

Basic Spring 4.0

Lesson 4: Spring MVC framework

Lesson Objectives

- Introduction to Spring MVC framework
 - Learn how to develop web applications using Spring
 - Understand the Spring MVC architecture and the request cycle of Spring web applications
 - Understand components like handler mappings, ViewResolvers and controllers
 - Use MVC Annotations like `@Controller`, `@RequestMapping` and `@RequestParam`
 - Introduction to REST web Services
 - REST Controllers on the top of MVC



Web applications have become a very important part of any enterprise system. The key requirements for a web framework is to simplify development of the web tier as much as possible. Spring provides a web framework based on the MVC (Model view Controller) paradigm. Although it is similar in some ways to other popular MVC frameworks such as Struts and WebWork, Spring web MVC provides significant advantages over those frameworks.

Spring MVC helps in building flexible and loosely coupled web applications. The Model-view-controller design pattern helps in separating the business logic, presentation logic and navigation logic. Models are responsible for encapsulating the application data. The Views render response to the user with the help of the model object. Controllers are responsible for receiving the request from the user and calling the back-end services.

4.1 : Spring MVC introduction

Spring MVC Framework Features

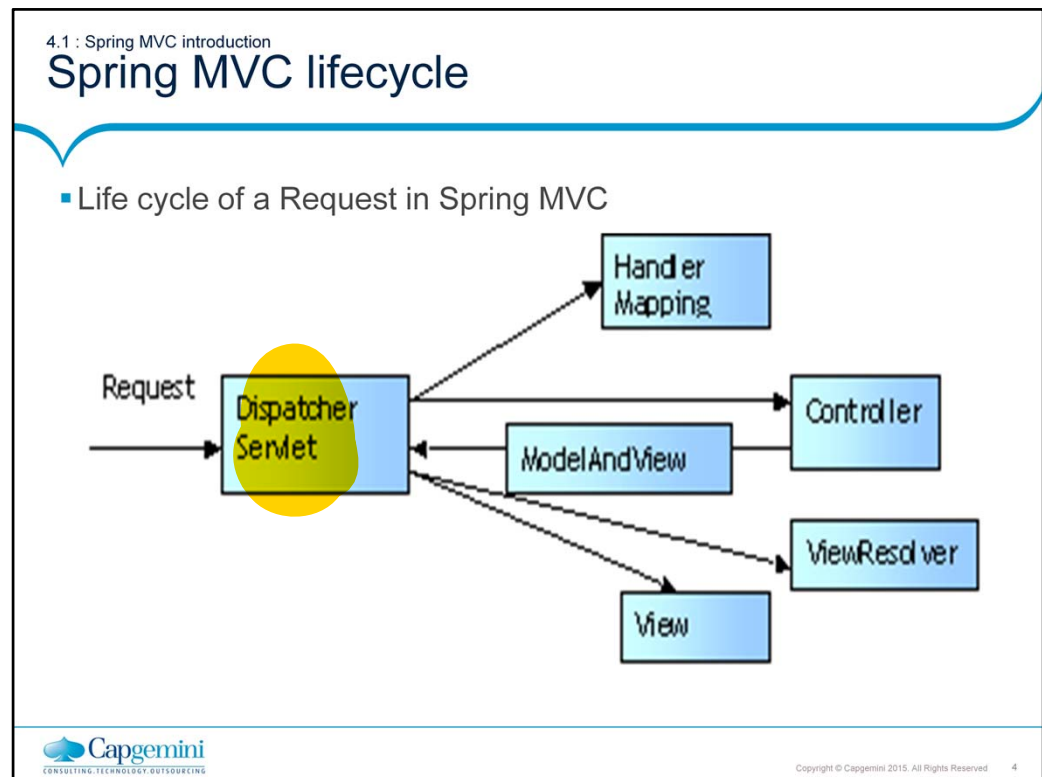
- Provides you with an out-of-the-box implementations of workflow typical to web applications
- Allows you to use a variety of different view technologies
- Enables you to fully integrate with your Spring based, middle-tier logic through the use of dependency injection
- Displays modular framework, with each set of components having specific roles and completely decoupled from the rest of the framework



Copyright © Capgemini 2015. All Rights Reserved 3

The Spring MVC is a web framework built within the Spring Framework. There are a number of challenges while creating a web-based application - like state management, workflow and validation, which are addressed by Spring's web framework. This framework can be used to automatically populate your model objects from incoming request parameters while providing validation and error handling as well. The entire framework is modular, with each set of components having specific roles and completely decoupled from the rest of the framework. This allows you to develop the front end of your web application in a very pluggable manner.

MVC provides out-of-the-box implementations of workflow typical to web applications. Its highly flexible, allowing you to use a variety of different view technologies. It also enables you to fully integrate with your Spring based, middle-tier logic through the use of dependency injection. You can use Spring web MVC to make services that are created with other parts of Spring available to your users, by implementing web interfaces.



Life cycle of a request in Spring MVC: From the time that a request is received by Spring until the time that a response is returned to the client, many pieces of Spring MVC framework are involved.

The process starts when a client (typically a web browser) sends a request. It is first received by a DispatcherServlet. Like most Java-based MVC frameworks, Spring MVC uses a front-controller servlet (here DispatcherServlet) to intercept requests. This in turn delegates responsibility for a request to other components of an application for actual processing.

