	codewitharrays.in freelance project available to buy contact on 8007592194	
SR.NO	Project NAME	Technology
1	Online E-Learning Platform Hub	React+Springboot+MySql
2	PG Mates / RoomSharing / Flat Mates	React+Springboot+MySql
3	Tour and Travel management System	React+Springboot+MySql
4	Election commition of India (online Voting System)	React+Springboot+MySql
5	HomeRental Booking System	React+Springboot+MySql
6	Event Management System	React+Springboot+MySql
7	Hotel Management System	React+Springboot+MySql
8	Agriculture web Project	React+Springboot+MySql
9	AirLine Reservation System / Flight booking System	React+Springboot+MySql
10	E-commerce web Project	React+Springboot+MySql
11	Hospital Management System	React+Springboot+MySql
12	E-RTO Driving licence portal	React+Springboot+MySql
13	Transpotation Services portal	React+Springboot+MySql
14	Courier Services Portal / Courier Management System	React+Springboot+MySql
15	Online Food Delivery Portal	React+Springboot+MySql
16	Muncipal Corporation Management	React+Springboot+MySql
17	Gym Management System	React+Springboot+MySql
18	Bike/Car ental System Portal	React+Springboot+MySql
19	CharityDonation web project	React+Springboot+MySql
20	Movie Booking System	React+Springboot+MySql

	freelance_Project available to buy contact on 8007592194	
21	Job Portal web project	React+Springboot+MySql
22	LIC Insurance Portal	React+Springboot+MySql
23	Employee Management System	React+Springboot+MySql
24	Payroll Management System	React+Springboot+MySql
25	RealEstate Property Project	React+Springboot+MySql
26	Marriage Hall Booking Project	React+Springboot+MySql
27	Online Student Management portal	React+Springboot+MySql
28	Resturant management System	React+Springboot+MySql
29	Solar Management Project	React+Springboot+MySql
30	OneStepService LinkLabourContractor	React+Springboot+MySql
31	Vehical Service Center Portal	React+Springboot+MySql
32	E-wallet Banking Project	React+Springboot+MySql
33	Blogg Application Project	React+Springboot+MySql
34	Car Parking booking Project	React+Springboot+MySql
35	OLA Cab Booking Portal	React+NextJs+Springboot+MySql
36	Society management Portal	React+Springboot+MySql
37	E-College Portal	React+Springboot+MySql
38	FoodWaste Management Donate System	React+Springboot+MySql
39	Sports Ground Booking	React+Springboot+MySql
40	BloodBank mangement System	React+Springboot+MySql

41	Bus Tickit Booking Project	React+Springboot+MySql
42	Fruite Delivery Project	React+Springboot+MySql
43	Woodworks Bed Shop	React+Springboot+MySql
44	Online Dairy Product sell Project	React+Springboot+MySql
45	Online E-Pharma medicine sell Project	React+Springboot+MySql
46	FarmerMarketplace Web Project	React+Springboot+MySql
47	Online Cloth Store Project	React+Springboot+MySql
48	Train Ticket Booking Project	React+Springboot+MySql
49	Quizz Application Project	JSP+Springboot+MySql
50	Hotel Room Booking Project	React+Springboot+MySql

51	Online Crime Reporting Portal Project	React+Springboot+MySql
52	Online Child Adoption Portal Project	React+Springboot+MySql
53	online Pizza Delivery System Project	React+Springboot+MySql
54	Online Social Complaint Portal Project	React+Springboot+MySql
55	Electric Vehical management system Project	React+Springboot+MySql
56	Online mess / Tiffin management System Project	React+Springboot+MySql
57		React+Springboot+MySql
58		React+Springboot+MySql
59		React+Springboot+MySql
60		React+Springboot+MySql

Spring Boot + React JS + MySQL Project List

Sr.No	Project Name	YouTube Link
1	Online E-Learning Hub Platform Project	https://youtu.be/KMjyBaWmgzg?si=YckHuNzs7eC84-IW
2	PG Mate / Room sharing/Flat sharing	https://youtu.be/4P9cIHg3wvk?si=4uEsi0962CG6Xodp
3	Tour and Travel System Project Version 1.0	https://youtu.be/-UHOBywHaP8?si=KHHfE_A0uv725f12
4	Marriage Hall Booking	https://youtu.be/VXz0kZQi5to?si=llOS-QG3TpAFP5k7
5	Ecommerce Shopping project	https://youtu.be/vJ_C6LkhrZ0?si=YhcBylSErvdn7paq
6	Bike Rental System Project	https://youtu.be/FIzsAmIBCbk?si=7ujQTJqEgkQ8ju2H
7	Multi-Restaurant management system	https://youtu.be/pvV-pM2Jf3s?si=PgvnT-yFc8ktrDxB
8	Hospital management system Project	https://youtu.be/IynIouBZvY4?si=CXzQs3BsRkjKhZCw
9	Municipal Corporation system Project	https://youtu.be/cVMx9NVyI4I?si=qX0oQt-GT-LR_5jF
10	Tour and Travel System Project version 2.0	https://youtu.be/_4u0mB9mHXE?si=gDiAhKBowi2gNUKZ

Sr.No	Project Name	YouTube Link
11	Tour and Travel System Project version 3.0	https://youtu.be/Dm7nOdpasWg?si=P_Lh2gcOFhlyudug
12	Gym Management system Project	https://youtu.be/J8_7Zrkg7ag?si=LcxV51ynfUB7OptX
13	Online Driving License system Project	https://youtu.be/3yRzsMs8TLE?si=JRI_z4FDx4Gmt7fn
14	Online Flight Booking system Project	https://youtu.be/m755rOwdk8U?si=HURvAY2VnizIyJlh
15	Employee management system project	https://youtu.be/lD1iE3W_GRw?si=Y_jv1xV_BljhrD0H
16	Online student school or college portal	https://youtu.be/4A25aEKfei0?si=RoVgZtxMk9TPdQvD
17	Online movie booking system project	https://youtu.be/Lfjv_U74SC4?si=fiDvrhhrjb4KSlSm
18	Online Pizza Delivery system project	https://youtu.be/Tp3izreZ458?si=8eWAOzA8SVdNwlyM
19	Online Crime Reporting system Project	https://youtu.be/0UlzReSk9tQ?si=6vN0e70TVY1GOwPO
20	Online Children Adoption Project	https://youtu.be/3T5HC2HKyT4?si=bntP78niYH802I7N

Q - 1) What is Spring Boot ?

Spring Boot is an extension of the Spring framework designed to simplify the process of building and deploying Java-based applications. It provides a range of features to streamline development and deployment:

Auto-Configuration: Spring Boot automatically configures your application based on the dependencies you include. For example, if you add a dependency for a database, Spring Boot will automatically configure the necessary settings for connecting to that database. Standalone: Applications built with Spring Boot are standalone and self-contained, meaning they include an embedded server (like Tomcat or Jetty) and can run independently of external server setups. Production-Ready: Spring Boot includes built-in support for monitoring and managing applications in production. Features like health checks and metrics are provided out of the box. Microservices Support: It is commonly used in developing microservices due to its ease of setup and integration with Spring Cloud, which offers tools for building distributed systems. Convention Over Configuration: It provides sensible defaults for configurations, reducing the need for boilerplate code and XML configurations. Spring Boot Starter Projects: These are convenient dependency descriptors you can include in your project to get a set of commonly used functionalities. Overall, Spring Boot aims to make it easier and faster to develop production-ready applications with minimal configuration and setup.

Q - 2) How does Spring Boot differ from the Spring Framework?

Spring Boot differs from the Spring Framework in the following ways:

Auto-Configuration: Spring Boot automatically configures components based on the dependencies present in the project, reducing the need for manual configuration.

Embedded Servers: Spring Boot includes embedded servers like Tomcat, Jetty, and Undertow, allowing you to run applications directly without needing to deploy them to an external server.

Starter Dependencies: Spring Boot provides a set of pre-defined starter dependencies that simplify dependency management.

Production-Ready Features: Spring Boot comes with production-ready features such as health checks, metrics, and monitoring via Spring Boot Actuator.

Minimal Setup: Spring Boot reduces the complexity of configuration and setup compared to the traditional Spring Framework, making it easier to get started with Spring applications.

Q - 3) What are the main features of Spring Boot?

The main features of Spring Boot include:

Auto-Configuration: Automatically configures the application based on the project’s dependencies.

Starter Dependencies: Provides a set of pre-configured dependencies for different functionalities like web, data access, security, etc.

Embedded Servers: Includes embedded servers (Tomcat, Jetty, Undertow) to run applications without needing an external server.

Spring Boot CLI: Command Line Interface for quickly developing and running Spring Boot applications.

Spring Boot Actuator: Adds production-ready features like monitoring, metrics, health checks, and externalized configuration.

Easy Integration: Simplifies the integration of Spring projects with other technologies and frameworks.

Q - 4) What is the purpose of the `@SpringBootApplication` annotation?

The @SpringBootApplication annotation is a convenience annotation that combines three commonly used annotations:

@Configuration: Indicates that the class can be used by the Spring IoC container as a source of bean definitions.

@EnableAutoConfiguration: Tells Spring Boot to automatically configure the application based on the dependencies present in the classpath.

@ComponentScan: Instructs Spring to scan the current package and its sub-packages for components, configurations, and services.

Together, these annotations make it easier to set up a Spring Boot application by reducing the amount of configuration required.

Q - 5) How do you start a Spring Boot application?

You can start a Spring Boot application in several ways:

Using `main()` Method: Run the `main()` method of the class annotated with `@SpringBootApplication`. For example:

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

Using the Command Line: If using Spring Boot CLI, you can start the application with the command:

```
spring run app.groovy
```

Using Maven or Gradle: If you have a Maven or Gradle project, you can use the following commands to start the application:

Maven: `mvn spring-boot:run`

Gradle: `gradle bootRun`

Using the Executable JAR: You can package your Spring Boot application as an executable JAR and run it with:

```
java -jar myapplication.jar
```

Q - 6) What is the default embedded server used in Spring Boot?

The default embedded server used in Spring Boot is **Tomcat**. When you create a Spring Boot application with web capabilities, Tomcat is included by default as the embedded server. However, you can easily switch to other embedded servers like Jetty or Undertow by excluding the Tomcat dependency and including the desired server dependency.

Q - 7) How can you create a Spring Boot project using Spring Initializer?

You can create a Spring Boot project using Spring Initializr in the following ways:

Web Interface:

Go to the [Spring Initializr website} (https://start.spring.io/). Select your project options, such as project type (Maven/Gradle), language (Java/Kotlin/Groovy), Spring Boot version, project metadata (Group, Artifact, etc.), and dependencies (e.g., Spring Web, Spring Data JPA). Click on the “Generate” button to download the project as a ZIP file. Unzip the file and open it in your preferred IDE. The project is ready to run or further develop. Using IDEs (like IntelliJ IDEA, Eclipse, or VS Code):

Open your IDE and navigate to the Spring Initializr project creation option. Follow similar steps to the web interface to configure the project and dependencies. The IDE will automatically set up the project based on your selections. Using Spring Boot CLI:

```
spring init --dependencies=web,data-jpa,my-dependency my-project
```

This command generates a Spring Boot project with the specified dependencies.

Q - 8) What is a `SpringApplication` class?

The `SpringApplication` class is a crucial part of starting a Spring Boot application. It is used to bootstrap and launch a Spring application from a Java main method. The `SpringApplication` class does several things:

- Sets up the default configuration.
- Starts the Spring application context.

- Performs a classpath scan.
- Starts the embedded server (if it's a web application).
- It also provides a variety of customization options through methods like `setBanner()`, `setApplicationContext()`, and `setWebApplicationType()`.

A typical use of `SpringApplication` looks like this:

```
public static void main(String[] args) {

    SpringApplication.run(MyApplication.class, args);

}
```

Q - 9) How do you run a Spring Boot application on a different port?

You can run a Spring Boot application on a different port in several ways:

Using `application.properties` or `application.yml` file:

- Add the following property to `application.properties`:

```
server.port=8081
```

- Or in `application.yml`:

```
server:

  port: 8081
```

Using Command Line Arguments:

- You can override the default port by passing the `--server.port` argument when starting the application:

```
java -jar myapplication.jar --server.port=8081
```

- Or if using Maven/Gradle:

```
mvn spring-boot:run -Dspring-boot.run.arguments="--
server.port=8081"
```

Programmatically:

- You can set the port programmatically in your main class:

```
SpringApplication app = new SpringApplication(MyApplication.class);
app.setDefaultProperties(Collections.singletonMap("server.port", "8081"));
app.run(args);
```

Q - 10) What are Spring Boot starters?

Spring Boot starters are a set of convenient dependency descriptors that you can include in your application. They simplify the process of adding dependencies to your project by providing a curated list of dependencies for a specific use case. For example:

- `spring-boot-starter-web`: Includes dependencies for building web applications, such as Spring MVC, Tomcat (as an embedded server), and validation libraries.
- `spring-boot-starter-data-jpa`: Includes dependencies for working with JPA and Hibernate in a Spring Boot application.
- `spring-boot-starter-test`: Includes testing libraries such as JUnit, Mockito, and Spring Test.

By using starters, you don't have to manually include a long list of individual dependencies; instead, you can add a single starter, and Spring Boot will bring in all the necessary dependencies.

Q - 11) Explain the purpose of the `application.properties` or `application.yml` file.

The `application.properties` or `application.yml` file in a Spring Boot application is used for externalizing configuration properties. This allows you to configure various aspects of your application without hardcoding values directly in your code. Common uses include:

- Setting server properties: Port number, context path, etc.
- Database configuration: URL, username, password, driver class name.
- Logging configuration: Log levels, log file paths.
- Custom application properties: Any custom properties that can be injected into your beans using `@Value` or `@ConfigurationProperties`.

The choice between `.properties` and `.yml` depends on your preference. `.yml` files support hierarchical data, which can be more readable for complex configurations.

Example `application.properties`:

```
server.port=8081
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=secret
```

Example `application.yml`:

```yaml
server:
  port: 8081
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: root
    password: secret
```

Q- 12) How do you enable Spring Boot Actuator in your application?

To enable Spring Boot Actuator in your application:

1. Add the Actuator dependency:

   - If using Maven, add this dependency to your `pom.xml`:

     ```xml
     <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
     </dependency>
     ```

   - If using Gradle, add this line to your `build.gradle`:

     ```groovy

     implementation 'org.springframework.boot:spring-boot-starter-actuator'

     ```

2. Configure Actuator in `application.properties` or `application.yml` (optional):

   - Customize the endpoints and security settings if necessary:

     ```properties

    management.endpoints.web.exposure.include=*

    management.endpoint.health.show-details=always

     ```

   - The above properties expose all endpoints and show health details.

3. Run the Application:

   - Once the application is started, the Actuator endpoints will be available under the `/actuator` path by default, e.g., `http://localhost:8080/actuator`.

Q - 13) What are Spring Boot Actuator endpoints?
Spring Boot Actuator endpoints are predefined HTTP endpoints that provide insights into the application's health, metrics, environment, and other operational aspects. Some common Actuator endpoints include:

- `/actuator/health`: Provides the health status of the application, indicating whether it is running correctly.

- `/actuator/metrics`: Displays various metrics such as memory usage, request count, and more.

- `/actuator/env`: Shows the environment properties, including system properties, environment variables, and configuration properties.

- `/actuator/info`: Displays arbitrary application information. You can customize it to show details like version, description, etc.

- `/actuator/loggers`: Allows you to view and configure the log levels of the application.

- `/actuator/beans`: Lists all the beans in the Spring ApplicationContext.

These endpoints can be enabled or disabled, secured, and customized as needed. By default, only a few endpoints (e.g., `/actuator/health`, `/actuator/info`) are exposed publicly.

Q - 14) What is a REST controller in Spring Boot?
A REST controller in Spring Boot is a special type of controller used to create RESTful web services. It is annotated with `@RestController`, which is a convenience annotation that combines `@Controller` and `@ResponseBody`. This means that the controller will return data directly in the form of JSON or XML (instead of rendering a view) for each request.

Example of a simple REST controller:

```
@RestController

public class MyController {

@GetMapping("/greet")

public String greet() {

    return "Hello, World!";

}

}
```

In this example, the `greet()` method returns a simple string that will be serialized to JSON and sent as a response to the client.

Q - 15) How do you define a REST endpoint in Spring Boot?
To define a REST endpoint in Spring Boot, you typically use the following annotations within a class annotated with `@RestController`:

- `@GetMapping`: To define a GET endpoint.

- `@PostMapping`: To define a POST endpoint.

- `@PutMapping`: To define a PUT endpoint.

- `@DeleteMapping`: To define a DELETE endpoint.

- `@PatchMapping`: To define a PATCH endpoint.

These annotations map HTTP requests to specific handler methods in your controller.

Example:

```
@RestController

@RequestMapping("/api")

public class MyController {

@GetMapping("/users/{id}")

public User getUser(@PathVariable Long id) {

    return userService.findUserById(id);

}


@PostMapping("/users")

public User createUser(@RequestBody User user) {

    return userService.saveUser(user);

}


@PutMapping("/users/{id}")
public User updateUser(@PathVariable Long id, @RequestBody User user)
{

    return userService.updateUser(id, user);

}



@DeleteMapping("/users/{id}")

public void deleteUser(@PathVariable Long id) {

    userService.deleteUser(id);

}

}
```

- `@RequestMapping("/api")`: Defines the base URL for all endpoints in this controller.

- `@GetMapping("/users/{id}")`: Maps a GET request to retrieve a user by their ID.

- `@PostMapping("/users")`: Maps a POST request to create a new user.

- `@PutMapping("/users/{id}")`: Maps a PUT request to update an existing user.

- `@DeleteMapping("/users/{id}")`: Maps a DELETE request to remove a user by their ID.

Each of these annotations provides a clear and concise way to define RESTful endpoints in your Spring Boot application.

Q - 16) What is the use of the `@RequestMapping` annotation?

The `@RequestMapping` annotation in Spring Boot is used to map HTTP requests to handler methods of MVC and REST controllers. It can be applied at both the class and method levels to define the base URL and specific request patterns. The annotation supports multiple attributes to specify the URL, HTTP method, request parameters, headers, and more.

Example:

```
@RestController

@RequestMapping("/api")

public class MyController {

@RequestMapping(value = "/users", method = RequestMethod.GET)

public List<User> getUsers() {

    return userService.findAllUsers();

}



@RequestMapping(value = "/users", method = RequestMethod.POST)

public User createUser(@RequestBody User user) {

    return userService.saveUser(user);

}

}
```

In this example:

- The `@RequestMapping("/api")` at the class level sets a base URL for all methods in the controller.

- The `@RequestMapping(value = "/users", method = RequestMethod.GET)` at the method level maps GET requests to `/api/users`.

- The `@RequestMapping(value = "/users", method = RequestMethod.POST)` maps POST requests to `/api/users`.

Q - 17) How do you handle exceptions in Spring Boot?

In Spring Boot, exceptions can be handled using the following approaches:

1. `@ExceptionHandler` Annotation:

   - Used within a controller to handle exceptions thrown by the controller’s methods.

   - Example:

```

@RestController

public class MyController {



    @GetMapping("/user/{id}")

    public User getUser(@PathVariable Long id) {

        return userService.findUserById(id).orElseThrow(() -> new UserNotFoundException(id));

    }



   @ExceptionHandler(UserNotFoundException.class)

    public ResponseEntity<String> handleUserNotFound(UserNotFoundException ex) {

        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);

    }

}

```

2. `@ControllerAdvice` Annotation:

   - A global exception handler that can handle exceptions across multiple controllers.

~~~~
   - Example:

     ```

     @ControllerAdvice

     public class GlobalExceptionHandler {



        @ExceptionHandler(UserNotFoundException.class)

         public ResponseEntity<String>
handleUserNotFound(UserNotFoundException ex) {

             return new ResponseEntity<>(ex.getMessage(),
HttpStatus.NOT_FOUND);

         }



         @ExceptionHandler(Exception.class)

         public ResponseEntity<String>
handleGeneralException(Exception ex) {

             return new ResponseEntity<>("An error occurred: " +
ex.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);

         }

     }

     ```



3. Custom Error Pages:

   - Configure custom error pages in `application.properties` or use
`ErrorController` to handle specific HTTP errors.

   - Example:

     ```

     server.error.whitelabel.enabled=false
~~~~

```
```

Q - 18) What is Spring Boot DevTools?

Spring Boot DevTools is a development-time tool that provides features to enhance the development experience. It includes:

- Automatic Restart: Automatically restarts the application whenever there are changes in the classpath. This is very useful for quick iterations during development.

- LiveReload Integration: Automatically reloads the browser when resources change.

- Enhanced Logging: Provides additional logging for development.

- Disable Caching: Disables template caching, allowing for real-time updates to views without needing a restart.

- Property Defaults: Provides sensible defaults for development, such as enabling debug logging and showing detailed error messages.

To use DevTools, add the following dependency to your `pom.xml` or `build.gradle`:

- Maven:

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

 <artifactId>spring-boot-devtools</artifactId>

  <optional>true</optional>
```

- Gradle:

```
developmentOnly ‘org.springframework.boot:spring-boot-devtools’
```

Q - 19) How do you enable hot swapping in a Spring Boot application?
Hot swapping allows you to reload classes without restarting the entire application. In Spring Boot, hot swapping can be enabled using Spring Boot DevTools, JRebel, or similar tools.

- Using Spring Boot DevTools:

- Simply add the DevTools dependency to your project as described above.

- DevTools will automatically restart the application whenever you save changes to the classpath.

- Using JRebel:

- JRebel is a third-party tool that provides more advanced hot swapping capabilities.

- You need to install JRebel and integrate it with your IDE.

- Using IDE-Specific Features:

- Some IDEs like IntelliJ IDEA and Eclipse provide their own hot swap functionality. This typically involves running the application in debug mode.

Q - 20) What is the difference between `@Component`, `@Service`, and `@Repository` in Spring Boot?
All three annotations—`@Component`, `@Service`, and `@Repository`—are used to define Spring beans. They are specializations of the `@Component` annotation but are used to indicate the role of the bean within the application.

- `@Component`:

- It is a generic stereotype for any Spring-managed component.

- It can be used for any Spring bean that does not fit into the other specific stereotypes.

- Example:

  ```java
  @Component
  public class MyComponent {
      // Some logic
  }
  ```

- `@Service`:

- A specialization of `@Component`, it is used to define a service layer bean.

- Indicates that the class contains business logic or service methods.

- Example:

  ```
  @Service
  public class MyService {
      // Business logic
  }
  ```

- `@Repository`:

- A specialization of `@Component`, it is used to define a data access layer bean (DAO).

- It provides additional features for exception translation, converting database-specific exceptions into Spring’s

`DataAccessException`.

- Example:

  ```
  @Repository
  public class MyRepository {
      // Data access logic
  }
  ```

The main difference lies in their intended use within the application's architecture, which helps in better organizing the code and applying appropriate AOP aspects automatically.

Q - 21) How do you configure a Spring Boot application to connect to a database?

To configure a Spring Boot application to connect to a database, follow these steps:

1. Add the necessary dependencies:

 - For example, to connect to a MySQL database, you would add the following dependency to your `pom.xml`:

   ```
   <dependency>
       <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
       <scope>runtime</scope>
   </dependency>
   <dependency>
   ```

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

      <artifactId>spring-boot-starter-data-jpa</artifactId>

   </dependency>

 - For Gradle, you would add:

   ```groovy

   implementation 'mysql:mysql-connector-java'

   implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

   ```

2. Configure database properties:

 - Add database connection details in the `application.properties` or `application.yml` file.

   ```

  spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase

   spring.datasource.username=root

   spring.datasource.password=secret

  spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver



   spring.jpa.hibernate.ddl-auto=update

   spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

   ```

 - In `application.yml`:

   ```yaml
   spring:
     datasource:
       url: jdbc:mysql://localhost:3306/mydatabase
       username: root
       password: secret
       driver-class-name: com.mysql.cj.jdbc.Driver


     jpa:
       hibernate:
         ddl-auto: update
       show-sql: true
       properties:
         hibernate:
           dialect: org.hibernate.dialect.MySQL5Dialect
   ```

3. Use `@Entity` classes and `@Repository` interfaces:

 - Define your JPA entities and repositories, and Spring Boot will automatically handle the connection to the database.

Q - 22) What is Spring Data JPA in Spring Boot?
Spring Data JPA is a part of the larger Spring Data family and is used for implementing JPA (Java Persistence API) based repositories. It provides a layer of abstraction over JPA, making it easier to implement data access layers with minimal boilerplate code. Spring Data JPA simplifies the development of repository interfaces by providing default implementations for common operations like saving,

deleting, and finding entities.

Key features of Spring Data JPA:

- Automatic Repository Implementation: By extending `JpaRepository`, you get a fully implemented repository with basic CRUD operations.

- Custom Query Methods: You can define methods in your repository interface following naming conventions, and Spring Data JPA will automatically generate the necessary queries.

- Pagination and Sorting: It provides built-in support for pagination and sorting of records.

- Auditing: It supports automatic auditing of entity changes (e.g., created/modified timestamps).

Example:

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

List<User> findByLastName(String lastName);

}
```

Q - 23) How do you define a JPA entity in Spring Boot?

To define a JPA entity in Spring Boot, you need to create a Java class and annotate it with `@Entity`. Additionally, you should define a primary key using the `@Id` annotation, and map the class fields to the corresponding database columns using JPA annotations.

Example:

```
import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity

public class User {
```

```
@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;



private String firstName;

private String lastName;

private String email;



// Getters and setters

}
```

In this example:

- `@Entity`: Marks the class as a JPA entity.

- `@Id`: Specifies the primary key of the entity.

- `@GeneratedValue`: Configures the way the primary key is generated (e.g., automatically by the database).

Q - 24) What is the purpose of the `@Entity` annotation?

The `@Entity` annotation is used to specify that a class is a JPA entity and will be mapped to a database table. This annotation is a key part of the Java Persistence API (JPA) and is used to define classes whose instances will be stored in the database.

Key points about the `@Entity` annotation:

- It is applied to the class level.

- The entity name defaults to the class name, but it can be customized using the `@Entity(name = "CustomName")` attribute.

- An entity class must have a primary key, which is defined using the

`@Id` annotation.

- The fields in the class represent the columns in the corresponding database table.

Example:

@Entity

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private Double price;

// Getters and setters

}

Q - 25) What is an H2 database and how do you use it in Spring Boot?
The H2 database is an open-source, in-memory, lightweight relational database that can be used for developing and testing applications. It is often used in Spring Boot applications for development and testing purposes because it doesn’t require any setup and can be easily integrated.

To use H2 in a Spring Boot application:

1. Add the H2 dependency:

   - In Maven:

     ```
     ```

```
      <dependency>
          <groupId>com.h2database</groupId>
         <artifactId>h2</artifactId>
          <scope>runtime</scope>
      </dependency>
```

   - In Gradle:

```
      implementation 'com.h2database:h2'
```

2. Configure H2 properties (optional):

   - In `application.properties` or `application.yml`, you can customize the H2 database settings.

```properties
      spring.datasource.url=jdbc:h2:mem:testdb
     spring.datasource.driver-class-name=org.h2.Driver
      spring.datasource.username=sa
      spring.datasource.password=password
      spring.h2.console.enabled=true
      spring.h2.console.path=/h2-console
```

   - This configuration sets up an in-memory database named `testdb`, enables the H2 console, and specifies the console path as `/h2-console`.

3. Access the H2 Console:

   - When you run your Spring Boot application, you can access the H2 console at `http://localhost:8080/h2-console`.

   - Use the database URL `jdbc:h2:mem:testdb`, username `sa`, and leave the password blank (or use the one you configured) to log in.

4. Use the H2 database in your application:

   - Spring Boot will automatically use the H2 database for your JPA entities, and you can perform operations like you would with any other database.

The H2 database is particularly useful for quick prototyping and testing because it is fast and doesn’t require any additional setup.

Q - 26) How do you enable the H2 console in Spring Boot?
To enable the H2 console in a Spring Boot application, follow these steps:

1. Add the H2 dependency:

   - In Maven:

```
<dependency>

   <groupId>com.h2database</groupId>

   <artifactId>h2</artifactId>

    <scope>runtime</scope>

</dependency>
```

   - In Gradle:

```
      ```

      implementation 'com.h2database:h2'

      ```
```

2. Configure the H2 console in `application.properties` or `application.yml`:

   - Add the following properties to enable and configure the H2 console.

   - In `application.properties`:

```
      ```

      spring.h2.console.enabled=true

      spring.h2.console.path=/h2-console

      ```
```

   - In `application.yml`:

~~~
      ```yaml

      spring:

        h2:

          console:

            enabled: true

            path: /h2-console

      ```
~~~

   - This configuration enables the H2 console and sets the console’s path to `/h2-console`.

3. Access the H2 console:

   - After starting your Spring Boot application, access the H2 console by navigating to `http://localhost:8080/h2-console` in your web browser.

   - Use the correct JDBC URL, username, and password to log in. For an in-memory database, the URL might be `jdbc:h2:mem:testdb`, with the default username `sa` and an empty password.

Q - 27) What is the difference between `save()` and `saveAndFlush()` in Spring Data JPA?

The `save()` and `saveAndFlush()` methods in Spring Data JPA are used to save an entity to the database, but they behave differently in terms of transaction and flushing behavior.

- `save()`:

  - Saves the entity and returns the saved entity.

  - The actual database write operation may be delayed until the transaction is committed, depending on the underlying JPA provider and transaction management.

  - If there are pending changes in the current session, they might not be flushed to the database immediately.

  Example:

```
userRepository.save(user);
```

- `saveAndFlush()`:

- Saves the entity and immediately flushes the changes to the database.

- The flush operation forces the JPA provider to write any pending changes in the current session to the database.

- This method is useful when you need to ensure that changes are reflected in the database immediately, such as when performing subsequent operations that depend on the saved data.

Example:

userRepository.saveAndFlush(user);

Key Difference:

`save()` might delay the actual database write until the transaction is committed, while `saveAndFlush()` forces an immediate database write.

Q - 28) How do you create a custom query in Spring Data JPA?
To create a custom query in Spring Data JPA, you can use the `@Query` annotation or the `Query` method keywords in your repository interface.

1. Using the `@Query` Annotation:

 - You can define a custom query using JPQL (Java Persistence Query Language) or native SQL directly within the `@Query` annotation.

 - Example:

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

       @Query("SELECT u FROM User u WHERE u.lastName = :lastName")

       List<User> findByLastName(@Param("lastName") String lastName);

       @Query(value = "SELECT * FROM users WHERE email = ?1",
nativeQuery = true)

       User findByEmail(String email);

   }
```

- In this example:

  - The first query uses JPQL to find users by their last name.

  - The second query uses native SQL to find a user by their email.

2. Using Query Method Keywords:

 - You can create custom queries based on method names by following Spring Data JPA’s naming conventions.

 - Example:

```java

   List<User> findByLastNameAndFirstName(String lastName, String firstName);

```

 - This method will automatically generate a query that looks for users with the specified last name and first name.

Q - 29) What is the use of the `@Query` annotation in Spring Data JPA?

The `@Query` annotation in Spring Data JPA is used to define custom queries in your repository interface. It allows you to write JPQL (Java Persistence Query Language) or native SQL queries directly within the repository interface, providing more flexibility than the standard query generation based on method names.

Key Features of `@Query`:

- Custom JPQL Queries: You can write JPQL queries that operate on JPA entities and their relationships.

- Native SQL Queries:** By setting the `nativeQuery` attribute to `true`, you can execute raw SQL queries directly against the database.

- Parameterized Queries: The `@Query` annotation supports parameters, which can be passed as method arguments and referenced in the query using `:parameterName`.

Examples:

- JPQL Query:

@Query(“SELECT u FROM User u WHERE u.email = :email”)

User findByEmail(@Param(“email”) String email);

- Native SQL Query:

@Query(value = “SELECT * FROM users WHERE email = ?1”, nativeQuery = true)

User findByEmail(String email);

Advantages:

- Flexibility: You can write complex queries that go beyond the capabilities of method name conventions.

- Optimization: You can optimize queries for performance by writing tailored SQL.

Q - 30) How do you configure logging in Spring Boot?

Logging in Spring Boot can be configured using various approaches. By default, Spring Boot uses Logback as the logging framework, but it also supports other logging frameworks like Log4j2 and Java Util Logging (JUL).

1. Configure Logging Levels:

 - You can set the logging level for specific packages or classes in the `application.properties` or `application.yml` file.

 - Example in `application.properties`:

   ```

   logging.level.org.springframework=INFO

   logging.level.com.example.myapp=DEBUG
   ```

~~~~
   ```

 - Example in `application.yml`:

   ```

   logging:

     level:

       org.springframework: INFO

       com.example.myapp: DEBUG

   ```



2. Configure Log Output Format:

 - You can customize the log format by setting the
`logging.pattern.console` or `logging.pattern.file` properties.

 - Example in `application.properties`:

   ```

   logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %msg%n

   logging.pattern.file=%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level
%logger{36} - %msg%n

   ```



3. Configure Log File:

 - To log output to a file, you can specify the file name and
location.

 - Example in `application.properties`:

   ```properties

   logging.file.name=application.log
~~~~

```
   logging.file.path=/var/logs/myapp/

   ```

4. Use a Different Logging Framework:

 - If you prefer to use Log4j2 or another logging framework, exclude the default Logback dependency and include the desired logging framework.

 - Example for Log4j2 in `pom.xml`:

```xml
   <dependency>

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

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

       <scope>runtime</scope>

       <exclusions>

           <exclusion>

              <groupId>ch.qos.logback</groupId>

              <artifactId>logback-classic</artifactId>

           </exclusion>

       </exclusions>

   </dependency>

   <dependency>

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

      <artifactId>spring-boot-starter-log4j2</artifactId>

   </dependency>
```

5. Custom Logback Configuration:

 - You can create a `logback-spring.xml` file in the `src/main/resources` directory to customize the Logback configuration.

 - Example:

   ```xml

   <configuration>

       <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">

           <encoder>

               <pattern>%d{yyyy-MM-dd HH:mm:ss} - %msg%n</pattern>

           </encoder>

       </appender>



       <root level="INFO">

           <appender-ref ref="CONSOLE"/>

       </root>

   </configuration>

   ```

This configuration controls logging output, formats, and levels for a Spring Boot application.

Q - 31) What is the purpose of the `spring-boot-starter-test` dependency?

The `spring-boot-starter-test` dependency is a comprehensive testing library provided by Spring Boot that includes a set of tools and libraries for writing unit tests, integration tests, and end-to-end tests for Spring Boot applications. It simplifies the testing process by bundling several useful testing frameworks and libraries into a single dependency.

Key Components Included:

- JUnit 5: The default testing framework for writing and running tests.

- Mockito: A popular mocking framework for creating mock objects and defining behavior in tests.

- Spring Test: Provides utilities for testing Spring components, including support for testing Spring MVC controllers.

- AssertJ: A fluent assertion library that makes assertions more readable and expressive.

- Hamcrest: A library for writing matcher objects that are used in assertions.

- JsonPath: Allows you to query and assert JSON responses.

- Spring Boot Test: Provides annotations and utilities to test Spring Boot components (e.g., `@SpringBootTest`, `@WebMvcTest`, `@DataJpaTest`).

Usage:

- To use `spring-boot-starter-test`, add the following dependency in your `pom.xml` (Maven) or `build.gradle` (Gradle) file:

- Maven:

  ```xml

  <dependency>

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

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

      <scope>test</scope>

  </dependency>
  ```

- Gradle:

  ```groovy

  testImplementation 'org.springframework.boot:spring-boot-starter-test'

  ```

Purpose:

- It simplifies the setup and execution of various types of tests within a Spring Boot application, ensuring that developers have all the necessary tools for effective testing.

Q - 32) How do you write unit tests for Spring Boot applications?
Writing unit tests in Spring Boot involves testing individual components, such as services, controllers, or repositories, in isolation from the rest of the application. The goal is to verify that each component functions correctly according to its specifications.

Steps to Write Unit Tests:

1. Set Up Your Test Class:

 - Use the `@SpringBootTest` annotation for integration tests or test configuration, but for pure unit tests, you don't typically need this.

 - Use JUnit 5 as the test framework.

2. Create a Test Class:

 - Annotate the test class with `@ExtendWith(MockitoExtension.class)` to use Mockito for mocking dependencies.

3. Mock Dependencies:

 - Use `@Mock` to create mock objects for dependencies.

 - Use `@InjectMocks` to inject the mocks into the class under test.

4. Write Test Methods:

 - Use the `@Test` annotation to mark test methods.

 - Define the behavior of mocks using `Mockito.when()`.

 - Use assertions to verify the expected outcomes using libraries like `AssertJ` or `Hamcrest`.

5. Example Unit Test:

```java
@ExtendWith(MockitoExtension.class)

public class UserServiceTest {


    @Mock

    private UserRepository userRepository;


    @InjectMocks

    private UserService userService;


    @Test

    public void testFindUserById() {

        User mockUser = new User(1L, "John", "Doe");
Mockito.when(userRepository.findById(1L)).thenReturn(Optional.of(mockU
ser));


         User user = userService.findUserById(1L);


         assertThat(user).isNotNull();

        assertThat(user.getFirstName()).isEqualTo("John");

        assertThat(user.getLastName()).isEqualTo("Doe");

     }

 }
```

Key Points:

- Isolation: Unit tests should test a single component without involving other components or external systems like databases or web servers.
- Mocking: Dependencies are mocked to isolate the behavior of the component under test.
- Assertions: Verify that the actual output matches the expected output.

Q - 33) What is the use of `@MockBean` in Spring Boot testing?

The `@MockBean` annotation is used in Spring Boot tests to create and inject mock objects into the Spring application context. It is particularly useful when you want to replace a real bean with a mock during the execution of a test, allowing you to control the behavior of the dependencies in isolation.

Key Features of `@MockBean`:

- **Bean Replacement**: Replaces a real bean in the Spring context with a mock, ensuring that your tests do not interact with real dependencies like databases, external services, or other complex components.
- **Integration with Spring Testing**: Works seamlessly with Spring Boot testing annotations like `@SpringBootTest`, `@WebMvcTest`, and `@DataJpaTest`.
- **Fine-Grained Control**: You can control the behavior of the mock using Mockito's `when()` and `verify()` methods, making it easy to simulate different scenarios.

Example:

```
@SpringBootTest

public class UserServiceTest {


    @MockBean

    private UserRepository userRepository;


    @Autowired

    private UserService userService;


    @Test

    public void testFindUserById() {

        User mockUser = new User(1L, "John", "Doe");

Mockito.when(userRepository.findById(1L)).thenReturn(Optional.of(mockU
ser));


        User user = userService.findUserById(1L);


        assertThat(user).isNotNull();

       assertThat(user.getFirstName()).isEqualTo("John");

    }

}
```

Use Cases:

- **Isolate Components**: Use `@MockBean` to isolate the component under test by replacing actual dependencies with mocks.

- **Simplify Testing**: Mocking beans like repositories or services simplifies testing by avoiding the need for actual database access or complex setup.

Q - 34) What is the difference between `@GetMapping` and `@PostMapping` in Spring Boot?

`@GetMapping` and `@PostMapping` are specialized annotations in Spring Boot that are used to handle HTTP GET and POST requests, respectively. They are shortcuts for the `@RequestMapping` annotation with specific HTTP methods.

Key Differences:

- HTTP Method:
  - `@GetMapping`: Handles HTTP GET requests, which are typically used to retrieve data from the server. GET requests are idempotent, meaning they do not change the state of the server.
  - `@PostMapping`: Handles HTTP POST requests, which are typically used to submit data to the server, such as form submissions, and create new resources. POST requests can modify the state of the server.
- Usage:
  - `@GetMapping`:

```java
@GetMapping("/users")
public List<User> getAllUsers() {
    return userService.findAllUsers();
}
```

  - `@PostMapping`:

```java
@PostMapping("/users")
public User createUser(@RequestBody User user) {
    return userService.saveUser(user);
}
```

- Parameter Binding:
  - `@GetMapping`: Typically used with query parameters or path variables to filter or specify the data to retrieve.

```
@GetMapping("/users/{id}")
public User getUserById(@PathVariable Long id) {
    return userService.findUserById(id);
}
```

– `@PostMapping`: Often used with a request body to send data to the server.

```
@PostMapping("/users")
public User createUser(@RequestBody User user) {
    return userService.saveUser(user);
}
```

- Idempotence:
  – `@GetMapping`: Is idempotent; calling the same GET request multiple times will not change the server state.
  – `@PostMapping`: Is not idempotent; calling the same POST request multiple times can result in creating multiple resources.

Summary:

- Use `@GetMapping` for operations that fetch or retrieve data.
- Use `@PostMapping` for operations that submit data to the server, often resulting in a change of state or creation of resources.

Q - 35) How do you handle file uploads in Spring Boot?

Handling file uploads in Spring Boot involves using the `MultipartFile` interface, which represents an uploaded file in a multipart request. The process typically includes creating a REST endpoint that accepts a file, processing the file, and saving it to a specific location.

Steps to Handle File Uploads:

1. Add Dependency:

   Ensure that the Spring Boot application has the necessary dependencies in `pom.xml` or `build.gradle` (e.g., `spring-boot-starter-web`).

2. Configure File Upload Properties (Optional):

In application.properties or application.yml, you can specify the maximum file size and total upload size.

```

spring.servlet.multipart.max-file-size=2MB

spring.servlet.multipart.max-request-size=2MB
```

3. Create a REST Controller:

Define an endpoint that accepts file uploads using the `@PostMapping` annotation and the `MultipartFile` parameter.

```
@RestController

@RequestMapping("/api/files")

public class FileUploadController {

   @PostMapping("/upload")

   public ResponseEntity<String> handleFileUpload(@RequestParam("file") MultipartFile file) {

       if (file.isEmpty()) {

           return ResponseEntity.badRequest().body("File is empty");

       }



       try {

           // Save the file locally

           Path path = Paths.get("uploads/" + file.getOriginalFilename());

           Files.write(path, file.getBytes());



           return ResponseEntity.ok("File uploaded successfully: " + file.getOriginalFilename());
```

```
        } catch (IOException e) {

            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Failed
to upload file");

        }

    }

}
```

4. Handle Multipart File:

The `MultipartFile` interface provides methods like `getBytes()`, `getInputStream()`, `getOriginalFilename()`, and `transferTo()` to work with the uploaded file.

5. Save or Process the File:

You can save the file to the file system, a database, or process it as needed.

Summary:

- File uploads are handled using the `MultipartFile` interface.

- The uploaded file can be processed and saved to a desired location.

- You can configure file upload properties, such as maximum file size, in your application properties.

Q - 36) What is Spring Boot's auto-configuration feature?

Spring Boot’s auto-configuration is a key feature that automatically configures Spring application components based on the dependencies that are present on the classpath and the properties defined in the application.

Key Aspects of Auto-Configuration:

1. Automatic Setup:

- Spring Boot automatically configures beans, components, and configurations by inspecting the classpath and the defined application properties.

- For example, if `spring-boot-starter-data-jpa` is included as a dependency, Spring Boot will automatically configure a `DataSource`, `EntityManager`, and JPA repositories without any manual configuration.

2. Conditional Configuration:

- Auto-configuration classes are annotated with `@ConditionalOnClass`, `@ConditionalOnMissingBean`, and other conditional annotations, which ensure that beans are only configured when certain conditions are met.

3. Custom Configuration:

- You can override or customize the auto-configuration by defining your own beans or using `@ConditionalOnProperty` to enable or disable specific configurations.

4. Disabling Auto-Configuration:

- Auto-configuration can be disabled globally using the `spring.autoconfigure.exclude` property or by excluding specific configurations with the `@SpringBootApplication(exclude = { ... })` annotation.

Example:

If you include the `spring-boot-starter-web` dependency, Spring Boot will automatically configure the necessary beans for running a web application, such as `DispatcherServlet`, `EmbeddedServletContainer`, and `ViewResolver`.

Summary:

- Auto-configuration simplifies the setup of Spring applications by automatically configuring beans based on classpath dependencies and application properties.

- It follows the convention-over-configuration principle, reducing the need for manual bean definitions.

Q - 37) How does Spring Boot achieve dependency injection?
Spring Boot achieves dependency injection using the core principles of the Spring Framework. Dependency injection (DI) is a design pattern where the dependencies of a class are provided by the Spring container rather than being created by the class itself.

Types of Dependency Injection in Spring Boot:

1. Constructor Injection:

- Dependencies are provided through a class constructor.

- Recommended as it makes dependencies explicit and supports immutability.

- Example:

```

  @Service

  public class UserService {



      private final UserRepository userRepository;



      @Autowired

      public UserService(UserRepository userRepository) {
          this.userRepository = userRepository;

      }


      public User getUser(Long id) {

          return userRepository.findById(id).orElse(null);

      }

  }
```

2. Setter Injection:

- Dependencies are provided through setter methods.

- Allows for optional dependencies and can be used for injecting dependencies after the object is created.

- Example:

```

@Service

public class UserService {


    private UserRepository userRepository;


    @Autowired

    public void setUserRepository(UserRepository userRepository) {

        this.userRepository = userRepository;

    }
    public User getUser(Long id) {

        return userRepository.findById(id).orElse(null);

    }

}
```

3. Field Injection:

- Dependencies are injected directly into the class fields using the `@Autowired` annotation.

- It is less recommended because it hides dependencies, making the class harder to test and maintain.

- Example:

  ```
  @Service

  public class UserService {



      @Autowired

      private UserRepository userRepository;



      public User getUser(Long id) {

          return userRepository.findById(id).orElse(null);

      }

  }
  ```

Spring Boot Features for DI:

- Component Scanning: Spring Boot automatically scans for classes annotated with `@Component`, `@Service`, `@Repository`, and `@Controller`, and registers them as beans in the Spring context.

- `@Autowired` Annotation: Spring Boot uses `@Autowired` to automatically inject dependencies into the class. This can be applied to constructors, setter methods, or fields.

Summary:

- Spring Boot uses the Spring Framework's DI capabilities to inject dependencies into components.

- DI can be achieved using constructor, setter, or field injection.

- Spring Boot automatically manages the lifecycle and injection of dependencies, making it easier to develop modular and testable code.

Q - 38) What is Spring Boot CLI?
Spring Boot CLI (Command Line Interface) is a command-line tool that allows developers to quickly prototype, develop, and run Spring applications using Groovy scripts without the need for boilerplate code or a full project setup.

Key Features of Spring Boot CLI:

1. Rapid Prototyping:

- Allows developers to write Spring applications in Groovy with minimal code.

- Ideal for quick prototyping and testing small snippets of code.

2. Groovy-Based:

- Groovy is a dynamic language that runs on the JVM and is compatible with Java.

- Spring Boot CLI leverages Groovy’s concise syntax to reduce boilerplate code.

3. Auto-Configuration:

- Spring Boot CLI automatically configures the application based on the dependencies declared in the script.

- No need to write `@SpringBootApplication`, `@EnableAutoConfiguration`, or similar annotations.

4. Embedded Server:

- The CLI includes an embedded web server (Tomcat or Jetty) to run web applications directly from the command line.

5. Dependencies Management:

- Dependencies can be managed using `grab` annotations in the Groovy script, similar to how Maven or Gradle manages dependencies.

Example of a Simple Web Application:

```
@RestController
class HelloController {
@RequestMapping("/")
String hello() {
    "Hello, Spring Boot CLI!"
}
}
```

~~~
Running the Application:

- Save the above code in a file named `app.groovy`.

- Run the application using the command:

  ```sh

  spring run app.groovy
~~~

Installation:

- Spring Boot CLI can be installed using SDKMAN, Homebrew, or directly from the Spring website.

Summary:

- Spring Boot CLI is a command-line tool for developing Spring applications using Groovy.
- It simplifies the process of writing and running Spring applications with minimal setup.
- Ideal for rapid prototyping and testing.

Q - 39) How do you create a custom Spring Boot starter?

Creating a custom Spring Boot starter involves packaging common configurations, dependencies, and auto-configuration logic into a reusable module that can be shared across multiple projects.

Steps to Create a Custom Spring Boot Starter:

1. Create a New Maven/Gradle Project:
   – Start by creating a new Maven or Gradle project that will contain the custom starter.
2. Add Dependencies:
   – Include the dependencies that your starter will manage in the `pom.xml` or `build.gradle` file.
   – Example for Maven:

```xml
<dependencies>
```

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <!-- Add other dependencies -->
    </dependencies>
```

3. Create Auto-Configuration Class:

   – Create a class annotated with `@Configuration` and `@

ConditionalOnClass`, `@ConditionalOnMissingBean`, etc., to provide the auto-configuration logic.

- Example:

```
@Configuration
@ConditionalOnClass(MyService.class)
public class MyServiceAutoConfiguration {


    @Bean
    @ConditionalOnMissingBean
    public MyService myService() {
        return new MyService();
    }
}
```

4. Create `spring.factories` File:

   – In `src/main/resources/META-INF/`, create a `spring.factories` file.

   – Register your auto-configuration class in this file.

   – Example:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=

com.example.MyServiceAutoConfiguration
```

5. Package and Publish:
   - Package your starter as a JAR file.
   - Publish it to a Maven repository (e.g., Maven Central, JFrog Artifactory) so that it can be reused in other projects.
6. Use the Custom Starter:
   - In another Spring Boot application, include the custom starter as a dependency.
   - Example:

```xml
<dependency>
    <groupId>com.example</groupId>
    <artifactId>my-custom-starter</artifactId>
     <version>1.0.0</version>
</dependency>
```

Summary:

- A custom Spring Boot starter packages common configurations and dependencies into a reusable module.
- It involves creating auto-configuration classes and registering them in the `spring.factories` file.
- The starter can be shared and reused across multiple Spring Boot projects.

Q - 40) How do you integrate Spring Boot with Thymeleaf?

Integrating Spring Boot with Thymeleaf involves setting up Thymeleaf as the view layer in a Spring Boot MVC application. Thymeleaf is a modern server-side Java template engine used for rendering dynamic web pages.

Steps to Integrate Thymeleaf with Spring Boot:

1. Add Thymeleaf Dependency:
   - Include the `spring-boot-starter-thymeleaf` dependency in your `pom.xml` or `build.gradle` file.
   - Example for Maven:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

2. Create Thymeleaf Templates:
   - Create HTML templates with the `.html` extension in the `src/main/resources/templates` directory.
   - Thymeleaf templates use special attributes like `th:text`, `th:href`, `th:each`, etc., to bind dynamic data.
3. Set Up a Controller:
   - Create a Spring MVC controller to handle requests and return the name of the Thymeleaf template to be rendered.
   - Example:

```java
@Controller
public class HomeController {

    @GetMapping("/")
    public String home(Model model) {
        model.addAttribute("message", "Hello, Thymeleaf!");
         return "home"; // Refers to src/main/resources/templates/home.html
    }
}
```

4. Configure Thymeleaf (Optional):

   - Spring Boot auto-configures Thymeleaf with sensible defaults, but you can customize the configuration in `application.properties`.
   - Example:

     ```properties

spring.thymeleaf.prefix=classpath:/templates/

spring.thymeleaf.suffix=.html

spring.thymeleaf.cache=false

```

5. Render Thymeleaf Views:

   - Thymeleaf templates can access model attributes, perform iterations, conditionals, and include fragments for reusable components.
   - Example Template (`home.html`):

```html
<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

    <title>Home</title>

</head>

<body>

    <h1 th:text="${message}">Hello, Thymeleaf!</h1>

</body>

</html>
```

Summary:

- Thymeleaf is integrated with Spring Boot by adding the `spring-boot-starter-thymeleaf` dependency.
- Thymeleaf templates are placed in the `templates` directory and are rendered by returning their names from controllers.

- Spring Boot auto-configures Thymeleaf, but you can customize its behavior through application properties.

Q - 41) . What is a `Bean` in Spring Boot?

In Spring Boot (and the Spring Framework in general), a `Bean` is an object that is managed by the Spring IoC (Inversion of Control) container. Beans are created, configured, and managed by the Spring container based on the configuration provided in the application.

Key Points about Beans:

- Lifecycle Management: Spring manages the entire lifecycle of beans, including their creation, initialization, and destruction.
- Dependency Injection: Beans can be injected into other beans or components using dependency injection.
- Configuration: Beans are defined in configuration classes or XML files and can be customized with various lifecycle callbacks and scopes.

Summary:

- A `Bean` is an object managed by the Spring IoC container.
- It is created and managed by Spring, enabling dependency injection and lifecycle management.

Q - 42) How do you define a Bean in Spring Boot?

In Spring Boot, you can define a bean using various methods, including Java configuration and component scanning.

1. Java Configuration:
   - Beans can be defined in a `@Configuration` class using the `@Bean` annotation.
   - Example:

```
@Configuration

public class AppConfig {


    @Bean

    public MyService myService() {
        return new MyService();
    }
}
```

2. Component Scanning:
   - Beans can be automatically discovered and registered by annotating classes with `@Component`, `@Service`, `@Repository`, or `@Controller`.
   - Example:

```
@Service
public class MyService {
    // Service logic
}
```

3. Using `@ComponentScan`:
   - `@ComponentScan` can be used to specify the packages to scan for components.
   - Example:

```
@SpringBootApplication
@ComponentScan(basePackages = "com.example.services")
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

Summary:

- Beans can be defined using the `@Bean` annotation in a `@Configuration` class or through component scanning with annotations like `@Component`, `@Service`, `@Repository`, or `@Controller`.

Q - 43) What is the purpose of the @Bean annotation?

The @Bean annotation is used in Spring to indicate that a method produces a bean to be managed by the Spring container. It is typically used in @Configuration classes to define and configure beans that are not automatically discovered by component scanning.

Key Purposes of @Bean:

1. Explicit Bean Definition:
   - @Bean is used to explicitly define a bean and its configuration. This allows you to create beans programmatically and customize their initialization.
2. Integration with Existing Code:
   - It is useful for integrating third-party libraries or existing code that cannot be annotated with Spring-specific annotations.
3. Customization and Configuration:
   - Allows for fine-grained control over bean initialization and configuration, including setting properties, initializing resources, and defining custom scopes.

Example:

```
@Configuration

public class AppConfig {



    @Bean

    public DataSource dataSource() {

        return new HikariDataSource(); // Example of creating and
configuring a DataSource bean

    }

}
```

Summary:

- The @Bean annotation defines a method that produces a bean to be managed by the Spring container.
- It is used for explicit bean definition and customization.

Q - 44) What is the `@Configuration` annotation used for in Spring Boot?

The `@Configuration` annotation is used in Spring Boot to indicate that a class declares one or more `@Bean` methods and should be processed by the Spring container to generate bean definitions and service requests for those beans.

Key Purposes of `@Configuration`:

1. Define Beans:
   - Classes annotated with `@Configuration` contain `@Bean` methods that define and configure beans to be managed by the Spring container.
2. Enable Dependency Injection:
   - The `@Configuration` class enables the Spring container to perform dependency injection and manage the lifecycle of the beans.
3. Centralize Configuration:
   - It centralizes the configuration of beans in one place, making it easier to manage and modify the application's setup.

Example:

```
@Configuration

public class AppConfig {



    @Bean

    public MyService myService() {

        return new MyService();

    }



    @Bean

    public MyRepository myRepository() {

        return new MyRepository();

    }

}
```

Summary:

- The `@Configuration` annotation indicates that a class provides Spring configuration and bean definitions.
- It is used to centralize bean creation and configuration logic.

Q - 45) How do you create a custom Spring Boot configuration?

Creating a custom Spring Boot configuration involves defining your own configuration classes and beans that tailor the application's setup according to your specific needs.

Steps to Create a Custom Spring Boot Configuration:

1. Create a Configuration Class:
   - Annotate a class with `@Configuration` to mark it as a source of bean definitions.
   - Example:

```
@Configuration
public class CustomConfig {
    // Bean definitions go here
}
```

2. Define Beans:
   - Inside the `@Configuration` class, define beans using the `@Bean` annotation.
   - Example:

```
@Configuration
public class CustomConfig {

    @Bean
    public CustomService customService() {
        return new CustomService();
    }
}
```

3. Use Profiles (Optional):

   – Use `@Profile` to specify different configurations for different environments (e.g., development, production).

   – Example:

```
@Configuration
@Profile("dev")
public class DevConfig {

    @Bean
    public DevService devService() {
        return new DevService();
    }
}
```

4. Customize Properties (Optional):

   – You can also create custom properties files or use `@PropertySource` to load additional properties.

   – Example:

```
@Configuration
@PropertySource("classpath:custom.properties")
public class CustomConfig {
  // Bean definitions and property bindings
}
```

5. Integrate with Spring Boot:

    – The custom configuration will automatically be picked up by Spring Boot and merged with other configurations.

Summary:

- Create a custom Spring Boot configuration by defining a class with `@Configuration` and `@Bean` annotations.
- You can use `@Profile` for environment-specific configurations and `@PropertySource` for custom properties.
- Spring Boot will integrate and apply your custom configuration in the application context.

Q - 46) What is the role of the `@ConditionalOnProperty` annotation in Spring Boot?

The `@ConditionalOnProperty` annotation is used to conditionally enable or disable a Spring bean or configuration based on the presence or value of a specific property in the application's configuration.

Key Purposes:

1. Conditional Bean Creation:
    – It allows you to create beans conditionally based on property values, which can be useful for enabling features only when certain conditions are met.
2. Flexibility in Configuration:
    – It provides flexibility in configuration by allowing beans or configurations to be included or excluded based on properties defined in `application.properties` or `application.yml`.
3. Example Use Case:
    – You might want to enable a feature or component only in a specific environment or under certain conditions.

Example:

```
@Configuration

public class MyConfig {



    @Bean

    @ConditionalOnProperty(name = "feature.enabled", havingValue =
"true")
```

```
    public MyFeatureService myFeatureService() {
        return new MyFeatureService();
    }
}
```

In this example, the `MyFeatureService` bean will only be created if the property `feature.enabled` is set to `true`.

Summary

- The `@ConditionalOnProperty` annotation conditionally creates beans or configurations based on the value of a property in the configuration files.

Q - 47) . How do you create a REST client using `RestTemplate` in Spring Boot?

`RestTemplate` is a synchronous HTTP client provided by Spring for making RESTful web service calls. To create a REST client using `RestTemplate` in Spring Boot:

1. Add Dependency:
   - If you are using Spring Boot Starter Web, `RestTemplate` is already included. Otherwise, add the `spring-boot-starter-web` dependency.
2. Create a `RestTemplate` Bean:
   - Define a `RestTemplate` bean in a configuration class to be used for making HTTP requests.
   - Example:

```
@Configuration
public class AppConfig {

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

3. Use RestTemplate to Make HTTP Requests:

   - Inject RestTemplate into your service or component and use it to perform HTTP operations.
   - Example:

```
@Service

public class MyService {


    @Autowired

    private RestTemplate restTemplate;


    public String getDataFromExternalApi() {

        String url = "https://api.example.com/data";

        ResponseEntity<String> response = 
restTemplate.getForEntity(url, String.class);

        return response.getBody();

    }

}
```

Summary:

- RestTemplate is used for making synchronous HTTP requests in Spring Boot.
- Define a RestTemplate bean and use it to perform HTTP operations such as GET, POST, PUT, and DELETE.

Q - 48) What is `WebClient` and how does it differ from `RestTemplate`?

WebClient is a non-blocking, reactive HTTP client introduced in Spring WebFlux, which is part of the Spring 5.0 and later releases. It provides a more modern and flexible approach for making HTTP requests compared to RestTemplate.

Differences between WebClient and RestTemplate:

1. Blocking vs. Non-Blocking:

- `RestTemplate`: Synchronous and blocking. It waits for the response before continuing execution.
- `WebClient`: Asynchronous and non-blocking. It supports reactive programming and does not block the thread while waiting for the response.

2. Reactive Programming Support:
   - `RestTemplate`: Does not support reactive programming.
   - `WebClient`: Built to support reactive programming with Project Reactor, allowing for more scalable applications.
3. API Design:
   - `RestTemplate`: Older and somewhat simpler API.
   - `WebClient`: More flexible and feature-rich API with support for reactive streams, streaming responses, and more advanced features.

Example of Using `WebClient`:

```
@Service

public class MyService {



    private final WebClient webClient;



    @Autowired

    public MyService(WebClient.Builder webClientBuilder) {

        this.webClient = 
webClientBuilder.baseUrl("https://api.example.com").build();

    }



    public Mono<String> getDataFromExternalApi() {

        return webClient.get()

                        .uri("/data")
                        .retrieve()
                       .bodyToMono(String.class);
    }
}
```

Summary:

- `WebClient` is a non-blocking, reactive HTTP client introduced in Spring WebFlux.
- It provides asynchronous and non-blocking capabilities, making it suitable for reactive applications, while `RestTemplate` is synchronous and blocking.

Q - 49) How do you implement security in a Spring Boot application using Spring Security?

Spring Security is a powerful and customizable authentication and access control framework for Java applications. To implement security in a Spring Boot application using Spring Security:

1. Add Spring Security Dependency:
   - Include the `spring-boot-starter-security` dependency in your `pom.xml` or `build.gradle` file.
   - Example for Maven:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

2. Configure Security:
   - Create a configuration class that extends `WebSecurityConfigurerAdapter` to customize security settings.
   - Example:

```
@Configuration
@EnableWebSecurity
```

```
public class SecurityConfig extends
WebSecurityConfigurerAdapter {


    @Override

    protected void configure(HttpSecurity http) throws
Exception {

        http

            .authorizeRequests()

            .antMatchers("/public/**").permitAll() // Allow
unauthenticated access to /public/*

            .anyRequest().authenticated() // Require
authentication for all other requests

            .and()

            .formLogin() // Enable form-based login

            .and()

            .httpBasic(); // Enable basic authentication

    }


    @Override

    protected void configure(AuthenticationManagerBuilder
auth) throws Exception {

        auth

            .inMemoryAuthentication()

            .withUser("user").password("{noop}password").rol
es("USER") // Example user

            .and()

            .withUser("admin").password("{noop}admin").roles
("ADMIN"); // Example admin
```

```
        }

    }
```

3. Define Users and Roles:

   – Users and roles can be defined in-memory, in a database, or through an external authentication provider.

4. Secure Endpoints:

   – Use method-level security annotations such as `@PreAuthorize` or `@Secured` to restrict access to specific methods.

   – Example:

```
@RestController
public class MyController {


    @GetMapping("/admin")
   @PreAuthorize("hasRole('ADMIN')")
    public String adminPage() {
        return "Admin page";
    }
}
```

Summary:

- Implement Spring Security by adding the `spring-boot-starter-security` dependency and configuring security settings using `WebSecurityConfigurerAdapter`.
- Define authentication mechanisms and secure endpoints using method-level security annotations.

Q - 50) What is CSRF protection and how do you disable it in Spring Boot?

CSRF (Cross-Site Request Forgery) protection is a security measure designed to prevent unauthorized actions on a web application performed by a malicious actor using an authenticated user’s credentials.

Disabling CSRF Protection:

1. Why Disable CSRF?

   – Disabling CSRF protection is generally discouraged unless absolutely necessary, such as in stateless REST APIs where CSRF protection is not applicable.

2. How to Disable CSRF Protection:

   – In Spring Security, you can disable CSRF protection by configuring the `HttpSecurity` object in your security configuration class.

   – Example:

```
@Configuration

@EnableWebSecurity

public class SecurityConfig extends
WebSecurityConfigurerAdapter {



    @Override

    protected void configure(HttpSecurity http) throws
Exception {

        http

            .csrf().disable() // Disable CSRF protection

            .authorizeRequests()

            .anyRequest().authenticated()

            .and()

            .formLogin();

    }

}
```

Summary:

- CSRF protection prevents unauthorized actions on behalf of authenticated users.

- It can be disabled in Spring Boot by configuring `HttpSecurity` and calling `csrf().disable()` in the security configuration. Disabling CSRF is typically done for specific use cases, such as stateless APIs.

Q - 51) How do you implement OAuth2 authentication in Spring Boot?

OAuth2 is a framework for authorization that allows third-party services to exchange access tokens for secure access to user data. To implement OAuth2 authentication in Spring Boot, you generally need to configure Spring Security with OAuth2 support.

Steps to Implement OAuth2 Authentication:

1. Add Dependencies:
   - Include `spring-boot-starter-oauth2-client` and `spring-boot-starter-security` dependencies in your `pom.xml` or `build.gradle` file.
   - Example for Maven:

```
<dependency>

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

<artifactId>spring-boot-starter-oauth2-client</artifactId>

</dependency>
```

2. Configure OAuth2 Client Properties:
   - Add OAuth2 client configuration in `application.properties` or `application.yml` with details from your OAuth2 provider (e.g., Google, GitHub).
   - Example for `application.yml`:

```
spring:

  security:

    oauth2:

      client:

        registration:

          google:
            client-id: your-client-id

            client-secret: your-client-secret

            scope: profile, email

            redirect-uri:
"{baseUrl}/login/oauth2/code/{registrationId}"

        provider:

          google:

            authorization-uri:
https://accounts.google.com/o/oauth2/auth

            token-uri: https://oauth2.googleapis.com/token

            user-info-uri:
https://www.googleapis.com/oauth2/v3/userinfo
```

3. Create a Security Configuration:

   – Extend `WebSecurityConfigurerAdapter` to configure OAuth2 login and security settings.

   – Example:

```
@Configuration

@EnableWebSecurity

public class SecurityConfig extends
WebSecurityConfigurerAdapter {



    @Override

    protected void configure(HttpSecurity http) throws
Exception {

        http

            .authorizeRequests()

            .anyRequest().authenticated()
```

```
                    .and()
                    .oauth2Login(); // Enable OAuth2 login
                }
            }
```

4. Handle OAuth2 Login:
    - Spring Security will handle the OAuth2 login flow, including redirecting to the OAuth2 provider and handling the authorization code exchange.
5. Access User Information:
    - You can access user details from the `OAuth2User` object in your controllers or services.
    - Example:

```
@RestController
public class UserController {

    @GetMapping("/user")
    public String userInfo(OAuth2User principal) {
        return "Hello, " + principal.getName();
    }
}
```

Summary:

- Implement OAuth2 authentication in Spring Boot by adding the appropriate dependencies, configuring OAuth2 client properties, and setting up security configuration to handle OAuth2 login.

Q - 52) What are Spring Profiles?

Spring Profiles are a way to define and manage different configurations for different environments (e.g., development, testing, production) in a Spring Boot application. They allow you to activate specific beans or configurations based on the environment.

Key Points about Spring Profiles:

1. Environment-Specific Configuration:
   - Profiles enable you to maintain different configurations for different environments, such as database settings or API keys.
2. Activation:
   - Profiles can be activated through application properties, command-line arguments, or programmatically.
3. Conditional Beans:
   - You can define beans that are only created if a specific profile is active using the `@Profile` annotation.

Example of Using Profiles:

```
@Configuration

@Profile("dev")

public class DevConfig {



    @Bean

    public DataSource dataSource() {

        // Development-specific DataSource configuration

        return new HikariDataSource();

    }

}
```

Summary:

- Spring Profiles allow you to define environment-specific configurations and activate them based on the current profile.

Q - 53) How do you create different environments (e.g., dev, prod) in Spring Boot using profiles?

To create and manage different environments in Spring Boot using profiles, follow these steps:

1. Define Profiles in Configuration Files:

- Create separate configuration files for each profile, such as `application-dev.properties`, `application-prod.properties`.
- Example:
  - `application-dev.properties`:

    ```
    spring.datasource.url=jdbc:h2:mem:testdb
    spring.datasource.username=sa
    spring.datasource.password=password
    ```

  - `application-prod.properties`:

    ```properties

spring.datasource.url=jdbc:mysql://prod-db-server:3306/mydb

spring.datasource.username=produser

spring.datasource.password=prodpassword

```

2. Activate Profiles:
   - Specify the active profile in `application.properties`, command-line arguments, or environment variables.
   - Example in `application.properties`:

     ```
     spring.profiles.active=dev
     ```

   - Example using command-line arguments:

     ```
     java -jar myapp.jar --spring.profiles.active=prod
     ```

3. Use `@Profile` Annotation:
   - Annotate beans or configuration classes with `@Profile` to specify which profile they belong to.
   - Example:

     ```
     @Configuration
     @Profile("dev")
     ```

```java
public class DevConfig {

    // Development-specific beans

}
```

Summary:

- Manage different environments in Spring Boot using profiles by defining environment-specific configuration files and activating the desired profile.

Q - 54) How do you deploy a Spring Boot application to an external server?

To deploy a Spring Boot application to an external server, follow these general steps:

1. Package the Application:
   - Build your Spring Boot application into a JAR or WAR file using Maven or Gradle.
   - Example for Maven:

     ```
     mvn clean package
     ```

   - Example for Gradle:

     ```
     ./gradlew build
     ```

2. Prepare the Server:
   - Ensure the target server has the required runtime environment (e.g., Java Runtime Environment for a JAR file or a servlet container for a WAR file).
3. Deploy the Application:
   - For JAR Files:
     - Copy the JAR file to the server.
     - Run the application using the `java -jar` command.
     - Example:

       ```
       java -jar myapp.jar
       ```

   - For WAR Files:
     - Deploy the WAR file to a servlet container like Apache Tomcat or Jetty.

- Copy the WAR file to the `webapps` directory of Tomcat or use the management interface to deploy.

4. Configure Server Settings:
   - Configure any necessary server settings, such as environment variables, ports, or context paths.
5. Monitor and Manage:
   - Monitor the application's logs and performance.
   - Use tools like Spring Boot Actuator for management and monitoring.

Summary:

- Package your Spring Boot application into a JAR or WAR file and deploy it to an external server by running the JAR file or deploying the WAR file to a servlet container.

Q - 55) What is the use of `@Autowired` in Spring Boot?

The `@Autowired` annotation is used in Spring Boot (and the Spring Framework) for automatic dependency injection. It allows Spring to automatically inject the required beans into your components, services, and controllers.

Key Uses of `@Autowired`:

1. Dependency Injection:
   - `@Autowired` can be used to inject dependencies into fields, constructors, or setter methods of a class.
   - Example with Field Injection:

```
@Service
public class MyService {

    @Autowired
    private MyRepository myRepository;

    // Service methods
}
```

- Example with Constructor Injection:

```
@Service
public class MyService {

    private final MyRepository myRepository;

    @Autowired
    public MyService(MyRepository myRepository) {
        this.myRepository = myRepository;
    }

    // Service methods
}
```

2. Optional Dependencies:
   - You can use `@Autowired(required = false)` to indicate that a dependency is optional.
   - Example:

```
@Autowired(required = false)
private OptionalService optionalService;
```

3. Qualifiers:
   - Use `@Qualifier` along with `@Autowired` to specify which bean to inject when multiple beans of the same type are available.
   - Example:

```
@Autowired
```

```
    @Qualifier("mySpecificBean")

    private MyService myService;
```

Summary:

- `@Autowired` is used for automatic dependency injection in Spring Boot, allowing Spring to manage and inject dependencies into your components, services, and controllers.

Q - 56) How do you inject a bean using a constructor in Spring Boot?

Constructor injection is a preferred method for dependency injection in Spring Boot as it makes the dependencies immutable and ensures they are provided when the bean is created.

Steps to Inject a Bean Using Constructor Injection:

1. Define the Bean:
   - Define the bean you want to inject using `@Component`, `@Service`, `@Repository`, or `@Configuration` annotations.
2. Use Constructor Injection in Your Class:
   - Annotate the constructor with `@Autowired`, though it is optional from Spring 4.3 onwards if there is only one constructor.

Example:

```
@Service

public class MyService {



    private final MyRepository myRepository;



    // Constructor injection

    @Autowired

    public MyService(MyRepository myRepository) {

        this.myRepository = myRepository;

    }
    // Service methods

}
```

Summary:

- Use constructor injection by defining a constructor in your class and annotating it with `@Autowired` to automatically inject the required beans.

Q - 57) What is the difference between `@Autowired` and `@Resource`?

1. `@Autowired`:

   – Source: Spring Framework.

   – Functionality: Injects dependencies by type. It supports automatic resolution of dependencies.

   – Default Behavior: If there are multiple beans of the same type, you need to use `@Qualifier` to specify which bean to inject.

   – Usage Example:

   ```
   @Autowired

   private MyService myService;
   ```

2. `@Resource`:

   – Source: Java EE (javax.annotation).

   – Functionality: Injects dependencies by name. If the bean name matches the field name, it will be injected. It also supports injection by type.

   – Default Behavior: It injects by name first and then by type if no name match is found.

   – Usage Example:

   ```
   @Resource(name = "myServiceBean")

   private MyService myService;
   ```

Summary:

- `@Autowired` is a Spring-specific annotation that injects by type and can be used with `@Qualifier` for more precise injection.

- `@Resource` is a Java EE annotation that injects by name first and then by type.

Q - 58) How do you schedule tasks in Spring Boot?

Spring Boot supports task scheduling through the `@Scheduled` annotation, which allows you to run tasks at fixed intervals or according to a cron expression.

Steps to Schedule Tasks:

1. Enable Scheduling:
   - Annotate a configuration class with `@EnableScheduling` to enable scheduling support.
   - Example:

   ```
   @Configuration

   @EnableScheduling

   public class SchedulingConfig {

   }
   ```

2. Define Scheduled Tasks:
   - Use the `@Scheduled` annotation on methods that you want to execute periodically.
   - Fixed Rate: Executes the task at a fixed interval, measured from the start of the previous execution.

   ```
   @Scheduled(fixedRate = 5000) // Every 5 seconds

   public void performTask() {

       // Task logic

   }
   ```

   - Fixed Delay: Executes the task at a fixed interval, measured from the end of the previous execution.

   ```
   @Scheduled(fixedDelay = 5000) // 5 seconds after the last execution

   public void performTask() {
   ```

```
    // Task logic
}
```

- Cron Expression: Executes the task according to a cron expression.

  ~~~java

  @Scheduled(cron = "0 0 * * * ?") // Every hour on the hour

  public void performTask() {

  // Task logic

  }

  ``

Summary:

- Schedule tasks in Spring Boot using the `@Scheduled` annotation with fixed rates, delays, or cron expressions. Enable scheduling with `@EnableScheduling`.

## Q - 59 ) How do you implement caching in Spring Boot?

Spring Boot provides caching support through the `@Cacheable`, `@CachePut`, and `@CacheEvict` annotations. To implement caching:

1. Add Cache Dependencies:
   - Include a caching provider dependency like `spring-boot-starter-cache` and a caching implementation (e.g., Ehcache, Redis).
   - Example for Maven:

```xml
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
~~~

2. Enable Caching:
   - Annotate a configuration class with `@EnableCaching`.
   - Example:

```
@Configuration

@EnableCaching

public class CacheConfig {

}
```

3. Use Cache Annotations:

   – `@Cacheable`: Caches the result of a method call.

```
@Cacheable("items")

public Item getItemById(Long id) {

    // Method logic

}
```

   – `@CachePut`: Updates the cache with the result of the method call.

```
@CachePut(value = "items", key = "#item.id")

public Item updateItem(Item item) {

    // Method logic

}
```

   – `@CacheEvict`: Removes items from the cache.

```
@CacheEvict(value = "items", allEntries = true)

public void clearCache() {

    // Method logic

}
```

Summary:

- Implement caching in Spring Boot by adding the `spring-boot-starter-cache` dependency, enabling caching with `@EnableCaching`, and using cache annotations like `@Cacheable`, `@CachePut`, and `@CacheEvict`.

Q - 60) What is Spring Boot Actuator's role in monitoring?

Spring Boot Actuator provides production-ready features for monitoring and managing your Spring Boot application. It exposes various endpoints that help you monitor and manage the application's health, metrics, and other operational aspects.

Key Roles of Spring Boot Actuator:

1. Health Checks:
   - Provides an endpoint (`/actuator/health`) to check the health status of the application and its dependencies (e.g., database, disk space).
2. Metrics:
   - Exposes metrics data (`/actuator/metrics`) that provides insights into application performance and usage, such as memory usage, garbage collection, and request statistics.
3. Application Info:
   - Provides an endpoint (`/actuator/info`) that exposes metadata about the application, such as build version and description.
4. Logging:
   - Allows for dynamic logging level changes via the `/actuator/loggers` endpoint.
5. Environment:
   - Provides details about the application's environment (`/actuator/env`), including configuration properties and environment variables.
6. Custom Endpoints:
   - You can create custom actuator endpoints to expose additional application-specific information or management functions.

Example Configuration:

```
management:
  endpoints:
    web:
      exposure:
        include: health, info, metrics
```

Summary:

- Spring Boot Actuator provides monitoring and management features through various endpoints, helping you to monitor application health, performance metrics, and environment details.

Q - 61) How do you create a custom Actuator endpoint?

To create a custom Actuator endpoint in Spring Boot, you can implement the `Endpoint` or `AbstractEndpoint` class and expose it through Spring Boot Actuator.

Steps to Create a Custom Actuator Endpoint:

1. Add Dependencies:
   - Ensure you have the `spring-boot-starter-actuator` dependency in your project.
2. Implement the Custom Endpoint:
   - Create a new class that implements `Endpoint` or extends `AbstractEndpoint`.
   - Example using `AbstractEndpoint`:

```
import
org.springframework.boot.actuate.endpoint.AbstractEndpoint;

import org.springframework.stereotype.Component;



@Component

public class CustomEndpoint extends
AbstractEndpoint<String> {



    public CustomEndpoint() {

        super("customEndpoint", String.class);

    }



    @Override

    public String invoke() {
```

```
                return "Custom Endpoint Response";
            }
        }
```

3. Expose the Endpoint:

   – By default, custom endpoints are exposed if the `management.endpoints.web.exposure.include` property includes your endpoint.

   – You can configure exposure in `application.properties` or `application.yml`:

```
management:
  endpoints:
    web:
      exposure:
        include: customEndpoint
```

Summary:

- Create a custom Actuator endpoint by implementing `Endpoint` or extending `AbstractEndpoint`, then expose it through Spring Boot Actuator configuration.

Q - 62) What is `@EnableAutoConfiguration` in Spring Boot?

`@EnableAutoConfiguration` is an annotation in Spring Boot that enables auto-configuration of Spring application context based on the dependencies present in the classpath.

Key Points:

1. Automatic Configuration:

   – It automatically configures beans and settings based on the project's dependencies and configurations. For example, if `spring-boot-starter-data-jpa` is present, it configures JPA-related beans automatically.

2. Component Scanning:

   – It works in conjunction with `@ComponentScan` and `@Configuration` to scan for components and configure beans automatically.

3. Usage:

    - Typically used in combination with `@SpringBootApplication`, which is a convenience annotation that includes `@EnableAutoConfiguration`.

Example:

```
@SpringBootApplication

public class MyApplication {

    public static void main(String[] args) {

        SpringApplication.run(MyApplication.class, args);

    }

}
```

Summary:

- `@EnableAutoConfiguration` enables Spring Boot's automatic configuration of the application context based on dependencies, making it easier to set up and configure a Spring Boot application.

Q - 63) How do you exclude a specific auto-configuration class from Spring Boot's auto-configuration?

You can exclude specific auto-configuration classes from Spring Boot's auto-configuration by using the `exclude` attribute of the `@SpringBootApplication` annotation or the `@EnableAutoConfiguration` annotation.

Steps to Exclude Auto-Configuration Classes:

1. Use `@SpringBootApplication` Annotation:
    - Exclude auto-configuration classes directly in the `@SpringBootApplication` annotation.
    - Example:

```
@SpringBootApplication(exclude =
{DataSourceAutoConfiguration.class})

public class MyApplication {

    public static void main(String[] args) {

        SpringApplication.run(MyApplication.class, args);
        }

    }
```

2. Use `@EnableAutoConfiguration` Annotation:

    - If you use `@EnableAutoConfiguration` directly, you can exclude auto-configuration classes like this:

    ```
    @Configuration

    @EnableAutoConfiguration(exclude =
    {DataSourceAutoConfiguration.class})

    public class MyConfig {

    }
    ```

Summary:

- Exclude specific auto-configuration classes by using the `exclude` attribute in the `@SpringBootApplication` or `@EnableAutoConfiguration` annotations.

Q - 64) How do you configure externalized configuration in Spring Boot?

Spring Boot provides multiple ways to configure externalized configuration, allowing you to manage configurations outside the application code. Here are some common methods:

1. Application Properties or YAML Files:

    - Use `application.properties` or `application.yml` files located in `src/main/resources` to define application-specific configurations.

    - Example (`application.properties`):

    ```properties

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=user

spring.datasource.password=pass

```

2. Command-Line Arguments:

    - Pass configuration properties as command-line arguments when starting the application.

   - Example:

```
java -jar myapp.jar
--spring.datasource.url=jdbc:mysql://localhost:3306/mydb
```

3. Environment Variables:
   - Set environment variables that Spring Boot will use to override properties.
   - Example:

```
export
SPRING_DATASOURCE_URL=jdbc:mysql://localhost:3306/mydb
```

4. Profile-Specific Configuration Files:
   - Define different configurations for different profiles by creating files like `application-dev.properties` or `application-prod.properties`.
   - Activate profiles in `application.properties`:

```
spring.profiles.active=dev
```

5. Configuration Server (Spring Cloud Config):
   - Use Spring Cloud Config Server to manage and serve configuration properties from a central repository.

Summary:

- Configure externalized configuration in Spring Boot using properties files, YAML files, command-line arguments, environment variables, profile-specific files, or a configuration server for centralized management.

Q - 65) What is the purpose of the `spring.config.name` property?

The `spring.config.name` property is used to specify the name of the configuration file(s) that Spring Boot should load. By default, Spring Boot looks for configuration files named `application.properties` or `application.yml`. If you want to use a different name for your configuration files, you can set this property to specify the name.

Usage:

1. Specify a Different Config File Name:
   - Set the `spring.config.name` property to the desired file name (without the extension) in `application.properties`, `application.yml`, or as a command-line argument.

- Example in `application.properties`:

```
spring.config.name=myapp
```

- Example command-line argument:

```
java -jar myapp.jar --spring.config.name=myapp
```

2. Multiple Config Files:

   - You can also specify multiple configuration files by separating the names with commas.
   - Example:

```
spring.config.name=app1,app2
```

Summary:

- The `spring.config.name` property specifies the name of the configuration file(s) to be loaded by Spring Boot, allowing customization of configuration file names.

Q - 66) How do you secure REST APIs in Spring Boot?

Securing REST APIs in Spring Boot involves using Spring Security to handle authentication and authorization. Here's how you can secure REST APIs:

1. Add Spring Security Dependency:

   - Include the `spring-boot-starter-security` dependency in your `pom.xml` or `build.gradle` file.
   - Example for Maven:

```
<dependency>

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

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

</dependency>
```

2. Configure Security Settings:

   - Create a `WebSecurityConfigurerAdapter` class to define security rules.
   - Example:

```
import org.springframework.context.annotation.Bean;

import
org.springframework.context.annotation.Configuration;

import
org.springframework.security.config.annotation.web.builders
.HttpSecurity;

import
org.springframework.security.config.annotation.web.builders
.AuthenticationManagerBuilder;

import
org.springframework.security.config.annotation.web.configur
ation.EnableWebSecurity;

import
org.springframework.security.config.annotation.web.configur
ation.WebSecurityConfigurerAdapter;

import
org.springframework.security.crypto.bcrypt.BCryptPasswordEn
coder;

import
org.springframework.security.crypto.password.PasswordEncode
r;



@Configuration

@EnableWebSecurity

public class SecurityConfig extends
WebSecurityConfigurerAdapter {



    @Override

    protected void configure(HttpSecurity http) throws
Exception {

        http
```

```
                .authorizeRequests()
               .antMatchers("/public/**").permitAll() // Public URLs
                .anyRequest().authenticated() // Secure all other URLs
                .and()
                .httpBasic(); // Basic authentication
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .inMemoryAuthentication()
            .withUser("user")
            .password(passwordEncoder().encode("password"))
            .roles("USER");
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

3. Implement User Authentication:

   - Use in-memory authentication, JDBC authentication, or integrate with an external authentication provider (e.g., LDAP, OAuth2).
4. Use JWT (JSON Web Tokens) for Stateless Authentication:
   - For more secure and stateless authentication, you can use JWT. Implement a filter to handle JWT validation.
5. Implement Role-Based Access Control (RBAC):
   - Define roles and permissions to control access to different parts of the API.

Summary:

- Secure REST APIs in Spring Boot using Spring Security to handle authentication and authorization. Configure security settings, implement user authentication, and use JWT for stateless authentication.

Q - 67) How do you use Swagger to document REST APIs in Spring Boot?

Swagger (now part of OpenAPI) is used to generate interactive API documentation for REST APIs. To use Swagger with Spring Boot, you can integrate the Springfox library or Springdoc OpenAPI.

Using Springdoc OpenAPI:

1. Add Springdoc OpenAPI Dependency:
   - Include the `springdoc-openapi-ui` dependency in your `pom.xml` or `build.gradle` file.
   - Example for Maven:

```
<dependency>

   <groupId>org.springdoc</groupId>

   <artifactId>springdoc-openapi-ui</artifactId>

    <version>2.1.0</version>

</dependency>
```

2. Configure Swagger:
   - Springdoc OpenAPI automatically generates documentation for your REST APIs without additional configuration.
3. Access Swagger UI:
   - Run your Spring Boot application and access the Swagger UI at `/swagger-ui.html` or `/swagger-ui/index.html`.

– Example URL:

```
http://localhost:8080/swagger-ui.html
```

Using Springfox:

1. Add Springfox Dependency:
   - Include the `springfox-boot-starter` dependency in your `pom.xml` or `build.gradle` file.
   - Example for Maven:

```
<dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>
```

2. Configure Swagger:
   - Create a configuration class to set up Swagger.
   - Example:

```
import org.springframework.context.annotation.Bean;

import
org.springframework.context.annotation.Configuration;

import springfox.documentation.builders.PathSelectors;

import
springfox.documentation.builders.RequestHandlerSelectors;

import springfox.documentation.spring.web.plugins.Docket;

import
springfox.documentation.swagger2.annotations.EnableSwagger2
;
```

```
@Configuration

@EnableSwagger2

public class SwaggerConfig {


    @Bean

    public Docket api() {

        return new Docket(DocumentationType.SWAGGER_2)

            .select()

           .apis(RequestHandlerSelectors.any())

            .paths(PathSelectors.any())

            .build();

    }

}
```

3. Access Swagger UI:

   - Run your Spring Boot application and access the Swagger UI at `/swagger-ui/`.
   - Example URL:

```
http://localhost:8080/swagger-ui/
```

Summary:

- Use Swagger to document REST APIs in Spring Boot by integrating Springdoc OpenAPI or Springfox. Add the appropriate dependency, configure Swagger, and access the Swagger UI for interactive documentation.

Q - 68) How does Spring Boot handle circular dependencies?

Spring Boot, using the Spring Framework, handles circular dependencies through a process called "dependency injection" and by utilizing proxy-based techniques.

Key Points:

1. Constructor Injection:

  - Spring cannot resolve circular dependencies with constructor injection alone because it requires all dependencies to be available at the time of object creation. If a circular dependency is detected, Spring will throw a `BeanCurrentlyInCreationException`.

2. Setter Injection:

  - Spring can resolve circular dependencies when using setter injection or field injection. Spring first creates the beans and injects them through setters or fields. This allows for the creation of circular references, as the dependencies are injected after the bean is instantiated.

3. Proxy-Based Injection:

  - For certain cases, Spring can use proxies to handle circular dependencies. This involves creating proxy objects that act as placeholders until the actual beans are fully created and injected.

Example of Circular Dependency:

```
@Component

public class A {

    @Autowired

    private B b;

}


@Component

public class B {

    @Autowired

    private A a;

}
```

Summary:

- Spring Boot handles circular dependencies by using setter or field injection, and proxy-based techniques, especially when dealing with bean creation.

Q - 69) What is a `@Primary` bean in Spring Boot?

The `@Primary` annotation in Spring Boot is used to indicate a preferred bean when multiple candidates are available for autowiring. It helps resolve ambiguity by specifying which bean should be injected when there are multiple beans of the same type.

Usage:

1. Define Multiple Beans:
   - You might have multiple beans of the same type but want one to be the default choice.
   - Example:

```
@Component
public class FooService implements Service {
    // Implementation
}


@Component
@Primary
public class BarService implements Service {
    // Implementation
}
```

2. Autowiring with `@Primary`:
   - When autowiring the `Service` type, Spring will choose the bean marked with `@Primary`.
   - Example:

```
@Autowired
private Service service; // Injects BarService
```

Summary:

- The `@Primary` annotation designates a preferred bean to be injected when multiple beans of the same type are available, helping to resolve autowiring ambiguities.

Q - 70) How do you create a multi-module Spring Boot project?

Creating a multi-module Spring Boot project involves setting up a parent project with multiple child modules, each representing a distinct part of your application. This approach helps in organizing and managing complex applications with multiple components.

Steps to Create a Multi-Module Spring Boot Project:

1. Create the Parent Project:
   - Create a new Maven or Gradle project to act as the parent project.
   - Example Maven `pom.xml`:

```
<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
http://maven.apache.org/POM/4.0.0">

   <modelVersion>4.0.0</modelVersion>

   <groupId>com.example</groupId>

   <artifactId>parent-project</artifactId>

   <version>1.0-SNAPSHOT</version>

    <packaging>pom</packaging>

    <modules>

        <module>module1</module>

        <module>module2</module>

    </modules>

    <dependencyManagement>

        <dependencies>
```

```
            <!-- Define shared dependencies here -->
        </dependencies>
    </dependencyManagement>
</project>
```

2. Create Child Modules:
   - Create individual Spring Boot modules as sub-projects within the parent project.
   - Example Maven `pom.xml` for `module1`:

```
<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 http://maven.apache.org/POM/4.0.0">
   <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.example</groupId>
        <artifactId>parent-project</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
   <artifactId>module1</artifactId>
    <dependencies>
        <!-- Module-specific dependencies -->
    </dependencies>
</project>
```

3. Add Module Dependencies:

- Define dependencies between modules in the child `pom.xml` files as needed.
- Example:

```
<dependency>
    <groupId>com.example</groupId>
    <artifactId>module2</artifactId>
</dependency>
```

4. Build the Project:
   - Build the entire project from the parent directory to compile all modules together.
   - Maven command:

```
mvn clean install
```

Summary:

- Create a multi-module Spring Boot project by setting up a parent project with multiple child modules, each with its own `pom.xml` or `build.gradle`. Manage shared dependencies in the parent project and build the entire project from the parent directory.

Q - 71) How do you use Spring Boot with Kafka?

Spring Boot integrates with Apache Kafka to provide support for building event-driven microservices. Here's how you can use Spring Boot with Kafka:

1. Add Kafka Dependencies:
   - Include the `spring-kafka` dependency in your `pom.xml` or `build.gradle` file.
   - Example for Maven:

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
     <version>3.0.0</version>
```

```
</dependency>
```

2. Configure Kafka Properties:

   – Add Kafka properties to your `application.properties` or `application.yml` file.

   – Example (`application.properties`):

   ```properties

spring.kafka.bootstrap-servers=localhost:9092

spring.kafka.consumer.group-id=my-group

spring.kafka.consumer.auto-offset-reset=earliest

```

3. Create Kafka Producer and Consumer:

   – Producer Configuration:

```
import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.stereotype.Service;



@Service

public class KafkaProducer {

    private final KafkaTemplate<String, String>
kafkaTemplate;



    public KafkaProducer(KafkaTemplate<String, String>
kafkaTemplate) {

        this.kafkaTemplate = kafkaTemplate;

    }



    public void sendMessage(String topic, String message) {
```

```
        kafkaTemplate.send(topic, message);
    }
}
```

- Consumer Configuration:

```
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;


@Service
public class KafkaConsumer {


    @KafkaListener(topics = "my-topic", groupId = "my-
group")
    public void listen(String message) {
        System.out.println("Received message: " + message);
    }
}
```

4. Enable Kafka:
   - Use `@EnableKafka` to enable Kafka support in your Spring Boot application.
   - Example:

```
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplicatio
n;
```

```
@SpringBootApplication

@EnableKafka

public class KafkaApplication {

    public static void main(String[] args) {

        SpringApplication.run(KafkaApplication.class, args);

    }

}
```

Summary:

- Integrate Spring Boot with Kafka by adding the `spring-kafka` dependency, configuring Kafka properties, and creating Kafka producer and consumer services.

Q - 72) How do you implement asynchronous processing in Spring Boot?

Asynchronous processing in Spring Boot allows methods to run in separate threads, improving application responsiveness. Here's how to implement it:

1. Add the `@EnableAsync` Annotation:
   - Enable asynchronous support in your Spring Boot application by adding the `@EnableAsync` annotation to a configuration class.
   - Example:

```
import
org.springframework.context.annotation.Configuration;

import
org.springframework.scheduling.annotation.EnableAsync;


@Configuration

@EnableAsync

public class AsyncConfig {

}
```

2. Create Asynchronous Methods:

- Use the @Async annotation on methods you want to run asynchronously.
- Example:

```
import org.springframework.scheduling.annotation.Async;

import org.springframework.stereotype.Service;



@Service

public class AsyncService {



    @Async

    public CompletableFuture<String> processAsync() {

        // Simulate long-running task

        return
CompletableFuture.completedFuture("Processing complete");

    }

}
```

3. Use the Asynchronous Method:
   - Call the asynchronous method and handle the result if needed.
   - Example:

```
@Autowired

private AsyncService asyncService;



public void doSomething() {

    CompletableFuture<String> future =
asyncService.processAsync();

    // Optionally, use future.get() to block and get the
result

}
```

Summary:

- Implement asynchronous processing in Spring Boot by enabling it with `@EnableAsync`, using the `@Async` annotation on methods, and handling the results with `CompletableFuture`.

Q - 73) How do you handle distributed transactions in Spring Boot?

Handling distributed transactions in Spring Boot typically involves coordinating transactions across multiple services or databases. Here are common approaches:

1. Use of Distributed Transaction Managers:
   - For managing distributed transactions, use distributed transaction managers like Atomikos or Bitronix. These are JTA (Java Transaction API) implementations that can manage transactions across multiple resources.
2. Implementing Saga Pattern:
   - The Saga pattern is used for managing distributed transactions through a series of local transactions. Each service performs its part of the transaction and communicates with others to ensure consistency. If any step fails, compensating actions are taken.
3. Use of Two-Phase Commit Protocol:
   - Implement the two-phase commit protocol (2PC) for coordinating transactions across multiple resources. This requires a transaction manager that supports 2PC.
4. Spring Cloud Support:
   - If using Spring Cloud, you can leverage distributed transaction management tools like Spring Cloud Sleuth and Spring Cloud Stream for tracking and coordinating transactions across services.

Example with Atomikos:

1. Add Dependencies:
   - Include Atomikos dependencies in your `pom.xml` or `build.gradle`.
2. Configure Atomikos:
   - Set up Atomikos transaction manager and configure your data sources to use it.

Summary:

- Handle distributed transactions in Spring Boot using distributed transaction managers, implementing the Saga pattern, or using two-phase commit protocols. Spring Cloud tools can also help in managing distributed transactions.

Q - 74) What is Spring Cloud and how does it relate to Spring Boot?

Spring Cloud is a set of tools and libraries designed to help developers build cloud-native applications. It provides solutions for common challenges in distributed systems, such as configuration management, service discovery, circuit breaking, and distributed tracing.

Relationship with Spring Boot:

1. Integration with Spring Boot:
   - Spring Cloud integrates seamlessly with Spring Boot, providing additional functionalities for building and deploying microservices. It builds upon the capabilities of Spring Boot to handle cloud-native application requirements.
2. Key Features:
   - Configuration Management: Spring Cloud Config for externalized configuration management.
   - Service Discovery: Spring Cloud Netflix Eureka or Spring Cloud Consul for service discovery.
   - Circuit Breaker: Spring Cloud Circuit Breaker with libraries like Hystrix or Resilience4j.
   - Distributed Tracing: Spring Cloud Sleuth for tracking requests across microservices.
   - API Gateway: Spring Cloud Gateway for routing and filtering requests.
3. Dependency Management:
   - Spring Cloud provides dependencies and starters that work with Spring Boot, making it easy to include and configure cloud-related features.

Summary:

- Spring Cloud extends the capabilities of Spring Boot by providing tools and libraries for cloud-native applications, including configuration management, service discovery, circuit breaking, and distributed tracing.

Q - 75) How do you implement service discovery in Spring Boot using Spring Cloud?

Service discovery in Spring Boot using Spring Cloud allows microservices to find and communicate with each other dynamically. Here’s how to implement it using Spring Cloud Eureka:

1. Add Dependencies:

   – Include Spring Cloud Eureka dependencies in your `pom.xml` or `build.gradle` files.

   – Eureka Server (Service Registry):

```
<dependency>

   <groupId>org.springframework.cloud</groupId>

   <artifactId>spring-cloud-starter-netflix-eureka-
server</artifactId>

</dependency>
```

   – Eureka Client (Service Registration):

```
<dependency>

   <groupId>org.springframework.cloud</groupId>

   <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>

</dependency>
```

2. Set Up Eureka Server:

   – Create a Spring Boot application and annotate it with `@EnableEurekaServer`.

   – Example:

```
import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplicatio
n;

import
org.springframework.cloud.netflix.eureka.server.EnableEurek
aServer;



@SpringBootApplication
```

```
@EnableEurekaServer

public class EurekaServerApplication {

    public static void main(String[] args) {

        SpringApplication.run(EurekaServerApplication.class,
args);

    }

}
```

- Configure Eureka server properties in `application.properties` or `application.yml`.

```
spring.application.name=eureka-server

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false
```

3. Set Up Eureka Client:

   - Annotate your Spring Boot application with `@EnableEurekaClient` or `@EnableDiscoveryClient`.
   - Example:

```
import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplicatio
n;

import
org.springframework.cloud.netflix.eureka.EnableEurekaClient
;



@SpringBootApplication

@EnableEurekaClient
```

```
public class MyServiceApplication {

    public static void main(String[] args) {

        SpringApplication.run(MyServiceApplication.class,
args);

    }

}
```

- Configure client properties in `application.properties` or `application.yml`.

  ```properties

  spring.application.name=my-service

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

```

Summary:

-Implement service discovery in Spring Boot using Spring Cloud Eureka by setting up a Eureka server (service registry) and configuring microservices as Eureka clients. This allows services to dynamically register themselves and discover other services.

Q - 76) What is a microservices architecture, and how do you implement it using Spring Boot?

A microservices architecture is an approach to building applications as a collection of loosely coupled, independently deployable services. Each service is responsible for a specific business function and communicates with other services over well-defined APIs, usually HTTP.

Implementation with Spring Boot:

1. Design Your Microservices:
   - Identify and design individual services based on business capabilities or functionalities.
2. Create Spring Boot Projects:
   - Create separate Spring Boot applications for each microservice. Each microservice will have its own `pom.xml` or `build.gradle` file, and its own `application.properties` or `application.yml`.
3. Define APIs:

- Use Spring MVC or Spring WebFlux to define RESTful APIs for communication between services.

4. Service Registration and Discovery:
   - Use Spring Cloud Netflix Eureka for service discovery and registration. Each service registers itself with Eureka, allowing other services to discover it.
5. Centralized Configuration:
   - Use Spring Cloud Config for externalized configuration management. This allows you to centralize configuration properties and manage them from a single location.
6. Resilience and Fault Tolerance:
   - Implement circuit breakers using Spring Cloud Circuit Breaker with libraries like Hystrix or Resilience4j to handle failures and improve resilience.
7. API Gateway:
   - Use Spring Cloud Gateway to route and filter requests to various microservices. It provides a unified entry point for your microservices.
8. Inter-Service Communication:
   - Use REST, gRPC, or messaging systems (like Kafka) for inter-service communication.
9. Logging and Monitoring:
   - Implement distributed tracing with Spring Cloud Sleuth and monitoring with Spring Boot Actuator.

Example:

```
// Sample REST controller in a microservice

@RestController

@RequestMapping("/api")

public class MyServiceController {



    @GetMapping("/greet")

    public ResponseEntity<String> greet() {

        return ResponseEntity.ok("Hello from MyService!");

    }
}
```

Summary:

- Implement a microservices architecture with Spring Boot by creating independent services, defining APIs, using service discovery (Eureka), centralizing configuration (Spring Cloud Config), and implementing resilience (Hystrix/Resilience4j). Use an API Gateway for routing and distributed tracing for monitoring.

Q - 77) How do you implement circuit breakers in Spring Boot using Hystrix or Resilience4j?

Circuit breakers are used to handle failures and prevent cascading failures in distributed systems. Spring Boot supports circuit breakers through Spring Cloud Circuit Breaker with libraries like Hystrix and Resilience4j.

Using Hystrix:

1. Add Hystrix Dependencies:
   - Include the `spring-cloud-starter-netflix-hystrix` dependency in your `pom.xml` or `build.gradle`.
   - Example for Maven:

```
<dependency>

   <groupId>org.springframework.cloud</groupId>


<artifactId>spring-cloud-starter-netflix-hystrix</artifactI
d>

</dependency>
```

2. Enable Circuit Breaker:
   - Use the `@EnableCircuitBreaker` annotation in your Spring Boot application.
   - Example:

```
import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplicatio
n;
```

```
import
org.springframework.cloud.netflix.hystrix.EnableHystrix;



@SpringBootApplication

@EnableHystrix

public class MyApplication {

    public static void main(String[] args) {

       SpringApplication.run(MyApplication.class, args);

    }

}
```

3. Define Circuit Breaker Logic:

   – Use the `@HystrixCommand` annotation to define methods that should be protected by a circuit breaker.

   – Example:

```
import
com.netflix.hystrix.contrib.javanica.annotation.HystrixComm
and;

import org.springframework.stereotype.Service;



@Service

public class MyService {



    @HystrixCommand(fallbackMethod = "fallbackMethod")

    public String riskyMethod() {

        // Simulate failure
        throw new RuntimeException("Service failed");
    }

    public String fallbackMethod() {
        return "Fallback response";
    }
}
```

Using Resilience4j:

1. Add Resilience4j Dependencies:
   - Include the `spring-cloud-starter-circuitbreaker-resilience4j` dependency.
   - Example for Maven:

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

2. Define Circuit Breaker Logic:
   - Use the `@CircuitBreaker` annotation to define methods protected by a circuit breaker.
   - Example:

```
import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import org.springframework.stereotype.Service;


@Service
```

```
public class MyService {


    @CircuitBreaker(name = "myService", fallbackMethod =
"fallbackMethod")

    public String riskyMethod() {

        // Simulate failure

        throw new RuntimeException("Service failed");

    }


    public String fallbackMethod(Throwable t) {

        return "Fallback response";

    }

}
```

Summary:

- Implement circuit breakers in Spring Boot using Hystrix or Resilience4j by adding the relevant dependencies, enabling the circuit breaker feature, and annotating methods with `@HystrixCommand` or `@CircuitBreaker` to handle failures gracefully and provide fallback methods.

Q - 78) How do you use Spring Boot with Docker?

Using Docker with Spring Boot involves creating a Docker image of your Spring Boot application and running it in a Docker container. Here's how to do it:

1. Create a Dockerfile:

   – Define a Dockerfile in the root directory of your Spring Boot project.

   – Example Dockerfile:

```
# Use a base image with Java

FROM openjdk:17-jdk-slim
```

```
# Set the working directory

WORKDIR /app


# Copy the jar file into the container

COPY target/myapp.jar /app/myapp.jar


# Run the jar file

ENTRYPOINT ["java", "-jar", "myapp.jar"]


# Expose the application port

EXPOSE 8080
```

2. Build the Docker Image:
   - Use the Docker CLI to build the image from the Dockerfile.
   - Command:

     ```
     docker build -t myapp:latest .
     ```

3. Run the Docker Container:
   - Start a container from the image using Docker.
   - Command:

     ```
     docker run -p 8080:8080 myapp:latest
     ```

4. Access the Application:
   - Access your Spring Boot application by navigating to `http://localhost:8080` in your browser or via HTTP requests.

Summary:

- Use Docker with Spring Boot by creating a Dockerfile, building a Docker image, and running the application in a Docker container. The Dockerfile should include steps to copy the jar file and set up the runtime environment.

Q - 79) What is the purpose of Spring Cloud Config?

Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. It allows you to manage and centralize application configuration across multiple services and environments.

Key Purposes:

1. Centralized Configuration Management:
   - Manage configuration properties in a central repository (e.g., Git, SVN) and provide these properties to multiple microservices.
2. Dynamic Configuration Updates:
   - Support dynamic configuration updates without restarting the services. Applications can refresh their configuration at runtime.
3. Environment-Specific Configuration:
   - Store configuration properties for different environments (e.g., development, staging, production) and switch between them based on profiles.

Components:

1. Config Server:
   - The Config Server hosts the configuration data and serves it to client applications. It can fetch configurations from various sources, such as Git repositories or files.
2. Config Client:
   - The Config Client retrieves configuration data from the Config Server and applies it to the application. It can automatically refresh its configuration based on updates from the Config Server.

Example:

- Config Server:

```
import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplication;

import
org.springframework.cloud.config.server.EnableConfigServer;
```

```
@SpringBootApplication

@EnableConfigServer

public class ConfigServerApplication {

    public static void main(String[] args) {

        SpringApplication.run(ConfigServerApplication.class,
args);

    }

}
```

- Config Client:

```
import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplication;

import
org.springframework.cloud.context.config.annotation.RefreshScope;

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

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

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



@SpringBootApplication

public class ConfigClientApplication {

    public static void main(String[] args) {

        SpringApplication.run(ConfigClientApplication.class,
args);

    }
}


@RestController

@RefreshScope

@RequestMapping("/config")

public class ConfigController {

    @Value("${config.property}")

    private String configProperty;


    @GetMapping

    public String getConfigProperty() {

        return configProperty;

    }

}
```

Summary:

- Spring Cloud Config provides centralized configuration management and dynamic updates for microservices, enabling consistent and environment-specific configurations across services.

Q - 80) How do you centralize configuration in a microservices architecture using Spring Boot?

To centralize configuration in a microservices architecture, you can use Spring Cloud Config to manage and distribute configuration properties from a single source to all your microservices.

Steps to Centralize Configuration:

1. Set Up Spring Cloud Config Server:
   - Create a Spring Boot application that acts as the Config Server. It fetches configuration properties from a central repository (e.g., Git, filesystem) and exposes them to clients.

Example Config Server Setup:

```
# application.yml for Config Server

spring:

  cloud:

    config:

      server:

        git:

          uri: https://github.com/your-repo/config-repo
```

```
@SpringBootApplication

@EnableConfigServer

public class ConfigServerApplication {

    public static void main(String[] args) {

       SpringApplication.run(ConfigServerApplication.class,
args);

    }

}
```

2. Configure Microservices to Use Config Server:

   – Modify each microservice to include `spring-cloud-starter-config` and configure it to point to the Config Server.

Example Client Configuration:

```
# application.yml for Client Microservice

spring:

  application:

    name: my-service

  cloud:
    config:

      uri: http://localhost:8888
```

```
@SpringBootApplication

@EnableDiscoveryClient

public class MyServiceApplication {

    public static void main(String[] args) {

       SpringApplication.run(MyServiceApplication.class, args);

    }

}
```

3. Define Configuration Properties in a Repository:
   - Store configuration properties in a version-controlled repository, such as Git. Organize them by application name and profile.

   Example Git Repository Structure:

```
config-repo/

├── my-service.yml

└── application.yml
```

4. Use Configuration Properties in Microservices:
   - Access configuration properties in microservices using the `@Value` annotation or `@ConfigurationProperties`.

   Example Usage:

```
@RestController

@RefreshScope

public class MyController {

    @Value("${my.property}")

    private String myProperty;
```

```
        @GetMapping("/property")
        public String getProperty() {
            return myProperty;
        }
    }
```

5. Handle Dynamic Configuration Changes:
   - Use `@RefreshScope` to enable dynamic updates of configuration properties without restarting the service. Trigger a refresh by hitting the `/actuator/refresh` endpoint if using Spring Boot Actuator.

Summary:

- Centralize configuration in a microservices architecture using Spring Cloud Config by setting up a Config Server, configuring microservices to use it, storing properties in a central repository, and accessing these properties dynamically.

Q - 81) How do you use Spring Boot with a message broker like RabbitMQ?

To use Spring Boot with RabbitMQ, you need to configure Spring Boot to connect to RabbitMQ and define message producers and consumers.

1. Add RabbitMQ Dependencies:
   - Include the `spring-boot-starter-amqp` dependency in your `pom.xml` or `build.gradle`.
   - Example for Maven:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

2. Configure RabbitMQ Connection:
   - Set RabbitMQ properties in `application.properties` or `application.yml`.

– Example configuration:

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

3. Define RabbitMQ Configuration:

   – Create a configuration class to define queues, exchanges, and bindings.

   – Example:

```
import org.springframework.amqp.core.Bindings;

import org.springframework.amqp.core.Queue;

import org.springframework.amqp.core.TopicExchange;

import org.springframework.amqp.rabbit.core.RabbitTemplate;

import
org.springframework.amqp.rabbit.listener.SimpleMessageListe
nerContainer;

import
org.springframework.amqp.rabbit.listener.api.ChannelAwareMe
ssageListener;

import
org.springframework.amqp.rabbit.connection.ConnectionFactor
y;

import org.springframework.context.annotation.Bean;

import
org.springframework.context.annotation.Configuration;
```

```
@Configuration

public class RabbitConfig {



    @Bean

    public Queue queue() {

        return new Queue("myQueue", false);

    }



    @Bean

    public TopicExchange exchange() {

        return new TopicExchange("myExchange");

    }



    @Bean

    public Binding binding(Queue queue, TopicExchange
exchange) {

        return
BindingBuilder.bind(queue).to(exchange).with("routing.key")
;

    }



    @Bean

    public SimpleMessageListenerContainer
messageListenerContainer(ConnectionFactory
connectionFactory) {

        SimpleMessageListenerContainer container = new
SimpleMessageListenerContainer();
```

```
        container.setConnectionFactory(connectionFactory);

        container.setQueueNames("myQueue");

        container.setMessageListener(new ChannelAwareMessageListener() {

            @Override

            public void onMessage(Message message, Channel channel) throws Exception {

                // Process message

            }

        });

        return container;

    }

}
```

4. Send and Receive Messages:

   – Use `RabbitTemplate` to send messages and define message listeners to receive messages.

   – Example sender:

```
import org.springframework.amqp.core.Message;

import org.springframework.amqp.rabbit.core.RabbitTemplate;

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

import org.springframework.stereotype.Service;



@Service

public class MessageSender {
    @Autowired

    private RabbitTemplate rabbitTemplate;



    public void sendMessage(String message) {

        rabbitTemplate.convertAndSend("myExchange",
"routing.key", message);

    }

}
```

– Example receiver:

```
import
org.springframework.amqp.rabbit.annotation.RabbitListener;

import org.springframework.stereotype.Component;



@Component

public class MessageReceiver {



    @RabbitListener(queues = "myQueue")

    public void receiveMessage(String message) {

        // Process received message

    }

}
```

Summary:

- Use Spring Boot with RabbitMQ by adding dependencies, configuring connection properties, defining queues, exchanges, and bindings, and implementing message producers and consumers.

Q - 82) What is the role of Spring Cloud Gateway in a Spring Boot microservices architecture?

Spring Cloud Gateway is a powerful, flexible API gateway that provides a single entry point for routing requests to various microservices. It is used to handle cross-cutting concerns such as routing, filtering, and load balancing in a microservices architecture.

Key Roles:

1. Routing:
   - Routes incoming HTTP requests to appropriate microservices based on the request path, host, or other criteria.
   - Example:

```
spring:
  cloud:
    gateway:
      routes:
        - id: my-service
          uri: lb://my-service
          predicates:
            - Path=/my-service/**
```

2. Filtering:
   - Apply filters to requests and responses for tasks like authentication, logging, and transformation.
   - Example:

```
spring:
  cloud:
    gateway:
      filters:
```

```
                    - AddRequestHeader=Hello,World
```

3. Load Balancing:
   - Integrate with Spring Cloud LoadBalancer to distribute traffic among multiple instances of a microservice.
4. Security:
   - Implement security features such as authentication and authorization at the gateway level.
5. Rate Limiting:
   - Control the rate of incoming requests to prevent abuse and overload.
6. Dynamic Routing:
   - Support for dynamic routing and service discovery integration with Spring Cloud Eureka or other service registries.

Summary:

- Spring Cloud Gateway serves as an API gateway for routing, filtering, and load balancing in a microservices architecture, providing a unified entry point and handling cross-cutting concerns.

Q - 83) How do you implement logging and monitoring in a Spring Boot microservices architecture?

Logging and monitoring are critical for managing and debugging microservices. Implement them in a Spring Boot microservices architecture using the following tools and techniques:

1. Centralized Logging:
   - Use a centralized logging system like ELK Stack (Elasticsearch, Logstash, Kibana) or Graylog.
   - Configure logback or log4j2 to send logs to Logstash.
   - Example `logback-spring.xml` configuration:

```
<appender name="LOGSTASH"
class="ch.qos.logback.logstash.LogstashTcpSocketAppender">

   <remoteAddress>logstash.example.com</remoteAddress>

    <port>5044</port>

    <encoder>
```

```
        <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level
%logger{36} - %msg%n</pattern>

    </encoder>

</appender>


<root level="INFO">

    <appender-ref ref="LOGSTASH"/>

</root>
```

2. Distributed Tracing:

   – Implement distributed tracing with tools like Spring Cloud Sleuth and Zipkin or Jaeger.

   – Spring Cloud Sleuth provides automatic instrumentation and correlation of requests across microservices.

   – Example configuration:

```
spring:

  sleuth:

    sampler:

      probability: 1.0
```

3. Application Metrics:

   – Use Spring Boot Actuator to expose metrics and health checks.

   – Integrate with monitoring tools like Prometheus and Grafana.

   – Example `application.yml` for Actuator metrics:

```
management:

  endpoints:

    web:

      exposure:
        include: health,info,prometheus
```

4. Health Checks:

   - Implement health checks to monitor the status of your microservices.
   - Example endpoint:

```
import org.springframework.boot.actuate.health.Health;

import
org.springframework.boot.actuate.health.HealthIndicator;

import org.springframework.stereotype.Component;



@Component

public class CustomHealthIndicator implements
HealthIndicator {



    @Override

    public Health health() {

        // Perform health check logic

        return Health.up().withDetail("status", "Service is
up").build();

    }

}
```

5. Alerting:

   - Configure alerts based on log patterns or metrics thresholds using monitoring tools.

Summary:

- Implement logging and monitoring in a Spring Boot microservices architecture using centralized logging systems, distributed tracing, Spring Boot Actuator for metrics and health checks, and integration with monitoring and alerting tools.

Q - 84) How do you use distributed tracing in Spring Boot?

Distributed tracing helps track requests across multiple microservices, providing visibility into request flows and performance bottlenecks. Use Spring Boot with distributed tracing tools like Spring Cloud Sleuth and Zipkin or Jaeger to achieve this.

1. Add Dependencies:

   – Include the `spring-cloud-starter-sleuth` and `spring-cloud-starter-zipkin` dependencies in your `pom.xml` or `build.gradle`.

   – Example for Maven:

```
<dependency>

   <groupId>org.springframework.cloud</groupId>

   <artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

<dependency>

   <groupId>org.springframework.cloud</groupId>

   <artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>
```

2. Configure Zipkin or Jaeger:

   – Set up a Zipkin or Jaeger server and configure Spring Boot to send trace data to it.

   – Example configuration for Zipkin:

```
spring:

  zipkin:

    base-url: http://localhost:9411/

  sleuth:

    sampler:

      probability: 1.0
```

3. Automatic Instrumentation:

   – Spring Cloud Sleuth automatically instruments your application, adding trace and span IDs to logs and creating trace records.

4. Custom Tracing:

   – Create custom spans and tags to enrich trace data.

   – Example:

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

import org.springframework.stereotype.Service;

import brave.Tracer;

@Service

public class MyService {

@Autowired

private Tracer tracer;


public void someMethod() {

    // Create a custom span

    try (
Tracer.SpanInScope ws = tracer.withSpanInScope(tracer.nextSpan().name("custom-span").start())) {

        // Business logic

    }

  }

}
```

5. View Traces:

  - Access traces and spans in Zipkin or Jaeger UI to analyze request flows and identify performance issues.

Summary:

- Use distributed tracing in Spring Boot with Spring Cloud Sleuth and Zipkin or Jaeger by adding dependencies, configuring tracing endpoints, and using automatic or custom tracing for visibility across microservices.

Q - 85) What is the purpose of the `@HystrixCommand` annotation?

The `@HystrixCommand` annotation is used in Spring Boot applications to implement circuit breaker patterns for handling service failures and improving fault tolerance. It is part of the Hystrix library, which provides resilience and stability to distributed systems.

Purpose and Usage:

1. Fault Tolerance:
   - `@HystrixCommand` helps to handle failures gracefully by defining fallback methods that are executed when the primary method fails or when the circuit is open.
2. Configuration:
   - Configure the circuit breaker parameters like timeout, circuit breaker threshold, and fallback behavior using `@HystrixCommand` attributes.
3. Example Usage:
   - Annotate a method with `@HystrixCommand` and provide a fallback method to handle failures.
   - Example:

```
import com.netflix.hystrix.HystrixCommandProperties;

import com.netflix.hystrix.HystrixThreadPoolProperties;

import org.springframework.stereotype.Service;

import
com.netflix.hystrix.contrib.javanica.annotation.HystrixComm
and;



@Service

public class MyService {
    @HystrixCommand(fallbackMethod = "fallbackMethod")
    public String riskyMethod() {
        // Code that might fail
        return "Success";
    }

    public String fallbackMethod() {
        return "Fallback response";
    }
}
```

4. Metrics and Monitoring:
   - Hystrix provides metrics and monitoring capabilities to observe the state of the circuit breaker, such as the number of successful and failed requests, circuit status, and execution time.
5. Isolation:
   - Hystrix allows you to isolate calls to external services or components, preventing failures from cascading and affecting other parts of the system.

Summary:

- The `@HystrixCommand` annotation in Spring Boot is used to implement circuit breakers, providing fault tolerance by handling failures with fallback methods and isolating failures to improve system stability and resilience.

Q - 86) How do you use the Spring Retry module in Spring Boot?

The Spring Retry module provides capabilities for automatically retrying operations that can fail intermittently. To use Spring Retry in a Spring Boot application:

1. Add Dependencies:
   - Include `spring-retry` and `spring-boot-starter-aop` dependencies in your `pom.xml` or `build.gradle`.

– Example for Maven:

```
<dependency>
   <groupId>org.springframework.retry</groupId>
   <artifactId>spring-retry</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

2. Enable Spring Retry:

– Use the `@EnableRetry` annotation in your Spring Boot application configuration to enable retry functionality.

– Example:

```
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplicatio
n;
import org.springframework.retry.annotation.EnableRetry;


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

3. Annotate Methods for Retry:

   – Use the `@Retryable` annotation on methods that you want to automatically retry upon failure.

   – Example:

```
import org.springframework.retry.annotation.Backoff;

import org.springframework.retry.annotation.Recover;

import org.springframework.retry.annotation.Retryable;

import org.springframework.stereotype.Service;



@Service

public class MyService {



    @Retryable(value = { RuntimeException.class },
maxAttempts = 3, backoff = @Backoff(delay = 2000))

    public String retryableMethod() {

        // Code that may fail

        return "Success";

    }



    @Recover

    public String recover(RuntimeException e) {

        return "Recovery method";

    }
}
```

4. Configuration Options:

   – Configure retry policies and backoff strategies in `application.properties` or `application.yml`.

   – Example:

```yaml
spring:
  retry:
    maxAttempts: 5
    backoff:
      initialInterval: 1000
      maxInterval: 5000
      multiplier: 1.5
```

Summary:

- Use the Spring Retry module by adding dependencies, enabling retry with `@EnableRetry`, annotating methods with `@Retryable`, and configuring retry policies. It provides automatic retry capabilities and recovery mechanisms for transient failures.

Q - 87) How do you secure Spring Boot microservices with OAuth2 and JWT?

To secure Spring Boot microservices with OAuth2 and JWT, follow these steps:

1. Add Dependencies:

   – Include the `spring-boot-starter-oauth2-resource-server` and `spring-boot-starter-security` dependencies in your `pom.xml` or `build.gradle`.

   – Example for Maven:

```xml
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-oauth2-resource-
```

```
server</artifactId>

</dependency>

<dependency>

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

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

</dependency>
```

2. Configure JWT Security:

   – Configure your application to use JWT for authentication and authorization by setting up JWT properties and security configuration.

   – Example `application.yml`:

```
spring:

  security:

    oauth2:

      resourceserver:

        jwt:

          issuer-uri: https://your-auth-server
```

3. Create Security Configuration:

   – Configure Spring Security to use OAuth2 with JWT by extending `WebSecurityConfigurerAdapter` and setting up `JwtAuthenticationConverter`.

   – Example:

```
import org.springframework.context.annotation.Bean;

import
org.springframework.context.annotation.Configuration;

import
org.springframework.security.config.annotation.web.builders
.HttpSecurity;
```

```
import
org.springframework.security.config.annotation.web.configur
ation.WebSecurityConfigurerAdapter;

import
org.springframework.security.config.annotation.web.builders
.WebSecurity;

import
org.springframework.security.web.authentication.UsernamePas
swordAuthenticationFilter;



@Configuration

public class SecurityConfig extends
WebSecurityConfigurerAdapter {



    @Override

    protected void configure(HttpSecurity http) throws
Exception {

        http

            .authorizeRequests()

            .anyRequest().authenticated()

            .and()

            .oauth2ResourceServer().jwt();

    }

}
```

4. Generate and Validate JWTs:
   - Ensure your OAuth2 Authorization Server generates and signs JWTs. Configure the public key or issuer URI in your Spring Boot application to validate incoming JWTs.
5. Testing:

- Test your security configuration by making authenticated requests with JWTs to ensure proper access control and protection.

Summary:

- Secure Spring Boot microservices with OAuth2 and JWT by adding dependencies, configuring JWT settings, creating security configurations, and ensuring proper JWT generation and validation. This setup provides secure authentication and authorization for your microservices.

Q - 88) How do you implement a custom authentication mechanism in Spring Boot?

To implement a custom authentication mechanism in Spring Boot, you can create a custom `AuthenticationProvider` or `UserDetailsService` and configure Spring Security to use it. Here's a step-by-step guide:

1. Add Dependencies:
   - Ensure you have `spring-boot-starter-security` in your `pom.xml` or `build.gradle`.
   - Example for Maven:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

2. Create a Custom UserDetails Service:
   - Implement `UserDetailsService` to load user-specific data.
   - Example:

```
import
org.springframework.security.core.userdetails.UserDetails;

import
org.springframework.security.core.userdetails.UserDetailsSe
rvice;

import
org.springframework.security.core.userdetails.UsernameNotFo
undException;
```

```
import org.springframework.stereotype.Service;



@Service

public class CustomUserDetailsService implements
UserDetailsService {



    @Override

    public UserDetails loadUserByUsername(String username)
throws UsernameNotFoundException {

        // Retrieve user details from your data source

        // Example: User user =
userRepository.findByUsername(username);

        // return new
org.springframework.security.core.userdetails.User(user.get
Username(), user.getPassword(), new ArrayList<>());

        throw new UsernameNotFoundException("User not
found");

    }

}
```

3. Create a Custom Authentication Provider:

   – Implement `AuthenticationProvider` if you need more control over the authentication process.

   – Example:

```
import
org.springframework.security.authentication.AuthenticationP
rovider;

import org.springframework.security.core.Authentication;

import
org.springframework.security.core.AuthenticationException;
```

```
import org.springframework.stereotype.Component;


@Component

public class CustomAuthenticationProvider implements
AuthenticationProvider {


    @Override

    public Authentication authenticate(Authentication
authentication) throws AuthenticationException {

        // Implement custom authentication logic

        // Example: Check user credentials and return a
valid Authentication object

        return null;

    }


    @Override

    public boolean supports(Class<?> authentication) {

        return true;

    }

}
```

4. Configure Spring Security:

   - Extend `WebSecurityConfigurerAdapter` to use your custom authentication provider or user details service.
   - Example:

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

```java
import org.springframework.context.annotation.Bean;

import
org.springframework.context.annotation.Configuration;

import
org.springframework.security.config.annotation.web.builders
.HttpSecurity;

import
org.springframework.security.config.annotation.web.builders
.AuthenticationManagerBuilder;

import
org.springframework.security.config.annotation.web.configur
ation.WebSecurityConfigurerAdapter;

import
org.springframework.security.core.userdetails.UserDetailsSe
rvice;



@Configuration

public class SecurityConfig extends
WebSecurityConfigurerAdapter {



    @Autowired

    private CustomUserDetailsService
customUserDetailsService;



    @Autowired

    private CustomAuthenticationProvider
customAuthenticationProvider;



    @Override

    protected void configure(AuthenticationManagerBuilder
        auth) throws Exception {

        auth.authenticationProvider(customAuthenticationProvider);
    }

    @Override
    protected void configure(HttpSecurity http) throws
        Exception {
        http
            .authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin();
    }
}
```

5. Test Your Custom Authentication:
   - Ensure that your custom authentication mechanism works as expected by testing login and access control.

Summary:

- Implement a custom authentication mechanism in Spring Boot by creating a custom `UserDetailsService` or `AuthenticationProvider`, configuring Spring Security to use it, and testing the custom authentication logic.

Q - 89) What is Spring Boot's reactive programming support?

Spring Boot's reactive programming support allows developers to build scalable, non-blocking, and asynchronous applications using reactive programming paradigms. It leverages the Project Reactor library, which provides a set of reactive types (e.g., `Mono` and `Flux`) to handle data streams and asynchronous operations.

Key Aspects:

1. Reactive Types:

   - `Mono`: Represents a single value or an empty result.
   - `Flux`: Represents a stream of values.

2. Non-blocking Operations:
   - Reactive programming enables non-blocking I/O operations, improving scalability and performance, especially for applications with high concurrency and I/O-bound tasks.
3. Integration with Spring WebFlux:
   - Spring WebFlux is a reactive stack that supports reactive controllers, handlers, and web clients for building reactive web applications.

Example:

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

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

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

import reactor.core.publisher.Flux;


@RestController

@RequestMapping("/api")

public class ReactiveController {


    @GetMapping("/data")

    public Flux<String> getData() {

        return Flux.just("Value1", "Value2", "Value3");

    }

}
```

4. Reactive Data Access:

- Spring Data Reactive repositories enable reactive data access with databases such as MongoDB and Cassandra.
- Example:

```
import
org.springframework.data.repository.reactive.ReactiveCrudRe
pository;

import reactor.core.publisher.Flux;



public interface MyReactiveRepository extends
ReactiveCrudRepository<MyEntity, String> {

    Flux<MyEntity> findByName(String name);

}
```

5. Reactive Web Client:

- Spring WebFlux provides a `WebClient` for making asynchronous HTTP requests.
- Example:

```
import
org.springframework.web.reactive.function.client.WebClient;

import reactor.core.publisher.Mono;



@Service

public class MyService {



    private final WebClient webClient;



    public MyService(WebClient.Builder webClientBuilder) {

        this.webClient =
webClientBuilder.baseUrl("http://example.com").build();
    }

    public Mono<String> getData() {
        return webClient.get()
                       .uri("/data")
                        .retrieve()
                       .bodyToMono(String.class);
    }
}
```

Summary:

- Spring Boot's reactive programming support enables building non-blocking, scalable applications using Project Reactor. It integrates with Spring WebFlux for reactive web applications and provides reactive data access and web client capabilities.

Q - 90) How do you use WebFlux in Spring Boot?

To use Spring WebFlux in Spring Boot, follow these steps:

1. Add Dependencies:
   - Include `spring-boot-starter-webflux` dependency in your `pom.xml` or `build.gradle`.
   - Example for Maven:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

2. Create Reactive Controllers:
   - Use `@RestController` or `@Controller` to define reactive endpoints, returning `Mono` or `Flux`.

- Example:

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

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

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

import reactor.core.publisher.Flux;



@RestController

@RequestMapping("/api")

public class ReactiveController {



    @GetMapping("/data")

    public Flux<String> getData() {

        return Flux.just("Value1", "Value2", "Value3");

    }

}
```

3. Configure Reactive Data Access:
   - Use reactive repositories with databases that support reactive operations, such as MongoDB or Cassandra.
   - Example:

```
import
org.springframework.data.repository.reactive.ReactiveCrudRe
pository;

import reactor.core.publisher.Flux;
```

```
public interface MyReactiveRepository extends
ReactiveCrudRepository<MyEntity, String> {

    Flux<MyEntity> findByName(String name);

}
```

4. Use Reactive WebClient:

   - Make asynchronous HTTP requests with `WebClient`, which replaces `RestTemplate` in reactive applications.
   - Example:

```
import
org.springframework.web.reactive.function.client.WebClient;

import reactor.core.publisher.Mono;



@Service

public class MyService {



    private final WebClient webClient;



    public MyService(WebClient.Builder webClientBuilder) {

        this.webClient =
webClientBuilder.baseUrl("http://example.com").build();

    }



    public Mono<String> getData() {

        return webClient.get()

                       .uri("/data")

                        .retrieve()
```

```
                                    .bodyToMono(String.class);
                }
            }
```

5. Run and Test:
   - Run your Spring Boot application and test the reactive endpoints and data access to ensure they work as expected.

Summary:

- To use WebFlux in Spring Boot, add the `spring-boot-starter-webflux` dependency, create reactive controllers and data access, and use `WebClient` for asynchronous HTTP requests. WebFlux provides a reactive programming model for building non-blocking web applications.

Q - 91) What is the difference between blocking and non-blocking I/O in Spring Boot?

Blocking I/O:

- Definition: In blocking I/O, a thread is blocked until the I/O operation is completed. During this time, the thread is not available to handle other tasks.
- Behavior: When a thread performs a blocking I/O operation (e.g., reading from a file or waiting for a network response), it waits for the operation to finish before it can continue.
- Scalability: Blocking I/O can be less scalable because each thread handles one I/O operation at a time. This can lead to a high number of threads and increased resource consumption.

Non-Blocking I/O:

- Definition: Non-blocking I/O allows a thread to continue executing other tasks while waiting for an I/O operation to complete. The thread does not wait idly.
- Behavior: In non-blocking I/O, a thread performs an I/O operation and can check periodically or be notified when the operation is complete. This approach is more efficient for handling multiple I/O operations concurrently.
- Scalability: Non-blocking I/O is more scalable because threads are not tied up waiting for I/O operations. Instead, threads can handle multiple I/O operations with fewer resources.

Example in Spring Boot:

- Blocking I/O: Traditional Spring MVC applications use blocking I/O by default.
- Non-Blocking I/O: Spring WebFlux uses non-blocking I/O, allowing for better scalability and responsiveness by using reactive types like `Mono` and `Flux`.

Summary:

- Blocking I/O ties up threads while waiting for operations to complete, potentially leading to resource limitations. Non-blocking I/O allows threads to continue working on other tasks while waiting for I/O operations, improving scalability and efficiency.

Q - 92) How do you handle backpressure in reactive programming with Spring Boot?

Handling Backpressure:

Backpressure in reactive programming is a mechanism to handle scenarios where a data producer emits items faster than a consumer can process them. In Spring Boot's reactive programming, you can handle backpressure by:

1. Using Reactive Types:
   - Flux: A publisher that can emit multiple items.
   - Mono: A publisher that emits a single item or none.
2. Operators for Backpressure:
   - `onBackpressureBuffer()`: Buffers items when the downstream consumer is slower than the upstream producer.

```
import reactor.core.publisher.Flux;



Flux.range(1, 1000)

    .onBackpressureBuffer()

    .subscribe(System.out::println);
```

   - `onBackpressureDrop()`: Drops items when the downstream consumer cannot keep up.

```
Flux.range(1, 1000)

    .onBackpressureDrop()
    .subscribe(System.out::println);
```

– `onBackpressureLatest()`: Keeps only the latest item when the downstream is slow.

```
Flux.range(1, 1000)
    .onBackpressureLatest()
    .subscribe(System.out::println);
```

3. Flow Control:

   – `limitRate()`: Controls the rate of items emitted to the subscriber.

```
Flux.range(1, 1000)
    .limitRate(10)
    .subscribe(System.out::println);
```

4. Custom Strategies:

   – Implement custom backpressure handling by defining how items should be processed when the consumer cannot keep up.

Summary:

- Handle backpressure in reactive programming by using operators like `onBackpressureBuffer()`, `onBackpressureDrop()`, and `onBackpressureLatest()`, and by controlling the rate of items with operators like `limitRate()`. These mechanisms ensure smooth data flow and prevent overwhelming consumers.

Q - 93) How do you configure and use Spring Boot with GraphQL?

To configure and use Spring Boot with GraphQL, follow these steps:

1. Add Dependencies:

   – Include `spring-boot-starter-graphql` and `graphql-java-kickstart` dependencies in your `pom.xml` or `build.gradle`.

   – Example for Maven:

```
<dependency>
```

```
    <groupId>com.graphql-java-kickstart</groupId>

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

</dependency>
```

2. Define GraphQL Schema:

   – Create a GraphQL schema file (`schema.graphqls`) in the `src/main/resources` directory.

   – Example:

```
type Query {

    hello: String

}
```

3. Create Data Fetchers/Resolvers:

   – Implement GraphQL resolvers to provide data for queries and mutations.

   – Example:

```
import
org.springframework.graphql.data.method.annotation.QueryMap
ping;

import org.springframework.stereotype.Component;



@Component

public class QueryResolver {



    @QueryMapping

    public String hello() {

        return "Hello, world!";

    }
}
```

4. Configure GraphQL:
   - Configure GraphQL settings if needed (e.g., GraphQL endpoint, security).
   - Example in `application.properties`:

     ```
     spring.graphql.path=/graphql
     ```

5. Testing:
   - Use tools like GraphiQL or Postman to test GraphQL queries and mutations.

Summary:

- Configure Spring Boot with GraphQL by adding dependencies, defining a GraphQL schema, creating resolvers, and configuring settings. This setup enables you to build and expose GraphQL APIs in your Spring Boot application.

Q - 94) How do you implement role-based access control in Spring Boot?

To implement role-based access control (RBAC) in Spring Boot, follow these steps:

1. Add Dependencies:
   - Ensure you have `spring-boot-starter-security` in your `pom.xml` or `build.gradle`.
   - Example for Maven:

     ```
     <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-security</artifactId>
     </dependency>
     ```

2. Define Roles and Authorities:
   - Create roles and authorities in your database or application configuration.
3. Configure Security:
   - Extend `WebSecurityConfigurerAdapter` and configure role-based access using `antMatchers()` or `authorizeRequests()`.
   - Example:

```
import
org.springframework.context.annotation.Configuration;

import
org.springframework.security.config.annotation.web.builders
.HttpSecurity;

import
org.springframework.security.config.annotation.web.configur
ation.WebSecurityConfigurerAdapter;



@Configuration

public class SecurityConfig extends
WebSecurityConfigurerAdapter {



    @Override

    protected void configure(HttpSecurity http) throws
Exception {

        http

            .authorizeRequests()

           .antMatchers("/admin/**").hasRole("ADMIN")

           .antMatchers("/user/**").hasAnyRole("USER",
"ADMIN")

            .anyRequest().authenticated()

            .and()

            .formLogin();

    }

}
```

4. Assign Roles to Users:

– Use Spring Security's UserDetails and GrantedAuthority to manage roles for users.

– Example:

```
import org.springframework.security.core.GrantedAuthority;

import
org.springframework.security.core.userdetails.UserDetails;



public class MyUserDetails implements UserDetails {

    private String username;

    private String password;

    private Collection<? extends GrantedAuthority>
authorities;



    // Implement required methods

}
```

5. Testing:

   – Test role-based access control by accessing protected resources with different user roles.

Summary:

• Implement role-based access control in Spring Boot by configuring security settings, defining roles and authorities, and using UserDetails to manage user roles. Configure access rules to restrict or grant access to resources based on user roles.

Q - 95) How do you use Spring Boot with NoSQL databases like MongoDB?

To use Spring Boot with NoSQL databases like MongoDB, follow these steps:

1. Add Dependencies:

   – Include spring-boot-starter-data-mongodb dependency in your pom.xml or build.gradle.

   – Example for Maven:

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

<artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

2. Configure MongoDB:

   – Configure MongoDB connection settings in `application.properties` or `application.yml`.

   – Example:

```
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=mydatabase
```

3. Create MongoDB Entities:

   – Define your MongoDB documents using `@Document` annotation.

   – Example:

```
import org.springframework.data.annotation.Id;
import
org.springframework.data.mongodb.core.mapping.Document;


@Document(collection = "users")
public class User {
    @Id
    private String id;
    private String name;
    private String email;
    // Getters and Setters

}
```

4. Create Repositories:

   – Use `MongoRepository` or `ReactiveMongoRepository` to create repositories for CRUD operations.

   – Example:

```
import
org.springframework.data.mongodb.repository.MongoRepository
;



public interface UserRepository extends
MongoRepository<User, String> {

    User findByName(String name);

}
```

5. Using MongoDB Operations:

   – Inject and use repositories

to perform operations on MongoDB documents.

• Example:

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

import org.springframework.stereotype.Service;



@Service

public class UserService {



    @Autowired
```

```
        private UserRepository userRepository;


        public User getUserByName(String name) {
            return userRepository.findByName(name);
        }
    }
```

Summary:

- Use Spring Boot with MongoDB by adding the appropriate dependency, configuring MongoDB connection, defining entities with `@Document`, and creating repositories. Spring Data MongoDB simplifies interactions with MongoDB, allowing you to perform CRUD operations and manage documents efficiently.

Q - 96) How do you optimize the performance of a Spring Boot application?

To optimize the performance of a Spring Boot application, consider the following practices:

1. Profiling and Monitoring:
   - Use tools like Spring Boot Actuator, Prometheus, and Grafana to monitor and profile your application's performance.
2. Caching:
   - Implement caching to reduce the load on your database and improve response times. Use `@Cacheable`, `@CachePut`, and `@CacheEvict` annotations.
   - Example:

```
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;


@Service
public class MyService {
    @Cacheable("items")
    public Item getItemById(String id) {
        // Method implementation
    }
}
```

3. Database Optimization:
   - Optimize database queries and use appropriate indexing.
   - Use connection pooling (e.g., HikariCP) to manage database connections efficiently.
4. Asynchronous Processing:
   - Use asynchronous methods to handle long-running tasks and improve responsiveness.
   - Example:

```
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Service;

@Service
public class MyService {
    @Async
    public CompletableFuture<Void> asyncTask() {
        // Asynchronous task implementation
    }
}
```

5. Configuration Tuning:
   - Tune JVM parameters and application properties, such as thread pool sizes and timeout settings.
   - Example in `application.properties`:

```properties
server.servlet.context-parameters.max-threads=200
```

6. Resource Management:
   - Optimize memory usage and manage resource allocation efficiently.
7. Code Optimization:
   - Optimize code for performance by minimizing resource-intensive operations and using efficient algorithms.

Summary:

- Optimize Spring Boot performance by profiling and monitoring, implementing caching, optimizing database operations, using asynchronous processing, tuning configurations, managing resources, and optimizing code.

Q - 97) What is the role of `@EnableScheduling` in Spring Boot?

Role of `@EnableScheduling`:

- Purpose: The `@EnableScheduling` annotation is used to enable Spring's scheduled task execution capability. It allows you to run scheduled tasks using annotations like `@Scheduled`.
- Usage: When you annotate a configuration class with `@EnableScheduling`, Spring scans for `@Scheduled` annotations and configures the task scheduler accordingly.

Example:

```
import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.scheduling.annotation.EnableScheduling;

import org.springframework.scheduling.annotation.Scheduled;

import org.springframework.stereotype.Component;


@SpringBootApplication

@EnableScheduling
```

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


@Component
class ScheduledTasks {
    @Scheduled(fixedRate = 5000)
    public void performTask() {
        System.out.println("Task executed every 5 seconds");
    }
}
```

Summary:

- The `@EnableScheduling` annotation enables scheduling of tasks in Spring Boot applications, allowing you to execute tasks at fixed intervals or with cron expressions using the `@Scheduled` annotation.

Q - 98) How do you customize the embedded server in Spring Boot?

To customize the embedded server in Spring Boot, you can modify various server properties and configurations:

1. Application Properties:
   - Use `application.properties` or `application.yml` to set server-specific properties.
   - Example:

```
server.port=8081
server.servlet.context-path=/myapp
server.tomcat.max-connections=200
```

2. Programmatic Configuration:

   – Customize the embedded server programmatically by configuring `EmbeddedServletContainerCustomizer` or `WebServerCustomizer` beans.

   – Example:

```
import
org.springframework.boot.web.server.WebServerFactoryCustomi
zer;

import
org.springframework.boot.web.server.TomcatContextCustomizer
;

import org.springframework.stereotype.Component;



@Component

public class Customizer implements
WebServerFactoryCustomizer<ConfigurableServletWebServerFact
ory> {

    @Override

    public void
customize(ConfigurableServletWebServerFactory factory) {

        factory.setPort(8081);

       factory.setContextPath("/myapp");

    }

}
```

3. Using Specific Server Libraries:

   – If using Tomcat, