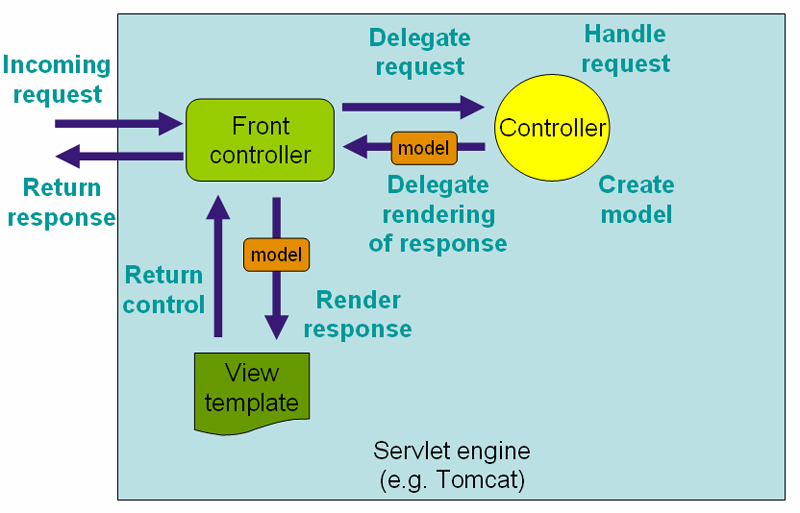
**Spring Web MVC**

**1.2 Dispatcher Servlet**



The DispatcherServlet, as any Servlet, needs to be declared and mapped according to the Servlet specification using Java configuration or in web.xml.

The DispatcherServlet uses Spring configuration to discover the delegate components it needs for request mapping, view resolution, exception handling, [and more](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-servlet-special-bean-types).

Below is an example of the Java configuration that registers and initializes the DispatcherServlet. This class is auto-detected by the Servlet container (see [Servlet Config](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-container-config)):

**public** **class** **MyWebApplicationInitializer** **implements** WebApplicationInitializer {

@Override

**public** **void** onStartup(ServletContext servletCxt) {

*// Load Spring web application configuration*

AnnotationConfigWebApplicationContext ac = **new** AnnotationConfigWebApplicationContext();

ac.register(AppConfig.class);

ac.refresh();

*// Create and register the DispatcherServlet*

DispatcherServlet servlet = **new** DispatcherServlet(ac);

ServletRegistration.Dynamic registration = servletCxt.addServlet("app", servlet);

registration.setLoadOnStartup(1);

registration.addMapping("/app/\*");

}

}

Below is an example of web.xml configuration to register and initialize the DispatcherServlet:

<web-app>

<listener>

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

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/app-context.xml</param-value>

</context-param>

<servlet>

<servlet-name>app</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value></param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>app</servlet-name>

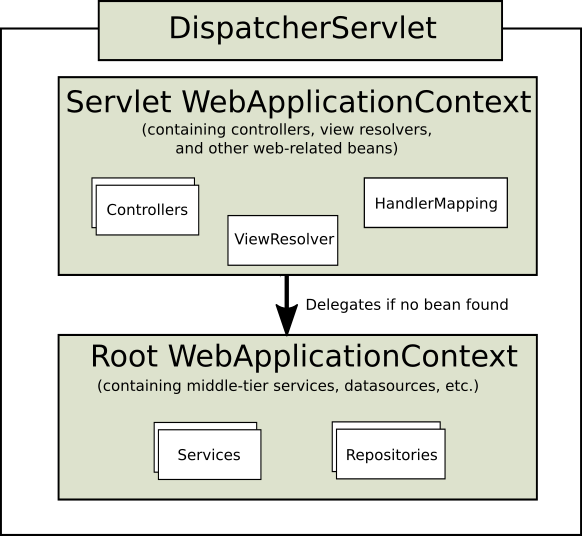
<url-pattern>/app/\*</url-pattern>

</servlet-mapping>

</web-app>

**1.2.1 Context Hierarchy**

* DispatcherServlet expects a WebApplicationContext, an extension of a plain ApplicationContext, for its own configuration.
* WebApplicationContext has a link to the ServletContext and Servlet it is associated with.
* It is also bound to the ServletContext such that applications can use static methods on RequestContextUtils to look up the WebApplicationContextif they need access to it.
* For many applications having a single WebApplicationContext is simple and sufficient.
* It is also possible to have a context hierarchy where one root WebApplicationContext is shared across multiple DispatcherServlet (or other Servlet) instances, each with its own child WebApplicationContext configuration.
* The root WebApplicationContext typically contains infrastructure beans such as data repositories and business services that need to be shared across multiple Servlet instances.
* Those beans are effectively inherited and could be overridden (i.e. re-declared) in the Servlet-specific, child WebApplicationContext which typically contains beans local to the given Servlet:



