Spring Web

Spring Framework

Lorenzo Dee, VP Consulting
Orange & Bronze Software Labs

Getting Started

Spring Framework (Web/MVC)

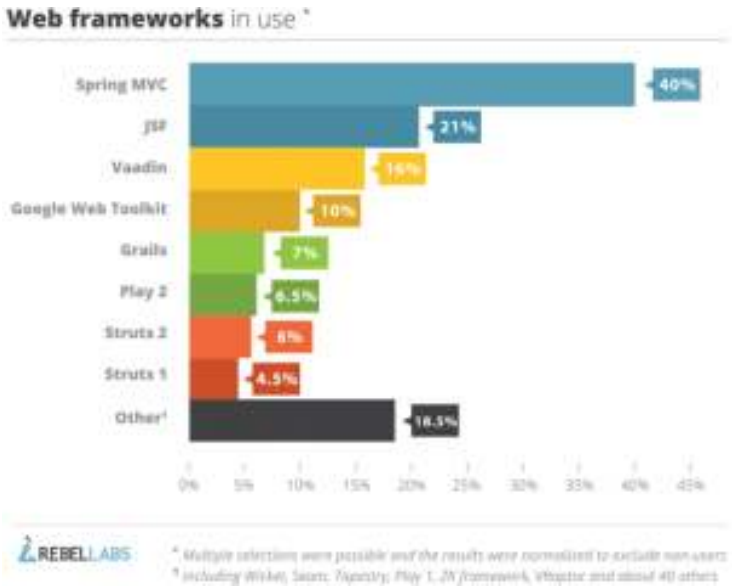
Getting Started
Spring Framework (Web/MVC)

- Spring model-view-controller (MVC) overview
- `DispatcherServlet`
- Controller programming model overview
- Spring MVC views
- *Lab*
- Simplifying configuration
- *Lab*

Spring Web/MVC Overview
Getting Started with Spring Web/MVC

- Request-driven framework
- Spring Framework has:
  - Spring Web module (`spring-web-xxx.jar`)
  - Spring Web MVC module (`spring-webmvc-xxx.jar`)

Getting Started with Spring Web/MVC



Source: Java Tools and Technologies Landscape for 2014

Spring Web/MVC Overview

Getting Started with Spring Web/MVC

- Uses Spring for its own configuration
- Supports various web technologies
- Flexible and extensible

`DispatcherServlet`

Getting Started with Spring Web/MVC

from Martin
Fowler's PoEAA

- Heart of Spring MVC
- Implements the *front controller pattern*
  - analogous to Struts `ActionServlet` and JSF `FacesServlet`
- Handles every incoming request
- Coordinates request-handling activities

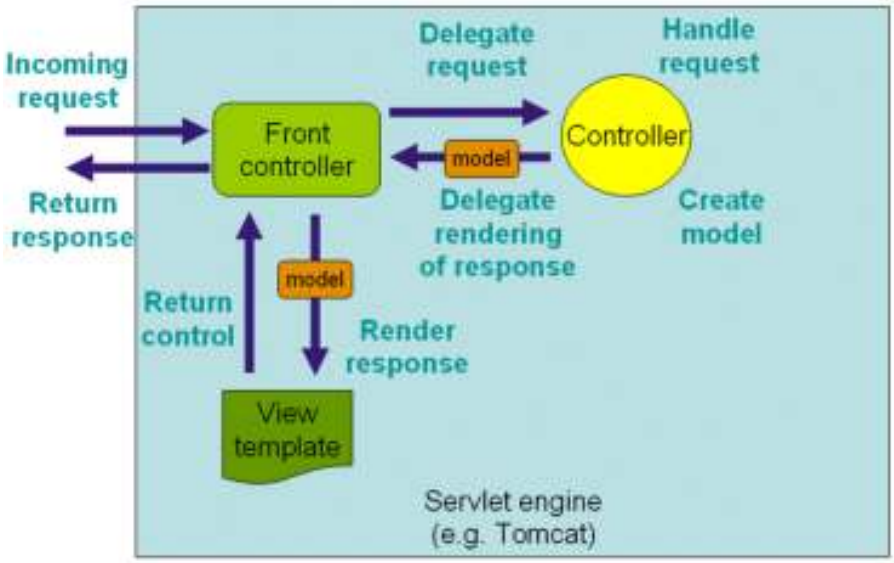
`DispatcherServlet`

Getting Started with Spring Web/MVC

- The `DispatcherServlet` doesn't do all the work
  - delegates to other components
- Spring MVC infrastructure components
  - handler mappings
  - handler adapters
  - view resolvers
- User-provided components
  - controllers
  - handler interceptors

DispatcherServlet

Getting Started with Spring Web/MVC



DispatcherServlet

Getting Started with Spring Web/MVC

XML

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

    <servlet-mapping>
        <servlet-name>example</servlet-name>
        <url-pattern>/example/*</url-pattern>
    </servlet-mapping>
</web-app>
```

DispatcherServlet
Getting Started with Spring Web/MVC

Upon initialization of a `DispatcherServlet`, Spring MVC looks for a file named `[`*servlet-name*`]-servlet.xml` in the `WEB-INF` directory of your web application and creates the beans defined there,

XML

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

With the above Servlet configuration in place, you will need to have a file called `/WEB-INF/`example`-servlet.xml` in your application.

DispatcherServlet
Getting Started with Spring Web/MVC

The *default* location of the configuration file can be changed with `contextConfigLocation` initialization parameter.

XML

```xml
<web-app>
    <servlet>
        <servlet-name>example</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring/web-config.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    ...
</web-app>
```

DispatcherServlet

Getting Started with Spring Web/MVC

XML

```
<web-app>
    <servlet>
        <servlet-name>example</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>
                /WEB-INF/spring/*-beans.xml
                classpath:com/example/application-config.xml
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    ...
</web-app>
```

wildcards are accepted

can use `classpath` prefix

DispatcherServlet

Getting Started with Spring Web/MVC

How does Spring MVC interact with other Spring configured objects?

Several options

1. All in a `DispatcherServlet`
2. Separate configuration
3. All in root context

All in a `DispatcherServlet`

Getting Started with Spring Web/MVC

XML

```
<web-app>
    <servlet>
        <servlet-name>example</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>
                /WEB-INF/spring/web-config.xml
                classpath:com/myapp/application-config.xml
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    ...
</web-app>
```


Separate configuration

Getting Started with Spring Web/MVC

XML

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:com/myapp/application-config.xml</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<servlet>
    <servlet-name>example</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/web-config.xml</param-value>
    </init-param>...
</servlet>
<!-- servlet-mappings -->
```

All in Root Context

Getting Started with Spring Web/MVC

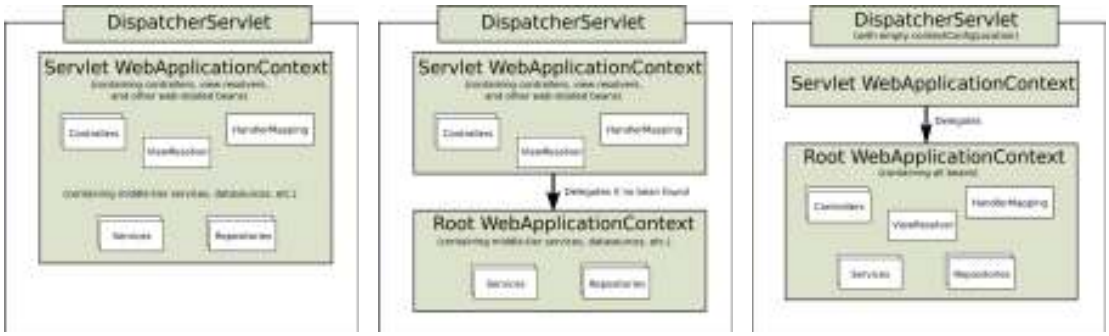
XML

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:com/myapp/application-config.xml
        /WEB-INF/spring/web-config.xml
    </param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<servlet>...
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value></param-value>
    </init-param>...
</servlet>...
```

empty value

DispatcherServlet

Getting Started with Spring Web/MVC



`DispatcherServlet`

Getting Started with Spring Web/MVC

Which one is better?

It depends! If Spring MVC is not the only entry-point to your other Spring configured objects.

Spring MVC
`DispatcherServlet`

Other Servlets (e.g. web services, non-MVC)

Other Spring configured objects (e.g. services, repositories that make up the back-end of the application)

Controller Programming Model

Getting Started with Spring Web/MVC

- Annotation-based programming model (since Spring 2.5)
- Supersedes the `Controller` base-class hierarchy (deprecated in Spring 3.0)

Controller Programming Model
Getting Started with Spring Web/MVC

Steps to creating a controller:

1. Declare controller class (a POJO)
2. Define one or more request-handling methods
3. Map URLs to methods using annotations
4. Implement method body
   - Invoke application services
   - Populate a `Model`
   - Select a `View`

Controller Programming Model
Getting Started with Spring Web/MVC

JAVA

```java
@Controller
public class AccountController {
    private AccountManager accountManager;
    @Autowired
    public AccountController(AccountManager accountManager) {
        this.accountManager = accountManager;
    }

    public String list(Model model) {
        model.addAttribute("accounts", accountManager.findAll());
        return "/WEB-INF/views/accounts/list.jsp";
    }
}
```

Controller Programming Model
Getting Started with Spring Web/MVC

Steps to creating a controller:

1. Declare controller class (a POJO)
2. Define one or more request-handling methods
3. **Map URLs to methods using annotations**
4. Implement method body
   - Invoke application services
   - Populate a `Model`
   - Select a `View`

Controller Programming Model
Getting Started with Spring Web/MVC

Mapping requests:

- By URL only

```java
@RequestMapping("/accounts")
public String list(Model model) {...}
```

- By URL *and* request method

```java
@RequestMapping(method=RequestMethod.GET, value="/accounts")
public String list(Model model) {...}
```

Controller Programming Model
Getting Started with Spring Web/MVC

Mapping requests *(continued)*:

- By presence of a request parameter (`/url/path?`*paramName*`=205008`)

```java
@RequestMapping(value="/url/path", params={"paramName"})
public String byId(Model model) {...}
```

JAVA

- By presence of a request parameter with a specific value (`/url/path?`*offline=true*)

```java
@RequestMapping(value="/url/path", params={"offline=true"})
public String offline(Model model) {...}
```

JAVA

Controller Programming Model
Getting Started with Spring Web/MVC

`@RequestMapping` can be applied to *class* and *method* level

```java
@Controller
@RequestMapping("/accounts")
public class AccountController {
    ...
    @RequestMapping(method=RequestMethod.GET)
    public String list(Model model) {...}
    @RequestMapping(method=RequestMethod.GET,
                    value="/method2")
    public String method2(Model model) {...}
}
```

JAVA

"/accounts/method2"

Any method without `@RequestMapping` is ignored.

Controller Programming Model
Getting Started with Spring Web/MVC

Steps to creating a controller:

1. Declare controller class (a POJO)
2. Define one or more request-handling methods
3. Map URLs to methods using annotations
4. **Implement method body**
   - Invoke application services
   - Populate a `Model`
   - Select a `View`

Controller Programming Model
Getting Started with Spring Web/MVC

Select a `View`

- A `@Controller` selects the view to render the model
- By default the view name is expected to be a JSP path

JAVA

```
@RequestMapping(method=RequestMethod.GET)
public String list(Model model) {
    ...
    return "/WEB-INF/views/accounts/list.jsp";
}
```

- The returned `String` is known as the *view name*

Controller Programming Model
Getting Started with Spring Web/MVC

- Request-handling methods have flexible signatures
- List of supported arguments (in any order):
  - `org.springframework.ui.Model`
  - `HttpServletRequest`, `HttpServletResponse`
  - `HttpSession`
  - `java.util.Locale`
  - `java.security.Principal`
  - more options explained later

Controller Programming Model
Getting Started with Spring Web/MVC

Access Request Parameters

- When using URI Templates, access the parameters using the `@PathVariable` annotation
  - Parameter is extracted from the request
  - Type conversion is applied (primitive or wrapper)

JAVA

```java
@RequestMapping(value="/accounts/{id}")
public String list(@PathVariable("id") String id, Model model) {
    ...
}
```

http://somewhere.org/leaveapp/accounts/**123456789**

`@PathVariable` was introduced in Spring 3.0.

Controller Programming Model
Getting Started with Spring Web/MVC

Access Request Parameters (continued)

- `@PathVariable` annotations can be limited via *regular expressions*
  - Useful for resolving ambiguity in URLs

```java
@RequestMapping(value="/accounts/{id:[\\d]*}")
public String list(@PathVariable("id") Long id, Model model) {...}
```

- The *value* in `@PathVariable` annotatation is *optional*, since it defaults to argument with same name (but **must** compile with debug symbols enabled)

```java
@RequestMapping(value="/accounts/{id}")
public String list(@PathVariable("id") String id, Model model) {...}
```


Controller Programming Model
Getting Started with Spring Web/MVC

Access Request Parameters (continued)

- Access URL parameters via `@RequestParam`
  - Parameter is extracted from the request
  - Type conversion is applied (primitive or wrapper)
  - Causes an exception if missing or wrong type

```java
@RequestMapping(value="/accounts/show")
public String list(@RequestParam("id") String id, Model model) {
    ...
}
```

http://somewhere.org/leaveapp/accounts/show?id=**123456789**

Controller Programming Model
Getting Started with Spring Web/MVC

Access Request Parameters (continued)

- URL parameters via `@RequestParam` can be optional
  - **Must** be an object (set to `null` if not in URL)

JAVA

```
@RequestMapping(value="/accounts/show")
public String list(
    @RequestParam(value="id", required=false) String id) {...}
```

- Parameter name can be optional (defaults to argument name)

JAVA

```
@RequestMapping(value="/accounts/show")
public String list(@RequestParam String id) {
    ...
}
```

http://somewhere.org/leaveapp/accounts/show?**id**=123456789

Controller Programming Model
Getting Started with Spring Web/MVC

`@PathVariable` vs `@RequestParam`

- `@PathVariable` uses variable in the URL path

```
http://somewhere.net/accounts/{id}
http://somewhere.net/accounts/3
```

- `@RequestParam` uses variable in the query string of the URL (*after* the question mark **?**)

```
http://somewhere.net/accounts/show?id=3
```

Controller Programming Model
Getting Started with Spring Web/MVC

Formatting Request Parameters

- Date and number formats can be defined
  - `@PathVariable` and/or `@RequestParam` parameters
  - Avoids custom `PropertyEditor`s for dates or numbers

JAVA

```
@RequestMapping(value="/transactions/{day}")
public String show(
    @PathVariable("day")
    @DateTimeFormat(iso=ISO.DATE) Date day,
    @RequestParam("hourlyRate")
    @NumberFormat(style=Style.CURRENCY) double rate)
```

Set to `2.50`

/transactions/2015-02-14?hourlyRate=$2.50

Also, `Style.PERCENT` and numeric patterns: `#.##`, `#0.00`, `#,###.##`.

Controller Programming Model
Getting Started with Spring Web/MVC

`<spring:url />` Tag

- A drop-in replacement for JSTL `<c:url />` tag
- Allows creation of URL templates (remember `@PathVariables`?)
  - Variables are URL encoded

JSP

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<spring:url value="/accounts/{number}">
    <spring:param name="number" value="${account.number}" />
</spring:url>
```

Controller Programming Model
Getting Started with Spring Web/MVC

Add Model Attributes

- A model is *always* created
- Use the model to provide attributes to be used by views

JAVA

```
@RequestMapping(method=RequestMethod.GET)
public String list(Model model) {
    model.addAttribute("accounts", accountManager.findAll());
    return "/WEB-INF/views/accounts/list.jsp";
}
```

- All model attributes available in the JSP for rendering

JSP

```
<c:forEach var="account" items="${accounts}">...</c:forEach>
```


Controller Programming Model
Getting Started with Spring Web/MVC

Test Controllers

JAVA

```
public class AccountControllerTests {
    private AccountController controller;
    @Before
    public void setUp() throws Exception {
        controller = new AccountController(new StubAccountManager());
    }
    @Test
    public void testList() throws Exception {
        Model model = new BindingAwareModelMap();
        controller.list(model);
        List<Account> accounts = (List<Account>) model.asMap.get("accounts");
        assertEquals(1, accounts.size());
    }
}
```

Controller Programming Model

Getting Started with Spring Web/MVC

Steps to creating a controller:

1. Declare controller class (a POJO)
2. Define one or more request-handling methods
3. Map URLs to methods using annotations
4. Implement method body
   - Invoke application services
   - Populate a `Model`
   - Select a `View`

Don't forget to add tests to ensure your controllers are making the right calls and configured properly.

Views

Getting Started with Spring Web/MVC

org.springframework.web.servlet.`View`

- A *strategy* for rendering the model
- Variation by content type (e.g. HTML, CSV, PDF)
- Variation by rendering technology
  (e.g. JSP, FreeMarker, Velocity, Facelets, Thymeleaf)

JAVA

```
public interface View {
    ...
    String getContentType();
    void render(Map<String, ?> model,
        HttpServletRequest request, HttpServletResponse response) throws Exception;
}
```

Views
Getting Started with Spring Web/MVC

Available view types

- Display views
  - JSP, Apache Tiles, FreeMarker, Velocity, Thymeleaf
- File-generating views
  - Apache POI, jExcelApi (Excel)
  - iText (PDF)
  - JasperReports
  - XSLT (transformation)
- Data-delivery views
  - JSON, Java-XML marshalling, Atom, RSS

Views
Getting Started with Spring Web/MVC

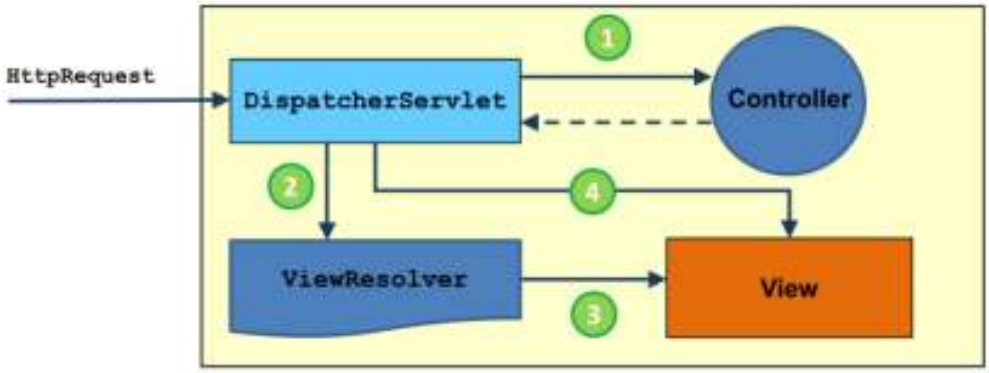
The `ViewResolver` abstraction

- a *factory* for creating `View` instances

```java
public interface ViewResolver {
    View resolveViewName(String viewName, Locale locale);
}
```

- decouples controller from view implementation
  - possible to switch from JSP to Velocity, etc.

Views
Getting Started with Spring Web/MVC



View resolution sequence

Views
Getting Started with Spring Web/MVC

- The *default* `ViewResolver` implementation is `InternalResourceViewResolver`.
  - It interprets view names as JSP paths
  - Creates instances of `JstlView`

Views

Getting Started with Spring Web/MVC

- `InternalResourceViewResolver` is commonly configured with the following prefix/suffix

XML

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

- Controllers can then return "logical view name"

JAVA

```java
@RequestMapping("/accounts")
public String list(Model model) { ...
    return "/WEB-INF/accounts/list.jsp" "accounts/list";
}
```


Pop Quiz

Getting Started with Spring Web/MVC

By default, when you use a `ContextLoaderListener`, it delegates to an `XmlWebApplicationContext` which loads `/WEB-INF/applicationContext.xml` for the root context. What overrides the default config location?

Pop Quiz
Getting Started with Spring Web/MVC

By default, what configuration file will be loaded by the `DispatcherServlet` defined below?

```XML
<web-app>
    <servlet>
        <servlet-name>merchant</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>merchant</servlet-name>
        <url-pattern>/merchant/*</url-pattern>
    </servlet-mapping>
</web-app>
```


Pop Quiz
Getting Started with Spring Web/MVC

Given that a `DispatcherServlet` is mapped to `/` (i.e. all requests), which annotation will be mapped to this URL `http://.../products/35650204`?

a. `@Controller("/products/{productId}")`

b. `@RequestParam("/products/{productId}")`

c. `@RequestMapping("/products/{productId}")`

d. `@RequestMapping("/product/{productId}")`

Pop Quiz
Getting Started with Spring Web/MVC

Which of the following statements about Spring MVC are true?

a. `@RequestParam` maps path variables in URI templates (e.g. `/accounts/{accountId}`)

b. `@PathVariable` maps path variables in URI templates (e.g. `/accounts/{accountId}`)

c. `@RequestParam` maps request parameters (e.g. `/accounts?accountId=12345`)

d. `@PathVariable` maps request parameters (e.g. `/accounts?accountId=12345`)

Choose all that are true.

Pop Quiz
Getting Started with Spring Web/MVC

Which of the following statements about `@RequestParam` Spring MVC are true?

a. It maps request parameters and are *required* by default

b. It maps request parameters and are *optional* by default

c. Mapped parameters can be made optional by setting the `required` attribute to `false`

d. Mapped parameters can be made optional by setting the `optional` attribute to `true`

e. Mapped parameters that are made optional must map to types that can be made `null`

Choose all that are true.

Getting Started
Spring Framework (Web/MVC)

Lab

Simplifying Configuration
Getting Started with Spring Web/MVC

- Simplifying controller setup
  - `<mvc>` namespace
  - mapping `DispatcherServlet` to root path (`/`)
  - views without a controller
- Applying conventions
  - model attribute names
  - view names
  - URL mapping

Simplifying Configuration
Getting Started with Spring Web/MVC

- Spring MVC uses the custom **<mvc> namespace** to simplify XML configuration (new in Spring 3.0)

XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
</beans>
```


Simplifying Configuration
Getting Started with Spring Web/MVC

- Namespace tags simplify common configuration tasks

XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="...">

    <mvc:annotation-driven />

    <mvc:default-servlet-handler />

</beans>
```

Simplifying Configuration
Getting Started with Spring Web/MVC

- Mapping a `DispatcherServlet` adds a servlet element to the URL
  - **/myapp**/accounts/3

```
<servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
...
<servlet-mapping>
    <servlet-name>main</servlet-name>
    <url-pattern>/myapp</url-pattern>
</servlet-mapping>
```

XML

- As of Spring MVC 3.0.4, the `DispatcherServlet` can be mapped to `/`

Simplifying Configuration
Getting Started with Spring Web/MVC

- As of Spring MVC 3.0.4, the `DispatcherServlet` can be mapped to **/**
- An additional element must be defined to pass requests to servlet container
  - used to serve static assets (e.g. images, JavaScript files, etc.)
  - only required when mapped to **/**

```
<servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>...
<servlet-mapping>
    <servlet-name>main</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

XML

Simplifying Configuration
Getting Started with Spring Web/MVC

XML

```
<!-- web.xml -->
<servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>...DispatcherServlet</servlet-class>
</servlet>...
<servlet-mapping>
    <servlet-name>main</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

- The additional elements pass requests to servlet container

XML

```
<!-- main-servlet.xml -->
<mvc:annotation-driven />
<mvc:default-servlet-handler />
```


Views Without a Controller
Simplifying Configuration

- Some views don't need a controller
  - We just want to render a view
- Use `<mvc:view-controller>` for this
  - Specify URL and view name in configuration

XML

```
<mvc:annotation-driven />
<mvc:view-controller path="/login" view-name="login" />
<mvc:view-controller path="/welcome" view-name="welcome"
```

logical view names

Convention for Model Attribute Names
Simplifying Configuration

- A *model attribute* name may be left unspecified
  - If so, a *default* name is used:
    - based on concrete *type* of the attribute
    - for arrays and collections: *type* suffixed with `"List"`

JAVA

```
Account account = accountManager.findOne(accountNumber);
model.addAttribute("account", account);
model.addAttribute(account); // added as "account"

List<Account> accounts = accountManager.findAll();
model.addAttribute(accounts); // added as "accountList"
```


Shortcut for Adding a Single Model Attribute
Simplifying Configuration

- A single *model attribute* can simply be returned

JAVA

```
@RequestMapping("/accounts/list")
public List<Account> list() {
    return accountManager.findAll();
        // added as a model attribute named? "accountList"
}
```

JAVA

```
@RequestMapping("/accounts/list")
public @ModelAttribute("accounts") List<Account> list() {
    return accountManager.findAll();
}
```

Convention for View Names
Simplifying Configuration

- Request-handling methods may leave *view name* unspecified
  - return `null`, method returns `void`, or return model only (no view name)
- If so, a *default* logical view name is selected
  - via `RequestToViewNameTranslator`
  - leading slash (`/`) and extension removed from URL
  - view resolver is required

JAVA

```java
@RequestMapping("/accounts/list.html")
public void list(Model model) {...}
    // goes to "accounts/list" view
```

XML

```xml
<mvc:view-controller path="/welcome" view-name="..." />
<!-- since view-name is not specified, goes to "welcome" view -->
```


Convention for URL Mappings
Simplifying Configuration

- Controller class name can be used to generate URL mappings
  - `AccountsController` → `/accounts`, `/accounts/*`
  - `WelcomeController` → `/welcome`, `/welcome/*`

XML

```xml
<bean class="org.springframework.web...DefaultAnnotationHandlerMapping">
    <property name="order" value="0" />
</bean>
<bean class="org.springframework.web...ControllerClassNameHandlerMapping">
    <property name="order" value="1" />
</bean>
```

Convention for URL Mappings
Simplifying Configuration

- Controller class name can be used to generate URL mappings
  - `AccountsController` → `/accounts`, `/accounts/*`
  - `WelcomeController` → `/welcome`, `/welcome/*`
- Method-level `@RequestMapping` further narrows down a method within a controller
  - 1st check `HTTP` request method/verb (`GET`, `POST`, etc)
  - 2nd check for presence of request parameter(s)
  - 3rd fall-back on the method name if necessary

Convention for URL Mappings
Simplifying Configuration

Mapped using controller-class-name conventions with method-level annotations

XML

```
<bean class="org.springframework.web...DefaultAnnotationHandlerMapping">
    <property name="order" value="0" />
</bean>
<bean class="org.springframework.web...ControllerClassNameHandlerMapping">
    <property name="order" value="1" />
</bean>
```

JAVA

```
@Controller // mapped to /accounts and /accounts/*
public class AccountsController {
    @RequestMapping(method=RequestMethod.GET)
    public void list(Model model) {...} // mapped to /accounts/list
    @RequestMapping(method=RequestMethod.GET)
    public void show(String id) {...} // mapped to /accounts/show
}
```

Convention for URL Mappings
Simplifying Configuration

A Note on URL Strategy

- Using **controller-class-name** conventions results in a different URL strategy
  - `GET /accounts/show?id=1` *VS* `GET /accounts/1`
  - `GET /accounts/list` *VS* `GET /accounts`
- Spring MVC 3.1 supports URI templates (remember `@PathVariables`?)

