



INSPIRING PERFORMANCE



MVC

Nikita Salnikov-Tarnovski

Webmedia AS

05.04.2010



The problem



```
<%  
    String error = request.getParameter("error");  
    if(error != null) {  
%>  
<font color="red"><%= error %></font>  
<%  
    }  
    else {  
        java.util.Date now = new java.util.Date();  
%>  
<font color="green"><%= now %></font>  
<%  
    }  
%>
```



Model 1

- That was called “Model 1” long ago
- No one can really develop it: no programmer, no designer
- Tool support was also wanting
- Who will save us?



Model 2 or MVC



- The Model 2 was invented
- We call it MVC now
- M – is for the Model
- V – is for the View
- C – is for the Controller



Model



- Data or *what* we want to show to our user
- May be just plain Java classes
- May be more sophisticated specialized data structures such as XML
- Should be totally independent from the source of that data.



View

- User interface or *how* we want to show our data to user
- Renders the collected data and is totally oblivious to who gave it to him
- May be a html page, or jsp page, or pdf file, or an image etc



Controller




- Some code that *decides* which data to show and which view to use
- Servlet or some java class that servlet delegates its work to
- The only entry point to the business logic



Why bother?

- Why the complexity?
- Clear separation of concerns
- Can be developed independently and by different people
- Can be replaced independently.



- 
- So MVC is *a way to do things*, it describes how we should design our web application
 - In a nutshell you write servlets, that based on the user input collect some data and to choose how to present that data to the user
 - But that is too low-level
 - We are smart, aren't we? Let us use some *framework!*

Web framework



Most MVC web frameworks provide at least the following

- A way to map request to java classes processing it
- A way to select view component for displaying data to the user
- Reusable components for implementing common logic (sorting a list, submitting a form, validating user input etc)
- Reusable view components for displaying common UI elements (e.g. custom jsp tags)




Spring MVC



- Part of the Spring framework
- <http://www.springsource.org/>
- Open-source project, very good documentation
- <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/mvc.html>
- Request based web framework
- We will talk about Spring 3.0



- 
-
- Spring has a very interesting architecture
 - Its aim is to give possibility to develop your application with low-coupled high-cohesion components
 - Your code must not depend on the environment
 - Reuse and testability are in great honor



