Spring MVC

Do Less, Accomplish More

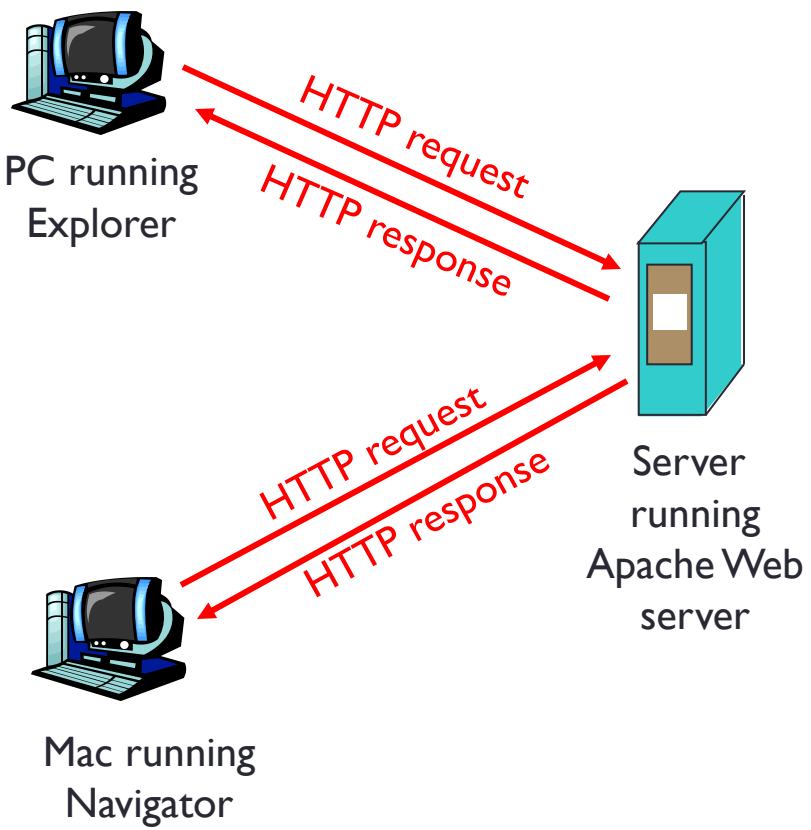
Web and HTTP Review

- A web page consists of
  - Content
  - Tags that operate on the content
  - Objects
- objects can be HTML files, JPEG images, Java applets, audio files, CSS files, etc…
- web page consists of a base HTML-file which includes several referenced objects
- each object is addressable by a URL