Getting Started
Spring Framework (Web/MVC)

Lab

Configuration Options (1)

Spring Framework (Web/MVC)

Configuration Options (1)

Spring Framework (Web/MVC)

- Spring MVC infrastructure beans
- URL mappings
- Handler interceptors and handler adapters
- Exception resolvers
- Message source

Spring MVC Infrastructure Beans
Configuring Spring Framework (Web/MVC)

`DispatcherServlet` looks for some beans by type:

- `HandlerMapping`
- `HandlerAdapter`
- `ViewResolver`
- `HandlerExceptionResolver`

Out-of-the-box implementations of the above interfaces are provided. Supports customizations via properties.

Spring MVC Infrastructure Beans
Configuring Spring Framework (Web/MVC)

- Multiple infrastructure beans are *chained* (GoF patterns: *Chain of Responsibility*)
  - The first one to return non-`null` value wins

XML

```
<bean class="org.springframework.web...DefaultAnnotationHandlerMapping">
    <property name="order" value="0" />
</bean>
<bean class="org.springframework.web...ControllerClassNameHandlerMapping">
    <property name="order" value="1" />
</bean>
```

Spring MVC Infrastructure Beans
Configuring Spring Framework (Web/MVC)

- Defaults exist for *all* infrastructure beans
- See `org.springframwork.web.servlet.DispatcherServlet.properties`

Example:

- HandlerAdapter
  - HttpRequestHandlerAdapter
  - SimpleControllerHandlerAdapter
  - AnnotationMethodHandlerAdapter

> NOTE: Configuring infrastructure beans explicitly cancels all defaults for that bean type!

Spring MVC Infrastructure Beans
Configuring Spring Framework (Web/MVC)

PROPERTIES

```
...
o.s.web.servlet.HandlerMapping=\
    o.s.web.servlet.handler.BeanNameUrlHandlerMapping,\
    o.s.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping
o.s.web.servlet.HandlerAdapter=\
    o.s.web.servlet.mvc.HttpRequestHandlerAdapter,\
    o.s.web.servlet.mvc.SimpleControllerHandlerAdapter,\
    o.s.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter
o.s.web.servlet.HandlerExceptionResolver=\
    o.s.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver,\
    o.s.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\
    o.s.web.servlet.mvc.support.DefaultHandlerExceptionResolver
...
```

`org.springframework` has been shortened to `o.s` for brevity

Spring MVC Infrastructure Beans
Configuring Spring Framework (Web/MVC)

With `<mvc:annotation-driven>`, the following beans are added:

PROPERTIES

```
...
o.s.web.servlet.HandlerMapping=\
    RequestMappingHandlerMapping
    o.s.web.servlet.handler.BeanNameUrlHandlerMapping,\
    o.s.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping
o.s.web.servlet.HandlerAdapter=\
    RequestMappingHandlerAdapter
    o.s.web.servlet.mvc.HttpRequestHandlerAdapter,\
    o.s.web.servlet.mvc.SimpleControllerHandlerAdapter,\
    o.s.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter
o.s.web.servlet.HandlerExceptionResolver=\
    ExceptionHandlerExceptionResolver
    o.s.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver,\
    o.s.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\
    o.s.web.servlet.mvc.support.DefaultHandlerExceptionResolver
...
```

`org.springframework` has been shortened to `o.s` for brevity

Spring MVC Infrastructure Beans
Configuring Spring Framework (Web/MVC)

Java-based configuration equivalent to `<mvc:annotation-driven>`

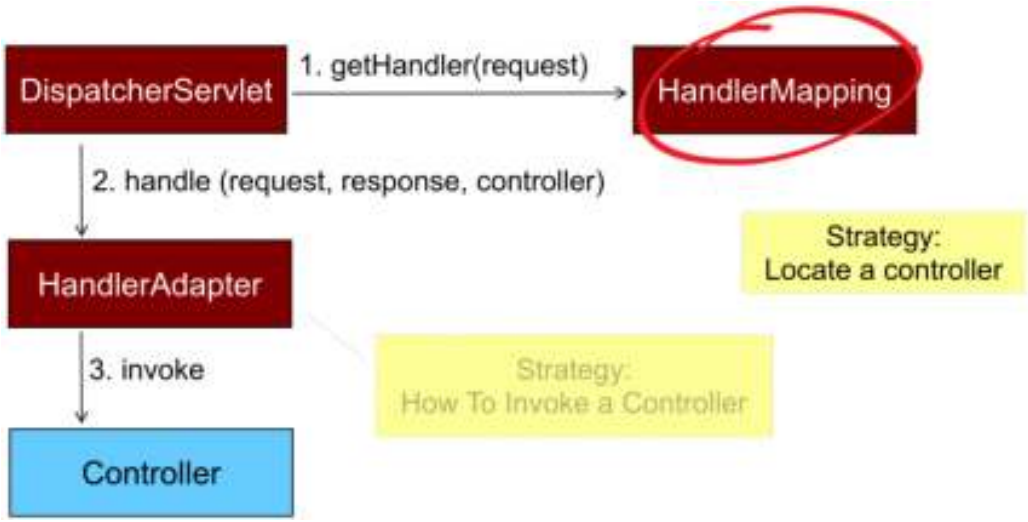
JAVA

```
@EnableWebMvc
@Configuration
public class MyWebConfig extends WebMvcConfigurerAdapter {
    // Override some methods
}
```

Configuration Options (1)
Spring Framework (Web/MVC)

- Spring MVC infrastructure beans
- **URL mappings**
- Handler interceptors and handler adapters
- Exception resolvers
- Message source

URL Mappings
Configuring Spring Framework (Web/MVC)



URL Mappings
Configuring Spring Framework (Web/MVC)

- `DefaultAnnotationHandlerMapping`
- `RequestMappingHandlerMapping`
- `ControllerClassNameHandlerMapping`
- `SimpleUrlHandlerMapping`

URL Mappings
Configuring Spring Framework (Web/MVC)

JAVA

```java
public interface HandlerMapping {
    ...
    /**
     * Return a handler and any interceptors for this request...
     * Returns {@code null} if no match was found. This is not an error...
     */
    HandlerExecutionChain getHandler(
            HttpServletRequest request) throws Exception;
}
```

The returned `HandlerExecutionChain` contains a handler `Object`, rather than an interface, so that handlers are not constrained in any way. For example, a `HandlerAdapter` could be written to allow another framework's handler objects to be used.

URL Mappings
Configuring Spring Framework (Web/MVC)

JAVA

```java
public interface HandlerMapping {
    ...
    HandlerExecutionChain getHandler(
            HttpServletRequest request) throws Exception;
}
```

JAVA

```java
public class HandlerExecutionChain {
    ...
    public Object getHandler() {...}
    ...
    public HandlerInterceptor[] getInterceptors() {...}
    ...
}
```


URL Mappings
Configuring Spring Framework (Web/MVC)

`DefaultAnnotationHandlerMapping`

- An all annotations approach

JAVA

```java
@Controller
@RequestMapping("/accounts")
public class AccountController {
    @RequestMapping(method=RequestMethod.GET)
    public String list(Model model) {...}
}
```

- Deprecated as of Spring MVC 3.2 in favor of `RequestMappingHandlerMapping`.

URL Mappings
Configuring Spring Framework (Web/MVC)

`RequestMappingHandlerMapping`

- `HandlerMapping` implementation based on the new `HandlerMethod` abstraction (as of Spring MVC 3.1)
  - Allows `@RequestMapping`s for same URL to be in different controllers
  - Pluggable support for any argument/return type in handler methods
- Intended as a replacement of `DefaultAnnotationHandlerMapping`
- Configured as **the** `HandlerMapping` strategy with `<mvc:annotation-driven>`

URL Mappings
Configuring Spring Framework (Web/MVC)

`ControllerClassNameHandlerMapping`

- Map controllers by class-name convention and method-relative via `@RequestMapping`

JAVA

```java
@Controller
public class AccountsController {
    @RequestMapping(method=RequestMethod.GET)
    public String list(Model model) {...} // maps to "/accounts/list"
    @RequestMapping(method=RequestMethod.GET)
    public String show(String id) {...} // maps to "/accounts/show"
}
```

URL Mappings
Configuring Spring Framework (Web/MVC)

`SimpleUrlHandlerMapping`

- Declarative style URL-to-controller mapping

XML

```
<bean class="...SimpleUrlHandlerMapping">
    <property name="mappings">
        <value>
            /welcome=welcomeController
            /accounts/**=accountController
        </value>
    </property>
</bean>
```

Mapped to `/accounts/list`.

JAVA

```
@Controller
public class AccountController {
    @RequestMapping(method=RequestMethod.GET)
    public String list(Model model) {...}
}
```


Configuration Options (1)
Spring Framework (Web/MVC)

- Spring MVC infrastructure beans
- URL mappings
- **Handler interceptors and handler adapters**
- Exception resolvers
- Message source

Handler Interceptors
Configuring Spring Framework (Web/MVC)

- Useful for applying functionality that is common to many controllers
  - Add common model attributes (e.g. menus, preferences)
  - Set response headers
  - Audit requests
  - Measure performance (controller vs. view rendering time)

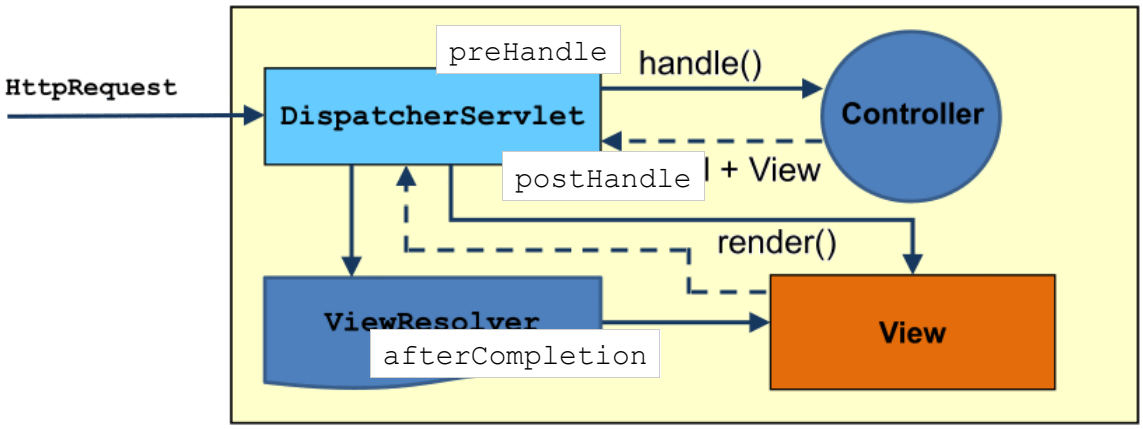
Handler Interceptors
Configuring Spring Framework (Web/MVC)

JAVA

```java
public interface HandlerInterceptor {
    boolean preHandle(
        HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception;
    void postHandle(
        HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception;
    void afterCompletion(
        HttpServletRequest request,
        HttpServletResponse response, Object handler,
        Exception ex) throws Exception;
}
```

`preHandle` and `postHandle` are called *before* and *after* controller invocation, respectively. `afterCompletion` is invoked after view rendering.

Handler Interceptors
Configuring Spring Framework (Web/MVC)



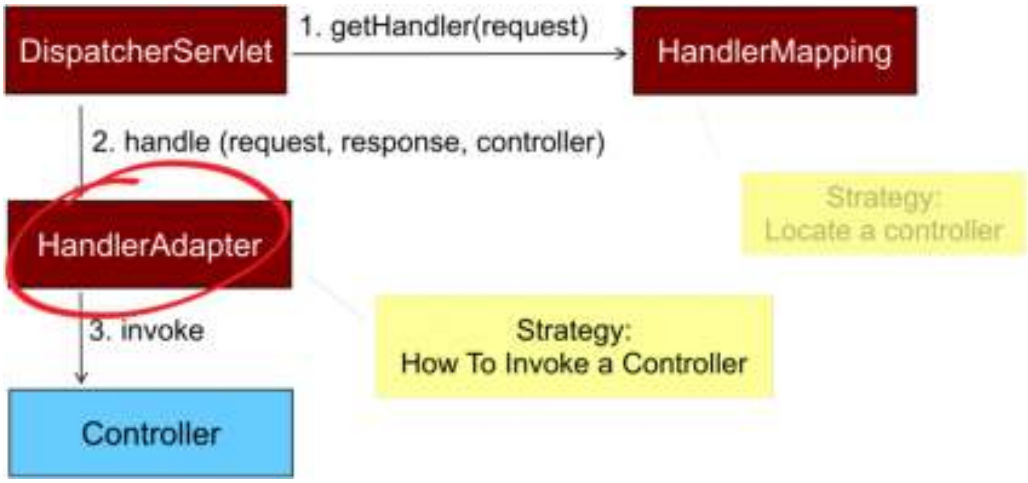
Handler Interceptors
Configuring Spring Framework (Web/MVC)

- Handler interceptors are configured on the level of the `HandlerMapping`

XML

```
<bean class="...DefaultAnnotationHandlerMapping">
    <property name="interceptors">
        <list>
            <bean class="...AuditInterceptor" />
            <bean class="...PerformanceInterceptor" />
        </list>
    </property>
</bean>
```

Handler Adapters
Configuring Spring Framework (Web/MVC)



It's all about invoking a controller.

Handler Adapters
Configuring Spring Framework (Web/MVC)

- `DispatcherServlet` invokes controllers through a `HandlerAdatper`
- To support many types of controllers, Spring MVC does not require the controller to implement a specific method or interface
- Simply create a new adapter and register it, to support a new type of controller
- Built-in adapters to be discussed:
  - `AnnotationMethodHandlerAdapter`
  - `RequestMethodHandlerAdapter`

Handler Adapters
Configuring Spring Framework (Web/MVC)

`AnnotationMethodHandlerAdapter`

- Adapts calls to `@RequestMapping` methods
- Enables flexible method signatures:
  - Introspect required input arguments
  - Interprets output/return values
- Configured by default as defined in `DispatcherServlet.properties`

Handler Adapters
Configuring Spring Framework (Web/MVC)

`RequestMethodHandlerAdapter`

- Adapts calls to `@RequestMapping` methods
  - Part of new `HandlerMethod` abstraction (in Spring MVC 3.1)
- Enables even more flexible method signatures
  - Custom argument and return types supported
- Other enhancements:
  - `@PathVariable`s automatically added to the `Model`
  - Supports parameterized URI template on redirect strings
  - Addition of consumes/produces argument to `@RequestMapping` (better REST support)
  - Now the default with `<mvc:annotation-driven>` (in Spring MVC 3.1)

Better REST support through consumes/produces attribute was introduced in Spring MVC 3.1.

Configuration Options (1)
Spring Framework (Web/MVC)

- Spring MVC infrastructure beans
- URL mappings
- Handler interceptors and handler adapters
- **Exception resolvers**
- Message source

Exception Resolvers
Configuring Spring Framework (Web/MVC)

- Controller execution can throws exceptions
- `HandlerExceptionResolver` — a Spring MVC strategy to handle exceptions thrown by controllers
- Prepares a model and selects an error view
- Multiple handlers are supported
  - Chain using their `order` property
  - Defaults: `AnnotationMethodHandlerExceptionResolver`, `ResponseStatusExceptionResolver`, and `DefaultHandlerExceptionResolver`

Exception Resolvers
Configuring Spring Framework (Web/MVC)

But wait! How are exceptions handled in a plain-vanilla Servlet environment?

Exception Resolvers
Configuring Spring Framework (Web/MVC)

JAVA

```java
public interface HandlerExceptionResolver {
    /**
     * Try to resolve the given exception that got thrown during on handler execution,
     * returning a ModelAndView that represents a specific error page if appropriate.
     * The returned ModelAndView may be {@linkplain ModelAndView#isEmpty() empty}
     * to indicate that the exception has been resolved successfully but that no view
     * should be rendered, for instance by setting a status code.
     * ...
     * @return a corresponding ModelAndView to forward to,
     * or {@code null} for default processing
     */
    ModelAndView resolveException(
            HttpServletRequest request, HttpServletResponse response,
            Object handler, Exception ex);
}
```

Exception Resolvers
Configuring Spring Framework (Web/MVC)

`SimpleMappingExceptionResolver`

- a `HandlerExceptionResolver` implementation
- maps exception class names to view names
- adds "exception" model attribute
- logs a message

Exception Resolvers
Configuring Spring Framework (Web/MVC)

XML

```xml
<bean class="...SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <entry key="DataAccessException" value="databaseError" />
        <entry key="InvalidCreditCardException" value="creditCardError" />
    </property>
    <property name="defaultStatusCode" value="500" />
    <property name="defaultErrorView" value="error" />
</bean>
```

Exception Resolvers
Configuring Spring Framework (Web/MVC)

`CustomExceptionResolver`

JAVA

```java
public class CustomExceptionResolver
        extends SimpleMappingExceptionResolver {
    protected String buildLogMessage(
            Exception e, HttpServletRequest request) {
        return "Custom log message...";
    }
    protected ModelAndView getModelAndView(
            String viewName, Exception e) {
        ModelAndView mav = super.getModelAndView(viewName, e);
        // add model attributes for the error view
        return mav;
    }
}
```


Exception Resolvers
Configuring Spring Framework (Web/MVC)

Listed below are *some* of the exceptions handled by `DefaultHandlerExceptionResolver` and the corresponding status codes:

Exception	Http Code
NoSuchRequestHandlingMethodException	404 (Not Found)
HttpRequestMethodNotSupportedException	405 (Method Not Allowed)
HttpMediaTypeNotSupportedException	415 (Unsupported Media Type)
MissingServletRequestParameterException	400 (Bad Request)
ServletRequestBindingException	400 (Bad Request)
ConversionNotSupportedException	500 (Internal Server Error)
TypeMismatchException	400 (Bad Request)
MethodArgumentNotValidException	400 (Bad Request)
MissingServletRequestPartException	400 (Bad Request)

Exception Resolvers
Configuring Spring Framework (Web/MVC)

`@ExceptionHandler`

- Allows tighter control of exception handling within a single controller
- Annotated methods called automatically when controller methods throw exception (only applicable to *current* controller)
- Method signatures support several parameters and return type:
  - `HttpServletRequest`, `HttpServletResponse`, `Exception`, etc.
  - `ModelAndView`, `String`, etc.

Exception Resolvers
Configuring Spring Framework (Web/MVC)

`@ExceptionHandler` example

JAVA

```java
@Controller
public class AccountController {
    // Other methods not shown for brevity
    @ExceptionHandler
    public String handleException(DataAccessException e) {
        return "databaseError";
    }
}
```

Configuration Options (1)
Spring Framework (Web/MVC)

- Spring MVC infrastructure beans
- URL mappings
- Handler interceptors and handler adapters
- Exception resolvers
- **Message source**

Message Source
Configuring Spring Framework (Web/MVC)

- Views often need internationalization
  - Java uses resource bundles (`*.properties`) to support i18n
- Spring MVC makes resource bundle properties accessible within views

Message Source
Configuring Spring Framework (Web/MVC)

XML

```
<bean id="messageSource"
    class="...ResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>/WEB-INF/messages/account</value>
        </list>
    </property>
</bean>
```

The above looks for `account*.properties` under `/WEB-INF/messages` (e.g. `account.properties`, `account_es.properties`).

PROPERTIES

```
Account.name=Account
Account.number=Number
```

PROPERTIES

```
Account.name=La cuenta
Account.number=Numero
```


Message Source
Configuring Spring Framework (Web/MVC)

`ReloadableResourceBundleMessageSource`

XML

```
<bean id="messageSource"
    class="...ReloadableResourceBundleMessageSource">
    <property name="basenames">...</property>
    <property name="cacheSeconds" value="${msgReloadSeconds}" />
</bean>
<context:property-placeholder location="/WEB-INF/config.properties" />
```

The above will reload after cache-seconds:

- -1 (default; never reload),
- 0 (always reload; never in production!)

Message Source
Configuring Spring Framework (Web/MVC)

