Learn Spring framework:

- Understand the core of Spring
- Understand Spring MVC
- Understand Spring AOP
- Understand Spring Data JPA
- Spring Dependency Injection (XML)
- Spring Dependency Injection (Annotations)

# 14 Tips for Writing Spring MVC Controller

Written by Nam Ha Minh
Last Updated on 20 June 2019 | Print Email

In this article, I'm going to share with you some fundamental techniques and good practices involve in writing controller class with Spring MVC framework. Typically in Spring MVC, we write a controller class to handle requests coming from the client. Then the controller invokes a business class to process business-related tasks, and then redirects the client to a logical view name which is resolved by the Spring's dispatcher servlet in order to render results or output. That completes a round trip of a typical request-response cycle.

Here are summary of the tips you will learn throughout this article:



## 1. Using the @Controller stereotype

This is the simplest way for creating a controller class to handle one or multiple requests. Just by annotating a class with the `@Controller` stereotype, for example:

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

@Controller
public class HomeController {

    @RequestMapping("/")
    public String visitHome() {

        // do something before returning view name

        return "home";
    }
}
```

As you can see, the `visitHome()` method handles requests coming to the application's context path (/) by redirecting to the view named **home**.

**NOTE:** the `@Controller` stereotype can only be used with annotation-driven is enabled in the Spring's configuration file:

```
<annotation-driven />
```

When annotation-driven is enabled, Spring container automatically scans for classes under the package specified in the following statement:

```
<context:component-scan base-package="net.codejava.spring" />
```

The classes annotated by the `@Controller` annotation are configured as controllers. This is the most preferable way because its simplicity: no need to declare beans for controllers in the configuration file.

**NOTE:** By using the `@Controller` annotation, you can have a multi-actions controller class that is able to serve multiple different requests. For example:

```
@Controller
public class MultiActionController {

    @RequestMapping("/listUsers")
    public ModelAndView listUsers() {

    }

    @RequestMapping("/saveUser")
    public ModelAndView saveUser(User user) {

    }

    @RequestMapping("/deleteUser")
    public ModelAndView deleteUser(User user) {

    }
}
```

As you can see in the above controller class, there are 3 handler methods that processes 3 different requests */listUsers*, */saveUser* and */deleteUser*, respectively.

## 2. Implementing the Controller Interface

Another (and maybe classic) way of creating a controller in Spring MVC is having a class implemented the `Controller` interface. For example:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class MainController implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        System.out.println("Welcome main");
        return new ModelAndView("main");
    }
}
```

The implementing class must override the `handleRequest()` method which will be invoked by the Spring dispatcher servlet when a matching request comes in. The request URL pattern handled by this controller is defined in the Spring's context configuration file as follows:

```
<bean name="/main" class="net.codejava.spring.MainController" />
```

However, a drawback of this approach is the controller class cannot handle multiple request URLs.

## 3. Extending the AbstractController Class

Having your controller class extended the `AbstractController` class if you want to easily control the supported HTTP methods, session and content caching. Consider the following example:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class BigController extends AbstractController {

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        System.out.println("You're big!");
        return new ModelAndView("big");
    }
}
```

This creates a single-action controller with configurations regarding supported methods, session and caching can be specified in the bean declaration of the controller. For example:

```
<bean name="/big" class="net.codejava.spring.BigController">
    <property name="supportedMethods" value="POST"/>
</bean>
```

This configuration indicates that the only POST method is supported by this controller's hander method. For other configuration (session, caching) see: AbstractController.

Spring MVC also offers several controller classes which are designed for specific purposes: AbstractUrlViewController, MultiActionController, ParameterizableViewController, ServletForwardingController, ServletWrappingController, UrlFilenameViewController.

## 4. Specifying URL Mapping for Handler Method

This is the mandatory task you must do when coding a controller class which is designed for handling one or more specific requests. Spring MVC provides the **`@RequestMapping`** annotation which is used for specifying URL mapping. For example:

```
@RequestMapping("/login")
```

That maps the URL pattern */login* to be handled by the annotated method or class. When this annotation is used at class level, the class becomes a single-action controller. For example:

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

@Controller
@RequestMapping("/hello")
public class SingleActionController {

    @RequestMapping(method = RequestMethod.GET)
    public String sayHello() {
        return "hello";
    }
}
```