The Spring MVC uses a Controller component for handling the request. But a typical application may have several controllers. To determine which controller should handle the request, DispatcherServlet starts by querying one or more HandlerMappings. A HandlerMapping typically maps URL patterns to Controller objects.

Once the DispatcherServlet has a Controller object, it dispatches the request to the Controller which performs the business logic (a well-designed Controller object delegates responsibility of business logic to one or more service objects). Upon completion of business logic, the Controller returns a ModelAndView object to the DispatcherServlet. The ModelAndView object contains both the model data and logical name of view. The DispatcherServlet queries a ViewResolver with this logical name to help find the actual JSP. Finally the DispatcherServlet dispatches the request to the View object, which is responsible for rendering a response back to the client.

4.1 : Spring MVC introduction

Dispatcher Servlet

- The central component of Spring MVC is **DispatcherServlet** .
- It acts as the front controller of the Spring MVC framework
- Every web request must go through it so that it can manage the entire request-handling process.

4.1 : Spring MVC introduction

Configuring the Dispatcher Servlet in web.xml

```
<servlet>
  <servlet-name>basicspring</servlet-name>
  <servlet-class> org.springframework.web.servlet.DispatcherServlet
</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>basicspring</servlet-name>
  <url-pattern>*.obj</url-pattern>
</servlet-mapping>
```

The servlet-name
given to the servlet
is significant

- Steps to build a homepage in Spring MVC:
 - Write the controller class that performs the logic behind the homepage
 - Configure controller in the DispatcherServlet's context configuration file
 - Configure a view resolver to tie the controller to the JSP
 - Write the JSP that will render the homepage to the user



Copyright © Capgemini 2015. All Rights Reserved 6

Configuring the DispatcherServlet

The **dispatcher servlet** is at the heart of the Spring MVC and functions as Spring MVC's front controller. Like any other servlet, **it must be configured in web.xml** file. Place the **<servlet>** declaration (in the first listing above) in the web.xml file.

The servlet-name given to the servlet is significant. By default, when DispatcherServlet is loaded, it will load the Spring application context from an xml file whose name is based on the name of the servlet. In the above example, **the servlet-name is basicspring** and so the DispatcherServlet will try to load the application context from a file called **basicspring-servlet.xml**.

Next, you must indicate which URL's will be handled by **DispatcherServlet**. Add the following **<servlet-mapping>** tag (the second xml listing) to web.xml to let DispatcherServlet handle **all url's that end in .obj**. The URL pattern is arbitrary and could be any extension.

DispatcherServlet is now configured and ready to dispatch requests to the web layer of your application.

Pls. see the above steps that are followed while building a simpleMVC application. These steps are explained in detail in the coming slides.


4.1 : Spring MVC introduction

Breaking Up the Application Context

- Configuring the ContextLoaderListener in web.xml

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

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/basicsspring-service.xml
    /WEB-INF/basicsspring-data.xml
  </param-value>
</context-param>
```

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 7

We have seen that DispatcherServlet will load the Spring application context from a single XML file whose name is `<servlet-name>-servlet.xml`. But you can also split the application context across multiple XML files. Ideally splitting it into logical pieces across application layers can make maintenance easier by keeping each of the Spring configuration files focused on a single layer of the application. To ensure that all the configuration files are loaded, you will need to configure a context loader in your web.xml file. A context loader loads context configuration files in addition to the one that DispatcherServlet loads. The most commonly used context loader is a servlet listener called ContextLoaderListener that is configured in web.xml as shown in the first listing above.

With ContextLoaderListener configured, we need to tell it the location of the Spring configuration file(s) to load. If not specified, context loader will look for a Spring configuration file at `/WEB-INF/applicationContext.xml`. We will specify one or more Spring configuration file(s) by setting the contextConfigLocation parameter in the servlet context as seen in the second listing above. The context loader will use contextConfigLocation to load the two context configuration files – one each for the service and data layer.

4.2 Annotation-based controller configuration

Controller

- The most typical handler used in Spring MVC for handling web requests is a controller.
- From Spring2.5 onwards `@Controller` annotation is used to design a controller class

```
@Controller
public class HelloController {
    @RequestMapping("/helloWorld")
    public String showMessage() {
        return "hello";
    }
}
```


4.2 : Annotation-based controller configuration

Implementing Controllers

| Annotation Name | Description |
|-----------------|--|
| @Controller | Indicates that an annotated class is a "Controller" |
| @RequestMapping | Map web requests onto specific handler classes and/or handler methods. |
| @RequestParam | @RequestParam annotation is used to retrieve the URL parameter and map it to the method argument. |
| @ModelAttribute | Annotation that binds a method parameter or method return value to a named model attribute, exposed to a web view. |



Copyright © Capgemini 2015. All Rights Reserved 9

@Controller: Indicates that an annotated class is a "Controller"

Example:

```
@Controller
```

```
Public class LoginController{
```

@RequestMapping: @RequestMapping can be applied to the controller class as well as methods. It maps web requests onto specific handler classes and/or handler methods.

Example:

```
@Controller
```

```
@RequestMapping("loginController")
```

```
Public class LoginController{
```

```
    @RequestMapping("loadForm")
```

```
    public String loadData(){}
```

```
}
```

URL need to be used for making the request for the above mentioned controller is

