**@SpringBootApplication**

Many Spring Boot developers always have their main class annotated with **@Configuration**, **@EnableAutoConfiguration** and **@ComponentScan**. Since these annotations are so frequently used together (especially if you follow the best practices above), Spring Boot provides a convenient @SpringBootApplication alternative.

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan with their default attributes

**@Controller and @RestController**

Spring 4.0 – Introduced RestController annotation

Spring RestController annotation is a convenience annotation that is itself annotated with **@Controller** and **@ResponseBody**  which eliminates the need to annotate every request handling method of the controller class with the *@ResponseBody* annotation.

**Spring MVC @Controller**

We can annotate classic controllers with the **@Controller** annotation. This is simply a specialization of the **@Component** class, which allows us to auto-detect implementation classes through the class path scanning.

We typically use **@Controller** in combination with a **@RequestMapping** annotation for request handling methods.

**Let's see a quick example of the Spring MVC controller:**

|  |
| --- |
| **@Controller**  @RequestMapping("/books")  public class SimpleBookController {  @RequestMapping("/id", produces = "application/json", method = RequestMethod.GET)  public **@ResponseBody** Book getBook( int id) {  return findBookById(id);  }  private Book findBookById(int id) {  // ...  }  } |

We annotated the request handling method with @ResponseBody. This annotation enables automatic serialization of the return object into the HttpResponse.

**Spring MVC @RestController**

@RestController is a specialized version of the controller. It includes the @Controller and @ResponseBody annotations, and as a result, simplifies the controller implementation

|  |
| --- |
| @**RestController**  @RequestMapping("books-rest")  public class SimpleBookRestController {  @GetMapping("/{id}", produces = "application/json")  public Book getBook(@PathVariable int id) {  return findBookById(id);  }  private Book findBookById(int id) {  // ...  }  } |

The controller is annotated with the **@RestController** annotation; therefore, the **@ResponseBody** isn't required.

**Every request handling method of the controller class automatically serializes return objects into HttpResponse.**

**@GetMapping** & **@RequestMapping**

There is no difference in semantic.

Specifically, @GetMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).

**RequestMapping can be used at class level:**

This annotation can be used both at the class and at the method level. In most cases, at the method level applications will prefer to use one of the HTTP method specific variants @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, or @PatchMapping.

while **GetMapping** only applies to method:

Annotation for mapping HTTP GET requests onto specific handler methods.

**@CrossOrigin**

Cross-Origin Resource Sharing (CORS) is a security concept that allows restricting the resources implemented in web browsers. It prevents the JavaScript code producing or consuming the requests against different origin.

**@Autowired**

@Autowired annotation is used for dependency injection. In spring boot application, all loaded beans are eligible for auto wiring to another bean. The annotation @Autowired in spring boot is used to auto-wire a bean into another bean.

**@RequestBody**

used to send and receive data via the REST API.

The @RequestBody annotation is applicable to handler methods of Spring controllers. This annotation indicates that Spring should deserialize a request body into an object. This object is passed as a handler method parameter.

**ResponseEntity:**

**ResponseEntity** is meant to represent the entire HTTP response. You can control anything that goes into it: status code, headers, and body.

@**ResponseBody** is a marker for the HTTP response body and @ResponseStatus declares the status code of the HTTP response.

@**ResponseStatus** isn't very flexible. It marks the entire method so you have to be sure that your handler method will always behave the same way. And you still can't set the headers. You'd need the HttpServletResponse.

Basically, ResponseEntity lets you do more.

**@Value**

Spring's @Value annotation provides a convenient way to inject property values into components. It's also quite useful to provide sensible defaults for cases where a property may not be present.

syntax

@Value("${some.key:my default value}")

private String stringWithDefaultValue;

**@PropertySource** - https://zetcode.com/spring/propertysource/

**Exception:**

**@ControllerAdvice – class Level**

@ControllerAdvice is a specialization of the @Component annotation which allows to handle exceptions across the whole application in one global handling component.

@ExceptionHandler - Method Level

@PathVariable is to obtain some placeholder from the URI (Spring call it an URI Template)

@RequestParam is to obtain a parameter from the URI as well

If the URL http://localhost:8080/MyApp/user/1234/invoices?date=12-05-2013

gets the invoices for user 1234 on December 5th, 2013, the controller method would look like:

@RequestMapping(value="/user/{userId}/invoices", method = RequestMethod.GET)

public List<Invoice> listUsersInvoices(

@PathVariable("userId") int user,

@RequestParam(value = "date", required = false) Date dateOrNull) {

...

}

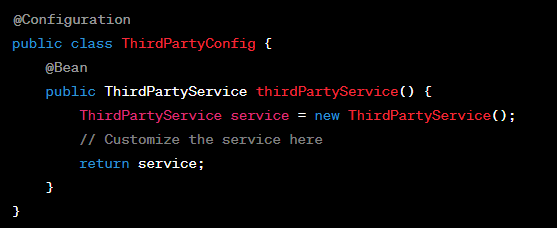
What is @configuration and @Bean

In Spring Framework, **@Configuration** and **@Bean** are annotations used for defining and configuring beans in the Spring application context.

The use of **@Configuration** and **@Bean** annotations is a fundamental part of Spring's dependency injection mechanism and provides several real-time use cases:

**Customizing Third-Party Beans**:

You can use **@Configuration** and **@Bean** to customize and configure third-party beans that you don't have control over. For example, you can create a configuration class to define a bean of a third-party library and configure it with custom settings.



In this example, we'll define two profiles: "usa" and "europe," and configure different data sources for each region. We'll also use various annotations like **@Configuration**, **@Bean**, and **@Profile**.

Define Configuration Classes:

Create configuration classes for the "usa" and "europe" profiles, each with its own data source configuration.

|  |
| --- |
| // USA DataSource Configuration  @Configuration  @Profile("usa")  public class USADataSourceConfig {  @Value("${usa.datasource.url}")  private String url;  @Value("${usa.datasource.username}")  private String username;  @Value("${usa.datasource.password}")  private String password;  @Bean  public DataSource dataSource() {  DriverManagerDataSource dataSource = new DriverManagerDataSource();  dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");  dataSource.setUrl(url);  dataSource.setUsername(username);  dataSource.setPassword(password);  return dataSource;  }  } |

|  |
| --- |
| // Europe DataSource Configuration  @Configuration  @Profile("europe")  public class EuropeDataSourceConfig {  @Value("${europe.datasource.url}")  private String url;  @Value("${europe.datasource.username}")  private String username;  @Value("${europe.datasource.password}")  private String password;  @Bean  public DataSource dataSource() {  DriverManagerDataSource dataSource = new DriverManagerDataSource();  dataSource.setDriverClassName("org.postgresql.Driver");  dataSource.setUrl(url);  dataSource.setUsername(username);  dataSource.setPassword(password);  return dataSource;  }  } |

In these configuration classes, we use the **@Profile** annotation to specify which profile each class is associated with. We also use **@Value** to read database properties from application properties files, which should be defined separately for each profile (e.g., **application-usa.properties** and **application-europe.properties**).

**‘Set up Application Properties:**

Define separate application properties files for each region (e.g., **application-usa.properties** and **application-europe.properties**) to configure database properties specific to that region.

Properties File:

|  |
| --- |
| **# application-usa.properties**  **usa.datasource.url=jdbc:mysql://localhost:3306/usa\_db**  **usa.datasource.username=root**  **usa.datasource.password=root\_password** |

|  |
| --- |
| **# application-europe.properties**  **europe.datasource.url=jdbc:postgresql://localhost:5432/europe\_db**  **europe.datasource.username=postgres**  **europe.datasource.password=postgres\_password** |

**Create a Service Using the Data Source:**

Create a service that uses the configured data source. The service should be aware of the active profile and use the appropriate data source.

|  |  |
| --- | --- |
| @Service  public class DatabaseService {  @Autowired  private DataSource dataSource;  public void performDatabaseOperation() {  // Example: Query the database using the dataSource  try (Connection connection = dataSource.getConnection()) {  String sql = "SELECT \* FROM example\_table";  try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) {  try (ResultSet resultSet = preparedStatement.executeQuery()) {  while (resultSet.next()) {  // Process each row of the result set | |
| int id = resultSet.getInt("id");  String name = resultSet.getString("name");  // Perform your business logic here  System.out.println("ID: " + id + ", Name: " + name);  }  }  }  } catch (SQLException e) {  e.printStackTrace();  // Handle any database-related exceptions here  }  }  } | |

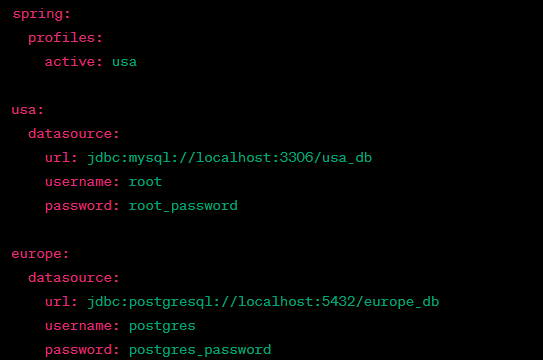
Activate Profiles:

In your **application.properties** file, specify which profile to activate. For example, to activate the "usa" profile, you can add the following line:

Alternatively, you can specify the active profile using a command-line argument or an environment variable.

Finally, use the **DatabaseService** in your application to perform database operations, and Spring will inject the appropriate data source based on the active profile.

**Yaml File:**



**@PropertySource:**

**The @PropertySource annotation is used to specify the source of external properties files and load them into the Spring Environment. It allows you to define where Spring should look for property files and makes the properties available for use within your Spring application.**

**You typically use @PropertySource when you want to externalize configuration properties, such as database connection details, API keys, or any other configuration settings, into separate property files.**

