# Spring Boot

**Overview:**

* Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".
* We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need minimal Spring configuration.
* It is a Spring module that provides the **RAD (*Rapid Application Development*)** feature to the Spring Framework. It is used to create a stand-alone Spring-based application that you can just run because it needs minimal Spring configuration.

**Features**

* Create stand-alone Spring applications
* **Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)**
* Provide opinionated 'starter' dependencies to simplify your build configuration
* Automatically configure Spring and 3rd party libraries whenever possible
* Provide production-ready features such as metrics, health checks, and externalized configuration
* **Absolutely no code generation and no requirement for XML configuration**

Diagram

Description automatically generated

**Spring vs Spring Boot**

|  |  |
| --- | --- |
| **Spring** | **Spring Boot** |
| **Spring Framework** is a widely used Java EE framework for building applications. | **Spring Boot Framework** is widely used to develop **REST APIs**. |
| It aims to simplify Java EE development that makes developers more productive. | It aims to shorten the code length and provide the easiest way to develop **Web Applications**. |
| The primary feature of the Spring Framework is **dependency injection**. | The primary feature of Spring Boot is **Autoconfiguration**. It automatically configures the classes based on the requirement. |
| It helps to make things simpler by allowing us to develop **loosely coupled** applications. | It helps to create a **stand-alone** application with less configuration. |
| The developer writes a lot of code (**boilerplate code**) to do the minimal task. | It **reduces** boilerplate code. |
| To test the Spring project, we need to set up the sever explicitly. | Spring Boot offers **embedded server** such as **Jetty** and **Tomcat**, etc. |
| It does not provide support for an in-memory database. | It offers several plugins for working with an embedded and **in-memory** database such as **H2**. |
| Developers manually define dependencies for the Spring project in **pom.xml**. | Spring Boot comes with the concept of **starter** in pom.xml file that internally takes care of downloading the dependencies **JARs** based on Spring Boot Requirement. |

**Spring Boot vs Spring MVC**

|  |  |
| --- | --- |
| **Spring Boot** | **Spring MVC** |
| **Spring Boot** is a module of Spring for packaging the Spring-based application with sensible defaults. | **Spring MVC** is a model view controller-based web framework under the Spring framework. |
| It provides default configurations to build **Spring-powered** framework. | It provides **ready to use** features for building a web application. |
| There is no need to build configuration manually. | It requires build configuration manually. |
| There is **no requirement** for a deployment descriptor. | A Deployment descriptor is **required**. |
| It avoids boilerplate code and wraps dependencies together in a single unit. | It specifies each dependency separately. |
| It **reduces** development time and increases productivity. | It takes **more** time to achieve the same. |

**Spring Beans Scope**

* singleton(default)
* Prototype
* Request
* Session
* global session

|  |  |
| --- | --- |
| **Scope** | **Description** |
| singleton | Scopes a single bean definition to a single object instance per Spring IoC container. |
| prototype | Scopes a single bean definition to any number of object instances. |
| request | Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext. |
| session | Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext. |
| global session | Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext. |

**Singleton Scope**

* When a bean is a singleton, only one shared instance of the bean will be managed, and all requests for beans with an id or ids matching that bean definition will result in that one specific bean instance being returned by the Spring container.
* To put it another way, when you define a bean definition and it is scoped as a singleton, then the Spring IoC container will create exactly one instance of the object defined by that bean definition. This single instance will be stored in a cache of such singleton beans, and all subsequent requests and references for that named bean will result in the cached object being returned.

A diagram of a computer code

Description automatically generated with medium confidence

Ways to define:

<bean id="accountService" class="com.foo.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton scope is the default); using spring-beans-2.0.dtd -->

<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>

<!-- the following is equivalent and preserved for backward compatibility in spring-beans.dtd -->

<bean id="accountService" class="com.foo.DefaultAccountService" singleton="true"/>

**Prototype Scope**

* The non-singleton, prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made (that is, it is injected into another bean or it is requested via a programmatic getBean() method call on the container). As a rule of thumb, you should use the prototype scope for all beans that are stateful, while the singleton scope should be used for stateless beans.

