Introduction to Spring Boot

Idea

Spring Boot is a framework built on top of the Spring framework. It provides a set of defaults and conventions, allowing developers to quickly create applications with minimal configuration.

Spring Web MVC is an MVC architecture which is used for web applications. Via annotations, it abstracts away a lot of details, thus simplifying the development.

Apache Tomcat is used to provide a web server environment in which Java code can also run. A file with `yml` extension can be created inside a `resources` directory in order to provide server settings (change listening port, for example).

Changing the logo :)

As you know, when starting the demo Spring Boot application, you see the default Spring logo.

To change the logo, you first need to create a file named `banner.txt` that contains your custom logo and then put it in the `/src/main/resources` directory (next to the `application.properties` file).

To create a stunning logo, you can use this Spring Boot Banner Generator.

Project structure

- `.gradle` -- if gradle is used
- `build.gradle` -- if gradle is used
- `src`
    - `main`
        - `java` -- source code
        - `resources`
            - `static` -- directory for web development
            - `templates` -- directory for web development
            - `application.properties` -- for environment-specific parameters
    - test
        - `java`
- `pom.xml` -- xml file that contains specifications for spring initializer
- `...`

`build.gradle`

Dependencies used by Spring Boot application are specified in the `build.gradle` file.

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

The first dependency adds the Spring Boot framework to this project, and the second brings test libraries integrated with Spring Boot. These dependencies are called **starters** and are enough for the simplest Spring Boot application. Each starter dependency is a group of related dependencies.

There are more of starters dependencies. All the starters use a similar naming pattern: `spring-boot-starter-*`, where `*` denotes a type of application. It can be, for example, `web` or `security`.

`application.properties`

Spring Boot uses the **convention-over-configuration** approach. As such, a developer only needs to specify unconventional aspects of the application, while all other aspects work by default.

To configure a Spring Boot application, there is an `application.properties` file located in `src/main/resources`. This file is empty in a newly generated project, but the application still works thanks to default implicit configurations.

The properties can also be stored in the YAML format within the `application.yml` file. We intend to add examples of it in the future.

Dependency injection

Dependency injection (DI) is a way of organizing and connecting different parts of a software application. It allows the dependencies of a class to be provided externally rather than being created or managed within the class itself.

There are three types of DI:

- **Constructor Injection** -- dependencies are provided as parameters in the constructor:

```java
public class MyClass {
    private Dependency dependency;

    public MyClass(Dependency dependency) {
        this.dependency = dependency;
    }
}
```

- **Setter Injection** -- dependencies are set through the setter methods:

```java
public class MyClass {
    private Dependency dependency;

    public void setDependency(Dependency dependency) {
        this.dependency = dependency;
    }
}
```

- **Method Injection** -- dependencies are passed as parameters to the method of the class:

```java
public class MyClass {
    ...
    public void doSomething(Dependency dependency) {
        // Use the dependency here
    }
}
```

IoC containers

IoC, which stands for **Inversion of Control**, is the principle used by frameworks. IoC means that, first, the framework calls the source code, then the source code calls the library functions.

IoC is used by Spring to implement dependency injection. Thus, objects can define the dependencies they need to run successfully. These dependencies are defined through constructor arguments, factory method arguments, or properties set on the object instance.

Spring IoC containers

For the Spring application, we need a few components -- *containers* -- to implement the required functionality.

The Spring containers manage the lifecycle of the application from start to finish. They manage various components created for the application and handle any required dependency injections.

The Spring containers can be configured through metadata. Two types of metadata are used in Spring: **XML** and **annotations** (annotations are more preferable). The XML approach involves defining class-related data in an external XML file, which can be loaded and used in the Spring application.

The annotations allow us to build objects with the required features and configurations. Such objects are called **POJO classes**.

<u>The Spring container consists of POJO classes and Metadata.</u>

POJO

The term **POJO** stands for **Plain Old Java Object**. A POJO is the most basic object type and contains no ties to frameworks. This means that POJOs are valid objects for any application.

POJOs can have properties, getters and setters, methods, but it cannot extend or implement framework-specific classes and interfaces or contain annotations.

JavaBeans

A **JavaBean** is a POJO with some additional requirements and restrictions:

- classes are required to be serializable (meas the state of the class can be converted to the byte stream)
- they need private fields and a no-argument constructor.

