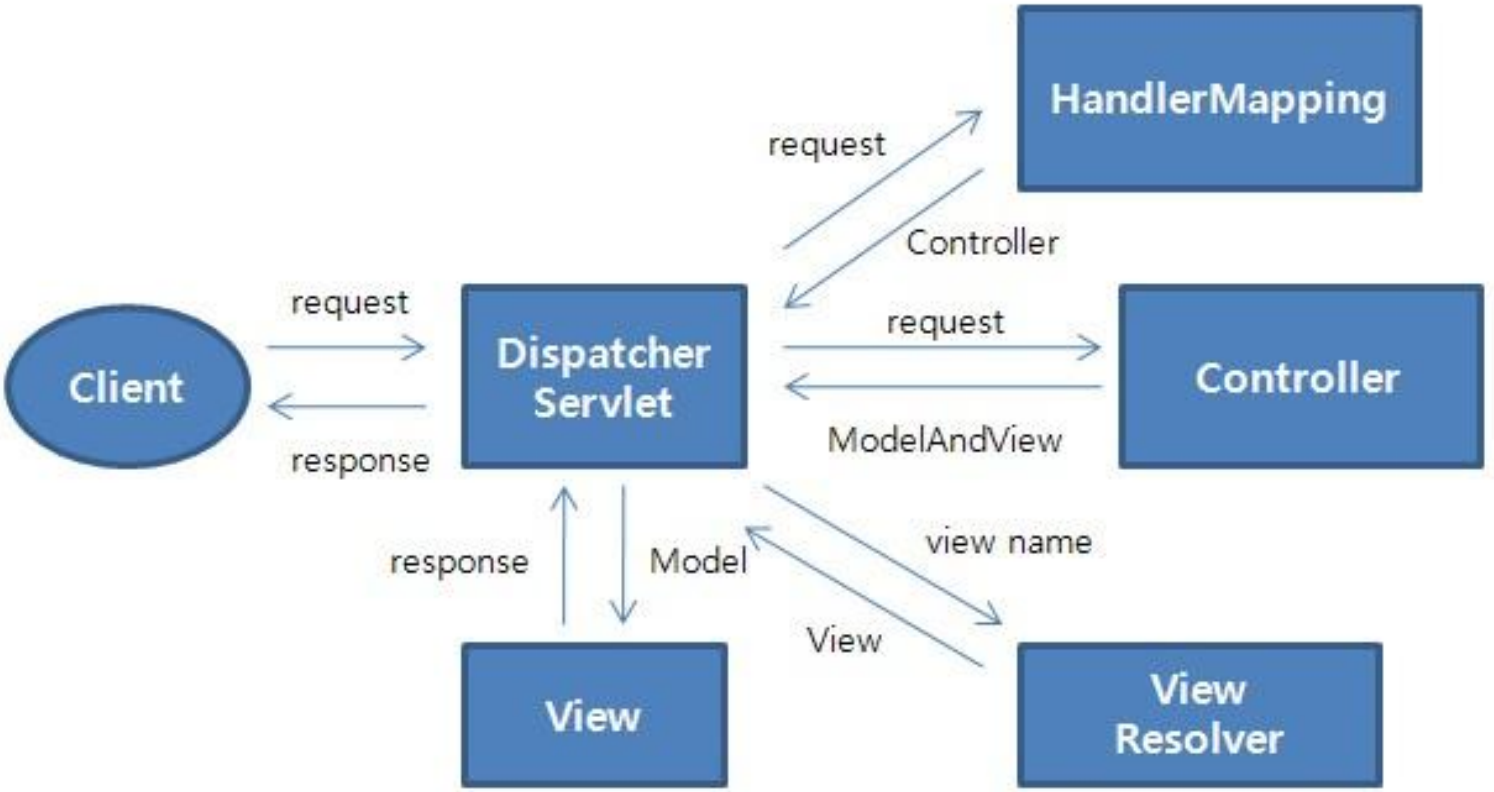
Resolvers & Upload & Exceptions & Internationalization & Tiles

