



Enterprise Architecture

SPR03_1: Web Applications with Spring – Best Practices

Practices targeting applications using the Spring
MVC framework.

Version 1.0.0

Summary of revisions:

Version	Edited by	Date	Description of changes
1.0.0	Rao Pathangi	03/21/2015	Included feedback from Gregory Trotti and Mohammed Ehfaj Khan
0.6.0	Rao Pathangi	1/10/2015	First draft

Contents

Introduction: modernizing web applications.....	3
1. Web application architecture	4
1.1 Leverage Spring's "View Resolver" to drive "View" agnostic applications	4
1.2 Collocate web and mobile applications if non-functional requirements are comparable	5
1.3 Develop Spring MVC Controllers as POJOs.....	5
1.4 WebSockets aren't recommended in a Discover environment as yet	6
1.5 Offload Session persistence and management to an enterprise solution.....	6
1.6 Use @ModelAttribute and @SessionAttribute to augment conversation state.....	6
1.8 Use Spring Web Flow to implement web based conversations or screen-flows	8
2 Web application organization	8
2.1 Use <mvc:resources> to segregate static content	8
2.2 Spread Spring Configuration across multiple XML files for maintainability	9
2.3 Prefer Spring's annotation driven request mapping in lieu of <*>HandlerMapping schemes	9
3 Request processing	10
3.1 Enforce fine grained control over request processing with @RequestMapping.....	10
3.2 Leverage @Valid to validate input prior to further request processing.....	11
3.3 Validate portions of Value Objects using validation groups (JSR 303).....	13
4 Exception Handling	14
4.1 Add @ResponseStatus to application specific, developer created exceptions	14
4.2 Utilize @ControllerAdvice to implement global exception handling logic.....	14

Introduction: modernizing web applications

Web applications at Discover, customer facing or otherwise, are being influenced by three key forces: mobile devices, rise of REST API, and increasing need for application security. The first two forces in particular are changing how we approach web application development in general and are demanding greater separation between the logic that renders the **View** and the **Models** that populate **Views**. Innovation in JavaScript frameworks and device specific frameworks have lent sophistication to rendering a “**View**” that is rich, responsive, asynchronous, and seamless across big-browsers and mobile devices.

In addition to the developments mentioned above, the practice of software development in general is reacting to the advent of Cloud as a deployment platform. Discover applications must modernize to benefit from these innovations and simultaneously stay nimble to meet business requirements.

Architecturally, the following are key areas where Discover web applications can improve:

- Increase loose coupling between **Models** and **Views**.
- Adopt flexible architectures where web and mobile “**Views**” can evolve in parallel without sacrificing performance.
- Make it easy for web applications to adopt new innovations in frameworks or Java itself.
- Implement application security in a non-invasive manner without the application code having intimate knowledge of the implementation.
- Leverage Servlet 3.1 and/or framework enhancements to transmit state changes to “**Views**”.

This document will illustrate some of the best practices associated with Web applications developed in Java using Spring MVC with a renewed focus on application architecture, security, and portability across platforms – Cloud or on-premise.

1. Web application architecture

1.1 Leverage Spring's "View Resolver" to drive "View" agnostic applications

The following diagram depicts Spring's MVC implementation. The Dispatcher Servlet Receives a logical name of the view from the "Resource Controller." The Dispatcher Servlet enquires the "View Resolver" to determine the view and how the model should be rendered. Views are no longer limited to JSP, they can be XML, JSON, PDF, text etc. This is how Model and View separation is achieved in Spring. Spring has a rich set of View Resolver components that enable "View" agnostic development. The following "View Resolvers" are available out of the box.

