Barebones Spring MVC

James Douglas
www.earldouglas.com

October 17, 2010

Contents

1 About This Book

This is a book about Java Web application development using the Spring MVC framework. It distills much of what I have learned from developing enterprise applications with Spring MVC, guiding usage of the components of Spring MVC that I most frequently encounter in practice and on discussion forums. This book provides a brief overview of these components by taking you through the development of an example Spring MVC application from scratch.

My goals in writing this book are to guide developers who are unfamiliar with Spring MVC, and to supply a convenient reference to more seasoned developers.

1.1 Intended Audience

This book is written for developers who are interested in working with Spring MVC, whether they are newcomers or have been working with it for years. It assumes basic familiarity with Spring and Java Web applications, though I have included links to reference information to help fill these gaps.

I generally don't build a Spring MVC application from scratch, opting instead to build upon a framework I have already prepared, or a sample I have already developed. Since understanding the fundamental concepts is in no way the same as generating boilerplate code from memory, I find it valuable to maintain a barebones framework from which to springboard new project development.

1.2 Topics Covered

The following components of Spring MVC are covered:

- Core Spring MVC
- Server-side validation
- Rich client-side validation
- Security
- Database integration
- RESTful Web services
- Externalization and Internationalization

This book does not cover advanced, uncommon, or deep-dive topics into Spring MVC, Spring Web Flow, Spring JavaScript, JavaScript frameworks, etc., although there is some basic usage of both Spring JavaScript and the Dojo Toolkit's UI library Dijit. Instead, this book focuses on the development of an example application which features the common Spring MVC components.

2 Example Application

This book is based on a minimal example application which serves two goals: to provide a useful foundation in something not unlike a real-world application, and to avoid diving into a specific problem domain which eclipses the concepts as they are presented.

The example application is a limited employee directory. A user interacts with a Web front-end to view a list of employees, add new employees, and edit or delete existing employees.

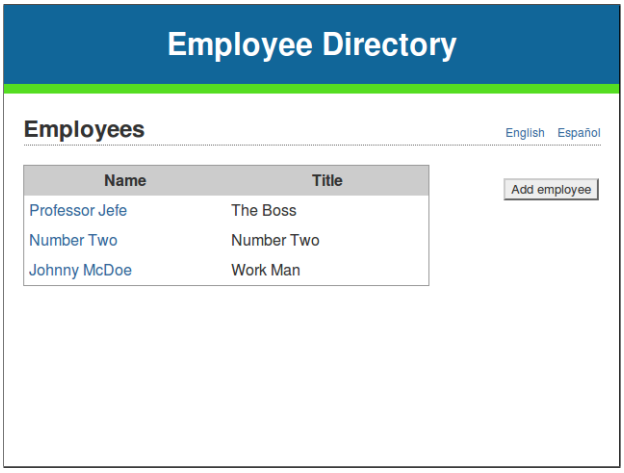

Figure 1: The *employees* view

Figure 1 shows the main view of the Employee Directory. It lists all employees in the directory, provides links to view and edit the details of each, and includes a button for adding a new employee to the directory.

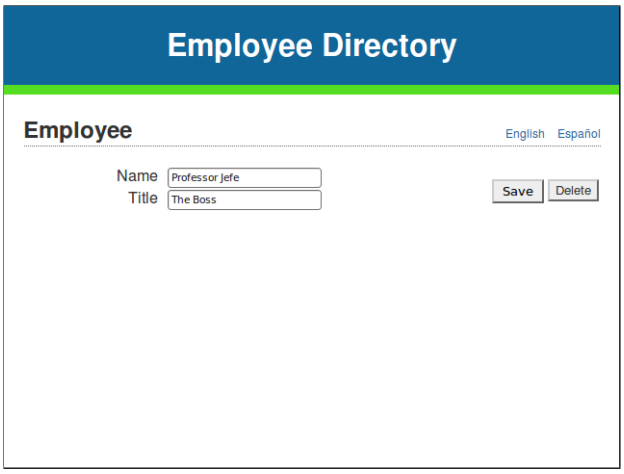

Figure 2: The *employee* view

Figure 2 shows a detailed view of a single employee. When the details of an employee are viewed, or when a new employee is to be added, this view is presented.

2.1 Application Skeleton

The Employee Directory is structured as a Maven project. At the root of the project is the POM, where dependencies and other project configuration is maintained. Application source code is placed in `src/main/java`, with corresponding configuration in `src/main/resources`. Web application content and configuration is within `src/main/webapp`. This includes the `web.xml` deployment descriptor, the JSP view templates, and other Web content (stylesheets, images, etc.).

2.2 Source Code

This book includes snippets of source code for the Employee Directory. The full source code is available online, from http://www.earldouglas.com/barebones-spring-mvc. I recommend keeping it handy for reference and experimentation as you progress through the content.

Source code included in this book is formatted as follows.

Listing 1: CodeConventions.java

```
package com.earldouglas.barebones.springmvc;

public class CodeConventions {

    public static void main(String[] arguments) {
        System.out.println("Hello World!");
    }
}
```

2.3 Section References

- The Maven POM is covered in detail in the Apache Maven Project Reference.
- The `web.xml` deployment descriptor, and the WAR file format in general, are covered in Wikipedia.

