Controllers

Cognizant
Passion for building stronger businesses

About the Author

Created By:	Ramesh C.P.(161646)
Credential Information:	SCJP, 8+ years of experience in technical training
Version and Date:	SpringMVC/PPT/1011/3.0

Cognizant Certified Official Curriculum

Icons Used

Questions

Tools

Hands on Exercise

Coding Standards

Test Your Understanding

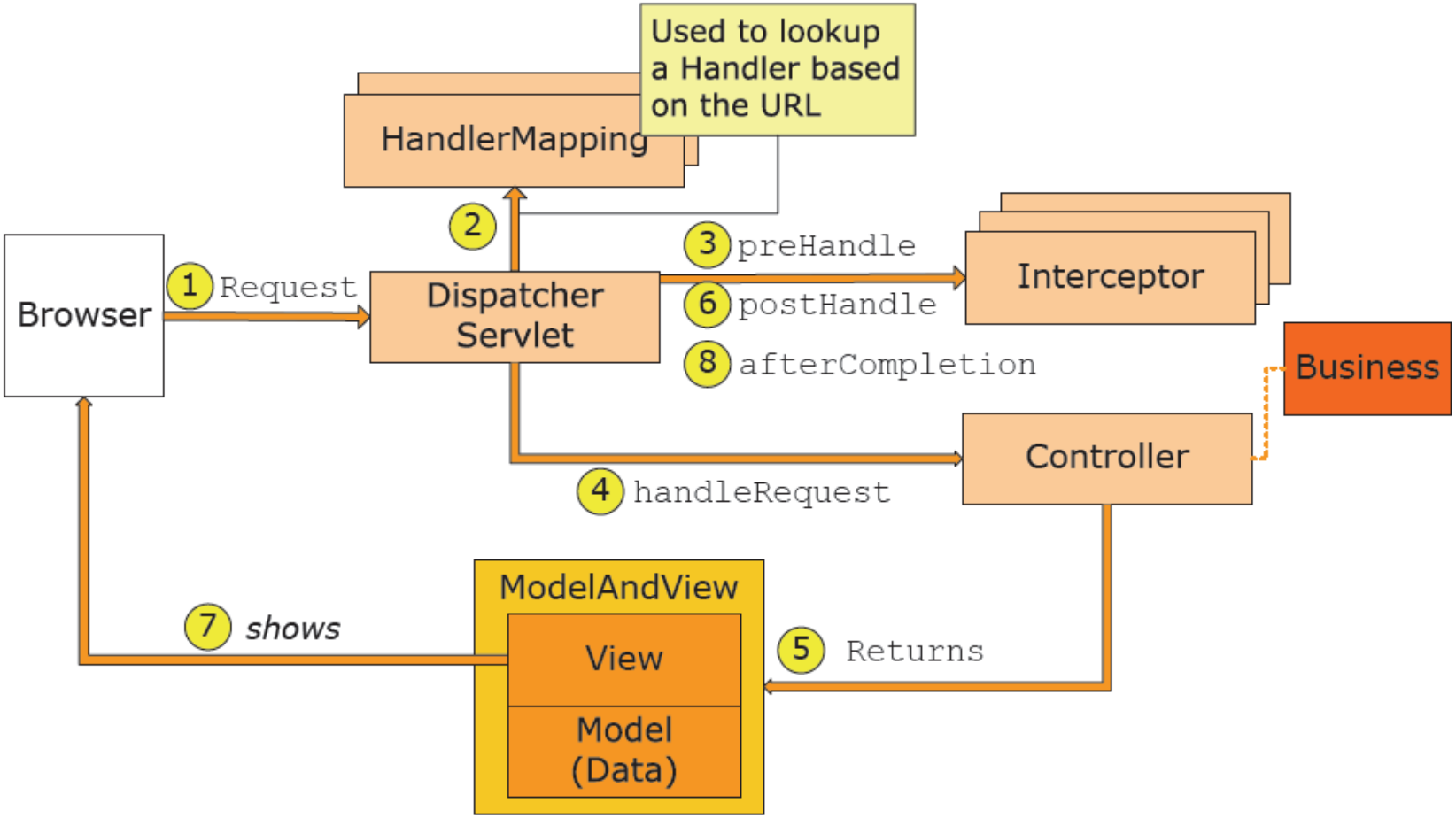
Reference

Demonstration

A Welcome Break

Contacts

Review of Architecture



Controllers: Overview

- Introduction:
  - Controllers provide access to the application behavior which is typically defined by a service interface.
  - Controllers interpret user input and transform such input into a sensible model which will be represented to the user by the view.
  - Spring has implemented the notion of a controller in a very abstract way enabling a wide variety of different kinds of controllers to be created.
  - Since 2.5.x release, Spring has introduced an annotation-based programming model for MVC controllers, @Controller, that uses annotations such as @RequestMapping, @RequestParam and @ModelAttribute.

Controllers

- A Controller processes the request
  - Is an implementation of the `Controller` Interface
  - Many Controller implementations exist
  - Can write into the response and return null, but this is not their primary use case