- BeanNameViewResolver
- ContentNegotiatingViewResolver
- FreeMarkerViewResolver
- InternalResourceViewResolver
- JasperReportsViewResolver
- ResourceBundleViewResolver
- TilesViewResolver
- UrlBasedViewResolver
- VelocityLayoutViewResolver
- VelocityViewResolver
- XmlViewResolver
- XsltViewResolver

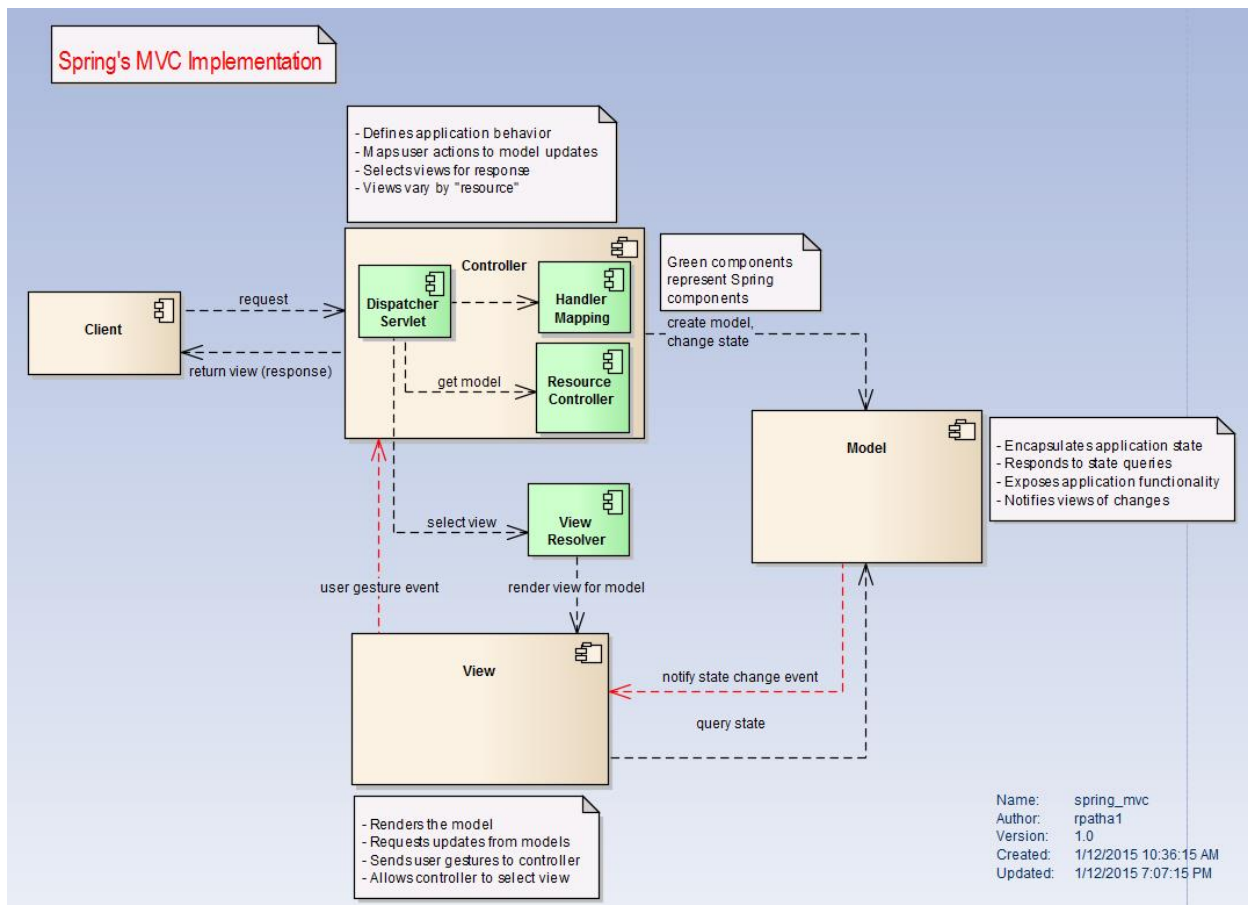


Figure 1: Spring's MVC implementation (green components)

1.2 Collocate web and mobile applications if non-functional requirements are comparable

If the mobile application GUI can be segregated either via native applications or separate mobile GUI deployments, and if the non-functional requirements across mobile and web application are comparable, then it is worthwhile to collocate both mobile and web applications into a single Spring Web application with different **View Resolvers** serving web and mobile content (see figure #1 above).