3 Core Application

3.1 Components

The core of the Employee Directory is the largest single segment of its construction. It consists of an in-memory repository of employees, with an HTML front-end for user interaction, using the following components:

3.1.1 Classes

- `Employee`: a domain class representing an employee
- `EmployeeService`: a simple CRUD-like interface for fetching, saving, and deleting Employees
- `InMemoryEmployeeService`: an EmployeeService implementation which contains an in-memory collection of Employees
- `BindableEmployee`: a flat class designed to bind to HTML forms
- `EmployeeController`: a Spring MVC controller to interact with the user

3.1.2 Views

- `employee.jsp`: a template for an HTML form for creating or updating an employee
- `employees.jsp`: a template for a list of employees in the Employee Directory

3.1.3 Configuration

- `style.css`: template CSS configuration for the views
- `pom.xml`: project dependencies and management
- `web.xml`: the J2EE Web deployment descriptor containing the Spring DispatcherServlet, the Front Controller for a Spring MVC application
- `spring-mvc-servlet.xml`: the Spring configuration for the various Spring beans and Spring infrastructure

3.2 Classes

At the architectural bottom of the code is the domain class `Employee`.

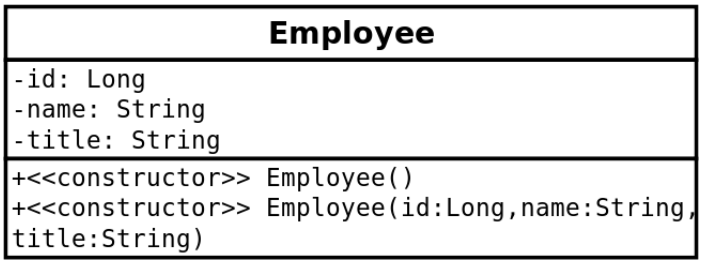

Figure 3: The `Employee` class

Listing 2: Employee.java

```java
public class Employee {

    private Long id;
    private String name;
    private String title;

    public Edmployee() {
    }

    public Employee(Long id, String name, String title) {
        this.id = id;
        this.name = name;
        this.title = title;
    }

    // Getters and setters omitted for brevity.
}
```

A service interface is defined by `EmployeeService`, which provides the standard CRUD behavior.

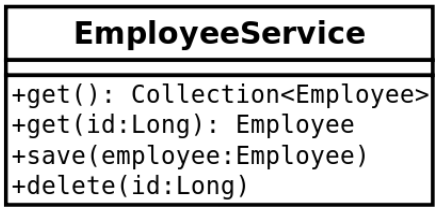

Figure 4: The `EmployeeService` interface

Listing 3: EmployeeService.java

```java
public interface EmployeeService {

    public Collection<Employee> get();

    public Employee get(Long id);
```

```
    public Employee save(Employee employee);

    public void delete(Long id);
}
```

A simple in-memory `EmployeeService` is implemented by `InMemoryEmployeeService`.

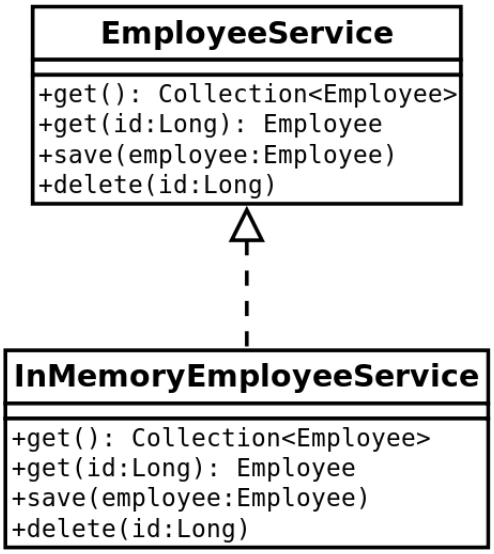

Figure 5: The `EmployeeService` interface and `InMemoryEmployeeService` class

Listing 4: InMemoryEmployeeService.java

```
@Repository
public class InMemoryEmployeeService implements EmployeeService {

    private long maxId = 3L;

    private final Map<Long, Employee> employees = new HashMap<Long, Employee>() {
        private static final long serialVersionUID = 1L;
        {
            put(1L, new Employee(1L, "Professor Jefe", "The Boss"));
            put(2L, new Employee(2L, "Number Two", "Number Two"));
            put(3L, new Employee(3L, "Johnny McDoe", "Work Man"));
        }
    };

    @Override
    public Employee get(Long id) {
        return employees.get(id);
    }

    @Override
    public Collection<Employee> get() {
        return employees.values();
    }

    private synchronized long nextId() {
```

```
        return ++maxId;
    }

    @Override
    public Employee save(Employee employee) {
        if (employee.getId() == null) {
            employee.setId(nextId());
        }
        employees.put(employee.getId(), employee);
        return employee;
    }

    @Override
    public void delete(Long employeeId) {
        employees.remove(employeeId);
    }
}
```

This service implementation will provide core services to the Web front-end, implemented as `EmployeeController`. The `EmployeeController` is responsible for handling HTTP requests, and translating between the UI model and the domain model classes.