```
http://localhost:8080/projectname/loginController/loadForm.obj
```

Example:

```
@Controller
Public class LoginController{
    @RequestMapping("loadForm")
    public String loadData(){}
```

URL need to be used for making the request for the above mentioned controller is

<http://localhost:8080/projectname/loadForm.obj>

@RequestParam annotation is used to retrieve the URL parameter and map it to the method argument.

Example:

```
@Controller
Public class LoginController{
    @RequestMapping("checkLogin")
    public String isValidUser(@RequestParam("username"
String uname)
    {}
}
```

URL need to be used for making the request for the above mentioned controller is

<http://localhost:8080/projectname/checkLogin.obj?username=igate>

@ModelAttribute Annotation that binds a method parameter or method return value to a named model attribute, exposed to a web view.

Example:

```
@Controller
Public class LoginController{
    @RequestMapping("checkLogin")
    public String isValidUser(@ModelAttribute("user")
UserBean userBean)
    {}
}
```

4.2.1 Building a basic Spring MVC application

Handler Mapping

- A handler mapping is a bean configured in the web application context that implements the Handler Mapping interface. It is responsible for returning an appropriate handler for a request.
- Handler mappings usually map a request to a handler according to the request's URL. It is an arbitrary Java object that can handle web requests.
- The most typical handler used in Spring MVC for handling web requests is a controller.

4.2.1 Building a basic Spring MVC application

ModelAndView Class

- This class fully encapsulates the view and model data that is to be displayed by the view. Eg:

```
ModelAndView("hello","now",now);
```

```
Map myModel = new HashMap();  
myModel.put("now",now);  
myModel.put("products",getProductManager().getProducts());  
return new ModelAndView("product","model",myModel);
```

- Every controller returns a ModelAndView
- Views in Spring are addressed by a view name and are resolved by a view resolver



Copyright © Capgemini 2015. All Rights Reserved 12

The ModelAndView class allows you to specify a response to the client, resulting from actions performed in controller. This object holds both – the view the client will be presented with and the model used to render the view. Every controller must return a ModelAndView. See the two examples above.

The first parameter is the logical name of a view component that will be used to display the output from this controller. The next two parameters represent the model object that will be passed to the view and its value.

In the second example, the view component is product and the model is an object that is to be returned and its value is MyModel which is a HashMap object containing multiple values.

In the end, the dispatcher servlet needs a concrete view instance to render the view.

4.2.1 Building a basic Spring MVC application

ModelAndView

- After a controller has finished handling the request, it returns a model and a view name, or sometimes a view object, to DispatcherServlet.
- The model contains the attributes that the controller wants to pass to the view for display.
- If a view name is returned, it will be resolved into a view object for rendering. The basic class that binds a model and a view is ModelAndView.

```
@Controller
public class HelloController {
    @RequestMapping("/helloWorld")
    public String handleMyRequest(
        Map<String, Object> model) {
        String now = new java.util.Date().toString();
        model.put("now", now);
        return "hello";
    }
}
```

```
@Controller
public class HelloController {
    @RequestMapping("/helloWorld")
    public ModelAndView handleRequest(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ..... {
        String now = new java.util.Date().toString();
        return new ModelAndView(
            "hello", "now", now );
    }
}
```



Copyright © Capgemini 2015. All Rights Reserved 13

Spring 2.5 introduced an annotation-based programming model for MVC controllers that uses annotations such as `@Controller`, `@RequestMapping`, `@RequestParam`, `@ModelAttribute` etc. Controllers implemented in this style do not have to extend specific base classes or implement specific interfaces.

See the listing above for a simple example.

`@Controller` annotation indicates that this class is a controller class. This is a specialization of the `@Component` annotation. Thus `<context: component-scan>` will pick up and register `@Controller`-annotated classes as beans, just as if they were annotated with `@Component`.

`@RequestMapping` annotation serves two purposes. First, it identifies `handleRequest()` as a request-handling method. Second, it specifies that this method should handle requests whose path is `/helloWorld`. See the second listing above.

Notice that `handleMyRequest()`, a user-defined method, replaces the `handleRequest()` method. `@RequestMapping` annotation identifies it as a request-handling method.

The `handleMyRequest()` takes a `Map` as a parameter, which represents the model—the data that's passed between the controller and a view. But the signature of a request-handling method can have anything as an argument.

Notice that `handleMyRequest()` returns a `String` value which is the logical name of the view that should render the results. `ViewResolver` uses this to resolve actual view.

Note : `@RequestMapping` annotation can be used at class level too to map URLs onto an entire class. The `@RequestMapping` at class level defines the root URL path that the controller will handle. Method-level `@RequestMapping`s narrow the scope of class-level `@RequestMapping`.

4.2.1 Building a basic Spring MVC application

ViewResolver

- When DispatcherServlet receives a model and a view name, it will resolve the logical view name into a view object for rendering.
- DispatcherServlet resolves views from one or more view resolvers.
- A view resolver is a bean configured in the web application context that implements the ViewResolver interface.
- Its responsibility is to return a view object for a logical view name.

4.2.1 Building a basic Spring MVC application

Resolving Views: The ViewResolver

| View resolver | How it works |
|------------------------------|---|
| InternalResourceViewResolver | Resolves logical view names into View objects that are rendered using template file resources |
| BeanNameViewResolver | Looks up implementations of the View interface as beans in the Spring context, assuming that the bean name is the logical view name |
| ResourceBundleViewResolver | Uses a resource bundle that maps logical view names to implementations of the View interface |
| XmlViewResolver | Resolves View beans from an XML file that is defined separately from the application context definition files |



Copyright © Capgemini 2015. All Rights Reserved 15

So far, we have seen how model objects are passed to the view through the ModelAndView object. In Spring MVC, a view is a bean that renders results to the user. The view most likely is a JSP. But you could also use other view technologies like Velocity and FreeMarker templates or even views that produce PDF and MS-Excel documents.

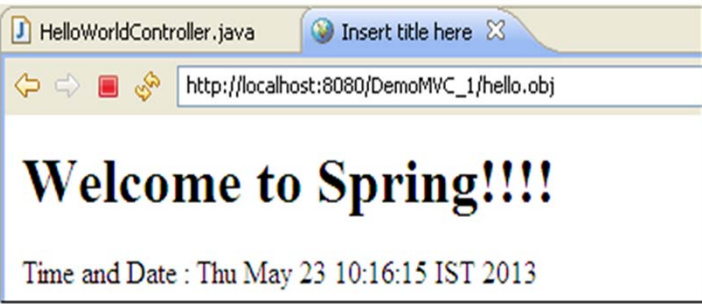
View resolvers resolve the view name given by the ModelAndView object to a View bean. Spring provides a number of useful view resolvers, some of which are shown in the table above. See Spring docs for more.

InternalResourceViewResolver resolves a logical view name by affixing a prefix and a suffix to the view name returned by the ModelAndView object. It then loads a View object with the path of the resultant JSP. By default, the view object is an InternalResourceView, which simply dispatches the request to the JSP to perform the actual rendering. But, if the JSP uses JSTL tags, then you may replace InternalResourceView with JstlView as seen in the code demos earlier.

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

Demo

- Refer DemoMVC_1



Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 16

Please refer to the DemoMVC_1 web project.

4.2.2 : Spring MVC annotations

Handling User Input

```
@Controller
public class LoginFormController {
    @RequestMapping(value = "/login", method = RequestMethod.GET)
    public String onSubmit(@RequestParam("username") String username,
        @RequestParam("password") String password, Model model) {

        model.addAttribute("username", username);
        if (username.equals("majrul") && password.equals("majrul123"))
            return "success";
        else return "failure";
    }
}
```



output



Copyright © Capgemini 2015. All Rights Reserved 17

We have seen simple Controllers so far. But Controllers may need to perform business logic using information passed in as URL parameters or as submitted form data.

See the code listing above for an example. Here, `LoginFormController` is mapped to `/login` at the method level. This means that `onSubmit()` handles requests for `/login`. "method" attribute is set to `GET` indicating that this method will only handle HTTP GET requests for `/login`. The `onSubmit()` method takes two `String` parameters (`username` and `password`) and a `Model` object as parameters. The `username` parameter is annotated with `@RequestParam("username")` to indicate that it should be given the value of the `username` query parameter in the request. Same goes for `password` too. `onSubmit()` will use these parameters to authenticate user.


Now, see the `model` parameter. In earlier examples we passed in a `Map<String, Object>` to represent the model. Here we're using a new `Model` parameter. `Model` provides a few convenient methods for populating the model, such as `addAttribute()`. The `addAttribute()` method is similar to `Map`'s `put()` method, except that it finds out key part of the map by itself.

Notice how the values set in `addAttribute()` can be accessed in the jsp page:

```
<!-- success.jsp ->
<body>
<h1>Welcome to our Application!!</h1>
    Login Valid! Welcome dear <i><b>${username}</b></i>
</body>
```

Demo

- Refer DemoMVC_2 application



Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 18

Please refer to the DemoMVC_2web project.
Refer to HelloWorldController.java.
Invoke the application as :
http://localhost:8080/DemoMVC_2/hi/hello.obj?name=CapGemini

4.2.2: Spring MVC annotations

Validating input with Bean Validation

- Bean Validation (JSR – 303) Annotations:

| Annotation Name | Description |
|-----------------------------------|---|
| Annotations for validation | |
| @Valid | To trigger validation of a @Controller input |
| @Size | Validates that the fields meet criteria on their length. |
| @NotNull | Validates that the fields contains value. |
| @Pattern | @Pattern annotation along with a regular expression ensures that the entered value is valid |
| @Email | Validates that the field value is a valid emailid. |
| @DateTimeFormat | In Spring New Date & Time API can be used in Controllers for Form Binding |



Copyright © Capgemini 2015. All Rights Reserved 19

Before an object can be processed further, it is essential to ensure that all the data in the object is valid and complete. Faulty form input must be rejected. For example, username must not contain spaces, password must be minimum 6 characters long, email must be correct etc.

The @Valid annotation (part of the JavaBean validation specification) tells Spring that the User object should be validated as it's bound to the form input. If anything goes wrong while validating the User object, the validation error will be carried to the processForm() method via the BindingResult that's passed in on the second parameter. If the BindingResult's hasErrors() method returns true, then that means that validation failed.

How do we declare validation rules?

JSR-303 defines some annotations that can be placed on properties to specify validation rules. The code above shows the properties of the User class that are annotated with validation annotations.

@Size annotation validates that the fields meet criteria on their length.

@Pattern annotation along with a regular expression ensures that the value given to the email property fits the format of an email address and that the username is only made up of alphanumeric characters with no spaces.

Notice how we've set the message attribute with the message to be displayed in the form when validation fails. With these annotations, when a user submits a registration form to AddUserFormController's processForm() method, the values in the User object's fields will be validated. If any of those rules are violated, then the handler method will send the user back to the form.

@Valid : To trigger validation of a @Controller input

@Controller

