SYSTEM REQUIREMENTS

**Eclipse Software:**

Eclipse is an integrated development environment (IDE) used in computer programming. It contains a base workspace and an extensible plug-in system for customizing the environment. We used this software to develop our project because of it’s compatibility with java programming language. The Eclipse SDK includes the Eclipse Java development tools (JDT), offering an IDE with a built-in Java incremental compiler and a full model of the Java source files. This allows for advanced refactoring techniques and code analysis. The IDE also makes use of a workspace, in this case a set of metadata over a flat file space allowing external file modifications as long as the corresponding workspace resource is refreshed afterward.  
Eclipse implements the graphical control elements of the Java toolkit called Standard  
Widget Toolkit (SWT), whereas most Java applications use the Java standard Abstract  
Window Toolkit (AWT) or Swing. Eclipse's user interface also uses an intermediate graphical  
user interface layer called JFace, which simplifies the construction of applications based on  
SWT.

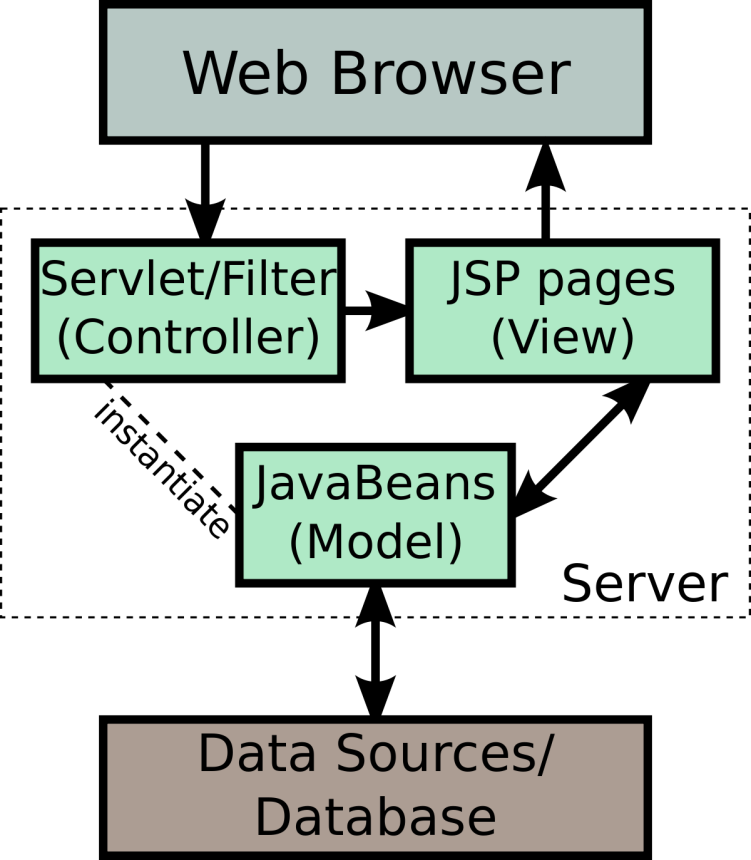
**MYSQL**

MySQL is an open-source relational database management system (RDBMS). A  
relational database organizes data into one or more data tables in which data types may be related to each other; these relations help structure the data. SQL is a language programmers use to create, modify and extract data from the relational database, as well as control user access to the database. In addition to relational databases and SQL, an RDBMS like MySQL works with an operating system to implement a relational database in a computer's storage system, manages users, allows for network access and facilitates testing database integrity and creation of backups

TECHNOLOGIES INVOLVED

**Jakarta Server Pages (JSP):**  
Jakarta Server pages is one of the original java web technology which is being widely  
used to create dynamic web pages that can connect to java backend. It is built on top of the  
Java Servlet specification. JSP may be viewed as a high-level abstraction of Java servlets.  
JSPs are translated into servlets at runtime, therefore JSP is a Servlet; each JSP servlet is  
cached and re-used until the original JSP is modified. Jakarta Server Pages can be used  
independently or as the view component of a server-side model–view–controller design,  
normally with JavaBeans as the model and Java servlets (or a framework such as Apache  
Struts) as the controller.

JSP allows Java code and certain predefined actions to be interleaved with static web  
markup content, such as HTML. The resulting page is compiled and executed on the server to  
deliver a document. The compiled pages, as well as any dependent Java libraries, contain Java  
bytecode rather than machine code. Like any other .jar or Java program, code must be  
executed within a Java virtual machine (JVM) that interacts with the server's host operating  
system to provide an abstract, platform-neutral environment. JSPs are usually used to deliver HTML and XML documents, but through the use of OutputStream, they can deliver other types of data as well. The Web container creates JSP implicit objects like request, response, session, application, config, page, pageContext, out and exception. JSP Engine creates these objects during translation phase. Architecturally, JSP may be viewed as a high-level abstraction of Java servlets. JSPs are translated into servlets at runtime, therefore JSP is a Servlet; each JSP servlet is cached and re-used until the original JSP is modified. JSP can be used independently or as the view component of a server-side modelâ€“viewâ€“controller design, normally with JavaBeans as the model and Java servlets as the controller.