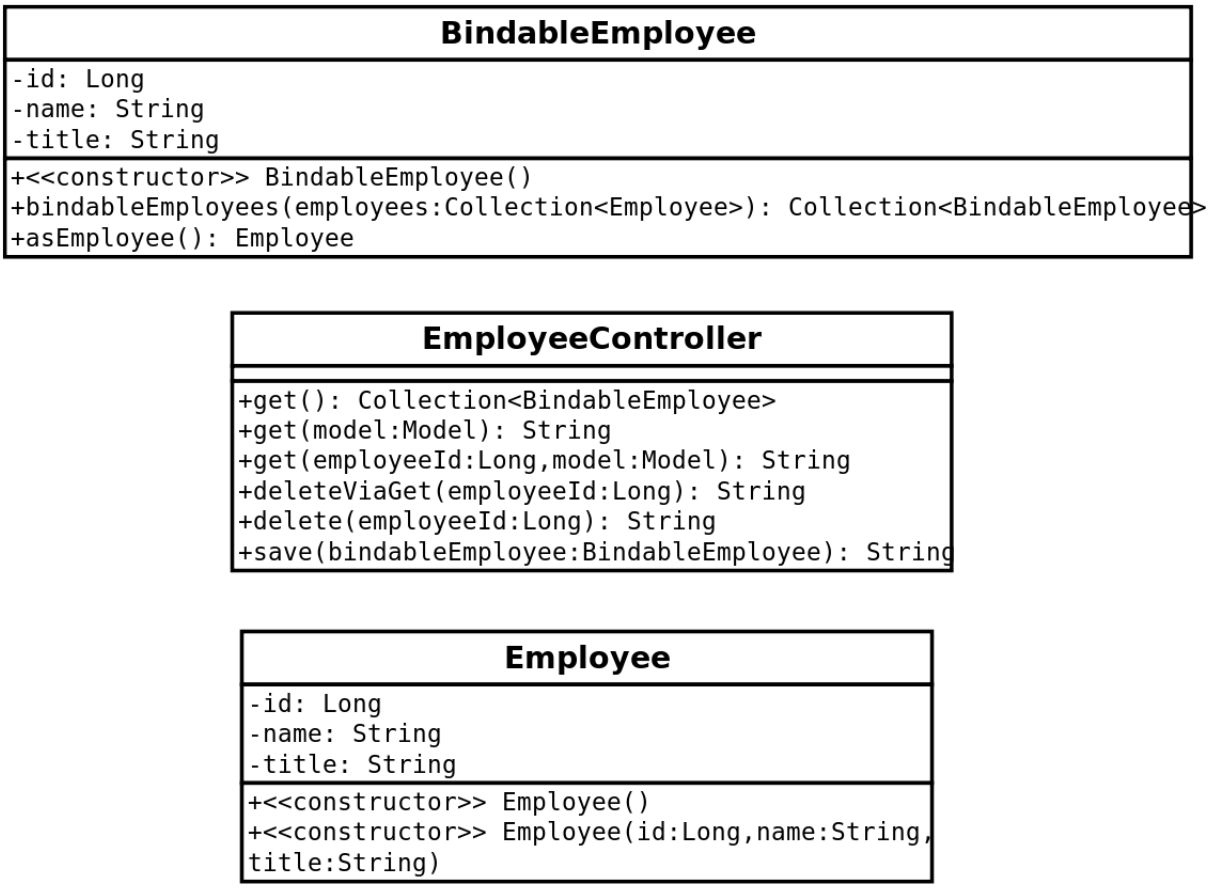

Figure 6: The `BindableEmployee`, `EmployeeController`, and `Employee` classes

Listing 5: EmployeeController.java

```
@Controller
@RequestMapping("/employees")
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @RequestMapping(method = RequestMethod.GET)
    public Collection<BindableEmployee> get() {
        return BindableEmployee.bindableEmployees(employeeService.get());
    }

    @RequestMapping(value = "/new", method = RequestMethod.GET)
    public String get(Model model) {
        return get(null, model);
    }

    @RequestMapping(value = "/{employeeId}", method = RequestMethod.GET)
    public String get(@PathVariable Long employeeId, Model model) {
        Employee employee = employeeService.get(employeeId);
        if (employee != null) {
            model.addAttribute(new BindableEmployee(employee));
        } else {
            model.addAttribute(new BindableEmployee());
        }
        return "employee";
    }

    @RequestMapping(value = "/{employeeId}/delete", method = RequestMethod.GET)
    public String deleteViaGet(@PathVariable Long employeeId) {
        return delete(employeeId);
    }

    @RequestMapping(value = "/{employeeId}", method = RequestMethod.DELETE)
    public String delete(@PathVariable Long employeeId) {
        employeeService.delete(employeeId);
        return "redirect:../../employees";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String save(BindableEmployee bindableEmployee) {
        employeeService.save(bindableEmployee.asEmployee());
        return "redirect:employees";
    }
}
```

There are several things going on in `EmployeeController`. The class is annotated with `@Controller` to indicate to Spring its function as an MVC controller and its candidacy for component scanning by the Spring container. In addition, it is annotated with a class-level `@RequestMapping` to base all of its method-level `@RequestMapping`s on a top-level URL pattern. Each method is also annotated with `@RequestMapping` to further constrain their specific associated request patterns.

The `get(Long, Model)`, `deleteViaGet(Long)`, and `delete(Long)` methods are each configured to map to RESTful URLs which contain the identifier of the `Employee` object on which to operate.

The `save(BindableEmployee` method contains no URL information in its `@RequestMapping`, so it will map simply to `/employees`, that of the class-level annotation. This is in contrast to the other methods, such as `get(Model)`, which specifies `/new`. This combines with the class-level annotation to map to `/employees/new`.

All of the methods specify a HTTP request method in the RESTful style.

`EmployeeController` presents `Employee`-like data to the user both as textual data and as an HTML form. This is cause for a special class to be designed with the Web UI in mind, specifically to bind to the HTML form. This role is filled by `BindableEmployee`.

Listing 6: BindableEmployee.java

```
public class BindableEmployee {

    private Long id;

    private String name;

    private String title;

    public BindableEmployee() {
    }

    public BindableEmployee(Employee employee) {
        this.id = employee.getId();
        this.name = employee.getName();
        this.title = employee.getTitle();
    }

    // Getters and setters omitted for brevity.

    public static Collection<BindableEmployee> bindableEmployees(
            Collection<Employee> employees) {
        Collection<BindableEmployee> bindableEmployees = new ArrayList<BindableEmployee>();
        for (Employee employee : employees) {
            bindableEmployees.add(new BindableEmployee(employee));
        }
        return bindableEmployees;
    }

    public Employee asEmployee() {
        return new Employee(id, name, title);
    }

}
```

`BindableEmployee` knows both how to convert itself into the domain class `Employee` via its `asEmployee()` method and how to convert a collection of instances of `Employee` into a collection of instances of `BindableEmployee`. This is a convenient location for this functionality, and an important one as well. Because this conversion is only concerned with connecting the domain to a thin Web layer, the appropriate location for related computation is in the Web layer and out of the domain.