- In JSTL views, use format tags

JSP

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<fmt:message key="Account.name" />
<fmt:message key="Account.number" />
```

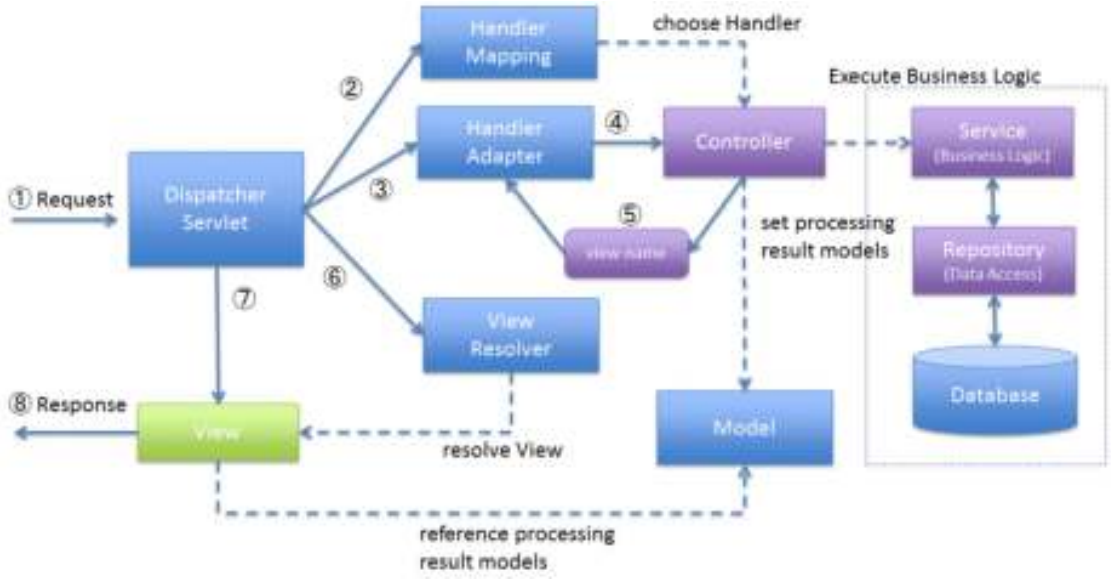
- Also integrated with the Spring `<form>` tag library

Configuration Options (1)
Spring Framework (Web/MVC)

- Spring MVC infrastructure beans
- URL mappings
- Handler interceptors and handler adapters
- Exception resolvers
- Message source

Configuration Options (1)

Spring Framework (Web/MVC)



Source: https://github.com/terasolunaorg

Pop Quiz

Configuring Spring Framework (Web/MVC)

The `DispatcherServlet` (in Spring Web/MVC) configures infrastructure beans like `HandlerMapping`, `HandlerAdapter`, and `ViewResolver`.

Note that the infrastructure beans are *strategy* interfaces. In other words, `HandlerMapping`, `HandlerAdapter`, and `ViewResolver` are interfaces.

How does the `DispatcherServlet` determine the *default* set of implementation classes to use to configure its infrastructure beans?

Pop Quiz
Configuring Spring Framework (Web/MVC)

What is the difference between `HandlerMapping` and `HandlerAdapter`?

Pop Quiz
Configuring Spring Framework (Web/MVC)

What is the difference between configuring a `DispatcherServlet` with and without `<mvc:annotation-driven />`?

Pop Quiz

Configuring Spring Framework (Web/MVC)

The `DispatcherServlet` (in Spring Web/MVC) configures a *default* set of implementation classes for infrastructure beans like `HandlerMapping`, `HandlerAdapter`, and `ViewResolver`.

What happens when a Spring MVC infrastructure bean (e.g. `ViewResolver`, `HandlerExceptionResolver`) is *already* configured?

Managing Layouts

Spring Framework (Web/MVC)

Managing Layouts
Spring Framework (Web/MVC)

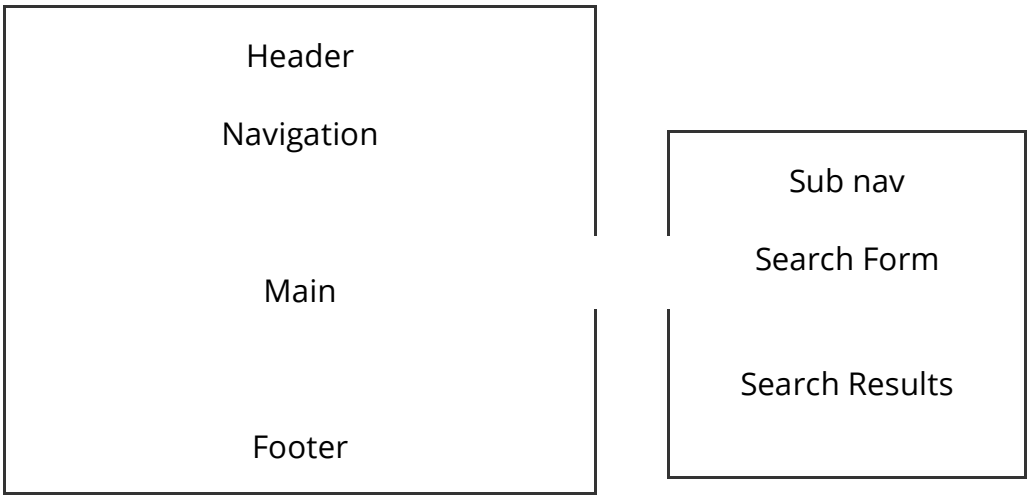
- Page layout and structure
- Creating reusable templates with Apache Tiles
- Configuring Apache Tiles in Spring MVC
- *Lab*
- Creating reusable templates with SiteMesh
- Configuring SiteMesh in Spring MVC
- *Lab*

Page Layout and Structure
Managing Layouts with Spring Framework (Web/MVC)

- Application pages usually have common elements
  - header, footer, navigation, main content
- Common elements are usually arranged the same way on every page
  - `<jsp:inlude>` affords a degree of re-use (header.jsp, footer.jsp, etc.), but must be repeated everywhere. And changing the layout will require changing every page that includes common elements.
  - Solution: create a *template* that holds common elements of a page, and leave placeholders where dynamic content is needed. Changing this *template*, changes the layout of common elements.

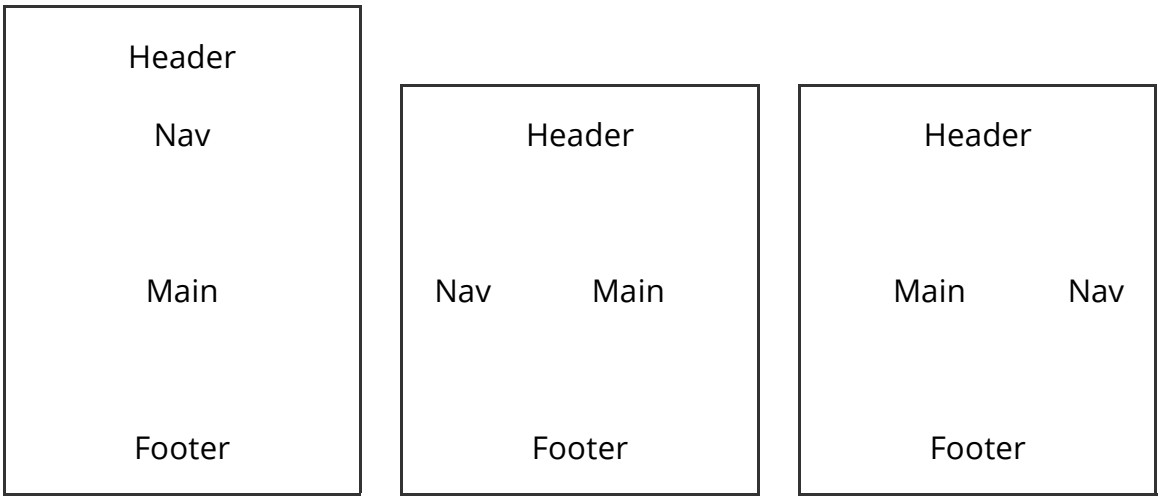
Page Layout and Structure

Managing Layouts with Spring Framework (Web/MVC)



Page Layout and Structure

Managing Layouts with Spring Framework (Web/MVC)



Templates with Apache Tiles
Managing Layouts with Spring Framework (Web/MVC)

- A web-page layout templating framework
- An independent Apache project (formerly part of Struts)
- Implements a *composite* view pattern
- Allows re-usable layout templates and defining fragments (tiles) for composing views

Templates with Apache Tiles
Managing Layouts with Spring Framework (Web/MVC)

- Uses an external XML file to define the structure/layout of a page
- A Tiles configuration file contains one or more Tiles `<definition>`s
- Tiles `<definition>`s are reusable fragments consisting of a template and attributes

Templates with Apache Tiles
Managing Layouts with Spring Framework (Web/MVC)

XML

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
       "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">

<tiles-definitions>
    <definition name="standard" template="/WEB-INF/layouts/standard.jsp" />
    <definition name="admin" template="/WEB-INF/layouts/admin.jsp" />
    <definition name="accounts/list" extends="standard">
        <put-attribute name="title" value="accounts.list.title" />
        <put-attribute name="main" value="/WEB-INF/views/accounts/list.jsp" />
    </definition>
</tiles-definitions>
```


Templates with Apache Tiles
Managing Layouts with Spring Framework (Web/MVC)

JSP

```jsp
<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles" %>
```

/WEB-INF/layouts/standard.jsp

HTML

```html
<head>
    <title><tiles:insertAttribute name="title" /></title>
</head>
<body>
    <header>...</header>
    <div id="main">
        <tiles:insertAttribute name="main" />
    </div>
    <footer>...</footer>
</body>
```

Templates with Apache Tiles
Managing Layouts with Spring Framework (Web/MVC)

/WEB-INF/layouts/standard.jsp

HTML

```
<body>
    <header>...</header>
    <div id="main">
        <tiles:insertAttribute name="main" />
    </div>
    <footer>...</footer>
</body>
```

main attribute defined as /WEB-INF/views/accounts/list.jsp

/WEB-INF/views/accounts/list.jsp

HTML

```
<table><c:forEach var="account" items="${accountList}">
    <tr>...</tr>
</c:forEach></table>
```


Templates with Apache Tiles
Managing Layouts with Spring Framework (Web/MVC)

Tiles Attributes

- Placeholders for dynamic data
- An attribute value can be:
  - a simple string value
  - a template (if it starts with "/")
  - another tiles definition

XML

```
<tiles-definitions>
    <definition name="standard" template="…" />
    <definition name="welcomeBody" template="…" />
    <definition name="welcome" extends="standard">
        <put-attribute name="main" value="welcomeBody" />
    </definition>
</tiles-definitions>
```

Templates with Apache Tiles
Managing Layouts with Spring Framework (Web/MVC)

Tiles Attributes (continued)

- Two approaches to insert attributes in JSP
  - `<tiles:insertAttribute>` (renders to JSP output)
  - `<tiles:importAttribute>` (adds attribute to model)

JSP

```
<tiles:importAttribute name="navigationTab" />
<c:if test="${navigationTab eq 'accounts'}"><a class="active">...</a></c:if>
```

XML

```
<tiles-definitions>...
    <definition name="accounts/list" extends="standard">
        <put-attribute name="navigationTab" value="accounts" />
    </definition>
</tiles-definitions>
```


Configuring Apache Tiles
Managing Layouts with Spring Framework (Web/MVC)

To configure Tiles with Spring:

1. Provide one or more Tiles configuration files to the `TilesConfigurer`
2. Switch from `InternalResourceViewResolver` to a `TilesViewResolver`

Configuring Apache Tiles
Managing Layouts with Spring Framework (Web/MVC)

Provide one or more Tiles configuration files to the `TilesConfigurer`

XML

```xml
<bean class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/layouts/tiles.xml</value>
            <value>/WEB-INF/layouts/accounts/tiles.xml</value>
            ...
        </list>
    </property>
</bean>
```


Configuring Apache Tiles
Managing Layouts with Spring Framework (Web/MVC)

Switch from `InternalResourceViewResolver` to a `TilesViewResolver`

XML

```xml
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    ...
</bean>
<bean class="org.springframework.web.servlet.view.tiles3.TilesViewResolver" />
```

Configuration Options (2)

Spring Framework (Web/MVC)

Configuration Options (2)

Spring Framework (Web/MVC)

- XML configuration and the `<mvc>` namespace
- Using Java-based Configuration
- Running in a Servlet 3 environment without `web.xml`
- *Lab*

<mvc> Namespace

Configuring Spring Framework (Web/MVC)

XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven/>

</beans>
```


<mvc> Namespace

Configuring Spring Framework (Web/MVC)

- Serve static resources when `DispatcherServlet` is mapped to root path ("/")
- Views without a controller
- Handler interceptors

<mvc> Namespace

Configuring Spring Framework (Web/MVC)

- As of Spring MVC 3.0.4, the `DispatcherServlet` can be mapped to `/`
- An additional element must be defined to pass requests to servlet container
  - used to serve static assets (e.g. images, JavaScript files, etc.)
  - only required when mapped to `/`

XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="...">

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

</beans>
```


<mvc> Namespace

Configuring Spring Framework (Web/MVC)

- `<mvc:resources>`
  - Provides a convenient way to serve static resources when `DispatcherServlet` is mapped to "/"
  - Static resources from locations other than the web application root, including locations on the classpath.

XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc" ...>
    <mvc:resources mapping="/js/**" location="/js/" />
    <mvc:resources mapping="/css/**" location="/css/" />
    <mvc:resources mapping="/images/**" location="/images/" />
    <mvc:resources mapping="/webjars/**"
        location="classpath:/META-INF/resources/webjars" />
</beans>
```

<mvc> Namespace

Configuring Spring Framework (Web/MVC)

XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc" ...>
    <mvc:resources mapping="/js/**" location="/js/" />
    <mvc:resources mapping="/css/**" location="/css/" />
    <mvc:resources mapping="/images/**" location="/images/" />
    <mvc:resources mapping="/webjars/**"
        location="classpath:/META-INF/resources/webjars/" />
</beans>
```

JSP

```
<head>...
    <spring:url value="/webjars/bootstrap/x.y.z/css/bootstrap.min.css"
                var="bootstrapCss"/>
    <link href="${bootstrapCss}" rel="stylesheet"/>
</head>
```


<mvc> Namespace

Configuring Spring Framework (Web/MVC)

- Multiple resource locations may be specified using a comma-separated list of values.
- The locations specified will be checked in the specified order for the presence of the resource for any given request.

XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc" ...>

    <mvc:resources mapping="/css/**"
        location="/css/, classpath:/META-INF/resources/webjars/" />

</beans>
```

<mvc> Namespace

Configuring Spring Framework (Web/MVC)

```xml
<mvc:resources mapping="/images/**" location="/images/"
    cache-period="3600" />
```

XML

And in Spring MVC 4.2, the `<cache-control>` element was introduced.

```xml
<mvc:resources mapping="/images/**" location="/images/">
    <mvc:cache-control max-age="3600" cache-public="true"/>
</mvc:resources>
```

XML

<mvc> Namespace

Configuring Spring Framework (Web/MVC)

- Some views don't need a controller
  - We just want to render a view
  - Use `<mvc:view-controller>` for this
- Specify URL and view name in configuration

```xml
<mvc:view-controller path="/" view-name="home"/>
<mvc:view-controller path="/login" view-name="auth/login"/>
```

XML

<mvc> Namespace

Configuring Spring Framework (Web/MVC)

- Interceptors can be specified using `<mvc>` namespace

XML

```xml
<!-- Applied to all handler mappings -->
<mvc:interceptors>
    <bean class="...AuditInterceptor" />
    <bean class="...PerformanceInterceptor" />
</mvc:interceptors>

<!-- Applied to a subset handler mappings -->
<mvc:interceptors>
    <mvc:mapping path="/secure/*" />
    <bean class="...SecurityInterceptor" />
</mvc:interceptors>
```


Java-based Configuration

Configuring Spring Framework (Web/MVC)

- Java-based configuration can be loaded from `web.xml`
  - alternative to XML-based configuration

XML

```xml
<context-param>
  <param-name>contextClass</param-name>
  <param-value>
    org.springframework.web.context.support.AnnotationConfigWebApplicationContext
  </param-value>
</context-param>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <!-- classpath:..../application-config.xml -->
  <param-value>com.acme.app.AppConfig</param-value>
</context-param>
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

JAVA

```java
package com.acme.app;
@Configuration
public class AppConfig {
  // services, repositories,
  // database, etc.
}
```

Java-based Configuration
Configuring Spring Framework (Web/MVC)

XML

```
<servlet>
  <servlet-name>example</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextClass</param-name>
    <param-value>
    org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
  </init-param>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <!-- /WEB-INF/spring/web-config.xml -->
    <param-value>com.acme.web.MyWebConfig</param-value>
  </init-param>...
</servlet>
<!-- servlet-mappings -->
```

JAVA

```
package com.acme.web;
@Configuration
@EnableWebMvc
public class MyWebConfig {
  // controllers, views, etc.
}
```


Java-based Configuration
Configuring Spring Framework (Web/MVC)

- `@EnableWebMvc` equivalent to `<mvc:annotation-driven>`

JAVA

```
@EnableWebMvc
@Configuration
public class MyWebConfig {
    // Declare @Bean methods (e.g. controllers)
}
```

Java-based Configuration
Configuring Spring Framework (Web/MVC)

```
<context:component-scan base-package="..." />
```

JAVA

```java
@EnableWebMvc
@Configuration
@ComponentScan(basePackages = "com.acme")
public class MyWebConfig {
    // Declare @Bean methods (e.g. controllers)
}
```


Java-based Configuration
Configuring Spring Framework (Web/MVC)

JAVA

```java
@EnableWebMvc
@Configuration
public class MyWebConfig extends WebMvcConfigurerAdapter {
    // Declare @Bean methods (e.g. controllers)
    // Override some methods for infrastructure beans
}
```

Java-based Configuration
Configuring Spring Framework (Web/MVC)

- Serve static resources when `DispatcherServlet` is mapped to root path ("/")
- Views without a controller
- Handler interceptors

Java-based Configuration
Configuring Spring Framework (Web/MVC)

- Serve static resources when `DispatcherServlet` is mapped to root path ("/")

JAVA

```java
@EnableWebMvc
@Configuration
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/css/**")
                .addResourceLocations(
                    "/css/", "classpath:/META-INF/resources/webjars/");
    }

}
```

Java-based Configuration
Configuring Spring Framework (Web/MVC)

- Some views don't need a controller
  - We just want to render a view

XML

```
<mvc:view-controller mapping="/" view-name="home" />
```

JAVA

```
@EnableWebMvc
@Configuration
public class MyWebConfig extends WebMvcConfigurerAdapter {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
    }
}
```


Java-based Configuration
Configuring Spring Framework (Web/MVC)

XML

```
<mvc:interceptors>
    <bean class="...AuditInterceptor" />
</mvc:interceptors>
<mvc:interceptors>
    <mvc:mapping path="/secure/*" />
    <bean class="...SecurityInterceptor" />
</mvc:interceptors>
```

JAVA

```
@EnableWebMvc
@Configuration
public class MyWebConfig extends WebMvcConfigurerAdapter {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(
                new AuditInterceptor());
        registry.addInterceptor(
                new SecurityInterceptor()).addPathPatterns("/secure/*");
    }
}
```

Servlet 3.x Environment without `web.xml`

Configuring Spring Framework (Web/MVC)

- Servlet 3.0 provides a Java-based way to configure web applications.
- Web applications that wish to use this approach must implement `ServletContainerInitializer` and declare the implementation of this in `META-INF/services`.
- Spring has it's own implementation of the `ServletContainerInitializer`, the `SpringServletContainerInitializer`
  - delegates to a Spring class for initializing a web application: `WebApplicationInitializer`.

Servlet 3.x Environment without `web.xml`

Configuring Spring Framework (Web/MVC)

Implementations of `WebApplicationInitializer` include:

- `AbstractContextLoaderInitializer`
  - registers a `ContextLoaderListener`
- `AbstractDispatcherServletInitializer`
  - extends `AbstractContextLoaderInitializer` and registers a `DispatcherServlet`
- `AbstractAnnotationConfigDispatcherServletInitializer`
  - extends `AbstractDispatcherServletInitializer` and expects Java-based configuration classes

Servlet 3.x Environment without `web.xml`

Configuring Spring Framework (Web/MVC)

Load XML-based root application context configuration.

JAVA

```java
public class SampleInitializer extends AbstractContextLoaderInitializer {
    @Override
    protected WebApplicationContext createRootApplicationContext() {
        XmlWebApplicationContext webApplicationContext =
                new XmlWebApplicationContext();
        webApplicationContext.setConfigLocation("....xml");
        // webApplicationContext.setConfigLocations("...1.xml", "...2.xml");
        return webApplicationContext; // or null if there's none
    }
}
```


Servlet 3.x Environment without `web.xml`

Configuring Spring Framework (Web/MVC)

Or, load Java-based root application context configuration.

JAVA

```java
public class SampleInitializer extends AbstractContextLoaderInitializer {
    @Override
    protected WebApplicationContext createRootApplicationContext() {
        AnnotationConfigWebApplicationContext webApplicationContext =
            new AnnotationConfigWebApplicationContext();
        webApplicationContext.register(
            MyAppConfig.class
            /*, and other @Configuration annotated classes */);
        return webApplicationContext; // or null if there's none
    }
}
```

Warning: Not to be confused with `AnnotationConfigApplicationContext`.

Servlet 3.x Environment without `web.xml`

Configuring Spring Framework (Web/MVC)

JAVA

```java
public class SampleInitializer extends AbstractDispatcherServletInitializer {
    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return ...; // or null if there's none
    }
    @Override
    protected WebApplicationContext createServletApplicationContext() {
        return ...; // mostly controllers, custom views, etc.
    }
    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

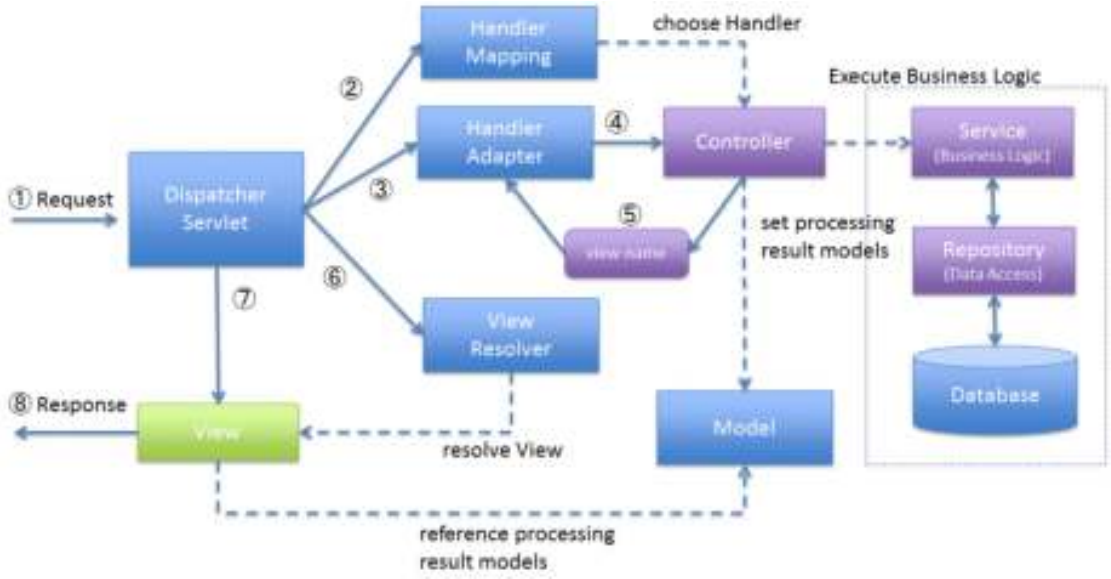

Servlet 3.x Environment without `web.xml`

Configuring Spring Framework (Web/MVC)

JAVA

```java
public class SampleInitializer
        extends AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { MyAppConfig.class };
    }
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { MyWebAppConfig.class };
    }
    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

Configuration Options (2)
Spring Framework (Web/MVC)



Source: https://github.com/terasolunaorg

Configuration Options (2)
Spring Framework (Web/MVC)

Lab

Using Views

Spring Framework (Web/MVC)

Using Views

Spring Framework (Web/MVC)

- Views and view resolvers
- Setting up a view resolver chain
- Alternating views and Content Negotiation
- JSON and XML Views
- *Lab*

Views and View Resolvers
Using Views in Spring Framework (Web/MVC)

Views

- URL-based views
  - `JstlView`
- Content-generating views
  - `AbstractExcelView`, `AbstractPdfView`

View Resolvers

- `UrlBasedViewResolver`
- `BeanNameViewResolver`
- `XmlViewResolver`

Views and View Resolvers
Using Views in Spring Framework (Web/MVC)

URL-based views

- all views are *recommended* to be stored under `/WEB-INF`
  - hidden from direct browser access
  - rendering requires a model
- the logical view name matches to a file
  - e.g. JSP, FreeMarker template, XSLT, etc.
- the URL may not be a file (e.g. Tiles definition)

Views and View Resolvers
Using Views in Spring Framework (Web/MVC)

`JstlView`

- exposes the model attributes as request attributes
- adds attributes for JSTL format/message tags
- forwards to a JSP page

JAVA

```java
JstlView view = new JstlView("/WEB-INF/views/accounts/show.jsp");
...
view.render(model, request, response);
```


Views and View Resolvers
Using Views in Spring Framework (Web/MVC)

`InternalResourceViewResolver`

- Interprets view names as JSP paths
- Creates instances of `JstlView`

XML

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

Views and View Resolvers
Using Views in Spring Framework (Web/MVC)

XML

```xml
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
    p:prefix="/WEB-INF/views/" p:suffix=".jsp" />
```

JAVA

```java
@EnableWebMvc
@Configuration
public class MyWebConfig extends WebMvcConfigurerAdapter {
    @Bean
    public InternalResourceViewResolver getInternalResourceViewResolver() {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }
}
```


Views and View Resolvers
Using Views in Spring Framework (Web/MVC)

XML

```xml
<mvc:view-resolvers>
    <mvc:jsp prefix="/WEB-INF/views" suffix=".jsp" />
</mvc:view-resolvers>
```

JAVA

```java
@EnableWebMvc
@Configuration
public class MyWebConfig extends WebMvcConfigurerAdapter {
    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.jsp("/WEB-INF/views", ".jsp");
    }
}
```

The `mvc:view-resolvers` and `configureViewResolvers()` were made available in Spring MVC 4.1.

Views and View Resolvers
Using Views in Spring Framework (Web/MVC)

`UrlBasedViewResolver`

- Interprets a view name as a URL
- Many subclasses
  - `InternalResourceViewResolver` (JSP/Servlets)
  - `FreeMarkerViewResolver`
  - `XsltViewResolver`
  - etc.
- Supports "`redirect:`" prefix

Views and View Resolvers
Using Views in Spring Framework (Web/MVC)

Content-generating views

- extends from a base class
  - `AbstractExcelView`, `AbstractPdfView`, etc.
- creates view content using an API
  - POI, iText
- the base class writes generated content to the response stream

Views and View Resolvers
Using Views in Spring Framework (Web/MVC)

JAVA
```
AccountsExcelView view = new AccountsExcelView();
view.render(model, request, response);
```

JAVA
```
public class AccountsExcelView extends AbstractExcelView {
    @Override
    protected void buildExcelDocument(
            Map model, HSSFWorkbook workbook,
            HttpServletRequest request,
            HttpServletResponse response) {
        Account account = ((List<Account>)
                model.get("accounts")).get(0);
        HSSFSheet sheet = workbook.getSheetAt(0);
        ...
    }
}
```


Views and View Resolvers
Using Views in Spring Framework (Web/MVC)

`BeanNameViewResolver`

- interprets a view name as a bean name

XML
```
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver" />

<bean name="accounts/list.xls"
    class="...AccountsExcelView" />
```

JAVA
```
    @RequestMapping("/accounts")
    public String list(Model model) {
        ...
        return "accounts/list.xls";
    }
```

Views and View Resolvers
Using Views in Spring Framework (Web/MVC)

XML

```xml
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver" />
```

JAVA

```java
@EnableWebMvc
@Configuration
public class MyWebConfig extends WebMvcConfigurerAdapter {
    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.beanName();
    }
}
```


Views and View Resolvers
Using Views in Spring Framework (Web/MVC)

`XmlViewResolver`

- picks up view beans from a given XML configuration file
  - works just like `BeanNameViewResolver`, but matches only `View`s defined in the given XML configuration file
- reduces bean-configuration file clutter
  - keep view beans separate

XML

```xml
<bean class="org.springframework.web.servlet.view.XmlViewResolver">
    <property name="location" value="/WEB-INF/spring/views.xml" />
</bean>
```

Views and View Resolvers
Using Views in Spring Framework (Web/MVC)

XML

```xml
<bean class="org.springframework.web.servlet.view.XmlViewResolver"
    p:location="/WEB-INF/spring/views.xml" />
```

JAVA

```java
@EnableWebMvc
@Configuration
public class MyWebConfig extends WebMvcConfigurerAdapter {
    @Autowired
    private ApplicationContext applicationContext;
    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        XmlViewResolver xmlViewResolver = new XmlViewResolver();
        xmlViewResolver.setLocation(
                applicationContext.getResource("/WEB-INF/spring/views.xml"));
        registry.viewResolver(xmlViewResolver);
    }
}
```


Using Views
Spring Framework (Web/MVC)

- Views and view resolvers
- **Setting up a view resolver chain**
- Alternating views and Content Negotiation
- JSON and XML Views
- *Lab*

Setting Up `ViewResolver` Chains

Using Views in Spring Framework (Web/MVC)

`ViewResolver` Chain

- the `DispatcherServlet` discovers `ViewResolver` beans by type
- multiple `ViewResolver` beans are possible
  - each is given a "chance" to match
  - the order can be specified
- the first resolver to return a non-`null` `View` *wins*

Setting Up `ViewResolver` Chains

Using Views in Spring Framework (Web/MVC)

XML

```xml
<bean class="...web.servlet.view.BeanNameViewResolver">
    <property name="order" value="1" />
</bean>
<bean class="...web.servlet.view.InternalResourceViewResolver">
    <property name="order" value="2" />
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
</bean>
```

Setting Up `ViewResolver` Chains

Using Views in Spring Framework (Web/MVC)

XML

```xml
<mvc:view-resolvers>
    <mvc:bean-name />
    <mvc:jsp prefix="/WEB-INF/views" suffix=".jsp" />
</mvc:view-resolvers>
```

JAVA

```java
@EnableWebMvc
@Configuration
public class MyWebConfig extends WebMvcConfigurerAdapter {
    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.beanName();
        registry.jsp("/WEB-INF/views", ".jsp");
    }
}
```


Setting Up `ViewResolver` Chains

Using Views in Spring Framework (Web/MVC)

`ViewResolver` Chain Order

- all `ViewResolver` beans implement `Ordered`
- some `UrlBasedViewResolver`s can be anywhere in the chain
  - depends on the view type served
  - Tiles, Velocity, FreeMarker view resolvers check for files and return `null` if they don't exist
- others are better placed near the end of chain (i.e. last in ordering)
  - JSTL/JSP and XSLT view resolvers *always forwards* rather than returning `null`
  - these resolvers can be subclassed to do a resource check

Alternating Views and Content Negotiation
Using Views in Spring Framework (Web/MVC)

- Previous techniques work well when each resource is associated with *only one* view/media type
- Clients may request different content types for the same resource
  - via filename extension, HTTP request `Accept` header, request parameter, etc.
- The process of determining which type to render is known as **Content Type Negotiation**

Alternating Views and Content Negotiation
Using Views in Spring Framework (Web/MVC)

Content Type Negotiation Examples

- GET `/accounts/list` (by default, returns HTML)
- GET `/accounts/list.htm`
- GET `/accounts/list.xls`
- GET `/accounts/list?format=text/html`
- GET `/accounts/list?format=application/vnd.ms-excel`

Alternating Views and Content Negotiation
Using Views in Spring Framework (Web/MVC)

Options to handle **Content Type Negotiation**

1. Separate controller methods for each view/media type
2. Controller logic determines (e.g. via filename extension, HTTP `Accept` header) view/media type, and selects view name
3. Special view resolver

Alternating Views and Content Negotiation
Using Views in Spring Framework (Web/MVC)

Option #1: Separate controller methods

JAVA

```java
@RequestMapping("/accounts.htm")
public String listHtml(Model model) {
    ...
    return "accounts/list";
}

@RequestMapping("/accounts.xls")
public String listExcel(Model model) {
    ...
    return "accounts/list.xls";
}
```

Alternating Views and Content Negotiation
Using Views in Spring Framework (Web/MVC)

Option #2: Controller logic

JAVA

```java
@RequestMapping("/accounts")
public String list(Model model) {
    ...
    // if request URL ends with ".xls"
        return "accounts/list.xls";
    // otherwise
        return "accounts/list";
}
```


Alternating Views and Content Negotiation
Using Views in Spring Framework (Web/MVC)

Option #3: Special view resolver

- `ContentNegotiatingViewResolver` (*CNVR for short*)
  - Introduced in Spring 3.0
- Does not do view resolution itself, delegates to other view resolvers
- Configured to use the following to determine desired content type:
  - filename extensions
  - request parameters
  - HTTP `Accept` header

Alternating Views and Content Negotiation
Using Views in Spring Framework (Web/MVC)

`ContentNegotiatingViewResolver` (*CNVR for short*) (continued)

- Each `View` class is associated with a content type

JAVA

```
public interface View { ...
    String getContentType();
    void render(… model, … request, … response) throws Exception;
}
```

- Many views already define content types
  - `AbstractPdfView` → application/pdf
  - `AbstractExcelView` → application/vnd.ms-excel
  - `JstlView` → text/html
  - `MappingJacksonJsonView` → application/json

Alternating Views and Content Negotiation
Using Views in Spring Framework (Web/MVC)

XML

```
<bean class="...web.servlet.view.ContentNegotiatingViewResolver">
    <!-- All configuration is now done by the manager - since Spring V3.2 -->
    <property name="contentNegotiationManager" ref="cnManager"/>
</bean>

<!--
Setup a simple strategy:
1. Only path extension is taken into account, Accept headers are ignored.
2. Return HTML by default when not sure.
 -->
<bean id="cnManager" class="...web.accept.ContentNegotiationManagerFactoryBean">
    <property name="ignoreAcceptHeader" value="true"/>
    <property name="defaultContentType" value="text/html" />
</bean>
```

Alternating Views and Content Negotiation
Using Views in Spring Framework (Web/MVC)

XML

```xml
<mvc:annotation-driven content-negotiation-manager="cnManager" />

<bean id="cnManager" class="...web.accept.ContentNegotiationManagerFactoryBean">
    <property name="ignoreAcceptHeader" value="true"/>
    <property name="defaultContentType" value="text/html" />
</bean>

<mvc:view-resolvers>
    <mvc:content-negotiation />
    ... <!-- mvc:jsp -->
</mvc:view-resolvers>
```


Alternating Views and Content Negotiation
Using Views in Spring Framework (Web/MVC)

JAVA

```java
@EnableWebMvc
@Configuration
public class MyWebConfig extends WebMvcConfigurerAdapter { ...
    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.enableContentNegotiation();
        ... // registry.jsp();
    }

    @Override
    public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
        configurer.ignoreAcceptHeader(true);
        configurer.defaultContentType(MediaType.TEXT_HTML);
    }
}
```

Alternating Views and Content Negotiation

Using Views in Spring Framework (Web/MVC)

By *default*, content negotiation is checked in this order:

1. Path extension (suffix) in the URL (`../accounts/lists`*.xls*)
2. A URL parameter (`.../accounts/list?`*format*`=xls`) (*disabled by default*)
3. HTTP `Accept` header property

Sometimes, this is referred to as the **PPA** strategy (**p**ath extension, then **p**arameter, then **A**ccept header).

JSON Views

Using Views in Spring Framework (Web/MVC)

Generating JSON

- Uses the `MappingJackson2JsonView`
  - converts contents of `Model` to JSON
- Contents in `Model` may need annotating
  - Tells Jackson how to convert Java objects to JSON
  - Similar to JPA and JAXB annotations
- No `ViewResolver` supplied, but simple to write one

JSON Views
Using Views in Spring Framework (Web/MVC)

- Suggested implementation
  - Always returning a `MappingJackson2JsonView`
  - Only works alongside CNVR

JAVA

```java
public class JsonViewResolver implements ViewResolver {
    @Override
    public View resolveViewName(String viewName, Locale locale) {
        return new MappingJackson2JsonView();
    }
}
```


JSON Views
Using Views in Spring Framework (Web/MVC)

- Request must contain `application/json` in HTTP `Accept` header

XML

```xml
<bean class="...web.servlet.view.ContentNegotiatingViewResolver">
    ...
</bean>
<bean class="...JsonViewResolver" />
<bean class="...web.servlet.view.InternalResourceViewResolver"
    p:prefix="/WEB-INF/views/" p:suffix=".jsp" />
```

JSON Views
Using Views in Spring Framework (Web/MVC)

XML

```
<mvc:view-resolvers>
  <mvc:content-negotiation />
  <bean class="...JsonViewResolver" />
  <mvc:jsp prefix="/WEB-INF/views/" suffix=".jsp" />
</mvc:view-resolvers>
```


JSON Views
Using Views in Spring Framework (Web/MVC)

JAVA

```
@EnableWebMvc
@Configuration
public class MyWebConfig extends WebMvcConfigurerAdapter { ...
    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.enableContentNegotiation();
        registry.viewResolver(new JsonViewResolver());
        registry.jsp("/WEB-INF/views/", ".jsp");
    }
}
```

XML Views
Using Views in Spring Framework (Web/MVC)

Generating XML

- Uses the `MarshallingView`
  - converts contents of `Model` to XML via `Marshaller` (e.g. `Jaxb2Marshaller`)
- Contents in `Model` may need annotating
  - Tells JAXB how to convert Java objects to XML
- No `ViewResolver` supplied, but simple to write one

XML Views
Using Views in Spring Framework (Web/MVC)

- Suggested implementation
  - Always returning a `MarshallingView`
  - Only works alongside CNVR

JAVA

```java
public class XmlMarshallingViewResolver implements ViewResolver {
    private Marshaller marshaller;
    @Autowired
    public XmlMarshallingViewResolver(Marshaller marshaller) {
        this.marshaller = marshaller;
    }
    @Override
    public View resolveViewName(String viewName, Locale locale) {
        return new MarshallingView(marshaller);
    }
}
```

XML Views
Using Views in Spring Framework (Web/MVC)

- Request must contain `application/xml` in HTTP `Accept` header

XML

```
<bean class="...web.servlet.view.ContentNegotiatingViewResolver">
    ...
</bean>
<oxm:jaxb2-marshaller id="marshaller" >
    <oxm:class-to-be-bound name="..."/>
    <oxm:class-to-be-bound name="..."/>
    ...
</oxm:jaxb2-marshaller>
<bean class="...XmlMarshallingViewResolver">
    <constructor-arg ref="marshaller" />
</bean>
<bean class="...web.servlet.view.InternalResourceViewResolver"
    p:prefix="/WEB-INF/views" p:suffix=".jsp" />
```


XML Views
Using Views in Spring Framework (Web/MVC)

XML

```
<mvc:view-resolvers>
    <mvc:content-negotiation />
    <bean class="...XmlMarshallingViewResolver">
        <constructor-arg ref="marshaller" />
    </bean>
    <mvc:jsp prefix="/WEB-INF/views/" suffix=".jsp" />
</mvc:view-resolvers>

<oxm:jaxb2-marshaller id="marshaller" >
    <oxm:class-to-be-bound name="..."/>
    <oxm:class-to-be-bound name="..."/>
    ...
</oxm:jaxb2-marshaller>
```

XML Views
Using Views in Spring Framework (Web/MVC)

JAVA

```java
@EnableWebMvc
@Configuration
public class MyWebConfig extends WebMvcConfigurerAdapter { ...
    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.enableContentNegotiation();
        registry.viewResolver(new XmlMarshallingViewResolver(marshaller()));
        registry.jsp("/WEB-INF/views/", ".jsp");
    }

    @Bean
    public Jaxb2Marshaller marshaller() {
        Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
        marshaller.setClassesToBeBound(new Class[] {...});
        // marshaller.setContextPath("...") or marshaller.setContextPaths(...)
        return marshaller;
    }
}
```


Using Views
Spring Framework (Web/MVC)

Lab

Pop Quiz
Using Views in Spring Framework (Web/MVC)

Why is it considered a good practice to put JSP files, which serve as views, under `/WEB-INF`?

Pop Quiz
Using Views in Spring Framework (Web/MVC)

When chaining `ViewResolvers`, why should an `InternalResourceViewResolver` need to be last (in order)?

Building HTML Forms

Spring Framework (Web/MVC)

Building HTML Forms
Spring Framework (Web/MVC)

- Overview of HTML Forms
- Form rendering
- Type conversion
- Data binding
- Form validation (using Spring and JSR 330 validation)
- Form object management
- *Lab*

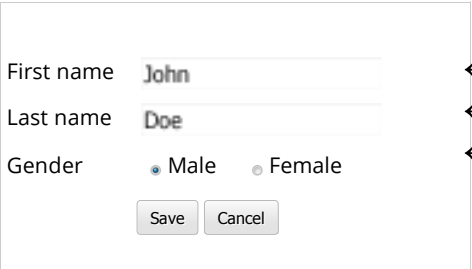
Overview of HTML Forms

HTML Forms with Spring Framework (Web/MVC)

- Central to Spring MVC is HTML form processing
- Spring MVC provides the following to support HTML form processing:
  - form tags
  - data binding
  - validation
  - error processing

Overview of HTML Forms

HTML Forms with Spring Framework (Web/MVC)



JAVA

```
public class PersonForm {
    private String firstName;
    private String lastName;
    private Gender gender;
    ...
}
```

Overview of HTML Forms

HTML Forms with Spring Framework (Web/MVC)

1. Initial `HTTP GET` to present the HTML form
2. Subsequent `HTTP POST` to submit the form
   - Apply request values to the *form object*
   - Perform validation
   - Invoke application service (e.g. to save/persist changes)
3. `POST-Redirect-GET` on success
   - Redirect to the next page, instead of rendering

Overview of HTML Forms

HTML Forms with Spring Framework (Web/MVC)

Exposing Domain Objects

- Domain objects can be used as web-layer form objects
- Potential security concern
  - Must set allowed fields for data binding
- Puts a strain on OO design of domain objects
  - Mandatory no-args constructor, getters, and setters
- Web-layer (or presentation-layer) logic likely to creep into domain layer

Overview of HTML Forms

HTML Forms with Spring Framework (Web/MVC)

Web-layer Form Objects

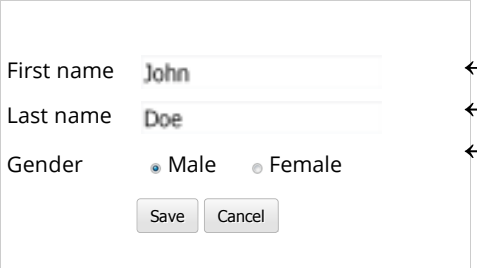
- Form pages may differ from *domain objects*
  - Data aggregated from multiple domain objects
  - UI-specific needs
- *Prefer* separate web-layer *form objects*
  - Model exactly what the web page needs
  - Encapsulate:
    - Web-layer logic
    - Validation logic
    - Logic for copying to and from the domain object

Form Rendering

HTML Forms with Spring Framework (Web/MVC)

Displaying an HTML Form

- Spring `form` tags bind the form object to the HTML `<form>`



JAVA

```
public class PersonForm {
    private String firstName;
    private String lastName;
    private Gender gender;
    ...
}
```

Form Rendering
HTML Forms with Spring Framework (Web/MVC)

Spring `form` Tags

- Custom JSP tag library
- Makes it easier to build HTML `<form>` pages
- Advantages
  - Populate HTML `<form>` fields with formatted values
  - Render field-specific error messages

Form Rendering
HTML Forms with Spring Framework (Web/MVC)

Using Spring `form` tags

- Add the `taglib` directive

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

JSP

- Use the Spring `form` tags

```
<form method="post" action="...">
</form>

<form:form method="post" action="..." modelAttribute="paymentForm">
</form:form>
```

JSP

Form Rendering
HTML Forms with Spring Framework (Web/MVC)

Spring `form:input` tag

JSP

```
<form method="post" action="...">
    <input type="text" name="amount" value="${paymentForm.amount}" />
    <!-- but amount is not formatted -->
</form>

<form:form method="post" action="..." modelAttribute="paymentForm">
    <form:input path="amount" />
    <!-- amount is formatted -->
</form:form>
```

Property path (relative to the `modelAttribute` specified on the `form` tag).

Form Rendering
HTML Forms with Spring Framework (Web/MVC)

Spring `form:select` and `form:option` tags

JSP

```
<form method="post" action="...">
    <select name="size">
        <option value="S">Small</option>
        <option value="M" selected="selected">Medium</option>
        <option value="L">Large</option>
    </select>
</form>

<form:form method="post" action="..." modelAttribute="orderForm">
    <form:select path="size">
        <form:option label="Small" value="S" />
        <form:option label="Medium" value="M" />
        <form:option label="Large" value="L" />
    </form:select>
</form:form>
```

Option dynamically selected based on the value of `orderForm.size`.

Form Rendering
HTML Forms with Spring Framework (Web/MVC)

Spring `form:select` and `form:option` tags with items

JSP

```
<form method="post" action="...">
    <select name="size">
        <!-- loop over collection of restaurants -->
    </select>
</form>

<form:form method="post" action="..." modelAttribute="...">
    <form:select path="restaurantId" items="${restaurants}"
        itemLabel="name" itemValue="id" />
</form:form>
```

JAVA

```
public class Restaurant {
    private String id;
    private String name;
}
```


Form Rendering
HTML Forms with Spring Framework (Web/MVC)

Spring `form:select` and `form:options` tags with list of options

JSP

```
<form:form method="post" action="..." modelAttribute="...">
    <form:select path="restaurantId">
        <form:option value="--">Please select</form:option>
        <form:options items="${restaurants}"
            itemLabel="name" itemValue="id" />
    </form:select>
</form:form>
```

JAVA

```
public class Restaurant {
    private String id;
    private String name;
}
```

Form Rendering
HTML Forms with Spring Framework (Web/MVC)

- All tags have equivalent HTML form fields
  - checkbox, checkboxes
  - hidden
  - label
  - password
  - radiobutton, radiobuttons
  - textarea
- The `<form:errors>` tag displays error messages (to be discussed in the succeeding sections)

Type Conversion
HTML Forms with Spring Framework (Web/MVC)

- A general type conversion system was introduced in Spring 3
- The `Converter` SPI to implement type conversion logic is simple and strongly typed:

JAVA

```java
package org.springframework.core.convert.converter;

public interface Converter<S, T> {
    T convert(S source);
}
```

- Use in data binding, and SpEL (Spring Expression Language), etc.

Type Conversion
HTML Forms with Spring Framework (Web/MVC)

- In web (or even desktop) applications, there is the need to convert:
  - *from String* to support client postback process,
  - as well as *to String* to support the view rendering process.
- But the general type conversion system *does not* address localizing String values
- A `Formatter` SPI was introduced in Spring 3 to address localizing
  - parse and print localized field values

Type Conversion
HTML Forms with Spring Framework (Web/MVC)

JAVA
```java
package org.springframework.format;
public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

JAVA
```java
public interface Printer<T> {
    String print(T fieldValue, Locale locale);
}
```

JAVA
```java
import java.text.ParseException;
public interface Parser<T> {
    T parse(String clientValue, Locale locale) throws ParseException;
}
```

Not to be confused with `java.util.Formatter`.

Type Conversion
HTML Forms with Spring Framework (Web/MVC)

Formatters parse string data

→

```
BigDecimal amount = new
      BigDecimal("10000.32");
```

Formatters can format objects for displaying or editing

```
BigDecimal amount = new
      BigDecimal("5000.24");
```

→

Type Conversion
HTML Forms with Spring Framework (Web/MVC)

Three ways to apply formatting of data to/from String for webapps:

1. Annotations
2. JSP tags (using the `fmt` tag library)
3. Register custom `Formatter`s

Type Conversion
HTML Forms with Spring Framework (Web/MVC)

Annotations (usually inside the web-layer form objects)

JAVA

```
public class AccountForm {
    @DateTimeFormat(iso=ISO.DATE)
    private java.util.Date endDate;
}
```

Formatting is applied when Spring tags are used for display and parsing.

The `@DateTimeFormat` annotation can be applied, as of Spring 4 and JDK 8, to JSR-310 `java.time` types too.

Type Conversion
HTML Forms with Spring Framework (Web/MVC)

Annotations (usually inside the web-layer form objects)

JAVA

```
public class AccountForm {
    @NumberFormat(style=Style.CURRENCY)
    private BigDecimal amount;
    @NumberFormat(style=Style.NUMBER, pattern="#,###.00")
    private BigDecimal interestAmount;
    @DateTimeFormat(iso=ISO.DATE)
    private java.util.Date endDate;
}
```

JSP

```
<form:form ... modelAttribute="accountForm">
    <form:input path="amount" />
    <form:input path="interestAmount" />
    <form:input path="endDate" />
</form:form>
```

Type Conversion
HTML Forms with Spring Framework (Web/MVC)

JSP tags (using the `fmt` tag library)

```jsp
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<fmt:formatNumber value="${accountForm.interestAmount}" type="percent" />
<fmt:formatDate value="${accountForm.endDate}" pattern="MM/dd/yyyy" />
```

JSP

```java
public class AccountForm {
    @NumberFormat(style=Style.NUMBER, pattern="#,###.00")
    private BigDecimal interestAmount;
    @DateTimeFormat(iso=ISO.DATE)
    private java.util.Date endDate;
}
```

JAVA

This is plain-vanilla JSP, and is independent of the Spring Framework. In other words, the `@DateTimeFormat` annotation is not used by the `fmt:formatDate` tag.

Type Conversion
HTML Forms with Spring Framework (Web/MVC)

Register custom `Formatter`s

- the `<mvc>` namespace registers a set of formatters by default

```xml
<mvc:annotation-driven />
```

XML

- register a conversion service to add your own formatters

```xml
<mvc:annotation-driven
        conversion-service="conversionService" />
```

XML

Type Conversion
HTML Forms with Spring Framework (Web/MVC)

Sample custom `Formatter`

JAVA

```java
public class SSNFormatter implements Formatter<SocialSecurityNumber> {
    public String print(SocialSecurityNumber ssn, Locale locale) {
        return ssn.getPrefix() + "-" + ssn.getSuffix();
    }
    public SocialSecurityNumber parse(String text, Locale locale)
        throws ... {
        SocialSecurityNumber ssn = new SocialSecurityNumber();
        ssn.setPrefix(text.substring(0, 3));
        ssn.setSuffix(text.substring(4, 6));
        return ssn;
    }
}
```


Type Conversion
HTML Forms with Spring Framework (Web/MVC)

To add sample custom formatter in the prevous slide...

XML

```xml
<mvc:annotation-driven
        conversion-service="conversionService" />
<bean id="conversionService"
        class="...format.support.FormattingConversionServiceFactoryBean">
    <property name="formatters">
        <list>
            <bean class="...SSNFormatter" />
        </list>
    </property>
</bean>
```

Data Binding
HTML Forms with Spring Framework (Web/MVC)

In a web browser, what happens when you submit a form?

Data Binding
HTML Forms with Spring Framework (Web/MVC)

HTML

```
<form method="post" action="/persons">
    <input type="text" name="name" value="" />
    <input type="text" name="age" value="" />
    <input type="text" name="formula" value="" />
    ...
</form>
```

- name: Gareth Wylie
- age: 24
- formula: a + b == 13%!

HTTP

```
POST /persons HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: [size of the request body]

name=Gareth+Wylie&age=24&formula=a+%2B+b+%3D%3D+13%25%21
```

Data Binding
HTML Forms with Spring Framework (Web/MVC)

Spring MVC binds the request to the form object

HTTP

```
POST /persons HTTP/1.1
name: Gareth Wylie
age: 24
formula: a + b == 13%!
...
```

JAVA

```
public class PersonForm {
    private String name;
    private Integer age;
    private String formula;
     // Getters and setters required,
     // but omitted...
}
```

When binding, type conversions occur (e.g. String to Integer).

Data Binding
HTML Forms with Spring Framework (Web/MVC)

Request parameter names are EL expressions matching object structure

HTTP

```
POST /persons/1 HTTP/1.1
name: Leslie
age: 31
children["Amy"].name: Amy
children["Amy"].age: 3
...
addresses[0].street: Boardwalk Ave
addresses[0].city: Monopoly
```

JAVA

```
public class PersonForm {
    private String name;
    private int age;
    private Map<String, PersonForm> children;
    private List<AddressForm> addresses;
}
```

JAVA

```
public class AddressForm {
    private String street, city;
}
```

Data Binding
HTML Forms with Spring Framework (Web/MVC)

- Declare the *form object* as an input argument

JAVA

```
@Controller
public class AccountsController {
    @RequestMapping(method=RequestMethod.POST)
    public String save(PersonForm personForm)
}
```

Submitted form data copied in automagically

- Optionally annotate it with `@ModelAttribute`

JAVA

```
    @RequestMapping(method=RequestMethod.POST)
    public String save(@ModelAttribute("person")
                       PersonForm personForm) {...}
```


Data Binding
HTML Forms with Spring Framework (Web/MVC)

- Allowed fields (white list)

JAVA

```
@Controller
public class AccountsController {
    @InitBinder
    public void initBinder(WebDataBinder binder) {
        binder.setAllowedFields("date", "amount");
    }
}
```

- Disallowed fields (black list)

JAVA

```
    @InitBinder
    public void initBinder(WebDataBinder binder) {
        binder.setDisallowedFields("id", "*Id")
    }
```

Wildcards may be used

Data Binding
HTML Forms with Spring Framework (Web/MVC)

- Required fields

JAVA

```
@Controller
public class AccountsController {
    @InitBinder
    public void initBinder(WebDataBinder binder) {
        binder.setRequiredFields("name", "age");
    }
}
```

- Customize data binding

JAVA

```
    @InitBinder
    public void initBinder(WebDataBinder binder) {
        binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));
    }
```


Data Binding
HTML Forms with Spring Framework (Web/MVC)

To check for binding errors, add a `BindingResult` argument *immediately* after the form object

JAVA

```
@RequestMapping(method=RequestMethod.PUT)
public String update(PersonForm personForm, BindingResult result) {
    if (result.hasErrors()) {
        return "persons/edit";
    }
    // continue on success
}
```

JSP

```
<form:form ... modelAttribute="personForm">
    <form:input path="age" />
    <form:errors path="age" />
</form:form>
```

Data Binding
HTML Forms with Spring Framework (Web/MVC)

Add any of the following to your `MessageSource` to customize binding error messages

PROPERTIES

```
// Type mismatch errors for a named field on a named model attribute
typeMismatch.personForm.amount=Incorrect contribution amount

// Type mismatch errors for a named field
typeMismatch.amount=Incorrect amount

// Type mismatch errors for all fields of a specific type
typeMismatch.common.money.MonetaryAmount=Incorrect monetary amount

