UNIVERSITY OF SRI JAYEWARDENEPURA

Faculty of Technology

Department of Information and Communication Technology

ITS 4243 - Microservices and Cloud Computing

Assignment 01

Index number – ICT/21/908

Name – Rathnayaka R.M.S.

Part 1

1. What is Spring Boot and why is it used?

Spring Boot is a powerful, open-source micro-framework built on top of the core Spring Framework.

Its primary purpose is to simplify the development, setup, and deployment of new Spring applications. It's used because it eliminates most of the complex configuration and setup that was traditionally required by the Spring Framework.

It achieves this through:

- **Auto-Configuration:** Spring Boot intelligently configures my application based on the dependencies (JARs) I have on your classpath. For example, if it sees the spring-boot-starter-web dependency, it automatically configures a web server (like Tomcat), Spring MVC, and JSON support.
- **Embedded Servers:** It bundles an embedded server (like Tomcat, Jetty, or Undertow) directly into your application. This means you can run your application as a simple, stand-alone JAR file (java -jar myapp.jar) without needing to deploy a WAR file to an external web server.
- **Opinionated "Starter" Dependencies:** It provides "starter" packages (e.g., spring-boot-starter-data-jpa, spring-boot-starter-security) that group all common dependencies for a specific task, simplifying dependency management.
- **Production-Ready Features:** It includes features like health checks, metrics, and externalized configuration out-of-the-box.

In short, Spring Boot lets developers focus on writing business logic instead of "boilerplate" configuration.

2. Explain the difference between Spring Framework and Spring Boot.

The easiest way to think about it is: Spring Boot is an extension of the Spring Framework, not a replacement for it. I am still using the Spring Framework, but Spring Boot helps you manage it.

Feature	**Spring Framework**	**Spring Boot**
Core Idea	A comprehensive, un-opinionated framework providing core features like Dependency Injection (DI) and transaction management.	An opinionated extension of Spring designed to simplify application setup and deployment.
Configuration	Requires explicit, manual configuration for almost everything (e.g., setting up a data source, configuring Spring MVC).	Provides auto-configuration ("convention over configuration") to set up the application automatically based on dependencies.
Server	You must manually set up and deploy your application as a WAR file to an external server (like Tomcat).	Includes an embedded server, allowing you to package and run your application as a stand-alone JAR file.
Setup	Can be complex and time-consuming. Requires managing many individual dependencies.	Very fast setup using "starters" to manage dependencies and get an application running in minutes.

3. What is Inversion of Control (IoC) and Dependency Injection (DI)?

These are core software design principles that form the foundation of the Spring Framework.

- Inversion of Control (IoC): This is a broad design principle. In traditional programming, my code creates and manages the objects it needs.

  For example: MyService service = new MyServiceImpl();.

With IoC, this control is "inverted." I delegate the creation and lifecycle management of your objects (called "beans" in Spring) to an external container (the Spring IoC Container). I code simply declares what it needs, and the container provides it.

- **Dependency Injection (DI):** This is the pattern or mechanism used to achieve IoC. It's the process by which the IoC container "injects" the dependencies (i.e., other objects) into my bean.

Analogy:

- **Without IoC:** I want a sandwich. I go to the fridge, get the bread, get the cheese, get the ham (your dependencies), and assemble the sandwich yourself.
- **With IoC/DI:** You go to a sandwich shop (the container) and say, "I need a ham and cheese sandwich" (your bean). The shop attendant (the injector) assembles the bread, ham, and cheese (the dependencies) and hands you the finished sandwich.

In Spring, DI is typically done in one of three ways:

1. **Constructor Injection (Recommended):** The container passes dependencies through the class constructor.
2. **Setter Injection:** The container calls setter methods (e.g., setMyService(...)) after creating the bean.
3. **Field Injection (Discouraged):** The container uses reflection to set private fields directly (using @Autowired).

4. What is the purpose of application.properties / application.yml?

These files are used for centralized and externalized application configuration.

Instead of hard-coding values like database passwords, server ports, or API keys directly in your Java code, you define them in this single file. Spring Boot automatically loads this file and uses the values to configure the application.

This is crucial because it allows you to:

- Change application behavior without recompiling the code.
- Use different configuration files for different environments (e.g., application-dev.properties for development, application-prod.properties for production).

application.properties uses a standard key=value format:

Properties

```
server.port=8080
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
```

application.yml uses YAML, a hierarchical format that is often more readable:

YAML

```
server:
  port: 8080
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: root
```

5. Explain what a REST API is and list HTTP methods used.

A REST API (Representational State Transfer Application Programming Interface) is an architectural style for designing networked applications. It's the most common standard for web-based services to communicate with each other.