```
Public class LoginController{
    @RequestMapping("checkLogin")
    public String isValidUser(@ModelAttribute("user")
    @Valid UserBean userBean)
    {}
}
```

@Size: Validates that the fields meet criteria on their length.

@Pattern annotation along with a regular expression ensures that the entered value is valid

Example:

```
public class UserBean
{
    @Size(min=7,max=10,message="Phone Number Should Accept
Only 10 digits")
    @Pattern(regexp = "[0-9]+$", message = "Phone Number should
contain only 10 digits")
    private String phoneNumber;
    //Getter and Setter methods
}
```

@NotNull / @NotEmpty: Validates that the fields contains value.

Example:

```
@NotEmpty(message="Please Enter Address")
private String address;
```

@Email: Validates that the field value is a valid emailid.

Example:

```
@Email(message="Please enter valid Email ID")
private String email;
```

@DateTimeFormat: In Spring New Date & Time API can be used in Controllers for Form Binding

Example:

```
@DateTimeFormat(pattern="M/d/yy h:mm")
private LocalDateTime birthDate;
```

4.2.2: Spring MVC annotations

Validating input : declaring validation rules

```
public class User {  
    @Size(min = 3, max = 20, message = "Username must be between 3 and 20  
characters long.")  
    @Pattern(regex = "[a-zA-Z0-9]+$", message = "Username must be alphanumeric  
with no spaces")  
    private String username;  
  
    @Size(min = 6, max = 20, message = "The password must be at least 6 characters  
long.")  
    private String password;  
  
    @Pattern(regex = "[A-Za-z0-9]+@[A-Za-z0-9.-]+[.][A-Za-z]{2,4}", message =  
"Invalid email address.")  
    private String email;  
  
    //getter and setter methods for all these properties  
}
```

4.2.2 : Spring MVC annotations

Processing forms : The JSP

addUser.jsp

```

<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
<sf:form method="POST" modelAttribute="user">
<table cellpadding="0">
<tr>
<th><sf:label path="username">Username:</sf:label></th>
<td><sf:input path="username" size="15" maxlength="15" />
<small id="username_msg">No spaces, please.</small><br />
<sf:errors path="username" /></td>
</tr>
<tr>
<th><sf:label path="password">Password:</sf:label></th>
<td><sf:password path="password" size="30" showPassword="true"/>
<small>6 characters or more (be tricky!)</small><br />
<sf:errors path="password" />
</td>
</tr>
<tr><th></th></tr>
<td><input name="commit" type="submit" value="Save User" /></td></tr>
</sf:form></div>

```



Copyright © Capgemini 2015. All Rights Reserved 22

The addUser.jsp page uses Spring's form binding library. The `<sf:form>` tag binds the User object (identified by the `modelAttribute` attribute - see above) that `showForm()` placed into the model to the various fields in the form. The `<sf:input>` and `<sf:password>` tags have a `path` attribute that references the property of the User object that the form is bound to. When the form is submitted, whatever values these fields contain will be placed into a User object and submitted to the server for processing.

Note that the `<sf:form>` specifies that it'll be submitted as an HTTP POST request. We thus now need another handler method that accepts POST requests. See the `processForm()` method in the code listing in the previous page which will process form submissions.

When the addUser.jsp form is submitted, the fields in the request will be bound to the User object (passed as an argument to `processForm()`). From there, some logic can be employed to persist user object into database.

Notice that User parameter is annotated with `@Valid`. This indicates that the User should pass validation before being persisted. We shall cover validation next.

4.2.2 : Spring MVC annotations


Displaying validation errors

```
<td>
  <sf:password path="password" size="30" showPassword="true"/>
  <small>6 characters or more (be tricky!)</small><br/>
  <sf:errors path="password" />
</td>
```

jsp

```
public String processForm(@Valid User user, BindingResult
bindingResult) {
    if (bindingResult.hasErrors()) {
        return "failure";
    }
    ....
}
```

controller

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 23

We now need to tell jsp to display the validation messages.

We have passed the `BindingResult` parameter to `processForm()`. This knows whether the form had any validation errors via its `hasErrors()` method. But the actual error messages are also in there, associated with the fields that failed validation. To display those errors to the users, use Spring's form binding JSP tag library to display the errors. ie, the `<sf:errors>` tag can render field validation errors. See code listing above to see how this is done.

The `<sf:errors>` tag's `path` attribute specifies the form field for which errors should be displayed. For example, the above listing displays errors (if any) for the field whose name is `password`. If there are multiple errors for a single field, they'll all be displayed, separated by an HTML `
` tag.

4.2.2 : Spring MVC annotations

Processing forms : The controller class