These classes can also be customized and configured using Spring metadata. To do this, we can add annotations. For example, the `@Bean` annotation can be added to a factory method to define its return value as a **Spring bean** (a POJO or JavaBean created and managed by the instance of the Spring IoC container).

With annotations, it is possible to add any configurations to preexisting classes without creating additional files.

`BeanFactory`

The `BeanFactory` is an interface that allows for the configuration and management of objects. It can produce container-managed objects known as *beans*, which can organize the backbone of your application. These beans look like regular Java objects, but they can be created during application startup, registered, and injected into different parts of the application by the container.

`ApplicationContext`

The `ApplicationContext` is an interface extending the `BeanFactory`. `ApplicationContext` objects provide bean configurations for the application setup. There are three main implementations that we typically see in applications:

- **`FileSystemXmlApplicationContext`** -- loads bean definitions from an XML file that is provided to the constructor through its full file path.
- **`ClassPathXmlApplicationContext`** -- loads bean definitions from an XML file that is provided as a classpath property.
- **`WebApplicationContext`** -- loads the servlet configuration from a file `web.xml`. Inside this file, configurations for each servlet are set.
- **`AnnotationConfigApplicationContext`** -- creates and annotation-based context from a `@Configuration` class (= the class contains Spring bean configurations.)

One way to create an application context:

```java
public class Application {

    public static void main(String[] args) {
        var context = new AnnotationConfigApplicationContext(Config.class);
    }
}
```

Spring beans

Spring can create all the necessary objects during the application startup and put them in a container. Then, each class can retrieve the objects it needs from this container.

These container-managed objects are known as **beans,** and they organize the backbone of your application. They look exactly like standard Java or Kotlin objects but can be created and instantiated during the application startup, and then injected into different parts of an application by the container.

❗ There is no need to use only Spring beans and ignore ordinary objects. Beans are usually used to form the backbone of your app and separate it by layers and configuration files.

Declaring beans

Beans are usually declared in classes with the `@Configuration` annotation. It is also possible to declare them in the class containing the `@SpringBootApplication` annotation.

By default, beans are singletons. But this default behavior can be changed.

By default, the name of a bean is the same as the name of the method that produces it. But this behaviour can be changed: `@Bean("aboba")` -- in this case, the name of the bean is "aboba".

To declare a bean, you need a method with the `@Bean` annotation. The result of executing this method will be a bean that is managed by the IoC container. The simple example:

```
@Configuration
public class Addresses {

    @Bean
    public String address() {
        return "Green Street, 102";
    }
}
```

Autowiring beans

Now that you have declared a bean, you can use it to create other beans that depend on it.

The Spring IoC container provides the DI mechanism that allows us to do that. A bean with a suitable type can be automatically injected into a method annotated with `@Bean`. The `@Autowired` annotation is used.

Example 1

```
@Configuration
public class Addresses {

    @Bean
    public String address() {
        return "Green Street, 102";
    }
}



public class Customer {
    private final String name;
    private final String address;

    public Customer(String name, String address) {
        this.name = name;
        this.address = address;
    }

    @Override
    public String toString() {
        return "Customer{ name='" + name + "', address='" + address + "' }";
    }
```

```java
}



@SpringBootApplication
public class DemoSpringApplication {

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

    @Bean
    public Customer customer(@Autowired String address) {
        return new Customer("Clara Foster", address);
    }

    @Bean
    public Customer showCustomer(@Autowired Customer customer) {
        System.out.println(customer);
        return customer;
    }
}
```

Spring DI injects the `address` bean into the `customer` method, and this bean can be used to construct a new object of the `Customer` class. Even if the argument had another name (e.g., `aboba`), this code would work as expected.

Also, Spring DI injects the `customer` bean into the `showCustomer` method. As a result, the line

```
Customer{name='Clara Foster', address='Green Street, 102'}
```

is written into `stdout`.

Example 2

```java
@Configuration
public class Addresses {
    @Bean
    public String address() {
        return "Green Street, 102";
    }
}


@Component
public class MyComponent {
    @Autowired
    private String address;

    ...
}
```

`@Qualifier`

When using `@Autowired`, the location of an injection point is determined only by the type of bean. But what if we have several beans of the same type and want to use a particular one?