**Configuration related**

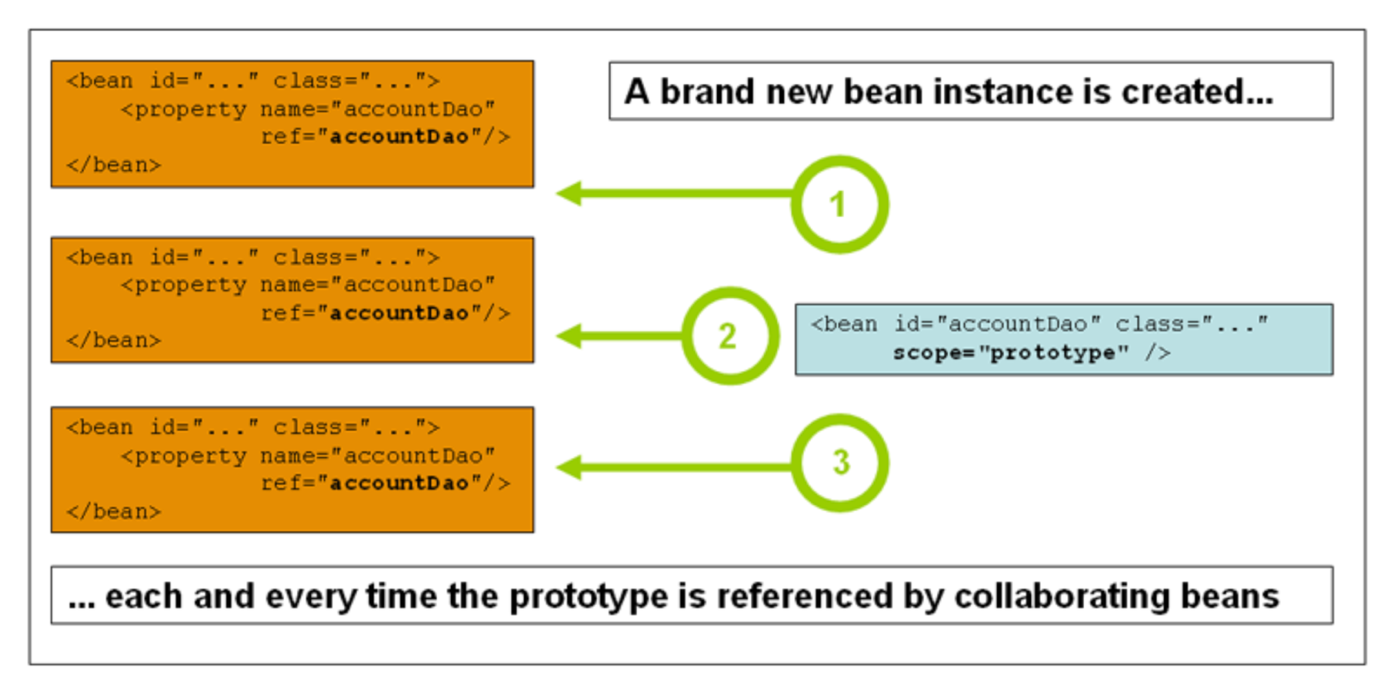
• @**Configuration** – Indicates a class declaring one or

more beans

• @**Import** – Import a configuration class

• @**ImportResource** – Import spring configuration from a resource file e.g. XML

• @**PropertySource** – Load properties from a file and map onto a POJO class



Ways to define:

<!-- using spring-beans-2.0.dtd -->

<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

<!-- the following is equivalent and preserved for backward compatibility in spring-beans.dtd -->

<bean id="accountService" class="com.foo.DefaultAccountService" singleton="false"/>

For more details refer - <https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html>

# Creating an API without Spring Boot

Add dependencies:

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>com.saliljavatrainings</groupId>

<artifactId>spring-rest-api-without-sb</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>war</packaging>

<name>Spring RestAPI without Spring Boot</name>

<properties>

<java-version>11</java-version>

<spring.version>5.2.3.RELEASE</spring.version>

<hibernate.version>5.4.1.Final</hibernate.version>

</properties>

<dependencies>

<!-- Spring Web MVC -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${spring.version}</version>

</dependency>

<!-- Required for converting JSON data to Java object and vice versa -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.9.10.1</version>

</dependency>

<!-- Servlet API -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.0.1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.17</version>

</dependency>

</dependencies>

<build>

<finalName>SpringRestJWithoutSpringBoot</finalName>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>2.3.2</version>

<configuration>

<source>${java-version}</source>

<target>${java-version}</target>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<version>3.2.3</version>

<configuration>

<warSourceDirectory>src/main/webapp</warSourceDirectory>

<warName>SpringRestJWithoutSpringBoot</warName>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

</project>

**Spring Configuration**

**SpringWebInitializer.java**

* SpringWebInitializer class extends Spring’s AbstractAnnotationConfigDispatcherServletInitializer to configure the WebApplicationContext.