Harmonizing Diversity

Spring MVC Flow



Spring MVC View Resolvers

- Flexible View Resolving Mechanism
- Resolve logical String-based view names to View types.
- Many out-of-the-box implementations[examples]:

1. **UrlBasedViewResolver**-This directly resolves a view name to a URL without any explicit mapping. The view names can be the URL themselves or a prefix or suffix can be added to get the URL from the view name.
2. **InternalResourceViewResolver**-This is a subclass of **UrlBasedViewResolver**. Out-of-the-box support for JSP
3. **FreeMarkerViewResolver**-This is a subclass of **UrlBasedViewResolver** that supports FreeMarkerView and its subclasses.
4. **VelocityViewResolver**-This is a subclass of **UrlBasedViewResolver** that supports VelocityView and its subclasses.
5. **ContentNegotiatingViewResolver**-This is an implementation of a view resolver based on the request file name or Accept header – mime-type. This class delegates view resolution to other view resolvers that are configured.

Multiple View Resolvers Configuration

```
<!-- lower order value has a higher priority -->
<bean
class="org.springframework.web.servlet.view.InternalResourceView
  Resolver">
  <property name="prefix" value="/WEB-INF/views/" />
  <property name="order" value="4" />
</bean>

<bean class="org.thymeleaf.spring4.view.ThymeleafViewResolver">
  <property name="templateEngine" ref="templateEngine" />
  <property name="viewNames" value="*.html" />
  <property name="order" value="3" />
</bean>
```

Spring MVC Views

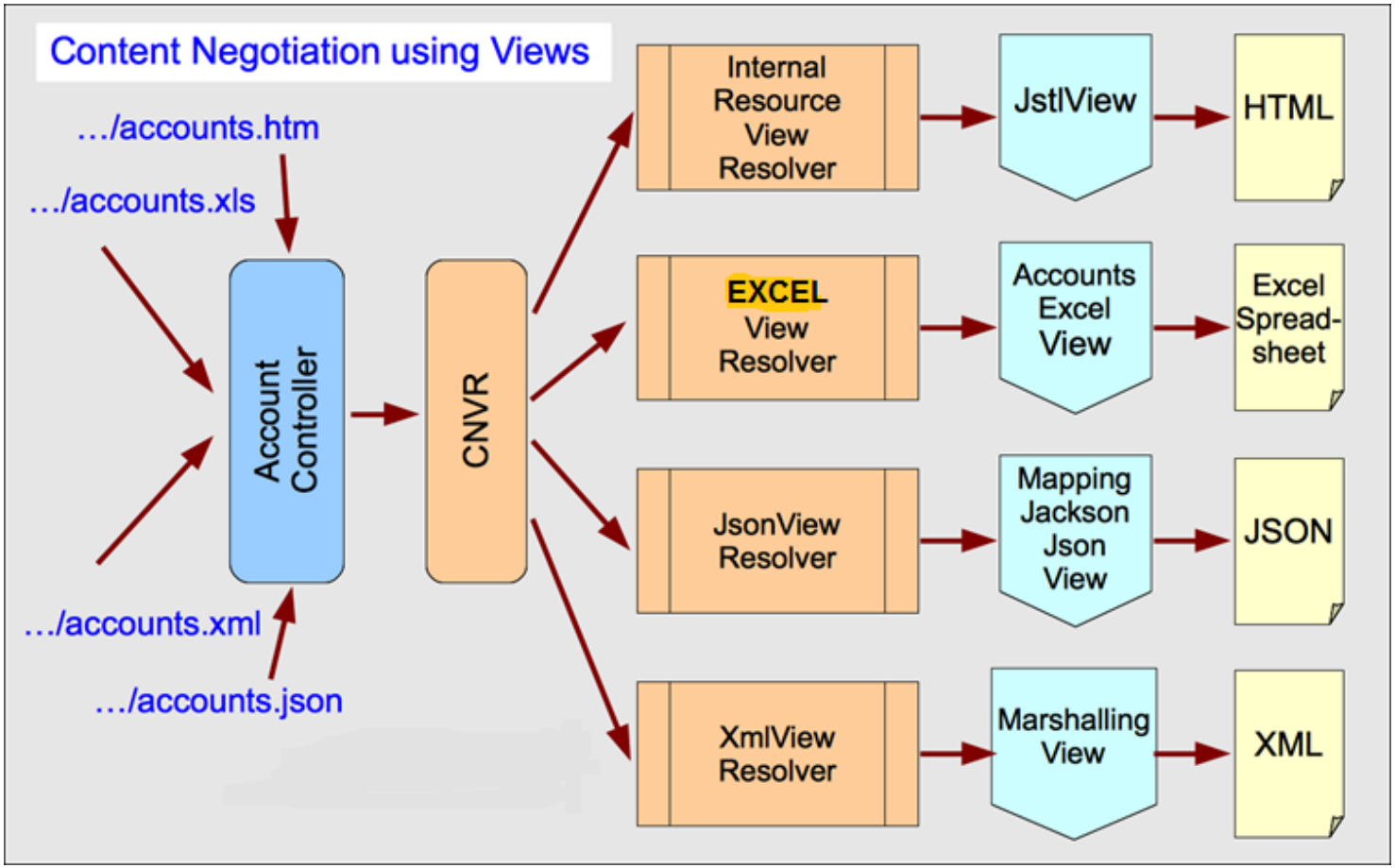
- Spring has flexible view support through the View Interface class
- Out-of-the-box view support for:
  - JspView - InternalResourceViewResolver
  - JSON - ContentNegotiatingViewResolver **
  - XML - ContentNegotiatingViewResolver **
  - PDF - ContentNegotiatingViewResolver **
  - Excel - ContentNegotiatingViewResolver **
  - Tiles - TilesViewResolver
  - Velocity - VelocityViewResolver
  - FreeMarker - FreeMarkerViewResolver

