Spring MVC Controllers Part II

"Code with Passion!"

Topics

- URI template
- Mapping requests with other means (in addition to URL)
- Handler method arguments - @PathVariable, @RequestParam
- Type conversion
- Handler method that directly creates a HTTP response
- Interceptor
- Automatic attr. key name generation

URI Template

What is a URI Template?

- URI Template is a URI that contains one or more variables
  - Variables are in the form of *{nameOfVariable}*

@RequestMapping(value="/owners/{ownerId}")) // Example

  - The *nameOfVariable* needs to be passed to a handler method as an argument with *@PathVariable* annotation if it needs to be accessed within the handler method (we will see examples in the next three slides)
  - Automatic type conversion occurs to the argument type
- When you substitute values for these variables, the URI template becomes a concrete URI.

/owners/3
/owners/5

URI Template Example #1

```
// Suppose the request URL is http://locahost:8080/myapp/owners/3,
// the value 3 will be captured as “ownerId” argument in String type.
@RequestMapping(value="/owners/{ownerId}",
                        method=RequestMethod.GET)

// The ownerId needs to be passed to a handler method as an
// argument with @PathVariable annotation
public String findOwner(@PathVariable String ownerId, Model model) {

    // You can now use ownerId in your business logic
    Owner owner = ownerService.findOwner(ownerId);
    model.addAttribute("owner", owner);
    return "displayOwner";
}
```

URI Template Example #2

```
// You can use multiple @PathVariable annotations to bind to multiple URI
// Template variables.

// Suppose the request URL is http://locahost:8080/myapp/owners/3/pets/5,
// the value 3 will be captured as “ownerId” argument in String type while
// the value 5 will be captured as “petId” argument in String type.
@RequestMapping(value="/owners/{ownerId}/pets/{petId}",
                            method=RequestMethod.GET)
public String findPet(@PathVariable String ownerId,
                            @PathVariable String petId,
                            Model model) {
    Owner owner = ownerService.findOwner(ownderId);
    Pet pet = owner.getPet(petId);
    model.addAttribute("pet", pet);
    return "displayPet";
}
```

URI Template Example #3

```
// You can use multiple @PathVariable annotations to bind to multiple URI
//  Template variables
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

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

Lab:

Exercise 1: URI Template

4945_spring4_mvc_controllers_part2.zip

Mapping Requests with Other Means (in addition to the URL)

URL Mappings through parameter conditions

```
// You can narrow URL mappings through parameter conditions: a sequence of
// "myParam=myValue" style expressions, mapping occurs only when each
// such parameter is found to have the given value.
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

  // Handles http://locahost:8080/myapp/owners/3/pets/5?myParam=myValue
  @RequestMapping(value = "/pets/{petId}", params="myParam=myValue")
  public void findPet(@PathVariable String ownerId, @PathVariable String petId,
                      Model model) {
    // implementation omitted
  }
}
```

Mappings through HTTP header conditions

```
// The addPet() method is only invoked when the “content-type” HTTP header
// matches the text/* pattern, for example, text/xml
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

@RequestMapping(value = "/pets", method = RequestMethod.POST,
                        headers="content-type=text/*")
  public void addPet(Pet pet, @PathVariable String ownerId) {
    // implementation omitted
  }
}
```

Handler Method Arguments

Objects that are auto-created by Spring

- You can simply use these as arguments in any of your handler method because they are auto-created by Spring MVC framework
- *ServletRequest* or *HttpServletRequest*
  - Request or response objects (Servlet API)
- *HttpSession*
  - Session object (Servlet API)
- *java.util.Locale*
  - For the current request locale, determined by the most specific locale resolver available
- *java.security.Principal*
  - Currently authenticated user

@PathVariable & @RequestParam

- *@PathVariable*
  - Extracts data from the request URI
    - http://host/catalog/items/123
  - Parameter values are converted to the declared method argument type
- *@RequestParam("namex")*
  - Extracts data from the request URI query parameters
    - http://host/catalog/items/?namex=abc
  - Parameter values are converted to the declared method argument type

@PathVariable - For URI Path Values

```
// Use the @PathVariable annotation to bind URI path value to a method
// parameter in your controller.
@Controller
@RequestMapping("/pets")
public class MyPetClass {

    // ...

    // Will handle ../pets/4 or ../pets/10.
    // “4” and “10” are converted to int type by Spring.
    @RequestMapping(value=”/{petId}”,method = RequestMethod.GET)
    public String getData(@PathVariable int petId, ModelMap model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ...
```

@RequestParam - For Query Parameters

```
// Use the @RequestParam annotation to bind query request parameters to a
// method parameter in your controller.
@Controller
@RequestMapping("/pets")
public class MyPetClass {

    // ...

    // Will handle ../pets?petId=4 or ../pets?petId=10.
    // “4” and “10” are converted to int type by Spring.
    @RequestMapping(method = RequestMethod.GET)
    public String getData(@RequestParam("petId") int petId, ModelMap model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ...
```

Request Header and Body

- *@RequestHeader("name")*
  - Annotated parameters for access to specific Servlet request HTTP headers

```
@RequestMapping(value="requestHeader1")
public @ResponseBody String withHeader1(@RequestHeader("Accept") String Accept) {
      return "Obtained 'Accept' header '" + Accept + "'";
}
```

- *@RequestBody*
  - Annotated parameters for access to the HTTP request body.

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
public void handle(@RequestBody String body, Writer writer)
            throws IOException {
     writer.write(body);
}
```

Lab:

Exercise 2: Handler method arguments

4945_spring4_mvc_controllers_part2.zip

Type Conversion

Type Conversion

- Type conversion happens automatically
- Built-in converters (implementations of Converter interface) used in the places where type conversion is required
  - *@RequestParam, @PathVariable, @RequestHeader*, etc
- *HttpMessageConverter* used for
  - *@RequestBody, @ResponseBody, HttpEntity, ResponseEntity*
- Can declare annotation-based conversion rules
  - @NumberFormat, @DateTimeFormat
- You can plug-in a custom converters (we will cover custom type conversion in “spring4_mvc_form”)

Handler Method Directly Creates a HTTP Response (No view selection)

Handler creates HTTP response

- Because the handler directly creates HTTP response, there occurs no view selection
  - Option #1: @ResponseBody
  - Option #2: HttpEntity<?> or ResponseEntity<?>

#1: @ResponseBody annotated Method

- If the method is annotated with *@ResponseBody*, the return type, String in the example below, is written to the response HTTP body – there is no view selection required

```
@RequestMapping(value="/response/annotation", method=RequestMethod.GET)
public @ResponseBody String responseBody() {
        return "The String ResponseBody";
}
```

#2: HttpEntity<?> or ResponseEntity<?>

- Provide access to the Servlet reponse HTTP headers and contents.
- The entity body will be converted to the response stream using HttpMessageConverter

```
@RequestMapping(value="/response/entity/headers", method=RequestMethod.GET)
public ResponseEntity<String> responseEntityCustomHeaders() {
      HttpHeaders headers = new HttpHeaders();
      headers.setContentType(MediaType.TEXT_PLAIN);
      return new ResponseEntity<String>("The String ResponseBody with custom header
    Content-Type=text/plain",
                  headers, HttpStatus.OK);
}
```

Interceptor

What Do Interceptors Do?

- Interceptors are useful when you want to apply specific functionality to certain requests
  - Authentication, authorization, logging, etc
- Interceptors must implement *HandlerInterceptor* interface or extend *HandlerInterceptorAdapter* class
  - *boolean preHandle(..)* method - the interceptor execution chain will continue only when this method returns 'true'. Else, DispatcherServlet assumes that this interceptor has already dealt with the response itself
  - *void postHandle(..)* method
- Things you need to do
  - #1: Write interceptor code
  - #2: Configure interceptor

#1: Write Interceptor Code

```
// This is logger interceptor
public class LoggingInterceptor extends HandlerInterceptorAdapter {

    private static final Logger logger =
       LoggerFactory.getLogger(LoggingInterceptor.class);

    public boolean preHandle(HttpServletRequest request,
                             HttpServletResponse response,
                             Object handler) throws Exception {
            logger.info("LoggingInterceptor: preHandle() entered");
        return true;
    }

    public void postHandle(HttpServletRequest request,
                           HttpServletResponse response,
                           Object handler,
                           ModelAndView modelAndView) throws Exception {
            logger.info("LoggingInterceptor: postHandle() exiting");
    }
}
```

#2: Configure Interceptor

```
@Bean
WebMvcConfigurerAdapter mvcConfigurer() {

    return new WebMvcConfigurerAdapter() {

        // Configure interceptors
        @Override
        public void addInterceptors(InterceptorRegistry registry) {

            registry.addInterceptor(new LoggingInterceptor());
            registry.addInterceptor(new ElapsedTimeInterceptor());

            // You can also add path patterns through which interceptors get applied
            // registry.addInterceptor(new
            // ElapsedTimeInterceptor()).addPathPatterns("/path1/path2/*");

        }
    };
```

Lab:

Exercise 3: Interceptor

4945_spring4_mvc_controllers_part2.zip

Automatic Attr. Key name Generation (in ModelMap or ModeAndView)

Key Name Generation Strategy

- Scalar object - use the short class name of the object's class
  - *x.y.User* instance added will have the key "*user*"
  - *x.y.Registration* instance added will have the key "*registration*"
- Collection object
  - An x.y.User[] array with one or more x.y.User elements added will have the key "*userList*"
  - An x.y.Foo[] array with one or more x.y.Foo elements added will have the key "*fooList*"
  - A java.util.ArrayList<User> with one or more x.y.User elements added will have the key "*userList*"

Automatic Key Generation for ModelMap

```
@Controller
public class DisplayShoppingCartController{

   @GetMapping
   public String handleRequest(ModelMap modelMap) {
       // Note that catItems is List of Item type
             List<Item> cartItems = new ArrayList<Item>();
       cartItems.add(new Item("Apple", 10.0));
       cartItems.add(new Item("Orange", 20.0));

       User user = new User("Sang Shin");

      // This is the same as modelMap.addAttribute("itemList", cartItems);
      modelMap.addAttribute(cartItems); // "itemList" is automatically generated as a key

      // This is the same as modelMap.addAttribute("user", user);
      modelMap.addAddtribute(user); //"user" is automatically generated as a key

      return "shoppingCart";
   }
}
```

Automatic Key Generation for ModelAndView

```
@Controller
public class DisplayShoppingCartController {

   @GetMapping
   public ModelAndView handleRequest() {

       // Note that catItems is List of Item type
             List<Item> cartItems = new ArrayList<Item>();
       cartItems.add(new Item("Apple", 10.0));
       cartItems.add(new Item("Orange", 20.0));

       User user = new User("Sang Shin");

      // “displayShoppingCart” is logical view
      ModelAndView mav = new ModelAndView("displayShoppingCart");

      // This is the same as mav.addObject(“itemList”, cartItems);
      mav.addObject(cartItems); // “itemList” is automatically generated as a key

      // This is the same as mav.addObject(“user”, user);
      mav.addObject(user); //”user” is automatically generated as a key

      return mav;
   }
}
```

Lab:

Exercise 4: Automatic Attr. Key Generation
4945_spring4_mvc_controllers_part2.zip

Code with Passion!