```

@Controller
public class AddUserFormController {
    @RequestMapping(value = "/AddUser", method = RequestMethod.GET)
    public String showForm(Model model) {
        model.addAttribute(new User());
        return "addUser";
    }
    @RequestMapping(method = RequestMethod.POST)
    public String processForm(@Valid User user, BindingResult bindingResult) {
        if (bindingResult.hasErrors()) return "failure";
        else {
            // some logic to persist user
            return "success";
        }
    }
}

```

addUser.jsp

Working with forms in a web application involves two operations: displaying the form and processing the form submission.

Our example allows to register new User. For this we have defined two handler methods to AddUserFormController to handle each of the operations. We first need to display the form in the browser before it can be submitted. The first method (showForm()) displays the registration form. See the figure above.

Once the form is displayed, it'll need a User object to bind to the form fields. The showForm() method creates a User object and places it in the model.

Demo

- Refer the following Demos:

- DemoMVC_3
- DemoMVC_4
- DemoMVC_5
- DemoMVC_6
- DemoMVC_7
- DemoMVC_Complete



4.3 : Introduction to REST

Introduction to REST web Services

■ ReST : Representational state transfer

- Is an architectural style of designing loosely coupled Web applications that rely on named resources rather than messages.
- Is a lightweight alternative to mechanisms like RPC (Remote Procedure Calls) and Web Services (SOAP)
- In a good REST design operations are self-contained, and each request carries with it (transfers) all the information (state) that the server needs in order to complete it.
- REST leverages aspects of the HTTP protocol to standard business-application needs. So:

| Application task | HTTP command |
|------------------|--------------|
| Create | POST |
| Read | GET |
| Update | PUT |
| delete | DELETE |



Copyright © Capgemini 2015. All Rights Reserved 26

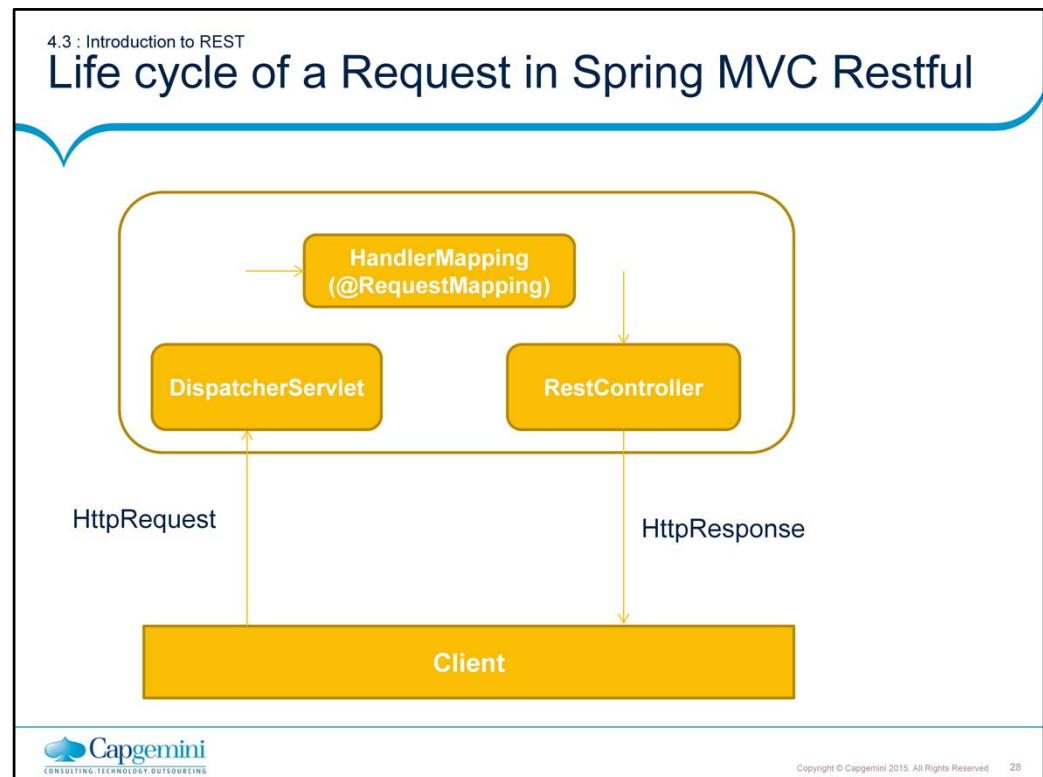
REST is a style of designing loosely coupled Web applications that rely on named resources — in the form of Uniform Resource Identifiers (URIs, for instance — rather than messages. REST leverages aspects of the HTTP protocol such as GET and POST requests.

These requests map quite nicely to standard business-application needs such as create read, update, and delete (CRUD)

4.3 : Introduction to REST

Introduction to REST web Services

- Principles of REST web Services
 - Use HTTP methods explicitly
 - Be stateless
 - Expose directory structure-like URIs
 - Transfer XML, JavaScript Object Notation (JSON), or both



REST (Representational State Transfer) is an architectural style with which Web Services can be designed that serves resources based on the request from client. A Web Service is a unit of managed code, that can be invoked using HTTP requests. You develop the core functionality of your application, deploy it in a server and expose to the network. Once it is exposed, it can be accessed using URI's through HTTP requests from a variety of client applications. Instead of repeating the same functionality in multiple client (web, desktop and mobile) applications, you write it once and access it in all the applications.

In the above diagram, from the time that a request is received by Spring until the time that a response is returned to the client, many pieces of Spring Restful webservice are involved.

The process starts when a client (typically a web browser) sends a request. It is first received by a DispatcherServlet. Like most Java-based MVC frameworks, Spring MVC uses a front-controller servlet (here DispatcherServlet) to intercept requests. This in turn delegates responsibility for a request to other components of an application for actual processing.

The Spring MVC uses a Controller component for handling the request. But a typical application may have several controllers. To determine which controller should handle the request, DispatcherServlet starts by querying one or more HandlerMappings. A HandlerMapping typically maps URL patterns to RestControllers.

Once the DispatcherServlet has an appropriate RestController selected, it dispatches the request to that Controller which performs the business logic (a well-designed RestController object delegates responsibility of business logic to one or more service objects). Upon completion of business logic, HTTPResponse is generated and sent back to the client.

4.4 : Rest Controllers

Why REST Controller ?

- Traditional Spring MVC controller and the RESTful web service controller differs in the way the HTTP response body is created
- Traditional MVC controller relies on the View technology
- RESTful controller simply returns the object and the object data is written directly to the HTTP response as JSON/XML



Copyright © Capgemini 2015. All Rights Reserved 29

The key difference between a traditional Spring MVC controller and the RESTful web service controller is the way the HTTP response body is created. While the traditional MVC controller relies on the View technology, the RESTful controller simply returns the object and the object data is written directly to the HTTP response as JSON/XML.

In Spring 3 `@ResponseBody` was used as an annotation over the Rest Controller methods on the return type indicating the HTTP response as JSON/XML, along with this we had to specify the MIME type as "application/json" or "application/xml".

For example: Spring 3 Rest Controller method:

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

    @RequestMapping(value = "/{name}", method = RequestMethod.GET, produces =
"application/json")
    public @ResponseBody Employee getEmployeeInJSON(@PathVariable String name) {
        employee.setName(name);
        employee.setEmail("employee1@info.com");
        return employee;
    }
}
```

