Spring6 Boot3 Configuration

written by: Hong Le Nguyen, last Update: 04.2025
Code to example: https://github.com/hong1234/testBeanConfig

1 - Spring Framework terminology

Spring container --
Spring container (a object of type ApplicationContext) creates beans, wires them together, manages their lifecycle and visibility.

Bean --
a Java object *is instantiated, managed by the Spring container*. Bean represents a reusable component that *can be wired together with other beans* to create the Spring application's functionality.

Bean wiring / dependency injection --
The Spring Container creates an instance of a bean based on request, then dependencies (reference to another bean) are injected.
Dependency injection happens at runtime, when the application is being put together after being compiled, and this allows a lot of flexibility, because the functionality of an application can be extended by modifying an external configuration without a recompile of the application.

Bean configuration --
The container requires bean definitions / configurations to create and manage beans. Each component provides the bean configuration for itself.

Bean Configuration contains the information needed for the container
- how to get Spring to use your class as a bean
- how to inject dependencies into bean
- how to inject configuration properties (parameters/externally stored values) into bean
- and more … e.g. what Bean's lifecycle details look like

There are 3 ways to provide bean configuration to the Spring Container –
- Annotation-based configuration / implicit configuration => see (2) & (4)
- Java-based configuration / explicit configuration => see (3) & (4)
- XML based configuration file
- or a mixture of them.

2 - Implicit configuration / Annotation-based configuration

@Component annotation

You annotate class with stereotype @Component or its specializations @Service, @Repository, and @Controller

```
package hong.demo.service;
@Component
public class Boy {
   private Outfit outfit;
}
```

Component scanning enable

The @ComponentScan annotation is used to specify the base packages to scan for annotated components. This annotation directs Spring to detect and register beans within the specified packages.

```
package hong.demo.config;
@Configuration
@ComponentScan(basePackages = {"hong.demo.service"})
public class AppConfig {
```

Bean wiring / dependency injection

To define a bean with dependencies, we have to decide how those dependencies are injected. Spring supports 3 types of dependency injection.
For example a bean of type Outfit named "boyDress" is injected in the property outfit of a bean type Boy. There are 2 beans/objects of type Outfit. Use @Qualifier("bean-name") annotation to select which object should be injected.

```
public class GirlDress implements Outfit {...}
public class BoyDress  implements Outfit {...}

package hong.demo.service;

@Component
public class Boy {

   //  field injection ---
   @Autowired
   @Qualifier("boyDress")
   private Outfit outfit;

   // or constructor injection ---
   private Outfit outfit;
   public Boy(@Qualifier("boyDress") Outfit outfit) {
      this.outfit = outfit;
   }

   // or setter injection ---
   @Autowired
   public void setOutfit(@Qualifier("boyDress") Outfit outfit) {
      this.outfit = outfit;
   }
```

3 - Explicit configuration / Javacode-based configuration

Although annotation-based configuration with component scanning and automatic wiring is preferable in many cases, there are times when annotation-based configuration isn't an option and you must configure explicitly.

For instance, *you want to wire components from some third-party library* into your application, you don't have the source code for that library, there's no opportunity to annotate its classes with @Component and @Autowired.

Let's assume that 2 beans should be declared from GirlDress, Girl classes of third-party package com.third.service

```
package com.third.service;

public class GirlDress implements Outfit {
   private String gdress;
   public GirlDress(String gdress){
      this.gdress = gdress;
   }
}
public class Girl {
   private Outfit outfit;
   public Girl(Outfit outfit){
      this.outfit = outfit;
   }
}
```

The @Configuration annotation is used to define configuration classes, which are sources of bean definitions for the Spring container.

Using @Bean annotation for bean definition
The girlDress() method annotated with @Bean, indicating that it returns a bean named girlDress (an instance of type Outfit) to be managed by the Spring container.

Manually wiring Bean
A bean named girlDress should be injected in the property outfit of bean type Girl named girl per constructor injection

```
package hong.demo.config;

import com.third.service.*;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class BeanConfig {
   ...
   private String gdress;

   @Bean
   public Outfit girlDress() {
      return new GirlDress(gdress);
   }

   // constructor injection ---
   @Bean
   public Girl girl(@Qualifier("girlDress") Outfit girlDress) {
      return new Girl(girlDress);
   }
```

4 - injecting configuration properties into bean

Let's assume that the value of parameter gdress, or bdress in constructor GirlDress(String gdress), or BoyDress(String bdress) is stored in file application.properties

```
src/main/resources/application.properties
   girl.dress=ROCK
   boy.dress=JEAN
```

The @PropertySource annotation in Spring
provides a declarative mechanism for loading properties from files into *the spring environment*. Properties files contain key-value pairs, e.g. app.log.level = DEBUG.

```
package hong.demo.config;
@Configuration
@ComponentScan(basePackages = {"hong.demo.service"})
// @PropertySource("classpath:api-endpoints.properties") // Multiple Configuration Files
@PropertySource("classpath:application.properties")
public class AppConfig {
}
```

Using the @Value annotation to access the value of externally stored properties in the bean

```
package hong.demo.service;
```

```
@Component("boyDress")
public class BoyDress implements Outfit {
   @Value("${boy.dress}")
   private String bdress;
   ...
}
```

Using the Spring Environment to access the externally stored properties in the bean

```
package hong.demo.config;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.env.Environment;

@Configuration
public class BeanConfig {
   @Autowired
   private Environment env;

   @Bean
   public Outfit girlDress() {
      return new GirlDress( env.getProperty("girl.dress") );
   }
   ...
}
```

5 - Using @Import annotation, configuration classes can be combined as desired

```
// @Import({BeanConfig.class,})
@Configuration
@ComponentScan(basePackages = {"hong.demo.service"})
public class AppConfig {
}
```

The @Import annotation imports the configuration from BeanConfig.class into the AppConfig class annotated with it.

6 - show all beans configured

```
import hong.demo.config.*;

public class MainRunner {
   public static void main(String[] args) {

      //  Class<?>[] configurations = new Class<?>[]{AppConfig.class};  // by using @import above
      Class<?>[] configurations = new Class<?>[]{AppConfig.class, BeanConfig.class};

      // a ApplicationContext is made with configs
      ApplicationContext context = new AnnotationConfigApplicationContext(configurations);

      for(String name: context.getBeanDefinitionNames()) {
         System.out.println(name);
      }

     // get a bean using Type
     AppService asv = context.getBean(AppService.class);
     asv.displayAllOutFits();
```

7 - Spring Profiles

provide a way to segregate parts of your application configuration and make it be available only in certain environments.

@Profile annotation

Any @Component, @Configuration or @ConfigurationProperties can be marked with @Profile to limit when it is loaded, as shown in the following example:

```
@Configuration(proxyBeanMethods = false)
@Profile("production")
public class ProductionConfiguration {
 // ...
}
```

Configuring Spring Profiles with .yml Files

Create the Common File:
This file will contain the default configuration that is common across all environments.

```
// application.yml
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: user
    password: pass
    driver-class-name: com.mysql.cj.jdbc.Driver
logging:
  level:
    root: INFO
```

Create Profile-Specific Configuration Files
These files will have the same structure as application.yml but will contain environment-specific overrides.

```
// application-dev.yml
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb_dev
logging:
  level:
    root: DEBUG

// application-prod.yml
spring:
  datasource:
    url: jdbc:mysql://prod-db-server:3306/mydb_prod
    username: prod_user
    password: prod_pass
logging:
  level:
    root: WARN
```

Activate a Profile at Runtime:

You can activate a specific profile while starting your Spring Boot application by using the --spring.profiles.active command-line argument.

```
java -jar myapp.jar --spring.profiles.active=dev
```

Alternatively, you can set the active profile in the application.yml file itself. But this approach is less flexible for deployment scenarios where profiles need to be switched without modifying configuration files.

```
spring:
  profiles:
    active: dev
```

or for properties file it will be below —

```
spring.profiles.active = dev
```

Can you use Spring properties and YAML files together?

Fortunately, developers aren't forced to choose between YAML and properties-based Spring configuration. The two formats can be used together.
If the same property is defined in both files, the YAML file loses and the traditional properties file wins.

Spring Boot 3 Autoconfiguration

written by: Hong Le Nguyen, last Update: 12.2024

1 - How Spring Boot autoconfiguration works

Autoconfiguration enable

You annotate the application entry point class with @SpringBootApplication, equivalent to declaring the @Configuration, @ComponentScan, and @EnableAutoConfiguration annotations.

The @EnableAutoConfiguration annotation enables the autoconfiguration of Spring ApplicationContext by scanning the classpath components, detecting auto-configuration classes and registering the beans that match various conditions.

Autoconfiguration process

Spring Boot reads *org.springframework.boot.autoconfigure.AutoConfiguration.imports files* from all jars in the classpath, gathering a list of auto-configuration classes. Each auto-configuration class can have multiple conditional annotations.

If conditions are met, Spring Boot executes the auto-configuration class, resulting in the creation of beans and other configurations.

2 - An (custom) autoconfiguration class

Condition annotations

Usually auto-configuration classes use @ConditionalOnClass and @ConditionalOnMissingBean annotations. This ensures that auto-configuration only applies when relevant classes are found and when you have not declared your own @Configuration.

@ConditionalOnClass({TwitterFactory.class, Twitter.class}) to specify that this autoconfiguration should take place only when the TwitterFactory.class and Twitter.class are present.

@ConditionalOnMissingBean on bean definition methods to consider this bean definition only if the TwitterFactory bean or Twitter bean is not already defined explicitly.

Locating auto-configuration candidates

Spring Boot checks for the presence of
a src/main/resources/META-INF/spring/ *org.springframework.boot.autoconfigure.AutoConfiguration.imports file* within your published jar.

ConfigurationProperties enable

The example annotated @EnableConfigurationProperties(TwitterProperties.class) to enable support for ConfigurationProperties and injected the TwitterProperties bean.

TwitterAutoConfiguration class

which contains the bean definitions that will be automatically configured based on some criteria.

```
@Configuration
@ConditionalOnClass({TwitterFactory.class, Twitter.class})
@EnableConfigurationProperties(TwitterProperties.class)
public class TwitterAutoConfiguration {
```

```
    private final TwitterProperties properties;

    @Bean
    @ConditionalOnMissingBean
    public TwitterFactory twitterFactory(){ ... }

    @Bean
    @ConditionalOnMissingBean
    public Twitter twitter(TwitterFactory twitterFactory){
      return twitterFactory.getInstance();
    }
}

@ConfigurationProperties(prefix = "twitter4j")
public class TwitterProperties {
    private String accessToken;
    ...
 }
```

When should I create and use custom auto-configurations?

Create custom auto-configurations if Spring Boot does not auto-configure a bean that is used in multiple projects in your organization and needs to be configured based on certain conditions. For simpler scenarios, @Configuration class within your application code would be the way to go.

3 - Overriding auto-configuration

3a - Overriding *bean auto-configuration*

All you need to do to override Spring Boot auto-configuration is to write explicit configuration. Spring Boot will see your configuration, step back, and let your configuration take precedence.

The auto-configuration uses Spring's conditional support (@ConditionalOnMissingBean annotation) to make runtime decisions to whether or not bean definitions should be used or ignored.

Spring Boot loads *application-level configuration before considering auto-configuration classes*.

Therefore, if you've already configured a TwitterFactory bean, then there will be a bean of type TwitterFactory by the time that auto-configuration takes place, and the auto-configured TwitterFactory bean will be ignored.

3b - Overriding *configuration properties*

The beans that are *automatically configured by Spring Boot* offer properties for fine-tuning. *When you need to adjust the settings, you can specify these properties via environment variables, Java system properties, JNDI , command-line arguments, or property files.*
There are, in fact, several ways to set properties for a Spring Boot application. Spring Boot will draw properties from several *property sources*, including the following (*):

1 Command-line arguments
2 JNDI attributes from java:comp/env
3 JVM system properties
4 Operating system environment variables
5 Randomly generated values for properties prefixed with random.* (referenced when setting other properties, such as `${random.long})
6 An application.properties or application.yml file outside of the application
7 An application.properties or application.yml file packaged inside of the application
8 Property sources specified by @PropertySource
9 Default properties

This list is *in order of precedence*. That is, any property set from a source higher in the list will override the same property set on a source lower in the list. Command-line arguments, for instance, override properties from any other property source.

4 – injecting configuration properties into beans

The Spring (Boot) environment
pulls properties from *property sources* (listed above *) and makes them available to beans in the application context. The beans that are *automatically configured by Spring Boot* are all configurable by properties drawn from the Spring environment.

Let's assume that the properties are stored in file application.yml

```
service:
   name    : 'import data'
   servicePath : '/data/import'
   poolSize: 3
```

Property values can be injected directly into your beans by using the @Value annotation, accessed through Spring Environment abstraction

```
@Component
public class MyBean {
   @Value("${service.name}")
   private String name;
   // ...
}
```

or be bound to structured object through @ConfigurationProperties

```
@Getter
@Setter
@Component
@ConfigurationProperties(prefix="service")
public class ServiceProperties {
   private  String name;
   private  String servicePath;
   private  int    poolSize;
}
```

@ConfigurationProperties Validation

```
@ConfigurationProperties(prefix="service")
@Validated
public class ServiceProperties {

   @NotNull
   private String name;

   @NotNull
   @Pattern(regexp = "\\abc$|\\xyz$")
   private  String servicePath;

   @Positive
   @Max(10)
   private  int poolSize;
   // ... getters and setters
}
```

To work with @ConfigurationProperties beans, you can inject them in the same way as any other bean, as shown in the following example:

```
@Service
public class MyService {

   private ServiceProperties properties;
   public MyService(ServiceProperties properties) {
      this.properties = properties;
   }
   // ...
}
```

Binding properties to third-party components --
As well as using @ConfigurationProperties to annotate a class, you can also use it on public @Bean methods. Doing so can be particularly useful when you want to bind properties to third-party components that are outside of your control.
To configure a bean from the Environment properties, add @ConfigurationProperties to its bean registration, as shown in the following example:

```
@ConfigurationProperties(prefix = "another")
@Bean
public AnotherComponent anotherComponent() {
   ...
}
```

5 - show all beans configured and call a service

```
package hong.demo;
import java.util.Arrays;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import hong.demo.service.AppService;

@SpringBootApplication
public class MainRunner {
   public static void main(String[] args) {
      SpringApplication.run(MainRunner.class, args);
   }

   @Bean
   public CommandLineRunner commandLineRunner(ApplicationContext ctx) {
      return args -> {
         String[] beanNames = ctx.getBeanDefinitionNames();
         Arrays.sort(beanNames);
         for (String beanName : beanNames) { System.out.println(beanName); }
         AppService asv = ctx.getBean("appService", AppService.class);
         asv.displayAllOutFits();
       };
    }
}
```

```
// make JAR     ./mvnw clean package
// run with Overriding configuration properties
java -jar target/testBeanConfig-0.0.1-SNAPSHOT.jar --service.name=testSERVICE -- service. servicePath=
```