Below is example configuration with a WebApplicationContext hierarchy:

**public** **class** **MyWebAppInitializer** **extends** AbstractAnnotationConfigDispatcherServletInitializer {

@Override

**protected** Class<?>**[]** getRootConfigClasses() {

**return** **new** Class<?>**[]** { RootConfig.class };

}

@Override

**protected** Class<?>**[]** getServletConfigClasses() {

**return** **new** Class<?>**[]** { App1Config.class };

}

@Override

**protected** String**[]** getServletMappings() {

**return** **new** String**[]** { "/app1/\*" };

}

}

And the web.xml equivalent:

<web-app>

<listener>

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

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/root-context.xml</param-value>

</context-param>

<servlet>

<servlet-name>app1</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/app1-context.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>app1</servlet-name>

<url-pattern>/app1/\*</url-pattern>

</servlet-mapping>

</web-app>

**1.2.2 Special Bean Types**

The DispatcherServlet delegates to special beans to process requests and render the appropriate responses.

By "special beans" we mean Spring-managed, Object instances that implement WebFlux framework contracts.

Those usually come with built-in contracts but you can customize their properties, extend or replace them.

The table below lists the special beans detected by the DispatcherHandler:

| **Bean type** | **Explanation** |
| --- | --- |
| [HandlerMapping](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-handlermapping) | Map a request to a handler along with a list of [interceptors](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-handlermapping-interceptor) for pre- and post-processing. The mapping is based on some criteria the details of which vary by HandlerMapping implementation.  The two main HandlerMapping implementations are RequestMappingHandlerMapping which supports @RequestMapping annotated methods and SimpleUrlHandlerMapping which maintains explicit registrations of URI path patterns to handlers. |
| HandlerAdapter | Help the DispatcherServlet to invoke a handler mapped to a request regardless of how the handler is actually invoked. For example, invoking an annotated controller requires resolving annotations. The main purpose of a HandlerAdapteris to shield the DispatcherServlet from such details. |
| [HandlerExceptionResolver](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-exceptionhandlers) | Strategy to resolve exceptions possibly mapping them to handlers, or to HTML error views, or other. See [Exceptions](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-exceptionhandlers). |
| [ViewResolver](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-viewresolver) | Resolve logical String-based view names returned from a handler to an actual View to render to the response with. See [View Resolution](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-viewresolver) and [View Technologies](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-view). |
| [LocaleResolver](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-localeresolver), [LocaleContextResolver](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html" \l "mvc-timezone) | Resolve the Locale a client is using and possibly their time zone, in order to be able to offer internationalized views. See [Locale](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-localeresolver). |
| [ThemeResolver](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-themeresolver) | Resolve themes your web application can use, for example, to offer personalized layouts. See [Themes](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-themeresolver). |
| [MultipartResolver](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-multipart) | Abstraction for parsing a multi-part request (e.g. browser form file upload) with the help of some multipart parsing library. See [Multipart resolver](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-multipart). |
| [FlashMapManager](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-flash-attributes) | Store and retrieve the "input" and the "output" FlashMap that can be used to pass attributes from one request to another, usually across a redirect. See [Flash attributes](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-flash-attributes). |

**1.2.3 Web MVC Config**

Applications can declare the infrastructure beans listed in [Special Bean Types](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-servlet-special-bean-types) that are required to process requests.

The DispatcherServlet checks the WebApplicationContext for each special bean.

If there are no matching bean types, it falls back on the default types listed in [DispatcherServlet.properties](https://github.com/spring-projects/spring-framework/blob/master/spring-webmvc/src/main/resources/org/springframework/web/servlet/DispatcherServlet.properties).

In most cases the [MVC Config](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-config) is the best starting point. It declares the required beans in either Java or XML, and provides a higher level configuration callback API to customize it.

**1.2.4 Servlet Config**

In a Servlet 3.0+ environment, you have the option of configuring the Servlet container programmatically as an alternative or in combination with a web.xml file. Below is an example of registering a DispatcherServlet:

**import** org.springframework.web.WebApplicationInitializer;

**public** **class** **MyWebApplicationInitializer** **implements** WebApplicationInitializer {

@Override

**public** **void** onStartup(ServletContext container) {

XmlWebApplicationContext appContext = **new** XmlWebApplicationContext();

appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

ServletRegistration.Dynamic registration = container.addServlet("dispatcher", **new** DispatcherServlet(appContext));

registration.setLoadOnStartup(1);

registration.addMapping("/");

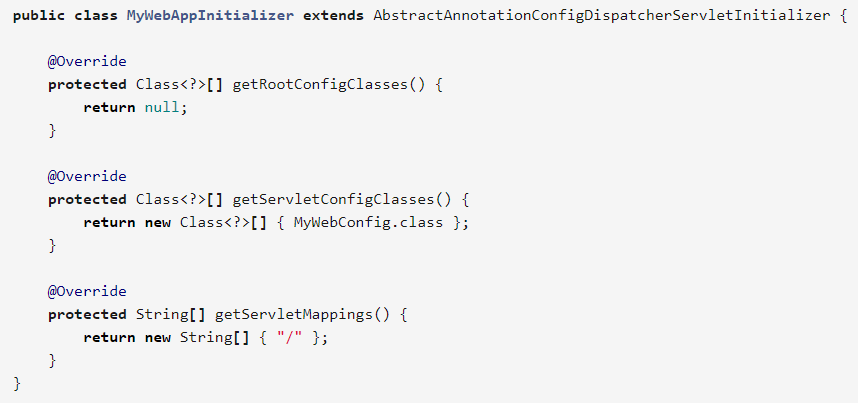
}

}

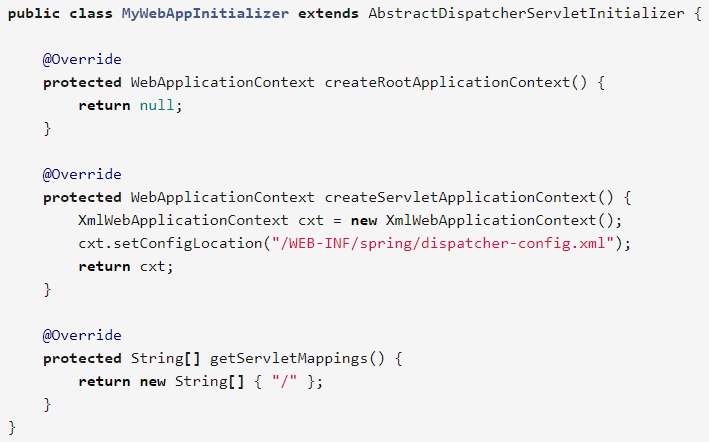
WebApplicationInitializer is an interface provided by Spring MVC that ensures your implementation is detected and automatically used to initialize any Servlet 3 container.

An abstract base class implementation of WebApplicationInitializernamed AbstractDispatcherServletInitializer makes it even easier to register the DispatcherServlet by simply overriding methods to specify the servlet mapping and the location of the DispatcherServlet configuration.

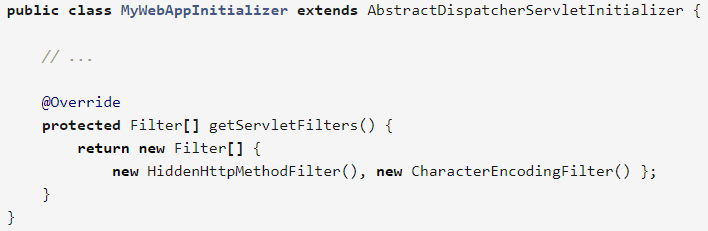
This is recommended for applications that use Java-based Spring configuration:



If using XML-based Spring configuration, you should extend directly from AbstractDispatcherServletInitializer:



AbstractDispatcherServletInitializer also provides a convenient way to add Filter instances and have them automatically mapped to the DispatcherServlet:



Each filter is added with a default name based on its concrete type and automatically mapped to the DispatcherServlet.

The isAsyncSupported protected method of AbstractDispatcherServletInitializer provides a single place to enable async support on the DispatcherServlet and all filters mapped to it. By default this flag is set to true.

**1.2.5 Processing**

The DispatcherServlet processes requests as follows:

* 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.
* 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.
* 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.
* If you specify a multipart file resolver, 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 [Multipart resolver](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-multipart) for further information about multipart handling.
* 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. Or alternatively for annotated controllers, the response may be rendered (within the HandlerAdapter) instead of returning a view.
* 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.

**1.2.6 Interception**

All HandlerMapping implementations supports handler interceptors that are useful when you want to apply specific functionality to certain requests, for example, checking for a principal. Interceptors must implement HandlerInterceptor from theorg.springframework.web.servlet package with three methods that should provide enough flexibility to do all kinds of pre-processing and post-processing:

* preHandle(..) — *before* the actual handler is executed
* postHandle(..) — *after* the handler is executed
* afterCompletion(..) — *after the complete request has finished*

The preHandle(..) method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns true, the handler execution chain will continue; when it returns false, the DispatcherServlet assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue executing the other interceptors and the actual handler in the execution chain.

