Spring MVC

CS544: Enterprise Architecture

Spring MVC

- In this module we will look at the theory of how Spring MVC works, show a Spring MVC example. Spring MVC focuses on us having to do less, and accomplish more, by providing an extra layer of intelligence in between the technology and us.
- We will look at:
  - Application Context in a Web Container
  - Request Mapping
  - URI Templates
  - Data Input / Data Output
  - Session & Flash Attributes
  - Exception Handling

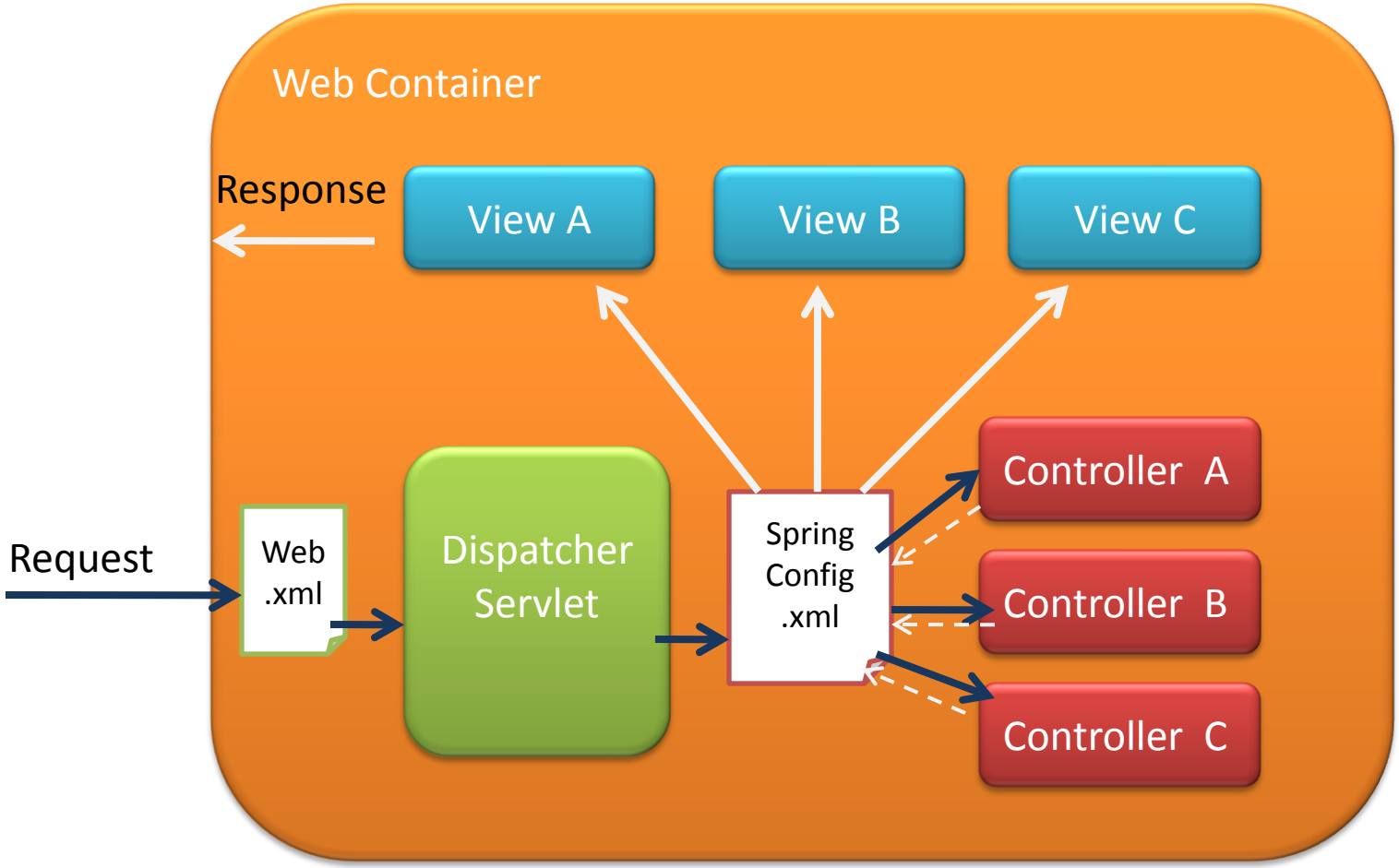
Spring MVC

- Spring MVC is a web development Framework that closely integrates with the rest of Spring (DI, AOP).
  - The jars for Spring MVC (web) are even included in the Core Spring download.

- Spring MVC is a **request based** web development framework that uses the **front-controller** pattern
  - Requests are first processed by the DispatcherServlet
  - After which they are mapped onto a handler method written by the programmer.