**package** com.saliljavatrainings.config;

**import** org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

**public** **class** SpringWebInitializer **extends** AbstractAnnotationConfigDispatcherServletInitializer {

@Override

**protected** Class[] getServletConfigClasses() {

**return** **new** Class[] { WebConfig.**class** };

}

@Override

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

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

}

@Override

**protected** Class[] getRootConfigClasses() {

**return** **new** Class[] {};

}

}

**WebConfig.java**

* WebConfig class implements WebMvcConfigurer to configure the Jackson message converters
* @EnableWebMvc annotation is used to enable Spring MVC support
* @ComponentScan annotation is used with the @Configuration annotation to tell Spring the packages to scan for annotated components.

package com.saliljavatrainings.config;

import java.util.List;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.http.converter.HttpMessageConverter;

import org.springframework.http.converter.json.MappingJackson2HttpMessageConverter;

import org.springframework.web.servlet.config.annotation.EnableWebMvc;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@EnableWebMvc

@Configuration

@ComponentScan("com.saliljavatrainings")

public class WebConfig implements WebMvcConfigurer {

public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {

converters.add(new MappingJackson2HttpMessageConverter());

}

}

**Creating REST Controller**

* Controller class for exposing a GET REST API

import org.apache.log4j.LogManager;

import org.apache.log4j.Logger;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class WelcomeController {

private Logger logger = Logger.getLogger(WelcomeController.class);

@GetMapping("/greet/{name}")

public String greet(@PathVariable String name{

System.out.println("In the greet method");

String greet = "Hello!!! " + name + " How are You?";

logger.info(greet);

return greet;

}

}

**Log4J Configuration**

* Used to log application details to a file

# Root logger option

log4j.rootLogger=DEBUG, stdout, file

# Redirect log messages to console

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target=System.out

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

# Redirect log messages to a log file

log4j.appender.file=org.apache.log4j.RollingFileAppender

#outputs to Tomcat home

log4j.appender.file.File=/Users/sabniss/Desktop/java-training/ApplicationLogs/SpringRestJWithoutSpringBoot.log

log4j.appender.file.MaxFileSize=1KB

log4j.appender.file.MaxBackupIndex=10

log4j.appender.file.layout=org.apache.log4j.PatternLayout

log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

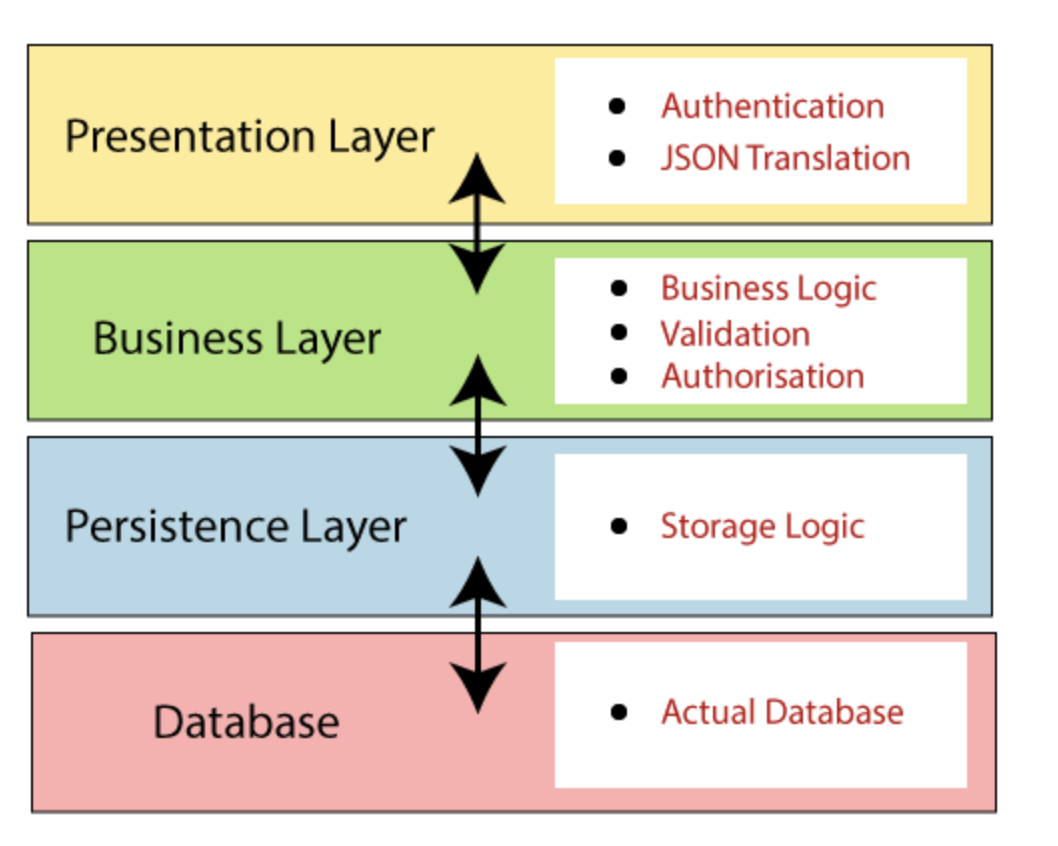
private Logger logger2 = LogManager.getLogger(WelcomeController.class);

