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

In whole, **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**.

The Spring container consists of POJO classes and Metadata.

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 a sub-interface of 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.

A class annotated with `@Configuration` indicates that it contains Spring bean configurations.

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 during the application startup, registered, and then injected into different parts of an application by the container.

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:

```java
@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

```java
@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 + "' }";
    }
}



@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`.

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`

```
@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`:

```
@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;
```

```java

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

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

```java
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:

```java
@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:

```java
@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`.

```java
@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) {
    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()));
    }
    }
}
```

Configuration bean

In Spring Boot, a *configuration bean* refers to a Java class that is used to configure and customize the behavior of the Spring application.

There are several annotations that are used to define a bean:

- `@Configuration` -- indicates a class that has one or more bean definitions and configuration settings

  ```java
  @Configuration
  public class MyConfiguration {
      @Bean
      public MyBean myBean() {
          return new MyBean();
      }
  }
  ```

- `@Component` -- indicates that a class is a Spring-managed component or bean. The class is instantiated at runtime.

  The role of the component can be specified. The `@Service` or `@Repository` annotations can be used instead of `@Component`.

Dependency injection

Dependency injection is a design pattern used in Spring Boot (and other frameworks) to manage dependencies between objects or components. For example, a controller class may depend on a service class to handle business logic.

`@Autowired`

The `@Autowired` annotation is used to indicate that a particular field, constructor, or method parameter requires dependency injection. Spring Boot automatically resolves and injects the corresponding dependency bean at runtime.