- Spring MVC is built on top of the Servlet API
  - All features of the Servlet API remain available

Front Controller / Dispatcher Servlet



Spring MVC:

BASIC EXAMPLE

Spring MVC Example – Web.xml

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>Example09.1</display-name>

  <servlet>
    <servlet-name>SpringMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/springconfig.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>SpringMVC</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file></welcome-file>
  </welcome-file-list>

</web-app>
```

Spring MVC – springconfig.xml

springconfig.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"
xmlns:mvc="http://www.springframework.org/schema/mvc"
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
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

  <!- scan for @RequestMapping annotations-->
  <mvc:annotation-driven />

  <!- scan for @Controller (and other component) annotations in the following package -->
  <context:component-scan base-package="springmvc.helloworld" />

  <!-- Resolves views to .jsp resources in the /WEB-INF/views directory -->
  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
  </bean>

</beans>
```

Spring MVC Basics – Controller

HelloWorld.java

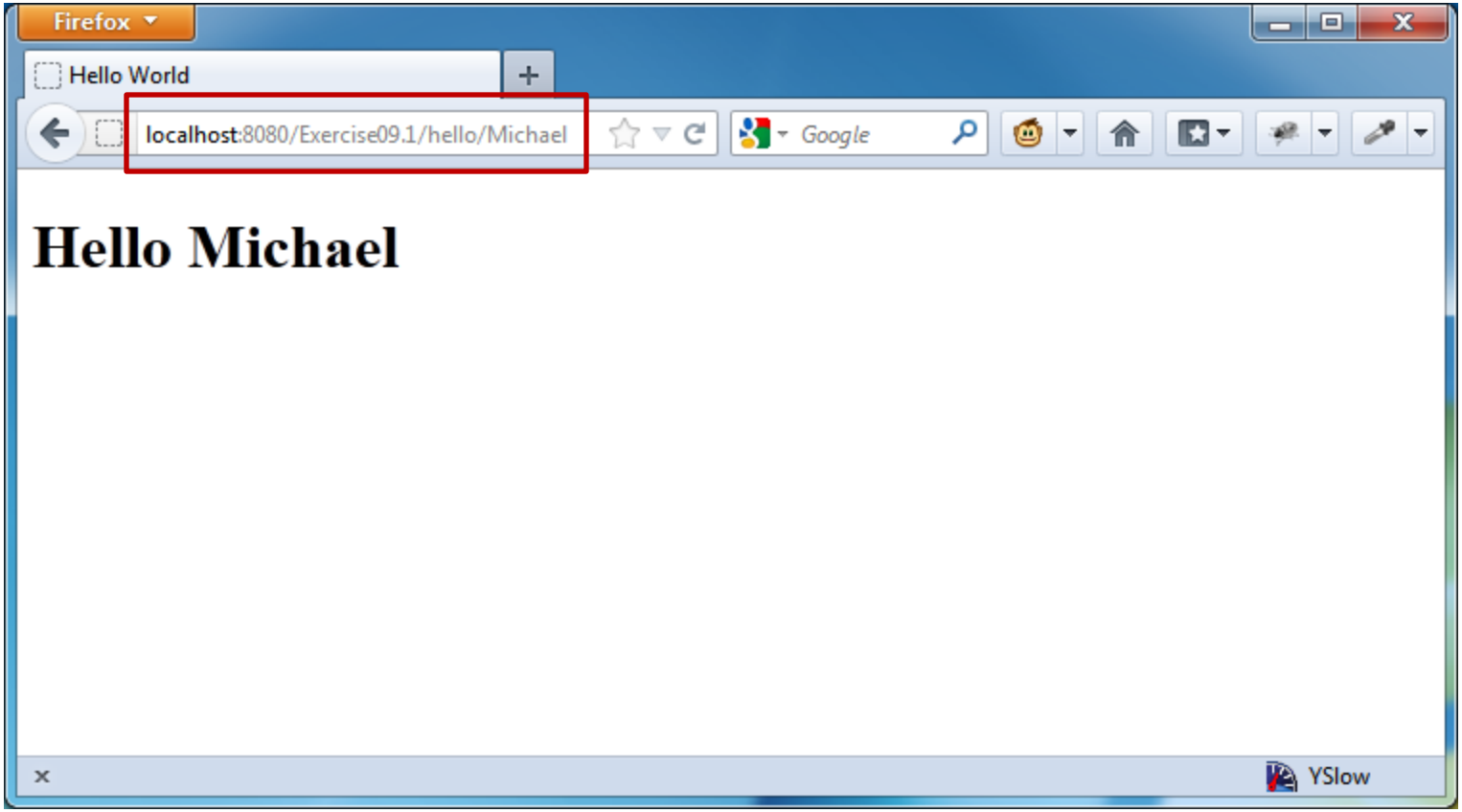
```
package springmvc.helloworld;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HelloWorld {

  @RequestMapping("/hello/{name}")
  public String Hello(@PathVariable String name, Model model) {
    model.addAttribute("name", name);
    return "helloView";
  }
}
```