JSP technology is the extension to Servlet technology. The main features of JSP technology are as follows:  
❖ A language for developing JSP pages, which are text-based documents that describe how to process a request and construct a response  
❖ An expression language for accessing server-side objects  
❖ Mechanisms for defining extensions to the JSP language  
Servlets provide URL mapping and request handling capabilities in your Java web applications. Request handling is the bread and butter of Java web application development. In order to respond to requests from the network, a Java web application must first determine what code will respond to the request URL, then marshal a response. Every technology stack has a way of accomplishing request-response handling. In Java, we use servlets (and the Java Servlet API) for this purpose. Think of a servlet as a tiny server whose job is to accept requests and issue responses.

Bootstrap:  
Bootstrap is the most popular open-source framework full of useful and common classes to use in any project. It helps to develop responsive and mobile-first websites faster and easier. It is known for its faster and effortless responsive web development assistance, Bootstrap web design methodology utilize HTML and CMS based templates for user interface components like forms, navigations, alerts, buttons, typography in addition to optional JavaScript extensions. Bootstrap is a web framework that focuses on simplifying the development of informative web pages (as opposed to web apps). The primary purpose of adding it to a web project is to apply Bootstrap's choices of color, size, font and layout to that project. As such,  
the primary factor is whether the developers in charge find those choices to their liking. Once added to a project, Bootstrap provides basic style definitions for all HTML elements. The result is a uniform appearance for prose, tables and form elements across web browsers. In addition, developers can take advantage of CSS classes defined in Bootst rap to further customize the appearance of their contents. For example, Bootstrap has provisioned for light-and dark-colored tables, page headings, more prominent pull quotes, and text with a highlight. Bootstrap also comes with several JavaScript components in the form of jQuery plugins. They provide additional user interface elements such as dialog boxes, tooltips, and carousels. Each Bootstrap component consists of an HTML structure, CSS declarations, and in some cases accompanying JavaScript code. They also extend the functionality of some existing interface elements, including for example an auto-complete function for input fields. The most prominent components of Bootstrap are its layout components, as they affect an entire web page. The basic layout component is called "Container", as every other element in the page is placed in it. Developers can choose between a fixed-width container  
and a fluid-width container. While the latter always fills the width of the web page, the former uses one of the four predefined fixed widths, depending on the size of the screen showing the page:

Smaller than 576 pixels  
576–768 pixels  
768–992 pixels  
992–1200 pixels Larger than  
1200 pixels  
Once a container is in place, other Bootstrap layout components implement a CSS Flexbox layout through defining rows and columns.  
A precompiled version of Bootstrap is available in the form of one CSS file and three JavaScript files that can be readily added to any project. The raw form of Bootstrap, however, enables developers to implement further customization and size optimizations. This raw form  
is modular, meaning that the developer can remove unneeded components, apply a theme and modify the uncompiled Sass files

**JavaScript**

JavaScript is one of the core technologies of the WWW (World Wide Web). It enables interactive web pages and is an essential part of web applications. It has application programming interfaces (APIs) for working with text, dates, regular expressions, standard data structures, and the Document Object Model (DOM). Almost all the websites and web browser uses JavaScript engines to execute client side page behavior. JavaScript engines were  
originally used only in web browsers, but they are now embedded in some servers, usually via Node.js. They are also embedded in a variety of applications created with frameworks such as Electron and Cordova

**Spring MVC**

Spring's Web 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 upload files. The default handler is a very simple Controller interface, just offering a ModelAndView handleRequest(request,response) method. This can already be used for application controllers, but you will prefer the included implementation hierarchy, consisting of, for example AbstractController, AbstractCommandController and SimpleFormController. Application controllers will typically be subclasses of those. Note that you can choose an appropriate base class: if you don't have a form, you don't need a form controller. This is a major difference to Struts. Spring Web MVC allows you to use any object as a command or form object - there is no need to implement a framework-specific interface or base class. Spring's data binding is highly flexible: for example, it treats type mismatches as validation errors that can be evaluated by the application, not as system errors. All this means that you don't need to duplicate your business objects' properties as simple, untyped strings in your form objects just to be able to handle invalid submissions, or to convert the Strings properly. Instead, it is often preferable to bind directly to your business objects. This is another major difference to Struts which is built around required base classes such as Action and ActionForm.

Compared to WebWork, Spring has more differentiated object roles. It supports the notion of a Controller, an optional command or form object, and a model that gets passed to the view. The model will normally include the command or form object but also arbitrary reference data; instead, a WebWork Action combines all those roles into one single object. WebWork does allow you to use existing business objects as part of your form, but only by making them bean properties of the respective Action class. Finally, the same Action instance that handles the request is used for evaluation and form population in the view. Thus, reference data needs to be modeled as bean properties of the Action too. These are (arguably) too many roles for one object.

