Spring MVC Framework for Web 2.0

Rohan V. Thakare santhoshkakade@gmail.com
Bhushan Sapre bhushansapre316@gmail.com
Dr. B. B. Meshram bbmeshram@vji.org.in
Santosh Kakade santosh20may@gmail.com

Abstract - When building rich user experience web applications, an abundance of web application frameworks is available and only little guidance for making the decision which one to choose. Web 2.0 applications allow individuals to manage their content online and to share it with other users and services on the Web. Such sharing requires access control to be put in place. Existing access control solutions, however, are unsatisfactory as they do not offer the functionality that users need in the open and user-driven Web environment. Out of all those web development framework the most popular is MVC Framework. Model–view–controller (MVC) is a software architecture, currently considered an architectural pattern used in software engineering. The pattern isolates "domain logic" (the application logic for the user) from the user interface (input and presentation), permitting independent development, testing and maintenance of each (separation of concerns). Model View Controller (MVC) pattern creates applications that separate the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

Key Word - Spring MVC, Architecture, XStudio, SOA, Controller.

I. INTRODUCTION

So exactly what defines a web site as being “Web 2.0”? There are many different opinions on this, making it difficult to pinpoint an exact definition. But it will get clear when we will go through all available web development frameworks. Various architectures for web development are as follows

- **N-tier Architecture**

  In software engineering, multi-tier architecture (often referred to as n-tier architecture) is a client–server architecture in which the presentation, the application processing, and the data management are logically separate processes.[6] For example, an application that uses middleware to service data requests between a user and a database employs multi-tier architecture. The most widespread use of multi-tier architecture is the three-tier architecture. N-tier application architecture provides a model for developers to create a flexible and reusable application. By breaking up an application into tiers, developers only have to modify or add a specific layer, rather than have to rewrite the entire application over. There should be a presentation tier, a business or data access tier, and a data tier. The concepts of layer and tier are often used interchangeably.

- **Service Oriented Architecture**

  In software engineering, a Service-Oriented Architecture (SOA) is a set of principles and methodologies for designing and developing software in the form of interoperableservices. These services are well-defined business functionalities that are built as software components (discrete pieces of code and/or data structures) that can be reused for different purposes. SOA design principles are used during the phases of systems development and integration. SOA also generally provides a way for consumers of services, such as web-based applications, to be aware of available SOA-based services. For example, several disparate departments within a company may develop and deploy SOA services in different implementation languages; their respective clients will benefit from a well-understood, well-defined interface to access them. XML is often used for interfacing with SOA services, though this is not required. JSON is also becoming increasingly common.

- **MVC Architecture**

  MVC is part of a new generation of application development and integration tools from Hyfinity. Based on native XML, MVC enables the rapid development of enterprise-scale browser-based applications. MVC provides intuitive graphical IDEs for rapidly assembling complete interactive applications. One of the key differences between MVC and more traditional approaches is the information flow concept that preserves document-based information from capture, validation through to data binding and routing. All of this is performed using native XML.

  MVC uses XStudio for developing XML applications and XPPlatform for hosting the resulting applications. MVC is primarily concerned with the development of interactive self-service applications and employs a tool within XStudio known as FormMaker for building complete web applications.

II. MVC ARCHITECTURE

As we discussed in the previous section, it is common to think of an application as having three main layers: presentation (UI), application logic, and resource management. In MVC, the presentation layer is split into controller and view. The most important separation is between presentation and application logic. The View/Controller split is less so. MVC encompasses more of the architecture of an application than is typical for a design pattern. Hence the term architectural pattern may be useful, or perhaps an aggregate design pattern.

- **Model**

  The domain-specific representation of the information on which the application operates. The model is another name for the application logic layer (sometimes also called the

Copyright © 2012 IJEIR, All right reserved
domain layer). Application (or domain) logic adds meaning to raw data (e.g., calculating if today is the user’s birthday, or the totals, taxes and shipping charges for shopping cart items). Many applications use a persistent storage mechanism (such as a database) to store data. MVC does not specifically mention the resource management layer because it is understood to be underneath or encapsulated by the Model.

**View**

Renders the model into a form suitable for interaction, typically a user interface element. MVC is often seen in web applications, where the view is the HTML page and the code which gathers dynamic data for the page.

**Controller**

Processes and responds to events, typically user actions, and may invoke changes on the model and view.

Though MVC comes in different flavours, the control flow generally works as follows:

1. The user interacts with the user interface in some way (e.g., user presses a button).
2. A controller handles the input event from the user interface, often via a registered handler or callback.
3. The controller accesses the model, possibly updating it in a way appropriate to the user’s action (e.g., controller updates user’s shopping cart). Complex controllers are often structured using the command pattern to encapsulate actions and simplify extension.
4. A view uses the model to generate an appropriate user interface (e.g., view produces a screen listing the shopping cart contents). The view gets its own data from the model. The model has no direct knowledge of the view. (However, the observer pattern can be used to allow the model to indirectly notify interested parties, potentially including views, of a change.)
5. The user interface waits for further user interactions, which begins the cycle anew.

Above figure summarizes the relationship between the Model, View, and Controller is provided below.

A. **Implementation Strategy**

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for uploading files.

B. **Features of Spring Web MVC**

- Clear separation of roles. Each role -- controller, validator, command object, form object, model object, DispatcherServlet, handler mapping, view resolver, and so on can be fulfilled by a specialized object.
- Powerful and straightforward configuration of both framework and application classes as JavaBeans. This configuration capability includes easy referencing across contexts, such as from web controllers to business objects and validators.
- Adaptability, non-intrusiveness, and flexibility. Define any controller method signature you need, possibly using one of the parameter annotations (such as @RequestParam, @RequestHeader, @PathVariable, and more) for a given scenario.
- Reusable business code, no need for duplication. Use existing business objects as command or form objects instead of mirroring them to extend a particular framework base class.
- Customizable binding and validation. Type mismatches as application-level validation errors that keep the offending value, localized date and number binding, and so on instead of String-only form objects with manual parsing and conversion to business objects.
- Customizable handler mapping and view resolution. Handler mapping and view resolution strategies range from simple URL-based configuration, to sophisticated, purpose-built resolution strategies. Spring is more flexible than web MVC frameworks that mandate a particular technique.
- Flexible model transfer. Model transfer with a name/value Map supports easy integration with any view technology.
- Customizable locale and theme resolution, support for JSPs with or without Spring tag library, support for JSTL, support for Velocity without the need for extra bridges, and so on.
- A simple yet powerful JSP tag library known as the Spring tag library that provides support for features such as data binding and themes. The custom tags allow for maximum flexibility in terms of markup code.

![MVC Architecture](image)

Fig. 1. MVC Architecture
• A JSP form tag library, introduced in Spring 2.0, that makes writing forms in JSP pages much easier.
• Beans whose lifecycle is scoped to the current HTTP request or HTTP Session. This is not a specific feature of Spring MVC itself, but rather of the WebApplicationContext container(s) that Spring MVC uses.

C. Pluggability of other MVC Implementation

If you do not want to use Spring's web MVC, but intend to leverage other solutions that Spring offers, you can integrate the web MVC framework of your choice with Spring easily. Simply start up a Spring root application context through its ContextLoaderListener, and access it through its ServletContext attribute (or Spring's respective helper method) from within a Struts or WorkWeb action. No "plug-ins" are involved, so no dedicated integration is necessary. From the web layer's point of view, you simply use Spring as a library, with the root application context instance as the entry point.

Your registered beans and Spring's services can be at your fingertips even without Spring's Web MVC. Spring does not compete with Struts or WorkWeb in this scenario. It simply addresses the many areas that the pure web MVC frameworks do not, from bean configuration to data access and transaction handling. So you can enrich your application with a Spring middle tier and/or data access tier, even if you just want to use, for example, the transaction abstraction with JDBC or Hibernate.

Spring's web MVC framework is, like many other web MVC frameworks, request-driven, designed around a central servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications. Spring's DispatcherServlet however, does more than just that. It is completely integrated with the Spring IoC container and as such allows you to use every other feature that Spring has.

The request processing workflow of the Spring Web MVC DispatcherServlet is illustrated in the following diagram. The pattern-savvy reader will recognize that the DispatcherServlet is an expression of the “Front Controller” design pattern (this is a pattern that Spring Web MVC shares with many other leading web frameworks).

The DispatcherServlet is an actual Servlet (it inherits from the HttpServlet base class), [3] and as such is declared in the web.xml of your web application. You need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the same web.xml file. This is standard J2EE servlet configuration; the following example shows such a DispatcherServlet declaration and mapping.

In the preceding example, all requests ending with .form will be handled by the exampleDispatcherServlet. This is only the first step in setting up Spring Web MVC. You now need to configure the various beans used by the Spring Web MVC framework. Upon initialization of a DispatcherServlet, the framework looks for a file named [servlet-name]-

servlet.xml in the WEB-INF directory of your web application and creates the beans defined there, overriding the definitions of any beans defined with the same name in the global scope.

Fig. 2. Request Processing Workflow

The Spring DispatcherServlet uses special beans to process requests and render the appropriate views. These beans are part of Spring Framework. You can configure them in the WebApplicationContext, just as you configure any other bean. However, for most beans, sensible defaults are provided so you initially do not need to configure them. These beans are described in the following table.

After you set up a DispatcherServlet, and a request comes in for that specific DispatcherServlet, the DispatcherServlet starts processing the request as follows:

1. The WebApplicationContext is searched for and bound in the request as an attribute that the controller and other elements in the process can use. It is bound by default under the key DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE.
2. The locale resolver is bound to the request to enable elements in the process to resolve the locale to use when processing the request (rendering the view, preparing data, and so on). If you do not need locale resolving, you do not need it.
3. The theme resolver is bound to the request to let elements such as views determine which theme to use. If you do not use themes, you can ignore it.
4. If you specify a multipart file resolver, the request is inspected for multipart; if multipart are found, the request is wrapped in a MultipartHttpServletRequest for further processing by other elements in the process.
5. An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) is executed in order to prepare a model or rendering.
6. If a model is returned, the view is rendered. If no model is returned, (may be due to a preprocessor or postprocessor intercepting the request, perhaps for
security reasons), no view is rendered, because the request could already have been fulfilled.

III. IMPLEMENTING CONTROLLER

Spring 2.5 introduced an annotation-based programming model for MVC controllers that uses annotations such as @RequestMapping, @RequestParam, @ModelAttribute, and so on. This annotation support is available for both Servlet MVC and Portlet MVC. Controllers implemented in this style do not have to extend specific base classes or implement specific interfaces. Furthermore, they do not usually have direct dependencies on Servlet or Portlet APIs, although you can easily configure access to Servlet or Portlet facilities. As you can see, the @Controller and @RequestMapping annotations allow flexible method names and signatures. In this particular example the method has no parameters and returns a ModelAndView, but various other (and better) strategies exist, as are explained later in this section. ModelAndView, @Controller, and @RequestMapping form the basis for the Spring MVC implementation. This section documents these annotations and how they are most commonly used in a Servlet environment.

The @Controller annotation indicates that a particular class serves the role of a controller. Spring does not require you to extend any controller base class or reference the Servlet API. However, you can still reference Servlet-specific features if you need to. The @Controller annotation acts as a stereotype for the annotated class, indicating its role. The dispatcher scans such annotated classes for mapped methods and detects @RequestMapping annotations. You can define annotated controller beans explicitly, using a standard Spring bean definition in the dispatcher's context. However, the @Controller stereotype also allows for auto detection, aligned with Spring general support for detecting component classes in the classpath and auto-registering bean definitions for them. To enable auto detection of such annotated controllers, you add component scanning to your configuration.

A. Mapping Requests with @RequestMapping

You use the @RequestMapping annotation to map URLs such as /appointments onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET"/"POST") or specific HTTP request parameters.

The following example shows a controller in a Spring MVC application that uses this annotation:

In the example, the @RequestMapping is used in a number of places. The first usage is on the type (class) level, which indicates that all handling methods on this controller are relative to the /appointments path. The get() method has a further @RequestMapping refinement: it only accepts GET requests, meaning that an HTTP GET for /appointments invokes this method. The post() has a similar refinement, and the getNewForm() combines the definition of HTTP method and path into one, so that GET requests for appointments/new are handled by that method.

B. Supported Handler Methods arguments and Return Types

Handler methods that are annotated with @RequestMapping can have very flexible signatures. Most of them can be used in arbitrary order

- Request or response objects (Servlet API). Choose any specific request or response type, for example HttpServletRequest or HttpServletResponse.
- Session object (Servlet API): of type HttpSession. An argument of this type enforces the presence of a corresponding session. As a consequence, such an argument is never null.
- @PathVariable annotated parameters for access to URI template variables.
- @RequestParam annotated parameters for access to specific Servlet request parameters. Parameter values are converted to the declared method argument type.
- @RequestHeader annotated parameters for access to specific Servlet request HTTP headers. Parameter values are converted to the declared method argument type.
- ModelMap for enriching the implicit model that is exposed to the web view.
- BindingResult validation results for a preceding command or form object.
- Status handle for marking form processing as complete, which triggers the cleanup of session attributes that have been indicated by the @SessionAttributes annotation at the handler type level.

The following return types are supported for handler methods:

- A ModelAndView object, with the model implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.
- A Model object, with the view name implicitly determined through a RequestToViewNameTranslator and the model implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.
- A Map object for exposing a model, with the view name implicitly determined through a RequestToViewNameTranslator and the model implicitly enriched with command objects and the
results of @ModelAttribute annotated reference data accessor methods.

- A View object, with the model implicitly determined through command objects and @ModelAttribute annotated reference data accessor methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above).
- void if the method handles the response itself (by writing the response content directly, declaring an argument of type ServletResponse / HttpServletRequest for that purpose) or if the view name is supposed to be implicitly determined through a RequestToViewNameTranslator (not declaring a response argument in the handler method signature).
- Any other return type is considered to be a single model attribute to be exposed to the view, using the attribute name specified through @ModelAttribute at the method level (or the default attribute name based on the return type class name). The model is implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.

IV. VIEWS

All MVC frameworks for web applications provide a way to address views. [5] Spring provides view resolvers, which enable you to render models in a browser without tying you to a specific view technology. Out of the box, Spring enables you to use JSPs, Velocity templates and XSLT views, for example.

The two interfaces that are important to the way Spring handles views are ViewResolver and View. The ViewResolver provides a mapping between view names and actual views. The View interface addresses the preparation of the request and hands the request over to one of the view technologies.

A. Resolving Views with the ViewResolver Interface

All handler methods in the Spring Web MVC controllers must resolve to a logical view name, either explicitly or implicitly. Views in Spring are addressed by a logical view name and are resolved by a view resolver. Spring comes with quite a few view resolvers. This table lists most of them; a couple of examples follow. The ResourceBundleViewResolver inspects the ResourceBundle identified by the basename, and for each view it is supposed to resolve, it uses the value of the property [viewname].(class) as the view class and the value of the property [viewname].url as the view url.

B. Redirecting To Views

As mentioned previously, a controller typically returns a logical view name, which a view resolver resolves to a particular view technology. For view technologies such as JSPs that are processed through the Servlet or JSP engine, this resolution is usually handled through the combination of InternalResourceViewResolver and InternalResourceView, which issues an internal forward or include via the Servlet API's RequestDispatcher.forward( ) method or RequestDispatcher.include( ) method. For other view technologies, such as Velocity, XSLT, and so on, the view itself writes the content directly to the response stream.

It is sometimes desirable to issue an HTTP redirect back to the client, before the view is rendered. This is desirable, for example, when one controller has been called with POSTed data, and the response is actually a delegation to another controller (for example on a successful form submission). In this case, a normal internal forward will mean that the other controller will also see the same POST data, which is potentially problematic if it can confuse it with other expected data. Another reason to perform a redirect before displaying the result is to eliminate the possibility of the user submitting the form data multiple times. In this scenario, the browser will first send an initial POST; it will then receive a response to redirect to a different URL; and finally the browser will perform a subsequent GET for the URL named in the redirect response. Thus, from the perspective of the browser, the current page does not reflect the result of a POST but rather of a GET. The end effect is that there is no way the user can accidentally re-POST the same data by performing a refresh. The refresh forces a GET of the result page, not a resend of the initial POST data.

C. Redirect View

One way to force a redirect as the result of a controller response is for the controller to create and return an instance of Spring'sRedirectView. In this case, DispatcherServlet does not use the normal view resolution mechanism. Rather because it has been given the (redirect) view already, the DispatcherServlet simply instructs the view to do its work.

TheRedirectView issues an HttpSessionResponse->.sendRedirect() call that returns to the client browser as an HTTP redirect. All model attributes are exposed as HTTP query parameters. This means that the model must contain only objects (generally Strings or objects converted to a String representation), which can be readily converted to a textual HTTP query parameter.

If you use RedirectView and the view is created by the controller itself, it is recommended that you configure the redirect URL to be injected into the controller so that it is not baked into the controller but configured in the context along with the view names.

D. The redirect: prefix

While the use of RedirectView works fine, if the controller itself creates the RedirectView, there is no avoiding the fact that the controller is aware that a redirection is happening. This is really suboptimal and couples things too tightly. The controller should not really
care about how the response gets handled. In general it should operate only in terms of view names that have been injected into it.

The special redirect: prefix allows you to accomplish this. If a view name is returned that has the prefix redirect:, the UrlBasedViewResolver (and all subclasses) will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the redirect URL.

The net effect is the same as if the controller had returned a RedirectView, but now the controller itself can simply operate in terms of logical view names. A logical view name such as redirect:/my/response/controller.html will redirect relative to the current servlet context, while a name such as redirect:http://myhost.com/some/arbitrary/path.html will redirect to an absolute URL. The important thing is that, as long as this redirect view name is injected into the controller like any other logical view name, the controller is not even aware that redirection is happening.

E. View Technologies

Spring provides a couple of out-of-the-box solutions for JSP and JSTL views. As of version 2.0, Spring provides a comprehensive set of data binding-aware tags for handling form elements when using JSP and Spring Web MVC. Each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and intuitive to use. The tag-generated HTML is HTML 4.01/XHTML 1.0 compliant.

Unlike other form/input tag libraries, Spring's form tag library is integrated with Spring Web MVC, giving the tags access to the command object and reference data your controller deals with. As you will see in the following examples, the form tags make JSPs easier to develop, read and maintain.

Advantages of MVC Framework

- Spring provides a very clean division between controllers, JavaBean models, and views.
- Spring’s MVC is very flexible. Just about every part of the Spring MVC framework is configurable via plugging in your own interface. Of course we also provide convenience classes as an implementation option.
- Spring, like WebWork, provides interceptors as well as controllers, making it easy to factor out behavior common to the handling of many requests.
- Spring Controllers are configured via IoC like any other objects. This makes them easy to test, and beautifully integrated with other objects managed by Spring.
- Spring MVC web tiers are typically easier to test than Struts web tiers, due to the avoidance of forced concrete inheritance and explicit dependence of controllers on the dispatcher servlet.
- Spring has a well-defined interface to business layer

V. 3-TIER ARCHITECTURE VS. MVC ARCHITECTURE

Communication

- 3-tier: The presentation layer never communicates directly with the data layer-only through the logic layer (linear topology)
- MVC: All layers communicate directly (triangle topology)

Usage

- 3-tier: Mainly used in web applications where the client, middleware and data tiers ran on physically separate platforms
- MVC: Historically used on applications that run on a single graphical workstation (applied to separate platforms as Model 2)

REFERENCES


AUTHOR’S PROFILE

Rohan V. Thakare

was born in Nagpur on 19-Dec-1986. He received BE degree (Computer Science & Engineering) from Vishwakarma Institute of Technology, Pune, Maharashtra, India in 2008. He is pursuing M.Tech degree in Computer Technology [Specialization in Network] from V.J.T.I, Mumbai, India.

Mr. Rohan is a member of Computer Society of India and his paper “Enhancing Mobile Phone Security” in International Journal of Internet Computing is to be published.