MVC in Spring



- Model

- You use plain Java classes, either standard or your domain specific

- View

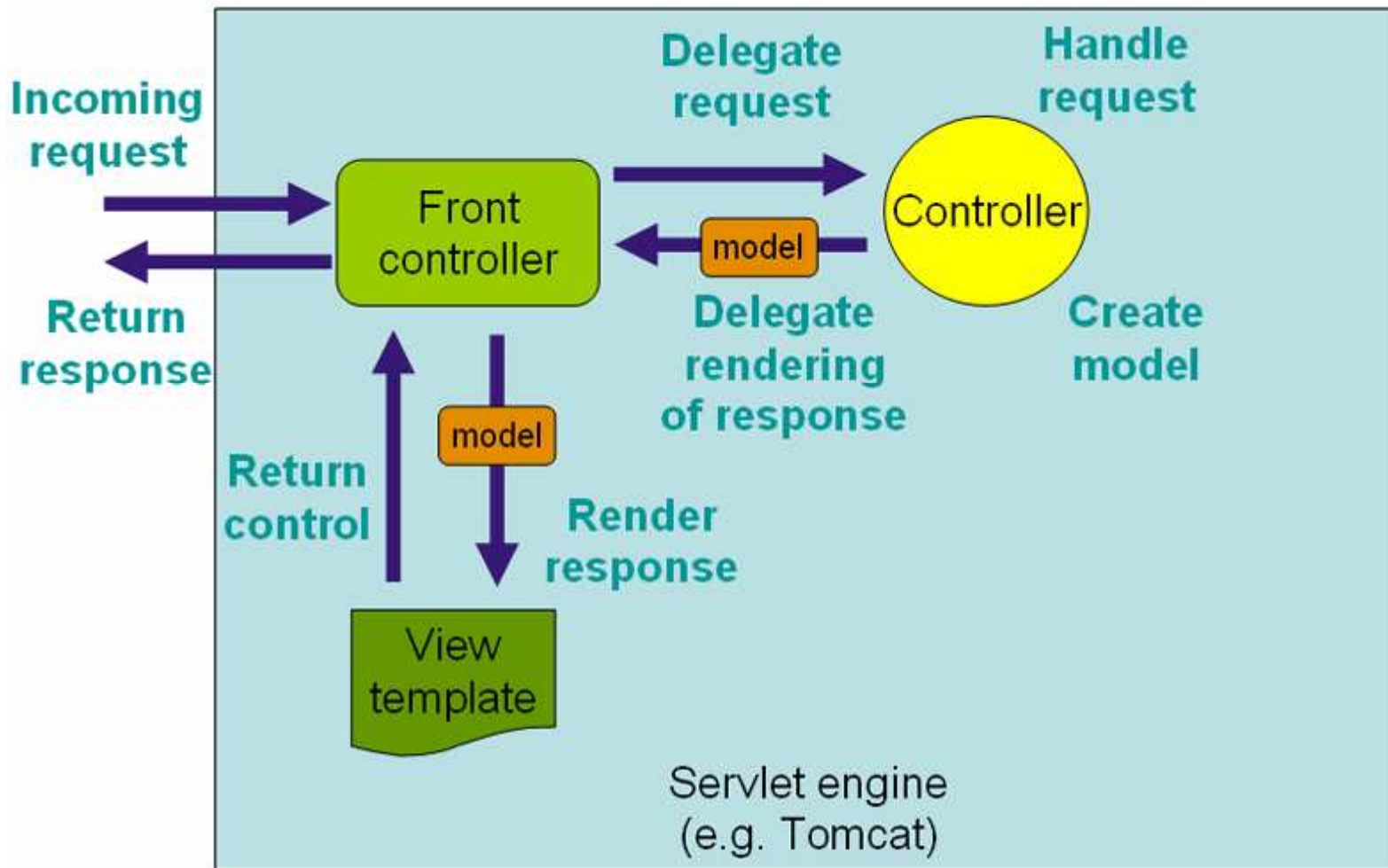
- A number of view technologies are supported out-of-the-box and you can plug-in your own

- Controller

- You use plain Java classes as your controller, without dependencies on Servlet API



Request lifecycle



DispatcherServlet

- The entry point to your Spring application
- The standard JEE Servlet
- Must be defined in web.xml
- Configures Spring WebApplicationContext
- And routes incoming requests to your application's controllers

Components



- **Controllers**

- Form the C part of the MVC.

- **Handler mappings**

- Handle the execution of a list of pre-processors and post-processors and controllers that will be executed if they match certain criteria (for example, a matching URL specified with the controller).

- **View resolvers**

- Resolves view names to views.

- **Locale resolver**

- A locale resolver is a component capable of resolving the locale a client is using, in order to be able to offer internationalized views




Components



- Theme resolver
 - A theme resolver is capable of resolving themes your web application can use, for example, to offer personalized layouts
- Multipart file resolver
 - Contains functionality to process file uploads from HTML forms.
- Handler exception resolvers
 - Contains functionality to map exceptions to views or implement other more complex exception handling code.

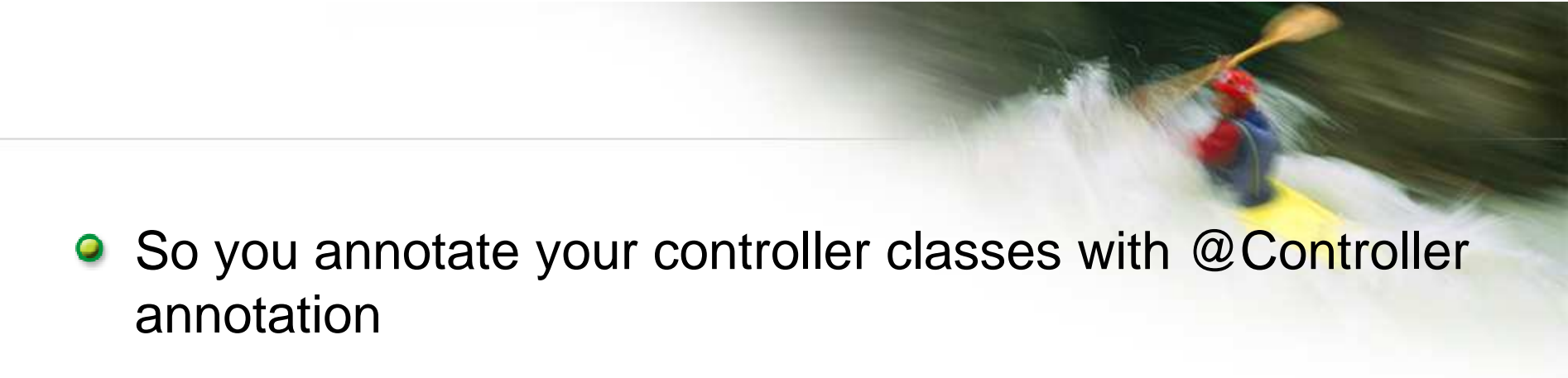


- 
-
- So you write your Java classes, implementing some of the components above
 - Annotate them with a bunch of Spring annotations
 - Let the DispatcherServlet know of them
 - And you are ready to serve your clients!

Your first controller

@Controller

```
public class HelloWorldController {  
    @RequestMapping("/helloWorld")  
    public ModelAndView helloWorld() {  
        ModelAndView mav = new ModelAndView();  
        mav.setViewName("helloWorld");  
        mav.addObject("message", "Hello World!");  
        return mav;  
    }  
}
```

- 
- So you annotate your controller classes with `@Controller` annotation
 - No superclass must be extended
 - Using `@RequestMapping` annotation you say to SpringMVC requests to which URL this class or method should service
 - And you do not think at all about servlets, `HttpRequests`, `HttpSessions` etc

Examples

```
@Controller
```

```
@RequestMapping("/appointments")
```

```
public class AppointmentsController {
```

```
@RequestMapping(method = RequestMethod.GET)
```

```
public Map<String, Appointment> get() {
```

```
    return appointmentBook.getAppointmentsForToday(); }
```

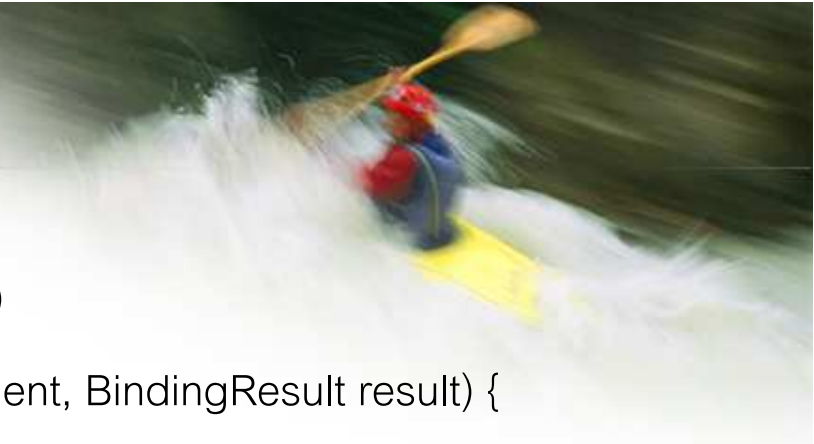
```
@RequestMapping(value="/new", method = RequestMethod.GET)
```

```
public AppointmentForm getNewForm() { return new AppointmentForm(); }
```



Examples

```
@RequestMapping(method = RequestMethod.POST)
public String add(@Valid AppointmentForm appointment, BindingResult result) {
    if (result.hasErrors()) {
        return "appointments/new";
    }
    appointmentBook.addAppointment(appointment);
    return "redirect:/appointments"
}
```



URI templates

```
@RequestMapping(  
    value="/owners/{ownerId}", method=RequestMethod.GET)  
public String findOwner(@PathVariable String ownerId, Model model) {  
    Owner owner = ownerService.findOwner(ownerId);  
    model.addAttribute("owner", owner);  
    return "displayOwner";  
}
```

<http://www.example.com/owners/1234>



More examples



- You can “nest” URI templates
 - /owners/42/pets/21
- You can specify that method serves the request only if some request header is given

```
@RequestMapping(  
    value = "/pets",  
    method = RequestMethod.POST,  
    headers="content-type=text/*")
```