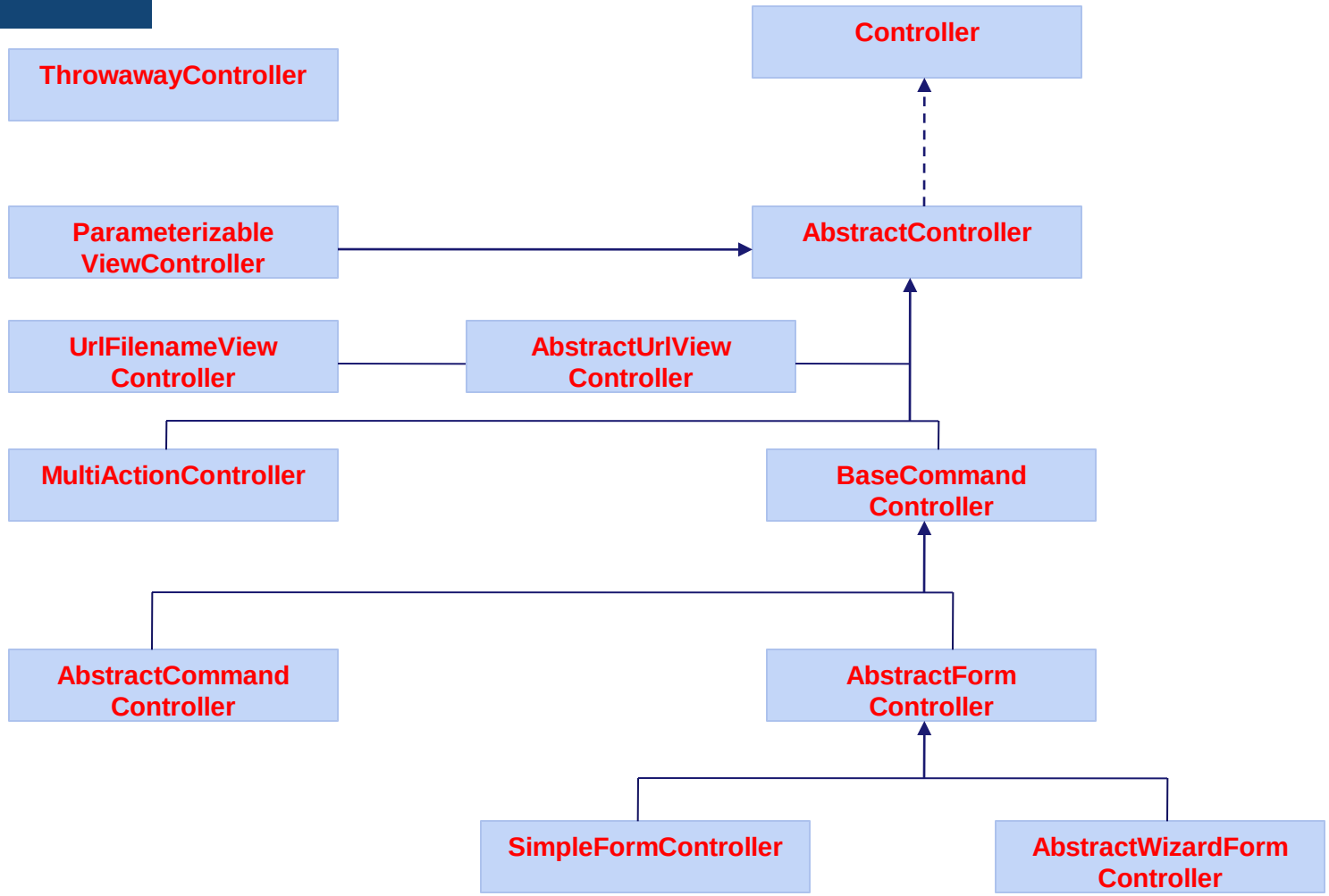
Controllers: Objectives

- Objective:

After completing this chapter you will be able to:

- Understand Spring Controller hierarchy
- Write different type of Controller using Annotation model
- Learn different annotation elements with examples
- Write exception handling and testing with Controller

Spring MVCs Controller Hierarchy



Existing Controller Implementations

Implementation	Description
`WebContentGenerator`	Convenient base class (is also base class of `WebContentInterceptor`)
`AbstractController`	Performs various checks and can synchronize the `handleRequest` using Session attribute as mutex
`UrlFilenameViewController`	Returns a view based on the last part of the request URL (optionally adding a prefix or postfix)
`ParameterizableViewCont…`	Returns a named view (often used to place JSP, etc., inside the WEB-INF)
`MultiActionController`	Uses a command pattern, each request is mapped to a method using a `MethodNameResolver`
`(AbstractCommandCont…)` `AbstractFormController`	Populates a bean from the request Subclasses: AbstractWizardFormController, SimpleFormController

How to Select a Controller

Controller Type	Classes	Purpose
View	ParameterizableViewController UrlFileNameViewController	Your controller only needs to display a static view – no processing or data retrieval is needed
Simple	Controller (Interface) AbstractController	Your controller is extremely simple, requiring little more functionality.
Throwaway	ThrowawayController	You want a simple way to handle requests as commands.

How to Select a Controller (Contd.)

Controller Type	Classes	Purpose
MultiAction	MultiActionController	Your application has several actions that perform similar or related logic
Command	BaseCommandController AbstractCommandController	Your controller will accept one or more parameters from the request and bind them to an object. Also capable of performing parameter validation