That's all the Java code there is to write. Simple!

3.3 View Templates

Next, the view templates are defined.

`employee.jsp` uses Spring's `form` tag library to build a form with text inputs for the name and title of an employee.

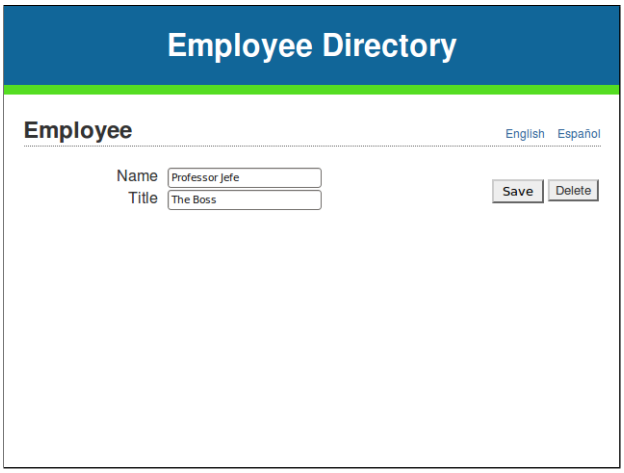

Figure 7: The `employee` view

Listing 7: employee.jsp

```
<c:url var="formUrl" value="/employees" />
<form:form action="${formUrl}" modelAttribute="bindableEmployee">
    <ul class="button">
        <c:if test="${not empty bindableEmployee.id}">
            <li><a href="<c:url value="/employees/${bindableEmployee.id}/delete" />">
                Delete
            </a></li>
        </c:if>
        <li><input id="submit" type="submit" value="Save" /></li>
    </ul>
    <form:hidden path="id" />
    <ul>
        <li>
            <label for="name">Name</label>
            <form:input path="name" />
        </li>
        <li>
            <label for="title">Title</label>
            <form:input path="title" />
        </li>
    </ul>
</form:form>
```

`employees.jsp` displays a list of the employees in the system, provides links to edit each, and includes a button to add a new employee to the system.

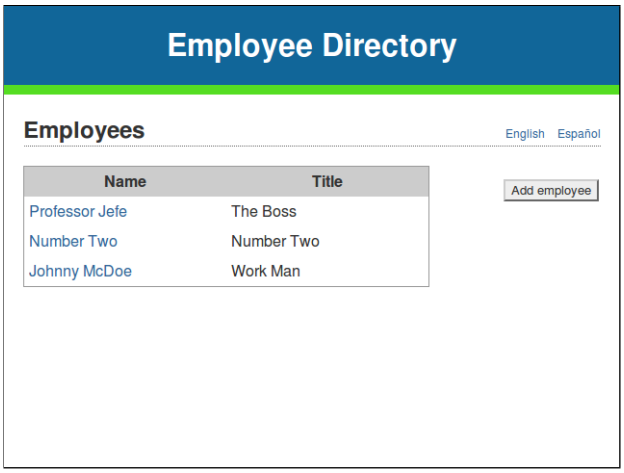

Figure 8: The *employees* view

Listing 8: employees.jsp

```
<ul class="button">
    <li><a href="<c:url value="/employees/new" />">Add Employee</a></li>
</ul>
<table cellpadding="0" cellspacing="0">
    <thead>
        <tr>
            <th>Name</th>
            <th>Title</th>
        </tr>
    </thead>
    <tbody>
        <c:forEach items="${bindableEmployeeList}" var="employee">
            <tr>
                <td>
                    <a href="<c:url value="/employees/${employee.id}" />">
                        <c:out value="${employee.name}" />
                    </a>
                </td>
                <td><c:out value="${employee.title}" /></td>
            </tr>
        </c:forEach>
    </tbody>
</table>
```

3.4 Spring Configuration

Next, the Spring configuration is defined.

Listing 9: spring-mvc-servlet.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:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:annotation-config />
    <mvc:annotation-driven />
    <mvc:default-servlet-handler />

    <bean class="com.earldouglas.barebones.springmvc.web.EmployeeController" />

</beans>
```

The `<mvc:annotation-driven />` element tells Spring to create a `DefaultAnnotationHandlerMapping` bean to set up handling of the `@RequestMapping` annotations in `EmployeeController`, while the lone bean definition registers a view resolver which looks for JSPs by view name. The `EmployeeController` is picked up by the `component-scan`, instantiated, and mapped to its applicable requests by `DefaultAnnotationHandlerMapping`.

Next, we have our Web deployment descriptor.

Listing 10: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5">

    <servlet>
        <servlet-name>spring-mvc</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>spring-mvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

Note that since the `DispatcherServlet` is named `spring-mvc`, by convention the Spring configuration is retrieved from `/WEB-INF/spring-mvc-servlet.xml`.

3.5 Section References

- The `component-scan` tag, the `@Controller` annotation, and classpath scanning is covered in the Spring 3 Reference, section 3.10.
- The `@RequestMapping` annotation is covered in the Spring 3 Reference, section 15.3.2.
- The Spring form tag library is covered in the Spring 3 Reference, section 16.2.4.

- The `DefaultAnnotationHandlerMapping` and handler mappings in general are covered in the Spring 3 Reference, section 18.5.
- Of note is Spring's convention over configuration support with the `ControllerClassNameHandlerMapping` class, covered in the Spring 3 Reference, section 15.10.
- Spring's comprehensive REST support is covered in the Spring 3 Reference, section 2.5.6.1.

4 Server-Side Validation

Form validation goes hand-in-hand with Web applications, and server-side form validation is an easy addition to the core application. JSR-303 Bean Validation specifies annotations for declarative validation rules, which can be standardized across the layers of an enterprise application from the database to the user interface.

4.1 Additions

The following additions are required:

- The JSR-303 validation API `javax.validation` to the Maven POM
- The `@Valid` annotation to controller method inputs
- `Errors` objects to controller method inputs for view error binding
- JSR-303 annotations to `BindableEmployee.java`
- A JSR-303-backed `Validator` to the Spring context
- A JSR-303 reference implementation `hibernate-validator` to the Maven POM
- `<form:errors />` elements to `employee.jsp`

4.2 Classes

There is only one controller method with input: `save(BindableEmployee)`. The `BindableEmployee` parameter is annotated with `@Valid`, which will trigger Spring will use its configured JSR-303 `Validator` to validate the `BindableEmployee`.

Spring needs a place to put the result of the validation, so a `BindingResult` is added to the controller method immediately after the corresponding `BindableEmployee` parameter. This will make binding errors available to the view.

Listing 11: EmployeeController.java