Spring MVC Basics – View

helloView.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml11.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Hello World</title>
</head>
<body>
    <h1>Hello ${name}</h1>
</body>
</html>
```

Spring MVC Basics – Output



Spring MVC:

APPLICATION CONTEXT

Application Context

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>

  <!-- The definition of the Root Spring Container shared by all Servlets and Filters -->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
  </context-param>

  <! -- Creates/Starts the Root Spring Container shared by all Servlets and Filters -->
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

  <!-- Creates the dispatcher servlet and its configuration -->
  <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/dispatcher-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping> <!-- Maps the dispatcher servlet to all requests in this project -->
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

Optional Root Applcation Context

Optional, defaults to: [servlet-name]-servlet.xml

Minimal web.xml

web.xml

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

  <servlet>
    <servlet-name>min-example</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>min-example</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file></welcome-file>
  </welcome-file-list>

</web-app>
```

No root context, or dispatcher context specified

Will look for default config: /WEB-INF/min-example-servlet.xml

Spring MVC:

REQUEST MAPPING

Request Mapping by Path

```
@Controller
public class CarController {

  ...

  @RequestMapping(value="/cars")
  public String getAll(Model model) {
    model.addAttribute("cars", carDao.getAll());
    return "carList";
  }

  ...
}
```

All requests for the path "/cars" will be mapped onto this method

Request Mapping by HTTP Method

```
@Controller
public class CarController {

  ...

  @RequestMapping(value="/cars", method=RequestMethod.GET)
  public String getAll(Model model) {
    model.addAttribute("cars", carDao.getAll());
    return "carList";
  }

  @RequestMapping(value="/cars", method=RequestMethod.POST)
  public String add(Car car) {
    carDao.add(car);
    return "redirect:/cars";
  }

  ...
}
```

Requests for "/cars" using a GET will be mapped to the **getAll()** method

Requests for "/cars" using a POST will be mapped to the **add()** method

Class Level Path Mapping

```
@Controller
public class CarController {

  @RequestMapping(value="/cars/{id}", method=RequestMethod.GET)
  public String get(@PathVariable int id, Model model) {
    model.addAttribute("car", carDao.get(id));
    return "carDetail";
  }

  @RequestMapping(value="/cars/{id}", method=RequestMethod.POST)
  public String update(Car car, @PathVariable int id) {
    carDao.update(id, car);
    return "redirect:/cars";
  }
}
```

Exactly the same as

```
@Controller
@RequestMapping(value="/cars")
public class CarController {

  @RequestMapping(value="/{id}", method=RequestMethod.GET)
  public String get(@PathVariable int id, Model model) {
    model.addAttribute("car", carDao.get(id));
    return "carDetail";
  }

  @RequestMapping(value="/{id}", method=RequestMethod.POST)
  public String update(Car car, @PathVariable int id) {
    carDao.update(id, car);
    return "redirect:/cars";
  }
}
```

Produces & Consumes

Web Services

```
@Controller
public class WebServiceController {

  ...

  @RequestMapping(value="/list", method=RequestMethod.GET, produces="text/xml" )
  public ModelAndView list() {
    ModelAndView mav = new ModelAndView();
    mav.setViewName("marshalview");

    mav.addObject("list", shoppingListService.getList());
    return mav;
  }

  @RequestMapping(value = "/list", method = RequestMethod.POST, consumes="text/xml")
  public RedirectView addItem(@RequestBody Item item) {
    shoppingListService.addToList(item);
    return new RedirectView("list");
  }

  ...
}
```

If the client is expecting text/xml content (indicated by Accepts header)

If the client is passing in text/xml content (indicated by Content-Type header)

Parameters and Headers

```
@Controller
public class CarController {

  ...

  @RequestMapping(value="/cars", params="myParam=myValue")
  public String getAll(Model model) {
    model.addAttribute("cars", carDao.getAll());
    return "carList";
  }

 @RequestMapping(value="/cars", headers="myHeader=myValue")
  public String getAll(Model model) {
    model.addAttribute("cars", carDao.getAll());
    return "carList";
  }
  ...
}
```

params="myParam" or params="!myParam" also possible

Only requests for "/cars?myParam=myvalue" will be mapped here

Only Requests that have an http header: myHeader: myValue Will be mapped here

Mapping to non-Controllers

Static content (html / css / js)
Or a view without controller

springconfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

  ...

