Spring MVC

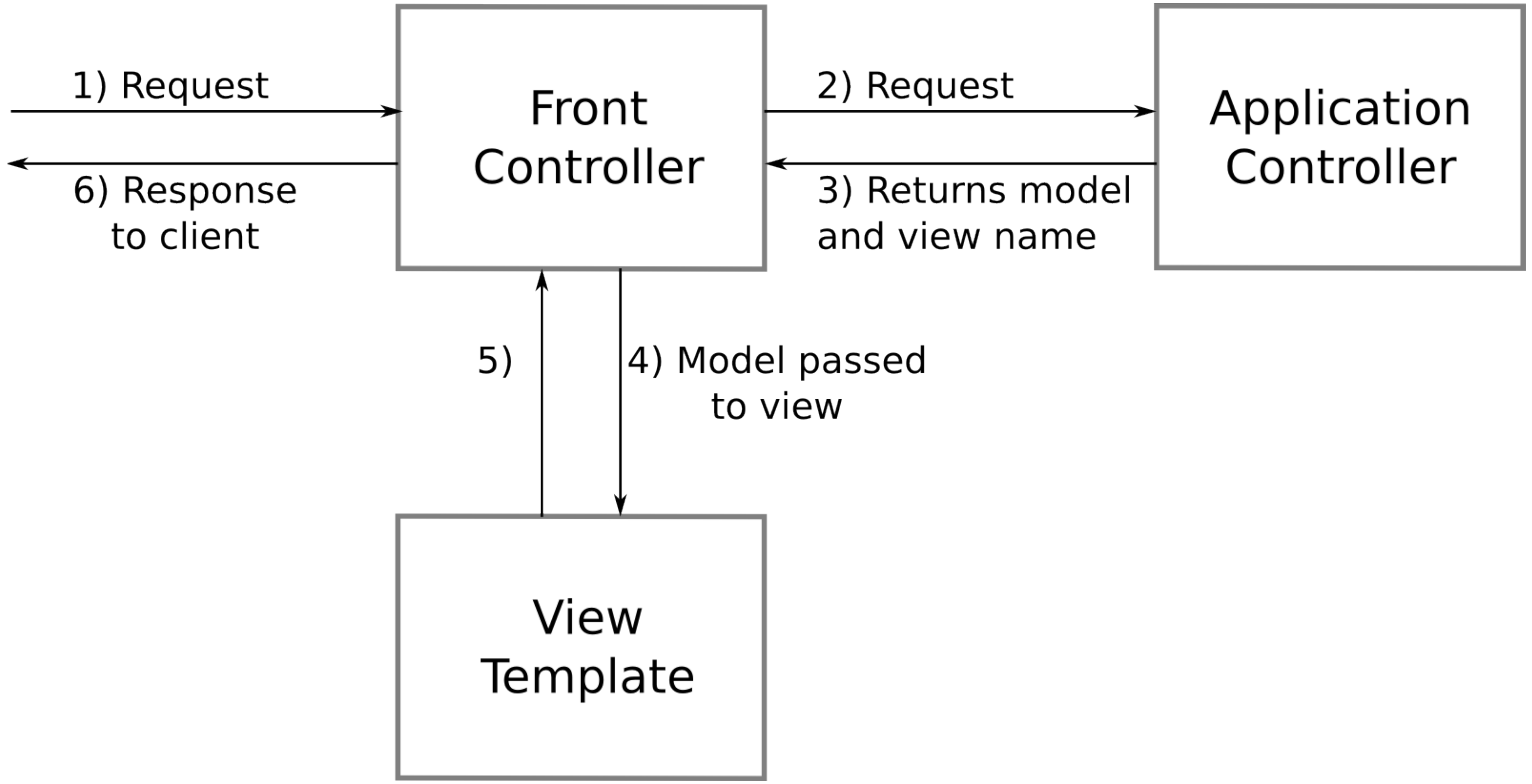
Richard Paul
Kiwiplan NZ Ltd
27 Mar 2009

What is Spring MVC

Spring MVC is the web component of Spring's framework.

Model - The data required for the request.
View - Displays the page using the model.
Controller - Handles the request, generates the model.

Front Controller Pattern



Controller Interface

```
public class MyController implements Controller {
  public ModelAndView handleRequest(
      HttpServletRequest request,
      HttpServletResponse response) {

    // Controller logic goes here
  }
}
```

This is the low level interface for controllers, most of the time you will not use the `Controller` interface directly.

The `ModelAndView`, is an object that holds the model objects as well as the view required to be rendered (simplified definition).

Controller Annotations

```
@Controller
public class ItemController {

  private ItemService itemService;
  @Autowired
  public ItemController(ItemService itemService) {
    this.itemService = itemService;
  }

  @RequestMapping(value="viewItem.htm", method=RequestMethod.GET)
  public Item viewItem(@RequestParam Long id) {
    return itemService.get(id);
  }

}
```

By convention a view with the name 'viewItem' (based on the request mapping URL) will be used to render the item.