```
@Controller
@RequestMapping("/employees")
public class EmployeeController {

    // Some methods omitted for brevity.

    @RequestMapping(method = RequestMethod.POST)
    public String save(@Valid BindableEmployee bindableEmployee,
            BindingResult bindingResult) {

        if (bindingResult.hasErrors()) {
            return "employee";
        }

        employeeService.save(bindableEmployee.asEmployee());
```

```
        return "redirect:employees";
    }
}
```

JSR-303 annotations are added to `BindableEmployee.java` to limit the pattern of `name` to two words and the pattern of `title` to at least one word.

Listing 12: BindableEmployee.java

```
public class BindableEmployee {

    private Long id;

    @Pattern(regexp = "\\w+ \\w+")
    private String name;

    @Pattern(regexp = "\\w+( \\w+)?")
    private String title;

    // Methods omitted for brevity.

}
```

The Spring MVC namespace will automatically configure a JSR-303-backed `Validator` as long as it is present on the classpath.

4.3 View Templates

Next, the Spring `<form:errors />` element is added to the view to show validation errors.

Listing 13: employee.jsp

```
<ul>
    <li>
        <label for="name"><spring:message code="heading.name" /></label>
        <form:input path="name" />
        <form:errors path="name" />
    </li>
    <li>
        <label for="title"><spring:message code="heading.title" /></label>
        <form:input path="title" />
        <form:errors path="title" />
    </li>
</ul>
```

When the form is submitted, the inputs are automatically validated, and any validation errors are displayed next to each corresponding input field in the form. An example of this is shown in Figure 9.

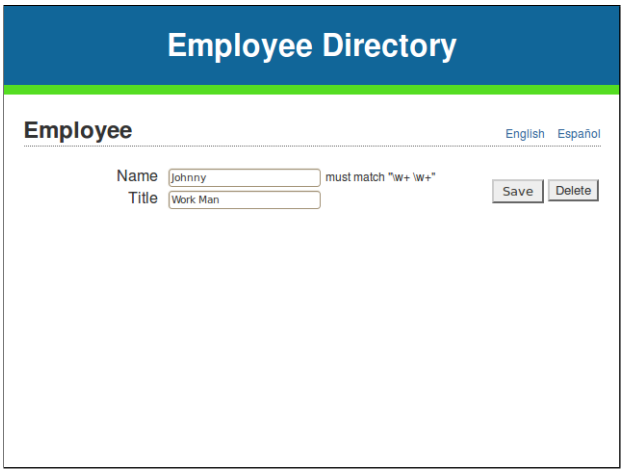

Figure 9: A server-side validation error

4.4 Section References

- Spring's support for the `@Valid` annotation is covered in the Spring 3 Reference, section 5.7.4.1.
- The `BindingResult` and data binding are covered in the Spring 3 Reference, section 5.7.3.
- Spring's support for the JSR-303 Bean Validation API is covered in the Spring 3 Reference, section 5.7.1.

5 Rich Client-Side Validation

The counterpart to server-side validation is client-side validation, which is made easy by Spring JavaScript.

5.1 Additions

The following additions are required:

- `spring-js` to the Maven POM
- The Spring JavaScript `ResourceServlet` to the Web deployment descriptor
- Dojo and Spring JavaScript scripts and layout to the views
- Spring JavaScript validation decorators to the views

5.2 Web Configuration

Spring JavaScript includes `ResourceServlet`, which provides various scripts and CSS layouts from both Dojo and Spring JavaScript. These add the functionality and look-and-feel needed for rich client-side validation. The `ResourceServlet` must be added to the Web deployment descriptor.

Listing 14: web.xml

```
<servlet>
    <servlet-name>Resources Servlet</servlet-name>
    <servlet-class>org.springframework.js.resource.ResourceServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Resources Servlet</servlet-name>
    <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

5.3 View Templates

The Dojo and Spring JavaScript scripts and layout must be added to each view which will provide rich client behavior.

Listing 15: employee.jsp

```
<head>
    <title><spring:message code="application.title" /></title>
    <script type="text/javascript"
        src="<c:url value="/resources/dojo/dojo.js" />"></script>
    <script type="text/javascript"
        src="<c:url value="/resources/spring/Spring.js" />"></script>
    <script type="text/javascript"
        src="<c:url value="/resources/spring/Spring-Dojo.js" />"></script>
```

```
    <link type="text/css" rel="stylesheet"
        href="<c:url value="/resources/dijit/themes/tundra/tundra.css" />" />
    <link type="text/css" rel="stylesheet"
        href="<c:url value="/style.css" />" />
</head>
```

Spring JavaScript uses the decorator pattern to cleanly introduce rich client behavior into views. Script-free HTML is first built to create a fully functioning application, and Spring JavaScript decorators are added *on top of* the existing DOM to introduce rich behavior. This means that a view is fully functional on its own, which allows the application to run in an environment where JavaScript support might be limited or non-existent.

