Maintaining State

Greater Success with Greater Breadth of Awareness

Spring MVC Model

- **ALL** [.NET, STRUTS, JSF] component based MVCs
- Manage the model
  - Gather, convert and validate request parameters
  - Developer focuses on application/business function
  - Model contains POJO objects that reflect state of app
  - **SPRING MVC uses Model interface instead of HTTP Objects**
- Goal of Spring MVC framework
  - As view-agnostic as possible - not bound to the HTTP
- **public interface Model**
  - Defines a holder for model attributes.
  - Allows for accessing the overall model as a java.util.Map.

JavaBean vs POJO vs Spring Bean

- JavaBean
  - Adhere to Sun's JavaBeans specification
  - Implements Serializable interface
  - Must have default constructor, setters & getters
  - Reusable Java classes for visual application composition
- POJO
  - 'Fancy' way to describe ordinary Java Objects
  - Doesn't require a framework
  - Doesn't require an application server environment
  - Simpler, lightweight compared to 'heavyweight' EJBs
- Spring Bean
  - Spring managed - configured, instantiated and injected
- A Java object can be a JavaBean, a POJO and a Spring bean all at the same time.

Model Scoped Attributes

- JSP page scope
  - The page scope restricts the scope and lifetime of attributes to the same page where it was created.
- Request scope
  - only be available for that request
  - Thread Safe
- Session Scope
  - Session is defined by set of session scoped attributes
  - Lifetime is a browser session
  - **Sessions are a critical state management service provided by the web container.**
- Context scope
  - Application level state
  - Lifetime is "usually" defined by deployment of application
  - Attributes available to every controller and request in the application

Managing state information

How to handle the different scopes of model information:

- Request scope: short term computed results to pass from one servlet to another (i.e., "forward")

  request.setAtttribute(key,value)

  model.addAttribute(key,value)

- Session scope: conversational state info across a series of sequential requests from a particular user

  HttpSession session = request.getSession(); session.setAttribute(key,value);

  @SessionAttributes - model.addAttribute(key,value)

- Application/context scope: global info available to all controllers in this application

  ```
  request.getServletContext().getAttribute("appName")
  ```

  - **OR**

  ```
  @Autowired
  ServletContext servletContext;

  servletContext.getAttribute("appName")
  ```

Request Scope Attribute

```
@RequestMapping(value = "/forward")
public String forward(Product product, Model model) {
   product.setDescription("Request Attribute Exists!!");
   model.addAttribute("requestAttribute", product);
   model.addAttribute("redirectParamTest", "Request Parameter EXISTS!");
   return "forward:/get_forward";
}

@RequestMapping(value = "/get_forward")
public String getForward(Model model) {
   return "ForwardRedirect";
}
```

- ForwardRedirect.jsp

```
<h4>${redirectParamTest}</h4>
<h4>${requestAttribute.description}</h4>
```

- Demo: ProductSessionExample – Forward

@SessionAttributes

- Class level annotation that indicates an object is to be **added/retrieved** from Session.

```
@Controller
@SessionAttributes({ "Leonardo", "Splinter" })
public class SessionController {

    @RequestMapping(value = { "/getSession" }, method = RequestMethod.GET)
    public String inputProduct(Model model, HttpSession session) {
        Product product = new Product();
        product.setName("Leonardo Turtle");
        model.addAttribute("Leonardo", product);
        model.addAttribute("Splinter", "Splinter");
        // add Regular attribute
        session.setAttribute("Donatello", "Donatello Turtle");
        return "SessionForm";
    }
}
```

- Retrieve from Model

```
Product product = (Product) model.asMap().get("Leonardo");
```

- Used to mark a session attribute as not needed *after* the request has been processed by the controller

```
status.setComplete();
```


See demo: ProductSessionExample - Session

Application level Attributes

- ServletContext contains Application level state information

- XML configuration:

```
<bean
   class="org.springframework.web.context.support.ServletContextAttributeExporter
   ">
    <property name="attributes">
        <map>
            <entry key="appName" value="State Management Demo" />
        </map>
    </property>
</bean>
```