Spring's view resolution is extremely flexible. A Controller implementation can even write a view directly to the response (by returning null for the ModelAndView). In the normal case, a ModelAndView instance consists of a view name and a model Map, which contains bean names and corresponding objects (like a command or form, containing reference data). View name resolution is highly configurable, either via bean names, via a properties file, or via your own ViewResolver implementation. The fact that the model (the M in MVC) is based on the Map interface allows for the complete abstraction of the view technology. Any renderer can be integrated directly, whether JSP, Velocity, or any other rendering technology. The model Map is simply transformed into an appropriate format, such as JSP request attributes or a Velocity template model.

**Pluggability of other MVC implementations**

There are several reasons why some projects will prefer to use other MVC implementations. Many teams expect to leverage their existing investment in skills and tools. In addition, there is a large body of knowledge and experience available for the Struts framework. Thus, if you can live with Struts' architectural flaws, it can still be a viable choice for the web layer; the same applies to WebWork and other web MVC frameworks.

If you don't 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 via its ContextLoaderListener, and access it via its ServletContext attribute (or Spring's respective helper method) from within a Struts or WebWork action. Note that there aren't any "plug-ins" involved, so no dedicated integration is necessary. From the web layer's point of view, you'll simply use Spring as a library, with the root application context instance as the entry point.

All your registered beans and all of Spring's services can be at your fingertips even without Spring's Web MVC. Spring doesn't compete with Struts or WebWork in this scenario, it just addresses the many areas that the pure web MVC frameworks don't, from bean configuration to data access and transaction handling. So you are able to 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.

### Features of Spring Web MVC

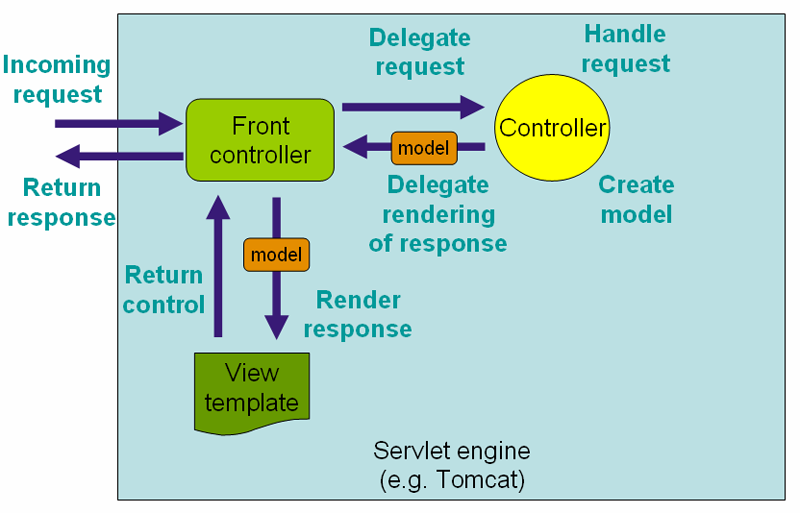
Spring's web module provides a wealth of unique web support features, including:

* Clear separation of roles - controller, validator, command object, form object, model object, DispatcherServlet, handler mapping, view resolver, etc. Each role can be fulfilled by a specialized object.
* Powerful and straightforward configuration of both framework and application classes as JavaBeans, including easy referencing across contexts, such as from web controllers to business objects and validators.
* Adaptability, non-intrusiveness. Use whatever controller subclass you need (plain, command, form, wizard, multi-action, or a custom one) for a given scenario instead of deriving from a single controller for everything.
* Reusable business code - no need for duplication. You can use existing business objects as command or form objects instead of mirroring them in order 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, etc 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. This is more flexible than some web MVC frameworks which mandate a particular technique.
* Flexible model transfer - model transfer via 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, etc.
* 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. For information on the tag library descriptor, see the appendix entitled [Appendix D, *spring.tld*](https://docs.spring.io/spring-framework/docs/2.5.x/reference/spring.tld.html)
* A JSP form tag library, introduced in Spring 2.0, that makes writing forms in JSP pages much easier. For information on the tag library descriptor, see the appendix entitled [Appendix E, *spring-form.tld*](https://docs.spring.io/spring-framework/docs/2.5.x/reference/spring-form.tld.html)
* 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. These bean scopes are described in detail in the section entitled [Section 3.4.4, “The other scopes”](https://docs.spring.io/spring-framework/docs/2.5.x/reference/beans.html#beans-factory-scopes-other)

## The DispatcherServlet

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 facilitating 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), and as such is declared in the web.xml of your web application. Requests that you want the DispatcherServlet to handle will have to be mapped using a URL mapping in the same web.xml file. This is standard J2EE servlet configuration; an example of such a DispatcherServlet declaration and mapping can be found below.

|  |
| --- |
| <web-app>  <servlet>  <servlet-name>example</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>example</servlet-name>  <url-pattern>\*.form</url-pattern>  </servlet-mapping>  </web-app> |

In the example above, all requests ending with .form will be handled by the 'example' DispatcherServlet. This is only the first step in setting up Spring Web MVC... the various beans used by the Spring Web MVC framework (over and above the DispatcherServlet itself) now need to be configured.