for(int i=0;i<200;i++)

logger2.info("Writing to log file?");

**Spring Boot Architecture**

First take a look at different layers in Spring Boot

****

**Presentation Layer:**The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer. In short, it consists of views i.e., frontend part.

**Business Layer:**The business layer handles all the business logic. It consists of service classes and uses services provided by data access layers. It also performs authorization and validation.

**Persistence Layer:**The persistence layer contains all the storage logic and translates business objects from and to database rows.

**Database Layer**: In the database layer, CRUD (create, retrieve, update, delete) operations are performed.

**Spring Boot Architecture Flow:**

**Diagram

Description automatically generated**

**Spring Initializr**

* **Spring Initializr** is a **web-based tool** provided by the Pivotal Web Service. With the help of **Spring Initializr**, we can easily generate the structure of the **Spring Boot Project**. It offers extensible API for creating JVM-based projects.

**Generating the project:** Go to <https://start.spring.io/> to generate the project,

Before creating a project, we must be friendly with UI. Spring Initializr UI has the following labels:

* **Project:** It defines the **kind** of project. We can create either **Maven Project** or **Gradle Project**. We will create a **Maven Project** throughout the tutorial.
* **Language:** Spring Initializr provides the choice among three languages **Java, Kotlin,** and **Groovy**. Java is by default selected.
* **Spring Boot:** We can select the Spring Boot **version**.
* **Project Metadata:** It contains information related to the project, such as **Group**, Artifact, etc. Group denotes the **package** name; **Artifact** denotes the **Application** name. The default Group name is **com.example**, and the default Artifact name is **demo**.
* **Dependencies:** Dependencies are the collection of artifacts that we can add to our project.

There is another **Options** section that contains the following fields:

* **Name:** It is the same as **Artifact**.
* **Description:** In the description field, we can write a **description** of the project.
* **Package Name:** It is also similar to the **Group** name.
* **Packaging:** We can select the **packing** of the project. We can choose either **Jar** or **War**.
* **Java:** We can select the **JVM** version which we want to use.

# Important Annotations

**Spring Annotations** allows us to configure dependencies and implement dependency injection through java programs.

**Annotations to inject**

* **@Autowired** - Inject an object
* **@Value** – Inject a property value

**Stereotype annotations**

* **@Component** – Generic stereotype
* **@Service** – For bean in service layer
* **@Repository** – For bean in database access layer
* **@Controller** – For bean in Controller layer

**@Scope**

Defines scope of bean from one of following:

* singleton(default)
* Prototype
* Request
* Session
* global session

**Others**

* @Bean – Method level annotation to create a spring bean
* @Primary – Mark a bean as primary in case of multiple autowire candidates
* @Lazy – Create bean on first access instead of startup
* @DependsOn – Make sure other beans are created before current bean
* @Qualifier – Give the name of the bean during autowiring used when there are multiple autowiring candidates
* @Profile – This is a way to segregate parts of your application configuration and make it available only in certain environment

**Spring Boot Annotations**

* Spring Boot Annotations is a form of metadata that provides data about a program.
* In other words, annotations are used to provide **supplemental** information about a program.
* It is not a part of the application that we develop. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program.

### **1. @Bean**

The “@Bean” is a method-level annotation that is analogous to XML <bean> tag. When you declare this annotation, **Java creates a bean with the method name and registers it with the BeanFactory.** The following shows how the usage of @Bean in a method statement:

@Bean

public ExampleBean exampleBean() {

return new ExampleBean();

}

### 2. **@Springbootapplication**

The “@SpringBootApplication” annotation **triggers auto-configuration and component scanning.** It combines three annotations: @Configuration, @ComponentScan, and @EnabeAutoConfiguration.

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MyClass {

public static void main(String[] args) {

SpringApplication.run(MyClass.class, args);

}

}

### 3. **@Configuration**

The “@Configuration” is a class-based annotation that indicates the definition of one or more Bean methods in the class. Once the Spring container encounters this annotation, it can **process these spring beans to generate bean definitions** and service requests at runtime.