1.3 Develop Spring MVC Controllers as POJOs

A true test of a good Controller is whether one can test the MVC Controller without a web environment. A well written Controller will have minimal dependency on Spring types in its implementation when the controller is stripped off its annotations. The Controller should be JUnit testable with a mock framework to provide stand-ins for services backed by databases. The following is an example of a Controller which when stripped off its annotations results in a POJO that can be tested with JUnit.

```
package com.discover.ea.spring.docs.springmvc;

import javax.validation.Valid;

@Controller
@RequestMapping(value = "/sandbox")
public class AccountCreationController {

    @RequestMapping(method = RequestMethod.GET, params="new")
    public String processRequest(@Valid AccountHolder account)
    {
        return "temp";
    }

    // add the returned object to the model before every request processed by this controller
    @ModelAttribute("accthldr")
    public AccountHolder getAccountHolder()
    {
        return new AccountHolder();
    }

    // add the returned object to the model before every request processed by this controller
    @ModelAttribute("accthldr")
    public AccountHolder applyRules()
    {
        // apply rules, take decisions with the data in accthldr and modify accthldr

        // ideally you would return a modified AccountHolder here.
        return new AccountHolder();
    }

    // The @ModelAttribute below ensures that the same "accthldr" bean is used in processing
    @RequestMapping(method = RequestMethod.POST)
    public String processSubmission(@ModelAttribute("accthldr") AccountHolder holder)
    {
        // do something with the "accthldr" bean from the request
        return "tempSquared";
    }
}
```

Figure 2: code snippet depicting a Spring Controller

1.4 WebSockets aren't recommended in a Discover environment as yet

Note: Spring's *WebSocket implementation* although promising is yet to be tested from a security standpoint in Discover's environment. Please refrain from using WebSocket implementations until ARB approves its usage for both internal and customer-facing applications.

1.5 Offload Session persistence and management to an enterprise solution

In enterprise grade applications, session persistence is best dealt at an infrastructure level; developers should not build session persistence solutions into their application code. Application servers such as IBM WebSphere and Pivotal tc Server¹ come with modules that [persist HTTP Sessions](#) to a backend datastore. Please discuss session management and persistence with your middleware engineer.

1.6 Use @ModelAttribute and @SessionAttribute to augment conversation state

```
package com.discover.ea.spring.docs.springmvc;

import javax.validation.Valid;

@Controller
@RequestMapping(value = "/sandbox")
public class AccountCreationController {

    @RequestMapping(method = RequestMethod.GET, params="new")
    public String processRequest(@Valid AccountHolder account)
    {
        return "temp";
    }

    // add the returned object to the model before every request processed by this controller
    @ModelAttribute("accthdr")
    public AccountHolder getAccountHolder()
    {
        return new AccountHolder();
    }

    // add the returned object to the model before every request processed by this controller
    @ModelAttribute("accthdr")
    public AccountHolder applyRules()
    {
        // apply rules, take decisions with the data in accthdr and modify accthdr
        // ideally you would return a modified AccountHolder here.
        return new AccountHolder();
    }

    // The @ModelAttribute below ensures that the same "accthdr" bean is used in processing
    @RequestMapping(method = RequestMethod.POST)
    public String processSubmission(@ModelAttribute("accthdr") AccountHolder holder)
    {
        // do something with the "accthdr" bean from the request
        return "tempSquared";
    }
}
```

Figure #3: code snippet depicting the use of @ModelAttribute

¹ [HTTP Session replication and management in tc Server](#)

The inline comments in the code snippet are self-explanatory. The key takeaway is that methods marked with the `@ModelAttribute` annotation will be called to “augment” the “model” before the Controller calls upon the `@RequestMapping` method for every HTTP request.