As detailed in the section entitled [Section 3.8, “The ApplicationContext”](https://docs.spring.io/spring-framework/docs/2.5.x/reference/beans.html#context-introduction), ApplicationContext instances in Spring can be scoped. In the web MVC framework, each DispatcherServlet has its own WebApplicationContext, which inherits all the beans already defined in the root WebApplicationContext. These inherited beans defined can be overridden in the servlet-specific scope, and new scope-specific beans can be defined local to a given servlet instance.

The framework will, on initialization of a DispatcherServlet, look for a file named *[servlet-name]-servlet.xml* in the WEB-INF directory of your web application and create the beans defined there (overriding the definitions of any beans defined with the same name in the global scope).

Consider the following DispatcherServlet servlet configuration (in the 'web.xml' file.)

|  |
| --- |
| <web-app>  <servlet>  <servlet-name>**golfing**</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>**golfing**</servlet-name>  <url-pattern>\*.do</url-pattern>  </servlet-mapping>  </web-app> |

With the above servlet configuration in place, you will need to have a file called '/WEB-INF/**golfing**-servlet.xml' in your application; this file will contain all of your Spring Web MVC-specific components (beans). The exact location of this configuration file can be changed via a servlet initialization parameter (see below for details).

The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications. It differs from a normal ApplicationContext in that it is capable of resolving themes (see [Section 13.7, “Using themes”](https://docs.spring.io/spring-framework/docs/2.5.x/reference/mvc.html#mvc-themeresolver)), and that it knows which servlet it is associated with (by having a link to the ServletContext). The WebApplicationContext is bound in the ServletContext, and by using static methods on the RequestContextUtils class you can always lookup the WebApplicationContext in case you need access to it.

The Spring DispatcherServlet has a couple of special beans it uses in order to be able to process requests and render the appropriate views. These beans are included in the Spring framework and can be configured in the WebApplicationContext, just as any other bean would be configured. Each of those beans is described in more detail below. Right now, we'll just mention them, just to let you know they exist and to enable us to go on talking about the DispatcherServlet. For most of the beans, sensible defaults are provided so you don't (initially) have to worry about configuring them.

**Special beans in the WebApplicationContext**

| **Bean type** | **Explanation** |
| --- | --- |
| Controllers | [Controllers](https://docs.spring.io/spring-framework/docs/2.5.x/reference/mvc.html#mvc-controller) are the components that form the 'C' part of the MVC. |
| Handler mappings | [Handler mappings](https://docs.spring.io/spring-framework/docs/2.5.x/reference/mvc.html#mvc-handlermapping) handle the execution of a list of pre- and post-processors and controllers that will be executed if they match certain criteria (for instance a matching URL specified with the controller) |
| View resolvers | [View resolvers](https://docs.spring.io/spring-framework/docs/2.5.x/reference/mvc.html#mvc-viewresolver) are components capable of resolving view names to views |
| Locale resolver | A [locale resolver](https://docs.spring.io/spring-framework/docs/2.5.x/reference/mvc.html#mvc-localeresolver) is a component capable of resolving the locale a client is using, in order to be able to offer internationalized views |
| Theme resolver | A [theme resolver](https://docs.spring.io/spring-framework/docs/2.5.x/reference/mvc.html#mvc-themeresolver) is capable of resolving themes your web application can use, for example, to offer personalized layouts |
| multipart file resolver | A [multipart file resolver](https://docs.spring.io/spring-framework/docs/2.5.x/reference/mvc.html#mvc-multipart) offers the functionality to process file uploads from HTML forms |
| Handler exception resolver(s) | [Handler exception resolvers](https://docs.spring.io/spring-framework/docs/2.5.x/reference/mvc.html#mvc-exceptionhandlers) offer functionality to map exceptions to views or implement other more complex exception handling code |

When a DispatcherServlet is set up for use and a request comes in for that specific DispatcherServlet, said DispatcherServlet starts processing the request. The list below describes the complete process a request goes through when handled by a DispatcherServlet:

1. The WebApplicationContext is searched for and bound in the request as an attribute in order for the controller and other elements in the process to use. It is bound by default under the key DispatcherServlet.WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE.
2. The locale resolver is bound to the request to let elements in the process resolve the locale to use when processing the request (rendering the view, preparing data, etc.) If you don't use the resolver, it won't affect anything, so if you don't need locale resolving, you don't have to use it.
3. The theme resolver is bound to the request to let elements such as views determine which theme to use. The theme resolver does not affect anything if you don't use it, so if you don't need themes you can just ignore it.
4. If a multipart resolver is specified, the request is inspected for multiparts; if multiparts are found, the request is wrapped in a MultipartHttpServletRequest for further processing by other elements in the process. (See the section entitled [Section 13.8.2, “Using the MultipartResolver”](https://docs.spring.io/spring-framework/docs/2.5.x/reference/mvc.html#mvc-multipart-resolver) for further information about multipart handling).
5. An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) will be executed in order to prepare a model (for rendering).
6. If a model is returned, the view is rendered. If no model is returned (which could be due to a pre- or postprocessor intercepting the request, for example, for security reasons), no view is rendered, since the request could already have been fulfilled.

Exceptions that are thrown during processing of the request get picked up by any of the handler exception resolvers that are declared in the WebApplicationContext. Using these exception resolvers allows you to define custom behaviors in case such exceptions get thrown.

The Spring DispatcherServlet also has support for returning the last-modification-date, as specified by the Servlet API. The process of determining the last modification date for a specific request is straightforward: the DispatcherServlet will first lookup an appropriate handler mapping and test if the handler that is found implements the interface *LastModified* interface. If so, the value of the long getLastModified(request) method of the LastModified interface is returned to the client.

You can customize Spring's DispatcherServlet by adding context parameters in the web.xml file or servlet initialization parameters. The possibilities are listed below.

**DispatcherServlet initialization parameters**

| **Parameter** | **Explanation** |
| --- | --- |
| contextClass | Class that implements WebApplicationContext, which will be used to instantiate the context used by this servlet. If this parameter isn't specified, the XmlWebApplicationContext will be used. |
| contextConfigLocation | String which is passed to the context instance (specified by contextClass) to indicate where context(s) can be found. The string is potentially split up into multiple strings (using a comma as a delimiter) to support multiple contexts (in case of multiple context locations, of beans that are defined twice, the latest takes precedence). |
| namespace | the namespace of the WebApplicationContext. Defaults to [servlet-name]-servlet. |

## Controllers

The notion of a controller is part of the MVC design pattern (more specifically, it is the 'C' in MVC). Controllers provide access to the application behavior which is typically defined by a service interface. Controllers interpret user input and transform such input into a sensible model which will be represented to the user by the view. Spring has implemented the notion of a controller in a very abstract way enabling a wide variety of different kinds of controllers to be created. Spring contains form-specific controllers, command-based controllers, and controllers that execute wizard-style logic, to name but a few.

Spring's basis for the controller architecture is the org.springframework.web.servlet.mvc.Controller interface, the source code for which is listed below.

|  |
| --- |
| public interface Controller {  /\*\*  \* Process the request and return a ModelAndView object which the DispatcherServlet  \* will render.  \*/  ModelAndView handleRequest(  HttpServletRequest request,  HttpServletResponse response) throws Exception;  } |

As you can see, the Controller interface defines a single method that is responsible for handling a request and returning an appropriate model and view. These three concepts are the basis for the Spring MVC implementation - ModelAndView and Controller. While the Controller interface is quite abstract, Spring offers a lot of Controller implementations out of the box that already contain a lot of the functionality you might need. The Controller interface just defines the most basic responsibility required of every controller; namely handling a request and returning a model and a view.

### AbstractController and WebContentGenerator

To provide a basic infrastructure, all of Spring's various Controller inherit from AbstractController, a class offering caching support and, for example, the setting of the mimetype.

**Features offered by the AbstractController**

| **Feature** | **Explanation** |
| --- | --- |
| supportedMethods | indicates what methods this controller should accept. Usually this is set to both GET and POST, but you can modify this to reflect the method you want to support. If a request is received with a method that is not supported by the controller, the client will be informed of this (expedited by the throwing of a ServletException). |
| requireSession | indicates whether or not this controller requires a HTTP session to do its work. If a session is not present when such a controller receives a request, the user is informed of this by a ServletException being thrown. |
| synchronizeOnSession | use this if you want handling by this controller to be synchronized on the user's HTTP session. |
| cacheSeconds | when you want a controller to generate a caching directive in the HTTP response, specify a positive integer here. By default the value of this property is set to *-1* so no caching directives will be included in the generated response. |
| useExpiresHeader | tweaks your controllers to specify the HTTP 1.0 compatible *"Expires"* header in the generated response. By default the value of this property is true. |
| useCacheHeader | tweaks your controllers to specify the HTTP 1.1 compatible *"Cache-Control"* header in the generated response. By default the value of this property is true. |

When using the AbstractController as the baseclass for your controllers you only have to override the handleRequestInternal(HttpServletRequest, HttpServletResponse) method, implement your logic, and return a ModelAndView object. Here is short example consisting of a class and a declaration in the web application context.

## Views and resolving them

All MVC frameworks for web applications provide a way to address views. 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 section entitled [Chapter 14, *View technologies*](https://docs.spring.io/spring-framework/docs/2.5.x/reference/view.html) has details of how to integrate and use a number of disparate view technologies.

The two interfaces which 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.

### Resolving views - the ViewResolver interface

As discussed in the section entitled [Section 13.3, “Controllers”](https://docs.spring.io/spring-framework/docs/2.5.x/reference/mvc.html#mvc-controller), all controllers in the Spring Web MVC framework return a ModelAndView instance. Views in Spring are addressed by a view name and are resolved by a view resolver. Spring comes with quite a few view resolvers. We'll list most of them and then provide a couple of examples.

**Table 13.4. View resolvers**

| **ViewResolver** | **Description** |
| --- | --- |
| AbstractCachingViewResolver | An abstract view resolver which takes care of caching views. Often views need preparation before they can be used, extending this view resolver provides caching of views. |
| XmlViewResolver | An implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring's XML bean factories. The default configuration file is /WEB-INF/views.xml. |
| ResourceBundleViewResolver | An implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle basename. The bundle is typically defined in a properties file, located in the classpath. The default file name is views.properties. |
| UrlBasedViewResolver | A simple implementation of the ViewResolver interface that effects the direct resolution of symbolic view names to URLs, without an explicit mapping definition. This is appropriate if your symbolic names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings. |
| InternalResourceViewResolver | A convenience subclass of UrlBasedViewResolver that supports InternalResourceView (i.e. Servlets and JSPs), and subclasses such as JstlView and TilesView. The view class for all views generated by this resolver can be specified via setViewClass(..). See the Javadocs for the UrlBasedViewResolver class for details. |
| VelocityViewResolver / FreeMarkerViewResolver | A convenience subclass of UrlBasedViewResolver that supports VelocityView (i.e. Velocity templates) or FreeMarkerView respectively and custom subclasses of them. |

### Redirecting to views

As has been mentioned, a controller normally returns a logical view name, which a view resolver resolves to a particular view technology. For view technologies such as JSPs that are actually processed via the Servlet/JSP engine, this is normally handled via InternalResourceViewResolver / InternalResourceView which will ultimately end up issuing an internal forward or include, via the Servlet API's RequestDispatcher.forward(..) or RequestDispatcher.include(). For other view technologies, such as Velocity, XSLT, etc., the view itself produces the content on 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 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 do a redirect before displaying the result is that this will eliminate the possibility of the user doing a double submission of form data. The browser will have sent the initial POST, will have seen a redirect back and done a subsequent GET because of that, and thus as far as it is concerned, the current page does not reflect the result of a POST, but rather of a GET, so there is no way the user can accidentally re-POST the same data by doing a refresh. The refresh would just force a GET of the result page, not a resend of the initial POST data.

#### RedirectView

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's RedirectView. In this case, DispatcherServlet will not use the normal view resolution mechanism, but rather as it has been given the (redirect) view already, will just ask it to do its work.

The RedirectView simply ends up issuing an HttpServletResponse.sendRedirect() call, which will come back to the client browser as an HTTP redirect. All model attributes are simply exposed as HTTP query parameters. This does mean that the model must contain only objects (generally Strings or convertible to Strings) which can be readily converted to a string-form HTTP query parameter.

If using RedirectView and the view is created by the controller itself, it is preferable for 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.

#### The redirect: prefix

While the use of RedirectView works fine, if the controller itself is creating the RedirectView, there is no getting around 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... it should generally think only in terms of view names that have been injected into it.

The special redirect: prefix allows this to be achieved. If a view name is returned which has the prefix redirect:, then 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 deal just 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.

#### The forward: prefix

It is also possible to use a special forward: prefix for view names that will ultimately be resolved by UrlBasedViewResolver and subclasses. All this does is create an InternalResourceView (which ultimately does a RequestDispatcher.forward()) around the rest of the view name, which is considered a URL. Therefore, there is never any use in using this prefix when using InternalResourceViewResolver / InternalResourceView anyway (for JSPs for example), but it's of potential use when you are primarily using another view technology, but still want to force a forward to happen to a resource to be handled by the Servlet/JSP engine. (Note that you may also chain multiple view resolvers, instead.)

As with the redirect: prefix, if the view name with the prefix is just injected into the controller, the controller does not have to be aware that anything special is happening in terms of handling the response.

## Using locales

Most parts of Spring's architecture support internationalization, just as the Spring web MVC framework does. DispatcherServlet enables you to automatically resolve messages using the client's locale. This is done with LocaleResolver objects.

When a request comes in, the DispatcherServlet looks for a locale resolver and if it finds one it tries to use it to set the locale. Using the RequestContext.getLocale() method, you can always retrieve the locale that was resolved by the locale resolver.