Form	AbstractFormController SimpleFormController	You need to display an entry form to the user and also process the data entered into the form

How to Select a Controller (Contd.)

Controller Type	Classes	Purpose
Wizard	AbstractWizardFormController	You want to walk your user through a complex, multipage entry form that ultimately gets processed as a single form

Controller Interface

- It represents a component that receives HttpServletRequest and HttpServletResponse.
- Method:

  ModelAndView handleRequest(HttpServletRequest, HttpServletResponse)
- All Spring MVC controllers either implement Controller directly or extend from one of the available base class implementations such as AbstractController, SimpleFormController, MultiActionController, or AbstractWizardFormController or use @Controller annotation.
- Spring 3 favors annotated controllers and annotation programming model instead of using Spring API.

Controller annotation

- The @Controller annotation indicates that a particular class serves the role of a *controller*. There is no need to extend any controller base class or reference the Servlet API.
- The basic purpose of the @Controller annotation is to act as a stereotype for the annotated class, indicating its role. The dispatcher will scan such annotated classes for mapped methods, detecting @RequestMapping annotations.
- The annotation based programming model allows a high degree of flexibility in the method's signature as it does not have any interface or base class requirements.

Setting Up for Annotations

- web.xml:
  - Configure the ContextLoaderListener (for your services)
  - Configure the DispatcherServlet (for the Spring MVC functionality)
- Spring MVC config file (<servlet-mapping>-servlet.xml)
  - Add your view resolver(s)
  - Add controller mappings (or use annotations)
  - Optionally define any marshallers

Annotation-based controller configuration

- You only need to add a single line of configuration to spring configuration xml file to flip on all of the annotation-driven features you'll need from Spring MVC.