This practice, known as progressive enhancement, allows a Web application to remain functional across a wealth of browsers, which may vary in their level of support of JavaScript and CSS. The most important takeaway from this idea is that the `onclick` attribute is never directly used in HTML code. It is only accessed by a decorator, meaning its behavior is only used when the decorator script itself is supported.

The form in `employee.jsp` is updated to insert Spring JavaScript decorators.

Listing 16: employee.jsp

```
    </form:form>
</div>
<script type="text/javascript">
    Spring.addDecoration(new Spring.ValidateAllDecoration( {
        elementId : "submit",
        event : "onclick"
    }));

    Spring.addDecoration(new Spring.ElementDecoration( {
        elementId : "name",
        widgetType : "dijit.form.ValidationTextBox",
        widgetAttrs : {
            regExp : "\\w+ \\w+",
            required : true
        }
    }));

    Spring.addDecoration(new Spring.ElementDecoration( {
        elementId : "title",
        widgetType : "dijit.form.ValidationTextBox",
        widgetAttrs : {
            regExp : "\\w+( \\w+)?",
            required : true
        }
    }));
</script>
```

Field decorators have been added to all of the form input fields, and a global validation director has been added to the form submission button. These are just a few examples of the vast set of features provided by Dojo.

The *employee* form will now validate on the client, displaying any validation errors dynamically. An example of this is shown in Figure 10.

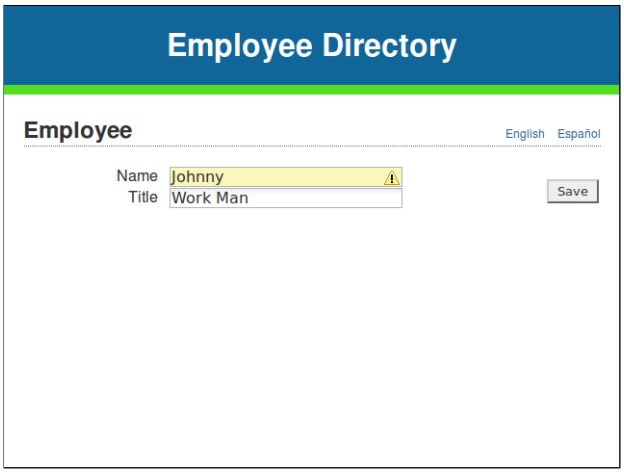

Figure 10: A client-side validation error

5.4 Section References

- Spring JavaScript is currently part of Spring Web Flow, and documentation is available in the Spring Web Flow 2 Reference, section 11.4.
- Dojo form widgets are documented in detail in Dojo's Dijit documentation.

6 Security

A Web application would seldom be complete without at least a minimal security layer to prohibit unauthenticated access to protected resources. In this section, basic security is introduced by adding HTML form-based authentication using Spring Security.

6.1 Additions

The following additions are required:

- Spring Security to the Maven POM
- Spring Security's `DelegatingFilterProxy` to the Web deployment descriptor
- An aplication-level Spring context containing Spring Security configuration

6.2 Web Configuration

Spring Security's `DelegatingFilterProxy` is essentially a J2EE `Filter` which nominally handles all requests and determines how to allow or reject access.

Listing 17: web.xml

```
<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>

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spring-mvc-security.xml
    </param-value>
</context-param>

<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

6.3 Spring Configuration

Spring's `ContextLoaderListener` is needed because there is now a parent Spring context which is extended by the `spring-mvc` context of before. The `contextConfigLocation` parameter specifies the location of the new parent configuration file.

Listing 18: spring-mvc-security.xml

```
<!-- Enable Spring Security with HTTP basic authentication. -->
<http auto-config="true">
    <http-basic />
    <intercept-url pattern="/**" access="ROLE_ADMIN" />
    <form-login />
</http>

<!-- An AuthenticationProvider with sample users and roles. -->
<authentication-manager>
    <authentication-provider>
        <user-service>
            <user name="admin" password="password"
                authorities="ROLE_ADMIN" />
        </user-service>
    </authentication-provider>
</authentication-manager>
```

This nearly minimal configuration sets up an in-memory repository of roles, and enforces access to every resource against this repository. Here, a form-based login page is provided by Spring, as shown in Figure 11.

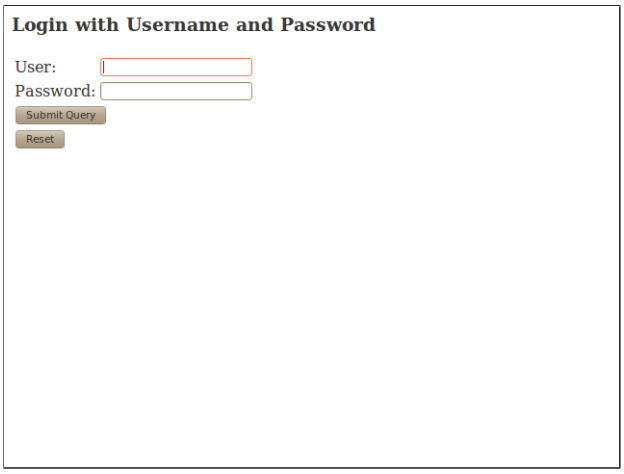

Figure 11: Spring Security form-based login page

6.4 Section References

- Java Servlet Filters are covered in The Essentials of Filters, from Oracle.
- The Spring Security Web site contains the reference documentation, plus useful links to literature and examples.

7 Database Integration

A major component of enterprise applications is information storage and retrieval via a relational database. Most of this behavior is confined to a special data tier, with a thin API exposed to the application tier for interaction with the domain. In this section, a Hibernate-based data tier is introduced for storage and retrieval of data.

7.1 Additions

The following additions are required:

- Multiple libraries to the Maven POM, including `hibernate`, `hibernate-annotations`, `persistence-api`, `jta`, `spring-orm`, `commons-dbcp`, and `hsqldb`
- JPA annotations on `Employee`
- A repository to provide database interaction
- Integration of `EmployeeController` with the repository
- A new Spring context, `persistence-context.xml`
- Optional externalization of the `DataSource` via JNDI

7.2 Classes

A persistable type is created to represent the domain model. In this simple example, this will closely resemble the UI model, but it is important to draw a distinction between the two, as they serve two very different purposes.

The purpose of a domain model is to represent the domain model. The purpose of a UI model is to represent the UI model. This is intentionally redundant, because it is easy to forget. The domain model can include potentially complex object hierarchies as well as database-specific metadata. The UI model will have forms and other data structures which will tend to be very flat, and contain very specific information meant to be rendered in a view.

Attempting to merge the two models can get painfully cumbersome, as the domain model tends not to map directly to the UI model. Furthermore, the resulting tight coupling will force any changes in one to necessitate changes in the other. It is much simpler to create simple conversion logic in the service tier to translate between the two models.

The `Employee` class is made persistable with JPA annotations.

Listing 19: Employee.java