Besides the automatic locale resolution, you can also attach an interceptor to the handler mapping (see [Section 13.4.3, “Intercepting requests - the HandlerInterceptor interface”](https://docs.spring.io/spring-framework/docs/2.5.x/reference/mvc.html#mvc-handlermapping-interceptor) for more information on handler mapping interceptors), to change the locale under specific circumstances, based on a parameter in the request, for example.

Locale resolvers and interceptors are all defined in the org.springframework.web.servlet.i18n package, and are configured in your application context in the normal way. Here is a selection of the locale resolvers included in Spring.

### AcceptHeaderLocaleResolver

This locale resolver inspects the accept-language header in the request that was sent by the browser of the client. Usually this header field contains the locale of the client's operating system.

### CookieLocaleResolver

This locale resolver inspects a Cookie that might exist on the client, to see if a locale is specified. If so, it uses that specific locale. Using the properties of this locale resolver, you can specify the name of the cookie, as well as the maximum age. Find below an example of defining a CookieLocaleResolver.

|  |
| --- |
| <bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">  <property name="cookieName" value="clientlanguage"/>    *<!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->*  <property name="cookieMaxAge" value="100000">  </bean> |

## Handling exceptions

Spring provides HandlerExceptionResolvers to ease the pain of unexpected exceptions occurring while your request is being handled by a controller which matched the request. HandlerExceptionResolvers somewhat resemble the exception mappings you can define in the web application descriptor web.xml. However, they provide a more flexible way to handle exceptions. They provide information about what handler was executing when the exception was thrown. Furthermore, a programmatic way of handling exception gives you many more options for how to respond appropriately before the request is forwarded to another URL (the same end result as when using the servlet specific exception mappings).

Besides implementing the HandlerExceptionResolver interface, which is only a matter of implementing the resolveException(Exception, Handler) method and returning a ModelAndView, you may also use the SimpleMappingExceptionResolver. This resolver enables you to take the class name of any exception that might be thrown and map it to a view name. This is functionally equivalent to the exception mapping feature from the Servlet API, but it's also possible to implement more finely grained mappings of exceptions from different handlers.

### The Controller - ControllerClassNameHandlerMapping

The ControllerClassNameHandlerMapping class is a HandlerMapping implementation that uses a convention to determine the mapping between request URLs and the Controller instances that are to handle those requests.

An example; consider the following (simplistic) Controller implementation. Take especial notice of the name of the class.

|  |
| --- |
| public class **ViewShoppingCartController** implements Controller {  public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {  *// the implementation is not hugely important for this example...*  }  } |

Here is a snippet from the attendent Spring Web MVC configuration file...

|  |
| --- |
| <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>    <bean id="**viewShoppingCart**" class="x.y.z.ViewShoppingCartController">  *<!-- inject dependencies as required... -->*  </bean> |

The ControllerClassNameHandlerMapping finds all of the various handler (or Controller) beans defined in its application context and strips 'Controller' off the name to define its handler mappings.

Let's look at some more examples so that the central idea becomes immediately familiar.

* WelcomeController maps to the '/welcome\*' request URL
* HomeController maps to the '/home\*' request URL
* IndexController maps to the '/index\*' request URL
* RegisterController maps to the '/register\*' request URL
* DisplayShoppingCartController maps to the '/displayshoppingcart\*' request URL

(Notice the casing - all lowercase - in the case of camel-cased *Controller* class names.)

In the case of MultiActionController handler classes, the mappings generated are (ever so slightly) more complex, but hopefully no less understandable. Some examples (all of the Controller names in this next bit are assumed to be MultiActionController implementations).

* AdminController maps to the '/admin**/\***' request URL
* CatalogController maps to the '/catalog**/\***' request URL

If you follow the pretty standard convention of naming your Controller implementations as xxx**Controller**, then the ControllerClassNameHandlerMapping will save you the tedium of having to firstly define and then having to maintain a potentially looooong SimpleUrlHandlerMapping (or suchlike).

The ControllerClassNameHandlerMapping class extends the AbstractHandlerMapping base class so you can define HandlerInterceptor instances and everything else just like you would with many other HandlerMapping implementations.

### The Model - ModelMap (ModelAndView)

The ModelMap class is essentially a glorified Map that can make adding objects that are to be displayed in (or on) a View adhere to a common naming convention. Consider the following Controller implementation; notice that objects are added to the ModelAndView without any associated name being specified.

|  |
| --- |
| public class DisplayShoppingCartController implements Controller {  public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {    List cartItems = *// get a List of CartItem objects*  User user = *// get the User doing the shopping*    ModelAndView mav = new ModelAndView("displayShoppingCart"); *<-- the logical view name*  mav.addObject(cartItems); *<-- look ma, no name, just the object*  mav.addObject(user); *<-- and again ma!*  return mav;  }  } |

The ModelAndView class uses a ModelMap class that is a custom Map implementation that automatically generates a key for an object when an object is added to it. The strategy for determining the name for an added object is, in the case of a scalar object such as User, to use the short class name of the object's class. Find below some examples of the names that are generated for scalar objects put into a ModelMap instance.