The following code snippet depicts the use of `@SessionAttribute` to store attributes identified by “creditScore,” “balanceTrnsfrAmt,” and “apr” into session. These will be stored in the session and can be referred to as seen in `processSubmission()` method or assigned to in the `getCreditScore()` method below.

```
package com.discover.ea.spring.docs.springmvc;

import javax.validation.Valid;

@Controller
@SessionAttributes({"creditScore", "balanceTrnsfrAmt", "apr"})
@RequestMapping(value = "/sandbox")
public class AccountCreationController {

    @RequestMapping(method = RequestMethod.GET, params="new")
    public String processRequest(@Valid AccountHolder account)
    {
        return "temp";
    }

    // add the returned object to the model before every request processed by this controller
    @ModelAttribute("acctHldr")
    public AccountHolder getAccountHolder()
    {
        return new AccountHolder();
    }

    // add the returned object to the model before every request processed by this controller
    @ModelAttribute("acctHldr")
    public AccountHolder applyRules()
    {
        // apply rules, take decisions with the data in acctHldr and modify acctHldr
        // ideally you would return a modified AccountHolder here.
        return new AccountHolder();
    }

    // The @ModelAttribute below ensures that the same "acctHldr" bean is used in processing
    @RequestMapping(method = RequestMethod.POST)
    public String processSubmission(@ModelAttribute("acctHldr") AccountHolder holder, @ModelAttribute("creditScore") String score )
    {
        // do something with the "acctHldr" bean from the request
        return "tempSquared";
    }

    @ModelAttribute("creditScore")
    public String getCreditScore()
    {
        // get credit score and return it.
        return "score";
    }
}
```

Figure #4: snippet depicting the use of `@SessionAttribute`

1.7 Avoid database calls in methods annotated with `@ModelAttribute`

Use the `@ModelAttribute` annotation with caution. Methods annotated with `@ModelAttribute` will be called before every `@RequestMapping` annotated handler method irrespective of whether the handler uses the data gathered in the method. Therefore, avoid database calls or heavy processing in methods annotated with `@ModelAttribute`.

1.8 Use Spring Web Flow to implement web based conversations or screen-flows

If the application demands a “wizard” style of interface where session information has to be maintained for a “Submit” at the end of 15 (arbitrary) screens, and where the interface has to be dynamic where the view depends on the flow of answers from the user, Spring WebFlow is recommended for implementing such conversation themed “*views*.”

2 Web application organization

2.1 Use `<mvc:resources>` to segregate static content

Typically, it is a best practice to separate static and dynamic content in scenarios where the static and dynamic content evolve at different rates. If static and dynamic content have to be deployed together, then, the `<mvc:resources>` element can be used to specify where the built-in handler should look for static content.

The following servlet mapping suggests that Spring’s `DispatcherServlet` be responsible for all content including static content.

```
<!-- Processes application requests -->
<servlet>
  <servlet-name>appServlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>appServlet</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

Figure #5: snippet of web.xml

To ensure that the `DispatcherServlet` is not burdened by the task of serving up static content, developers can configure the `<mvc:resources>` element in the “mvc” name space of the Spring Application Context XML file to delegate the task of serving up static content to a built-in handler.

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

In the above line, “mapping” suggests that path must begin with `/resources` and can include any sub-path thereafter. The “location” attribute refers to the location of the files to be served. Any URL whose path begins with `/resources` will be automatically served from `/resources`.

2.2 Spread Spring Configuration across multiple XML files for maintainability

Unless specified otherwise, a Spring web application by default will look for `<servlet-name>-servlet.xml` in WEB-INF as the Spring configuration file where `<servlet-name>` is the name of the DispatcherServlet in web.xml. However, it is better to organize configuration into several files such as datasource-context.xml, persistence-context.xml, service-context.xml, sandbox-security.xml etc., for better maintainability. These configuration files can be loaded individually via the following configurations in web.xml.

ContextLoaderListener is a servlet listener that loads additional configuration into a Spring application context in addition to the application context created by DispatcherServlet.

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

<!-- The definition of the Root Spring Container shared by all Servlets and Filters -->
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/sandbox-security.xml
    classpath:service-context.xml
    classpath:persistence-context.xml
    classpath:datasource-context.xml
  </param-value>
</context-param>
```

Figure #6: declaring configuration files

The XML files prefixed with “classpath” are to be loaded from the application classpath. This means these XML files could be in jar files scattered across the application too.