It's based on a few key principles:

- **Resources:** It exposes data as "resources" (e.g., a "User," a "Product").
- **URIs:** Each resource is identified by a unique URI (Uniform Resource Identifier), like /api/users/123.
- **HTTP Methods:** It uses standard HTTP methods to perform actions on those resources.
- **Stateless:** Each request from a client to the server must contain all the information needed to understand and process the request. The server does not store any client "session" state.
- **Representation:** Data is transferred in a standard format, most commonly JSON (JavaScript Object Notation).

The primary HTTP methods (verbs) used are mapped to CRUD (Create, Read, Update, Delete) operations:

- **GET: Read** a resource. (e.g., GET /api/users to get all users, or GET /api/users/123 to get a specific user).
- **POST: Create** a new resource. (e.g., POST /api/users with user data in the request body to create a new user).
- **PUT: Update** or **replace** an existing resource. (e.g., PUT /api/users/123 with complete user data to update that user).
- **DELETE: Delete** a resource. (e.g., DELETE /api/users/123 to delete that user).

PATCH: (Less common) Partially update an existing resource. (e.g., send only the email field to update, without sending the whole user object).

6. What is Spring Data JPA? What is an Entity and a Repository?

Spring Data JPA is a module of the Spring Data project that makes it incredibly easy to work with databases using JPA (Java Persistence API). Its main goal is to remove boilerplate code for the data access layer (DAO layer).

- **Entity (@Entity):** An Entity is a simple Java class (POJO) that is mapped to a table in your database.
    - I annotate the class with @Entity.
    - I annotate its primary key field with @Id.
    - Each *instance* of the entity class represents a *row* in the table.
    - Each *field* of the class typically maps to a *column* in the table.
- **Repository:** A Repository is an interface that defines the data access operations (like save, find, delete).
    - With Spring Data JPA, you don't write the implementation class.
    - I simply create an interface that extends one of the provided Spring Data interfaces, like JpaRepository<User, Long>.
    - Spring Data JPA automatically generates the implementation for all the standard CRUD methods (save(), findById(), findAll(), deleteById(), etc.) at runtime.
    - I can also define custom queries just by writing the method signature (e.g., User findByEmail(String email);), and Spring Data JPA will generate the SQL for you.

7. What is the difference between @Component, @Service, @Repository, @Controller, @RestController?

These are all stereotype annotations that tell Spring to create and manage a bean for that class.

- **@Component:** This is the generic, base annotation. It indicates that a class is a Spring-managed component. Any class marked with @Component will be picked up by component-scanning and registered as a bean in the Spring container.

The other four are specializations of @Component. They all do the *same thing* (register a bean) but add semantic meaning (to signal the class's *role*) and sometimes extra behavior.

- **@Service:** Used for the Business Logic Layer. It signals that this class contains business logic or coordinates transactions (e.g., UserService, PaymentService).
- **@Repository:** Used for the Data Access Layer (Persistence Layer). It signals that this class is responsible for database access (e.g., UserRepository).
    - **Extra Behavior:** This annotation also enables a feature that automatically translates low-level persistence exceptions (like a database's SQLException) into Spring's unified DataAccessException hierarchy, making exception handling cleaner.
- **@Controller:** Used for the Presentation Layer in a traditional Spring MVC web application. It signals that this class is a web controller. Methods in a @Controller typically return a view name (e.g., a "home.jsp" or "user-list" template) that will be rendered as an HTML page.
- **@RestController:** This is a convenience annotation used for building REST APIs. It combines @Controller and @ResponseBody.
    - **Extra Behavior:** It tells Spring that *every* method in this class will return data directly as the response body (e.g., as JSON or XML), not a view name.

8. What is @Autowired? When should we avoid it?

@Autowired is the annotation used to perform automatic dependency injection. I place it on a field, constructor, or setter method to tell Spring "find a bean that matches this type and inject it here."

When to avoid it:

It is strongly recommended to avoid field injection.