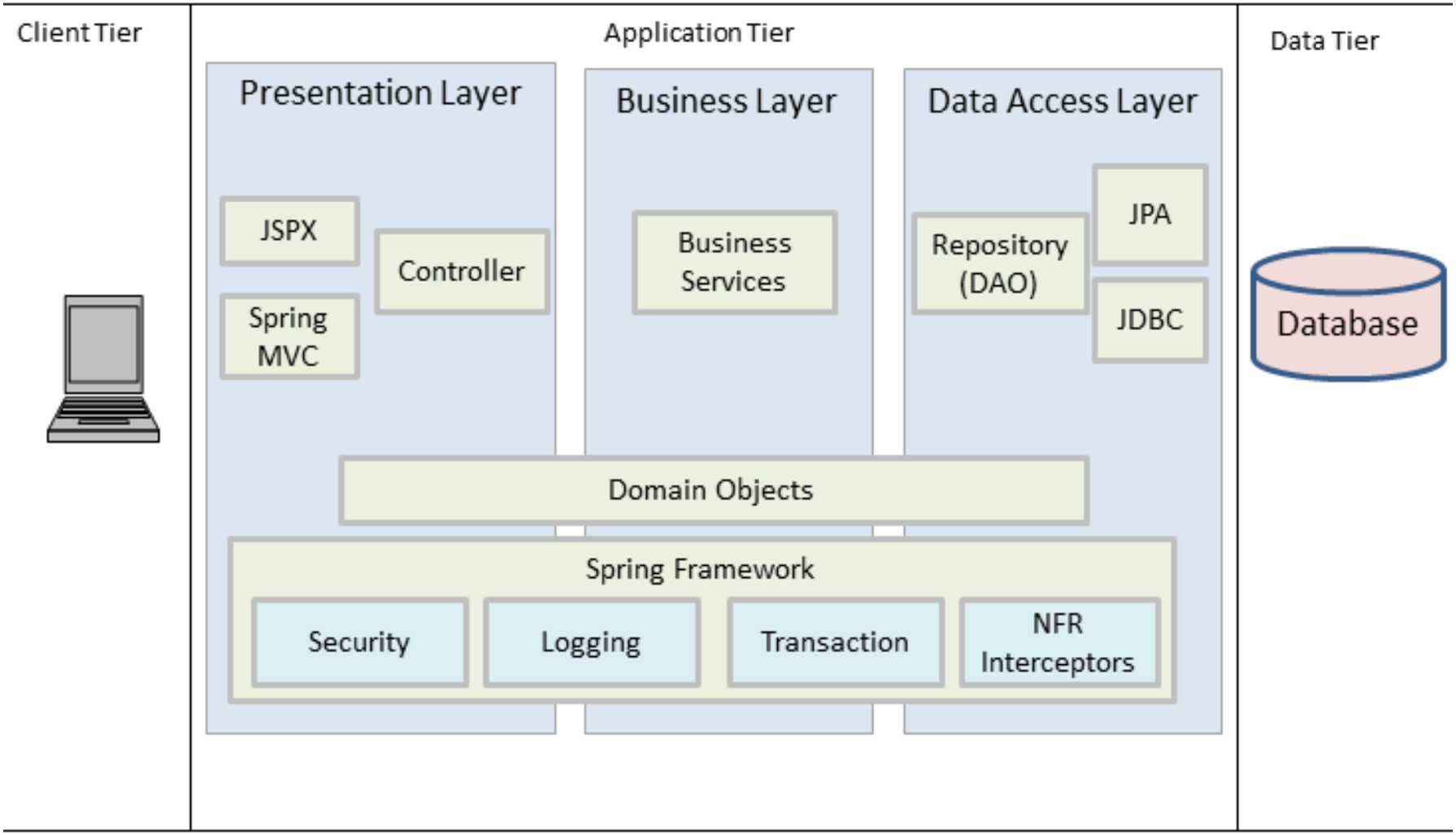
HTTP review

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, "displays" Web objects
  - *server:* Web server sends objects in response to requests



Spring MVC/Hibernate Architecture Basic Layers



JSF and Spring MVC

JSF is a **component-based** framework, and there are different libraries of powerful JSF components.