  <mvc:annotation-driven/>

- To configure Spring for autodiscovery, use <context:component-scan>. The <context:component-scan> element works by scanning a package and all of its subpackages, looking for classes that could be automatically registered as beans in the Spring container. The base-package attribute tells <context:component-scan> the package to start its scan from.

  <context:component-scan base-package="com.springinaction.springidol">
  </context:component-scan>

- Annotation-driven and autodiscovery scan can dramatically reduce the amount of XML Spring configuration. You'll need only a handful of lines (as above) of XML, regardless of how many beans are in your Spring application context.

Spring XML Configuration Example

```
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="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" >

<mvc:annotation-driven/>
<context:component-scan base-package="com.springinaction.springidol">
    </context:component-scan>

</beans>
```

Spring MVC Annotations

- Spring 3 provides annotation-based configuration for controllers
  - @RequestParam
  - @PathVariable
  - @ResponseBody
  - @RequestBody
  - @Controller
  - @RequestMapping
  - @Valid
- To use annotations, the XML must be told to look for annotation-based controllers

ClassPath Scanning

- Controllers can be declared using the @Controller annotation
  - In the spring-servlet.xml file, inform Spring where to search for controllers
  - Use a component-scan element in your spring-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="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:component-scan base-package="org.example"/>

</beans>
```

Annotation Controller Mapping

- Use @RequestMapping to define which controllers and methods will be invoked
  - Used at both the class and method level
  - This technique "frees" the restrictions placed on controller methods earlier
  - Controller methods invoked no longer follow strict signature formats:

```
protected ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
```

How @RequestMapping Works

- When placed at the method level, it is used to further refine the URL mapping:

```
@Controller @RequestMapping(value="/customer")
public class CustomerController {
@RequestMapping(method=RequestMethod.GET) protected ModelAndView
   findEmployee(HttpServletRequest
req, HttpServletResponse resp) throws Exception {
...
}

GET http://localhost:8080/SpringWeb/mvcapp/customer
```

How @RequestMapping Works

- When placed at the class level, it is used to select which controller will be invoked

```
@Controller @RequestMapping(value="/customer")
public class CustomerController {
...
}
http://localhost:8080/SpringWeb/customer/createRecord
```

Simplest Possible Controller

```
@Controller
Public class HomeController {
    @RequestMapping("/home")
    public String showHomePage() {
        return "home";
    }
}
```

Declare as Controller

Handle requests for "/home"

Return view name "home"

Simplest Possible Controller (Contd.)

- First, the @Controller annotation indicates that the HomeController is a controller class.
- @RequestMapping annotation serves two purposes. First, it identifies showHomePage() as a request-handling method. And, more specifically, it specifies that this method should handle requests whose path is /home.
- The last thing that showHomePage() does is return a String value that's the logical name of the view that should render the results.
- What's most remarkable about HomeController (and most Spring MVC controllers) is that there's little that's Spring-specific about it. This would be a POJO if we remove the annotations.

Using MultiActionController

- Typical base implementation when not working with Forms
  - A request is mapped to a method

```
public xxxYyy productInfo(HttpServletRequest request,
                HttpServletResponse response) throws Exception{…
```

  - Request-to-method name mapping is performed by a MethodNameResolver (Strategy GoF design pattern)

MethodNameResolver	Description
`ParameterMethodNameResolver`	Method name is obtained from a request parameter (defaults to `action`)
`InternalPathMethodNameResolver` (default)	Method name is obtained from the last part of the request URL
`PropertiesMethodNameResolver`	Uses a properties file to map URL patterns to method names

Using MultiActionController (Contd.)

- Example using MultiActionController
  - Mapping all *.do to our MultiActionController

```
<bean id="handlerMapping"
      class="....SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="/catalog/*.do">catalogController</prop>
    </props>
  </property>
</bean>
<bean id="catalogController" class="demos.CatalogController"/>
```

Using MultiActionController (Contd.)

- The CatalogController implementation

```
public class CatalogController extends MultiActionController{
   ...
   public ModelAndView productInfo(HttpServletRequest request,
                HttpServletResponse response) throws Exception {
   String dvdID = request.getParameter("dvdID");
   DVD dvd = manager.getProductInfo(dvdID);
   return new ModelAndView("dvd-details", "dvd", dvd);
}
public ModelAndView list(HttpServletRequest request,
                HttpServletResponse response) throws Exception {
   Collection<DVD> dvds = manager.getAll();
   return new ModelAndView("dvd-list", "dvds", dvds);
}}
```

  - Request to catalog/list.do is dispatched to `list` method, catalog/productInfo.do is dispatched to `productInfo`

Form Controllers

- Use a `SimpleFormController` when working with a Form
- Use an `AbstractWizardFormController` when working with multiple forms
- Basic Controller configuration:

Property (`WebContentGenerator`)	Description
`CacheSeconds`	Number of seconds that content is cached
`SupportedMethods`	Supported HTTP methods
`RequireSession`	Specifies whether a session is required
`UseCacheControlHeader`	Specifies whether to use the HTTP 1.1 `cache-control` header
`UseExpiresHeader`	Specifies whether to use the HTTP 1.0 `expires` header

Handling Form Requests

- Spring MVC offers convenient functionality when working with Forms
  - *Data binding* of Form data to beans
  - *Validation* for presentation-side validation
  - Support for *Property Editors* to handle complex data types such as dates
- Several types of implementations are typically used:
  - Use Annotations to define form behavior, and how to bind parameters to objects
  - Use a `SimpleFormController` when working with a single Form (SimpleFormController is deprecated in Spring3)
  - Use an `AbstractWizardFormController` when working with multiple Forms (wizard style)
  - This lesson will use Annotations

Handling Controller inputs

```
@Controller
@RequestMapping("/appointments/")
public class AppointmentsController {
    @RequestMapping(value="query", method=GET)
    public String queryAppointment(@RequestParam("appointmentId") String
    appointmentId, Model model)
    {
        /* logic to get specific appointment details based on appointment id
            and saving appointment object in Model */
      model.addAttribute(appointment);
        return "appointmentDetails";
    }
}
```

http://localhost:8080/appointments/query?appointmentId=12345 could be the URL for displaying appointment details for appointment id – 12345.

Handling Controller inputs (Contd.)

- The queryAppointment() method takes a appointmentId and a Model object as parameters.
- The appointmentId parameter is annotated with @RequestParam("appointmentId") to indicate that it should be given the value of the appointment query parameter in the request.
- @RequestParam is useful for binding query parameters to method parameters where the names don't match. In this example, we really don't have to use RequestParam annotation. However it is the standard practice to always use @RequestParam .

Handling Controller inputs (Contd.)

- The queryAppointment() method takes a Model object as second parameter.
- The Model is just like a Map<String, Object> under the covers. It provides some convenient methods for populating the model, such as addAttribute(). The addAttribute() method does pretty much the same thing as Map's put() method, except that it figures out the key portion of the map on its own.
- The sample code skips the logic for getting appointment details based on appointment Id. It is just pure POJO logic. The last thing that queryAppointment() does is return a String value that's the logical name of the view that should render the results.

Controller – Processing Form

```
@Controller
@RequestMapping(“/appointments/”)
public class AppointmentsController {
@RequestMapping(method=RequestMethod.POST)
    public String addAppointment(@Valid Appointment appointment,
    BindingResult bindingResult) {
        if(bindingResult.hasErrors()) {
            return “edit";
        }
        //logic for saving appointment in database
        return "redirect:success”;
    }
}
```

Check for errors

Controller – Processing Form (Contd.)

- addAppointment() handles POST requests. When UI Form is submitted, the fields in the request will be bound to the Appointment object.
- The Appointment parameter is annotated with @Valid. This indicates that the Appointment should pass validation before being processed in.
- Should anything go wrong while validating the Appointment object, the validation error will be carried to the addAppointment() 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. In that case, the method will return edit as the view name to display the form again so that the user can correct any validation errors.

Controller with Fileupload

- Spring's built-in multipart support handles file uploads in web applications. You enable this multipart support with pluggable MultipartResolver objects, defined in the org.springframework.web.multipart package.
- The following example shows how to use the CommonsMultipartResolver

```
<bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- one of the properties available; the maximum file size in bytes -->
    <property name="maxUploadSize" value="100000"/>
</bean>
```

Controller with Fileupload (Contd.)

