defined (elsewhere) instances.



Spring container

We will be using two spring modules.

Spring Core/ORM

Spring allows you to define and inject resources like a JDBC Data-Source, a Hibernate Session-Factory, data access objects (DAOs), and bussines objects.

Spring Security

Page request are passed through a security filter (web.xml).

- Non authorized page request are redirected to a login page.
- Remember-me button allows long-time authentication.
- Allows password encryption and database (MySQL) usage.

Inversion of Control

Replace instantiation with dependency injection!

- Instantiation

```
public StockBoImpl() {
    this.stockDao = new StockDaoImpl();
}
```

- Injection (using the interface StockDao!)

```
@Autowired
public StockBoImpl(StockDao stockDao) {
    this.stockDao = stockDao;
}
```

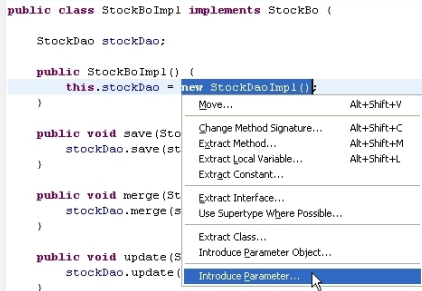
```
public class StockBoImpl implements StockBo {
    StockDao stockDao;

    public StockBoImpl() {
        this.stockDao = new StockDaoImpl();
    }

    public void save(Stock stock) {
        stockDao.save(stock);
    }

    public void merge(Stock stock) {
        stockDao.merge(stock);
    }

    public void update(Stock stock) {
        stockDao.update(stock);
    }
}
```



```
@Service("stockBo")
public class StockBoImpl implements StockBo {
    StockDao stockDao;

    @Autowired
    public StockBoImpl(StockDao stockDao) {
        this.stockDao = stockDao;
    }

    public void save(Stock stock) {
        stockDao.save(stock);
    }

    public void merge(Stock stock) {
        stockDao.merge(stock);
    }
}
```

Init application context and use bean reference!

- Init context (spring) in a managed bean (tomcat)

```
@ManagedBean(name = "applicationService")
@ApplicationScoped
public class AppBookcase implements Serializable {
    ApplicationContext appContext;
    public AppBookcase() {
        appContext = new ClassPathXmlApplicationContext(
            "/config/BeanLocations.xml");
    }
    .....
    public ApplicationContext getAppContext() {
        return appContext;
    }
}
```

- Use bean reference

```
@ManagedBean
@SessionScoped
public class ViewBean {
    @ManagedProperty(value = "#{applicationService}")
    private AppBookcase application;

    public void setApplication(AppBookcase application) {
        this.application = application;
    }
    .....
    StockBo stockBo =
        application.getAppContext().getBean(StockBO);
    .....
}
}
```

Übersicht

- 1 System Architecture
- 2 User Interface
 - JSF
 - Facelets
 - Backing Beans
- 3 Application and Domain Layer
- 4 Infrastructure

User Interface

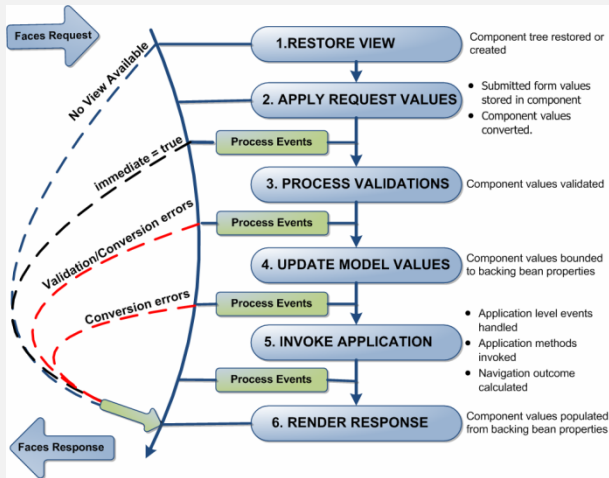
We use JSF to implement the Web-pages of the User Interface.

Web-page description

- ❶ Describe the elements of the page, i.e. what is to be displayed.
- ❷ Describe the view of each element, i.e. how it is displayed (rendered).
- ❸ Describe the value of the fields and the actions.
 - Elements are JSF-tags defined in the tag libraries.
 - The view of each tag is defined in the implementation of the tag library and can be customized using skins and/or css-styles.
 - The value of the elements is defined in Backing-Beans (Java-classes).

The same page can be used in different places, e.g. a page to request address details can be used in several places.