SpringMVC is not a component framework, but is **request-based**.

```
@RequestMapping(value = {"/","/product"}, method = RequestMethod.GET)
public String getAddNewProductForm( @ModelAttribute("newProduct")
Product product) {// get info to display and add to the Model Map
  return "ProductForm";}

@RequestMapping(value = "/product", method = RequestMethod.POST)
public String processAddNewProductForm @ModelAttribute("newProduct")
Product productToBeAdded) {
// set values in productToBeAdded and persist it

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

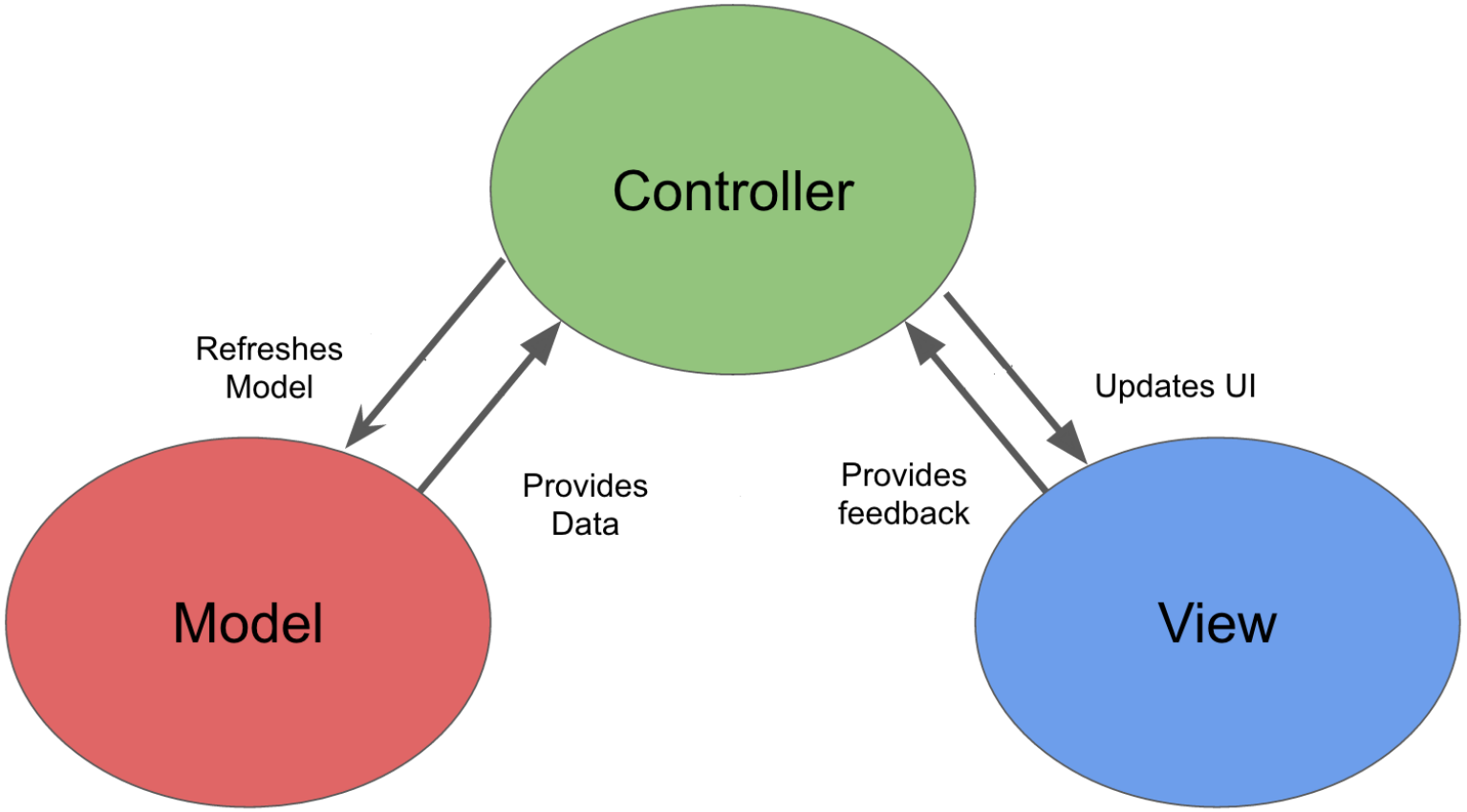
The main differences are:

JSF	SpringMVC
Component based - **Lots of powerful web components like calendars, tables, trees, datepickers, etc.** - **Not usually used for REST webservices**	Request based - No powerful web components - Well suited for REST webservices because it is request based
Uses XHTML with web components	Can be used with different view technologies like JSP, Velocity, FreeMarker, ThymeLeaf, JSON, XML, Excel, etc.
Embedded request handling	Explicit request handling

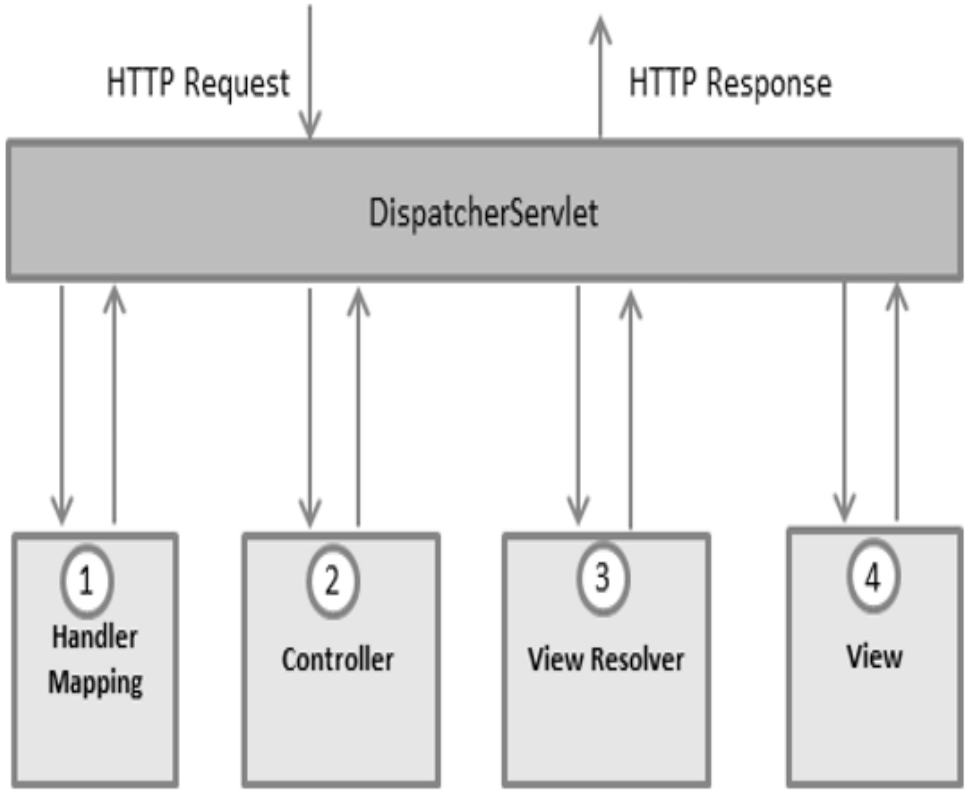
JSF and Spring MVC

- JSF – a good choice if you need a server side web framework with powerful web components. *If interested see Lesson10-2-JEE7 JSF in resources/lectures on sakai.*
- If you need a server side web framework with simple HTML controls, both will work but SpringMVC is simpler (if you are already using Spring)
- You can build a server-side web application with both frameworks, and they both have support for conversion, validation, scope, navigation, templates, ajax, etc.
- All Enterprise Systems use the MVC pattern.

MVC Pattern


Controller
Refreshes
Model
Updates UI
Provides
Data
Provides
feedback
Model
View

Spring DispatcherServlet Life Cycle



All client requests and responses go through the DispatcherServlet.

1)The DispatcherServlet uses the HandlerMapping to route the request to the correct Controller via the URL in the request.

2)The Controller gets the user input and will normally delegate the business logic to service objects.

The model is the info to be sent back to the DispatcherServlet from the Controller with the name of the View to render the info.

3)The DispatcherServlet uses the View Resolver to map the logical View name to the correct View

4)The View renders the model info into a response for the client's browser.

Configuring Your Web Application

SpringBoot will configure the Dispatcher Servlet and the View Resolver for us when we select: Spring Starter Project with web dependency.

If we want to set up the web config without SpringBoot we can configure the Dispatcher Servlet and then create a web config file :

for example in web.xml