```
@Controller public class FileUpoadController {
    @RequestMapping(value = "/form", method = RequestMethod.POST)
    public String handleFormUpload(@RequestParam("name") String name,
    @RequestParam("file") MultipartFile file)  {
            if (!file.isEmpty()) {
                // store the file somewhere
                return "redirect:uploadSuccess";
            }
            else {
                return "redirect:uploadFailure";
            }
    }
}
```

Similar Controller except MultipartFile as method parameter

@RequestBody Annotation

- The @RequestBody method parameter annotation indicates that a method parameter should be bound to the value of the HTTP request body.
- To receive that message as a Appointment object, we only need to annotate a handler method's Appointment parameter with @RequestBody

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
public void addAppointment(@RequestBody Appointment appointment,
    Writer writer)
Throws Exception {
     //save the appointment in database;
}
```

@ResponseBody Annotation

- The @ResponseBody annotation is similar to @RequestBody.
- This annotation can be put on a method and indicates that the return type should be written straight to the HTTP response body (and not placed in a Model, or interpreted as a view name).
- The above example will result in the text Hello World being written to the HTTP response stream.

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
@ResponseBody
public String helloWorld() {
    return "Hello World";
}
```

@ModelAttribute Annotation as Input

- @ModelAttribute has two usage scenarios in controllers.
- When you place it on a method parameter, @ModelAttribute maps a model attribute to the specific, annotated method parameter (see the example below).
- This is how the controller gets a reference to the object holding the data entered in the form.

```
@Controller
@RequestMapping("/appointments/")
public class AppointmentsController {
    @RequestMapping(value="query", method=GET)
    public String cancelAppointment(@ModelAttribute("appointment") Appointment
    appointment, BindingResult result)  {
        //code for error check and canceling appointment
     }
}
```

@ModelAttribute Annotation as Output

- You can also use @ModelAttribute at the method level to provide *reference data* for the model (see the populateAppointmentTypes() method in the following example).

```
@Controller
@RequestMapping("/appointments/")
public class AppointmentsController {
    @ModelAttribute("appointmentTypes")
    public Collection<AppointmentType> populateAppointmentTypes()  {
        return this.appointment.appointmentTypes();
     }
}
```

@SessionAttributes Annotation

- The type-level @SessionAttributes annotation declares session attributes used by a specific handler.
- This will typically list the names of model attributes or types of model attributes which should be transparently stored in the session or some conversational storage, serving as form-backing beans between subsequent requests.

```
@Controller
@RequestMapping("/editAppointment")
@SessionAttributes(“appointmentId")
public class EditAppointmentForm {
// ...
}
```

@CookieValue Annotation

- The @CookieValue annotation allows a method parameter to be bound to the value of an HTTP cookie.
- Let us consider that the following cookie has been received with an http request.

JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84

- The following code sample demonstrates how to get the value of the JSESSIONID cookie:

```
@RequestMapping("/displayHeaderInfo.do")
public void displayHeaderInfo(@CookieValue("JSESSIONID")
    String cookie) {
    //...
}
```

@RequestHeader Annotation

- The @RequestHeader annotation allows a method parameter to be bound to a request header.
- The following code sample demonstrates how to get the value of the Accept-Encoding and Keep-Alive headers:

```
@RequestMapping("/displayHeaderInfo.do")
public void displayHeaderInfo(
    @RequestHeader("Accept-Encoding")  String encoding,
    @RequestHeader("Keep-Alive") long keepAlive)  {
    //...
}
```

Handling Exceptions

- Spring HandlerExceptionResolvers exception and @ExceptionHandler annotation ease the pain of unexpected exceptions that occur while your request is handled by a controller that matched the request.
- You use the @ExceptionHandler method annotation within a controller to specify which method is invoked when an exception of a specific type is thrown during the execution of controller methods.
- By default, the DispatcherServlet registers the DefaultHandlerExceptionResolver. This resolver handles certain standard Spring MVC exceptions by setting a specific response status code:

Handling Exceptions Example