// Any "type mismatch" error
typeMismatch=Incorrect value
```


Data Binding
HTML Forms with Spring Framework (Web/MVC)

Missing fields and property access exceptions will be converted to `FieldError`s, collected in the `Errors` instance, using the following error codes:

- Missing field error: "required"
- Type mismatch error: "typeMismatch"
- Method invocation error: "methodInvocation"

Form Validation
HTML Forms with Spring Framework (Web/MVC)

- Spring 4.0 supports Bean Validation API 1.1. (JSR 349) (formerly JSR 303 - Bean Validation 1.0) for validating *form objects*
  - Hibernate Validator is the reference implementation
  - Annotation-driven
  - Both API and implementation must be on the classpath
- Custom validation

Form Validation
HTML Forms with Spring Framework (Web/MVC)

- `@NotNull`
  - The field cannot be `null`
- `@Size(min="", max="")`
  - A `String` field must have a length in the range (min, max)
- `@Pattern(regexp="")`
  - A `String` field is not null and matches the given pattern
- `@NotEmpty`
  - Not standard, but support in Hibernate Validator
  - Check that a `String` is not empty (not null and length > 0),
    or that a `Collection` (or array) is not empty (not null and length > 0)

Form Validation
HTML Forms with Spring Framework (Web/MVC)

Field-level constraints (Bean Validation API)

JAVA

```java
public class PersonForm {
    @NotEmpty
    private String name;
    @NotNull @Past
    private Date birthDay;
    @Pattern(regexp="\\d{12}")
    private String ssn;
}
```


Form Validation
HTML Forms with Spring Framework (Web/MVC)

Property-level constraints (Bean Validation API)

JAVA

```java
public class PersonForm {
    private String name;
    private Date birthDay;
    private String ssn;

    @NotEmpty
    public String getName() {...}
    @NotNull @Past
    public Date getBirthDay() {...}
    @Pattern(regexp="\\d{12}")
    public Date getSsn() {...}
}
```

Class-level constraints are also available.

Form Object Management

HTML Forms with Spring Framework (Web/MVC)

- To invoke validation, annotate controller method parameter with `@Valid`

JAVA

```
@RequestMapping(method=RequestMethod.POST)
public String update(@Valid PersonForm personForm,
        BindingResult result) {
    if (result.hasErrors()) {
        return "persons/edit";
    }
    // continue on success...
}
```

- Errors are registered in the `BindingResult` (combined with binding errors like typeMismatch)

Form Validation

HTML Forms with Spring Framework (Web/MVC)

To display errors,

JSP

```
<form:form ... modelAttribute="personForm">
    <form:errors path="*" />

    <form:input path="name" />
    <form:errors path="name" />

    <form:input path="age" />
    <form:errors path="age" />
</form:form>
```

Form Validation
HTML Forms with Spring Framework (Web/MVC)

To configure validation,

- Use `<mvc:annotation-driven />`
  - Registers `LocalValidatorFactoryBean`
    - Enables JSR-349 globally within Spring MVC
    - JSR-349 dependencies must be on the classpath

Form Validation
HTML Forms with Spring Framework (Web/MVC)

Error messages defined in `MessageSource` override default values

PROPERTIES

```
// "field not null" errors for a named field on a named model attribute
NotNull.personForm.amount=Required purchase amount

// "field not null" errors for a named field
NotNull.amount=Required amount

// "field not null" errors for all fields of a specific type
NotNull.common.money.MonetaryAmount=Required monetary amount

// Any "field not null" error
NotNull=Required value
```

Form Validation
HTML Forms with Spring Framework (Web/MVC)

Custom validation (Bean Validation API)

JAVA
```java
@Constraint(validatedBy = NumericValidator.class)
public @interface Numeric {
    String message() default "{numberExpected}";
    Class<?>[] groups() default {};
}
```

JAVA
```java
@Numeric
private String merchantNumber;
```

JAVA
```java
public class NumericValidator
        implements ConstraintValidator<Numeric, String> {
    @Override public void initialize(Numeric annotation) {...}
    @Override
    public boolean isValid(String value, ConstraintValidatorContext ctx) {
        return (value == null) || value.matches("[1-9][0-9]*");
    }
}
```


Form Validation
HTML Forms with Spring Framework (Web/MVC)

Custom validation (Spring Validator)

JAVA
```java
package org.springframework.validation;

public interface Validator {
    boolean supports(Class<?> clazz);
    void validate(Object target, Errors errors);
}
```

Not to be confused with `javax.validation.Validator` interface.

Form Validation
HTML Forms with Spring Framework (Web/MVC)

Custom validation (Spring Validator)

JAVA

```java
public class PersonValidator implements Validator {
    @Override
    public boolean supports(Class<?> clazz) {
        return clazz instanceof PersonForm.class;
    }
    @Override
    public void validate(Object target, Errors errors) {
        if (! ((PersonForm) target).getSsn().matches("[1-9][0-9]*")) {
            errors.rejectValue("ssn", "Bad format");
        }
    }
}
```

JAVA

```java
@InitBinder
public void initBinder(WebDataBinder binder) {
    binder.setValidator(new PersonValidator());
}
```


Form Validation
HTML Forms with Spring Framework (Web/MVC)

Custom validation (Spring Validator)

JAVA

```java
@Component
public class PersonValidator implements Validator {
    ...
}
```

JAVA

```java
// Inside @Controller
@Autowired
private PersonValidator personValidator;
@InitBinder
public void initBinder(WebDataBinder binder) {
    binder.addValidators(personValidator);
}
@RequestMapping(...)
public ... save(@Valid PersonForm ...) {...}
```

Form Validation
HTML Forms with Spring Framework (Web/MVC)

Custom validation (Spring Validator)

- Explicitly call a *form object* method from the controller

JAVA
```java
public class PersonForm {
    ...
    public void validate(Errors errors) {
        // custom validation checks
    }
}
```

JAVA
```java
// Inside @Controller
@RequestMapping(method=RequestMethod.PUT)
public String update(
        PersonForm personForm, BindingResult result) {
    personForm.validate(result);
}
```


Form Object Management
HTML Forms with Spring Framework (Web/MVC)

- The processing of HTML form submission requires access to the *form object* across two or more requests
  - Initial `GET`
  - Subsequent `POST`s
- Three (3) alternatives to manage the *form object*
  - Create it on every request
  - Retrieve it on every request
  - Store it in the session between requests

Form Object Management
HTML Forms with Spring Framework (Web/MVC)

- Create it on every request

JAVA

```java
@Controller
@RequestMapping("/persons")
public class PersonsController { ...
    @RequestMapping(value="/create", method=RequestMethod.GET)
    public String create(Model model) {
        model.addAttribute("personForm", new PersonForm()); ...
        ...
    }
    @RequestMapping(method=RequestMethod.POST)
    public String save(@Valid PersonForm personForm, ...) {
        ...
    }
}
```


Form Object Management
HTML Forms with Spring Framework (Web/MVC)

- Retrieve it on every request

JAVA

```java
@Controller
@RequestMapping("/persons")
public class PersonsController { ...
    @RequestMapping(value="/{id}/edit", method=RequestMethod.GET)
    public String edit(@PathVariable("id") String id, Model model) {
        // find and convert person to personForm
        model.addAttribute("personForm", personForm); ...
    }
    @RequestMapping(value="/{id}", method=RequestMethod.PUT)
    public String update(@PathVariable("id") String id,
            @Valid PersonForm personForm, ...) {
        ...
    }
}
```

Form Object Management
HTML Forms with Spring Framework (Web/MVC)

- Store it in the session between requests

JAVA

```java
@Controller
@RequestMapping("/persons")
@SessionAttributes("personForm")
public class PersonsController { ...
    @RequestMapping(value="/{id}/edit", method=RequestMethod.GET)
    public String edit(@PathVariable("id") String id, ...) {
        // find and convert person to personForm
        model.addAttribute("personForm", personForm); ...
    }
    @RequestMapping(value="/{id}", method=RequestMethod.PUT)
    public String update(@PathVariable("id") String id,
            @Valid PersonForm personForm, ..., SessionStatus status) {
        ... // convert personForm to person and update
        status.setComplete();
        ...
    }
}
```


Form Object Management
HTML Forms with Spring Framework (Web/MVC)

- Create it on every request
  - Works for new objects
  - Form contains all required data
- Retrieve it on every request
  - Works for editing existing objects
  - Scales well, simple
- Store it in the session between requests
  - Works for new and existing objects
  - Performs better, but doesn't scale well

Building HTML Forms

Spring Framework (Web/MVC)

Lab

Site Personalization

Spring Framework (Web/MVC)

Site Personalization

Spring Framework (Web/MVC)

- Internationalization support
- Managing locales
- Look-and-feel changes using themes
- *Lab*

`MessageSource`

Internationalization Support with Spring Framework (Web/MVC)

- Spring's abstraction for resolving messages from a standard `ResourceBundle`
- Define a bean named `messageSource` in a Spring application context
- Get access to localized messages

MessageSource

Internationalization Support with Spring Framework (Web/MVC)

JAVA

```java
package org.springframework.context;

public interface MessageSource {
    ...
    String getMessage(
            String code, Object[] args, Locale locale)
        throws NoSuchMessageException;
    String getMessage(
            String code, Object[] args, String defaultMessage, Locale locale)
        throws NoSuchMessageException;
}
```


Configuring a MessageSource

Internationalization Support with Spring Framework (Web/MVC)

XML

```xml
<beans ...> ...
    <bean class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basenames">
            <list>
                <value>/WEB-INF/messages/account</value>
            </list>
        </property>
    </bean>
</beans>
```

```
/WEB-INF/messages
    /account_en.properties, /account_en_US.properties, /account_en_GB.properties
    /account_zh.properties, /account_zh_CN.properties, /account_zh_TW.properties
```

Resolving Messages in Java

Internationalization Support with Spring Framework (Web/MVC)

JAVA

```java
@Controller
public class AccountsController {
    private MessageSource messageSource;
    @Autowired
    public AccountsController(..., MessageSource messageSource) {
        ... this.messageSource = messageSource;
    }
    @InitBinder
    public void initBinder(WebDataBinder binder, Locale locale) {
        String datePattern = messageSource.getMessage(
                "date.pattern", null, "MM-dd-yyyy", locale);
        ... // later used to initialize the date format for a custom date editor
    }
    ...
}
```


Resolving Messages in Views

Internationalization Support with Spring Framework (Web/MVC)

- Spring MVC builds on Spring's `MessageSource`
- `JstlView` enables message resolution via JSTL

JSP

```jsp
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<html>
<head>
    <title><fmt:message key="home.title" /></title>
</head>
<body>
    <h1><fmt:message key="home.heading" /></h1>
</body>
</html>
```

PROPERTIES

```properties
home.title=Home
home.heading=Hello World!
```

Resolving Messages in Views

Internationalization Support with Spring Framework (Web/MVC)

- The Spring `<form>` tags resolve error codes
- Also supports switching locales

JSP

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
<body>
    <form:form action="..." modelattribute="...">
        <div>
            <label for="..."><spring:message code="person.firstName" ... /></label>
            <span>
                <form:input id="..." path="firstName" ... />
                <form:errors path="firstName" ... />
            </span>
        </div>
    </form:form>
</body>
</html>
```


Internationalization Support

Site Personalization with Spring Framework (Web/MVC)

`CharacterEncodingFilter`

- Spring MVC provides a filter that can apply character encoding to *requests*.
  - Useful because some browsers do not set a character encoding even if specified in the HTML page or form.
- The filter can work in two modes:
  - Enforce the encoding
  - Add if the encoding is not already specified
- UTF-8 supports a wide range of languages (*recommended*)
- ISO 8859-1 (a.k.a. Latin-1) supports most Western European languages

Internationalization Support

Site Personalization with Spring Framework (Web/MVC)

Example configuration of `CharacterEncodingFilter` (via `web.xml`)

```XML
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>...
<filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```


Managing Locales

Site Personalization with Spring Framework (Web/MVC)

`LocaleResolver`

- An abstration for determining the current locale
- The `DispatcherServlet` looks for a `LocaleResolver`
  - The bean id is expected to be "localeResolver"
- The default is to use the browser locale
  - from `ServletRequest.getLocale()` which is based on the `Accept-Language` header
- The `LocaleResolver` can also store a different locale as requested by the user

Managing Locales
Site Personalization with Spring Framework (Web/MVC)

`LocaleResolver` types

- `AcceptHeaderLocaleResolver` (default)
  - Reads the locale from the request
- `CookieLocaleResolver`
  - Reads/writes the locale from/to a cookie
- `SessionLocaleResolver`
  - Reads/writes the locale from/to the HTTP session

Managing Locales
Site Personalization with Spring Framework (Web/MVC)

`LocaleChangeInterceptor`

- A Spring MVC provided interceptor
- Allows for changing the current locale on every request, via a configurable request parameter.
  - By default, expects a "locale" request parameter

    `http://localhost:8080/myapp?`**`locale=en`**
- Uses the configured `LocaleResolver` to store the new chosen locale

Managing Locales
Site Personalization with Spring Framework (Web/MVC)

Configure locale switching

- Configure the `LocaleChangeInterceptor`

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

- Specify the `LocaleResolver`
  - cannot be the `AcceptHeaderLocaleResolver` (since it does not support writing the locale)

XML
```
<bean id="localeResolver"
    class="org.springframework.web.servlet.i18n.CookieLocaleResolver" />
```


Managing Locales
Site Personalization with Spring Framework (Web/MVC)

To retrieve the current locale in a JSP:

- Expose the `RequestContext` as a request-scope attribute (`ServletRequest.setAttribute(String, Object)`)
- Access using EL

XML
```
<bean class="org.springframework.web...InternalResourceViewResolver">
    <property name="prefix" value="..." />
    <property name="suffix" value="..." />
    <property name="requestContextAttribute" value="requestContext" />
</bean>
```

JSP
```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
    <c:out value="${requestContext.locale.language}" />
```

Look-and-feel Changes Using Themes
Site Personalization with Spring Framework (Web/MVC)

What is a Theme?

- Themes allow dynamic determination of look-and-feel-related values.
  - e.g. path to CSS files, images, etc.
- Each theme has a name and is backed by a dedicated `MessageSource` instance
- A user can change an application's look-and-feel at runtime by switching the theme

Look-and-feel Changes Using Themes
Site Personalization with Spring Framework (Web/MVC)

`ThemeResolver`

- An abstraction for determining the current theme name
- The `DispatcherServlet` looks for a `ThemeResolver`
  - The bean id is expectd to be "themeResolver"
- `FixedThemeResolver` is configured by default
  - Default theme name is "theme"

Look-and-feel Changes Using Themes
Site Personalization with Spring Framework (Web/MVC)

`ThemeResolver` types

- `FixedThemeResolver` (enabled by default)
  - Uses a single theme (and cannot be changed at runtime)
- `CookieThemeResolver`
  - Reads/writes the theme name from/to a cookie
- `SessionThemeResolver`
  - Reads/writes the theme name from/to the HTTP session

Look-and-feel Changes Using Themes
Site Personalization with Spring Framework (Web/MVC)

- Create <theme-name>`.properties` on the classpath

/WEB-INF/classes/theme.properties PROPERTIES

```
main.css=/styles/fresh-spring.css
branding.image=/images/branding-green.gif
```

/WEB-INF/classes/hot-summer.properties PROPERTIES

```
main.css=/styles/hot-summer.css
branding.image=/images/branding-orange.gif
```

- Use theme tag to resolve theme properties

JSP

```
<spring:theme var="mainCss" code="main.css" />
<c:url var="mainCssUrl" value="${mainCss}" />
<link type="text/css" rel="stylesheet" href="${mainCssUrl}" />
```

Look-and-feel Changes Using Themes
Site Personalization with Spring Framework (Web/MVC)

`ThemeChangeInterceptor`

- A Spring MVC provided interceptor
- Detects requests to change the theme name
  - By default, expects a "theme" request parameter

    `http://localhost:8080/myapp?theme=hot-summer`

- Uses the configured `ThemeResolver` to store the new chosen theme

Look-and-feel Changes Using Themes
Site Personalization with Spring Framework (Web/MVC)

To retrieve the current theme in a JSP:

- Expose the `RequestContext` as a request-scope attribute (`ServletRequest.setAttribute(String, Object)`)
- Access using EL

XML

```
<bean class="org.springframework.web...InternalResourceViewResolver">
    <property name="prefix" value="..." />
    <property name="suffix" value="..." />
    <property name="requestContextAttribute" value="requestContext" />
</bean>
```

JSP

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
    <c:out value="${requestContext.theme.name}" />
```

Site Personalization
Spring Framework (Web/MVC)

Lab

Overview of REST Concepts

Overview of REST Concepts

- Core REST concepts
  - What it is and what it is not
  - REST architectural style constraints
  - Uniform interface (GET, PUT, POST, DELETE)
- Richardson Maturity Model
- HATEOAS

What is REST?
Overview of REST Concepts

- **Re**presentational **S**tate **T**ransfer
  - Term introduced by Roy Fielding in 2000 for his doctoral dissertation at UC Irvine. He's the author of HTTP spec.
- Web apps not just usable by browser clients
  - Programmatic clients can also connect via HTTP
- REST is an architectural style that describes best practices for:
  - using HTTP as an application protocol, not just transport
  - emphasizing scalability

Photo from roy.gbiv.com.

What REST is NOT

Overview of REST Concepts

- An API or framework
  - it is an *architectural style*
  - web service APIs that adhere to the REST architectural constraints are called RESTful APIs
- Equal to HTTP
  - REST principles can be followed using other protocols
- The opposite of SOAP
  - REST vs. SOAP is a false dichotomy

Overview of REST Concepts

The REST architectural style describes six constraints:

- Client-server
- Stateless
- Cacheable
- Layered system
- Code on demand (optional)
- Uniform interface

Uniform Interface
Overview of REST Concepts

The uniform interface constraint is fundamental to the design of any REST service. The uniform interface simplifies and decouples the architecture, which enables each part to evolve independently.

The four constraints for this uniform interface are:

- Identification of resources
- Manipulation of resources through representations
- Self-descriptive messages
- Hypermedia as the engine of application state (HATEOAS)

Identifiable Resources
Uniform Interface

- Everything is a resource
  - Customer
  - Car
  - Shopping cart
- Resources are identified by URIs (Uniform Resource Identifier)
  - Meaning URLs
  - REST community prefers the term URI
- Each URI adds value to the client
  - Don't just expose your entire domain

```
http://travelx.com/bookings/hotel/ibis
                                /city/hongkong/room/1234
```

Manipulation of Resources Through Representations

Uniform Interface

- Interacting with resources using a *constrained* set of operations
  - Many nouns (resources)
  - Few/limited verbs (operations)
- HTTP has this kind of limited set of operations
  - `GET`, `PUT`, `POST`, `DELETE`
  - `HEAD` and `OPTIONS` for meta-data

Uniform Interface

Overview of REST Concepts

`GET`

- retrieves a *representation* of a *resource*
- is a safe operation (no side effects)
- is cacheable
  - server may return ETag header when accessed
  - clients send this header on subsequent retrieval
  - if the resource has not changed, 304 (Not Modified) is returned, with empty body
  - similar solution exists for Last-Modified HTTP header

Uniform Interface
Overview of REST Concepts

GET Examples

HTTP
```
GET /transfers/121 HTTP/1.
Host: www.somebank.com
Accept: application/xml
```

`Accept` header defines representation

HTTP
```
HTTP/1.1 200 OK
Date: ...
Content-Length: 1456
Content-Type: application/xml

<transfer id="121" amount="300.00">
    <credit>S123</credit>
    <debit>C456</debit>
</transfer>
```

HTTP
```
GET /transfers/121 HTTP/1.
Host: www.somebank.com
Accept: application/json
```

HTTP
```
HTTP/1.1 200 OK
Date: ...
Content-Length: 86
Content-Type: application/json

{ id: 121, amount: 300.00,
    credit: 'S123', debit: 'C456' }
```


Uniform Interface
Overview of REST Concepts

GET Examples

HTTP
```
GET /transfers/121 HTTP/1.1
Host: www.somebank.com
```

HTTP
```
HTTP/1.1 200 OK
Date: ...
ETag: "b4bdb3-5b0-43ad74ee73ec0"
Content-Length: 1456
...
```

HTTP
```
GET /transfers/121 HTTP/1.1
If-None-Match: "b4bdb3-
    5b0-43ad74ee73ec0"
Host: www.somebank.com
...
```

HTTP
```
HTTP/1.1 304 Not Modified
Date: ...
ETag: "b4bdb3-5b0-43ad74ee73ec0"
Content-Length: 0
```

Uniform Interface
Overview of REST Concepts

`POST`

- creates a new resource
  - usually as child of existing resource
  - URI of child in `Location` response header
- is not safe, not idempotent (cannot just resend)

```
POST /transfers HTTP/1.1
Host: www.somebank.com
Accept: application/xml

<transfer>...</transfer>
```

HTTP

```
HTTP/1.1 201 Created
Date: ...
Content-Length: 0
Location: http://www.somebank.com/
        transfers/1299
...
```

HTTP

Uniform Interface
Overview of REST Concepts

`PUT`

- updates a resource or creates it with a known destination URI
- idempotent operation (multiple invocations yields same result)
- not safe! (has side-effects)

```
PUT /transfers/123 HTTP/1.1
Host: www.somebank.com
Accept: application/xml

<transfer>...</transfer>
```

HTTP

```
HTTP/1.1 204 No Content
Date: ...
Content-Length: 0
...
```

HTTP

Uniform Interface
Overview of REST Concepts

`PUT` Outcomes

```
PUT /transfers/123 HTTP/1.1
Host: www.somebank.com
Accept: application/xml

<transfer>...</transfer>
```

HTTP

```
HTTP/1.1 204 No Content
Date: ...
Content-Length: 0
...
```

HTTP

```
PUT /transfers/123 HTTP/1.1
Host: www.somebank.com
Accept: application/xml

<transfer>...</transfer>
```

HTTP

```
HTTP/1.1 201 Created
Date: ...
Content-Length: 0
Location: http://.../transfers/123
...
```

HTTP

Uniform Interface
Overview of REST Concepts

`DELETE`

- deletes a resource
- idempotent (post condition is always the same)
- not safe!

```
DELETE /transfers/123 HTTP/1.1
Host: www.somebank.com
```

HTTP

```
HTTP/1.1 204 No Content
Date: ...
Content-Length: 0
...
```

HTTP

Uniform Interface
Overview of REST Concepts

	Safe	Idempotent	Cacheable
GET	✓	✓	✓
PUT	✗	✓	✗
POST	✗	✗	✗
DELETE	✗	✓	✗

Overview of REST Concepts

- Core REST concepts
  - What it is and what it is not
  - REST architectural style constraints
  - Uniform interface (GET, PUT, POST, DELETE)
- Richardson Maturity Model
- HATEOAS

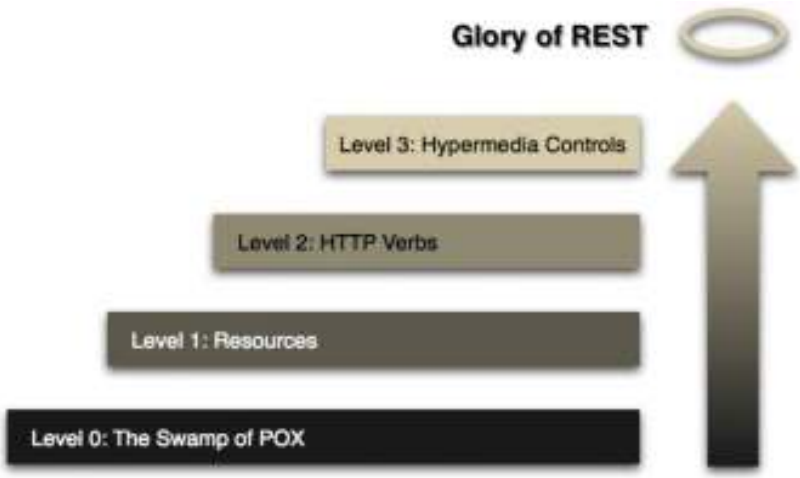
HATEOAS
Overview of REST Concepts

- **H**ypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate
  - Probably the world's worst acronym!
  - RESTful responses contain the links you need
  - Just like HTML pages do
  - Warning: no standard for this yet
  - Least understood part of Roy Fielding's dissertation

  http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Note: Pronunciations of HATEOAS vary. Some people pronounce it as "hate-ee-os," similar to "hideous," or as "hate O-A-S". People also refer to it as a hypermedia-driven system.

Richardson Maturity Model
Overview of REST Concepts



Source: martinfowler.com: Richardson Maturity Model

HATEOAS
Overview of REST Concepts

No clearly accepted standard for hypermedia. Here are *some* hypermedia specs.

- Collection+JSON
- JSON API
- HAL
- JSON-LD
- Siren
- Uber
- Mason
- ...and more.

HATEOAS
Overview of REST Concepts

An example collection of *orders* with `hal+json`.

APPLICATION/HAL+JSON

```
{
    "_links": {
        "self": { "href": "/orders" },
        "next": { "href": "/orders?page=2" },
        "curies": [{ "name": "ea",
                     "href": "http://example.com/docs/rels/{rel}",
                     "templated": true }],
        "ea:find": {...},
        "ea:admin": [...]
    },
    "currentlyProcessing": 14,
    "shippedToday": 20,
    "_embedded": {...}
}
```

Reference: stateless.co/hal_specification.html

HATEOAS
Overview of REST Concepts

The example collection of *orders* with `hal+json` continued.