Note: if the individual files aren't mentioned for additional application context configuration, ContextLoaderListener will look for a Spring configuration file at /WEB-INF/applicationContext.xml

2.3 Prefer Spring's annotation driven request mapping in lieu of `<*>HandlerMapping` schemes

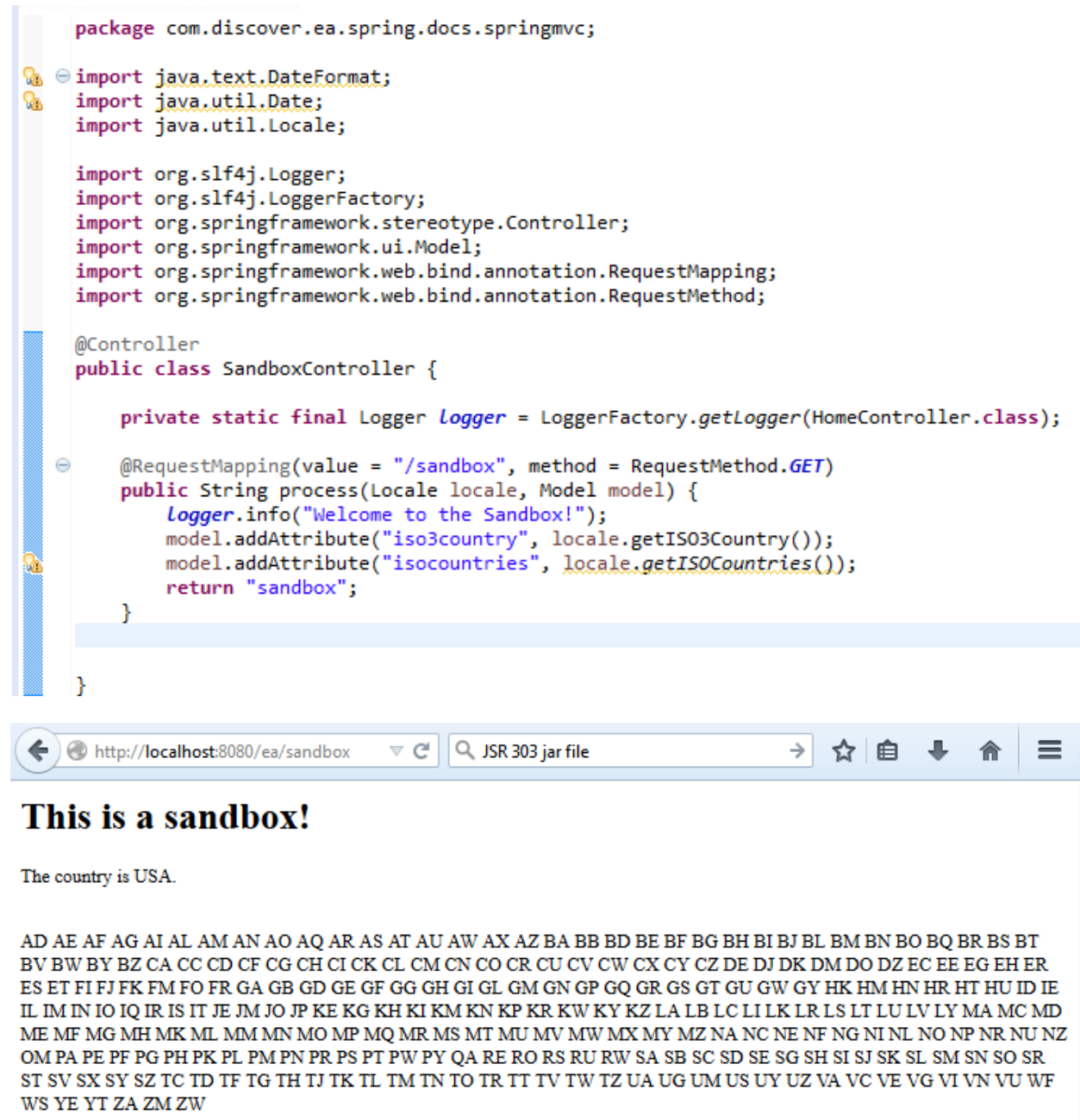
Enabling annotation based mapping requires the following element in `<servlet-name>-servlet.xml`

```
<mvc:annotation-driven/>
```

The single line configuration change provides the following out-of-the-box: JSR-303 JavaBean validation; support; message conversion; and field formatting. (Note: JSR is an acronym for Java Specification Request – a description for a specification for the Java platform.)