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class ConfigClass {

@Bean

public MyBean mybean() {

return new MyBean();

}

}

### 4. **@ComponentScan**

You can use the “@ComponentScan” annotation with the “@Configuration” annotation to define the components you need the program to scan. There are a few arguments in this annotation. **The framework scans the current package with sub-packages if you do not specify any argument.** You can use the ‘basePackages’ argument to define the specific packages to scan.

package TestPackage;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan(basePackages = "TestPackage")

public class TestClass {

}

### 

### 5. **@EnableAutoconfiguration**

This annotation allows you to **auto-configure the program based on your requirements.** The framework decides this auto-configuration based on the jars included in the program and the classpath. For example, suppose you added the “tomcat-embedded.jar” file, then it automatically configures the [TomcatServletWebServerFactory](https://www.javaprogramto.com/2020/04/spring-boot-embeddedservletcontainercustomizer-configurableembeddedservletcontainer.html) if there is no explicit declaration for its related factory bean. Using the “exclude” and “excludeClassName” arguments.

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;

@EnableAutoConfiguration

public class TestClass {

public static void main(String[] args) {

SpringApplication.run(TestClass.class, args);

}

}

### **6. @RequestMapping**

The “@RequestMapping” Annotation is used to **map HTTP requests to REST and MVC controller methods** in Spring Boot applications. You can apply this to either class-level or method-level in a controller. Furthermore, you can declare multiple request mappings using a list of values in the annotation.

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.ResponseBody;

@Controller

public class ControllerClass {

@ResponseBody

@RequestMapping("/cart")

public String getCart() {

return "this is the cart!";

}

@ResponseBody

@RequestMapping("/catalogue")

public String getCatalogue() {

return "this is the catalogue";

}

}

### **7. @GetMapping**

The “@GetMapping” is a shortcut for the  “@RequestMapping(method = RequestMethod.GET)” annotation, which **handles the HTTP GET requests corresponding to the specified URL.** The following class uses the “@RestController” annotation to indicate it can handle web requests. The “@GetMapping” maps /hello to the hello() method

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class TestController {

@GetMapping("/hello")

public String hello() {

return "Hello Spring Boot!";

}

}

### 8. @RequestParam

The “@RequestParam” annotation enables**extracting input query parameters, form data,** etc. For example, suppose you have an endpoint /API/book which takes a query string parameter called id. Then you can specify that parameter in the following manner using the @RequestParam annotation.

@GetMapping("/api/book")

@ResponseBody

public String getBook(@RequestParam String id) {

return "Book ID: " + id;

}

### 9. **@Service**

The @Service is a class-level annotation used to **indicate that the class is a service class that defines the business logic.** For instance, the below @Service annotation indicates that BankService is a service class that offers bank services.

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

import org.springframework.stereotype.Service;

@Service

public class BankService {

private final BankInfo bankInfo;

@Autowired

public BankService(BankInfo bankInfo) {

this.bankInfo = bankInfo;

}

}

### 10. **@Component**

This component allows the framework to**automatically detect the component classes** without any need to write any explicit code. Spring framework scans classes with @component, initialize them, and injects the required dependencies.

import org.springframework.stereotype.Component;

@Component

public class TestComponentAnnotation {

public int multiply(int x, int y) {

return x \* y;

}

}

### 11. **@Repository**

The “@Repository” is a specialized version of the “ @Component” annotation. It indicates that**the class is a repository that contains data storage and other operations such as updating, deleting, searching, and retrieving data on objects.**

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import com.howtodoinjava.demo.model.BookEntity;

@Repository

public interface BookRepository extends JpaRepository<BookEntity, Long>

{

}

### 12. **@Controller**

This class-based annotation is also a specialization of the “@Component” annotation, **marking the class as a controller.** It usually combines with handler methods annotated with the @RequestMapping annotation and is used with Spring MVC applications. Spring scans these classes with this annotation and catches @RequestMapping annotations. Below is an example of its usage.

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

@Controller

public class TestController {

@RequestMapping("/hello")

public String hello()

{

return "Hello Spring!";

}

}

### 13. **@Autowired**

The “@Autowired” annotation can **auto wire bean on a constructor,  setter method, property, or methods with multiple parameters.** When you use @Autowired annotation on setter methods, you can eliminate the <property> tag in the XML configuration file.