APPLICATION/HAL+JSON

```
{
    "_links": { ...
        "curies": [{ "name": "ea",
                     "href": "http://example.com/docs/rels/{rel}",
                     "templated": true }],
        "ea:find": {
            "href": "/orders{?id}", "templated": true
        },
        "ea:admin": [{
            "href": "/admins/2", "title": "Fred"
        }, {
            "href": "/admins/5", "title": "Kate"
        }]
    }, ..., "_embedded": {...}
}
```

Reference: stateless.co/hal_specification.html

HATEOAS
Overview of REST Concepts

APPLICATION/HAL+JSON

```
{   "_links": { ... }, "currently Processing": 14, "shippedToday": 20,
    "_embedded": {
        "ea:orders": [{
            "_links": {
                "self": { "href": "/orders/123" },
                "ea:basket": { "href": "/baskets/98712" },
                "ea:customer": { "href": "/customers/7809" }
            },
            "total": 30.00, "currency": "USD", "status": "shipped"
        }, {
            "_links": {
                "self": { "href": "/orders/124" },
                "ea:basket": { "href": "/baskets/97213" },
                "ea:customer": { "href": "/customers/12369" }
            },
            "total": 20.00, "currency": "USD", "status": "processing"
        }]
    }
}
```

Reference: stateless.co/hal_specification.html

Summary
Overview of REST Concepts

- Core REST concepts
  - It is **not** an API, **not** limited to HTTP, **nor** the opposite of SOAP.
  - REST architectural style constraints
  - Uniform interface (GET, PUT, POST, DELETE)
- Richardson Maturity Model
- HATEOAS

RESTful Web Services with Spring MVC

Enterprise Integration with Spring Framework

RESTful Web Services with Spring MVC

Enterprise Integration with Spring Framework

- REST and Java
- Using Spring's `RestTemplate` for clients access
- Spring MVC REST support
- *Lab*

REST and Java

RESTful Web Services with Spring MVC

- Multiple Java frameworks for REST support
- Options include:
  - JAX-RS
  - Spring MVC 3.x (and above) with updated REST-support
- Both are valid choices depending on requirements and developer experience

JAX-RS
REST and Java

- JAX-RS is a Java EE 6 standard for building RESTful applications
  - Focuses on programmatic clients, not browsers
  - Various implementations
    - Jersey (RI), RESTEasy, Restlet, CXF
    - All implementations provide Spring support
  - Good option for full REST support using a standard
  - No support for building clients in standard
    - Although some implementations do offer it

Spring MVC with REST-support
REST and Java

- Spring MVC provides REST support as well
  - since version 3.0
  - using familiar and consistent programming model
  - Spring MVC does not implement JAX-RS
- Offers both programmatic client support (HTTP-based web services) and browser support (RESTful web applications)
- Includes `RestTemplate` for building programmatic clients in Java

RESTful Clients with `RestTemplate`

RESTful Web Services with Spring MVC

- Provides access to RESTful services
- Support URI templates, `HttpMessageConverter`s and custom `execute()` with callbacks
- `Map` or `String...` for vars, `java.net.URI` or `String` for URL

HTTP	RestTemplate
GET	`getForObject(String url, Class<T> responseType, Object... urlVariables)`
HEAD	`headForHeaders(String url, Object... urlVariables)`
OPTIONS	`optionsForAllow(String url, Object.. urlVariables)`
POST	`postForLocation(String url, Object request, Object... urlVariables)`
POST	`postForObject(String url, Object request, Class<T> responseType, Object... urlVariables)`
PUT	`put(String url, Object request, Object... urlVariables)`
DELETE	`delete(String url, Object... urlVariables)`

RESTful Clients with `RestTemplate`

RESTful Web Services with Spring MVC

- Just call constructor in your code

JAVA

```java
RestTemplate template = new RestTemplate();
```

- Has default `HttpMessageConverter`s
  - Same as on the server, depending on classpath
- Or, use external configuration
  - e.g. To use Apache Commons HTTP client

XML

```xml
<bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
    <property name="requestFactory">
        <bean class="org.springframework.http.client.CommonsClientHttpRequestFactory" />
    </property>
</bean>
```

RESTful Clients with `RestTemplate`

RESTful Web Services with Spring MVC

JAVA

```java
RestTemplate template = new RestTemplate();
String uri = "http://www.myshop.com/orders/{id}";

// GET
Order order = template.getForObject(uri, Order.class, "123");
// POST
Order newOrder = ...;
URI orderLocation = template.postForLocation(uri, newOrder);
// PUT
URI itemLocation = ...;
OrderItem item = ...; item.setQuantity(2);
template.put(itemLocation, item);
// DELETE
template.delete(orderLocation);
```


Spring MVC REST Support

RESTful Web Services with Spring MVC

- Spring MVC provides:
  - URI templates to parse *RESTful* URLs
  - views for multiple resource representations
  - support for clients, including browsers
- This section covers:
  - additional features for building *RESTful* web services
  - support for RESTful web applications targeting browser-based clients is available (but is outside this course's scope)

Spring MVC REST Support
RESTful Web Services with Spring MVC

- Request/Response processing
- Using MessageConverters
- Content Negotiation

Request/Response Processing
RESTful Web Services with Spring MVC

Can map HTTP requests based on method

- allows same URL to be mapped to multiple methods
- often used for form-based controllers (GET and POST)
- essential to support RESTful resource URLs
  - including PUT and DELETE

JAVA

```
@RequestMapping(value="/orders", method=RequestMethod.GET)
public ... listOrders(Model model) {
    // find all orders (possibly paginated) and add them to the model
} ...
@RequestMapping(value="/orders", method=RequestMethod.POST)
public ... createOrder(OrderForm orderForm, ...) {
    // process order data from the request
}
```

Request/Response Processing
RESTful Web Services with Spring MVC

Method	Path	Action
GET	/*resource-name*/	List resources (possibly paginated)
GET	/*resource-name*/`{id}`	Retrieve resource with given identifier
PUT	/*resource-name*/`{id}`	Update resource at given identifier
POST	/*resource-name*/	Add new resource (and return new identifier)
DELETE	/*resource-name*/`{id}`	Delete resource with given identifier
GET*	/*resource-name*/`{id}/edit`	Return HTML <form> to edit resource with given identifier (via subsequent PUT)
GET*	/*resource-name*/`create`	Return HTML <form> to create new resource (via subsequent POST)

* Only for HTML browser-based clients.

Request/Response Processing
RESTful Web Services with Spring MVC

- Web apps just use a handful of HTTP status codes
  - 200 OK, 404 Not Found, 302/303 for redirects, and 500 Internal Server Error for unhandled exceptions
- RESTful applications use additional codes to communicate with their clients
- Use `@ResponseStatus` on controller method
  - instead of setting status on `HttpServletResponse`

Request/Response Processing
RESTful Web Services with Spring MVC

Some common HTTP response codes

- 200 - after a successful GET where content is returned
- 201 - when new resource was created on POST or PUT
  - location header should contain URI of new resource
- 204 - when the response is empty
  - e.g. after successful update with PUT or DELETE
- 404 - when requested resource was not found
- 405 - when HTTP method is not supported by resource
- 409 - when a conflict occurs while making changes
  - e.g. when POSTing unique data that already exists
- 500 - internal server error

For more HTTP status codes, refer to online resources like Wikipedia.

Request/Response Processing
RESTful Web Services with Spring MVC

When using `@ResponseStatus`, void methods no longer imply a default view name.

- There will be no `View` at all
- Example below gives a response with an empty body

JAVA

```
@RequestMapping(value="/orders", method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED) // 201
public void createOrder(HttpServletRequest request,
            HttpServletResponse response) {
    Order order = createOrder(request);
    // determine full URI of newly created Order based on request
    response.addHeader("Location",
        getLocationForChildResource(request, order.getId()));
    response.setStatus(HttpServletResponse.SC_CREATED);
}
```

Request/Response Processing
RESTful Web Services with Spring MVC

JAVA

```
@RequestMapping(value="/orders", method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED) // 201
public ResponseEntity<Void> createOrder(HttpServletRequest request,
            UriComponentsBuilder ucb) {
    Order order = createOrder(request);
    // determine full URI of newly created Order based on request
    response.addHeader("Location",
        getLocationForChildResource(request, order.getId()));
    response.setStatus(HttpServletResponse.SC_CREATED);
    HttpHeaders headers = new HttpHeaders();
    headers.setLocation(ucb.path("/orders/{id}").buildAndExpand(
            order.getId()).toUri());
    return new ResponseEntity<Void>(headers, HttpStatus.CREATED);
}
```


Request/Response Processing
RESTful Web Services with Spring MVC

- Location header value must be full URL
- URL of created child resource usually a sub-path
  - POST to http://www.myshop.com/store/orders gives http://www.myshop.com/store/orders/1234
  - Use `UriComponentsBuilder` for encoding where needed

JAVA

```
@RequestMapping(value="/orders", method=RequestMethod.POST)
public ResponseEntity<Void> createOrder(HttpServletRequest request,
            UriComponentsBuilder ucb) {
    Order order = createOrder(request);
    HttpHeaders headers = new HttpHeaders();
    headers.setLocation(ucb.path("/orders/{id}").buildAndExpand(
            order.getId()).toUri());
    return new ResponseEntity<Void>(headers, HttpStatus.CREATED);
}
```

Request/Response Processing
RESTful Web Services with Spring MVC

- Can also annotate exception classes with `@ResponseStatus`
  - Given status code used when exception is thrown from controller method

JAVA
```
@ResponseStatus(HttpStatus.NOT_FOUND) // 404
public class OrderNotFoundException extends RuntimeException {
    ...
}
```

JAVA
```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)
public ... showOrder(@PathVariable("id") long id, Model model) {
    Order order = orderRepository.findOne(id);
    if (order == null) throw new OrderNotFoundException(id);
    model.addAttribute(order);
    ...
}
```


Request/Response Processing
RESTful Web Services with Spring MVC

- For existing exceptions you cannot annotate, use `@ExceptionHandler` method in controller
- Method signature similar to request handling method
- Also supports `@ResponseStatus`

JAVA
```
@ResponseStatus(HttpStatus.CONFLICT) // 409
@ExceptionHandler({ DataIntegrityViolationException.class })
public void conflict() {
    ...
}
```

Using MessageConverters
RESTful Web Services with Spring MVC

`HttpMessageConverter`

- Converts between HTTP request/response and Java object
- Various implementations registered by default when using `<mvc:annotation-driven />`
  - XML (using JAXP Source or JAXB2 mapped object)
  - Feed data (i.e. Atom/RSS)
  - Form-based data
  - JSON
  - `byte[]`, String, BufferedImage
- Define `HandlerAdapter` explicitly to register other `HttpMessageConverter`s

Using MessageConverters
RESTful Web Services with Spring MVC

JAVA

```java
package org.springframework.http.converter;
...
/**
 * Strategy interface that specifies a converter that
 * can convert from and to HTTP requests and responses.
 */
public interface HttpMessageConverter<T> {

    boolean canRead(Class<?> clazz, MediaType mediaType);

    boolean canWrite(Class<?> clazz, MediaType mediaType);

    List<MediaType> getSupportedMediaTypes();

    T read(Class<? extends T> clazz, HttpInputMessage inputMessage)
            throws IOException, HttpMessageNotReadableException;

    void write(T t, MediaType contentType, HttpOutputMessage outputMessage)
            throws IOException, HttpMessageNotWritableException;

}
```

Why Use MessageConverters for Input

RESTful Web Services with Spring MVC

Web service applications often process request content differently that regular web applications

- Not just binding request parameters
  - e.g. a PUT or POST containing XML or JSON document
- Still preferable to not process `HttpServletRequest` directly in controller
- Need something else to map request to method parameter

Why Use MessageConverters for Output

RESTful Web Services with Spring MVC

Writing to response is often different from web application as well

- Controller methods might *not* want to:
  - use separate `View` to render result,
  - write representation to `HttpServletResponse` directly
- Need something else to map return value to response

Using MessageConverters
RESTful Web Services with Spring MVC

`@RequestBody`

- Use converters for *request* data by annotating method parameter with `@RequestBody`

JAVA

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT) // 204
public void updateOrder(@RequestBody Order updatedOrder,
        @PathVariable("id") long id) {
    // process updated order data and return empty response
    orderManager.updateOrder(id, updatedOrder);
}
```


Using MessageConverters
RESTful Web Services with Spring MVC

`@ResponseBody`

- Use converters for response data by annotating method with `@ResponseBody`

JAVA

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)
@ResponseStatus(HttpStatus.OK) // 200
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
    Order order = orderRepository.findOne(id);
    if (order == null) throw new OrderNotFoundException(id);
    return order;
}
```

- Converter handles rendering to response
  - no `ViewResolver` and `View` involved anymore

Content Negotiation
RESTful Web Services with Spring MVC

- HTTP clients can request a particular resource representation through media/MIME types
  - using `Accept` header in HTTP request
  - e.g. `Accept: text/html, application/xhtml+xml, application/xml;q=0.9, */*;q=0.8`
- Best available representation will be returned
- Java object (backing resource) typically the same
  - only representation changes

Content Negotiation
RESTful Web Services with Spring MVC

- `@ResponseBody` method returns just the model (Java object)
- So, how is the correct `HttpMessageConverter` chosen for the requested representation?
  - Not controller's responsibility
  - No `ViewResolver` involved either

NOTE: Remember to use `<mvc:annotation-driven />`, or to register custom converters. Otherwise, no `HttpMessageConverter`s will be configured at all.

Content Negotiation
RESTful Web Services with Spring MVC

- `HttpMessageConverter` selected automatically for `@ResponseBody`-annotated methods
  - Based on `Accept` header in HTTP request
  - Each converter has list of supported media types
- Allows multiple representations for a single controller method
  - without affecting controller implementation
  - alternative for content-based `View` selection as used without `@ResponseBody`

Content Negotiation
RESTful Web Services with Spring MVC

JAVA

```java
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)
@ResponseStatus(HttpStatus.OK) // 200
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
    ...; return order;
}
```

HTTP

```http
GET /orders/121 HTTP/1.1
Host: www.myshop.com
Accept: application/xml
```

→

```http
HTTP/1.1 200 OK
Date: ...
Content-Length: ...
Content-Type: application/xml

<order id="121">...
</order>
```

`Accept` header defines representation

HTTP

```http
GET /orders/121 HTTP/1.1
Host: www.myshop.com
Accept: application/json
```

→

```http
HTTP/1.1 200 OK
Date: ...
Content-Length: ...
Content-Type: application/json

{ id: 121, items: ... }
```

Content Negotiation
RESTful Web Services with Spring MVC

- When mixing browser-based and RESTful clients
  - Two methods on controller *for same* URL:
    - One uses a message converter (`@ResponseBody`)
    - Another uses a view (e.g. JSP)

Content Negotiation
RESTful Web Services with Spring MVC

- Use `produces` to differentiate RESTful GET from a `View` (e.g. HTML, PDF)

JAVA

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET,
    produces={ "application/json", "application/xml" })
@ResponseStatus(HttpStatus.OK) // 200
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
    Order order = orderRepository.findOne(id);
    if (order == null) throw new OrderNotFoundException(id);
    return order;
}

@RequestMapping(value="/orders/{id}", method=RequestMethod.GET)
public String show(@PathVariable("id") long id, Model model) {
    model.addAttribute(getOrder(id));
    return "show";
}
```

Content Negotiation
RESTful Web Services with Spring MVC

- Use `consumes` to differentiate RESTful POST/PUT from a *form* submission

JAVA

```
@RequestMapping(value="/orders", method=RequestMethod.POST,
    consumes={ "application/json", "application/xml" })
public ResponseEntity<Void> saveOrder(
        @RequestBody Order order, UriComponentsBuilder ucb) {
    orderRepository.save(order);
    ... // return Location HTTP header
}

@RequestMapping(value="/orders", method=RequestMethod.POST)
public String save(Order order, BindingResult result, ...) {
    if (result.hasErrors()) { return "create"; }
    orderRepository.save(order);
    return "redirect:list";
}
```


Content Negotiation
RESTful Web Services with Spring MVC

- Can use `MediaType` constants for `produces` and `consumes` attributes

JAVA

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET,
    produces={ MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE })
@ResponseStatus(HttpStatus.OK) // 200
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
    Order order = orderRepository.findOne(id); ...
    return order;
}

@RequestMapping(value="/orders", method=RequestMethod.POST,
    consumes={ MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE })
public ResponseEntity<Void> saveOrder(
        @RequestBody Order order, UriComponentsBuilder ucb) {
    orderRepository.save(order);
    ... // return Location HTTP header
}
```

Spring MVC REST Support
RESTful Web Services with Spring MVC

HTTP

```
GET /orders/121 HTTP/1.1
Host: www.myshop.com
Accept: application/json
```

→

HTTP

```
HTTP/1.1 200 OK
Date: ...
Content-Length: ...
Content-Type: application/json

{ id: 121, items: ... }
```

JAVA

```
@RequestMapping(value="/orders/{id}", method=RequestMethod.GET,
    produces={ "application/json", "application/xml" })
@ResponseStatus(HttpStatus.OK) // 200
public @ResponseBody Order getOrder(@PathVariable("id") long id) {
    Order order = orderRepository.findOne(id);
    if (order == null) throw new OrderNotFoundException(id);
    return order;
}
```


Spring MVC REST Support
RESTful Web Services with Spring MVC

HTTP

```
POST /orders HTTP/1.1
Host: www.myshop.com
Content-Type: application/json

{ items: [...] }
```

→

HTTP

```
HTTP/1.1 201 Created
Date: ...
Content-Length: 0
Location: http://www.myshop.com/orders/44872
```

JAVA

```
@RequestMapping(value="/orders", method=RequestMethod.POST, consumes={...})
public ResponseEntity<Void> placeOrder(
                @RequestBody Order newOrder, UriComponentsBuilder ucb) {
    Long newId = orderManager.placeOrder(newOrder);
    HttpHeaders headers = new HttpHeaders();
    headers.setLocation(ucb.path("/orders/{id}").buildAndExpand(newId).toUri());
    return new ResponseEntity<Void>(headers, HttpStatus.CREATED);
}
```

Spring MVC REST Support
RESTful Web Services with Spring MVC

HTTP

```
PUT /orders/121/item/abc H
Host: www.myshop.com
Content-Type: application/json

/* an item */
{ productId: 10045,
  quantity: 10, ... }
```

→

HTTP

```
HTTP/1.1 204 No Content
Date: ...
Content-Length: 0
```

JAVA

```
@RequestMapping(value="/orders/{orderId}/items/{itemId}",
                method=RequestMethod.PUT, consumes={...})
@ResponseStatus(HttpStatus.NO_CONTENT) // 204
public void updateOrderItem(@PathVariable("orderId") long orderId,
                @PathVariable("itemId") String itemId, @RequestBody Item item) {
    Order order = orderRepository.findOne(orderId);
    if (order == null) throw new OrderNotFoundException(orderId);
    order.updateItem(itemId, item); ...
}
```


Spring MVC REST Support
RESTful Web Services with Spring MVC

HTTP

```
DELETE /orders/121 HTTP/1.
Host: www.myshop.com
```

→

HTTP

```
HTTP/1.1 204 No Content
Date: ...
Content-Length: 0
```

JAVA

```
@RequestMapping(value="/orders/{id}",
                method=RequestMethod.DELETE)
@ResponseStatus(HttpStatus.NO_CONTENT) // 204
public void updateOrderItem(@PathVariable("orderId") long orderId) {
    orderRepository.delete(orderId);
}
```

RESTful Web Services with Spring MVC
Enterprise Integration with Spring Framework

- REST and Java
- Using Spring's `RestTemplate` for clients access
- Spring MVC REST support
  - `@ResponseStatus`, `@RequestBody`, `@ResponseBody`
  - `ResponseEntity<T>`, `UriComponentsBuilder`
  - `produces` and `consumes` in `@RequestMapping`

RESTful Web Services with Spring MVC
Enterprise Integration with Spring Framework

JAVA

```java
@Controller("/accounts") class AccountsController {
    // RESTful method
    @RequestMapping(method=GET, produces={"application/xml", "application/json"})
    @ResponseStatus(HttpStatus.OK)
    public @ResponseBody List<Account> listWithMarshalling() {
        return accountManager.getAccounts();
    }
    // View-based method
    @RequestMapping(method=GET)
    public String listWithView(Model model) {
        // Call RESTful method to avoid repeating account lookup logic
        model.addAttribute( listWithMarshalling() );
        // Return the view to use for rendering the response
        return "accounts/list";
    }
}
```

RESTful Web Services with Spring MVC
Enterprise Integration with Spring Framework

JAVA

```java
// @Controller("/accounts")
    // RESTful method
@RequestMapping(value="/{id}", method=PUT, consumes={...})
public ResponseEntity<?> updateWithMarshalling(@PathVariable("id") long id,
        @Valid @RequestBody AccountForm form, BindingResult result, ...) {
    if (result.hasErrors()) {
        // throw an exception, or return errors as response body
        return new ResponseEntity<...>(..., HttpStatus.BAD_REQUEST);
    }
    // convert form to account and save
    return new ResponseEntity<Void>(HttpStatus.NO_CONTENT);
}
    // View-based method
@RequestMapping(value="/{id}", method=PUT)
public String updateWithView(@PathVariable("id") long id,
        @Valid AccountForm form, BindingResult result, ...) {
    if (result.hasErrors()) { return "edit"; }
    // convert form to account and save
    return "redirect:accounts/list";
}
```


RESTful Web Services with Spring MVC
Enterprise Integration with Spring Framework

Lab

Exception Handling

Spring Framework (Web/MVC)

Exception Handling
Spring Framework (Web/MVC)

- Using `@ResponseStatus` with Exceptions
- Adding Exception handlers to Controllers
- Global exception handling using Controller Advices and Exception resolvers
- Exception handling for RESTful interactions
- *Lab*

`@ResponseStatus` with Exceptions

Exception Handling in Spring Framework (Web/MVC)

- Custom exceptions can be annotated with `@ResponseStatus`
  - When these exceptions are raised, the `ResponseStatusExceptionResolver` handles it by setting the status of the response accordingly
  - By default, the `DispatcherServlet` registers the said exception resolver
- Controller methods can be annotated with `@ResponseStatus`
  - More on this when we cover `@ExceptionHandler` in controllers

`@ResponseStatus` with Exceptions

Exception Handling in Spring Framework (Web/MVC)

JAVA

```java
@ResponseStatus(value=HttpStatus.NOT_FOUND, reason="No such account")
public class AccountNotFoundException extends RuntimeException {
    // ...
}
```

JAVA

```java
@Controller
@RequestMapping("/accounts")
public class AccountsController { ...
    @RequestMapping(value="/{id}", method=RequestMethod.GET)
    public String show(@PathVariable("id") String id, ...) {
        Account account = accountRepository.findById(id);
        if (account == null) throw new AccountNotFoundException(...);
        ...
    } ...
}
```

Exception Handlers in Controller

Exception Handling in Spring Framework (Web/MVC)

- Methods in controllers can be annotated with `@ExceptionHandler`
  - Such methods apply to exceptions thrown by `@RequestMapping` methods of *that* controller
  - Annotation value can be an Exception type, or an array of Exception types
  - Thrown exception is matched to exception type(s)
  - Method with matching annotation is invoked
  - If annotation value is not set, the exception types listed as method arguments are used for matching
- You can also declare an `@ExceptionHandler` method within an `@ControllerAdvice` class in which case it handles exceptions from `@RequestMapping` methods from many controllers.

Exception Handlers in Controller

Exception Handling in Spring Framework (Web/MVC)

JAVA

```java
@Controller
public class ExceptionHandlingController { ...
    // @RequestMapping methods omitted...
    @ExceptionHandler
    public String handleIOException(IOException ex) {
        ... // return logical view name
    }
    @ExceptionHandler(AccountNotFoundException.class)
    public String handleAccountNotFound(Exception ex) {
        ... // return logical view name
    }
    @ExceptionHandler({SQLException.class, DataAccessException.class})
    public String handleDatabaseError(Exception ex) {
        ... // return logical view name
    } ...
}
```

Exception Handlers in Controller

Exception Handling in Spring Framework (Web/MVC)

Exception handler methods have flexible signatures, so you can pass in obvious servlet-related objects such as `HttpServletRequest`, `HttpServletResponse`, `HttpSession` and/or `Principle`.

JAVA

```java
@Controller
public class ExceptionHandlingController {
    // @RequestMapping methods omitted...

    @ExceptionHandler(Exception.class)
    public void handleError(
            HttpServletRequest req, HttpServletResponse res,
            Exception exception) {
        logger.error("Request: " + req.getRequestURL() + " raised " + exception);
        res.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
    }
}
```


Exception Handlers in Controller

Exception Handling in Spring Framework (Web/MVC)

Using `@ResponseStatus` with `@ExceptionHandler`

JAVA

```java
@Controller
public class ExceptionHandlingController {
    // @RequestMapping methods omitted...

    // Convert an exception to an HTTP Status code (409)
    @ResponseStatus(value=HttpStatus.CONFLICT, reason="Data integrity violation")
    @ExceptionHandler(DataIntegrityViolationException.class)
    public void conflict() {
        // Nothing to do
    }

}
```

Exception Handlers in Controller

Exception Handling in Spring Framework (Web/MVC)

JAVA

```java
@Controller
public class ExceptionHandlingController {
    // @RequestMapping methods omitted...

    // Specify the name of a specific view that will be used to display the error
    @ExceptionHandler({SQLException.class, DataAccessException.class})
    public String databaseError() {
        // Nothing to do.
        // Returns the logical view name of an error page, passed to
        // the view-resolver(s) in the usual way.
        return "databaseError";
    }
}
```

Note that the exception is **not** available to this view. It is **not** added to the model.

Exception Handlers in Controller

Exception Handling in Spring Framework (Web/MVC)

Note: the `Model` may not be a parameter of any `@ExceptionHandler` method. Instead, setup a model inside the method using a `ModelAndView` as shown below.

JAVA

```java
@Controller
public class ExceptionHandlingController {
    // @RequestMapping methods omitted...

    // Setup a model and return the view name
    @ExceptionHandler(Exception.class)
    public ModelAndView handleError(HttpServletRequest req, Exception exception) {
        ModelAndView mav = new ModelAndView();
        mav.addObject("exception", exception);
        mav.addObject("url", req.getRequestURL());
        mav.setViewName("error");
        return mav;
    }
}
```

Global Exception Handling

Exception Handling in Spring Framework (Web/MVC)

- Controller advices
- Exception resolvers

Controller-Advice Classes

Global Exception Handling in Spring Framework (Web/MVC)

- A controller advice allows the same exception handling techniques but apply them across the whole application, not just to an individual controller. You can think of them as an annotation-driven interceptor.
- Any class annotated with `@ControllerAdvice` becomes a controller-advice and three types of method are supported:
  1. Exception handling methods annotated with `@ExceptionHandler`.
  2. Model enhancement methods (for adding additional data to the model) annotated with `@ModelAttribute`. Note that these attributes are not available to the exception handling views.
  3. Binder initialization methods (used for configuring form-handling) annotated with `@InitBinder`.