3 Request processing

3.1 Enforce fine grained control over request processing with @RequestMapping



```
package com.discover.ea.spring.docs.springmvc;

import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class SandboxController {

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

    @RequestMapping(value = "/sandbox", method = RequestMethod.GET)
    public String process(Locale locale, Model model) {
        logger.info("Welcome to the Sandbox!");
        model.addAttribute("iso3country", locale.getISO3Country());
        model.addAttribute("isocountries", locale.getISOCountries());
        return "sandbox";
    }

}
```

The browser screenshot shows the URL `http://localhost:8080/ea/sandbox` and the page content:

This is a sandbox!

The country is USA.

AD AE AF AG AI AL AM AN AO AQ AR AS AT AU AW AX AZ BA BB BD BE BF BG BH BI BJ BL BM BN BO BQ BR BS BT BV BW BY BZ CA CC CD CF CG CH CI CK CL CM CN CO CR CU CV CW CX CY CZ DE DJ DK DM DO DZ EC EE EG EH ER ES ET FI FJ FK FM FO FR GA GB GD GE GF GG GH GI GL GM GN GP GQ GR GS GT GU GW GY HK HM HN HR HT HU ID IE IL IM IN IO IQ IR IS IT JE JM JO JP KE KG KH KI KM KN KP KR KW KY KZ LA LB LC LI LK LR LS LT LU LV LY MA MC MD ME MF MG MH MK ML MM MN MO MP MQ MR MS MT MU MV MW MX MY MZ NA NC NE NF NG NI NL NO NP NR NU NZ OM PA PE PF PG PH PK PL PM PN PR PS PT PW PY QA RE RO RS RU RW SA SB SC SD SE SG SH SI SJ SK SL SM SN SO SR ST SV SX SY SZ TC TD TF TG TH TJ TK TL TM TN TO TR TT TV TW TZ UA UG UM US UY UZ VA VC VE VG VI VN VU WF WS YE YT ZA ZM ZW

In the above Controller example, the method process is associated with “/sandbox” relative to the context and the request processing is limited to “GET” only.

Figure 7: precise control via @RequestMapping

The complete list of parameters that can be specified in the annotation is described below. This enables precise control over request processing.

Optional Elements	
Modifier and Type	Optional Element and Description
String[]	consumes The consumable media types of the mapped request, narrowing the primary mapping.
String[]	headers The headers of the mapped request, narrowing the primary mapping.
RequestMethod[]	method The HTTP request methods to map to, narrowing the primary mapping: GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE.
String	name Assign a name to this mapping.
String[]	params The parameters of the mapped request, narrowing the primary mapping.
String[]	produces The producible media types of the mapped request, narrowing the primary mapping.
String[]	value The primary mapping expressed by this annotation.

Figure #8: @RequestMapping parameters

3.2 Leverage @Valid to validate input prior to further request processing

JSR-303 or Java Bean validation is supported out of the box with Spring. The @Valid annotation can be specified as shown below in a Controller implementation. Also note that @Valid works on nested objects too. In the example below, AccountHolder object's will invoke validator on Address as well considering that AccountHolder contains Address.

```
package com.discover.ea.spring.docs.springmvc;

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

@Controller
@RequestMapping(value = "/sandbox", method = RequestMethod.GET, params="new")
public class AccountCreationController {

    public String processRequest(@Valid AccountHolder account)
    {
        return "temp";
    }
}
```

```

package com.discover.ea.spring.docs.springmvc;

import javax.validation.constraints.Max;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;

public class Address {

    @NotNull (message="city is mandatory")
    @Pattern(regexp = "[a-z-A-Z]*", message = "city has invalid characters")
    private String city;

    @NotNull (message="state is mandatory")
    private String state;

    @NotNull (message="address is mandatory")
    private String address;

    @NotNull (message="zipcode is mandatory")
    @Max(5)
    private String zipcode;
}