In Spring 4 `@RestController` is a combination of `@Controller` + `@ResponseBody` annotations. Thus in Spring 4 we do not need to use `@ResponseBody`; we directly can use `@RestController` and return view and or the object to the HTTP response.

In our example discussed in next subsequent slides we have kept view (jsp) itself as the `ResponseBody`.

4.4 : Rest Controllers

Spring 4 support for RESTful web services

- In Spring 4 REST is built on the top of MVC
 - REST methods: GET, PUT, DELETE, and POST, can be handled by Controllers
 - Using `@PathVariable` annotation controllers can handle requested parameterized URLs

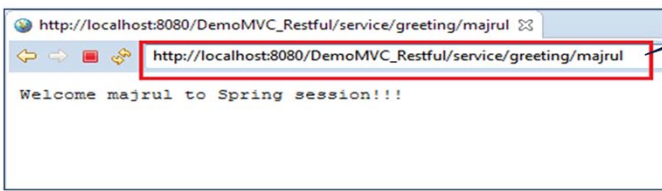
Spring supports different Request methods like GET, PUT, DELETE, POST corresponding to respective HTTP methods.

4.4 : Rest Controllers

MVC (RESTful) controller

```
@RestController
@RequestMapping("/service/greeting")
public class SpringRestController {
    @RequestMapping(value = "/{name}", method = RequestMethod.GET)
    public String sayHello(@PathVariable Optional<String> name) {
        String result = "Welcome " + name + " to Spring session!!!";
        return result;
    }
}
```

No "name" required. "Do not repeat yourself" with Java 8 version



output

Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved 31

Let us see what a RESTful controller looks like.

Restful controller

`@RequestMapping` annotation which says that this controller will handle requests for `/service/greeting`. It implies and supports the fact that this controller is focused on displaying greeting message.

URL (uniform resource locator) is a means of locating a resource. But, since, no two resources can share the same URL, URL could also be a means of uniquely identifying a resource ie URI. Many URLs don't locate or identify anything—they make demands.

Rather than identify something, they demand some action be taken. For example consider below URL:

`http://localhost:8080/DemoMVC_Restful/service/greeting?name=capgemini`, this kind of URL will be handled by the `SpringRestController`'s `sayHello()` method. Here, the URL is not locating or identifying a resource. The base portion of the URL is verb-oriented ie "sayXXX()".

RESTful URLs on the other hand are resource-oriented. So, the URL in the example above should look like the following :

`http://localhost:8080/DemoMVC_Restful/service/greeting/capgemini`

This URL now correctly locates and identifies a resource. Notice, the base portion of the URL is resource-oriented. Also notice that it has no query parameters, instead it has a parameterized path. The RESTful URL's input is part of the URL's path.