The `@Qualifier` annotation allows to specify the name of the bean:

```java
@Bean
public String address1() {
    return "Green Street, 102";
}

@Bean
public String address2() {
    return "Apple Street, 15";
}

@Bean
public Customer customer(@Qualifier("address2") String address) {
    return new Customer("Clara Foster", address);
}
```

Without it, we get an error that several beans can be injected.

Components

A **component** is a special kind of class that can be auto-detected by Spring IoC and used for dependency injection. It is typically annotated with `@Component` or one of its specialized annotations (`@Service`, `@Repository`, or `@Controller`). Components are automatically detected and instantiated by the Spring container.

Example of the usage:

```java
@Component
public class PasswordGenerator {
    private static final String CHARACTERS = "abcdefghijklmnopqrstuvwxyz";
    private static final Random random = new Random();

    public String generate(int length) {
        StringBuilder result = new StringBuilder();

        for (int i = 0; i < length; i++) {
            int index = random.nextInt(CHARACTERS.length());
            result.append(CHARACTERS.charAt(index));
        }

        return result.toString();
    }
}
```

By default, there is only one bean for every component. This means when you have a `@Component` class and you retrieve that bean from the Spring container, you will always get the same instance of the class.

A bean created with `@Component` is a singleton by default.

Autowiring components

Lets autowire the `PasswordGenerator` class, used above, to the `Runner` class:

```java
@Component
public class Runner implements CommandLineRunner {
    private final PasswordGenerator generator;

    @Autowired
    public Runner(PasswordGenerator generator) {
        this.generator = generator;
    }

    @Override
    public void run(String... args) {
        System.out.println("A short password: " + generator.generate(5));
        System.out.println("A long password: " + generator.generate(10));
    }
}
```

When constructor injection is used, the `@Autowired` annotation can be omitted. When it comes to the field injection, the annotation is required, otherwise fields will be null:

```java
@Component
public class Runner implements CommandLineRunner {
    private final PasswordGenerator generator;

    public Runner(PasswordGenerator generator) {
        this.generator = generator;
    }

    public void run(String... args) { ... }
}
// OK. The `generator` field is initialized
```

```java
@Component
public class Runner implements CommandLineRunner {
    private final PasswordGenerator generator;

    public void run(String... args) { ... }
}
// The `generator` field is null
```

Specializations of components

As mentioned above, there are several specializations of components depending on their role in the Spring application:

- `@Component` indicates a generic Spring component.
- `@Service` indicates a business logic component but doesn't provide any additional functions.
- `@Controller` / `@RestController` indicates a component that can work in a REST web services.
- `@Repository` indicates a component that interacts with an external data storage (e.g., a database).

If your component doesn't need to communicate with a database or return an HTTP result, you can use the `@Service` annotation. Also, `@Component` can always be replaced with `@Service`.

`CommandLineRunner`

In Spring Boot, the `CommandLineRunner` interface is used to run code that needs to be executed after the Spring Boot application has started up and the Spring application context has been fully initialized.

The `CommandLineRunner` interface has a single method called `run()`, which accepts an array of `String` arguments.

When the Spring Boot application starts, any beans implementing the `CommandLineRunner` interface are identified by the Spring container, and their `run()` method is invoked.

Some cases to use `CommandLineRunner`:

1. Data initialization. You can load or initialize data into the application's database or other data sources (for example, by reading from a file).
2. If your application needs to interact with external systems or services upon startup, you can establish connections or perform necessary setup tasks.

3. Scheduling tasks. `CommandLineRunner` can be used to schedule recurring tasks using `@Scheduled` annotations or the `TaskScheduler` interface.
4. Logging. You can log important startup information or generate reports.

Example:

```
@Component
public class MyCommandLineRunner implements CommandLineRunner {

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Application started! "
                              + "Performing initialization tasks...");
        // Perform initialization logic here
    }
}
```

More about `ApplicationContext`

`BeanFactory` is a **root** interface for accessing the Spring IoC container. `ApplicationContext` is an interface that extends `BeanFactory`. Since `ApplicationContext` has more functionality, it is more preferable than `BeanFactory`.

There are two ways to create metadata (bean definitions): by using an XML configuration file or annotations. `BeanFactory` doesn't support annotation-based configuration, while `ApplicationContext` does.

An application context is created based on a configuration class, which describes what objects (beans) will be created inside the IoC container.