- You can map request parameters to arguments

```
public String setupForm(@RequestParam("petId") int petId,  
                        ModelMap model) {
```



See also

- For the extensive list of supported methods' arguments and return types see Spring MVC documentation
- <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/mvc.html#mvc-ann-requestmapping-arguments>



Interceptors



- You can provide pre- or post-processors for the requests and response
- To implement security, caching, locale switching, client routing etc



View resolving

- Your controllers return a *logical view name*
- And doesn't concerns themselves what does it really mean
- You use *view resolvers* to find out what to display to the client
- It may be jsp file, redirect to another url, Excel file or whatever





- Spring has a number of built-in view resolvers
- Each with its own configuration of how to map logical names to real jsp files e.g.
- You can chain them, so the first one able to serve the requested view will do it



Example

```
<bean id="viewResolver"  
    class="org.springframework.web.servlet.view.UrlBasedViewResolver">  
    <property name="viewClass"  
        value="org.springframework.web.servlet.view.JstlView"/>  
    <property name="prefix" value="/WEB-INF/jsp/" />  
    <property name="suffix" value=".jsp" />  
</bean>
```



Views

- Spring provides integration with different view technologies
 - JSP, Tiles, Velocity, FreeMaker, XSLT, Document views such as PDF and Excel, JasperReports, Feeds, XML Marshalling, JSON
- For more information RTFM

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



Spring MVC and JSP

- Spring provides custom tag library for JSPs
- It serves mostly 2 purposes:
 - Displaying html elements using model data provided by the controller
 - Displaying validation errors for those html elements

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```





```
<form:form commandName="user">
```

```
  <table>
```

```
    <tr>
```

```
      <td>First Name:</td>
```

```
      <td><form:input path="firstName" /></td>
```

```
    </tr>
```

```
    <tr>
```

```
      <td> <input type="submit" value="Save Changes" /> </td>
```

```
    </tr>
```

```
  </table>
```

```
</form:form>
```



Validation errors

```
<td>First Name:</td>
```

```
<td><form:input path="firstName" /></td>
```

```
<%-- Show errors for firstName field --%>
```

```
<td><form:errors path="firstName" /></td>
```



Binding and Validation



- So we have seen a lot of magic
- You can map requests to methods with arbitrary signatures
- You can use very different, almost arbitrary, types for those methods' parameters
- But wait, HTTP parameters and headers are all strings!



Conversions

- Spring uses a couple of mechanisms to convert between different types
- E.g. from HTTP's Strings to other types, including your domain ones
- And vice versa of course, to display your model data on HTML page



PropertyEditors

- Part of JavaBeans specification
- PropertyEditors for standard java.lang. types are provided by Java itself
- You write your own PropertyEditor for your specific class
- They convert between given type and String and vice versa.



Conversion SPI

- Spring has its own more generic mechanism for converting between 2 arbitrary types
- And some helper classes to simplify your work of writing them for you own domain
- More information

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



Validation



- But what if input data is incorrect and you cannot convert it into your destination type?
- You use validation capabilities of Spring
- Validation errors can be displayed to the user
- In an intended way



Validation example

```
public class PersonValidator implements Validator {  
    /** * This Validator validates just Person instances */  
    public boolean supports(Class clazz) {  
        return Person.class.equals(clazz);  
    }  
    public void validate(Object obj, Errors e) {  
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");  
        Person p = (Person) obj;  
        if (p.getAge() < 0) { e.rejectValue("age", "negativevalue"); }  
    }  
}
```



JSR-303

- There is a standard specification for Java object's constraints definition and validation

```
public class PersonForm {  
    @NotNull  
    @Size(max=64)  
    private String name;  
    @Min(0)  
    private int age;  
}
```

Spring MVC and validations



- You can tell Spring MVC to validate inputs

```
@RequestMapping("/foo", method=RequestMethod.POST)
```

```
public void processFoo(@Valid Foo foo, BindingResult result) { /* ... */ }
```

You must register all your custom validators and converters, of course





WORK HARD. PLAY HARD.

Thank You. Questions?