```
<servlet-name>springmvc</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
                <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/config/springmvc-config.xml</param-value>
                        </init-param>
```

Web Config (cont)

- SpringBoot configures a View Resolver for us
- Without SpringBoot we can configure with a bean:

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

return "ProductForm";

resolves to:

/WEB-INF/jsp/ProductForm.jsp

The subsequent RequestDispatcher forward is done by the framework

View Mapping Example

```
@Controller
public class ProductController {
@Autowired
ProductService productService;
@Autowired
CategoryService categoryService;


@RequestMapping({"/listproducts"})
public String list(Model model) {
model.addAttribute("products", productService.getAllProducts());
return "products";
}
```

- Model = a map of Product objects
- productService will make the calls to our Product DAO
- The return is to our page products.jsp

Mapping Example [Cont.]

```
@RequestMapping(value = {"/","/product"}, method = RequestMethod.GET)
public String getAddNewProductForm(  @ModelAttribute("newProduct") Product
product) {
  List<Category> categories = categoryService.findAll();
  model.addAttribute("categories", categories);
  return "ProductForm";
}

@RequestMapping(value = "/product", method = RequestMethod.POST)
public String processAddNewProductForm( @ModelAttribute("newProduct") Product
product) {

 Category category = categoryService.findOne(product.getCategory().getId());
 product.setCategory(category);
 productService.addProduct(product);

    return "redirect:/products";
}
```

@RequestParam

- Placed on Method argument

http://localhost:8080/MemberSpringMVC/products/product?id=1

```
 @RequestMapping("/product")
public String getProductById(Model model, @RequestParam("id") Long id)
{
        Product product = productService.findOne(id);
```

Handling multiple values [e.g., multiple selection list]

http://localhost:8091/store/sizechoices?sizes=small&sizes=medium&sizes=large

```
  public String getSizes(@RequestParam(“sizes")String sizeArray[]
```

Spring MVC Data Binding

- Automatically maps request parameters to domain objects
- Simplifies code by removing repetitive tasks
- Built-in Data Binding handles simple String to data type conversions
- HTTP request parameters [String types] are converted to model object properties of varying data types.

Data Binding example

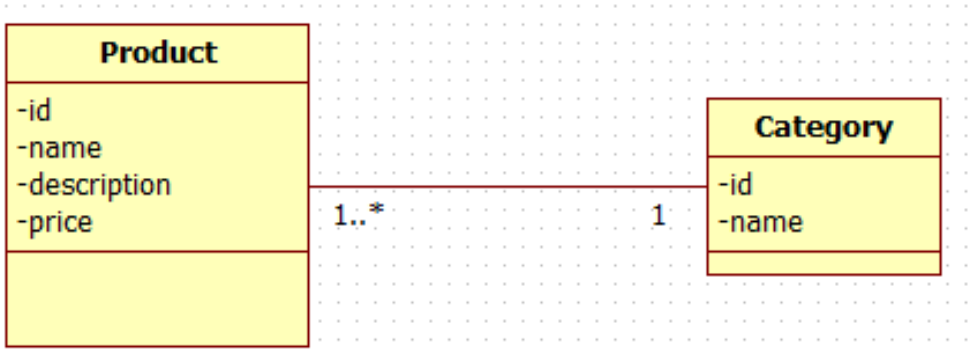
```
package app04a.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import app04a.domain.Product;

@Controller
public class ProductController {

    @RequestMapping(value={"/","/product"}, method = RequestMethod.GET)
    public String inputProduct() {
        return "ProductForm";
    }

    @RequestMapping(value="/product", method = RequestMethod.POST)
    public String saveProduct(Product product ) {
        return "ProductDetails";
    }
}
```

Data Binding example

➢Without data binding

```
Product product = new Product();
product.setName(request.getParameter(“name").trim());
product.setDescription(request.getParameter("description").trim());
try {
      price = request.getParameter(“price").trim());
product.setPrice(Float.parseFloat(price ));
} catch (NumberFormatException e) {
}
```

Data Binding - Relationships




```
<form action="product" method="post">
     <legend>Add a product</legend>
        <p>
     <label for="category">Category </label>
       <select name="category.id">
              <option value="-">--Select Category--</option>
           <c:forEach var="category" items="${categories}">
             <option value="${category.id}" >
           ${category.name}</option>
           </c:forEach>
       </select>

      <label for="name">Product Name: </label>
         <input type="text" id="name" name="name" >
```

Spring MVC Form Tag Library

- Facilitates the development of JSP pages
- Integrated with Spring MVC data binding features – bound to an object in the model and populated with values from the model attributes.
- Each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and intuitive to use.
- Access Form Tag Library:

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

Form Tag Library

Tag Name	Description
`form`	Renders an HTML form tag and exposes the binding path to the other form tags.
`input`	Renders an input element with the default type of `text`. The type of the input element can be can be (optionally) specified (like `email`, `date` etc.) . Note that you can't use `radiobutton` or `checkbox` for those types.
`password`	Renders an input element of type `password`.
`hidden`	Renders an input element of type `hidden`.
`select`	Renders an HTML select element. The `option` and/or `options` tag are used to specify the options to render.
`option`	Renders a single HTML option element inside a select element.
`options`	Renders a collection of HTML option elements inside a select element.
`radiobutton`	Renders an HTML input element of the type `radio button`.
`radiobuttons`	Renders multiple HTML input elements of the type `radio button`.
`checkbox`	Renders an HTML input element of the type `checkbox`.
`checkboxes`	Renders multiple HTML input elements of the type `checkbox`.
`textarea`	Renders an HTML Textarea element.
`errors`	Displays binding and/or validation errors to the user. It can be used to either specify the path for field-specific error messages or to specify an * to display all error messages.
`label`	Renders a HTML Label and associate it with an input element.
`button`	Renders an HTML Button element.

Form Tag

- This tag renders an HTML 'form' tag and exposes a **binding path** to inner tags for binding.
- All the other tags in this library are nested tags of the form tag.