Note that postHandle is less useful with @ResponseBody and ResponseEntity methods for which the response is written and committed within the HandlerAdapter and before postHandle. That means its too late to make any changes to the response such as adding an extra header. For such scenarios you can implement ResponseBodyAdvice and either declare it as an [Controller Advice](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-ann-controller-advice) bean or configure it directly on RequestMappingHandlerAdapter.

**1.2.7 Exceptions**

If an exception occurs during request mapping or is thrown from a request handler such as an @Controller, the DispatcherServlet delegates to a chain of HandlerExceptionResolver beans to resolve the exception and provide alternative handling, which typically is an error response.

The table below lists the available HandlerExceptionResolver implementations:

| *Table 2. HandlerExceptionResolver implementations* | |
| --- | --- |
| **HandlerExceptionResolver** | **Description** |
| SimpleMappingExceptionResolver | A mapping between exception class names and error view names. Useful for rendering error pages in a browser application. |
| [DefaultHandlerExceptionResolver](https://docs.spring.io/spring-framework/docs/5.0.6.RELEASE/javadoc-api/org/springframework/web/servlet/mvc/support/DefaultHandlerExceptionResolver.html) | Resolves exceptions raised by Spring MVC and maps them to HTTP status codes. Also see alternative ResponseEntityExceptionHandler and [REST API exceptions](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-ann-rest-exceptions). |
| ResponseStatusExceptionResolver | Resolves exceptions with the @ResponseStatus annotation and maps them to HTTP status codes based on the value in the annotation. |
| ExceptionHandlerExceptionResolver | Resolves exceptions by invoking an @ExceptionHandler method in an @Controlleror an @ControllerAdvice class. See [@ExceptionHandler methods](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-ann-exceptionhandler). |

Container error page

If an exception remains unresolved by any HandlerExceptionResolver and is therefore left to propagate, or if the response status is set to an error status (i.e. 4xx, 5xx), Servlet containers may render a default error page in HTML. To customize the default error page of the container, you can declare an error page mapping in web.xml:

<error-page>

<location>/error</location>

</error-page>

Given the above, when an exception bubbles up, or the response has an error status, the Servlet container makes an ERROR dispatch within the container to the configured URL (e.g. "/error"). This is then processed by the DispatcherServlet, possibly mapping it to an @Controller which could be implemented to return an error view name with a model or to render a JSON response as shown below:



**1.2.8 View Resolution**

Spring MVC defines the ViewResolver and View interfaces that enable you to render models in a browser without tying you to a specific view technology. ViewResolver provides a mapping between view names and actual views. View addresses the preparation of data before handing over to a specific view technology.

The table below provides more details on the ViewResolver hierarchy:

| **ViewResolver** | **Description** |
| --- | --- |
| AbstractCachingViewResolver | Sub-classes of AbstractCachingViewResolver cache view instances that they resolve. Caching improves performance of certain view technologies. It’s possible to turn off the cache by setting the cache property to false. Furthermore, if you must refresh a certain view at runtime (for example when a FreeMarker template is modified), you can use the removeFromCache(String viewName, Locale loc) method. |
| XmlViewResolver | 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 | Implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle base name, 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. Examples can be found in the chapter on [View Technologies](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-view). |
| UrlBasedViewResolver | Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings. |
| InternalResourceViewResolver | Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..). See the UrlBasedViewResolverjavadocs for details. |
| FreeMarkerViewResolver | Convenient subclass of UrlBasedViewResolver that supports FreeMarkerView and custom subclasses of them. |
| ContentNegotiatingViewResolver | Implementation of the ViewResolver interface that resolves a view based on the request file name or Accept header. See [Content negotiation](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-multiple-representations). |

**1.2.9 Locale**

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.

Locale resolvers and interceptors are 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.

##### TimeZone

In addition to obtaining the client’s locale, it is often useful to know their time zone. The LocaleContextResolver interface offers an extension to LocaleResolver that allows resolvers to provide a richer LocaleContext, which may include time zone information.

When available, the user’s TimeZone can be obtained using the RequestContext.getTimeZone() method. Time zone information will automatically be used by Date/Time Converter and Formatter objects registered with Spring’s ConversionService.

##### Header resolver

This locale resolver inspects the accept-language header in the request that was sent by the client (e.g., a web browser). Usually this header field contains the locale of the client’s operating system. Note that this resolver does not support time zone information.

##### Cookie resolver

This locale resolver inspects a Cookie that might exist on the client to see if a Locale or TimeZone is specified. If so, it uses the specified details. 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.

##### Session resolver

The SessionLocaleResolver allows you to retrieve Locale and TimeZone from the session that might be associated with the user’s request. In contrast to CookieLocaleResolver, this strategy stores locally chosen locale settings in the Servlet container’s HttpSession. As a consequence, those settings are just temporary for each session and therefore lost when each session terminates.