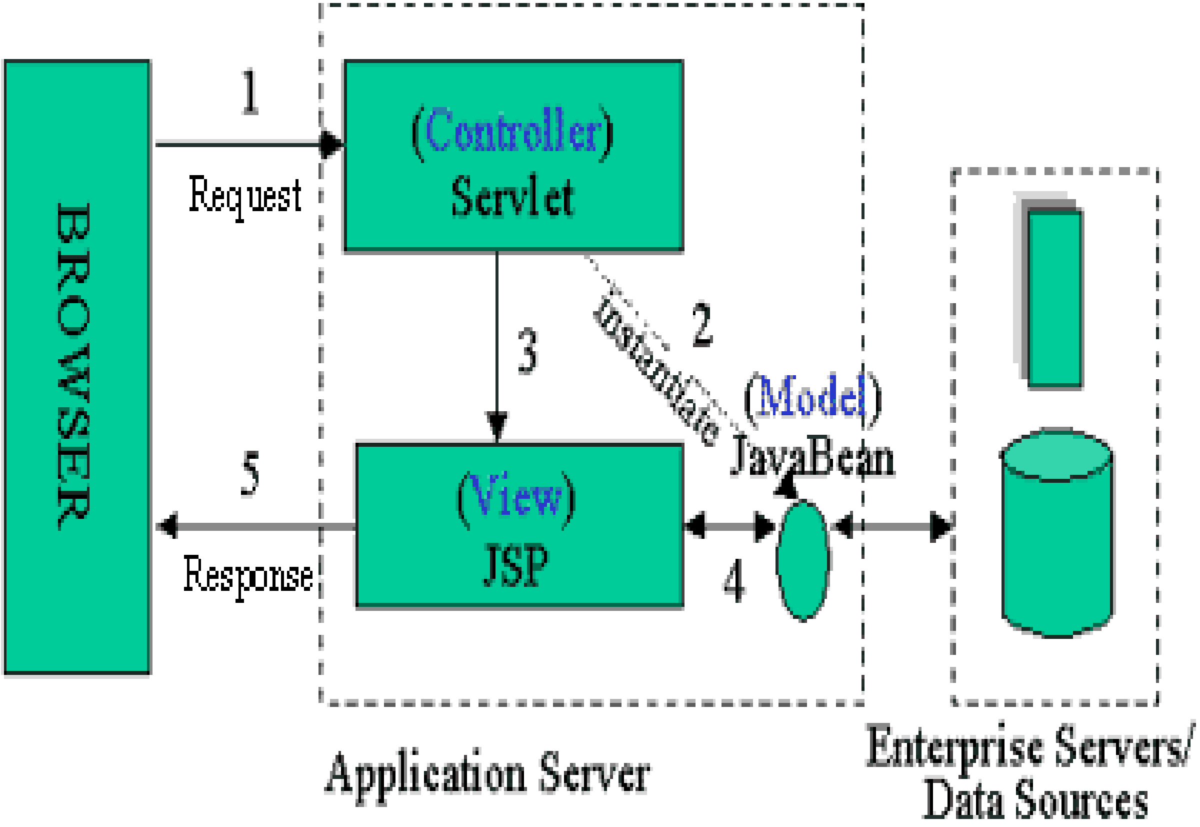
Spring Framework
Module 8 – Spring 3 MVC