This is Field Injection (the bad practice):

Java

```
@Service
public class MyService {

    @Autowired
    private MyRepository myRepository; // <-- AVOID THIS

    public void doSomething() {
        myRepository.save(...);
    }
}
```

Why you should avoid it:

1. **Hard to Test:** It makes unit testing very difficult. When I create new MyService() in a test, the myRepository field will be null. I cannot easily set it because it's private. I am forced to use reflection to set the mock, which is complex and fragile.
2. **Hides Dependencies:** It's *too* easy to add 10-15 @Autowired fields, hiding the fact that my class is complex and likely violating the Single Responsibility Principle.
3. **Immutability:** I cannot make the field final, which means the dependency can be changed after the object is created (it's mutable), which is generally less safe.

What to use instead: Constructor Injection (The Best Practice)

I should always prefer constructor injection.

Java

```
@Service
public class MyService {

    private final MyRepository myRepository; // 1. Field is final

            // 2. Dependency is injected via the constructor
            // @Autowired is optional if there's only one constructor
            public MyService(MyRepository myRepository) {
                this.myRepository = myRepository;
            }

            public void doSomething() {
```

```
        myRepository.save(...);
    }
}
```

Why this is better:

1. **Easy to Test:** I can easily create an instance in a test: MyRepository mockRepo = mock(MyRepository.class); MyService service = new MyService(mockRepo);. No reflection needed.
2. **Explicit Dependencies:** The constructor clearly lists all required dependencies.
3. **Immutability:** Dependencies can be marked final, ensuring they are set only once.

9. Explain how Exception Handling works in Spring Boot (@ControllerAdvice).

Spring Boot provides a centralized, powerful way to handle exceptions across my entire application using **@ControllerAdvice** (or **@RestControllerAdvice** for REST APIs).

This approach keeps my controller methods clean by removing the need for try-catch blocks for common errors.

Here's the workflow:

1. I create a special class and annotate it with **@ControllerAdvice**.
2. Inside this class, I create methods to handle specific exceptions. I annotate these methods with **@ExceptionHandler(**ExceptionName.class).
3. When code in any of my @Controller classes throws an exception (e.g., a ResourceNotFoundException I created), Spring's dispatcher servlet catches it.
4. It then looks for any registered @ControllerAdvice bean that has an @ExceptionHandler method for that *specific exception*.
5. It executes that method. This method is responsible for building a clean, user-friendly error response (e.g., a JSON object) and returning it with the correct HTTP status code (like 404 NOT_FOUND).

Example:

Java

```java
// 1. A custom exception
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}

// 2. The Controller method that throws it
@RestController
@RequestMapping("/api/users")
public class UserController {
    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        // This might throw the exception
        return userService.findUserById(id)
            .orElseThrow(() -> new ResourceNotFoundException("User not found
with id: " + id));
    }
}

// 3. The global exception handler
@RestControllerAdvice
public class GlobalExceptionHandler {

    // 4. This method handles *only* ResourceNotFoundException
    @ExceptionHandler(ResourceNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND) // Sets HTTP status to 404
    public ErrorResponse handleResourceNotFound(ResourceNotFoundException
ex) {
        // ErrorResponse is a custom POJO that will be serialized to JSON
        return new ErrorResponse(ex.getMessage(), "RESOURCE_NOT_FOUND");
    }
}
```

10. What is the role of Maven/Gradle in a Spring Boot project?

Maven and Gradle are build automation and dependency management tools. Their role is critical.

1. **Dependency Management:** This is their most important job. Modern applications depend on dozens (or hundreds) of third-party libraries (JAR files).
   - I declare the *direct* dependencies you need in a config file (pom.xml for Maven, build.gradle for Gradle). For Spring Boot, I just declare "starters" like spring-boot-starter-web.
   - The tool then resolves and downloads all the necessary libraries, *and* all the libraries *those* libraries depend on (transitive dependencies).
   - This prevents "JAR Hell," where you have to manually find and manage compatible versions of all libraries.

2. **Build Lifecycle:** They provide a standard, automated process for building your project. This includes common tasks (called "goals" or "tasks") like:
    - **compile:** Compiling your .java source code into .class files.
    - **test:** Running all my unit tests.
    - **package:** Taking the compiled code and packaging it into a final distributable format. For Spring Boot, this is typically a single, executable JAR file that contains your code, all dependencies, and the embedded server.
    - **clean:** Deleting all a-generated build files.

In short, Maven and Gradle build my project and manage all its libraries, so I don't have to.

Part 2

GitHub Repo Link - https://github.com/MaxDoom57/StudentAPI-ict21908.git