One way to create an application context:

```
public class Person {
   private String name;

   public Person(String name) {
       this.name = name;
   }
}



@Configuration
public class Config {

    @Bean
```

```java
    public Person personMary() {
        return new Person("Mary");
    }
}



public class DemoApplication {
    public static void main(String[] args) {
        var context = new AnnotationConfigApplicationContext(Config.class);
        System.out.println(
            Arrays.toString(context.getBeanDefinitionNames())
        );

// output: [ <several internal beans for the Spring app>, config, personMary ]
    }
}
```

❗ `@Configuration` contains the `@Component` annotation inside, which also tells `ApplicationContext` to create a bean based on the `Config` class. So, before creating the `personMary` bean, `ApplicationContext` also creates a `config` bean and places it in the IoC container.

`getBean()`

`ApplicationContext` overloads the `getBean()` methods inherited from `BeanFactory`:

- `T getBean(Class<T> requiredType)`
- `Object getBean(String name)` <-------- returns an Object class!
- `T getBean(String name, Class<T> requiredType)`

```java
context.getBean("personMary"); // returns an Object object
context.getBean("personMary", Person.class) // returns a Person object
```

`@ComponentScan`

Imagine we create beans, using `@Component` annotation. We want to put these components into the same application context as some `@Bean` objects. For the configuration class to know about the existence of the `@Component` classes, the `@ComponentScan` annotation is used.

```java
@Component
public class Book { ... }

```

```java
@Component
public class Movie { ... }



@ComponentScan
@Configuration
public class Config {

    @Bean
    public Person personMary() {
        return new Person("Mary");
    }
}



public class Application {
    public static void main(String[] args) {
        var context = new AnnotationConfigApplicationContext(Config.class);

 System.out.println(Arrays.toString(context.getBeanDefinitionNames()));
    }
}
```

`@ComponentScan` scans all the classes stored in the start package and all its sub-packages. It looks for `@Component` classes or its specializations(`@Service`, `@Controll` etc).

By default, the start package is the current one. You can change the default behavior of `@ComponentScan` and explicitly specify one or more base packages for scanning:

```java
@ComponentScan(basePackages = "packageName")
// OR
@ComponentScan(value = "packageName")
// OR
@ComponentScan("packageName")
```

Scopes of beans

Be default, the bean is singleton. Still, Spring supports six bean scopes: singleton, prototype, request, session, application, and websocket. The first two scopes can be used in any Spring application, including console-based ones, while the other four are only available in web applications and rely on HTTP-based concepts such as HTTP requests, sessions, etc.

To set up a scope, you should use the `@Scope` annotation. It can be applied to both `@Bean`-annotated method and `@Component`-annotated classes.

- **Singleton scope**

Spring bean has a singleton scope by default. With it, the container creates only one instance of a bean for the whole `ApplicationContext` and injects it into other beans when expected.

The singleton scope can also be specified explicitly:

```java
@Scope("singleton")
// OR
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
// OR
@Singleton
```

The last option is called a *shortcut annotation*.

- **Prototype scope**

  To turn a bean into a non-singleton, we can use *the prototype scope*. When we use it, the container returns a new bean every time it should be injected into a target.

  ```java
  @Scope("prototype")
  // OR
  @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
  // OR
  @Prototype
  ```

- **Request scope**

  Available only in web applications.

  Allows a bean to be created for the whole lifecycle of an HTTP request. If a request is processed by several Spring components, the request-scoped bean will be available in all these components.

  ```java
  @Scope("request")
  // OR
  @RequestScope
  ```

- **Session scope**

  Available only in web applications.

  Allows a bean to be created for the whole HTTP session, including sequences of HTTP requests connected by cookies or a session ID.

  ```java
  @Scope("session")
  // OR
  @SessionScope
  ```

- **Application scope**

  Available only in web applications.

Allows a bean to be created for several applications (`ApplicationContext`) running in the same `ServletContext`. This scope is broader than the singleton scope, which is only scoped to a single application context.

```
@Scope("application")
// OR
@ApplicationScope
```

- **Websocket scope**

  Available only in web applications.