Java Server Faces



JSF Lifecycle

The JSF lifecycle does (the plumbing) data transfer between web-pages and backing beans invokes actions (flow of control) and listeners (events).

Extending JSF

The standard JSF component libraries (core and html, prefixes f: and h: resp.) are part of the standard JSF implementations. Further component libraries and implementations can be added (myfaces tomahawk, trinidad, primefaces, prefixes t: tr: and p:).

Configuration

- JSF components can be configured (attributes, skins/themes, css-styles).
- JSF components use partial page requests (PPR) and have attributes to call custom Javascript or AJAX functions.

See for example the primefaces showcase at
<http://www.primefaces.org/showcase/ui/home.jsf>

JSF components

It is possible to develop custom components.

Moving parts, artefacts

- UIComponent Class, the computational aspect implementing the logic of the component.
- Render Class, the view aspect (rendering), which generates HTML to be displayed in the browser.
- UI Component Tag Class, the tag handler class
- Tag Library Descriptor File, associates tag handler with usable tag

It is possible to build JSF components using widgets from Web 2.0 libraries such as Dojo. See

<http://www.ibm.com/developerworks/web/library/wa-aj-jsfdojo/index.html>

Example JSF page

```
<h:form id="stockform">
  <p:fieldset legend="Available Stocks" appendToBody="false">
    <p:dataTable id="stocktable" var="stock" value="#{viewBeanStockEdit.stocks}"
      styleClass="topDataTable" >
      <p:column filterBy="#{stock.stockCode}">
        <f:facet name="header">
          <h:outputText value="Code" />
        </f:facet>
        <h:outputText value="#{stock.stockCode}" />
      </p:column>
      <p:column>
        <f:facet name="header">
          <h:outputText value="Name" />
        </f:facet>
        <h:outputText value="#{stock.stockName}" />
      </p:column>
      <p:column style="width:32px">
        <p:commandButton oncomplete="dialog.show()"
          image="ui-icon ui-icon-search" update="stockform:display">
          <f:setPropertyActionListener value="#{stock}"
            target="#{viewBeanStockEdit.editStockListener}" />
        </p:commandButton>
      </p:column>
    </p:fieldset>
    <p:commandButton value="AddRow"
      actionListener="#{viewBeanStockEdit.addStock}" update="display"
      oncomplete="dialog.show()" >
  </p:commandButton>
</h:form>
```



```
<p:dialog header="Stock Details" widgetVar="dialog" showEffect="fold"
  hideEffect="fold" modal="false" >
  <h:panelGrid columns="2" cellpadding="5" id="display">
    <h:panelGroup>
      <h:outputText value="Stock Code: " />
      <h:message for="inputTextStockName"></h:message>
    </h:panelGroup>
    <h:inputText value="#{viewBeanStockEdit.selectedStock.stockCode}" id="inputTextStockName"
      validator="#{viewBeanStockEdit.validateStockCode}"
      required="#{param['requireStockValidation']=='1'}" requiredMessage="value required" />

    <h:panelGroup >
      <h:outputText value="Stock Name: " />
      <h:message for="inputTextStockCode" styleClass="color:red;"></h:message>
    </h:panelGroup>
    <h:inputText value="#{viewBeanStockEdit.selectedStock.stockName}" id="inputTextStockCode"
      validator="#{viewBeanStockEdit.validateStockName}"
      required="#{param['requireStockValidation']=='1'}" requiredMessage="value required" />

    <p:commandButton value="Save"
      actionListener="#{viewBeanStockEdit.saveStock}" update="display stocktable">
      <f:param name="requireStockValidation" value="1"></f:param>
    </p:commandButton>
    <p:commandButton value="Delete" actionListener="#{viewBeanStockEdit.deleteStock}"
      update="display stocktable" disabled="#{!viewBeanStockEdit.editStock}">
    </p:commandButton>
  </h:panelGrid>
</p:dialog>
</h:form>
```

Facelets

Facelets

Facelets is the view declaration language (view-handler) for Java server faces (Java EE).

- Facelets converts HTML elements (with jsfc attribute) into the corresponding JSF components.
- Facelets builds the JSF component tree as defined by the view of a JSF application.
- Facelets supports the JSF UI components (templates).