- Programmatic access:

```
@Autowired
ServletContext servletContext;

servletContext.getAttribute("appName");
```

Main Point

- State information can be stored in request, session, or context/application scope and also as hidden fields or cookies.
- *Deeper levels of consciousness are more powerful and have broader scope.*

Static Resources

- Want to handle static content, e.g., image file, js, css, etc.
- Need to identify them to the DispatcherServlet - since no Controller exists for serving static resources.
- Using Spring:
  - Declare resources folder[s]
  - Serve static content from there
  - Use mvc:resources – A Spring help element to map "url path" to a physical file path location.
- All references to `/resource/` will be mapped to the context root (webapp): `/css/` folder.

```
<mvc:resources mapping="/resource/**" location="/css/"/>
```

- Alternative: serves content from servlet containers
  - If we are using DefaultServletHttpRequestHandler, then we can replace :

```
<mvc:resources mapping="/js/**" location="/js/"/>
<mvc:resources mapping="/css/**" location="/css/"/>
<mvc:resources mapping="/images/**" location="/images/"/>
```

  - with :

```
<mvc:default-servlet-handler />
```

- path pattern - Apache ant
  - ('*') matches zero or more characters, up to the occurrence of a '/'.
  - ('**') matches zero or more characters. This could include the path separator '/'.

Static Resources

```
@Configuration
@EnableWebMvc
@ComponentScan("edu.mum.cs")
public class WebApplicationContextConfig implements WebMvcConfigurer {

   /*
    * Ensures that dispatcher servlet can be mapped to '/' and that static resources
    * are still served by the containers default servlet.
    */
   @Override
   public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
      configurer.enable();
   }


   @Override
   public void addResourceHandlers(final ResourceHandlerRegistry registry) {
      registry.addResourceHandler("/myresources/**").addResourceLocations("/resources/");
   }

}
```

Request GET versus POST

Difference between GET and POST:

- GET request has no message body, so parameters are limited to what can fit into Query String.

  GET /advisor/selectBreadTaste.do?color=dark&taste=salty

- GET requests are *idempotent*
- GET is to retrieve data

Idempotent means that multiple calls with the same operation doesn't change the server

- POST is to send data to be processed and stored
- POST has a body
- POST "more secure" since parameters not visible in browser bar

Post/Redirect/Get (PRG) Pattern

- POST-REDIRECT-GET, or the PRG pattern for short. The rules of the pattern are as follows:
- Never show pages in response to POST
- Always load pages using GET
- Navigate from POST to GET using REDIRECT

- Forward – if operation can be safely repeated upon a browser reload of the resulting web page [Use with GET].

- Redirect - If operation performs an edit on the datastore, to avoid the possibility of inadvertently duplicating an edit to the database[Use with POST].

Spring MVC Forward & Redirect

- Work Just like JSP Forward & Redirect
- SYNTAX:

```
return "forward:/demo";
return "redirect:/demo";
```

- WHERE:

```
@RequestMapping(value="/demo" )
public String getDemo (Model model) {}
```

- EXTERNAL REDIRECT:

```
return "redirect:http://www.mum.edu";
```

See demo: ProductSessionExample - Redirect

Flash Attributes

- Efficient solution for the *Post/Redirect/Get* pattern.
- Attributes are saved [in Session] temporarily before the redirect
- Attributes are added to the Model of the target controller and are deleted [from Session] immediately.

```
@RequestMapping(value = "/product", method = RequestMethod.POST)
public String saveProduct(Product newProduct, Model model,
   RedirectAttributes redirectAttributes,
    HttpServletRequest request) {

   redirectAttributes.addFlashAttribute(newProduct);
   // Returned as a parameter on GET URL
   redirectAttributes.addAttribute("name", "Kemosabe");
   return "redirect:/details";
}
```

- String & primitive types are added to URL [e.g., GET]

```
redirectAttributes.addAttribute(newProduct.name);
```