  Allows a bean to be created for the complete life-cycle of a WebSocket session.

  ```
  @Scope("websocket")
  // OR
  @WebSocketScope
  ```

Bean lifecycle

When a Spring application is launched, the Spring Container gets started. The container is mainly responsible for managing the **lifecycle** of beans from their creation to destruction.

```
container started
⇓
dependencies injected
⇓
Bean initialized. Ready for use
⇓
bean s used by the app
⇓
Bean destroyed
```

A significant difference exists between singletons and beans annotated with `@Scope("prototype")`. Spring doesn't destroy prototypes and doesn't allow us to customize their lifecycle.

Initialization and destruction

The Spring container performs this initialization automatically, but it also allows us to customize the initialization. We may want to load resources, read a file, connect to a database, etc.

The same is about destruction. Before destroying the bean, some custom cleanup may be necessary, such as closing some connections, cleaning files, and so on.

- The methods **can** have any names and access modifiers.
- The methods **must not** have any arguments. Otherwise, an exception will be thrown.

Option 1: using annotations

When it comes to a `@Component` class, we can use `@PostConstruct` and `@PreDestroy` annotations to the methods. Spring will can methods annotated this way only once:

```java
@Component
class TechLibrary {
    private final List<String> bookTitles =
            Collections.synchronizedList(new ArrayList<>());

    @PostConstruct
    public void init() {
        bookTitles.add("Clean Code");
        bookTitles.add("The Art of Computer Programming");
        bookTitles.add("Introduction to Algorithms");
        System.out.println("The library has been initialized: " +
bookTitles);
    }

    @PreDestroy
    public void destroy() {
        bookTitles.clear();
        System.out.println("The library has been cleaned: " + bookTitles);
    }
}
```

After the bean instance is created and all its properties have been set (via dependency injection or other means), the Spring container calls the `init()` method.

When it comes to a `@Bean` method, you can specify methods inside the `@Bean` annotation:

```java
@Configuration
class Config {

    @Bean(initMethod = "init", destroyMethod = "destroy")
    public TechLibrary library() {
        return new TechLibrary();
    }
}

class TechLibrary {
    private final List<String> bookTitles =
            Collections.synchronizedList(new ArrayList<>());

    public void init() {
        bookTitles.add("Clean Code");
        bookTitles.add("The Art of Computer Programming");
        bookTitles.add("Introduction to Algorithms");
        System.out.println("The library has been initialized: " +
bookTitles);
    }

    public void destroy() {
        bookTitles.clear();
        System.out.println("The library has been cleaned: " + bookTitles);
    }
}
```

During the bean instantiation process, the Spring container checks for the presence of `initMethod` and `destroyMethod` attributes on the `@Bean` annotation. Then It looks for corresponding methods (`init` and `destroy` in this case) within the bean class (`TechLibrary` in this case).

Instead of using attributes inside `@Bean` annotation, we still can add the `@PostConstruct` and `@PreDestroy` annotations to the `init` and `destroy` methods.

Option 2: implement some interfaces

Another way to customize the initialization and destruction of beans is to implement the `InitializingBean` and `DisposableBean` interfaces and override their `afterPropertiesSet` and `destroy` methods:

```
@Component
class TechLibrary implements InitializingBean, DisposableBean {
    private final List<String> bookTitles =
            Collections.synchronizedList(new ArrayList<>());

    @Override
    public void afterPropertiesSet() throws Exception {
        bookTitles.add("Clean Code");
        bookTitles.add("The Art of Computer Programming");
        bookTitles.add("Introduction to Algorithms");
        System.out.println("The library has been initialized: " +
bookTitles);
    }

    @Override
    public void destroy() {
        bookTitles.clear();
        System.out.println("The library has been cleaned: " + bookTitles);
    }
}
```

After the bean instance is created and all its properties have been set (via dependency injection or other means), the Spring container calls the `afterPropertiesSet()` method.

Option 3: use an XML bean definition file

This is an outdated way mostly used for legacy applications. Forget it.

Post-processors

You can customize beans using the `BeanPostProcessor` interface. Some custom operations will be run before or after the bean is initialized. To use this feature, the `postProcessBeforeInitialization` or `postProcessAfterInitialization` methods need to be overriden.

Note that a `BeanPostProcessor` is executed for each bean defined in the Spring context, including the beans created by the framework. However, it is possible to keep beans from being modified using conditions.

Post-processors are an advanced concept. Unlike `@PostConstruct`, `@PreDestroy`, and other approaches for custom initialization, post-processors are used for processing multiple beans.

Logging

For our logging, we are going to use a **Logback** component. It is automatically added via `spring-boot-starter`.

The Logback component is accessible through an external interface called `slf4j`. The `Logger` class is imported from the `slf4j` library:

```java
import org.slf4j.Logger;
//...

@Component
public class Runner implements CommandLineRunner {
    private static final Logger LOGGER =
LoggerFactory.getLogger(DemoApplication.class);
    @Override
    public void run(String... args) {
        LOGGER.info("Spring Boot application was launched!");
    }
}
```

```
> 12:05:30.102 [main] INFO com.example.app.LoggingApplication - Spring Boot
application was launched!
```

The `Logger` class has several methods, each of them provides logging at different levels:

1. **Error:** used to track information when an application fails to execute successfully.
2. **Warn:** used when unexpected events happen that are not critical to the functionality of the application.
3. **Info:** used to log general information about the execution of the program.
4. **Debug:** used to help track program execution for fixing bugs or issues.
5. **Trace:** used to create more detailed and high-volume sets of information about program execution.

In the example above, the level (`INFO`) and the logger name (`LoggingApplication`) are shown.

File Output

By default, Spring Boot logs only to the console and does not write log files. If you want to write log files in addition to the console output, you need to set a `logging.file.name` or `logging.file.path` property in the `application.properties` file located in `src/main/resources/`.

- If there are both `name` and `path` properties, info from the `path` will be ignored.
- if there is only the `path` property, the default name for a log gile is `spring.log`.

Example:

```
logging.level.root=info
logging.pattern.console=%d{HH:mm} %-5level %logger{36} - %msg%n
logging.file.name=log/logfile.log
```

Also, we defined the lowest level of logging which is important for us. Here the information from `debug` and `trace` levels of logging will be ignored.

NOT EDITED PART

Spring Data JPA

JPA -- *Jakarta Persistence API* -- interface specification that maps java classes to database table. Thus, we can interact with the database without writing any SQL code.

`@Entity`

We want to have a table with students. Each customer has id (primary key) and name.

```java
package com.example.demo.student;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.SequenceGenerator;

@Entity
public class Student {

  @Id
  @SequenceGenerator(
      name = "customer_id_sequence",
      sequenceName = "customer_id_sequence"
  )
  @GeneratedValue(
      strategy = GenerationType.SEQUENCE,
      generator = "customer_id_sequence"
  )
  private final Long id;
  private final String name;

  public Student(Long id, String name) {
    this.id = id;
    this.name = name;
  }

  public Long getId() {
    return id;
  }

  public String getName() {
    return name;
  }
}
```

The code above wil generate a table called *Student* where entities are pairs `<id, name>`:

```
Hibernate:
    create sequence customer_id_sequence start with 1 increment by 50
Hibernate:
    create table student (
        id bigint not null,
        name varchar(255),
        primary key (id)
    )
```

`@RestController`

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
    @GetMapping("hello")
    public String hello() {
        return "hello world";
    }
}
```

`@RestController = @Controller + @RequestBody`. It means the class is the controller from MVC architecture and it generates an http response in a json format:

```java
//public class Student {
//  private final Long id;
//  private final String name;
//  ...
//}

@GetMapping("")
public Student generateStudent() {
    return new Student(1L, "Aboba");
}

// output (json format) = {"id":1,"name":"Aboba"}
```

The `@GetMapping(“/hello”)` tells Spring to use `hello()` method to answer requests that are sent to the `localhost:8080/hello` address. The annotation is used to handle HTTP GET requests for a specific URL mapping.

`@RequestMapping`

```java
@RestController
@RequestMapping(path = "v1/student")
public class StudentController {
  @GetMapping
  public List<Student> getStudents() {
    return List.of(
        new Student(1L, "Aboba1"),
        new Student(2L, "Aboba2")
    );
  }
}
```

`@RequestMapping(path = "v1/student")` ⇒ function `getStudents()` shows mapping on the `localhost:8080/v1/student` address.

JpaRepository

A default interface that maps Entity class to a database table. The name of the table is the same as the name of the class.

`JpaRepository` has two generic parameters. The first refers to the Entity class, the second refers to the type of primary key.

For example, we have the Entity class `Student`:

```java
@Entity
public class Student {
  @Id
  @SequenceGenerator(
      name = "student_id_sequence",
      sequenceName = "student_id_sequence",
      allocationSize = 1
  )
  @GeneratedValue(
      strategy = GenerationType.SEQUENCE,
      generator = "student_id_sequence"
  )
  private Long id;
  private String name;
  private Integer age;

  // constructors
  // getters and setters
}
```

We create a `StudentRepository` interface which allows us to use a database table named `Student`:

```
public interface StudentRepository extends JpaRepository<Student, Long> {
}
```

The second parameter is `Long` since `id` of `Student` class is a primary key and has a type `Long`.

`@GetMapping`

GET = retrieve data

`JpaRepository#findAll()` function is used to get entries from the table:

```
@SpringBootApplication
@RestController
@RequestMapping("customers")
public class DemoApplication {

  private final StudentRepository studentRepository;

  public DemoApplication(StudentRepository studentRepository) {
    this.studentRepository = studentRepository;
  }

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

  @GetMapping("")
  public List<Student> getStudents() {
    return studentRepository.findAll();
  }
}
```

`@PostMapping`

POST = add new data

`addStudent()` function requires an argument to be in a json format. We use a `NewStudentRequest` record to achieve it.

`JpaRepository#save()` function is used to add new entry to the table:

```
@SpringBootApplication
@RestController
@RequestMapping("customers")
public class DemoApplication {
  private final StudentRepository studentRepository;

  // constructor
  // main function
  // GET function

  record NewStudentRequest(String name, Integer age) {}

  @PostMapping("")
  public void addStudent(@RequestBody NewStudentRequest newStudentRequest) {
    Student newStudent = new Student(newStudentRequest.name,
newStudentRequest.age);
    studentRepository.save(newStudent);
  }
}
```

`@DeleteMapping`

DELETE = delete data

`JpaRepository` has various delete-functions. For example, let us delete by the primary key. We retrieve id from the url address. If the url address is `http://localhost:8080/customers/1`, then `studentId = 1`.

```
@SpringBootApplication
@RestController
@RequestMapping("customers")
public class DemoApplication {
  private final StudentRepository studentRepository;

  // constructor
  // main function
  // GET function
  // POST function

  @DeleteMapping("{studentId}")
  public void deleteStudent(@PathVariable("studentId") Long id) {
```

```java
    studentRepository.deleteById(id);
  }
}
```

`@PutMapping`

PUT = update data or create it

```java
@SpringBootApplication
@RestController
@RequestMapping("customers")
public class DemoApplication {
  private final StudentRepository studentRepository;

  // constructor
  // main function
  // GET function
  // POST function
  // DELETE function

  @PutMapping("{studentId}")
  public void updateStudent(@PathVariable("studentId") Long id, String newName) {
    Optional<Student> studentOptional = studentRepository.findById(id);
    deleteStudent(id);
    if (studentOptional.isPresent()) {
      Student student = studentOptional.get();
      addStudent(new NewStudentRequest(newName, student.getAge()));
    }
  }
}
```

All code together

```java
package com.example.demo;
import com.example.demo.student.Student;
import com.example.demo.student.StudentRepository;
import java.util.List;
import java.util.Optional;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
@RequestMapping("customers")
public class DemoApplication {
  private final StudentRepository studentRepository;

  public DemoApplication(StudentRepository studentRepository) {
    this.studentRepository = studentRepository;
  }

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



  @GetMapping("")
  public List<Student> getStudents() {
    return studentRepository.findAll();
  }



  record NewStudentRequest(String name, Integer age) {}

  @PostMapping("")
  public void addStudent(@RequestBody NewStudentRequest newStudentRequest) {
    Student newStudent = new Student(newStudentRequest.name,
newStudentRequest.age);
    studentRepository.save(newStudent);
  }



  @DeleteMapping("{studentId}")
  public void deleteStudent(@PathVariable("studentId") Long id) {
    studentRepository.deleteById(id);
  }



  @PutMapping("{studentId}")
  public void updateStudent(@PathVariable("studentId") Long id, String
newName) {
    Optional<Student> studentOptional = studentRepository.findById(id);
    deleteStudent(id);
    if (studentOptional.isPresent()) {
      Student student = studentOptional.get();
      addStudent(new NewStudentRequest(newName, student.getAge()));
    }
  }
}
```