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

public class Product {

private Product product;

@Autowired

public void setProduct( Product product ){

this.product = product;

}

public Product getProduct( ) {

return product;

}

}

### 14. **@SpringBootTest**

As the name suggests, @SpringBootTest annotation can be used on a test class that executes Spring Boot-based tests. You can use it easily for **integration testing** as it loads the full application context.

import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT)

public class DemoTest

{

@Autowired

private DemoController demoController;

@Test

public void contextLoads() throws Exception {

assertThat(demoController).isNotNull();

}

}

**@Controller and @RestController in Spring Boot**

Here is a simple example of **@Controller annotation** which is used to return a response for simple HTTP request on “/hello” path:

@Controller  
public class MyController {  
  
 @Autowired  
 private MyService myService;  
  
 @RequestMapping("/hello")  
 public String sayHello(Model model) {  
 model.addAttribute("message", myService.getHelloMessage());  
 return "hello";  
 }  
}

In this example, the MyController class is annotated with @Controller annotation and has a single method, sayHello(), which is mapped to the /hello URL.

The method takes a Model object as a parameter and adds an attribute called "message" to it. The method then returns a string "hello", which is the name of the view to be rendered.

@RestController

@RestController  
public class MyRestController {  
  
 @Autowired  
 private MyService myService;  
  
 @RequestMapping("/greeting")  
 public Greeting getGreeting() {  
 return myService.getGreeting();  
 }  
}

In this example, the MyRestController class is annotated with @RestController and has a single method, getGreeting(), which is mapped to the /greeting URL.

The method returns an object of the Greeting class, which will be **automatically converted to a JSON or XML representation** and sent as the response to the client.

It is also worth noting that the Greeting class should have getters and setters for the data that you want to be included in the response, otherwise, it will not be serialized properly or converted to JSON.

## Things to keep in mind while using @Controller and @RestController annotation in Spring Boot

Here are a few things to keep in mind while using the @Controller and @RestController annotations:

* When using the @Controller annotation, you will need to use the [@ResponseBody annotation](https://www.java67.com/2022/02/what-is-responsebody-and-requestbody-annotations-in-spring.html) to return JSON or XML data, otherwise, the response will be treated as a view name and a view resolver will try to resolve it while @ResponseBody annotation is not necessary as it is already included in the annotation.
* When using the @RestController annotation, you can also use the [@ResponseStatus annotation to set the HTTP status code of the response](https://javarevisited.blogspot.com/2021/09/how-to-return-different-http-status-from-sprnig-mvc-controller.html)and @RequestBody annotation to bind the request body to a method parameter.
* And while using the @RestController annotation, you can use the [@RequestMapping annotation](https://javarevisited.blogspot.com/2021/10/difference-between-requestmapping-and..html) to set the URL path to map the methods to.
* Another thing which is worth noting is that, If you are returning a complex object that contains child objects, make sure all the child object classes have getters and setters, otherwise, they will not be serialized properly.

## **What is RESTful API?**

* RESTful API is an interface that two computer systems use to exchange information securely over the internet. Most business applications have to communicate with other internal and third-party applications to perform various tasks.
* For example, to generate monthly payslips, your internal accounts system has to share data with your customer's banking system to automate invoicing and communicate with an internal timesheet application.
* RESTful APIs support this information exchange because they follow secure, reliable, and efficient software communication standards.

**What is an API?**

* An application programming interface (API) defines the rules that you must follow to communicate with other software systems.
* Developers expose or create APIs so that other applications can communicate with their applications programmatically.
* For example, the timesheet application exposes an API that asks for an employee's full name and a range of dates. When it receives this information, it internally processes the employee's timesheet and returns the number of hours worked in that date range.

You can think of a web API as a gateway between clients and resources on the web.

**Clients**

Clients are users who want to access information from the web. The client can be a person or a software system that uses the API. For example, developers can write programs that access weather data from a weather system. Or you can access the same data from your browser when you visit the weather website directly.

**Resources**

Resources are the information that different applications provide to their clients. Resources can be images, videos, text, numbers, or any type of data. The machine that gives the resource to the client is also called the server. Organizations use APIs to share resources and provide web services while maintaining security, control, and authentication. In addition, APIs help them to determine which clients get access to specific internal resources.

**REST**

* Representational State Transfer (REST) is a software architecture that imposes conditions on how an API should work.
* REST was initially created as a guideline to manage communication on a complex network like the internet. You can use REST-based architecture to support high-performing and reliable communication at scale.
* You can easily implement and modify it, bringing visibility and cross-platform portability to any API system.
* APIs that follow the REST architectural style are called REST APIs. Web services that implement REST architecture are called RESTful web services.