Controller Method Arguments

Controller method argument	Description
javax.servlet.http.HttpSession	Enforces the presence of a session.
@PathVariable	For access to URI template variables. See URI patterns.
@RequestParam	For access to the Servlet request parameters, including multipart files. Parameter values are converted to the declared method argument type. See @RequestParamas well as Multipart.
@RequestHeader	For access to request headers. Header values are converted to the declared method argument type. See @RequestHeader.
@CookieValue	For access to cookies. Cookies values are converted to the declared method argument type. See @CookieValue.
@RequestBody	For access to the HTTP request body. Body content is converted to the declared method argument type by using HttpMessageConverter implementations. See @RequestBody.
java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap	For access to the model that is used in HTML controllers and exposed to templates as part of view rendering.
RedirectAttributes	Specify attributes to use in case of a redirect (that is, to be appended to the query string) and flash attributes to be stored temporarily until the request after redirect. See Redirect Attributes and Flash Attributes.
@ModelAttribute	For access to an existing attribute in the model (instantiated if not present) with data binding and validation applied. See @ModelAttribute as well as Model and DataBinder. Note that use of @ModelAttribute is optional (for example, to set its attributes). See "Any other argument" at the end of this table.
BindingResult	For access to errors from validation and data binding for a command object (that is, a @ModelAttribute argument) or errors from the validation of a @RequestBodyor @RequestPart arguments. You must declare an Errors, or BindingResultargument immediately after the validated method argument.
SessionStatus + class-level @SessionAttributes	For marking form processing complete, which triggers cleanup of session attributes declared through a class-level @SessionAttributes annotation. See@SessionAttributes for more details.
@SessionAttribute	For access to any session attribute, in contrast to model attributes stored in the session as a result of a class-level @SessionAttributes declaration. See@SessionAttribute for more details.

Controller Method Return Types

Controller method return value	Description
@ResponseBody	The return value is converted through HttpMessageConverterimplementations and written to the response. See @ResponseBody.
HttpEntity<B>, ResponseEntity <B>	The return value that specifies the full response (including HTTP headers and body) is to be converted through HttpMessageConverterimplementations and written to the response. See ResponseEntity.
HttpHeaders	For returning a response with headers and no body.
String	A view name to be resolved with ViewResolver implementations and used together with the implicit model — determined through command objects and @ModelAttribute methods. The handler method can also programmatically enrich the model by declaring a Model argument (see Explicit Registrations).
View	A View instance to use for rendering together with the implicit model — determined through command objects and @ModelAttribute methods. The handler method can also programmatically enrich the model by declaring a Model argument (see Explicit Registrations).
java.util.Map, org.springframework.ui.Model	Attributes to be added to the implicit model, with the view name implicitly determined through a RequestToViewNameTranslator.
ModelAndView object	The view and model attributes to use and, optionally, a response status.
void	A method with a void return type (or null return value) is considered to have fully handled the response if it also has a ServletResponse, an OutputStream argument, or an @ResponseStatus annotation. The same is also true if the controller has made a positive ETag or lastModifiedtimestamp check (see Controllers for details). If none of the above is true, a void return type can also indicate “no response body” for REST controllers or a default view name selection for HTML controllers.
Any other return value	Any return value that does not match any of the earlier values in this table and that is a String or void is treated as a view name (default view name selection through RequestToViewNameTranslator applies), provided it is not a simple type, as determined by BeanUtils#isSimpleProperty. Values that are simple types remain unresolved.

More Model, ModelMap, ModelAndView

- Model is an interface while ModelMap is a class.
- Model has method asMap to get actual map.
- ModelMap is a class that is a custom[convenience] Map implementation that automatically generates a key for an object when an object is added to it.
- ModelAndView is just a container for both a ModelMap and a view object. It allows a controller to return both as a single value.

Main Point

- Understanding the function and capability of the POST, Redirect and GET, leads to a combination that overcomes an inherent weakness in web applications.
- *The development of consciousness, increases awareness and eliminates the restrictions that cause inherent weakness.*