  - Redirect - InternalResourceViewResolver
  - Forward - InternalResourceViewResolver

** "Candidates"

ContentNegotiatingViewResolver

- Scenario based on "file extension" in URL:



- Single Controller method..multiple views…CNVR

Resolving Image files for uploading

- HTTP multipart request is used by browsers to upload files/data to the server.
- Spring good support HTTP multipart request:
  - CommonsMultipartResolver
- DispatcherServlet XML/Java config:

```
<bean id="multipartResolver"
  class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
  <property name="maxUploadSize" value="10240000"/>
</bean>
```

```
@Bean
public CommonsMultipartResolver multipartResolver() {
  CommonsMultipartResolver resolver = new CommonsMultipartResolver();
  resolver.setDefaultEncoding("utf-8");
  resolver.setMaxUploadSize(10240000);
  return resolver;
}
```

Resolving Image files for uploading[Cont.]

- JSP configuration

```
<form:form modelAttribute="newProduct" class="form-horizontal"
  enctype="multipart/form-data">
  <div class="form-group">
    <label class="control-label col-lg-2" for="productImage">
      <spring:message code="addProduct.form.productImage.label"
        />
    </label>
    <div class="col-lg-10">
      <form:input path="productImage" id="productImage"
      type="file" class="form:input-large" />
    </div>
  </div>
</form:form>
```

Content-Type: multipart/form-data

Resolving Image files for uploading [Cont.]

```
@XmlRootElement
public class Product implements Serializable {

   @Pattern(regexp = "P[1-9]+", message =
      "{Pattern.Product.productId.validation}")
   @ProductId
   private String productId;

   @Size(min = 4, max = 50, message =
      "{Size.Product.name.validation}")
   private String name;

   @JsonIgnore
   private MultipartFile productImage;
}
```

Resolving Image files for uploading[Cont.]