  <!-- Maps '/' requests to the 'home' view -->
  <mvc:view-controller path="/" view-name="home"/>

  <!-- Handles HTTP GET requests for /resources/** by efficiently serving
       up static resources in the ${webappRoot}/resources/ directory -->
  <mvc:resources mapping="/resources/**" location="/resources/" />

  <!-- Lets us find resources (static and dynamic) through the web.xml -->
  <mvc:default-servlet-handler/>


  ...

</beans>
```

AOP & Request Mapping

- By default spring use subclass proxies (CGLIB)
  - No interface is required
- When using interface proxies:
  - Place @RequestMapping on interface
  - The proxy will need to have mappings as well

E.g. when using @Transactional in your controller

Spring MVC:

URI TEMPLATES

URI Templates

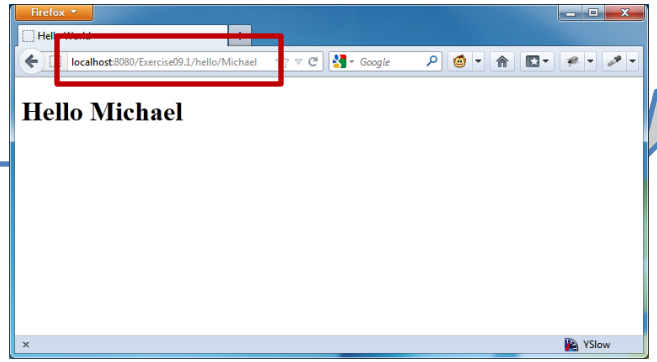
- A URI Template is a URI-like string, containing one or more variable names. When you substitute values for these variables, the template becomes a URI.*

HelloWorld.java

```
@Controller
public class HelloWorld {

  @RequestMapping("/hello/{name}")
  public String Hello(@PathVariable String name, Model model) {
    model.addAttribute("name", name);
    return "helloView";
  }
}
```

@PathVariable can be used to bind the variable to an input parameters

Parameter and Variable name have to match



*spring 3.1 documentation, chapter 16 Web MVC

Multiple Variables

```
@Controller
public class CustomerController {

  @RequestMapping(value="/customer/{customerId}/order/{orderId}")
  public String getOrder(@PathVariable long customerId,
                         @PathVariable long orderId, Model model) {
    // implementation ...
  }
}
```

Either directly on the method level, or combined from class and method

```
@Controller
@RequestMapping(value="/customer/{customerId}")
public class CustomerController {

  @RequestMapping(value="/order/{orderId}")
  public String getOrder(@PathVariable long customerId,
                         @PathVariable long orderId, Model model) {
    // implementation ...
  }
}
```

Regex and Path Patterns

```
// Regular Expression Matching
@RequestMapping(value="/email/{user:\w+}@{host:\w+}.{tld:\w+}")
public void getInfo(@PathVariable String user,
            @PathVariable String host, @PathVariable String tld) {
  // implementation ...
}


// Ant-Style path patterns
@RequestMapping(value="/customer/*/order/{orderId}")
public void getOrder(@PathVariable long orderId, Model model) {
  // implementation ...
}
```

Spring MVC:

DATA INPUT

Request Input

We've seen how path variables can be used for input

GET /cars/1

```
@RequestMapping(value="/cars/{id}", method=RequestMethod.GET)
public String get(@PathVariable int id, Model model) {
  model.addAttribute("car", carDao.get(id));
  return "carDetail";
}
```

But the same can of course be done with normal request parameters

params specification is optional!

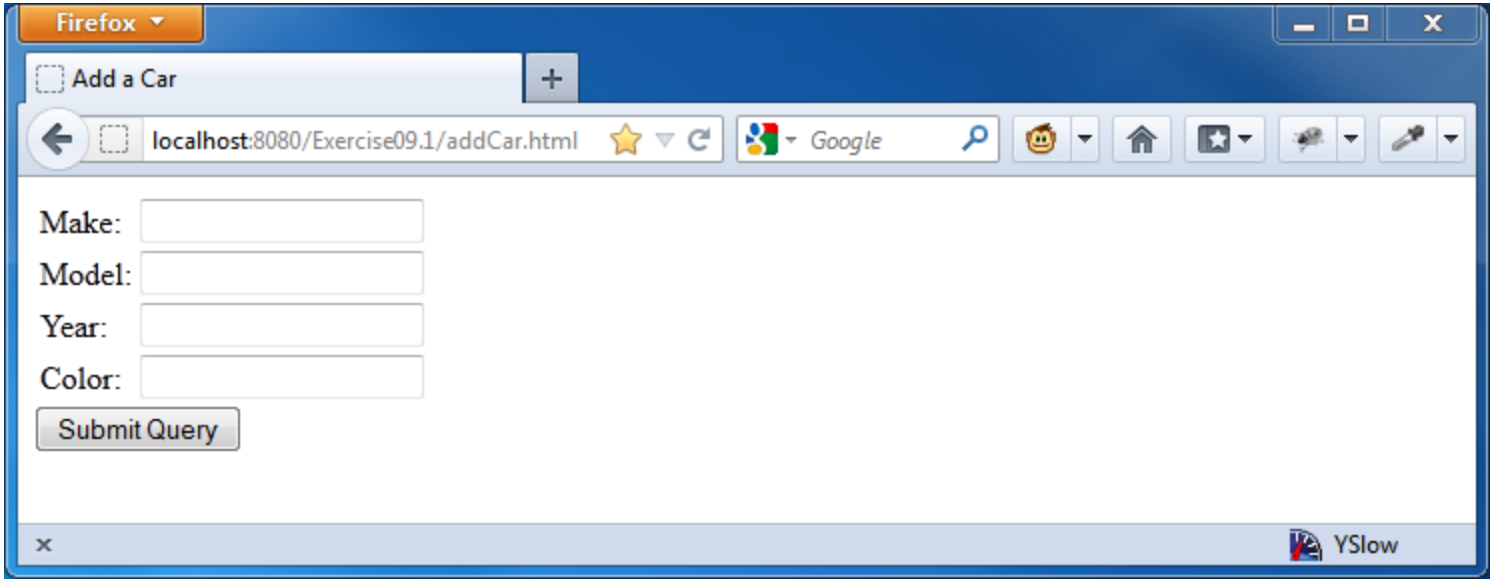
GET /cars?id=1