```
@RequestMapping(value = "/product", method = RequestMethod.GET)
public String inputProduct(@ModelAttribute("newProduct")
                Product newProduct) {
```

```
<form:form  modelAttribute="newProduct" action="addBook"
method="post >
```

```
@RequestMapping(value = "/product", method =
RequestMethod.POST)
public String saveProduct(@Valid @ModelAttribute("newProduct")
Product productToBeAdded, BindingResult result) {
```

Form examples with HTML output

```
<form:input id="name" path="name"/>
```

Generated HTML:

```
<input id="name" name="name" type="text" value=""/>
```

- **When categories is a LIST**

```
<form:select id="category" path="category.id"
items="${categories}" itemValue="id" itemLabel="name" />
```

- Generated HTML:

```
<select id="category" name="category.id">
    <option value="1">Computing</option>
    <option value="2">Travel</option >
    <option value="3">Health</option >
</select>
```

NOTE: *path is the "binding Path" defined on previous slide*

General Purpose Spring Tag Library

- Not Spring MVC specific
- Available to any Java Server Page used in the Spring Framework
- Tags for evaluating errors, setting themes and outputting internationalized messages.

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

Spring Tag Library

Tag Name	Description
`htmlEscape`	Sets the default HTML escape value for the current page. If set, this tag overrides the value set in the `defaultHtmlEscape` context-parameter.
`escapeBody`	Escapes the enclosed body.
`message`	Displays a message with the given code and for the selected locale. (See the section about internationalization (I18N) later in this chapter for more information.)
`theme`	Uses the variable from the currently selected theme. (See Chapter 9 for more information.)
`hasBindErrors`	Shows or hides part of the page (or element) based on whether an model attribute has bind (or validation) errors.
`nestedPath`	Selects a nested path to be used by the bind tag's path. For example, it can be used to bind `customer.address` to `street` instead of to `customer.address.street`.
`bind`	Binds an attribute of the model (attribute) to the enclosed input element.
`transform`	Transforms the bound attribute using Spring's type-conversion system. This tag must be used inside a `spring:bind` tag.
`url`	Similar to the jstl core URL tag. It is enhanced so the URL can contain URL template parameters.
`param`	Specifies a parameter and value to be assigned to the URL. This tag is used inside an `url` tag.
`eval`	Evaluates a SpEL expression and prints the result or assigns it to a variable.

Thymeleaf – an option to JSP

- Thymeleaf is very close to HTML.
- Your browser can be used to display a view without your full application running.
- Thymeleaf supports the Spring Expression Language
  - Variable expressions (${...}) execute on model attributes
  - asterisk expressions (*{...}) execute on the page controller
  - hash expressions (#{...}) are for internationalization
  - link expressions (@{...}) rewrite URLs.

Thymeleaf – an option to JSP

- Can use Thymeleaf fragments instead of Apache Tiles for creating a template to use for all your pages:
- See Alex's demo: **resources/shared/homelayout.html**
  - Shows how to define header, content, and footer with the needed CSS.
  - Then include in all your pages with:

```
<html layout:decorator="shared/homelayout“
    <th:block layout:fragment=“header">
    <th:block layout:fragment=“content">
    <th:block layout:fragment="footer">
```

Spring MVC Summary

- The basic ingredients of a Spring MVC application include

(from the server perspective):

1. web pages for the view (the known),
2. the back end domain logic and data model (knower or underlying intelligence)
3. the Spring framework and Dispatcher Servlet and managed beans as the controller to connect the view and model. (the process of knowing)

(what is the known and knower from the client perspective?)

Science of Consciousness

Knowledge is the wholeness of knower, known, and process of knowing.