- Saving image in Controller:

```
@RequestMapping(method = RequestMethod.POST)
public String processAddNewProductForm(@Valid @ModelAttribute("newProduct") Product newProduct,
BindingResult result, Model model, HttpServletRequest request) {

    MultipartFile productImage = newProduct.getProductImage();
    String rootDirectory = request.getSession().getServletContext().getRealPath("/");
    if (productImage != null && !productImage.isEmpty()) {
        try {
            productImage.transferTo(
            new File(rootDirectory + "resources\\images\\" + newProduct.getProductId() +
               ".png"));
        } catch (Exception e) {
           throw new RuntimeException("Product Image saving failed", e);
        }
    }

    productService.addProduct(newProduct);
    return "redirect:/products";
}
```

Main Point

- Spring MVC Views and View Resolvers offers a variety of ways to manage the presentation of data.
- *Life is the expression of the field of all possibilities resulting in a veritable explosion of variety in nature*

Handler Exception Resolver

- `HandlerExceptionResolver` interface
  - Used to resolve exceptions during Controller mapping & execution
  - Two Default implementations ["out of the box"]:
    - `ResponseStatusExceptionResolver` – supports `@ResponseStatus`
    - `ExceptionHandlerExceptionResolver` – supports `@ExceptionHandler`

- Exceptions can be handled EITHER individually OR Globally across ALL Controllers with `@ControllerAdvice` "interceptor"

ResponseStatusExceptionResolver

- Marks a method or exception class with the status code and reason that should be returned. "Customizes" exceptions as HTTP status codes
- The status code is applied to the HTTP response when the handler method is invoked, or whenever said exception is thrown.
- Could reside on Exception OR in @ControllerAdvice

ResponseStatusExceptionResolver (cont.)

- Customized Exception:

```
@ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "No products found under this
    category.")
public class NoProductsFoundUnderCategoryException extends RuntimeException {

    private String message;

    public NoProductsFoundUnderCategoryException(String message) {
        this.message = message;
    }
}
```

- ProductController.java

```
@RequestMapping("/products/{category}")
public String getProductsByCategory(Model model, @PathVariable String category) {
    List<Product> products = productService.getProductsByCategory(category);
    if (products == null || products.isEmpty()) {
    throw new NoProductsFoundUnderCategoryException();
    }
    model.addAttribute("products", products);
    return "products";
}
```

ExceptionHandlerExceptionResolver

- Method identified as `ExceptionHandler` for exception resolution
- Could reside in EITHER `ProductController` OR `@ControllerAdvice`

```
@ExceptionHandler(NoProductsFoundUnderCategoryException.class)
public ModelAndView handleError(HttpServletRequest req,
   NoProductsFoundUnderCategoryException exception) {
    ModelAndView mav = new ModelAndView();
    mav.addObject("msg", exception.getMessage());
    mav.addObject("exception", exception);
    mav.addObject("url", req.getRequestURL());
    mav.setViewName("noProductFound");
    return mav;
}
```

- Exception JSP: `noProductFound.jsp`

```
<div class="jumbotron">
    <div class="container">
        <h1 class="alert alert-danger">{msg}</h1>
    </div>
</div>
```

@ControllerAdvice

- Indicates the annotated class assists a "Controller"
- Works across ALL controllers
- It is typically used to define `@ExceptionHandler`, `@InitBinder`, and `@ModelAttribute` methods that apply to all `@RequestMapping` methods.