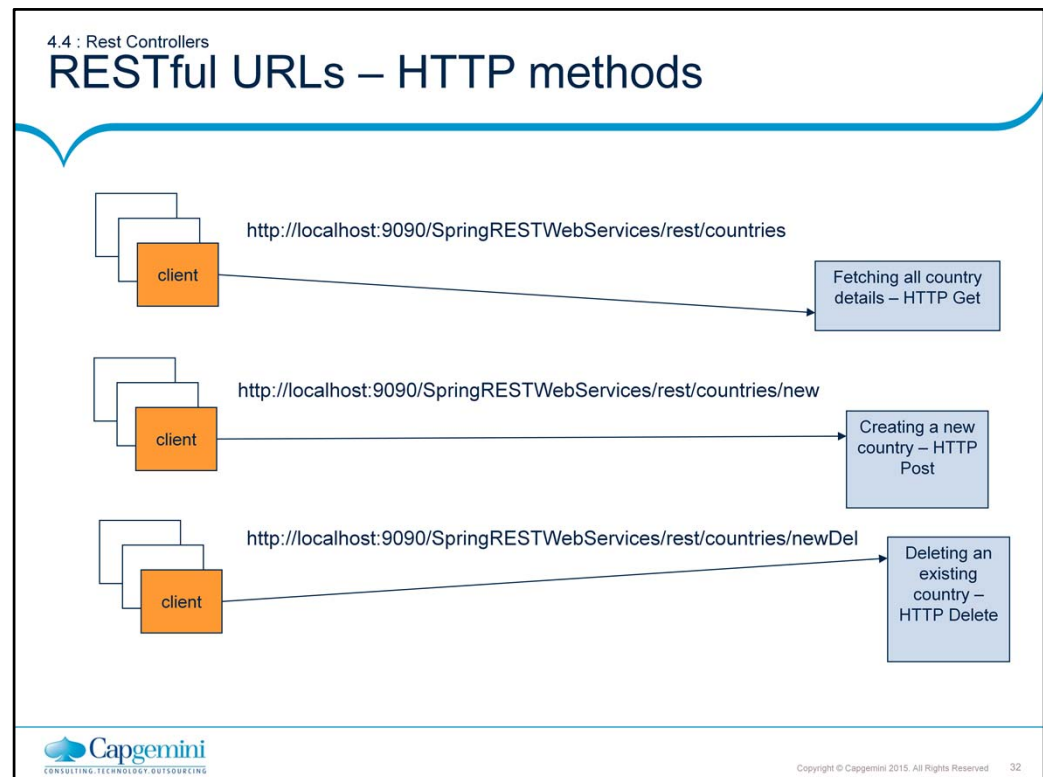
But how will you code the controller to take input from the URL's path?

At the method level, `{name}` in the `@RequestMapping` annotation, is a placeholder through which variable data will be pass into the method.

The `sayHello()` method is designed to handle GET requests for URLs that take the form `/ {name}`. The `@PathVariable` annotation corresponds to this id.

So, if a GET request comes in for `http://localhost:8080/DemoMVC_Restful/service/greeting/capgemini`, the `sayHello()` method will be called with `capgemini` passed in for the name parameter.

The method then uses that value to generate welcome message and place it into the model.



The slide demonstrates different RESTful URLs with respect to different HTTP methods.

Demo: SpringRESTWebServices can be used as a reference.

Note: At times if the HTTP Get is used over a couple of RestController methods it has to be combined with URL patterns to create unique identifications.

In the above slide first URL pattern demonstrates : HTTP Get method to fetch all country details.

Similarly if details for a particular country need to be fetched then the country id can be appended in URL and extracted via the @PathVariable

http://localhost:9090/SpringRESTWebServices/rest/countries/3 -> With this URL country details are fetched for country Id = 3

2. The second URL pattern demonstrates : HTTP Post method to create a new country

3. The third URL pattern demonstrates : HTTP Delete method to delete an existing country

Note: As HTML supports only Get and Post methods for the method attribute in the form tag; we also need to map the HTTP PUT(update) and HTTP DELETE (delete) methods to update and delete the resources respectively.

For this Spring provides us with a Filter-mapping which is to be given in web.xml file:

```

<filter>
  <filter-name>httpMethodFilter</filter-name>
  <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>httpMethodFilter</filter-name>
  <servlet-name>dispatcher</servlet-name>
</filter-mapping>
  
```

By using this Spring in-built filter the different methods of HTTP specification will be mapped to their actual HTTP implementations.

Here the filter will be intercepted for all the requests coming to DispatcherServlet.

Also in the JSP pages for updating and deleting a country need to pass a HTML hidden parameter : For example

<input type="hidden" name="_method" value="delete"/> to pass the "real" HTTP method to Spring Rest Controller.

4.4 : Rest Controllers

Demo: DemoMVC_Restful

- DemoMVC_Restful
- SpringRESTWebServices



Summary

- We have so far seen:
 - How to use Spring MVC architecture to build flexible and powerful web applications.
 - Components like handler mappings, ViewResolvers and controllers
 - MVC Annotations like @Controller, @RestController, @RequestMapping , @RequestParam, @PathVariable



Review Questions

- Question 1: If multiple handler mappings have been declared in an application, select the property that indicates which handler mapping has precedence?
 - Option 1: Order
 - Option 2: Sequence
 - Option 3: Index
 - Option 4: An application cant have multiple handler mappings
- Question 2: To figure out which controller should handle the request, DispatcherServlet queries
 - Option 1: HandlerMappings
 - Option 2: ModelAndView
 - Option 3: ViewResolver
 - Option 4: HomeController

