SpringBoot

Matteo Moi

Contents

1 Spring/SpringBoot fundamental concepts

1.1 Inversion of Control (IoC)

Definition: IoC is a design principle where the control flow of a program is inverted. Instead of the application code controlling the flow, the framework takes control of the flow and instantiates and manages the lifecycle of objects.

1.2 Dependency Injection

Definition: It is a technique where an object receives its dependencies from an external source(in this case Spring Framework) rather than creating them internally.

1.2.1 Types of Injection in Spring

- **Constructor Injection**

```
@Component
class Client {
    private final Service service;

    // Constructor injection
    @Autowired
    public Client(Service service) {
        this.service = service;
    }
```

- **Setter Injection**

```
@Component
class Client {
    private Service service;

    // Setter injection
    @Autowired
    public void setService(Service service) {
        this.service = service;
    }
```

- **Field Injection**

```
@Component
class Client {

    // Field injection
    @Autowired
    private Service service;
```

2 SpringBoot Annotation

- **@SpringBootApplication**
- **@Component**
- **@Configuration**

2.1 @SpringBootApplication

```java
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

This annotation is a shortcut that combines three fundamental annotations in Spring :

Annotation	Description
@Configuration	Indicates that the class can be used by the Spring IoC container as a source of bean definitions.
@EnableAutoConfiguration	Enables the auto-configuration, which automatically configures your application based on the dependencies you have added.
@ComponentScan	Instructs Spring to scan the current package and its sub-packages for components, configurations, and services, allowing it to detect and register beans with the application context.

2.2 @Component

It is used to mark a Java class as a "component" so that Spring can automatically detect and manage the class as a bean within its Inversion of Control (IoC) container without explicit configuration.

```java
@Component
public class MyComponent {
    public void performAction() {
        // Business logic here
    }
}
```

2.2.1 Specialized Stereotypes

Annotation	Description
@Service	Indicates that the class holds business logic
@Repository	Indicates that the class is a Data Access Object (DAO) and will interact with the database.
@Controller	Used in Spring MVC to denote a controller class that handles HTTP requests
@RestController	combines @Controller and @ResponseBody. It is used in RESTful web services

2.3 @Configuration

Purpose :

- Define Beans in Java: @Configuration classes are used to define beans using methods annotated with @Bean. This enables type-safe, refactor-friendly configuration
- Initialize Application Context: Acts as a source for the Spring container to generate and manage bean definitions at runtime.

```
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        // Configure and return the necessary JDBC DataSource
```

3 Spring Data

In Spring Data, an entity represents a database table, and a repository provides an abstraction to perform CRUD operations on the entity. Spring Data JPA automates the creation of the repository based on the interfaces you define.

3.1 Entity

The class is annotated with @Entity and the fields with annotations like @Id and @GeneratedValue to define the primary key and its auto-generation strategy.

3.1.1 JPA

```java
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    // Default constructor required by JPA
    public User() {}
```

3.1.2 Mongo

```java
@Document(collection = "users")  // Specifies the MongoDB
   collection name
public class User {

    @Id
    private String id;  // MongoDB will automatically generate
       the `_id` field
    private String name;
    private String email;

    // Default constructor required by Spring Data
    public User() {}
```

3.2 Repository

The repository interface provides methods to interact with the database(CRUD ops) extending JpaRepository (or CrudRepository) gives these functionalities automatically.

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long>
   {
```

```
3
4       // You can define custom query methods here
5       User findByEmail(String email);
6   }
```

3.2.1 Reactive Repository

B ased on non-blocking I/O this interface do not return objects or collections of objects; instead, return Mono and Flux objects which ones are reactive streams that are capable of returning either 0...1 or 0...m entities as they become available on the stream. (Supported by Mongo, not supported by JPA)

```
1   @Repository
2   public interface UserRepository extends
       ReactiveCrudRepository<User, Long> {
3
4       // You can define custom query methods here
5       Flux<User> findByEmail(String email);
6   }
```

3.3 Service

A service layer is a common way to encapsulate business logic and handle repository interactions.

```
1   @Service
2   public class UserService {
3
4       @Autowired
5       private UserRepository userRepository;
6
7       public List<User> getAllUsers() {
8           return userRepository.findAll();
9       }
10
11      public User getUserById(Long id) {
12          return userRepository.findById(id).orElse(null);
13      }
```

4 Spring Cloud

4.1 Spring Cloud Stream

Definition: SCS provides a streaming abstraction over messaging, based on the publish and subscribe integration pattern. SCS comes with built-in support for Apache Kafka and RabbitMQ.

4.1.1 Core concepts

- **Message**: A data structure that's used to describe data sent to and received from a messaging system.
- **Publisher(Supplier)**: Sends messages to the messaging system
- **Subscriber(Consumer)**: Receives messages from the messaging system
- **Destination**: Used to communicate with the messaging system. Publisher use output destinations and Subscriber input destinations. Destinations are mapped by specific binders to queues and topics in the underlying messaging system.
- **Binder**: provides the actual integration with a specific messaging system(similar to jdbc with a specific database)

4.2 System Messaging Impl

- **Publisher**

```
1     @Bean
2     public Supplier<String> myPublisher() {
3         return()-> new Date().toString();
4     }
```

- **Consumer**

```
1     @Bean
2     public Consumer<String> mySubscriber() {
3         return s -> System.out.println("Message received: "+s
              );
4     }
```

- **Supplier/Consumer**

```
1     @Bean
2     public Function<String> myProcessor() {
3         return s -> "Message received: "+s;
4     }
```

- **Properties**

```
        # To make Spring aware of these functions
        spring.cloud.function:
            definition: myPublisher; MyProcessor; mySubscriber
        # To tell Spring what destination to use for each
           function
        spring.cloud.stream.bindings
            myPublisher-out-0:
                destination: myProcessor-in
            myProcessor-in-0:
                destination: myProcessor-in
            myProcessor-out-0:
                destination: myProcessor-out
            myConsumer-in-0:
                destination: myProcessor-out
```

Result : myPublisher -> myProcessor -> mySubscriber

On default the supplier is triggered by default every second but if you want to trigger it by an external event like a REST Api is called then:

```
    @Autowired
    private StreamBridge streamBridge;

    @PostMapping
    void exApi(@RequestBody String body){
    streamBridge.send("myProcessor-in-0", body);

    }
```

5 TODO

- add to Spring data example with Generics

6 in progress...