```
@RequestMapping(value="/cars", params="id", method=RequestMethod.GET)
public String getParam(@RequestParam int id, Model model) {
  model.addAttribute("car", carDao.get(id));
  return "carDetail";
}
```

@RequestParam is optional

Method parameter name has to match request parameter name

Many Parameters




```
public class Car {
  private int id;
  private String make;
  private String model;
  private int year;
  private String color;
```

Do Less and Accomplish More

```
@RequestMapping(value="/cars", method=RequestMethod.POST)
public String addParams(String make, String model, int year, String color) {
  Car car = new Car(make, model, year, color);
  carDao.add(car);
  return "redirect:/cars";
}
```

You can receive the form parameters and combine them into a Car object yourself

But you may as well have Spring do all the work for you

```
@RequestMapping(value="/cars", method=RequestMethod.POST)
public String add(Car car) {
  carDao.add(car);
  return "redirect:/cars";
}
```

Only works if Car class adheres to the Java Bean Spec

Additional Parameters

- The following objects can be passed into Methods:

@PathVariable	HttpServletRequest
@RequestParam	HttpServletResponse
@RequestHeader	HttpSession
@RequestBody	InputStream
@RequestPart (file upload)	OutputStream
Map / Model / ModelMap	Reader
BindingResult / Errors	Writer
SessionStatus	Principal (security)
RedirectAttributes	Locale (internationalization)

- You can also define your own custom injectors
  - See Spring documentation

Spring MVC:

DATA OUTPUT

Data Output

- There are two main ways to output data:
  - Render a view
    - Several ways to specify view name
    - Providing it 'Model' data
  - Output an object
    - Use @ResponseBody on return type
    - Use message converters transforms to desired format
    - View name can be used to specify transformer

For web pages

For web services

Return String View Name

springconfig.xml

```
<!-- Resolves views selected for rendering by @Controllers to .jsp resources
     in the /WEB-INF/views directory -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
  <property name="prefix" value="/WEB-INF/views/" />
  <property name="suffix" value=".jsp" />
</bean>
```

Many other types of ViewResolvers and Views are supported out of the box: Tiles, Velocity, PDF, JExcel, Jasper Reports, XSLT

CarController.java

```
@RequestMapping(value="/cars/{id}", method=RequestMethod.GET)
public String get(@PathVariable int id, Model model) {
  model.addAttribute("car", carDao.get(id));
  return "carDetail";
}
```

Model out parameter

Add data to Model

Specify View

What is the name of our view? / Where will Spring MVC look for it?

View

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml11.dtd">
<html>
<head><title>Add a Car</title></head>
<body>
  <form action="../cars/${car.id}" method="post">
  <table>
    <tr>
      <td>Make:</td>
      <td><input type="text" name="make" value="${car.make}" /> </td>
    </tr>
    <tr>
      <td>Model:</td>
      <td><input type="text" name="model" value="${car.model}" /> </td>
    </tr>
    <tr>
      <td>Year:</td>
      <td><input type="text" name="year" value="${car.year}" /> </td>
    </tr>
    <tr>
      <td>Color:</td>
      <td><input type="text" name="color" value="${car.color}" /> </td>
    </tr>
  </table>
  <input type="submit" value="update"/>
  </form>
  <form action="delete?carId=${car.id}" method="post">
    <button type="submit">Delete</button>
  </form>
</body>
</html>
```