`@ControllerAdvice` was introduced in Spring 3.2.

Controller-Advice Classes

Global Exception Handling in Spring Framework (Web/MVC)

Any of the exception handlers you saw in the previous slides can be defined on a controller-advice class — but now they apply to exceptions thrown from *any* controller.

JAVA

```java
@ControllerAdvice
public class GlobalControllerExceptionHandler {
    @ResponseStatus(HttpStatus.CONFLICT)  // 409
    @ExceptionHandler(DataIntegrityViolationException.class)
    public void handleConflict() {
        // Nothing to do
    }
}
```

As of Spring 4.0, `@ControllerAdvice` accepts elements to define specific subsets of controllers to apply. When multiple selectors are applied, ***OR*** *logic is applied*.

Exception Resolvers

Global Exception Handling in Spring Framework (Web/MVC)

- `HandlerExceptionResolver`s can be configured to handle exceptions across multiple controllers (e.g. `SimpleMappingExceptionResolver`)
- Be careful, since configuring infrastructure beans will cancel the defaults

Exception Resolvers

Global Exception Handling in Spring Framework (Web/MVC)

XML

```xml
<bean class="...ExceptionHandlerExceptionResolver" p:order="0" />
<bean class="...ResponseStatusExceptionResolver" p:order="1" />
<bean class="...DefaultHandlerExceptionResolver" p:order="2" />
<bean class="...SimpleMappingExceptionResolver" p:order="3" >
    <property name="exceptionMappings">
        <entry key="DataAccessException" value="databaseError" />
        <entry key="InvalidCreditCardException" value="creditCardError" />
    </property>
    <property name="defaultStatusCode" value="500" />
    <property name="defaultErrorView" value="error" />
</bean>
```


Exception Handling for RESTful Services

Exception Handling in Spring Framework (Web/MVC)

RESTful requests may also generate exceptions. You can return standard HTTP Error (4xx and 5xx) response codes.

However, what if you want to return information about the error?

- Define an error class,
- and return an instance as `@ResponseBody`

Exception Handling for RESTful Services

Exception Handling in Spring Framework (Web/MVC)

JAVA

```java
public class ErrorInfo {
    public final String url;
    public final String ex;
    public ErrorInfo(String url, Exception ex) {
        this.url = url;
        this.ex = ex.getLocalizedMessage();
    }
}
```

JAVA

```java
// Inside @Controller
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(MyBadDataException.class)
@ResponseBody
public ErrorInfo handleBadRequest(HttpServletRequest req, Exception ex) {
    return new ErrorInfo(req.getRequestURL(), ex);
}
```


Web Application Security with Spring Security

Spring Framework (Web/MVC)

Web Application Security with Spring Security

Spring Framework (Web/MVC)

- Security Overview
- Motivation for Spring Security
- Spring Security in a Web environment
- Using Spring Security tag libraries
- Method security
- *Lab*

Security Overview

Web Application Security with Spring Security

- Security concepts
- Authentication
- Authorization

Security Overview
Web Application Security with Spring Security

Security Concepts

- Principal
  - User, device, or system, that performs an action
- Authentication
  - Establishing that a principal's credentials are valid
- Authorization
  - Deciding if a principal is allowed to perform an action
- Secured resource
  - Resource that is being secured

Security Overview
Web Application Security with Spring Security

Authentication

- The process of determining whether someone or something is, in fact, who or what it is declared to be.
- There are many authentication mechanisms
  - e.g. Basic, Digest, Form, X.509
- There are many storage options for credential and authority information
  - e.g. database, LDAP, in-memory (development)

Security Overview

Web Application Security with Spring Security

Authorization

- Authorization depends on authentication
  - Before deciding if a user can perform an action, user identity must be established
- The decision process is often based on roles
  - ADMIN can cancel orders
  - MEMBER can place orders
  - GUEST can browse the catalog

Motivation

Web Application Security with Spring Security

- Portability
- Flexibility
- Extensibility
- Separation of Concerns
- Consistency

Motivation: Portability

Web Application Security with Spring Security

- Servlet-spec (or container-managed) security is not portable
  - Requires container-specific adapters and role mappings
- Spring Security is portable across containers
  - Secured archive (e.g. WAR) can be deployed as-is
  - Also runs in stand-alone environments

Motivation: Flexibility

Web Application Security with Spring Security

- Supports commons authentication mechanisms
  - Basic, Form, X.509, Cookies, Single Sign-On (SSO), etc.
- Provides configurable storage options for user details (credential and authorities)
  - RDBMS, LDAP, Properties files, custom DAOs, etc.
- Uses Spring for configuration

Motivation: Extensibility

Web Application Security with Spring Security

- Security requirements often require customization
- With Spring Security, all of the following are extensible
  - How a principal is defined
  - Where authentication information is stored
  - How authorization decisions are made
  - Where security constraints are stored

Motivation: Separation of Concerns

Web Application Security with Spring Security

- Business logic is decoupled from security concerns
  - Leverages servlet filters and Spring AOP for an interceptor-based approach
- Authentication and authorization are decoupled
  - Changes to the authentication process have no impact on authorization

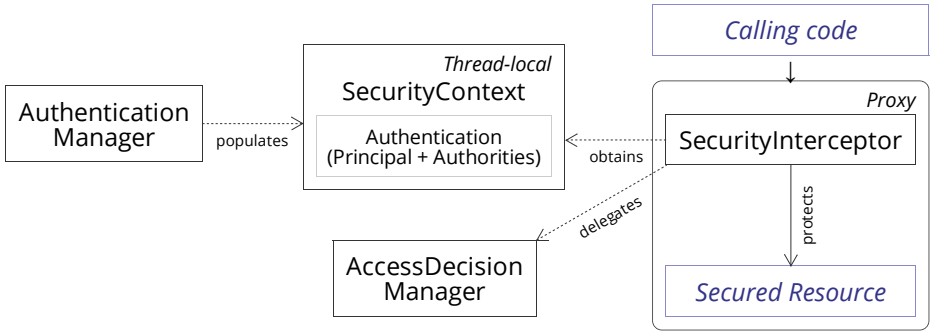
Motivation: Consistency

Web Application Security with Spring Security

- The goal of authentication is always the same regardless of the mechanism
  - Establish a security context with the authenticated principal's information
- The process of authorization is always the same regardless of resource type
  - Consult the attributes of the secured resource
  - Obtain principal information from security context
  - Grant or deny access

The Big Picture

Web Application Security with Spring Security



Spring Security in a Web Environment
Web Application Security with Spring Security

- Configuration in `web.xml`
- XML Configuration of Application Context
  - `<security>` namespace
  - `intercept-url` and EL expressions
  - working with roles
  - login and logout

Configuration in `web.xml`
Spring Security in a Web Environment

- Define the single proxy filter
  - *springSecurityFilterChain* is a mandatory name
  - Refers to an Spring-managed bean with same name

XML

```
<webapp ...>
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>
            org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</webapp>
```

XML Configuration of Application Context
Spring Security in a Web Environment

- Use `<security>` namespace

XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:security="http://www.springframework.org/schema/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
          http://www.springframework.org/schema/security
          http://www.springframework.org/schema/security/spring-security.xsd">
  ...
  <security:http>
    <security:intercept-url pattern="..." access="..." />
  </security:http>
</beans>
```


XML Configuration of Application Context
Spring Security in a Web Environment

- The `<http>` sets up a filter chain with the name of "springSecurityFilterChain"

XML

```
<beans ...>
  ...
  <security:http>
    <security:intercept-url pattern="..." access="..." />
    <security:form-login />
    <security:logout />
  </security:http>
</beans>
```

XML Configuration of Application Context
Spring Security in a Web Environment

- `intercept-urls` are evaluated in the order listed
  - the first match will be used
  - specific matches should be put on top
- `pattern` supports ANT-style path; `access` defines the access requirements for requests matching the given `pattern`

XML

```
<beans ...>
  ...
  <security:http>
    <security:intercept-url pattern="/accounts/*/edit" access="ROLE_ADMIN" />
    <security:intercept-url pattern="/accounts/*" access="ROLE_USER" />
    <security:intercept-url pattern="/accounts/**" access="IS_AUTHENTICATED_FULLY" />
    ...
  </security:http>
</beans>
```


XML Configuration of Application Context
Spring Security in a Web Environment

The `pattern` matches URLs using the following rules:

- `?` matches one character
- `*` matches zero or more characters
- `**` matches zero or more directories in a path
- `/` serves as a path separator

The query string in the URL is ignored.

XML Configuration of Application Context
Spring Security in a Web Environment

Examples (for `pattern` in `<intercept-url>`)

- `/status/t?st.jsp`
  - matches `status/test.jsp` and also `status/tast.jsp` or `status/txst.jsp`
- `/status/*/edit`
  - matches all paths ending with `/edit` in the `status` path (e.g. `/status/345/edit`)
- `/status/**/*.jsp`
  - matches all `.jsp` files underneath the `status` path
- `/**` or `**`
  - matches all (i.e. match any request)

XML Configuration of Application Context
Spring Security in a Web Environment

- In Spring Security 3.0, expression language (EL) provides more flexibility
  - Many built-in expressions available
  - Need to set the `use-expressions` attribute in the `http` element to `true`

XML

```
<beans ...>
  ...
  <security:http use-expressions="true">
    <security:intercept-url pattern="/accounts/*/edit" access="hasRole('ADMIN')" />
    <security:intercept-url pattern="/accounts/*" access="hasRole('ROLE_USER')" />
    <security:intercept-url pattern="/accounts/**" access="isAuthenticated()" />
    ...
  </security:http>
</beans>
```

XML Configuration of Application Context
Spring Security in a Web Environment

Some Security EL expressions:

- `hasRole('role')`
  - Checks whether the principal has the given role
- `hasAnyRole('role1', 'role2')`
  - Checks whether the principal has any of the given roles
- `isAnonymous()`
  - Allows access for unauthenticated principals
- `isAuthenticated()`
  - Allows access for authenticated or remembered principals

XML Configuration of Application Context
Spring Security in a Web Environment

- Checking if the user has a role

```XML
<security:intercept-url pattern="/accounts/*/edit"
    access="hasRole('ADMIN')" />
```

- *"or"* clause

```XML
<security:intercept-url pattern="/accounts/*/edit"
    access="hasAnyRole('ADMIN', 'MANAGER')" />
```

- *"and"* clause

```XML
<security:intercept-url pattern="/accounts/*/edit"
    access="hasRole('ADMIN') and hasRole('MANAGER')" />
```

- **NOTE:** Previous and new syntax *cannot* be mixed

```XML
<security:intercept-url pattern="/accounts/*/edit" access="ROLE_ADMIN" />
<security:intercept-url pattern="/accounts/*/edit"
    access="hasRole('MANAGER')" />
```

XML Configuration of Application Context
Spring Security in a Web Environment

Login and logout

XML

```
<beans ...>
  ...
  <security:http use-expressions="true">
    <security:form-login login-page="/login" />
    <security:intercept-url pattern="/accounts/*/edit" access="hasRole('ADMIN')" />
    <security:intercept-url pattern="/accounts/*" access="hasRole('USER')" />
    <security:logout />
  </security:http>
</beans>
```

Ensure login URL is accessible.

XML Configuration of Application Context
Spring Security in a Web Environment

While there is a *match-all* `intercept-url` (below), the login page can be configured to allow anonymous users (i.e. unauthenticated).

XML

```
<beans ...>
  ...
  <security:http use-expressions="true">
    <security:form-login login-page="/login" />
    <security:intercept-url pattern="/login" access="isAnonymous()" />
    ...
    <security:intercept-url pattern="/**" access="hasRole('USER')" />
    <security:logout />
  </security:http>
</beans>
```

XML Configuration of Application Context
Spring Security in a Web Environment

The `logout` element adds support for logging out by navigating to a particular URL

- default logout URL is `/logout`
- or specified in the `logout-url` attribute

XML

```
<beans ...>
  ...
  <security:http use-expressions="true">
    <security:form-login login-page="/login" />
    ...
    <security:logout />
  </security:http>
</beans>
```


XML Configuration of Application Context
Spring Security in a Web Environment

There are four requirements for a custom login page in Spring Security:

1. There is an input field named `j_username` which will contain the *name* used for the authentication credentials.
2. There is an input field named `j_password` which will contain the *password* used for the authentication credentials.
3. The URL to which these values are `POST`ed matches the URL defined in the `login-processing-url` attribute of the `form-login` element in your Spring Security configuration.
4. The location of the custom login form must be specified in the `login-page` attribute of the `form-login` element in your Spring Security configuration.

As of Spring 4.x, default value of `login-processing-url` is `/login`.

XML Configuration of Application Context
Spring Security in a Web Environment

An example login page

JSP
```
<c:url value="/login" var="loginUrl" />
<form action="${loginUrl}" method="POST">
  <input type="text" name="j_username" /><br/>
  <input type="text" name="j_password" /><br/>
  <input type="submit" name="submit" value="Login" />
</form>
```

Or,

JSP
```
<form action="<c:url value='/login' />" method="POST">
```


Configuring Web Authentication
Web Application Security with Spring Security

- DAO Authentication provider is default
- Plug-in specific `UserDetailsService` implementation to provide credentials and authorities
  - Built-in: JDBC, in-memory
  - Custom

XML
```
<beans ...> ...
  <security:http ...>...</security:http>
  ...
  <security:authentication-manager>
    <security:authentication-provider>
      ...
    </security:authentication-provider>
  </security:authentication-manager>
</beans>
```

In-Memory User Service
Configuring Web Authentication in Spring Security

- Useful for development and testing
  - Without encoding

XML

```
<security:authentication-manager>
  <security:authentication-provider>
    <security:user-service properties="/WEB-INF/users.properties" />
  </security:authentication-provider>
</security:authentication-manager>
```

```
admin=secret,ROLE_ADMIN
user=pass,ROLE_USER
```

  - With encoding

XML

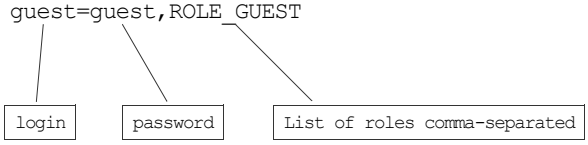
```
<security:authentication-manager>
  <security:authentication-provider>
    <security:password-encoder hash="md5" />
    <security:user-service properties="/WEB-INF/users.properties" />
  </security:authentication-provider>
</security:authentication-manager>
```


In-Memory User Service
Configuring Web Authentication in Spring Security

The properties file

PROPERTIES

```
admin=secret,ROLE_ADMIN,ROLE_MEMBER
testuser1=pass,ROLE_MEMBER
testuser2=pass,ROLE_MEMBER
guest=guest,ROLE_GUEST
```



In-Memory User Service
Configuring Web Authentication in Spring Security

Password salting

- Secure passwords by adding a well-known string
  - makes brute force attacks against password store more complex
- System-wide salt source
  - add static application-wide string

```xml
<security:password-encoder hash="md5">
  <security:salt-source system-wide="secret-sauce" />
</security:password-encoder>
```

XML

- Reflection-based salt source
  - uses a constant/immutable property of entity (e.g. id)

```xml
<security:password-encoder hash="md5">
  <security:salt-source user-property="id" />
</security:password-encoder>
```

XML

JDBC User Service
Configuring Web Authentication in Spring Security

Queries RDBMS for users and their authorities

- Provides default queries

```sql
SELECT username, password, enabled FROM users WHERE username = ?
```

SQL

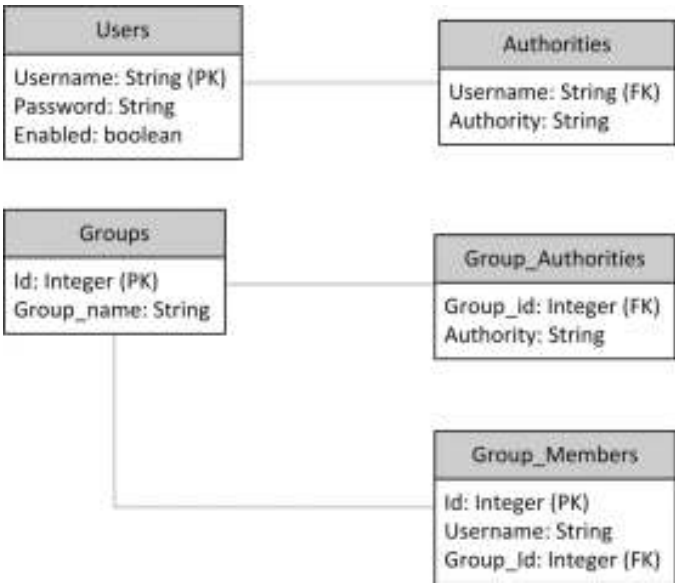
```sql
SELECT username, authority FROM authorities WHERE username = ?
```

SQL

```sql
SELECT g.id, g.group_name, ga.authority
FROM groups g, group_members gm, group_authorities ga
WHERE gm.username = ? AND g.id = ga.group_id AND g.id = gm.group_id
```

SQL

JDBC User Service
Configuring Web Authentication in Spring Security



Spring Security Reference: Security Database Schema

JDBC User Service
Configuring Web Authentication in Spring Security

Configuring a JDBC-based `UserDetailsService`

XML

```
<beans ...> ...
  <security:http ...>...</security:http>
  ...
  <security:authentication-manager>
    <security:authentication-provider>
      <security:jdbc-user-service data-source-ref="..." />
    </security:authentication-provider>
  </security:authentication-manager>
</beans>
```

The queries can be customized by using attributes such as `authorities-by-username-query`.

Configuring Web Authentication
Web Application Security with Spring Security

- Just discussed: In-memory, JDBC
- Other authentication options:
  - Implement a custom `UserDetailsService`
    - Delegate to an existing `User` repository or DAO
  - LDAP
  - X.509 Certificates
  - JAAS Login Module
  - Single Sign-On (SSO)
    - SiteMinder, Kerberos, JA-SIG Central Authentication Service

Authorization is not affected by changes to Authentication!

Tag Libraries
Web Application Security with Spring Security

- Spring Security (since 2.0) provides a tag library

JSP

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

Tag Libraries
Web Application Security with Spring Security

- Display properties of the `Authentication` object

JSP
```
<span>You are logged in as:
  <security:authentication property="principal.username" />
</span>
```

- Hide sections of output based on role

JSP
```
<security:authorize access="hasRole('MANAGER')">
  Click <a href="/admin/deleteAll">here</a> to delete all records.
</security:authorize>
```

- Role declaration can be centralized in Spring configuration (next slide)

Tag Libraries
Web Application Security with Spring Security

- Role declaration can be centralized in Spring configuration

JSP
```
<security:authorize access="hasRole('MANAGER')"
      url="/admin/deleteAll">
  Click <a href="/admin/deleteAll">here</a> to delete all records.
</security:authorize>
```

XML
```
<beans ...>
  ...
  <security:http use-expressions="true">
    <security:intercept-url pattern="/admin/*" access="hasRole('MANAGER')" />
    ...
  </security:http>
</beans>
```

Method Security
Web Application Security with Spring Security

Spring Security uses AOP for security at the method level

XML

```
<beans ...>
  ...
  <security:http ...>...</security:http>

  <security:global-method-security>
    ...
  </security:global-method-security>
</beans>
```

- XML configuration with Spring Security namespace
- Annotations based on Spring annotations or JSR-250 annotations

Method Security (Spring Annotations)
Web Application Security with Spring Security

- Spring Security annotations should be enable

XML

```
<beans ...>...
  <security:global-method-security secured-annotations="enabled" />
</beans>
```

- Then, in the Java methods of Spring-managed beans

JAVA

```
import org.springframework.security.access.annotation.Secured;
...
public class ItemManager {
    @Secured("ROLE_MEMBER")
    public Item findItem(long itemNumber) {...}
    @Secured({ "ROLE_MANAGER", "ROLE_ADMIN" })
    public void updateItem(Item item) {...}
}
```

Method Security (JSR-250 Annotations)

Web Application Security with Spring Security

- JSR-250 annotations should be enable

XML

```
<beans ...>...
  <security:global-method-security jsr250-annotations="enabled" />
</beans>
```

- Then, in the Java methods of Spring-managed beans

JAVA

```
import javax.annotation.security.RolesAllowed;
...
public class ItemManager {
    @RolesAllowed("ROLE_MEMBER")
    public Item findItem(long itemNumber) {...}
}
```


Method Security (Spring Pre- and Post-Annotations)

Web Application Security with Spring Security

- Spring Security 3.x Pre- and Post- annotations should be enable

XML

```
<beans ...>...
  <security:global-method-security pre-post-annotations="enabled" />
</beans>
```

- Then, in the Java methods of Spring-managed beans

JAVA

```
import org.springframework.security.access.prepost.*;
...
public class ItemManager {
    @PreAuthorize("hasRole('MEMBER')")
    public Item findItem(long itemNumber) {...}
}
```

Method Security
Web Application Security with Spring Security

You can enable more than one type of annotation in the same application, but...

XML

```xml
<beans ...>...
  <security:global-method-security
      secured-annotations="enabled"
      jsr250-annotations="enabled"
      pre-post-annotations="enabled" />
</beans>
```


Method Security
Web Application Security with Spring Security

...but only one type should be used for any interface or class as the behaviour **will not be well-defined** otherwise.

JAVA

```java
import org.springframework.security.access.annotation.*;
import org.springframework.security.access.prepost.*;
...
public class ItemManager {
    @PreAuthorize("hasRole('MEMBER')") // <-- bad!
    public Item findItem(long itemNumber) {...}
    @Secured("ROLE_MEMBER")            // <-- bad!
    public Item findItem(long itemNumber) {...}
}
```

Method Security
Web Application Security with Spring Security

If two annotations are found which apply to a particular method, then only one of them will be applied.

JAVA

```java
import org.springframework.security.access.annotation.*;
import org.springframework.security.access.prepost.*;
...
public class ItemManager {
    @PreAuthorize("hasRole('MEMBER')")
    @Secured("ROLE_MEMBER")              // <-- redundant!
    public Item findItem(long itemNumber) {...}

    @PreAuthorize("hasAnyRole('MEMBER', 'MANAGER')")
    @Secured("ROLE_MEMBER")              // <-- error-prone!
    public void updateItem(Item item) {...}
}
```


Method Security
Web Application Security with Spring Security

- Allows to apply security to several beans (usually for those written by others)

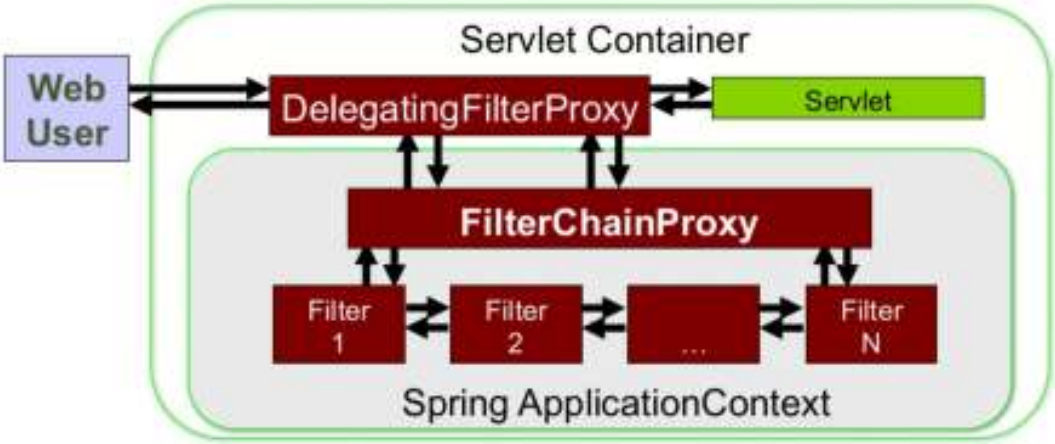
XML

