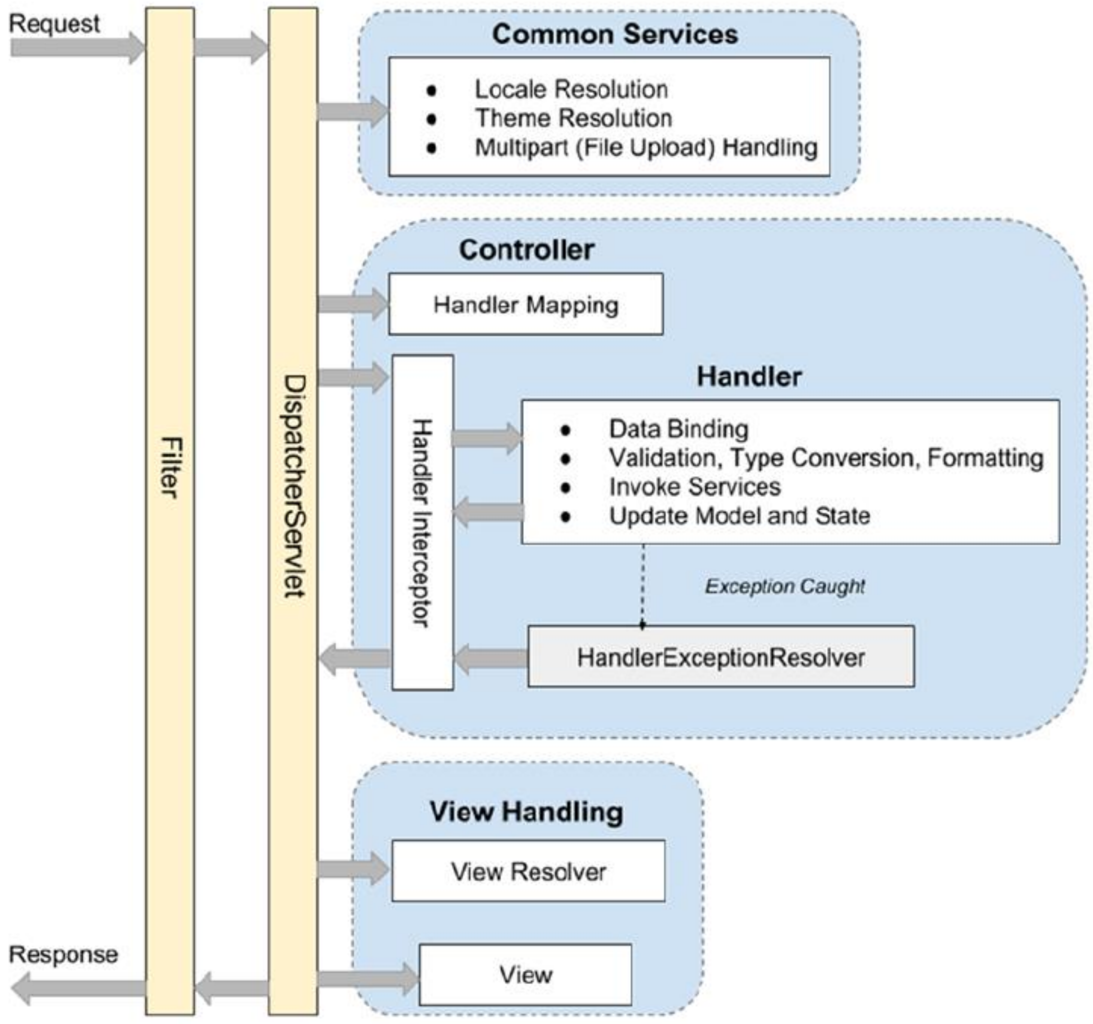
Spring6 Web MVC Configuration

Autor: Hong Le Nguyen Update : **06.2024**

Spring Web MVC Components and Request Life Cycle

The web layer is the top layer of an application, and its main function is to translate user actions into commands that lower-level layers can understand and transform the lower-level results into user-understandable data. Spring provides support for development of the web layer through frameworks like Spring Web MVC.

Let's see how Spring MVC handles a request. Figure shows *the main components* involved in *handling a request* in Spring MVC. The main components and their purposes are as follows:



Filter: The filter applies to every request. Several commonly used filters and their purposes are described in the next section.

Dispatcher servlet: is the entry point for any Spring (servlet-based-) web application. All Http requests first reach the DispatcherServlet. The servlet analyzes the request and dispatches it to the appropriate handler, a method of a controller class, for processing.

Common services: The common services will apply to every request to provide supports including i18n, theme, and file upload. Their configuration is defined in the DispatcherServlet's WebApplicationContext.