* An x.y.User instance added will have the name 'user' generated
* An x.y.Registration instance added will have the name 'registration' generated
* An x.y.Foo instance added will have the name 'foo' generated
* A java.util.HashMap instance added will have the name 'hashMap' generated (you'll probably want to be explicit about the name in this case because 'hashMap' is less than intuitive).
* Adding null will result in an IllegalArgumentException being thrown. If the object (or objects) that you are adding could potentially be null, then you will also want to be explicit about the name).

The strategy for generating a name after adding a Set, List or array object is to peek into the collection, take the short class name of the first object in the collection, and use that with 'List' appended to the name. Some examples will make the semantics of name generation for collections clearer...

* An x.y.User[] array with one or more x.y.User elements added will have the name 'userList' generated
* An x.y.Foo[] array with one or more x.y.User elements added will have the name 'fooList' generated
* A java.util.ArrayList with one or more x.y.User elements added will have the name 'userList' generated
* A java.util.HashSet with one or more x.y.Foo elements added will have the name 'fooList' generated
* An **empty** java.util.ArrayList will not be added at all (i.e. the addObject(..) call will essentially be a no-op).

### The View - RequestToViewNameTranslator

The RequestToViewNameTranslator interface is responsible for determining a logical View name when no such logical view name is explicitly supplied. It has just one implementation, the rather cunningly named DefaultRequestToViewNameTranslator class.

The DefaultRequestToViewNameTranslator maps request URLs to logical view names in a fashion that is probably best explained by recourse to an example.

|  |
| --- |
| public class RegistrationController implements Controller {    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {  *// process the request...*  ModelAndView mav = new ModelAndView();  *// add* ***data*** *as necessary to the model...*  return mav;  *// notice that no View or logical view name has been set*  }  } |

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">  <beans>  *<!-- this bean with the well known name generates view names for us -->*  <bean id="viewNameTranslator" class="org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator"/>  <bean class="x.y.RegistrationController">  *<!-- inject dependencies as necessary -->*  </bean>    *<!-- maps request URLs to Controller names -->*  <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>  <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <property name="prefix" value="/WEB-INF/jsp/"/>  <property name="suffix" value=".jsp"/>  </bean>  </beans> |

Notice how in the implementation of the handleRequest(..) method no View or logical view name is ever set on the ModelAndView that is returned. It is the DefaultRequestToViewNameTranslator that will be tasked with generating a logical view name from the URL of the request. In the case of the above RegistrationController, which is being used in conjunction with the ControllerClassNameHandlerMapping, a request URL of 'http://localhost/registration.html' will result in a logical view name of 'registration' being generated by the DefaultRequestToViewNameTranslator. This logical view name will then be resolved into the '/WEB-INF/jsp/registration.jsp' view by the InternalResourceViewResolver bean.

## Annotation-based controller configuration

There is a current trend to favor annotations over XML files for some types of configuration data. To facilitate this, Spring is now (since 2.5) providing support for configuring the MVC framework components using annotations.

Spring 2.5 introduces an annotation-based programming model for MVC controllers, using annotations such as @RequestMapping, @RequestParam, @ModelAttribute, etc. 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 API's, although they can easily get access to Servlet or Portlet facilities if desired.

### Mapping requests with @RequestMapping

The @RequestMapping annotation is used to map URLs like '/editPet.do' onto an entire class or a particular handler method. Typically the type-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.

#### Advanced @RequestMapping options

Ant-style path patterns are supported (e.g. "/myPath/\*.do"). At the method level, relative paths (e.g. "edit.do") are supported within the primary mapping expressed at the type level.

The handler method names are taken into account for narrowing if no path was specified explicitly, according to the specified org.springframework.web.servlet.mvc.multiaction.MethodNameResolver (by default an org.springframework.web.servlet.mvc.multiaction.InternalPathMethodNameResolver). Note that this only applies in case of ambiguous annotation mappings that do not specify a path mapping explicitly. In other words, the method name is only used for narrowing among a set of matching methods; it does not constitute a primary path mapping itself.

If you have a single default method (without explicit path mapping), then all requests without a more specific mapped method found will be dispatched to it. If you have multiple such default methods, then the method name will be taken into account for choosing between them.

Path mappings can be narrowed through parameter conditions: a sequence of "myParam=myValue" style expressions, with a request only mapped if each such parameter is found to have the given value. "myParam" style expressions are also supported, with such parameters having to be present in the request (allowed to have any value). Finally, "!myParam" style expressions indicate that the specified parameter is not supposed to be present in the request.