When the **`@RequestMapping`** annotation is used at method level, you can have a multi-action controller. For example:

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

@Controller
public class UserController {

    @RequestMapping("/listUsers")
    public String listUsers() {
        return "ListUsers";
    }

    @RequestMapping("/saveUser")
    public String saveUser() {
        return "EditUser";
    }

    @RequestMapping("/deleteUser")
    public String deleteUser() {
        return "DeleteUser";
    }
}
```

The `@RequestMapping` annotation can be also used for specifying multiple URL patterns to be handled by a single method. For example:

```
@RequestMapping({"/hello", "/hi", "/greetings"})
```

In addition, this annotation has other properties which may be useful in some cases, e.g. the **method** property which is covered next.

## 5. Specifying HTTP Request Methods for Handler Method

You can specify which HTTP method (GET, POST, PUT,…) is supported by a handler method by using the **method** property of the `@RequestMapping` annotation. Here's an example:

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

@Controller
public class LoginController {

    @RequestMapping(value = "/login", method = RequestMethod.GET)
    public String viewLogin() {
        return "LoginForm";
    }

    @RequestMapping(value = "/login", method = RequestMethod.POST)
    public String doLogin() {
        return "Home";
    }
}
```

As you can see, this controller has two methods that handle the same URL pattern */login*, but the former is for GET method and the latter is for POST method.

– nore information about using the `@RequestMapping` annotation, see:
questMapping annotation.

## 6. Mapping Request Parameters to Handler Method

One of cool features of Spring MVC is that, you can retrieve request parameters as regular parameters of the handler method by using the`@RequestParam` annotation. This is a good way to decouple the controller from the `HttpServletRequest` interface of Servlet API.

Let's see various examples. Consider the following method:

```
1  @RequestMapping(value = "/login", method = RequestMethod.POST)
2  public String doLogin(@RequestParam String username,
3                        @RequestParam String password) {
4
5  }
```

Spring binds the method parameters `username` and `password` to the HTTP request parameters with the same names. That means you can invoke a URL as follows (if request method is GET):

*http://localhost:8080/spring/login?username=scott&password=tiger*

Type conversion is also done automatically. For example, if you declare a parameter of type **integer** as follows:

```
1  @RequestParam int securityNumber
```

Then Spring will automatically convert value of the request parameter (String) to the specified type (integer) in the handler method.

In case the parameter name is different than the variable name. You can specify actual name of the parameter as follows:

```
1  @RequestParam("SSN") int securityNumber
```

The `@RequestParam` annotation also has additional 2 attributes which might be useful in some cases. The **required**attribute specifies whether the parameter is mandatory or not. For example:

```
1  @RequestParam(required = false) String country
```

That means the parameter **country** is optional, hence can be missing from the request. In the above example, the variable **country** will be null if there is no such parameter present in the request.

Another attribute is **defaultValue**, which can be used as a fallback value when the request parameter is empty. For example:

```
@RequestParam(defaultValue = "18") int age
```

Spring also allows us to access all parameters as a `Map` object if the method parameter is of type `Map<String, String>`. For example:

```
1  doLogin(@RequestParam Map<String, String> params)
```

Then the map `params` contains all request parameters in form of key-value pairs.

For more information about using the **`@RequestParam`** annotation, see: @RequestParam annotation.

## 7. Returning Model And View

After business logic is processed, a handler method should return a view which is then resolved by the Spring's dispatcher servlet. Spring allows us to return either a `String` or a `ModelAndView` object from the hander method. In the following example, the handler method returns a String represents a view named "LoginForm":

```
1  @RequestMapping(value = "/login", method = RequestMethod.GET)
2  public String viewLogin() {
3      return "LoginForm";
4  }
```

That's the simplest way of returning a view name. But if you want to send additional data to the view, you must return a `ModelAndView` object. Consider the following handler method:

```
1  @RequestMapping("/listUsers")
2  public ModelAndView listUsers() {
3
4      List<User> listUser = new ArrayList<>();
5      // get user list from DAO...
6
7      ModelAndView modelView = new ModelAndView("UserList");
8      modelView.addObject("listUser", listUser);
9
10     return modelView;
11 }
```

As you can see, this handler method returns a `ModelAndView` object that holds the view name "UserList" and a collection of **`User`** objects which can be used in the view.