```
@Controller
public class SimpleController {
// other controller method omitted @ExceptionHandler(IOException.class)
    public String handleIOException(IOException ex, HttpServletRequest
    request) {
    return ClassUtils.getShortName(ex.getClass());
    }
}
```

- The above sample code will invoke the 'handlerIOException' method when a java.io.IOException is thrown.
- The @ExceptionHandler value can be set to an array of Exception types. If an exception is thrown matches one of the types in the list, then the method annotated with the matching @ExceptionHandler will be invoked.

Testing Controllers

- Controllers are just POJOs - just new them up and test them! From a unit testing perspective, this is significant because it means that your Controller can be tested easily without having to mock anything or create any Spring-specific objects.
- Tests for Controllers can be written just as easily as service layer tests and can be run just as fast.
- Inject business mock dependencies using your favorite mocking library (Mockito or EasyMock)
- Use HttpServletMocks, if you need for your Servlet API dependency in your @Controller.

Testing Controllers (Contd.)

- Testing Controllers
  - Controllers are based on standard Java EE classes (e.g. HttpServletRequest)
  - These can be mocked very easily (Mock implementations are bundled with Spring
  - For example, for form unit testing you populate the (mocked) request with the form data you want to test and invoke `handleRequest`

Spring Mock

- The Spring framework provide a *spring-mock.jar* that includes classes for writing both unit tests and integration tests.
- Some mock classes in the jar are:
  - MockHttpServletRequest
  - MockHttpServletResponse
  - MockHttpSession
  - MockServletConfig
  - MockServletContext
  - MockPageContext
  - MockRequestDispatcher

Simple Controller for Unit Testing

```
@Controller
@RequestMapping("/appointments/")
public class AppointmentsController {
    private AppointmentComponent appointmentComponent;
    @RequestMapping(value="query", method=GET)
    public String queryAppointment(@RequestParam("appointmentId") String
    appointmentId, Model model)  {
        Appointment appointment =
            appointmentComponent.getAppointment(appointmentId);
        model.addAttribute(appointment);
        return "appointmentDetails";
    }
}
```

Controller Test Case

```
public class AppointmentsControllerTest {
@Test
public void shouldReturnValidAppointment() {
    //Setup mock object using EasyMock
    Appointment appointment = new Appointment("Name", "Type", "Date", "Comments");
    AppointmentComponent apptComponent = createMock(AppointmentController.class);
    apptComponent.getComponent(isA(String.class));
    expectLastCall().andReturn(appointment);

    HashMap<String, Object> model = new HashMap<String, Object>();
    AppointmentsController apptController = new AppointmentsController(apptComponent);
    String viewName = apptController. queryAppointment(appointmentId, model);

    assertEquals("appointmentDetails", viewName);
    assertSame(appointment, model.get("appointment"));
    verify(spitterService).getComponent(appointmentId);
    }
```

Setup Mocks using EasyMock

Create Controller and call handler method

Assert results

Controllers

Time for a Break !

Controllers

- Questions from participants

Test Your Understanding

1) Identify from the following which is not Spring annotation:
   a) @Controller
   b) @RequestBody
   c) @EasyTest
   d) @ExceptionHandler
2) Identify the annotation from the following which is used to get HTTP header information such as Accept-Encoding and Keep-Alive headers:
   a) @CookieId
   b) @ModelAttribute
   c) @RequestHeader

Controllers: Summary

- With annotation driven development, there is no need to extend any controller base class or reference the Servlet API.
- Annotation-driven and autodiscovery scan reduce the amount of XML Spring configuration.
- DispatcherServlet requests are mapped to @Controller methods and @RequestMapping is used to define mapping rules.
- Annotate a separate method in your @Controller as a @ExceptionHandler for exception handling.
- Controllers are just POJOs - just new them up and test them

Controllers: Source

- http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/mvc.html#mvc-controller
- http://static.springsource.org/spring/docs/3.0.x/javadoc-api/

You have successfully completed Controllers

Click here to proceed

Cognizant
Passion for building stronger businesses