```xml
<beans ...>
  ...
  <security:global-method-security>
      <security:protect-pointcut
          expression="execution(* com.orangeandbronze..*Service.*(..))"
          access="hasRole('USER')" />
  </security:global-method-security>
</beans>
```

Advanced: Working with Filters
Web Application Security with Spring Security

- "springSecurityFilterChain" (`DelegatingFilterProxy`) is declared in `web.xml`
- This servlet filter delegates to a chain of Spring-managed filters
  - Drive authentication
  - Enforce authorization
  - Manage logout
  - Maintain `SecurityContext` in HTTP session
  - *and more*

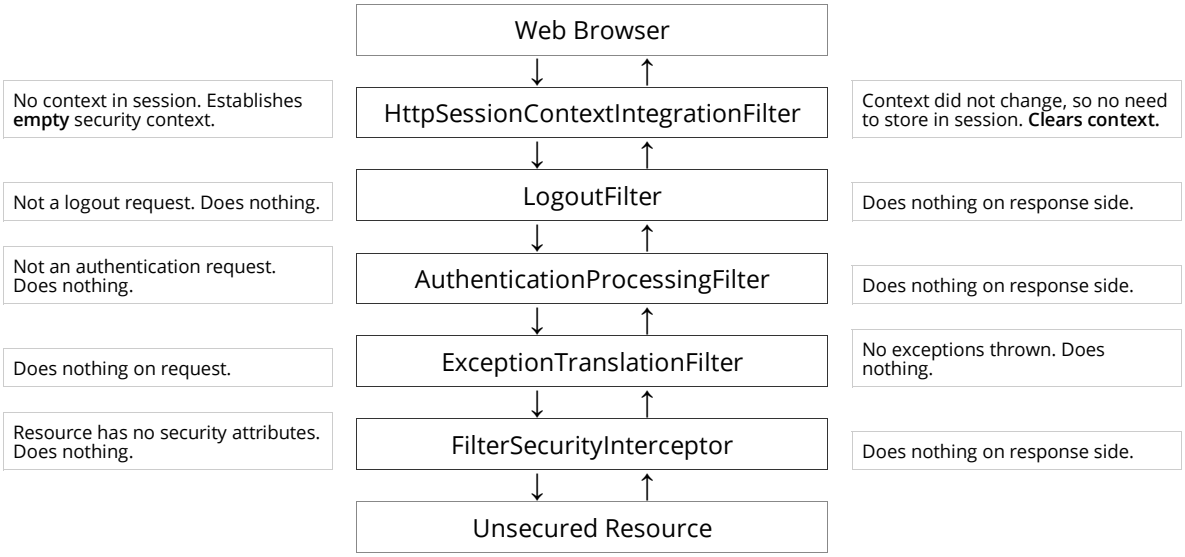
Advanced: Working with Filters
Web Application Security with Spring Security



Advanced: Working with Filters
Web Application Security with Spring Security

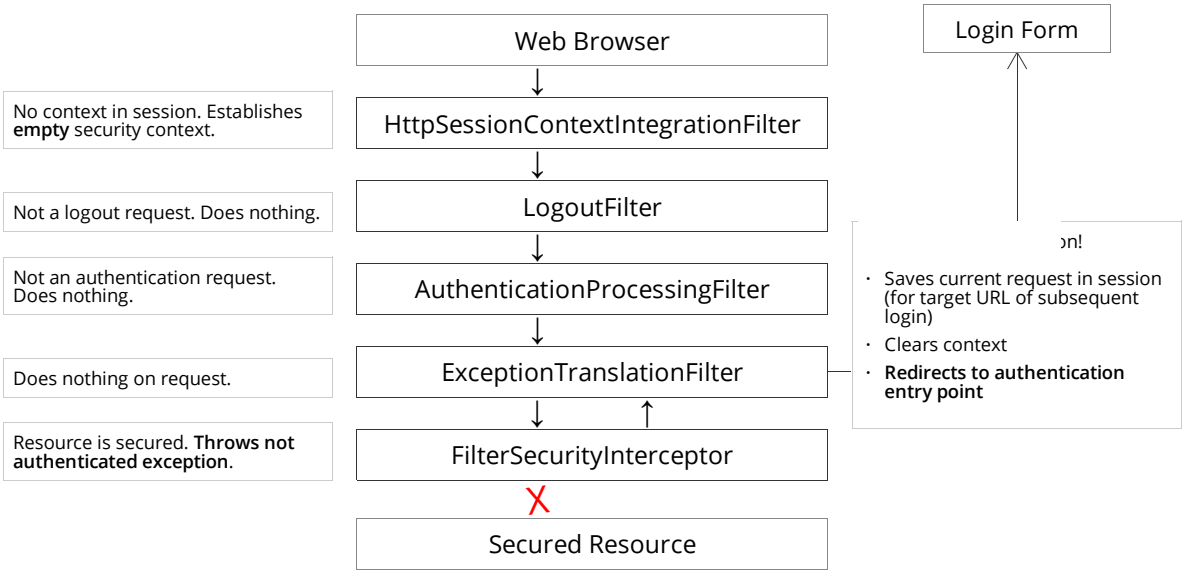
- With ACEGI Security (1.x)
  - Filters were manually configured as individual `<bean>` elements
  - Led to verbose and error-prone XML
- Spring Security 2.x (and above)
  - Filters are initialized with correct values by default
  - Manual configuration is not required unless you want to customize Spring Security's behavior
  - It is still important to understand how they work underneath

Access Unsecured Resource *Prior to Login*
Advanced: Working with Spring Security Filters



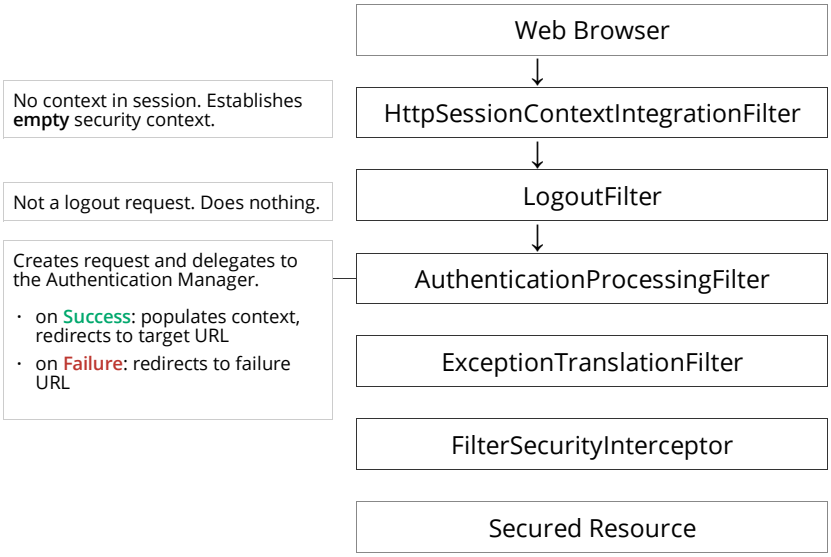
Access Secured Resource *Prior to Login*

Advanced: Working with Spring Security Filters



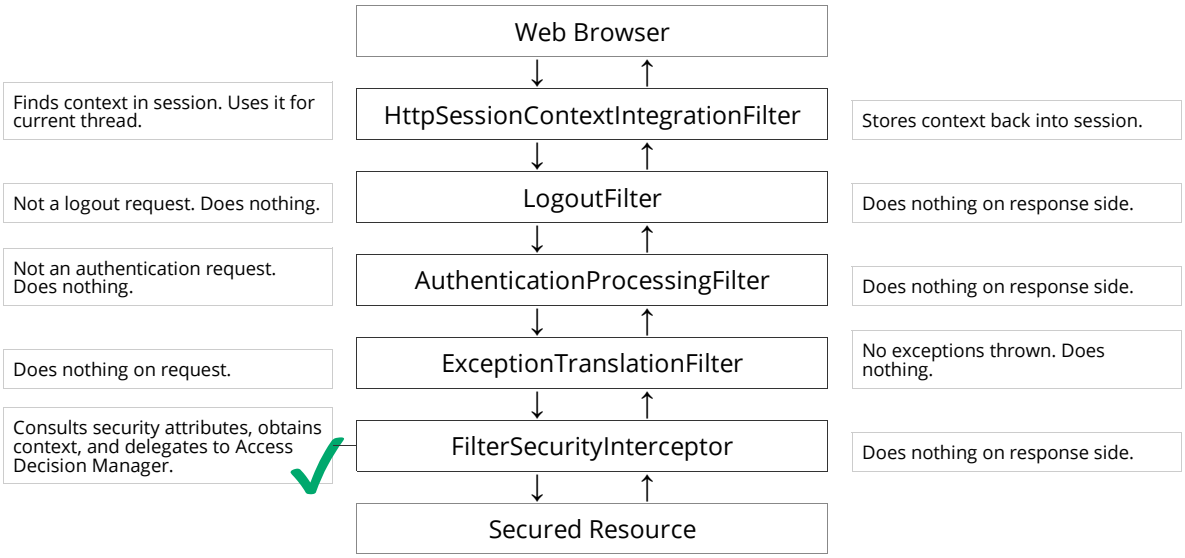
Submit Login Request

Advanced: Working with Spring Security Filters



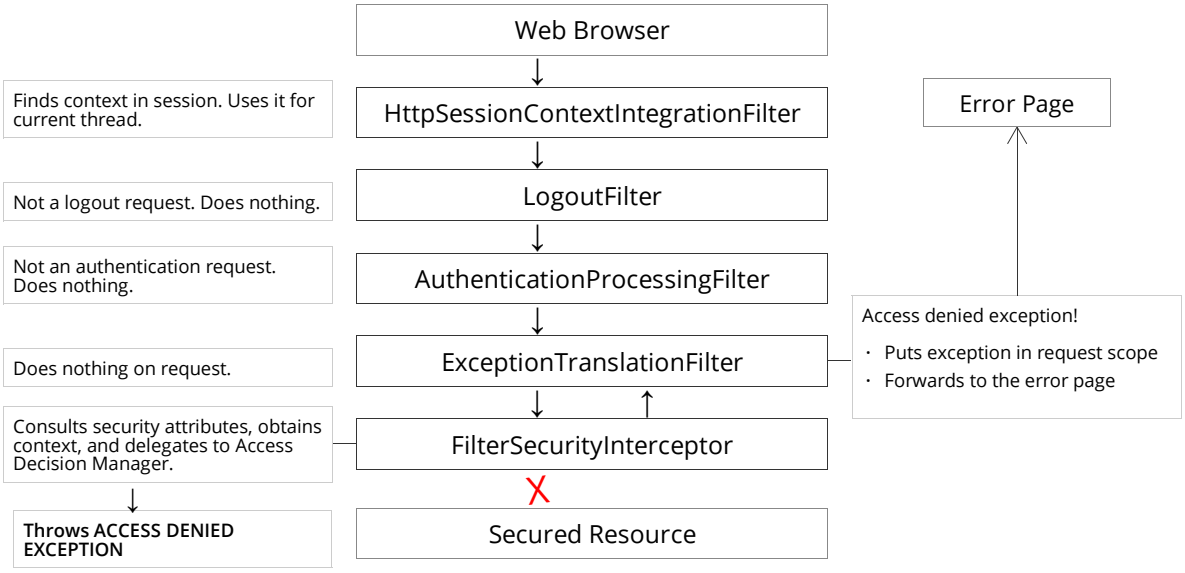
Access Secured Resource *with Required Role*

Advanced: Working with Spring Security Filters



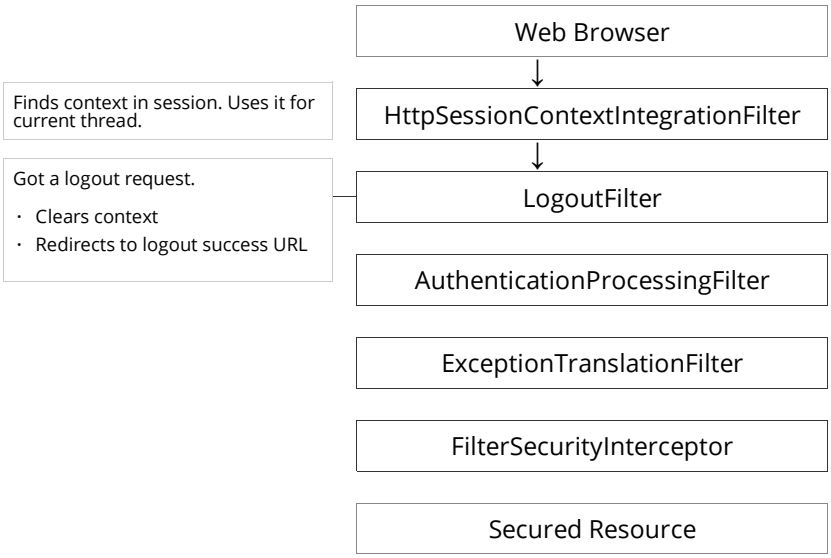
Access Secured Resource *without Required Role*

Advanced: Working with Spring Security Filters



Submit Logout Request

Advanced: Working with Spring Security Filters



The Filter Chain: Summary

Advanced: Working with Spring Security Filters

#	Filter Name	Main Purpose
1	HttpSessionContext IntegrationFilter	Establishes `SecurityContext` and maintains between HTTP requests
2	LogoutFilter	Clears `SecurityContextHolder` when logout requested
3	AuthenticationProcessingFilter	Puts `Authentication` into the `SecurityContext` on login request
4	ExceptionTranslationFilter	Converts Spring Security exceptions into HTTP response and redirect
5	FilterSecurityInterceptor	Authorizes web requests based on configuration attributes and authorities

Custom Filter Chain

Advanced: Working with Spring Security Filters

- A filter on the stack may be **replaced** by a custom filter

```xml
<security:http ...>
  <security:custom-filter position="FORM_LOGIN_FILTER" ref="myFilter" />
</security:http>
<bean id="myFilter" class="...MySpecialAuthenticationFilter" />
```

XML

- A filter can be **added** to the chain

```xml
<security:http ...>
  <security:custom-filter after="FORM_LOGIN_FILTER" ref="myFilter" />
</security:http>
<bean id="myFilter" class="...MySpecialAuthenticationFilter" />
```

XML

You can use `before` and `after` attributes if you want your filter to be inserted *before* or *after* another filter in the stack. The names "FIRST" and "LAST" can be used with the `position` attribute.

Web Application Security with Spring Security

Spring Framework (Web/MVC)

Lab

Building Rich Web Applications with Ajax

Spring Framework (Web/MVC)

Building Rich Web Applications with Ajax

Spring Framework (Web/MVC)

- Ajax and Spring MVC
- Using JavaScript frameworks
- Example: Spring MVC REST and jQuery
- *Lab*

Ajax and Spring MVC
Rich Web Apps with Ajax and Spring (Web/MVC)

What is Ajax?

- A term coined by Jesse James Garett (@jjg)*
- Ajax is a set of technologies that allow web applications to provide:
  - richer interaction
  - just-in-time information
  - dynamic information *without* requiring page refresh
- Ajax is an acronym for **A**synchronous **J**avaScript **a**nd **X**ML

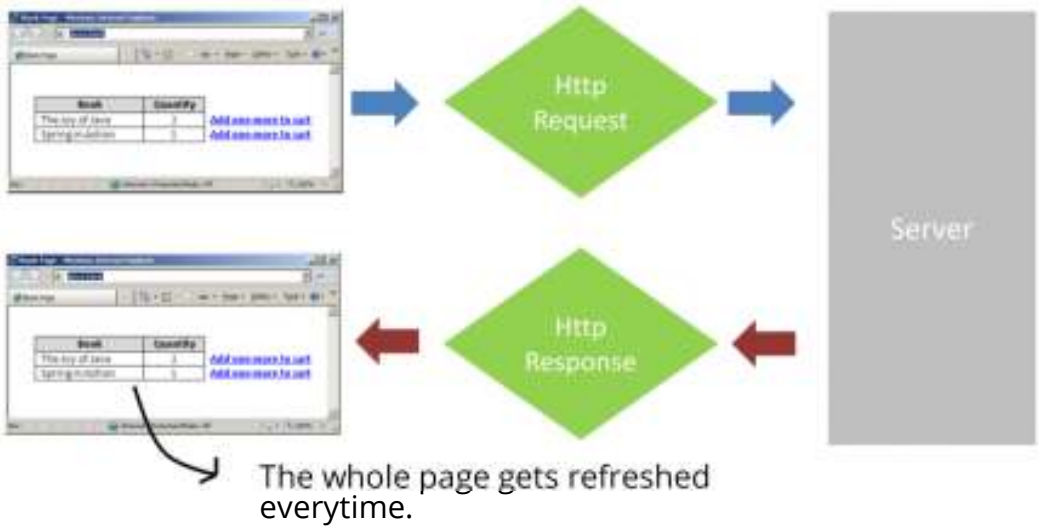
* Ajax: A New Approach to Web Applications

Ajax and Spring MVC
Rich Web Apps with Ajax and Spring (Web/MVC)

- Spring MVC allows you to choose an AJAX/JavaScript library
  - No JavaScript library is integrated by default
  - It is the developer's responsibility to choose a JavaScript library
    - jQuery is a common choice

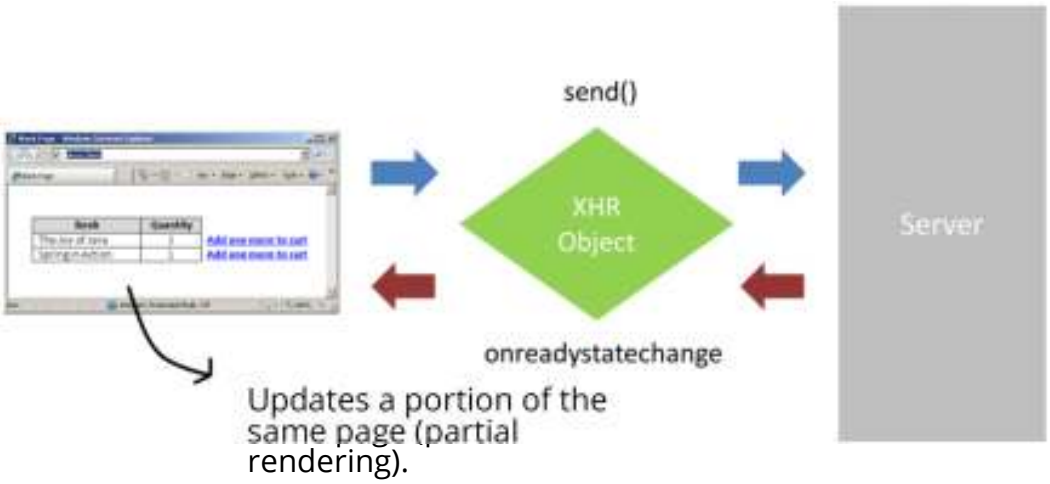
Ajax and Spring MVC

Rich Web Apps with Ajax and Spring (Web/MVC)

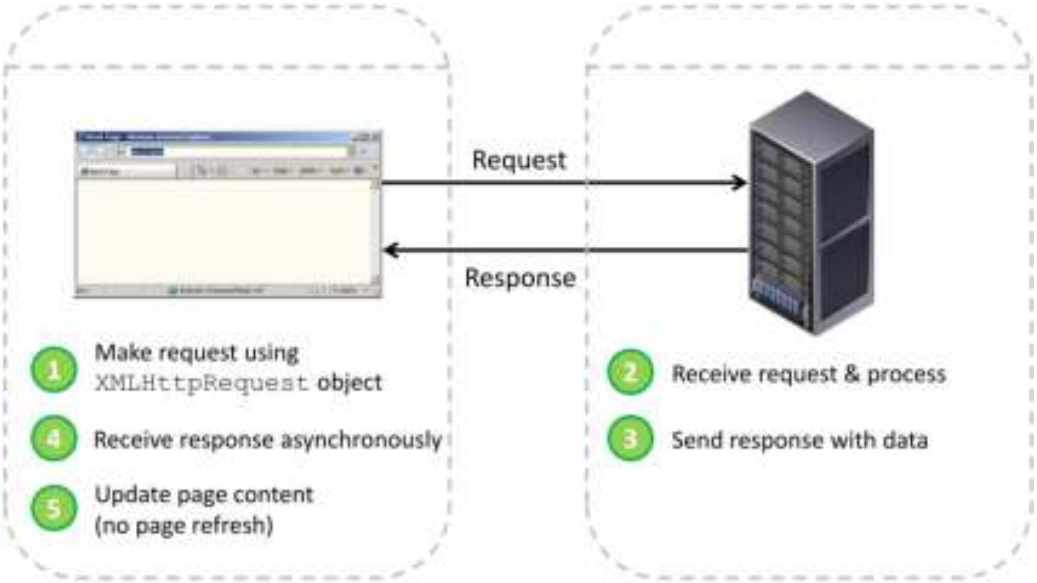


Ajax and Spring MVC

Rich Web Apps with Ajax and Spring (Web/MVC)



Ajax and Spring MVC
Rich Web Apps with Ajax and Spring (Web/MVC)



Ajax and Spring MVC
Rich Web Apps with Ajax and Spring (Web/MVC)

- All modern browsers support the `XMLHttpRequest` object (IE5 and IE6 use an ActiveXObject)
- The `XMLHttpRequest` object is used to exchange data with a server behind the scenes.
  - Updates parts of a web page, without reloading the whole page

JAVASCRIPT

```
...
var xhr = null;
xhr = new XMLHttpRequest();
...
```

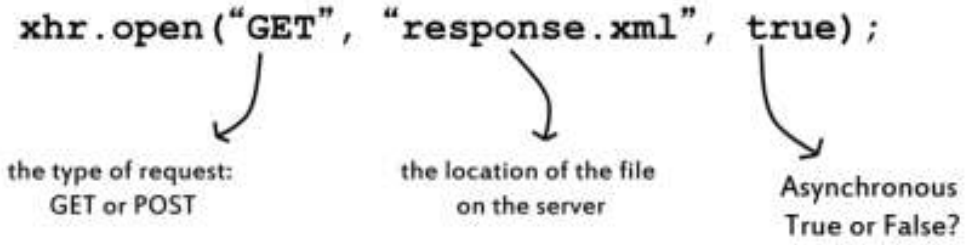
Ajax and Spring MVC
Rich Web Apps with Ajax and Spring (Web/MVC)

- The JavaScript function `handleAjaxResponse` will be invoked when the `readystate` property of the `XMLHttpRequest` object changes

```javascript
function handleAjaxResponse() {...}

var xhr = null;
xhr = new XMLHttpRequest();
xhr.onreadystatechange = handleAjaxResponse;
xhr.open("GET", "response.xml", true);
```

JAVASCRIPT



Ajax and Spring MVC
Rich Web Apps with Ajax and Spring (Web/MVC)

- To get the response from the server, use the `responseText` or `responseXML` property of the `XMLHttpRequest` object
- The `responseText` property returns the response as a string, and you can use it accordingly

```javascript
document.getElementById("myDiv").innerHTML = xhr.responseText;
```

JAVASCRIPT

```javascript
var data = JSON.parse(xhr.responseText);
```

JAVASCRIPT

- If the response from the server is XML, and you want to parse it as an XML object, use the `responseXML` property

```javascript
var xmlDoc = xhr.responseXML;
var data = xmlDoc.getElementsByTagName("someElement");
```

JAVASCRIPT

Ajax and Spring MVC
Rich Web Apps with Ajax and Spring (Web/MVC)

- The `onreadystatechange` event is triggered every time the `readyState` changes.
- The `readyState property holds the status of the`XMLHttpRequest`.
- When `readyState` is 4, and `status` is 200, the response is ready.

JAVASCRIPT

```javascript
var xhr = new XMLHttpRequest();
function handleAjaxResponse() {
    if (xhr.readyState == 4 && xhr.status == 200) {
        document.getElementById("myDiv").innerHTML = xhr.responseText;
    }
}
xhr.onreadystatechange = handleAjaxResponse;
...
```


Using JavaScript frameworks
Rich Web Apps with Ajax and Spring (Web/MVC)

Why use JavaScript frameworks for Ajax?

- Frameworks can:
  - Simplify the process of formulating an Ajax request
    - more expressive
    - less ceremony
  - Allow developers to focus on key aspects of the request
    - what server resource will be called (URL)
    - what parameters will be passed
    - what callback method to respond to

Using JavaScript frameworks
Rich Web Apps with Ajax and Spring (Web/MVC)

- jQuery is a library of JavaScript functions
- It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax, much simpler with an easy-to-use API that works across a multitude of browsers.

Using JavaScript frameworks
Rich Web Apps with Ajax and Spring (Web/MVC)

JavaScript way

Get me all of the elements that have the tag name of "p"

Get me the zeroth element

```
document
  .getElementsByTagName('p')[0]
  .innerHTML = 'Text';
```

Set the HTML inside this element to...

...to this value.

jQuery way

Get the paragraph elements

Change the HTML of those elements to this value.

```
$('p').html('Text');
```

jQuery uses a "selector engine", which means you can get elements with selectors just like CSS does.

Using JavaScript frameworks

Rich Web Apps with Ajax and Spring (Web/MVC)

`$.get(url [,data] [, success] [, dataType])`

- `url`: The URL of the server-side resource to request via GET
- `data`: Any data that should be passed via GET (string, Object, Array)
- `success`: A callback function that is executed if the request succeeds. This can be defined inline or can reference another function by name. The success callback function is passed the returned data.

```
function(Object data, String textStatus, jqXHR jqXHR)
```

- `dataType`: How to interpret the response body (html, text, xml, json, script, etc). By default, uses the content-type of the response body

Example: Spring MVC REST and jQuery

Rich Web Apps with Ajax and Spring (Web/MVC)

JAVASCRIPT

```
$(function() {
    $('#submitButton').bind('click', doAjaxCall);
})

function doAjaxCall(event) {
    $.get('/ajaxUrl', { searchString: 'foo', maximumResults: 5 },
    function(response) {
        // handle response data
    })
}
```

Using JavaScript frameworks
Rich Web Apps with Ajax and Spring (Web/MVC)

`$.getJSON(url [,data] [, success])`

- `url`: The URL of the server-side resource to request via GET
- `data`: Any data that should be passed via GET (string, Object, Array)
- `success`: A callback function that is executed if the request succeeds. This can be defined inline or can reference another function by name. The success callback function is passed the returned data.

The response is interpreted as a JSON object.

Using JavaScript frameworks
Rich Web Apps with Ajax and Spring (Web/MVC)

- The primary benefit of AJAX is that we can have one JSP page to handle both the request and result on the same page.
- Controller to handle the page request

Using JavaScript frameworks
Rich Web Apps with Ajax and Spring (Web/MVC)

- The `@ResponseBody` annotation instructs Spring MVC to serialize
- Spring MVC automatically serializes to JSON because the client accepts that content type

JAVA

```
// Inside a controller
@RequestMapping(value="/add", method=RequestMethod.POST)
public @ResponseBody Integer add(
        @RequestParam(value="inputNumber1", required=true) Integer inputNumber1,
        @RequestParam(value="inputNumber2", required=true) Integer inputNumber2) {
    // delegate to some service to do the actual adding
    Integer sum = someService.add(inputNumber1, inputNumber2);

    // @ResponseBody will automatically convert the returned value into JSON format
    return sum;
}
```


Building Rich Web Applications with Ajax
Spring Framework (Web/MVC)

Lab

<Thank You!>

training-sales@orangeandbronze.com