```

```

package com.discover.ea.spring.docs.springmvc;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Null;
import javax.validation.constraints.Size;

public class AccountHolder {

    @NotNull
    @Size (max=20)
    private String first_name;

    @NotNull
    @Size (max=20)
    private String last_name;

    @Valid
    private Address homeAddress;

    @Size (max=16)
    @NotNull
    private String accountNumber;
}

```

Figure #9: use of JSR-303 JavaBean validation API

3.3 Validate portions of Value Objects using validation groups (JSR 303)

```
package com.discover.ea.spring.docs.springmvc;

import javax.validation.Valid;

public class AccountHolder {

    @NotNull
    @Size (max=20)
    private String first_name;

    @NotNull
    @Size (max=20)
    private String last_name;

    @Valid
    @NotNull (groups={ChangeAddress.class})
    private Address homeAddress;

    @Size (max=16)
    @NotNull (groups={ChangeAccountNumber.class})
    private String accountNumber;

    public interface ChangeAddress {};

    public interface ChangeAccountNumber {};

}
```

Figure #10: use of validation groups in the AccountHolder value object

Validation groups are defined via markers in the form of interfaces ChangeAddress and ChangeAccountNumber.

In the following controller method, the AccountHolder ValueObject is validated only for the “accountNumber” attribute. Thus a partial validation is accomplished.

```
@RequestMapping (value="/accounts/create", method=RequestMethod.GET)
public AccountHolder createAccount(@Validated({AccountHolder.ChangeAddress.class}) AccountHolder accHolder) {

    return null;

}
```

Figure #11: a controller implementing a partial validation on a value object

Note: the @Validated annotation can take a number of groups separated by commas as shown below

```
@RequestMapping (value="/accounts/create", method=RequestMethod.GET)
public AccountHolder createAccount(@Validated({AccountHolder.ChangeAddress.class, AccountHolder.ChangeAccountNumber.class}) AccountHolder accHolder) {

    return null;

}
```

Figure #12: a controller implementing a partial validation on a value object with two validation groups.

4 Exception Handling

4.1 Add @ResponseStatus to application specific, developer created exceptions

Consider the following exception annotated with @ResponseStatus

```
// 404 status code
@ResponseStatus(value=HttpStatus.NOT_FOUND, reason="Account does not exist")
public class AccountNotFoundException extends RuntimeException {
```

When the above exception is thrown in a Controller method (shown below) and is not handled anywhere in the controller method, it will trigger the appropriate HTTP response with the specific status code. Had the exception not been handled as mentioned above, we will receive a HTTP 500 response code.

```
@RequestMapping (value="/accounts/{acct_id}", method=RequestMethod.GET)
public String viewAccount(@PathVariable("acct_id") int acct_id, Model mdl){

    AccountHolder account = accountCache.findAccount(acct_id);
    if(account == null)
        throw new AccountNotFoundException(acct_id);
    mdl.addAttribute(account);
    return "accountView";

}
```

Figure #13: use of @ResponseStatus annotation

4.2 Utilize @ControllerAdvice to implement global exception handling logic

@ControllerAdvice on a class allows the use of exception handling logic across the entire application as opposed to an individual controller. Any class annotated as @ControllerAdvice supports three types of methods:

- Methods annotated with @ExceptionHandler
- Methods annotated with @ModelAttribute
- Methods annotated with @InitBinder

In this section, our interest is only in @ExceptionHandler. Please ensure that your Spring Configuration defines the “mvc” namespace. If not, the @ControllerAdvice will not be loaded.

Here is an example:

```
package com.discover.ea.spring.docs.springmvc;

import java.sql.SQLException;

import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class ApplicationWideExceptionHandler {

    @ExceptionHandler(SQLException.class)
    public String handleSQLException(SQLException sqle){

        return "sql_error";

    }

}
```

Figure #14: use of @ControllerAdvice annotation

Note: If you already have SimpleMappingExceptionHandler configured, it may be easier to add exception classes to this resolver as opposed to writing @ControllerAdvice.