```
@Entity
public class Employee {

    @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column
    private Long id;

    @Column
    private String name;

    @Column
    private String title;

    public Employee() {

    }

    public Employee(Long id, String name, String title) {
        this.id = id;
        this.name = name;
        this.title = title;
    }

    // Getters and setters omitted for brevity.
}
```

A repository serves the domain as an opaque entry point into the database. It provides accessors and mutators for database tables represented only by domain objects.

Listing 20: HibernateEmployeeService.java

```
@Transactional
public class HibernateEmployeeService implements EmployeeService {

    @Autowired
    private SessionFactory sessionFactory;

    private Session session() {
        return sessionFactory.getCurrentSession();
    }

    @Override
    public void delete(Long employeeId) {
        session().delete(get(employeeId));
    }

    @Override
    public Employee get(Long employeeId) {
        return (Employee) session().createCriteria(Employee.class).add(
                Restrictions.idEq(employeeId)).uniqueResult();
    }

    @SuppressWarnings("unchecked")
    @Override
    public Collection<Employee> get() {
        return session().createCriteria(Employee.class).list();
    }

    @Override
    public Employee save(Employee employee) {
        session().saveOrUpdate(employee);
        return employee;
    }
}
```

This repository is meant to work with Hibernate, and so uses a Hibernate `SessionFactory`.

In this example, the service tier is contained entirely within `EmployeeController`, which is modified to interact with the new repository.

`BindableEmployee` provides an `Employee`-based constructor plus two helper methods, `asEmployee()` and `bindableEmployee(Collection<Employee>)`, which do the mapping between the UI model and the domain model.

Listing 21: BindableEmployee.java

```
public class BindableEmployee {

    private Long id;
    private String name;
    private String title;

    public BindableEmployee() {
    }

    public BindableEmployee(Employee employee) {
        this.id = employee.getId();
        this.name = employee.getName();
        this.title = employee.getTitle();
    }

    // Getters and setters omitted for brevity.

    public static Collection<BindableEmployee> bindableEmployees(
            Collection<Employee> employees) {
        Collection<BindableEmployee> bindableEmployees = new ArrayList<BindableEmployee>();
        for (Employee employee : employees) {
            bindableEmployees.add(new BindableEmployee(employee));
        }
        return bindableEmployees;
    }

    public Employee asEmployee() {
        return new Employee(id, name, title);
    }

}
```

7.3 Spring Configuration

Next, a new global Spring context is created to manage the database-related objects.

Listing 22: persistence-context.xml

```
<tx:annotation-driven />

<context:annotation-config />

<bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

```
<bean class="com.earldouglas.barebones.springmvc.service.HibernateEmployeeService" />

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean"
    init-method="createDatabaseSchema">
    <property name="dataSource" ref="hsqlDataSource" />
    <property name="packagesToScan" value="com.earldouglas.barebones.springmvc" />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                org.hibernate.dialect.HSQLDialect
            </prop>
            <prop key="hibernate.show_sql">false</prop>
        </props>
    </property>
</bean>

<bean id="hsqlDataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
    <property name="url" value="jdbc:hsqldb:mem:barebones-spring-mvc" />
    <property name="username" value="sa" />
    <property name="password" value="" />
</bean>
```

This context is made a parent context via `ContextLoaderListener` in the Web deployment descriptor.

Listing 23: web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spring-mvc-security.xml
        /WEB-INF/persistence-context.xml  Barebones Spring MVC: Database Integration
    </param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

A minor change is required in `spring-mvc-servlet.xml` to enable service tier transaction management.

Listing 24: spring-mvc-servlet.xml

```
<!--
    <bean
    class="com.earldouglas.barebones.springmvc.service.InMemoryEmployeeService"
    />
-->
```

The `DataSource` can optionally be externalized from the Spring configuration via JNDI. Configuration specifics, such as database username and password, are then kept out of the Spring configuration and delegated to the application server. This adds security by allowing the application server protect these sensitive data. For Apache Tomcat, the `DataSource` is added to `META-INF/context.xml`.