- Handle all `ProductNotFoundException` throws in any class:

```
@ControllerAdvice
public class ControllerExceptionHandler {

   @ExceptionHandler(ProductNotFoundException.class)
   public ModelAndView handleError(HttpServletRequest req,
     ProductNotFoundException exception) {
      ModelAndView mav = new ModelAndView();
      mav.addObject("invalidProductId", exception.getProductId());
      mav.addObject("exception", exception);
      mav.addObject("url", req.getRequestURL() + "?" +
        req.getQueryString());
      mav.setViewName("productNotFound");
      return mav;
   }
}
```

Main Point

- A well-defined exception and error handling approach is important for simplifying the development of web applications.
- *The removal of obstacles is an important aspect of the process of evolution.*

Internationalization

- i18n – 'i'+ 18 chars + 'n' == internationalization
- Support for multiple languages & data format with code rewrite
- Examples:
  - zh Chinese nl Dutch
  - hi Hindi el Greek
  - ja Japanese fr French
- L10n = 'l'+10 chars + 'n' = localization
- Support locale-specific [geographic/region/country] information

Egypt	EG	Libya	LY	China	CN
India	IN	Taiwan	TW		
Mynmar	MM	Mongolia	MN		

Java Locale class

- Locale(String language)
- Locale(String language, String Country)
- Locale(String language, String Country, String variant)

- Variant is browser specific code [windows, MAC, etc.]

- Message are stored in ".properties files indicating Locale
- E.g. messages_zh.properties
- Optionally messages_zh_CN.properties

Locale Resolvers

- Browser's Accept-Language header

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver">
  <property name="defaultLocale" value="en_US"/>
</bean>
```

- Session
  - uses a locale attribute in the user's session

```
<bean id="localeResolver"
  class="org.springframework.web.servlet.i18n.SessionLocaleResolver">
  <property name="defaultLocale" value="en_US"/>
</bean>
```

```
@Bean
public LocaleResolver localeResolver() {
    SessionLocaleResolver resolver = new SessionLocaleResolver();
    resolver.setDefaultLocale(new Locale("en"));
    return resolver;
}
```

- Cookie
  - uses a cookie sent back to the user

```
<bean id="localeResolver"
  class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
  <property name="defaultLocale" value="en_US"/>
</bean>
```

LocaleChangeInterceptor

- Used to handle Cookie or Session locale resolvers AUTOMATICALLY

```
<mvc:interceptors>
    <bean class=
        "org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
        <property name="paramName" value="language"/>
    </bean>
</mvc:interceptors>
```

This is the parameter that the interceptor looks for...

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
   LocaleChangeInterceptor localeChangeInteceptor = new
      LocaleChangeInterceptor();
   localeChangeInteceptor.setParamName("language");
   registry.addInterceptor(localeChangeInteceptor);
}
```

Tiles

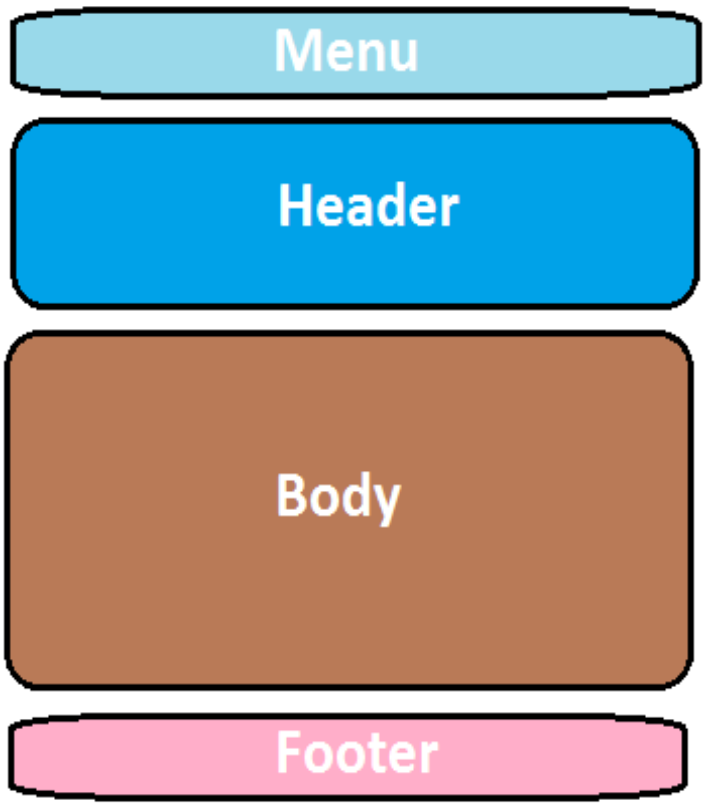
- **Composite View Pattern**
  - create pages using a consistent structure
  - pages share the same layout
  - individual pages differ in segments
  - segment placement maintains positional consistency across all the site