Andrey Stukalenko

Last update: March, 2012

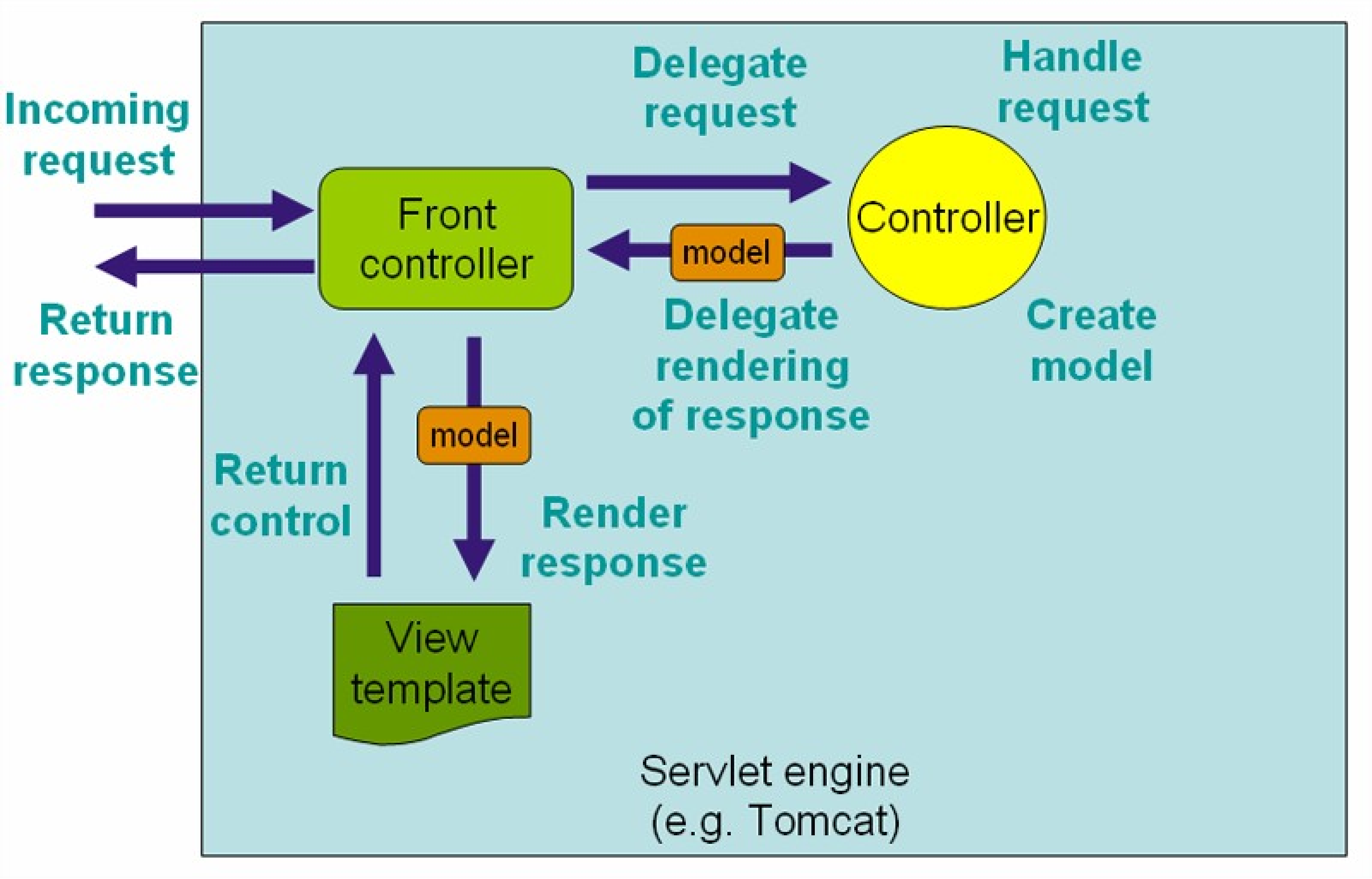
Spring:: MVC :: Introduction

Spring provides powerful and highly configurable MVC web framework raze different from classic model:



Spring:: MVC :: Introduction

Spring model is more flexible:



Spring:: MVC :: Comparison with Struts

- ***Spring MVC model is similar to Struts model, though it does not inherit from Struts***. Spring Controller is analogous to Struts Action in a way that it is multi-threaded service object with one instance executed by all clients. However, is has some benefits over Struts. For example:
  - Spring offers ***clear separation*** between controllers, JavaBean models and views.
  - ***Spring MVC is flexible***. As opposed to Struts, that demands from Action and Form to inherit specific classes and prevent you from inheriting your own classes, Spring MVC is fully interface-based. Moreover, you can substitute any Spring MVC part for your implementation of required interface. For this purpose Spring offers out-of-the-box implementations of all necessary interfaces.

Spring:: MVC :: Comparison with Struts

- Along with controllers Spring ***provides interceptors*** that are useful when you want to implement logic common to all requests.
- Spring is really ***independent from a specific view technology***. You don't have to write *JSPs* if you don't want to. You can use *Velocity*, *XSLT* or any other view technology. If you're going to use your own technology, for example a template language, you can do that by simply implementing the **`SpringView`** interface as you need.
- ***Spring controllers are configured via IoC*** as any other object. That facilitates testing and integration with other Spring-controlled objects.
- Generally, ***Spring MVC Web layer is easy testable***, as long as inheriting specific classes is not required, and controller is not tied to dispatcher servlet.

Spring:: MVC :: Comparison with Struts

- ***Web layer is a thin layer over business objects***, which is a good practice. Struts and other web-oriented frameworks are useless when you are implementing your business objects, whereas Spring provides a complex framework for all application layers.

- In Spring MVC you can have as many dispatcher servlets as you need, like in Struts v.1.1 or higher.

- The following example shows how simple Spring controller can access business objects defined in the same application context. This controller looks up **`Order`** object and returns it through **`handleRequest`**() method:

Spring:: MVC :: Example

```
public class OrderController  implements Controller {
    private OrderRepository repo;
    @Autowired
    public OrderController(OrderRepository orderRepository) {
        repo = orderRepository;
    }
    public ModelAndView handleRequest(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException
    {
        String id = request.getParameter("id");
        Order order = repo.getOrderById(id);
        return new ModelAndView("orderView", "order", order);
    }
}
```

Spring:: MVC :: Example

```
@Controller
public class OrderController {
   @Autowired
   private OrderRepository repo;
   public OrderController(OrderRepository orderRepository) {
       repo = orderRepository;
   }

  @RequestMapping(“/viewOrder”)
  public String viewOrder(int orderId, Model model)
   {
       Order order = repo.getOrderById(id);
       model.put(order);
       return “orderView”;
   }
}
```

Spring:: MVC :: Example

- Spring IoC isolates this controller from OrderRepository implementation, which can work on the basis of JDBC in the same manner as on the basis of web service.

- An interface can be implemented with:
  - POJO,
  - Stub,
  - Proxy
  - Remote object.

- Controller doesn’t include anything related to searching resources; what it includes is a code needed for web interaction.

Spring:: MVC :: Example

Spring MVC supports data binding, forms, ‘masters’ and more complex approaches.

Binding: binding and charting parameters of HTTP requests for objects properties (Java beans) and vice versa. Useful when interacting with user input.

Resolver, mapping: an object implementing mapping (for example, URL requests and views)

Spring:: MVC :: Architecture

`WebApplicationContext: ApplicationContext` adapted for work in Web .

`DispatcherServlet:` a servlet used in intercepting all user's requests. It implements "FrontController" pattern.

Model: **`java.util.Map`**

Stores data as key-value pair

View: **`interface View`**

When this interface is implemented the data is viewed

Controller: **`interface Controller`**

When this interface is implemented user's requests are handled

Spring:: MVC

«Open for extension, closed for modification»

http://objectmentor.com/resources/articles/ocp.pdf

Is request-driven.

`DispatcherServlet` serves as ”input”

Based in Inversion of Control (IoC): maintains flexibility in terms of components modification.

Can be used jointly with any web framework.

Spring:: MVC

Clear separation of object roles (controller, validator, form object, model object, view and so on).

The only model is `java.util.Map`. This guarantees easy integration with any view technology.

Reusable code: business objects can be model objects.

Spring:: MVC :: Example

- Several default controller’s subclasses used for different purposes.
- Binding and validation strategies (exceptions: application-level; property: not String-only).
- Strategies for searching (resolvers) responsible controllers and necessary views.

Spring:: MVC :: WebApplicationContext

- This is an extension of **`ApplicationContext`** that has some extra features necessary for web applications (for example, association with **`ServletContext`**).
- Adds three scopes of bean lifecycle that are only available in web context (request, scope, global).
- Special bean types can only exist in **`WebApplicationContext`**.