Session attributes

```
@Controller
@RequestMapping("editItem.htm")
@SessionAttribute("item")
public class ItemEditorController {

  @RequestMapping(method=RequestMethod.GET)
  public String setupForm(@RequestParam Long itemId, ModelMap model) {
    Item item = ...; // Fetch item to edit
    model.addAttribute("item", item);
    return "itemForm";
  }

  @RequestMapping(method=RequestMethod.POST)
  public String processSubmit(@ModelAttribute("item") Item item) {
    // Store item
    // ...
    return "redirect:/viewItem.htm?item=" + item.getId();
  }
}
```

A GET to 'editItem.htm' adds the item with the given ID to the user's session. When a POST is made, this item is pulled from the session and provided as an argument to the processSubmit method.

Flexible Method Arguments

Methods mapped with @RequestMapping can have very flexible method signatures.

Arguments

```
handle(@RequestParam Long id) // 'id' parameter from request
handle(@ModelAttribute Item item, BindingResult result)
handle(ModelMap modelMap) // The model to populate
handle(HttpSession session) // The user's session
handle(Locale locale) // The locale (from the locale resolver)
```

Return Type

```
String // View name
ModelAndView // Model objects and view name
Item // Or other object, assumed to be the model object
void // This method handles writing to the response itself
```

http://static.springframework.org/spring/docs/2.5.x/reference/mvc.html#mvc-ann-requestmapping-arguments

Testability

We no longer need to pass in a HttpServletRequest when unit testing controllers. Most use cases can simply pass objects into the method. Examples show with Mockito.

```
public Item view(@RequestParam Long id) {
  return itemService.get(id);
}

@Test
public void viewWithValidId() {
  Item item = new Item(3l);
  when(itemService.get(3l)).thenReturn(item);
  assertEquals(item, controller.view(3l));
}
```

Testability - with model attributes

```
public String submitItem(@ModelAttribute Item item) {
  itemService.save(item);
}

@Test
public void submitItem() {
  Item item = new Item(3l);
  controller.submitItem(item);
  verify(itemService.save(item));
}
```

Previously the item would have been manually placed into a mock session under the 'item' key before calling the test.

Binding

In order to bind request parameters to Java objects, Spring uses PropertyEditors.

Default Property Editors

```
ByteArrayPropertyEditor   // For Strings
CustomNumberEditor        // For numbers, e.g. 4, 3.2
...
```

Spring Provided Editors

```
StringTrimmerEditor       // For whitespace trimmed Strings
FileEditor                // For file uploads
...
```

Custom property editors can be written.
e.g. `CategoryPropertyEditor` for binding a `Category`.

Example - CategoryPropertyEditor

```
public class CategoryPropertyEditor extends PropertyEditorSupport {
  @Override
  public String getAsText() {
    Object value = getValue();
    if (value == null) {
      return "";
    }
    return ((Category) value).getId().toString();
  }
  @Override
  public void setAsText(String text) {
    this.setValue(null); // Always clear existing value
    if (!StringUtils.isBlank(text)) {
      long id = Long.parseLong(text);
      this.setValue(categoryService.get(id));
    }
  }
}
```

If an IllegalArgumentException is thown a binding error is stored and can be shown against the field in the form.

Registering Property Editors

Those property editors that aren't automatically wired up need to be initialised against the `DataBinder`.

```
@InitBinder
public void initBinder(DataBinder binder) {
  // Register a new editor each time as they are not thread safe
  binder.registerCustomEditor(
    Category.class, new CategoryPropertyEditor());
}
```

Validation

Spring provides a validation framework that is usually invoked in the controller layer.

Also possible to register different editors based on the field.

An alternative, which is still in draft form is JSR-303. It provides validation at a model level and can be used in the presentation, business & persistence layers.

Example - ItemValidator

Spring validator example.

```
public class ItemValidator implements Validator {

  public boolean supports(Class clazz) {
    return Item.class.isAssignableFrom(clazz);
  }

  public void validate(Object target, Errors errors) {
    Item item = (Item) target;
    ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
    if (item.getCategory() == null) {
      e.rejectValue("category", "item.category.required");
    }
  }
}
```

Errors are stored in the `Errors` object, if any errors are present, the form can be redisplayed with error messages instead of following the normal form submission.

Views

Spring supports a number of view technologies, the below example shows JSP with JSTL, specifically Spring's Form tags.

```
<form:form method="post" commandName="item">
  <form:label path="name">Name</form:label>
  <form:input path="name"/>
  <!-- Binding & Validation errors displayed here -->
  <form:error path="name" cssClass="error"/>

  <!-- Show a select list of categories. Categories already set
       on the item are automatically selected -->
  <form:select path="categories" items="${allCategories}"
    itemValue="id" itemLabel="name"/>

</form:form>
```

Form tags are include for all HTML input types, e.g. form:radio, form:textarea

Resolving Messages

Spring resolves all messages using `MessageSource` interface.

This message source is uses dot notation to access message coes. e.g. item.name.required

Using springs default MessageSource, properties files using a specific locale are used.

```
ApplicationResources.properties       // Fall back
ApplicationResources_en.properties    // English
ApplicationResources_es.properties    // Spanish
```

Custom implementations of MessageSource can be written. Useful if messages are pulled from a different backend e.g. XML

AJAX Support

AJAX support is limited in Spring 2.5

Generally use a separate library to generate JSON data for the controller to return.

Hopefully better support will be coming with Spring 3.0 with new view types for XML, JSON, etc.

Questions?

Spring Reference
http://static.springframework.org/spring/docs/2.5.x/reference/