Listing 25: context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
    <Resource name="jdbc/hsqlDataSource"
              auth="Container"
              type="javax.sql.DataSource"
              username="sa"
              password=""
              driverClassName="org.hsqldb.jdbcDriver"
              factory="org.apache.commons.dbcp.BasicDataSourceFactory"
              url="jdbc:hsqldb:mem:mydatabase"/>
</Context>
```

The `DataSource` bean is removed from the Spring configuration, and replaced by a JNDI-lookup:

Listing 26: persistence-context.xml

```
<jee:jndi-lookup id="hsqlDataSource" jndi-name="java:comp/env/jdbc/hsqlDataSource" />
```

7.4 Section References

- JPA annotations are part of the Java Persistence API, which is covered in Wikipedia.
- Spring's `jee` namespace provides easy JNDI integration, and is covered in the Spring Reference, section C.2.3.
- Hibernate documentation is available from JBoss.
- The multi-tier architecture is covered in Wikipedia.

8 RESTful Web Services

One of the awesome features of the Spring MVC is its ability to easily support multiple types of request/response content. In fact, the same Spring MVC beans can be used to serve conventional HTML, RESTful XML, JSON, Atom, etc. usually with only some minor configuration changes.

This section introduces a RESTful Web service which utilizes the existing Spring MVC beans and configuration.

8.1 Additions

The following additions are required:

- JAXB annotations to `BindableEmployee` to define its XML marshalling configuration
- The `spring-oxm` library to the Maven POM
- A supplement to the `InternalResourceViewResolver` in the Spring context with a `ContentNegotiatingViewResolver` and some JAXB marshalling configuration

8.2 Classes

JAXB annotations are similar in use to Hibernate annotations. In this example, `BindableEmployee` is simple and flat enough that it will marshal easily with a few JAXB annotations.

Listing 27: BindableEmployee.java

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType
@XmlRootElement
public class BindableEmployee {

    @XmlElement
    private Long id;

    @XmlElement
    @Pattern(regexp = "\\w+ \\w+")
    private String name;

    @XmlElement
    @Pattern(regexp = "\\w+( \\w+)?")
    private String title;

    // Methods omitted for brevity.
}
```

Spring MVC needs the ability to choose an appropriate view resolver depending on the specifics of the request. When a conventional `text/html` request is made from a Web browser, Spring MVC uses an `InternalResourceViewResolver` to delegate to a JSP view template as before. When an `application/xml` request is made by a Web service consumer, Spring MVC uses a `MarshallingView` with a JAXB marshaller to provide an XML representation of the `BindableEmployee`.

8.3 Spring Configuration

Listing 28: spring-mvc-servlet.xml

```
<bean
    class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
    <property name="viewResolvers">
        <list>
            <bean
                class="org.springframework.web.servlet.view.InternalResourceViewResolver">
                <property name="prefix" value="/WEB-INF/views/" />
                <property name="suffix" value=".jsp" />
            </bean>
            <bean class="org.springframework.web.servlet.view.BeanNameViewResolver" />
        </list>
    </property>
</bean>

<oxm:jaxb2-marshaller id="marshaller">
    <oxm:class-to-be-bound
        name="com.earldouglas.barebones.springmvc.web.BindableEmployee" />
</oxm:jaxb2-marshaller>

<bean name="employee"
    class="org.springframework.web.servlet.view.xml.MarshallingView">
    <constructor-arg ref="marshaller" />
</bean>
```

8.4 Testing

The HTML/XML duality of this example can be tested with `curl`:

```
> curl -H 'Accept: application/xml' localhost:8080/barebones-spring-mvc/employee
> curl -H 'Accept: text/html' localhost:8080/barebones-spring-mvc/employee
```

8.5 Section References

- Java Architecture for XML Binding (JAXB) is covered by Oracle.
- Representational State Transfer (REST) is covered in Wikipedia
- Spring's comprehensive REST support is covered in the Spring 3 Reference
- cURL is covered by the cURL manual.

9 Externalization and Internationalization

Message externalization in the Web view layer digs the various text out of view templates and keeps it centralized and manageable. It also provides a convenient launchpad for site internationalization. Spring MVC provides for easy introduction of message externalization and internationalization into the view layer.

9.1 Additions

The following additions are required:

- A `ResourceBundleMessageSource` bean to the Spring context
- A resource bundle, messages.properties, to contain messages from the JSP
- Message tanslations from `messages.properties` into Spanish in `messages_es.properties`
- `LocaleChangeInterceptor` and `SessionLocaleResolver` beans to the Spring context

9.2 Spring Configuration

Spring needs to know where it will find externalized messages. This is done with Java's resource bundle functionality, encapsulated in a Spring `ResourceBundleMessageSource`.

Listing 29: spring-mvc-servlet.xml

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value="messages" />
</bean>
```

There isn't much in the way of messages in this example, but the little that is there is moved into a properties file named in the above `ResourceBundleMessageSource`.

Listing 30: message.properties

```
application.title=Employee Directory

button.add-employee=Add employee
button.delete=Delete
button.save=Save

heading.employees=Employees
heading.employee=Employee
heading.name=Name
heading.title=Title

english=English
spanish=Espa ol
```

This is also done in Spanish. Translations were performed with the help of Google Translate, so as far as I know nothing below says "Your mother was a hamster."

Listing 31: messages_es.properties

```
application.title=Directorio de Empleados

button.add-employee=A adir empleado
button.delete=Eliminar
button.save=Guardar

heading.employees=Empleados
heading.employee=Empleado
heading.name=Nombre
heading.title=T tulo
```

Next, a couple of beans are added to the Spring context to allow detection of a user's desire to switch languages, and the ability to store that preference in the user's session.

Listing 32: spring-mvc-servlet.xml

```
<mvc:interceptors>
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor" />
</mvc:interceptors>
<bean id="localeResolver"
    class="org.springframework.web.servlet.i18n.SessionLocaleResolver" />
```

A user simply includes the HTTP request parameter `lang=es` to change the language to Spanish.

Figure 12: The *employee* view in Spanish

10 References

Dijit

http://docs.dojocampus.org/dijit/form/

http://www.dojotoolkit.org/reference-guide/dijit/index.html

Spring 3 Reference

http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/

Spring API

http://static.springsource.org/spring/docs/3.0.x/javadoc-api/

Spring Security

http://static.springsource.org/spring-security/site/

Spring Web Flow

http://static.springsource.org/spring-webflow/docs/2.0.x/reference/html/