Spring:: MVC :: WebApplicationContext

- Special bean types
  - Controllers
  - HandlerMapping
  - ViewResolver
  - LocaleResolver
  - ThemeResolver
  - MultipartResolver
  - HandlerExceptionResolver

Spring:: MVC :: WebApplicationContext

- Configuration
  - To initialize a context, add `ContextLoaderListener` to web.xml
  - During context initialization, beans defined in files **applicationContext.xml** and **[servlet-name]-servlet.xml** (except for lazy-init beans) are instantiated (for each `DispatcherServlet`)
  - File set used in beans instantiation can be changed by specifying `contextConfigLocation` parameter in application descriptor.

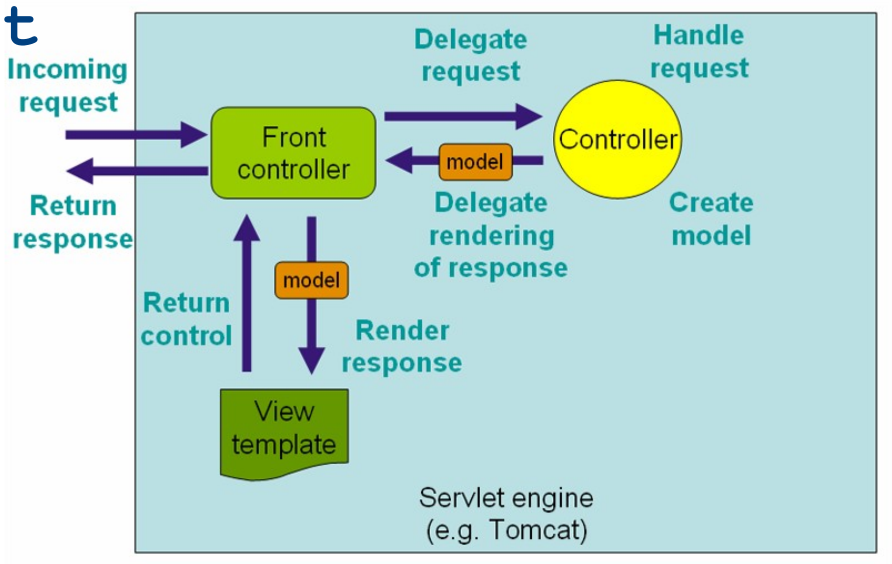
Spring:: MVC :: WebApplicationContext

Example:

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/context.xml
        /WEB-INF/handlers.xml
        /WEB-INF/myContext.xml
        /WEB-INF/validators.xml
        /WEB-INF/commonBeans.xml
    </param-value>
</context-param>
```

Spring:: MVC :: DispatcherServlet

- This is `HttpServlet`
- Implements “Front Controller” pattern.
- Serves as “input” dispatching requests to handlers.
- Has its own `WebApplicationContext` and hides the underlying logic.
- Inherits root context.
- Is aware of `ServletContext`



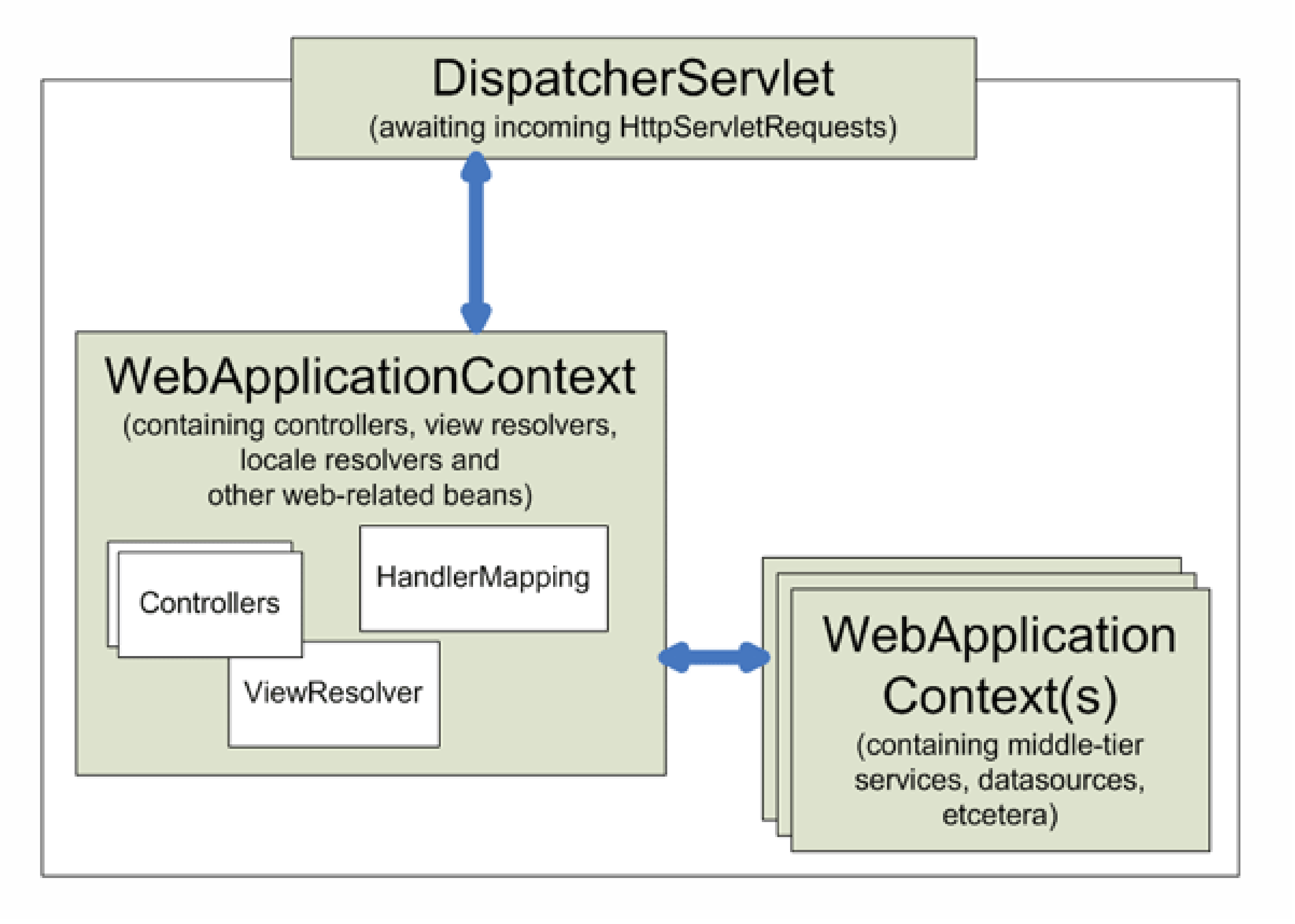
Spring:: MVC :: DispatcherServlet

- Configuration:

```
<servlet>
    <servlet-name>viewer</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>viewer</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Spring:: MVC :: DispatcherServlet

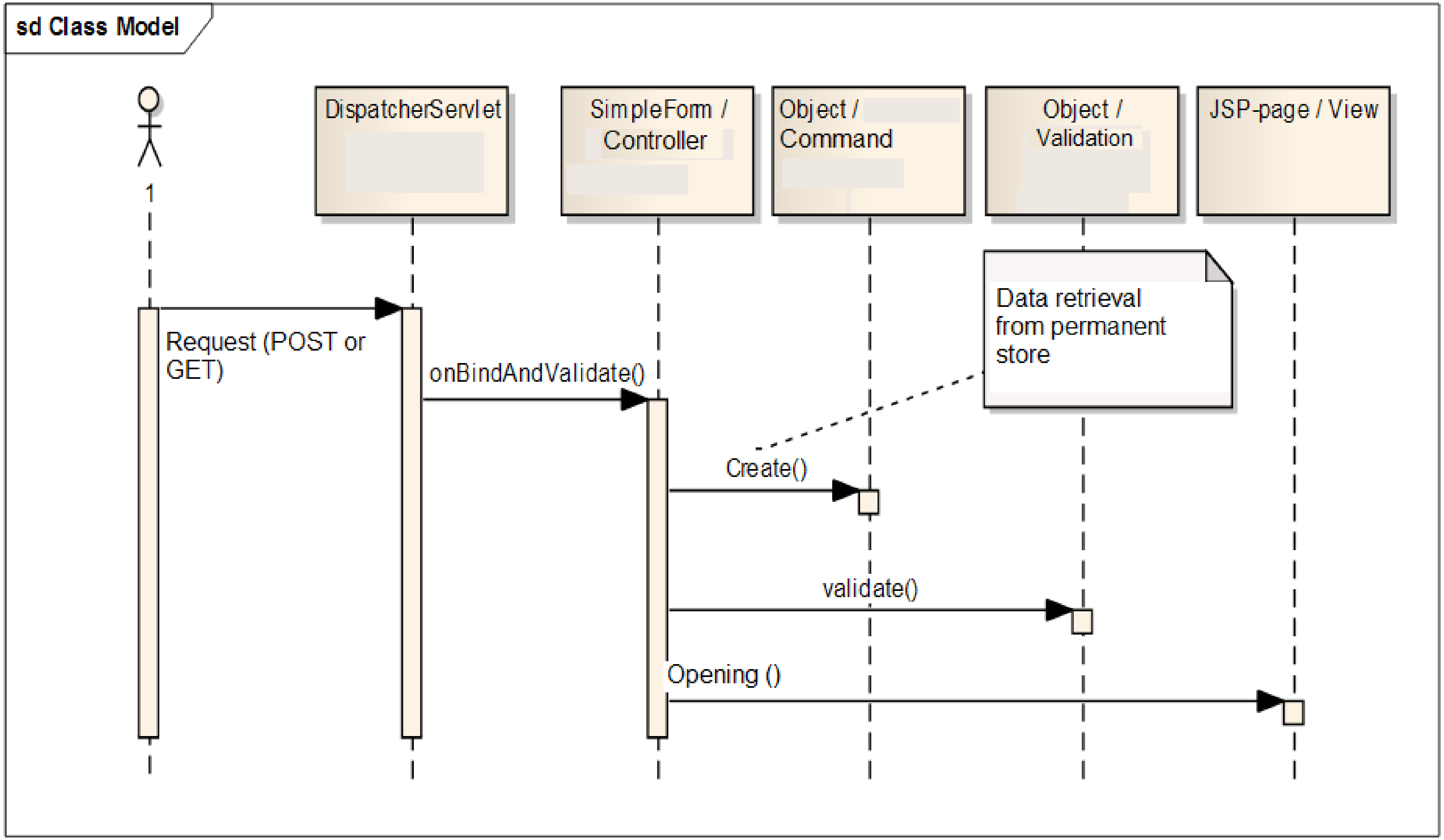
- Context hierarchy:


DispatcherServlet
(awaiting incoming HttpServletRequests)
WebApplicationContext
(containing controllers, view resolvers, locale resolvers and other web-related beans)
Controllers
HandlerMapping
ViewResolver
WebApplication Context(s)
(containing middle-tier services, datasources, etcetera)

Spring:: MVC :: DispatcherServlet

- Request handling:
  - **`WebApplicationContext`** is bound in the request so that controllers can use it.
  - The locale and theme resolver are bound to the request as well.
  - A handler is searched for. The execution chain is executed (interceptors and controllers).
  - If a model is returned, the view is searched for and rendered.