ModelAndView

springconfig.xml

```
<!-- Resolves views selected for rendering by @Controllers to .jsp resources
     in the /WEB-INF/views directory -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
  <property name="prefix" value="/WEB-INF/views/" />
  <property name="suffix" value=".jsp" />
</bean>
```

CarController.java

```
@RequestMapping(value="/cars", params="id", method=RequestMethod.GET)
public ModelAndView get(@RequestParam int id) {
  ModelAndView mav = new ModelAndView();
  mav.setViewName("carDetail");
  mav.addObject("car", carDao.get(id));
  return mav;
}
```

ModelAndView is 'old' pre-Spring 3

Use setViewName() not setView()

@ModelAttribute

- @ModelAttribute critical for form data
  - Especially empty forms
  - The form two slides ago can only display with

CarController.java

```
@RequestMapping(value="/addCar", method=RequestMethod.GET)
public String get(@ModelAttribute("car") Car car) {
    return "addCar";
}
```

Implicit View Name

- You can omit (not specify) a view name
  - Spring uses convention over configuration
  - Convention: convert the request url to view name

DispatcherServlet will instantiate an instance of this bean if one is not explicitly configured thereby providing convention over configuration

springconfig.xml

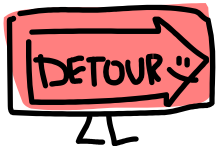
```
<!-- this bean generates view names for us from the request url -->
<bean class="org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator" />

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

CarController.java

```
@RequestMapping(value="/cars", method=RequestMethod.GET)
public void getAll(Model model) {
  model.addAttribute("cars", carDao.getAll());
}
```

No View Name given anywhere

Redirects



- Redirects are important!
  - After processing (POST) input -> always redirect
  - Known as Post/Redirect/Get Pattern*
  - Separation of concerns
  - No problems with refresh
  - No duplicate submissions

*See: http://en.wikipedia.org/wiki/Post/Redirect/Get

Redirects

Zehafta Berhe showed me:
`return "redirect:/comment/{postId}"`

CarController.java

```
@RequestMapping(value="/cars", method=RequestMethod.POST)
public String add(Car car) {
  carDao.add(car);
  return "redirect:/cars";
}
```

Causes browser to: GET /cars

Simply place redirect in front

May contain Path Variables, checks Model for data

ListController.java

```
@RequestMapping(value = "/list", method = RequestMethod.POST)
public RedirectView addItem(@RequestBody Item item)
{
  shoppingListService.addToList(item);
  return new RedirectView("list");
}
```

Pre Spring 3

Spring MVC:

SESSION & FLASH ATTRIBUTES

HttpSession

- If you want you can have direct access to the HttpSession, by requesting it as an additional parameter
  - Not very elegant ☹

HttpSession additional parameter

Creates session if it did not exist yet, or passes existing session

CarController.java

```
@RequestMapping(value="/cars/session")
public @ResponseBody String session(HttpSession session) {
  Enumeration<String> attributes = session.getAttributeNames();
  StringBuilder output = new StringBuilder();
  while (attributes.hasMoreElements()) {
    output.append(attributes.nextElement());
    output.append(" ");
  }
  return output.toString();
}
```

@SessionAttributes

- @Controller can specify @SessionAttributes
  - Intended for the duration of controller

Like Checkout or Authentication Controller

- lists the names of model attributes which should be transparently stored in the session
  - Once the specified attributes are added to the model, they are kept in the model
  - Achieved by Spring storing them in the session

CarController.java

```
@Controller
@SessionAttributes(value={"cars", "currentId"})
public class CarController {
        ...
}
```

Removed on Completion

```
@Controller
@SessionAttributes(value={"cars", "currentId"})
public class CarController {

  ...

  @RequestMapping(value="/cars/clear")
  public String clear(SessionStatus status) {
    // clears SessionAttributes specified on classlevel
    status.setComplete();
    return "redirect:/cars";
  }