Spring is also very flexible, as you can declare the `ModelAndView` object as a parameter of the handler method instead of creating a new one. Thus, the above example can be re-written as follows:

```
@RequestMapping("/listUsers")
public ModelAndView listUsers(ModelAndView modelView) {

    List<User> listUser = new ArrayList<>();
    // get user list from DAO...

    modelView.setViewName("UserList");
    modelView.addObject("listUser", listUser);

    return modelView;
}
```

You can learn more about the **`ModelAndView`** class by visiting: ModelAndView class.

## 8. Putting Objects into the Model

In an application that follows the MVC architecture, the controller (C) should pass data into the model (M) which is then used in the view (V). As we see in the previous example, the **`addObject()`** method of the **`ModelAndView`** class is for putting an object to the model, in form of name-value pair:

```
modelView.addObject("listUser", listUser);
modelView.addObject("siteName", new String("CodeJava.net"));
modelView.addObject("users", 1200000);
```

Again, Spring is very flexible. You can declare a parameter of type **`Map`** in the handler method, Spring uses this map to store objects for the model. Let's see another example:

```
@RequestMapping(method = RequestMethod.GET)
public String viewStats(Map<String, Object> model) {
    model.put("siteName", "CodeJava.net");
    model.put("pageviews", 320000);

    return "Stats";
}
```

This is even simpler than using **`ModelAndView`** object. Depending on your taste, you can use either Map or **`ModelAndView`** object. Thanks for the flexibility of Spring.

## 9. Redirection in Handler Method

In case you want to redirect the user to another URL if a condition is met, just append **redirect:/** before the URL. The following code snippet gives an example:

```
// check login status....

if (!isLogin) {
    return new ModelAndView("redirect:/login");
}


// return a list of Users
```

e above code, the user will be redirected to the */login* URL if he is not logged in.

## 10. Handling Form Submission and Form Validation

Spring makes it easy to handle form submission, by providing the **`@ModelAttribute`** annotation for binding form fields to a form backing object, and the **`BindingResult`** interface for validating form fields. The following code snippet shows a typical handler method that is responsible for handling and validating form data:

```
@Controller
public class RegistrationController {

    @RequestMapping(value = "/doRegister", method = RequestMethod.POST)
    public String doRegister(
        @ModelAttribute("userForm") User user, BindingResult bindingResult) {

        if (bindingResult.hasErrors()) {
            // form validation error

        } else {
            // form input is OK
        }

        // process registration...

        return "Success";
    }
}
```

Learn more about the **`@ModelAttribute`** annotation and the **`BindingResult`**interface from Spring's official documentation:

- Using @ModelAttribute on a method argument
- Using @ModelAttribute on a method
- Interface BindingResult

You can learn more and in-depth about form handling and form validation in Spring MVC by reading Spring MVC Form Handling Tutorial and Spring MVC Form Validation Tutorial.

## 11. Handling File Upload

Spring also makes it easy to handle file upload within a handler method, by automatically binding upload data to an array of **`CommonsMultipartFile`** objects. Spring uses Apache Commons FileUpload as the underlying multipart resolver.

The following code snippet shows how easy it is to get files uploaded from the client:

```
@RequestMapping(value = "/uploadFiles", method = RequestMethod.POST)
public String handleFileUpload(
        @RequestParam CommonsMultipartFile[] fileUpload) throws Exception {


    for (CommonsMultipartFile aFile : fileUpload){

        // stores the uploaded file
        aFile.transferTo(new File(aFile.getOriginalFilename()));

    }


    return "Success";
}
```

You can learn the complete solution for handling file upload with Spring MVC by following the Spring MVC File Upload Tutorial.

## 12. Autowiring Business Classes in the Controller

A controller should delegate the processing of business logic to relevant business classes. For this purpose, you can use the `@Autowired` annotation to let Spring automatically injects actual implementation of a business class to the controller. Consider the following code snippet of a controller class:

```
@Controller
public class UserController {

    @Autowired
    private UserDAO userDAO;

    public String listUser() {
        // handler method to list all users
        userDAO.list();
    }

    public String saveUser(User user) {
        // handler method to save/update a user
        userDAO.save(user);
    }

    public String deleteUser(User user) {
        // handler method to delete a user
        userDAO.delete(user);
    }

    public String getUser(int userId) {
        // handler method to get a user
        userDAO.get(userId);
    }
}
```

Here, all the business logics related to User management is provided by an implementation of the `UserDAO` interface. For example:

```
interface UserDAO {

    List<User> list();

    void save(User user);

    void checkLogin(User user);
}
```

By using the `@Autowired` annotation, the handler methods can delegate tasks to the business class, as we can see in the above example:

```
List<User> listUser = userDAO.list();
```

For more information about the `@Autowired` annotation, see: Annotation Type Autowired.

## 13. Accessing HttpServletRequest and HttpServletResponse

In some cases, you need to directly access the `HttpServletRequest` or `HttpServletResponse` objects within a handler method. By the flexibility of Spring, just add relevant parameter to the handler method. For example:

```
@RequestMapping("/download")
public String doDownloadFile(
        HttpServletRequest request, HttpServletResponse response) {

    // access the request

    // access the response

    return "DownloadPage";
}
```

Spring detects and automatically injects the `HttpServletRequest` and `HttpServletResponse` objects into the method. Then you can access the request and response such as getting `InputStream`, `OutputStream` or returning a specific HTTP code.

## 14. Following the Single Responsibility Principle

Finally, there are two good practices you should follow when designing and coding controllers in Spring MVC:

- A controller class should not execute business logic. Instead, it should delegate business processing to relevant business classes. This keeps the controller focusing on its designed responsibility is to control workflows of the application. For example:

```
@Controller
public class UserController {

    @Autowired
    private UserDAO userDAO;

    public String listUser() {
        // handler method to list all users
        userDAO.list();
    }

    public String saveUser(User user) {
        // handler method to save/update a user
        userDAO.save(user);
    }

    public String deleteUser(User user) {
        // handler method to delete a user
        userDAO.delete(user);
    }

    public String getUser(int userId) {
        // handler method to get a user
        userDAO.get(userId);
    }
}
```

- Create each separate controller for each business domain. For example, **`UserController`** for controlling workflows of the user management, **`OrderController`** for controlling workflows of order processing, etc. For example:

```
@Controller
public class UserController {

}


@Controller
public class ProductController {

}

@Controller
public class OrderController {

}

@Controller
public class PaymentController {

}
```

So far Ihave shared 14 tips that help you write controller classes in Spring MVC properly and efficiently. However that's not the end. If you have other tips or suggestions, feel free to share your thoughts. If you want to fully learn Spring framework, I recommend you to read this Pro Spring 5 book, or take this Spring Master Class course on Udemy.

## Other Spring Tutorials:

- Understand the core of Spring framework

- Understand Spring MVC
- Understand Spring AOP
- Spring MVC beginner tutorial with Spring Tool Suite
- Spring MVC Form Handling Tutorial
- Spring MVC Form Validation Tutorial
- Understand Spring Data JPA

## About the Author:

Nam Ha Minh is certified Java programmer (SCJP and SCWCD). He started programming with Java in the time of Java 1.4 and has been falling in love with Java since then. Make friend with him on Facebook.

### Add comment

Name E-mail

comment

500 symbols left

Notify me of follow-up comments

I'm not a robot reCAPTCHA Privacy - Terms

Send

### Comments

1 2 3 4 5 6 7 8

#40**Manikumar** 2019-01-17 01:51
Superb.. Thank You Guys...
Quote

#39**kiran** 2018-10-31 01:19
Very nice article! Keep it up
Quote

#38**Gilbert** 2018-10-19 12:38
Good advice. Thanks for share!
Quote

#37**nkc** 2018-05-02 19:56
thanks for sharing. Can u also pls share the best Spring MVC architecture to support in N-tire application (like.., UI frame work, All Spring components, DB, LDAP)

Quote

#36**Michał Ruszkowski** 2018-02-15 15:52

Very good overview. This "14 tips" gave me better explanation of Spring MVC Controllers than "tutorials" I've been reading.

Quote





/