Spring Boot

Spring Beans and Dependency Injection

You are free to use any of the standard Spring Framework techniques to define your beans and their injected dependencies. We often find that using @ComponentScan (to find your beans) and using @Autowired (to do constructor injection) works well.

If you structure your code as suggested above (locating your application class in a root package), you can add @ComponentScan without any arguments. All of your application components (@Component, @Service, @Repository, @Controller etc.) are automatically registered as Spring Beans.

The following example shows a @Service Bean that uses constructor injection to obtain a required RiskAssessor bean:

```
package com.example.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

    @Autowired
    public DatabaseAccountService(RiskAssessor riskAssessor) {
        this.riskAssessor = riskAssessor;
    }

    // ...

}
```

If a bean has one constructor, you can omit the @Autowired, as shown in the following example.

```
@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

    public DatabaseAccountService(RiskAssessor riskAssessor) {
        this.riskAssessor = riskAssessor;
    }

    // ...

}
```

Using the @SpringBootApplication Annotation

Many Spring Boot developers like their apps to use auto-configuration, component scan and be able to define extra configuration on their "application class". A single @SpringBootApplication annotation can be used to enable those three features, that is:

- @EnableAutoConfiguration: enable Spring Boot’s auto-configuration mechanism
- @ComponentScan: enable @Component scan on the package where the application is located (see the best practices)
- @Configuration: allow to register extra beans in the context or import additional configuration classes

```
package com.example.myapplication;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication // same as @Configuration @EnableAutoConfiguration
@ComponentScan
public class Application {

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

}
```

To finish our application, we need to create a single Java file. By default, Maven compiles sources from src/main/java, so you need to create that directory structure and then add a file named src/main/java/Example.java to contain the following code:

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Example {

    @RequestMapping("/")
    String home() {
        return "Hello World!";
    }

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

}
```

Although there is not much code here, quite a lot is going on. We step through the important parts in the next few sections.

The @RestController and @RequestMapping Annotations

The first annotation on our Example class is @RestController. This is known as a stereotype annotation. It provides hints for people reading the code and for Spring that the class plays a specific role. In this case, our class is a web @Controller, so Spring considers it when handling incoming web requests.

The @RequestMapping annotation provides “routing” information. It tells Spring that any HTTP request with the / path should be mapped to the home method. The @RestController annotation tells Spring to render the resulting string directly back to the caller.

The @EnableAutoConfiguration Annotation

The second class-level annotation is @EnableAutoConfiguration. This annotation tells Spring Boot to “guess” how you want to configure Spring, based on the jar dependencies that you have added. Since spring-boot-starter-web added Tomcat and Spring MVC, the auto-configuration assumes that you are developing a web application and sets up Spring accordingly.

Developing Web Application

Spring Boot is well suited for web application development. You can create a self-contained HTTP server by using embedded Tomcat, Jetty, Undertow, or Netty. Most web applications use the spring boot-starter-web module to get up and running quickly. You can also choose to build reactive web applications by using the spring-boot-starter-webflux module. If you have not yet developed a Spring Boot web application, you can follow the "Hello World!" example in the Getting started section.

The “Spring Web MVC Framework”

The Spring Web MVC framework (often referred to as “Spring MVC”) is a rich “model view controller” web framework. Spring MVC lets you create special @Controller or @RestController beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP by using @RequestMapping annotations. The following code shows a typical @RestController that serves JSON data:

```
@RestController
@RequestMapping(value="/users")
public class MyRestController {

    @RequestMapping(value="/{user}", method=RequestMethod.GET)
    public User getUser(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}/customers", method=RequestMethod.GET)
    List<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}", method=RequestMethod.DELETE)
    public User deleteUser(@PathVariable Long user) {
        // ...
    }

}
```

Spring MVC is part of the core Spring Framework, and detailed information is available in the reference documentation. There are also several guides that cover Spring MVC available at spring.io/guides. Spring MVC Auto-configuration Spring Boot provides auto-configuration for Spring MVC that works well with most applications. The auto-configuration adds the following features on top of Spring's defaults:

- Inclusion of ContentNegotiatingViewResolver and BeanNameViewResolver beans.
- Support for serving static resources, including support for WebJars (covered later in this document)).
- Automatic registration of Converter, GenericConverter, and Formatter beans.
- Support for HttpMessageConverters (covered later in this document).
- Automatic registration of MessageCodesResolver (covered later in this document).
- Static index.html support.
- Automatic use of a ConfigurableWebBindingInitializer bean (covered later in this document)

If you want to keep those Spring Boot MVC customizations and make more MVC customizations (interceptors, formatters, view controllers, and other features), you can add your own @Configuration class of type WebMvcConfigurer but without @EnableWebMvc.

If you want to provide custom instances of RequestMappingHandlerMapping, RequestMappingHandlerAdapter, or ExceptionHandlerExceptionResolver, and still keep the Spring Boot MVC customizations, you can declare a bean of type WebMvcRegistrations and use it to provide custom instances of those components.

If you want to take complete control of Spring MVC, you can add your own @Configuration annotated with @EnableWebMvc, or alternatively add your own @Configuration-annotated DelegatingWebMvcConfiguration as described in the Javadoc of @EnableWebMvc

HttpMessageConverters

Spring MVC uses the HttpMessageConverter interface to convert HTTP requests and responses. Sensible defaults are included out of the box. For example, objects can be automatically converted to JSON (by using the Jackson library) or XML (by using the Jackson XML extension, if available, or by using JAXB if the Jackson XML extension is not available). By default, strings are encoded in UTF 8.

If you need to add or customize converters, you can use Spring Boot's HttpMessageConverters class, as shown in the following listing:

```
import org.springframework.boot.autoconfigure.http.HttpMessageConverters;
import org.springframework.context.annotation.*;
import org.springframework.http.converter.*;

@Configuration(proxyBeanMethods = false)
public class MyConfiguration {

    @Bean
    public HttpMessageConverters customConverters() {
        HttpMessageConverter<?> additional = ...
        HttpMessageConverter<?> another = ...
        return new HttpMessageConverters(additional, another);
    }

}
```

Any HttpMessageConverter bean that is present in the context is added to the list of converters. You can also override default converters in the same way.