  ...
}
```

SessionStatus additional parameter

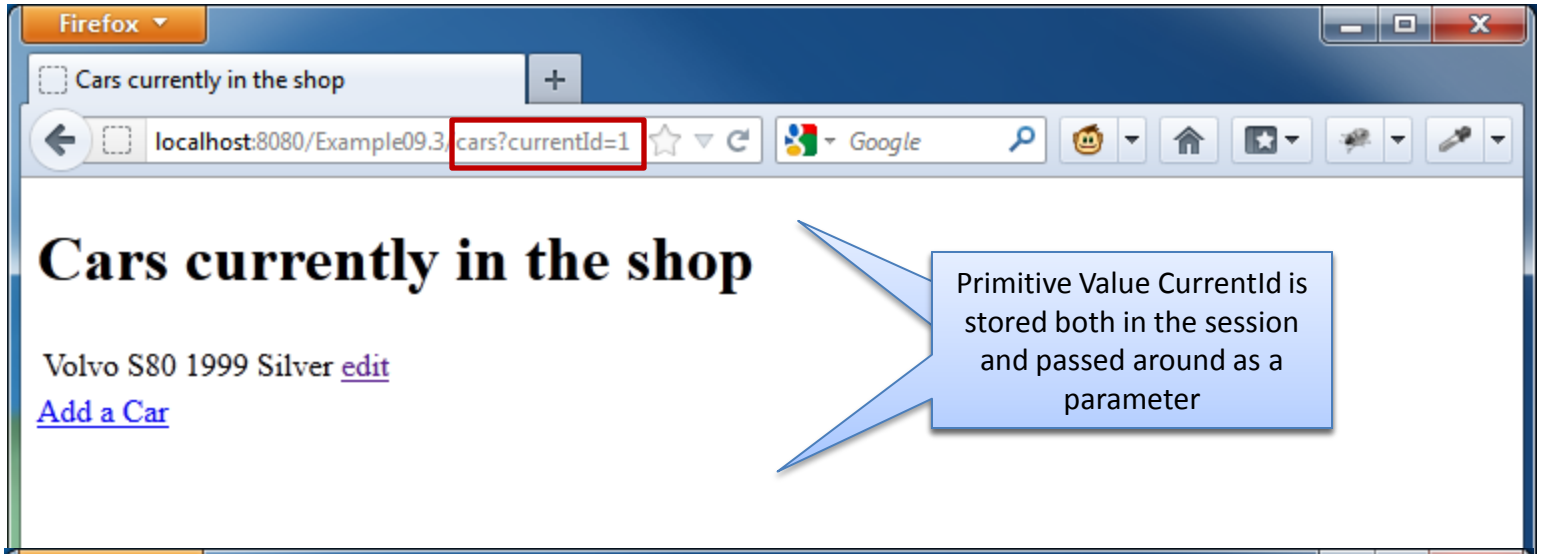
Storing / Retrieving

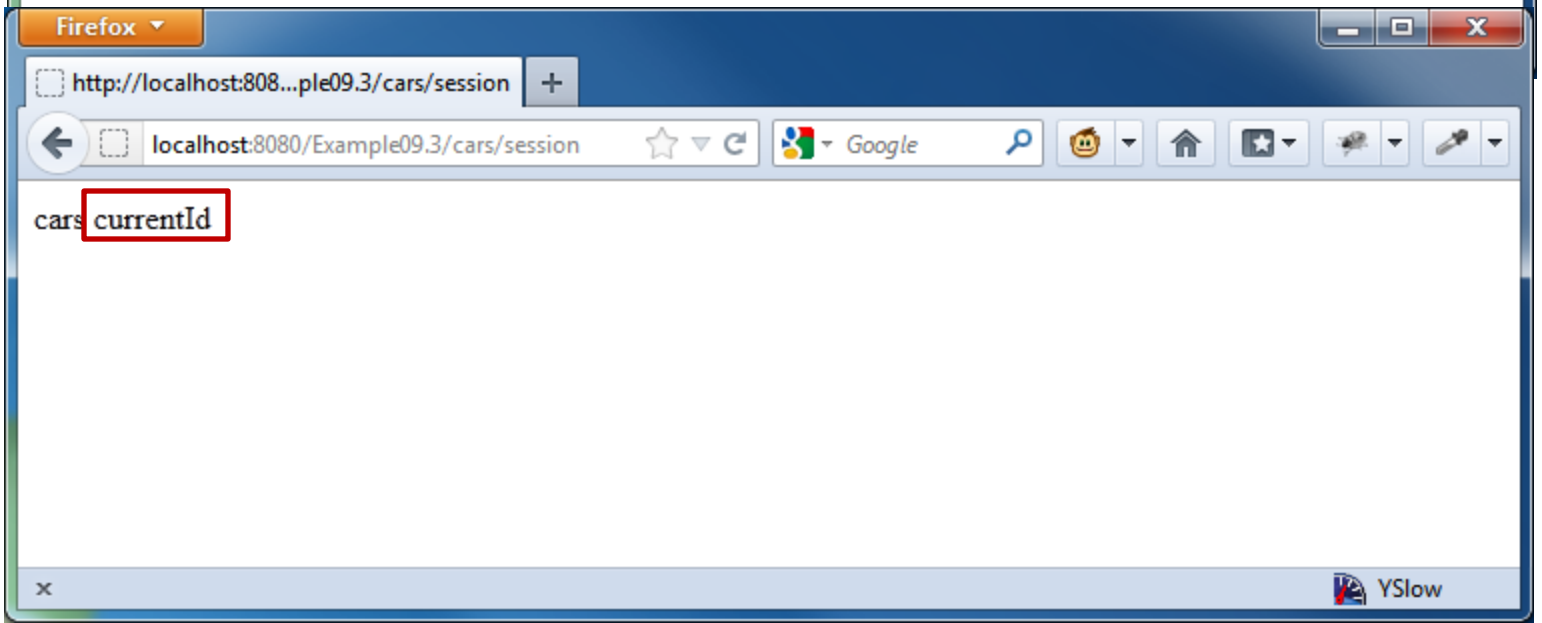
```
@Controller
@SessionAttributes(value={"cars", "currentId"})
public class CarController {