**Principles of the REST architectural style:**

* **Uniform interface**
  + The uniform interface is fundamental to the design of any RESTful webservice. It indicates that the server transfers information in a standard format. The formatted resource is called a representation in REST. This format can be different from the internal representation of the resource on the server application. For example, the server can store data as text but send it in an HTML representation format.
  + Uniform interface imposes four architectural constraints:
    - Requests should identify resources. They do so by using a uniform resource identifier.
    - Clients have enough information in the resource representation to modify or delete the resource if they want to. The server meets this condition by sending metadata that describes the resource further.
    - Clients receive information about how to process the representation further. The server achieves this by sending self-descriptive messages that contain metadata about how the client can best use them.
    - Clients receive information about all other related resources they need to complete a task. The server achieves this by sending hyperlinks in the representation so that clients can dynamically discover more resources.
* **Client-Server**
* **Statelessness**
  + In REST architecture, statelessness refers to a communication method in which the server completes every client request independently of all previous requests.
  + Clients can request resources in any order, and every request is stateless or isolated from other requests. This REST API design constraint implies that the server can completely understand and fulfill the request every time.
* **Layered system**
  + In a layered system architecture, the client can connect to other authorized intermediaries between the client and server, and it will still receive responses from the server. Servers can also pass on requests to other servers.
  + You can design your RESTful web service to run on several servers with multiple layers such as security, application, and business logic, working together to fulfill client requests. These layers remain invisible to the client.
* **Cacheability**
  + RESTful web services support caching, which is the process of storing some responses on the client or on an intermediary to improve server response time.
  + For example, suppose that you visit a website that has common header and footer images on every page. Every time you visit a new website page, the server must resend the same images.
  + To avoid this, the client caches or stores these images after the first response and then uses the images directly from the cache.
  + RESTful web services control caching by using API responses that define themselves as cacheable or noncacheable.
* **Code on demand**
  + In REST architectural style, servers can temporarily extend or customize client functionality by transferring software programming code to the client.
  + For example, when you fill a registration form on any website, your browser immediately highlights any mistakes you make, such as incorrect phone numbers. It can do this because of the code sent by the server.

**Benefits of using REST**

**Scalability**

* Systems that implement REST APIs can scale efficiently because REST optimizes client-server interactions.
* Statelessness removes server load because the server does not have to retain past client request information.
* Well-managed caching partially or completely eliminates some client-server interactions. All these features support scalability without causing communication bottlenecks that reduce performance.

**Flexibility**

* RESTful web services support total client-server separation. They simplify and decouple various server components so that each part can evolve independently.
* Platform or technology changes at the server application do not affect the client application.
* The ability to layer application functions increases flexibility even further. For example, developers can make changes to the database layer without rewriting the application logic.

**Independence**

* REST APIs are independent of the technology used. You can write both client and server applications in various programming languages without affecting the API design.
* You can also change the underlying technology on either side without affecting the communication.

**How do RESTful APIs work?**

* The basic function of a RESTful API is the same as browsing the internet. The client contacts the server by using the API when it requires a resource.
* API developers explain how the client should use the REST API in the server application API documentation.
* These are the general steps for any REST API call:
  + The client sends a request to the server. The client follows the API documentation to format the request in a way that the server understands.
  + The server authenticates the client and confirms that the client has the right to make that request.
  + The server receives the request and processes it internally.
  + The server returns a response to the client. The response contains information that tells the client whether the request was successful. The response also includes any information that the client requested.

The REST API request and response details vary slightly depending on how the API developers design the API.

**What does the RESTful API client request contain?**

RESTful APIs require requests to contain the following main components:

**Unique resource identifier**

* The server identifies each resource with unique resource identifiers. For REST services, the server typically performs resource identification by using a Uniform Resource Locator (URL).
* The URL specifies the path to the resource. A URL is similar to the website address that you enter into your browser to visit any webpage.
* The URL is also called the request endpoint and clearly specifies to the server what the client requires.

**Method**

* Developers often implement RESTful APIs by using the Hypertext Transfer Protocol (HTTP).
* An HTTP method tells the server what it needs to do to the resource. The following are four common HTTP methods:

**GET**

* Clients use GET to access resources that are located at the specified URL on the server.
* They can cache GET requests and send parameters in the RESTful API request to instruct the server to filter data before sending.

**POST**

* Clients use POST to send data to the server.
* They include the data representation with the request. Sending the same POST request multiple times has the side effect of creating the same resource multiple times.

**PUT**

* Clients use PUT to update existing resources on the server.
* Unlike POST, sending the same PUT request multiple times in a RESTful web service gives the same result.