Spring:: MVC :: DispatcherServlet



Spring:: MVC :: Web & RESTful service controllers

- Do the following:
  - Interpret user inputs (requests)
  - Create model
  - Pass model to view for rendering

Spring:: MVC :: Web & RESTful service controllers

- Basic interface:

```
package org.springframework.web.servlet.mvc;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;

public interface Controller {
    ModelAndView handleRequest(
        HttpServletRequest request,
        HttpServletResponse response
    ) throws Exception;
}
```

Spring:: MVC :: Web & RESTful service controllers

- AbstractController:
  - Provides access to **`ServletContext`** and **`ApplicationContext`**
  - Undertakes work with some HTTP headers
  - Manages caching and synchronization
  - Is a superclass for all other implementations

Spring:: MVC :: Web & RESTful service controllers

- AbstractController properties:
  - `supportedMethods`: supported HTTP methods (by default, GET,POST)
  - `requiresSession`: if no session is found while processing the request, an exception will be thrown
  - `synchronizeOnSession`: indicates whether the call to main method should be synchronized around the session
  - 4 properties managing depreciation and caching policy

Spring:: MVC :: AbstractController

- Workflow (and that defined by interface):
  - handleRequest() will be called by the DispatcherServlet ;
  - Inspection of supported methods (ServletException if request method is not support) ;
  - If session is required, try to get it (ServletException if not found) ;
  - Set caching headers if needed according to the cacheSeconds property ;
  - Call abstract method handleRequestInternal() (optionally synchronizing around the call on the HttpSession), which should be implemented by extending classes to provide actual functionality to return ModelAndView objects.

Spring:: MVC :: AbstractController

- Example of using AbstractController:

```
package samples;
public class SampleController extends AbstractController {
    public ModelAndView handleRequestInternal(
        HttpServletRequest request,
        HttpServletResponse response
    ) throws Exception {
        ModelAndView mav = new ModelAndView("hello");
        mav.addObject("message", "Hello World!");
        return mav;
    }
}
```

- .xml:

```
<bean id="sampleController" class="samples.SampleController">
    <property name="supportedMethods" value="GET"/>
</bean>
```

Spring:: MVC :: @Controller

- With @Controller annotation
- Don’t forget about context instruction:

```
<context:component-scan base-package=foo.bar.web"/>
 @Controller
 public class HelloWorldController {

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

Spring:: MVC :: @Controller

A common pitfall when working with annotated controllers is using AOP, on the whole, and declarative transaction management, in particular.

Standard JDK Dynamic Proxies only work with interfaces.

To use it with controllers you have to move the @RequestMappings annotations to the interface (as far as mapping mechanism can “see” **proxy**), which is notably uncomfortable.

Alternatively, activate – `proxy-target-class="true"` and use CGLib

Therefore (think of AOP and TX), when working with web applications, do everything DIRECTLY on classes

Spring:: MVC :: @RequestMapping

Configuring mapping with @RequestMapping annotations:

Specified for:

- Controller (class):
  - absolute path;
- Controller’s methods:
  - If class path is specified then the path is relative;
  - Absolute path if not specified for class;

Spring:: MVC :: Mapping, URL Patterns

- URL patterns are supported;
- @PathVariable is used to bind variable (argument) to the value of URL template;
- Used in implementing RESTful services;

Spring:: MVC :: Mapping, URL Patterns

- Examples:

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}”)
public String findPet(@PathVariable String ownerId,
   @PathVariable String petId, Model model) {
 // implementation
}

@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {
  @RequestMapping("/pets/{petId}")
  public void findPet(@PathVariable String ownerId,
   @PathVariable String petId, Model model) {
     // implementation
  }
}
```

Spring:: MVC :: @RequestMapping

value: specifies URL pattern . Supports:

- parameterized (/somepath/{someVar}) URL patterns
- Ant-style (/somepath/*.do)
- their combinations (/owners/*/pets/{petId})
- If there is no argument, method name is used in mapping

method: binding to request type:

- RequestMethod.POST
- RequestMethod.GET

params: indicates whether some specific parameters for the request are needed

- “myParam = myValue”: parameter with specified value
- “myParam”: parameter with arbitrary value
- “!myParam”: no parameter in the request

- headers: narrowing to request header (for example, "content-type=text/*»)

Spring:: MVC :: @RequestMapping, @RequestParam

- Binding to request parameters is made through @RequestParam annotation
- By default, such parameters are mandatory, but could be turned to optional with required=false command.
- Example:

```
@RequestMapping(method = RequestMethod.GET)
public String setupForm(
   @RequestParam("petId") int petId,
   ModelMap model)

@RequestMapping(method = RequestMethod.GET)
public String showItem(
   @RequestParam(value=”id”, required=false) int id,
   ModelMap model)
```

Spring:: MVC :: @RequestMapping

Supported arguments of handler methods:

- Request / Response objects (Servlet API). Subtypes can be used (for example, ServletRequest or HttpServletRequest);
- Servlet API Session object (HttpSession). An argument of this type enforces the presence of a corresponding session when it is absent;
- WebRequest and NativeWebRequest (org.springframework.web.context.request package) allows for request parameters access without ties to Servlet/Portlet API ;
- Java.util.Locale for the current request locale;
- InputStream / Reader for access to the request’s content;
- OutputStream / Writer for generating the response’s content;
- java.security.Principal containing the currently authenticated user;
- @PathVariable annotated parameters for access to URI template variables ;
- @RequestParam annotated parameters for access to request parameters ;

Spring:: MVC :: @RequestMapping

Supported arguments of handler methods :

- @RequestHeader annotated parameters for access to specific request headers;
- @RequestBody for access to the request body;
- java.util.Map / org.springframework.ui.Model / org.springframework.ui.ModelMap for access to model that is exposed to the web view;
- org.springframework.validation.Errors / org.springframework.validation.BindingResult : for access/validation results for an object. This argument shall follow the validated object **immediately**, as each object is validated and tied to validation result separately;
- org.springframework.web.bind.support.SessionStatus for controlling session state and session closing (it triggers cleanup of session attributes);

Spring:: MVC :: @RequestMapping

Supported method return type:

- ModelAndView: for returning model and View name;
- Model: for returning model. View name will be determined through RequestToViewNameTranslator ;
- Map is analogous to Model ;
- View is a view object. Model will be automatically inserted (changes introduced to model with handler argument will be inserted);
- String is a logical view name that is handled by ViewResolver;
- void: if method handles a response itself or if view name is determined through RequestToViewNameTranslator;
- If the method is annotated through @ResponseBody, the return type is written to the response HTTP body;
- Any other return type is considered to be a single model attribute using name specified through @ModelAttribute at the method level or automatically generated based on the return type;

Spring:: MVC :: @ModelAttribute

Two use cases:

- **Annotation on method** is used to populate the model with needed attributes;
- **Annotation on method argument** indicates that model attribute shall be bound to corresponding parameter. It is a common way to access data inputted to the field by user, for example:

Invoking @ModelAttribute methods is done **before** invoking handler that populates the model.