  @RequestMapping(value="/cars", method=RequestMethod.POST)
  public String getCars(Model m) {
    m.addAttribute("cars", carDao.getCars());
    m.addAttribute("currentId", 1);
    return "redirect:/cars";
  }

  @RequestMapping(value="/cars", method=RequestMethod.GET)
  public String viewCars(Model m) {
    List<Car> cars = (List<Car>)m.get("cars"); // just to demonstrate
    int num = m.get("currentId")
    return "cars";
  }
}
```

Method never explicitly uses HttpSession

Once “cars” is added to the model it will be available on subsequent requests

Interesting Side Effect





Flash Attributes

New in Spring 3.1

- Flash attributes provide a way for one request to store attributes intended for use in another.
- This is most commonly needed when redirecting — for example, the *Post/Redirect/Get* pattern.
- Flash attributes are saved temporarily before the redirect (typically in the session) to be made available to the request after the redirect and removed immediately.*

Also stores target URL to make sure it's the right 'next' request

*From: http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/mvc.html#mvc-flash-attributes

Specifying Flash Attributes

CarController.java

```
@Controller
public class CarController {

  @RequestMapping(value="/cars", method=RequestMethod.POST)
  public String add(Car car, RedirectAttributes redirectAttrs) {
    carDao.add(car);
    String msg = "Added " + car.getMake() + " " + car.getModel();
    redirectAttrs.addFlashAttribute("message", msg);
    return "redirect:/cars";
  }

}
```

RedirectAttributes additional parameter

Make sure you use the **addFlashAttribute()** method, not the addAttribute() method

Receiving Flash Attributes

CarController.java

```
@Controller
public class CarController {

  @RequestMapping(value="/cars", method=RequestMethod.GET)
  public String getAll(ModelMap model) {

    if (model.containsAttribute("message")) {
      System.out.println("Message: " + model.get("message"));
    } else {
      System.out.println("No Message");
    }

    model.addAttribute("cars", carDao.getAll());
    return "carList";
  }

}
```

Received Flash Attributes are automatically added to the Model

Whether your method requests a Model parameter or not

This code was written for demonstrative purposes only

Using Flash Attributes

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml11.dtd">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Cars currently in the shop</title>
</head>
<body>
  <h1>Cars currently in the shop</h1>
  <table>
  <c:forEach var="car" items="${cars}">
    <tr>
      <td>${car.make}</td>
      <td>${car.model}</td>
      <td>${car.year}</td>
      <td>${car.color}</td>
      <td><a href="cars/${car.id}">edit</a></td>
    </tr>
  </c:forEach>
  </table>

  <c:if test="${not empty message}">
    <p>Message: <strong>${message}</strong></p>
  </c:if>

  <a href="addCar.html"> Add a Car</a>
</body>
</html>
```

If a flash attribute is passed in, it will be available during view rendering

Spring MVC:

EXCEPTION HANDLING

Exception Handling

- @Controller level
  - Annotate methods with @ExceptionHandler
- Dispatcher Servlet Config
  - DefaultHandlerExceptoinResolver
    - Maps common exceptions to appropriate status codes
  - Add Custom HandlerExceptionResolver as needed

Exception Handling

CarController.java

```
@ExceptionHandler(value=NoSuchResourceException.class)
public ModelAndView handle(Exception e) {
  ModelAndView mav = new ModelAndView();
  mav.addObject("e", e);
  mav.setViewName("noSuchResource");
  return mav;
}
```

I was unable to use Spring 3 style Model / return String?

Active Learning

- What are the different ways that the view can be specified?

- Why is it generally not necessary to redirect GET requests?

Summary

- We've discussed:
  - The application context in a web container
  - Spring MVC Request Mapping
  - URI templates
  - Data input / Data Output
  - Sessions & Flash Attributes
  - Exception Handling

Main Point

- There is a lot to SpringMVC, but at its core it is a request centric web framework with URI templates.
- Science of Consciousness: Every action has a reaction – with Spring MVC we specify what the reaction (method) should be to each request.