Templating with UIComponents

A Web-page partitions the screen in logical regions (navigation, content, header, footer). A template defines the regions and includes contents from other files.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
  <title><ui:insert name="title">Default title</ui:insert></title>
  <link rel="stylesheet" type="text/css" href="#{request.contextPath}/css/pageLayout.css"/>
</head>
<body>
<div id="header">
  <ui:insert name="header">
    <ui:param name="headerBean" value="#{applicationBean.header}" />
    <ui:include src="/WEB-INF/template/header.xhtml"/>
  </ui:insert>
</div>
.....
<div id="content" >
  <ui:insert name="content">
    Content defined in the templating file comes here!
  </ui:insert>
</div>
.....
```

Templating with UIComponents

Files included by a template use the `ui:component` tag. Everything outside will be ignored. Parameters can be passed in from the template.

```
<ui:component
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:p="http://primefaces.prime.com.tr/ui"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ui="http://java.sun.com/jsf/facelets">
  <h:panelGroup id="infoPanel" layout="block" style="clear: both; padding: 20px;">
    <p:menubar>
      <p:submenu label="New" url="#{viewBean.newURL}"/>
      <p:submenu label="Edit" url="#{viewBean.editURL}"/>
      <p:submenu label="Delete" url="#{viewBean.deleteURL}"/>
      <p:submenu label="Properties" url="#{viewBean.propertiesURL}"/>
    </p:menubar>
  </h:panelGroup>
</ui:component>
```

Templating with UIComponents

A templating page defines the content of (named) regions and passes parameters to the template. Everything outside `ui:composition` tag will be ignored.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <ui:composition template="/WEB-INF/path/to/templateApp.xhtml">
    <ui:param name="applicationBean" value="#{BMCAApplicationBean}" />
    <ui:define name="title">BMC Distribution</ui:define>
    <ui:define name="content">
      <!-- use the form tag and define your page-->
      <h:form id="contentForm">
        This will be visible!
      </h:form>
    </ui:define>
  </ui:composition>
</html>
```

Backing Beans

The dynamic content of the web-page (JSF-components) is taken from so called Backing Beans (other content is taken from resource files). For a Web-page, e.g. bookSearch.xhtml we use two Java-classes.

ViewBean and Controller classes

- BookSearchViewBean.java to access data values (domain).
- BookSearchController.java to access values concerning the application process status.

In general, a viewBean can be seen as interface to information held in the business objects (domain layer) whereas the Controller(-Bean) is the interface to the process application status (application layer).

Backing Beans

In a typical application

- Backing-Beans are stateful, i.e. defined in session scope.
- Page request (navigation) are handled in the Controller Bean.
- Process validation (e.g. data consistency on page entry or page leave) are done in the Controller Bean.

Übersicht

- 1 System Architecture
- 2 User Interface
- 3 Application and Domain Layer
- 4 Infrastructure

Application Layer

The Application (process) as hierarchical structure

- The application consists of processes.
 - Each process consists of process steps.
 - Each process step corresponds to a Web-page (Dialog, Portlet)
-
- Processes are independent from each other, i.e., it is possible to navigate from one process to another.
 - Process-steps are sequential, i.e. it is not possible to leave or enter a step in a non-consistent state.

A process tries to model a (business) service. A process step is a means to display or gather information to and from the user.

Domain

The Domain Layer implements a model of the domain and services to solve the tasks. Usually we distinguish

- The Data model (database) which represents the domain knowledge.
- The Services which implement the manipulation of the data.

Services are called from the application layer. Data is displayed (and modified) in the User Interface. Business Objects implemented in the infrastructure are used to persist data.

Decouple Layers

Dependency injection and bean reference allows to decouple the layers.

Übersicht

- 1 System Architecture
- 2 User Interface
- 3 Application and Domain Layer
- 4 Infrastructure

Infrastructure

Repository Layer

- Implements the API, i.e. implements the Business Objects (BOs) which persist the domain objects (of the domain layer).
- The implementation of the Business Objects aggregates the domain objects from (simpler) Data Access Objects (DAOs).

The DAOs are mapped to database tables which in turn are stored in a database. Spring allows us to configure the mapping and the database.

- Mapping is done using Hibernate.
- Tables are stored in HSQLDB and MySQL databases.

Web-services, access to e.g. the enterprise service bus and interfaces to other (legacy) systems are also implemented in the infrastructure layer.

References

- The Java EE 6 Tutorial
<http://download.oracle.com/javaee/6/tutorial/doc/>
- The Hibernate reference documentation
<http://www.hibernate.org/docs>
- Spring Framework
<http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html>
- Spring Security
<http://static.springsource.org/spring-security/site/reference.html>
- An Introduction to Domain Driven Design, D. Haywood
<http://www.methodsandtools.com/archive/archive.php?id=97>