Best Practices: in @ModelAttribute methods check for populated model attribute and return the given one

Spring:: MVC :: @SessionAttribute

@SessionAttribute is specified at the level of controller class;

Declares which model attributes should be stored in session for transferring between requests;

Spring:: MVC :: @SessionAttribute

```
@Controller
@RequestMapping("/owners/{ownerId}/pets/{petId}/edit")
@SessionAttributes("pet")
public class EditPetForm {
    @ModelAttribute("types")
    public Collection<PetType> populatePetTypes() {
        return this.clinic.getPetTypes();
    }
    @RequestMapping(method = RequestMethod.POST)
    public String processSubmit(
            @ModelAttribute("pet") Pet pet,
            BindingResult result, SessionStatus status) {
        new PetValidator().validate(pet, result);
        if (result.hasErrors()) {
            return "petForm";
        }
        else {
            this.clinic.storePet(pet);
            status.setComplete();
            return "redirect:owner.do?ownerId=" + pet.getOwner().getId();
        }
    }
}
```

Spring:: MVC :: Exceptions

- Handling exceptions:
  - `HandlerExceptionResolver` interface and one ready `SimpleMappingExceptionResolver` implementation.
  - Mapping between exception classes and view names based on the `Properties`.
  - Catches exceptions thrown by handlers and passes them via model.
  - More flexible mechanism as compared to **web.xml**, as far as its own implementation can be done.

Spring:: MVC :: Exceptions

Handling exceptions. Example:

```
<bean id="exceptionResolver"
      class="org.springframework.web.servlet.
             handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <value>
            java.lang.Throwable=error
            java.lang.SecurityException=error403
        </value>
    </property>
</bean>
```

Spring:: MVC :: @ExceptionHandler

- Within a controller you can specify which method is invoked when an exception of a specific type is thrown during the execution of controller methods
- Method arguments are equal to handler arguments

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

Spring:: MVC :: View: Basic Concepts

- Render model built in controller;
- Out of the box, Spring enables you to use JSPs, Velocity templates, XSLT views, **`View/Tiles`** interface implementations;
- Interface implementations of **`ViewResolver`** are used for mapping between view names (with locales) and view objects;
- Currently, the popular view is **`Tiles`**

Spring:: MVC :: ViewResolver

ViewResolver is used for resolving views based on the results of invoking controller handlers

Must resolve view either explicitly (e.g., by returning view name, view or ModelAndView) or implicitly (i.e., based on conventions)

Standard resolvers:

- InternalResourceViewResolver: mapping between view names and internal application resource, e.g., JSP;
- XmlViewResolver, ResourceBundleViewResolver –mapping based on .xml- or .property files (latter supports i18n) ;
- BeanNameViewResolver: mapping based on bean name. Doing so requires that all views should be created as beans in the contexts;
- UrlBasedViewResolver: direct resolution of view name to URL;

Spring:: MVC :: ViewResolver

Resolvers for different view technologies:

- `FreeMarkerViewResolver`
- `VelocityViewResolver`
- `VelocityLayoutViewResolver`
- `JasperReportsViewResolver`
- `XsltViewResolver`

Spring:: MVC :: ViewResolver

Configuration example:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
  InternalResourceViewResolver">

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

Spring:: MVC :: ViewResolver

- All default resolvers of **`ViewResolver`** cache **`view`**
- It is possible to turn off the cache by setting the **`cache`** property to **`false`**.
- On a single occasion you can enforce a certain view resolution by using **`removeFromCache(String viewName, Locale loc)`** method.

Spring:: MVC :: ViewResolver

- All view resolvers implement `Ordered` interface. This allows to have several resolvers that work in a certain sequence.
- For this reason `order` property should be set. If not specified explicitly, such a resolver will work the last.
- This may be necessary in some cases to redefine certain views if a single resolver doesn’t support all `View` implementations used.

Spring:: MVC :: ViewResolver

- Example:
  - **applicationContext.xml:**

```
<bean id="jspViewResolver"
class="org.springframework.web.servlet.view.
  InternalResourceViewResolver">
    <property name="viewClass”
value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
<bean id="excelViewResolver“
  class="org.springframework.web.servlet.view.
  XmlViewResolver">
    <property name="order" value="1"/>
    <property name="location" value="/WEB-INF/views.xml"/>
</bean>
```

Spring:: MVC :: I10n

- For JSP, use **`spring:message`** custom tag. As a substitute for entries, you specify entry keys stored in **`properties`** files;
- Define **`MessageSource`** in a context;
- Entries text should be carried to **`properties`** files named according to supported locales;
- Specify locale resolution, a **`LocaleResolver`** implementation, in the context (this is optional, because there is a default resolver);

Spring:: MVC :: l10n

```
<%@ taglib prefix="spring"
          uri="http://www.springframework.org/tags" %>
<html>
    <head>
        <title>
            <spring:message code="error.page"/>
        </title>
    </head>
    <body>
        <h1><spring:message code="exception.thrown"/></h1>
        . . .
    </body>
</html>
```

Spring:: MVC :: l10n

applicationContext.xml:

```
<bean id="messageSource"
class="org.springframework.context.support.
  ResourceBundleMessageSource">
    <property name="basename" value="messages"/>
</bean>
```

messages_ru_RU.properties:

```
exception.thrown
error.page
```

Spring:: MVC :: l10n :: LocaleResolver

- **`AcceptHeaderLocaleResolver`** (default resolver): inspects “accept-language” header in the HTTP request;
- **`CookieLocaleResolver`**: inspect a cookie to retrieve locale;
- **`SessionLocaleResolver`**: retrieves locales from the sessions;
- **`FixedLocaleResolver`**: retrieves locale specified when bean is configured;

Spring:: MVC :: l10n :: LocaleResolver

You can:

- Customize browser (if AcceptHeader resolver is used) ;
- Invoke method RequestContextUtils.getLocaleResolver(request).setLocale(request, response, locale). A cookie or session attribute will be created;
- Directly in controller code through Locale
- Specify for HandlerMapping interceptor like LocaleChangeInterceptor. Specify locale as request parameter (the easiest way of language switch support);

Spring:: MVC :: l10n :: LocaleResolver

```
<bean id="localeChangeInterceptor"
class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="siteLanguage"/>
</bean>

<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>
<bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="localeChangeInterceptor"/>
        </list>
    </property>
    <property name="mappings">
        <value>/**/*.view=someController</value>
    </property>
</bean>
```

Spring:: MVC :: Themes

A theme is a collection of static resources that affect the visual style of the application.