Tiles View Resolver Configuration

```
<bean id="tilesViewResolver"
      class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="viewClass"
       value="org.springframework.web.servlet.view.tiles3.TilesView" />
    <property name="order" value="-2" />
 </bean>

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

Sample Template [layoutTemplate.jsp]




```
<title><tiles:insertAttribute name="title" />
<body>
   <ul class="nav nav-pills pull-right">
    <tiles:insertAttribute name="menu" />
   </ul>
   <h3 class="text-muted">Web Store</h3>

   <h1>
    <tiles:insertAttribute name="header" />
   </h1>
   <p>
      <tiles:insertAttribute name="subHeader" />
   </p>

   <div class="row">
     <tiles:insertAttribute name="body" />
   </div>

   <div class="footer">
    <tiles:insertAttribute name="footer" />
   </div>
</body>
```

Example Tiles Definition File

```
<tiles-definitions>

   <definition name="baseLayout" template="/WEB-
  INF/layouts/template/baseLayout.jsp">
       <put-attribute name="title" value="Sample Title" />
       <put-attribute name="heading" value="" />
       <put-attribute name="tagline" value="" />
       <put-attribute name="navigation" value="/WEB-
  INF/layouts/template/navigation.jsp" />
       <put-attribute name="content" value="" />
       <put-attribute name="footer" value="/WEB-
  INF/layouts/template/footer.jsp" />
   </definition>

</tiles-definitions>
```

Example Tiles Pages

```
<tiles-definitions>

   <definition name="welcome" extends="baseLayout">
      <put-attribute name="title" value="Products" />
      <put-attribute name="heading" value="Products" />
      <put-attribute name="tagline" value="All the
  available products in our store" />
      <put-attribute name="content" value="/WEB-
  INF/views/products.jsp" />
   </definition>

</tiles-definitions>
```

Tile Wildcards

```
<!-- Wild Card for all controllers...
     access by /welcome for view folder JSP location
     product/products for view/product folder JSP location
 -->
<definition name="*/*" extends="baseLayout">
     <put-attribute name="title" value="{2}.title" />
     <put-attribute name="heading" value="{2}.heading" />
     <put-attribute name="tagline" value="{2}.tagline" />
     <put-attribute name="body" value="/WEB-INF/views/{1}/{2}.jsp" />
</definition>
```

Tiles Inline in JSP

```
<%@ taglib prefix="tiles" uri="http://tiles.apache.org/tags-tiles"%>
<html>
<tiles:insertDefinition name="baseLayout">

 <tiles:putAttribute name="title">welcome.title</tiles:putAttribute>
 <tiles:putAttribute name="heading">welcome.head</tiles:putAttribute>
 <tiles:putAttribute name="tagline">welcome.tag</tiles:putAttribute>
 <tiles:putAttribute name="body">
   Language : <a href="?language=en_US">English</a>|
                      <a href="?language=zh_CN">Chinese</a>
    <h3><spring:message code="welcome.spring" text="default text"/></h3>
   <p> Current Locale : ${pageContext.response.locale}</p>
 </tiles:putAttribute>

</tiles:insertDefinition>
```

Main Point

- All websites have something in common: they are made of pages that share similar structures.
- *A facet of SCI is that Order is found everywhere*