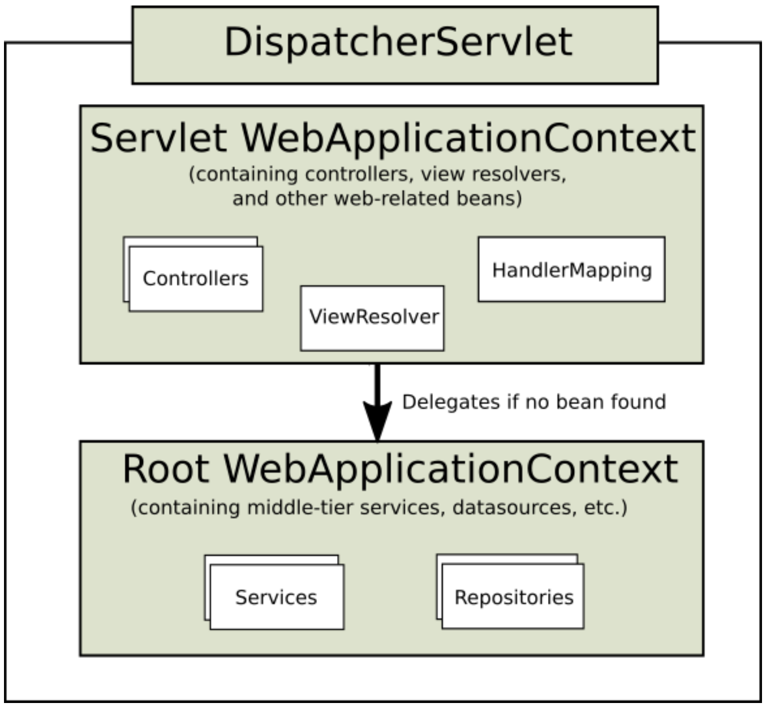
Handler mapping: This maps incoming requests to handlers (a method within a Spring MVC controller class). Since Spring 2.5, in most situations the configuration is not required because Spring MVC will automatically register a HandlerMapping implementation out of the box that maps handlers based on HTTP paths expressed through the @RequestMapping annotation at the type or method level within controller classes.

Handler interceptor: In Spring MVC, you can register interceptors for the handlers for implementing common checking or logic. For example, a handler interceptor can check to ensure that only the handlers can be invoked during office hours.

Handler exception resolver: In Spring MVC, the HandlerExceptionResolver interface (defined in package org.springframework.web.servlet) is designed to deal with unexpected exceptions thrown during request processing by handlers. By default, DispatcherServlet registers the DefaultHandlerExceptionResolver class (from package org.springframework.web.servlet.mvc.support). This resolver handles certain standard Spring MVC exceptions by setting a specific response status code. You can also implement your own exception handler by annotating a controller method with the @ExceptionHandler annotation and passing in the exception type as the attribute.

View Resolver: Spring MVC's ViewResolver interface (from package org.springframework.web.servlet) supports view resolution based on a logical name returned by the controller. There are many implementation classes to support various view-resolving mechanisms. For example, the UrlBasedViewResolver class supports direct resolution of logical names to URLs. The ContentNegotiatingViewResolver class supports dynamic resolving of views depending on the media type supported by the client (such as XML, PDF, and JSON). There also exists a number of implementations to integrate with different view technologies, such as FreeMarker (FreeMarkerViewResolver), Velocity (VelocityViewResolver), and JasperReports (JasperReportsViewResolver).

WebApplicationContext and hierarchy


DispatcherServlet
Servlet WebApplicationContext
(containing controllers, view resolvers, and other web-related beans)
Controllers
ViewResolver
HandlerMapping
Delegates if no bean found
Root WebApplicationContext
(containing middle-tier services, datasources, etc.)
Services
Repositories

The WebApplicationContext is created and injected into the DispatcherServlet before any request is made, and when the application is stopped, the Spring context is closed gracefully. The Spring components can be categorized in Spring infrastructure components, User-provided web components.

In a Spring MVC application, there can be *any number of DispatcherServlet instances* for various purposes (for example, handling user interface requests and RESTful-WS requests), and ***each*** *DispatcherServlet has its own* ***servlet-WebApplicationContext***, which defines the servlet-level characteristics.

Underneath the web layer, Spring MVC maintains *a* ***root-WebApplicationContext***, which includes the non-web configurations such as the data-source, data-access and service. *The root-WebApplicationContext will be available to all servlet-WebApplicationContexts*.