Generally, it is images and style sheets (CSS).

Spring:: MVC :: Themes

You can:

- Use **`spring:theme`** tag in pages instead of hard links to resources;
- Specify **`ThemeSource`** in the context;
- Carry links to different resource (images, style sheets) to properties files that are named in accordance with themes;
- Define theme resolution in the context: **`ThemeResolver`** implementation;

Spring:: MVC :: Themes

Example of JSP file:

```
<%@ taglib prefix="spring"
  uri=http://www.springframework.org/tags %>
<html>
    <head>
        <link rel="stylesheet" type="text/css"
              href="<spring:theme code="css"/>" />
    </head>
    <body background="<spring:theme code="bg"/>">
        . . .
    </body>
</html>
```

Spring:: MVC :: Themes

Example:

applicationContext.xml:

```
<bean id="themeSource"
class="org.springframework.ui.context.support.ResourceBundleThemeSource"/>
```

cool.properties:

```
css=/themes/cool/style.css
bg=/themes/cool/img/bg.jpg
```

Spring:: MVC :: Themes

`CookieThemeResolver`: theme name is retrieved from cookie;

`SessionThemeResolver`: theme name is retrieved from session;

`FixedThemeResolver`: retrieves theme name specified during bean configuration;

Spring:: MVC :: Themes

One-time theme change:

- Invoke **`RequestContextUtils.getThemeResolver(request).setThemeName(themeName)`** method. A cookie or session attribute will be created;
- Directly in controller code;
- Specify for **`HandlerMapping`** an interceptor like **`ThemeChangeInterceptor`**. Specify theme name as request parameter;

Spring:: MVC :: Themes :: ThemeResolver

```
<bean id="themeChangeInterceptor"
class="org.springframework.web.servlet.theme.ThemeChangeInterceptor">
    <property name="paramName" value="lookAndFeel"/>
</bean>
<bean id="themeResolver"
    class="org.springframework.web.servlet.theme.CookieThemeResolver"/>

<bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping>
    <property name="interceptors">
        <list>
            <ref bean="themeChangeInterceptor"/>
        </list>
    </property>
    <property name="mappings">
        <value>/**/*.view=someController</value>
    </property>
</bean>
```

Spring:: MVC :: Themes + l10n

Sometimes in the themes you want images to have text.

It is enough to create additional properties files that are named according to locales for each theme, for example:

```
cool.properties
cool_ru_RU.properties
cool_en_US.properties
dark.properties
dark_ru_RU.properties
dark_en_US.properties
```

Spring:: MVC :: Multipart

- Multipart support handles file uploads from html forms
- Form attribute enctype shall be placed to multipart/form-data
- Files uploaded from

```
 <input type=“file” />
```

- By default, Spring does no multipart handling (some developers want to do it themselves)

Spring:: MVC :: Multipart

`org.springframework.web.multipart` package:

- MultipartResolver interface
- CommonsMultipartResolver and CosMultipartResolver implementations
- Necessary libraries:
  - commons-io.jar and commons-fileupload.jar ;
  - or cos.jar ;

Spring:: MVC :: Multipart

- In application context create bean: `MultipartResolver` implementation;
- Each request is inspected for multiparts; if multipart is found the request is wrapped in a `MultipartHttpServletRequest`

- Example:

```
<bean id="multipartResolver”
 class="org.springframework.web.multipart.
     commons.CommonsMultipartResolver”>
    <property name="maxUploadSize" value="100000"/>
</bean>
```

Spring:: MVC :: Multipart :: MultipartResolver

- `maxInMemorySize`: set the maximum allowed size before uploads are written to disk (by default, 10KB) ;
- `maxUploadSize`: set the maximum total size (by default, it is not limited);
- `uploadTempDir`: set the temporary directory where uploaded files get stored. By default, container's temporary directory;
- `defaultEncoding`: set encoding to use for parsing headers of individual parts. Has less priority than one explicitly specified in request;

Spring:: MVC :: Multipart

Form example:

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

<form:form action="someRequest.do” enctype="multipart/form-data”>
        <input type=“text” name=“name”/>
        <input type="file" name="file"/>
        <input type="submit"/>
</form:form>
```

Spring:: MVC :: Multipart

Controller example:

```
@Controller
public class FileUpoadController {

    @RequestMapping(value = "/form", method = RequestMethod.POST)
    public String handleFormUpload(@RequestParam("name") String name,
        @RequestParam("file") MultipartFile file) {

        if (!file.isEmpty()) {
            byte[] bytes = file.getBytes();
            // store the bytes somewhere
           return "redirect:uploadSuccess";
       } else {
           return "redirect:uploadFailure";
       }
    }
}
```

Spring:: MVC :: Tag Library

There are 2 libraries:

- Spring
- Spring-form

Their main purpose is to facilitate binding of given API JSPs, form objects and form elements and vice versa;

Almost every tag has an attribute **`htmlEscape`** that allows to control HTTP/JS code shielding. **`defaultHtmlEscape`** parameter from **`web.xml`** is taken into account as well.

Spring:: MVC :: Tag Library

Connection is done as follows

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


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

Spring:: MVC :: Tag Library

Themes and locales:

- **`theme`** tag: using **`ThemeResolver`** and **`ThemeSource`** substitutes itself for resource path whose name is specified in code attribute.
- **`message`** tag: using **`LocaleResolver`** and **`MessageSource`** substitutes itself for localized message whose name is specified in code attribute.

Spring:: MVC :: Tag Library

`bind` tag:

- `path` attribute is the path to property of bean in dot notation that is to be bound;
- Attempts to perform binding, provides `BindStatus` object for further analysis;
- Properties `error`, `errors`, `errorCodes`, `errorsMessage`, `expression`, `path ;`
- `value`, `valueType ;`

Spring:: MVC :: Tag Library

`hasBindErrors` tag:

- **`name`** attribute is the name of the bean whose fields were bound previously;
- If binding errors are available the tag content will be rendered. Inside the tag, errors are available through **`Errors`** object.

Spring:: MVC :: Tag Library

Example of **`bind, hasBindErrors`** tags:

```
<form method="post" action="testPage.html">
    <spring:bind path="commandBean.fieldOne">
        <input type="text" name="fieldOne" value="${status.value}"/>
        <c:if test="${status.error}"> ... </c:if>
    </spring:bind>
    <spring:bind path="commandBean.fieldTwo">
        <input type="text" name="fieldTwo" value="${status.value}" />
        <c:if test="${status.error}">
            <c:forEach items="${status.errorMessages}" var="msg">${msg}<br/>
            </c:forEach>
        </c:if>
    </spring:bind>
    <spring:hasBindErrors name="commandBean">
         There were ${errors.errorCount} error(s) in total:
         <c:forEach var="errMsgObj" items="${errors.allErrors}">
             <spring:message code="${errMsgObj.code}"
                             text="${errMsgObj.defaultMessage}"/><br/>
```

Spring:: MVC :: Tag Library

`Transform` tag :

- **`value`** attribute: object you want to have transformed.
- Transforms the object to string using **`PropertyEditors`**, so that the reverse transformation will be available when sending the form.
- Can be only used inside **`bind`** tag.

Spring:: MVC :: Tag Library

Example of **`transform`** tag :

```
<form method="post">
  <spring:bind path="contract.contractType">
    <select name="<c:out value="${status.expression}"/>">
      <c:forEach items="${contractTypes}" var="type">
        <spring:transform value="${type}" var="typeString"/>
        <option value="<c:out value="${typeString}"/>"
          <c:if test="${status.value == typeString}"/> selected</c:if>>
          <c:out value="${typeString}"/>
        </option>
      </c:forEach>
    </select>
  </spring:bind>
</form>
```

Spring:: MVC :: Tag Library

`form` tag :

- Has attribute `commandName`: name of the form object. When describing paths of form elements, the first element (this is the name) is omitted..
- Generates html element `<form>`

Spring:: MVC :: Tag Library

`errors` tag :

Has attributes:

- **`commandName`**: property for which binding errors are rendered (* = all errors) ;
- **`cssClass`**: css class bound to block;

- Generates html element **`<span>`** that renders binding errors through <br/> ;
- You can create several of them in different page areas for rendering binding errors of specific properties;

Spring:: MVC :: Tag Library

Example of **errors** tag :

```
<form:form action="someRequest.do" commandName="someBean">
    <form:errors path="*" cssClass="errorText" />
    . . .
<form:form>
```

Spring:: MVC :: Tag Library

Input field:

  - **`input`** tag
  - **`hidden`** tag
  - **`textarea`** tag
  - **`password`** tag

- Generate **`input`** of specific type;
- **`value`** attribute indicates property value of form object, which is specified in **`path`** attribute;
- **`name`** attribute indicates property path so that binding is available during sending;

Spring:: MVC :: Tag Library

Example of input field tags:

```
<form:form action="someRequest.do" commandName="user">
    <form:input path="name" />
    <form:password path="passwd" />
    <form:hidden path="id" />
    <form:textarea path="notes" />
    . . .
<form:form>
```

Spring:: MVC :: Tag Library

Radiobuttons:

- `radiobutton` tag
- `checkbox` tag

- Generates corresponding html element;
- Used more than once with constant `path` value, but with different `value`. Once bound, the one whose property value specified in `path` matches `value` field, is selected.

Spring:: MVC :: Tag Library

Example of **the radiobutton** tags:

```
<form:form action="someRequest.do" commandName="user">
    Intrests:
    Pets  <form:checkbox path="prefs.interests"
               value="pets"/>
    Games <form:checkbox path="prefs.interests"
               value="games"/>
    . . .
    Gender:
    <form:radiobutton path="sex" value="M"/>
    <form:radiobutton path="sex" value="F"/>
    . . .
<form:form>
```

Spring:: MVC :: Tag Library

Radiobutton groups:

- `radiobuttons` tag
- `checkboxes` tag

Attributes:

- `path`: path to property of form object;
- `items`: `java.util.Map`, model element;

Generate corresponding html elements

`key` from `Map` is used as `value` element. `value` from `Map` is used as a text. The one whose property value specified in `path` matches `value` field, is selected.

Spring:: MVC :: Tag Library

Example of the radiobutton groups tags :

```
<form:form action="someRequest.do" commandName="user">
    Interests:
    <form:checkboxes path="prefs.interests"
                  items="${availInterests}"/>
    Gender:
    <form:radiobuttons path="sex" items="${availGenders}"/>
    . . .
<form:form>
```

Spring:: MVC :: Tag Library

The list:

- `select` tag
- `options` tag
- `option` tag

Binding logic is equal to radio button logic.

Spring:: MVC :: Tag Library

Examples of list tags :

```
<form:form action="someRequest.do" commandName="user">
        Gender:
        <form:select path="sex" />
            <form:options items="${availGenders}"/>
        </form:select>
<form:form>


<form:form action="someRequest.do" commandName="user">
        Gender:
        <form:select path="sex" />
            <form:option value="M">Male</form:option>
            <form:option value="F">Female</form:option>
        </form:select>
<form:form>
```

Exercises

№: 9 : Web application development based on Spring MVC

- 45 min for practice;
- 15 min for discussion;

TODO

Validation

JSR-303

Any questions!?