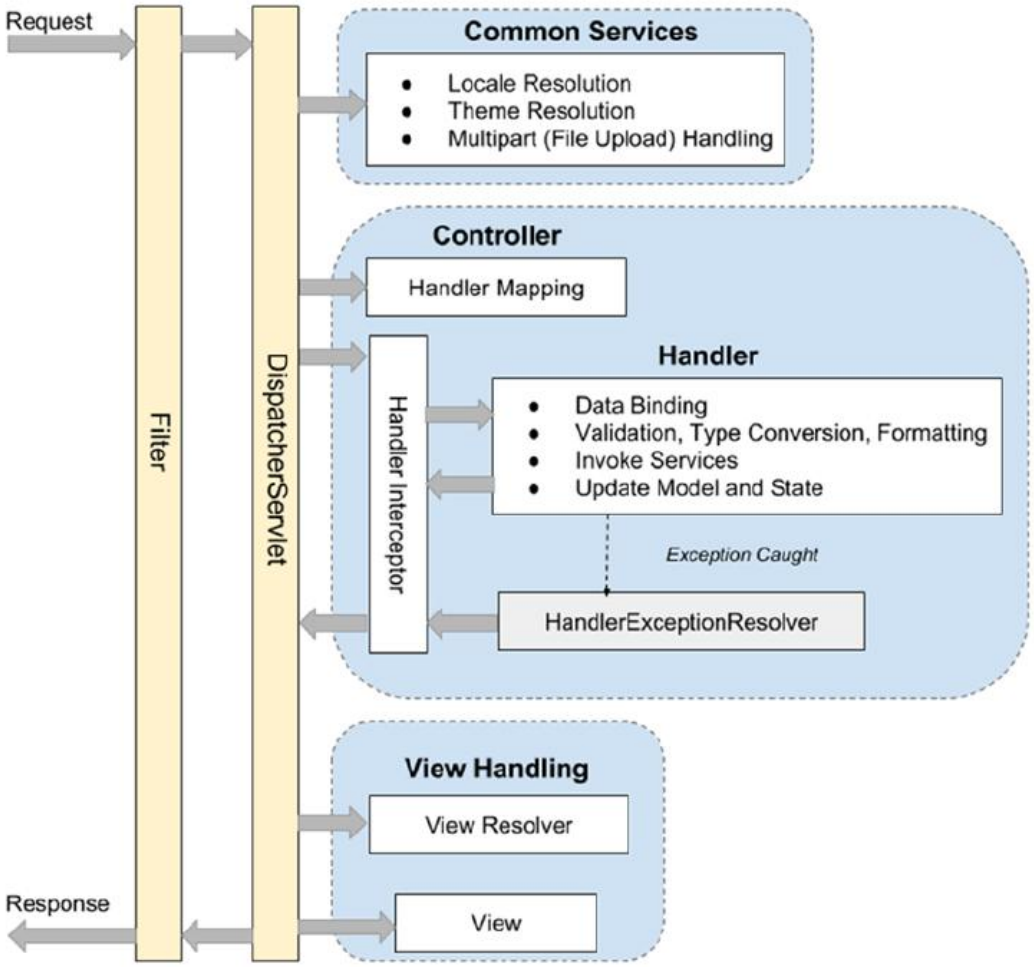
Spring6 Web MVC Configuration

Autor: Hong Le Nguyen last update : 11.2024

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: Several filters are used for purposes. Every request will visit these filters one by one before it hit DispatcherServlet.

DispatcherServlet: 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

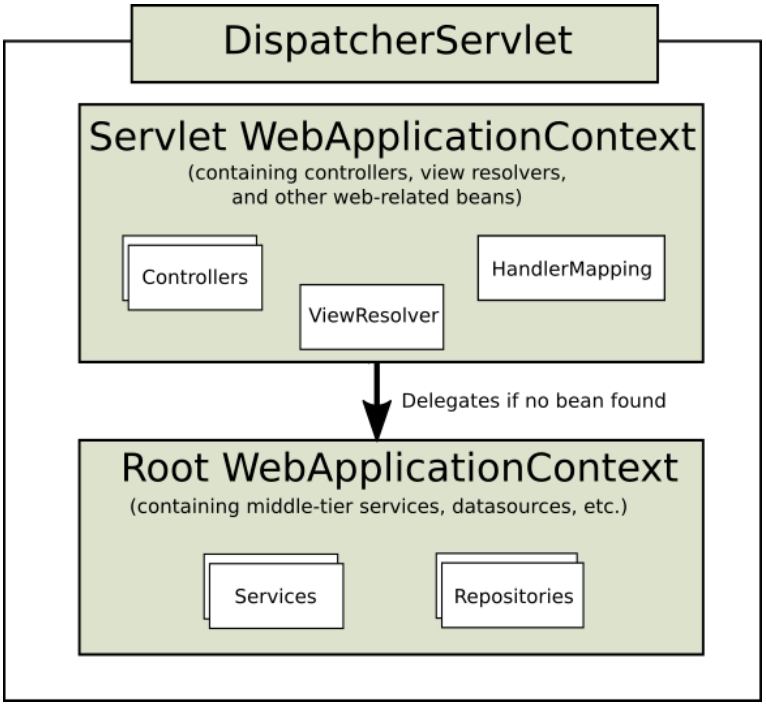
register a HandlerMapping implementation out of the box that maps handlers based on HTTP paths expressed through the @RequestMapping annotation at the type or method level within controller classes.

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



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 the bean springSecurityFilterChain will be used before any other registered jakarta.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;
  }
```