**DELETE**

* Clients use the DELETE request to remove the resource. A DELETE request can change the server state.
* However, if the user does not have appropriate authentication, the request fails.

**HTTP headers**

* Request headers are the metadata exchanged between the client and server.
* For instance, the request header indicates the format of the request and response, provides information about request status, and so on.

**Data**

* REST API requests might include data for the POST, PUT, and other HTTP methods to work successfully.

**Parameters**

RESTful API requests can include parameters that give the server more details about what needs to be done. The following are some different types of parameters:

* Path parameters that specify URL details.
* Query parameters that request more information about the resource.
* Cookie parameters that authenticate clients quickly.

**What are RESTful API authentication methods?**

* A RESTful web service must authenticate requests before it can send a response.
* Authentication is the process of verifying an identity.
* For example, you can prove your identity by showing an ID card or driver's license. Similarly, RESTful service clients must prove their identity to the server to establish trust.

RESTful API has four common authentication methods:

**HTTP authentication**

HTTP defines some authentication schemes that you can use directly when you are implementing REST API. The following are two of these schemes:

**Basic authentication**

In basic authentication, the client sends the user name and password in the request header. It encodes them with base64, which is an encoding technique that converts the pair into a set of 64 characters for safe transmission.

**Bearer authentication**

The term bearer authentication refers to the process of giving access control to the token bearer. The bearer token is typically an encrypted string of characters that the server generates in response to a login request. The client sends the token in the request headers to access resources.

**API keys**

API keys are another option for REST API authentication. In this approach, the server assigns a unique generated value to a first-time client. Whenever the client tries to access resources, it uses the unique API key to verify itself. API keys are less secure because the client has to transmit the key, which makes it vulnerable to network theft.

**OAuth**

OAuth combines passwords and tokens for highly secure login access to any system. The server first requests a password and then asks for an additional token to complete the authorization process. It can check the token at any time and also over time with a specific scope and longevity.

**What does the RESTful API server response contain?**

REST principles require the server response to contain the following main components:

**Status line**

The status line contains a three-digit status code that communicates request success or failure. For instance, 2XX codes indicate success, but 4XX and 5XX codes indicate errors. 3XX codes indicate URL redirection.

The following are some common status codes:

* 200: Generic success response
* 201: POST method success response
* 400: Incorrect request that the server cannot process
* 404: Resource not found

**Message body**

The response body contains the resource representation. The server selects an appropriate representation format based on what the request headers contain. Clients can request information in XML or JSON formats, which define how the data is written in plain text. For example, if the client requests the name and age of a person named John, the server returns a JSON representation as follows:

'{"name":"John", "age":30}'

**Headers**

The response also contains headers or metadata about the response. They give more context about the response and include information such as the server, encoding, date, and content type.

**Use JSON as the Format for Sending and Receiving Data**

* In the past, accepting and responding to API requests were done mostly in XML and even HTML. But these days, JSON (JavaScript Object Notation) has largely become the de-facto format for sending and receiving API data.
* This is because, with XML for example, it's often a bit of a hassle to decode and encode data – so XML isn’t widely supported by frameworks anymore.
* JavaScript, for example, has an inbuilt method to parse JSON data through the fetch API because JSON was primarily made for it.
  + JSON Examples:
    - {"name":"John"}
    - {"employees":[

{ "firstName":"John", "lastName":"Doe" },

{ "firstName":"Anna", "lastName":"Smith" },

{ "firstName":"Peter", "lastName":"Jones" }

]}

* To ensure the client interprets JSON data correctly, you should set the Content-Type type in the response header to application/json while making the request.

**Use Nouns Instead of Verbs in Endpoints**

* When you're designing a REST API, you should not use verbs in the endpoint paths. The endpoints should use nouns, signifying what each of them does.
* This is because HTTP methods such as GET, POST, PUT, PATCH, and DELETE are already in verb form for performing basic CRUD (Create, Read, Update, Delete) operations.
* GET, POST, PUT, PATCH, and DELETE are the commonest HTTP verbs. There are also others such as COPY, PURGE, LINK, UNLINK, and so on.
* So, for example, an endpoint should not look like this:

https://mysite.com/getPosts or https://mysite.com/createPost

* Instead, it should be something like this: https://mysite.com/posts
* In short, you should let the HTTP verbs handle what the endpoints do. So GET would retrieve data, POST will create data, PUT will update data, and DELETE will get rid of the data.

**Spring Security**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

Use properties:

spring.security.user.name=

spring.security.user.password=