Configuring and Booting a Spring Web Application

To configure Spring MVC support for web applications, we need to perform the following configurations

Configuring the root WebApplicationContext
Configuring the servlet filters required by Spring MVC
Configuring the dispatcher servlets within the application

The Servlet 3.0+ web container supports *code-based configuration*. To use code-based configuration, *a configuration class that extends AbstractAnnotationConfigDispatcherServletInitializer* must be declared. This specialized class implements *org.springframework.web.WebApplicationInitializer* interface. Objects of types implementing this interface are *detected automatically* by SpringServletContainerInitializer, which is bootstrapped automatically by any Servlet 3.0+ environment.

The following configuration class does all three configurations in a few lines of code:

```
public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

        @Override
        protected Class<?>[] getRootConfigClasses() {
                return new Class<?>[]{ DataConfig.class, ServiceConfig.class };
        }

        @Override
        protected Class<?>[] getServletConfigClasses() {
                return new Class<?>[]{ WebMvcConfig.class, WebSecurityConfig.class };
        }

        @Override
        protected String[] getServletMappings() {
                return new String[]{"/"};
        }


        @Override
        protected Filter[] getServletFilters() {
                CharacterEncodingFilter cef = new CharacterEncodingFilter();
                cef.setEncoding("UTF-8");
                cef.setForceEncoding(true);
                return new Filter[]{new HiddenHttpMethodFilter(), cef};
        }
}
```

As seen in the previous example, the following methods were overridden to plug in customized configurations:

getRootConfigClasses():
A root application context of type AnnotationConfigWebApplicationContext will be created using the configuration classes returned by this method.

getServletConfigClasses():
A web application context of type AnnotationConfigWebApplicationContext will be created using the configuration classes returned by this method.

getServletMappings():
The DispatcherServelt's mappings (context) are specified by the array of strings returned by this method.

getServletFilters():
As the name of the methods says, this one will return an array of implementations of javax.servlet.Filter that will be applied to every request.

DispatcherServlet Configuration / WebMvcConfig

This is done by creating a configuration class that defines all infrastructure beans needed for a Spring web application.

The interface WebMvcConfigurer defines callback methods to customize the Java-based configuration for Spring MVC enabled by using @EnableWebMvc. Although there can be more than one Java-based configuration class in a Spring application, only one is allowed to be annotated with @EnableWebMvc.

```
@Configuration
@EnableWebMvc
public class WebMvcConfig implements WebMvcConfigurer {

        @Autowired
        ApplicationContext ctx;
        ...

        @Bean
        public ObjectMapper objectMapper() {
                ...
        }

        @Bean
        public MappingJackson2HttpMessageConverter mappingJackson2HttpMessageConverter() {
                return new MappingJackson2HttpMessageConverter(objectMapper());
        }
```

In the previous configuration, you can observe that several methods are overridden to customize the configuration of mappingJackson2HttpMessageConverter bean.

The proxy-filter for web security (servlet-filter in web container) enable

```
import org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;

public class SecurityWebApplicationInitializer extends AbstractSecurityWebApplicationInitializer {}
```

By providing an empty class that extends AbstractSecurityWebApplicationInitializer, you are basically telling Spring that you want ***DelegatingFilterProxy*** enabled, so springSecurityFilterChain (a bean) will be used before any other registered javax.servlet.Filter.

XML-based configuration

The DispatcherServlet must be defined in the web.xml when the application is configured using old-style XML configuration.

/src/main/webapp/WEB-INF/**web.xml**

```
<servlet>
        <servlet-name>dispatcherServlet1</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
</servlet>
```

Next, let's configure the Spring context by creating a dispatcherServlet-servlet.xml file.

/src/main/webapp/WEB-INF/**dispatcherServlet1-servlet.xml**

Spring Boot Web Mvc Auto-Configuration

Web applications can be created easily by adding the **spring-boot-starter-web** dependency to your pom.xml or build.gradle file. This dependency provides all the necessary spring-web jars and some extra ones, such as tomcat-embed* and jackson (for JSON and XML).

Spring Boot provides all the necessary auto-configuration for creating the right web infrastructure, such as configuring the DispatcherServlet, providing defaults (unless you override it), setting up an embedded Tomcat server (so you can run your application without any application containers), and more.

To override the defaults, you need to provide your own Java-configuration or parameters in the application.properties file.

For example customizing the objectMapper bean

```
package com.hong.demo.config;

@Configuration
public class HttpConverterConfig {
  public static final String DATETIME_FORMAT = "dd-MM-yyyy HH:mm";

  @Bean
  @Primary
  public ObjectMapper objectMapper() {
    DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern(DATETIME_FORMAT);
    JavaTimeModule module = new JavaTimeModule();
    module.addSerializer(new LocalDateTimeSerializer(dateTimeFormatter));
    ObjectMapper objMapper = new ObjectMapper();
    objMapper.setSerializationInclusion(JsonInclude.Include.NON_NULL);
    objMapper.registerModule(module);
    return objMapper;
  }
```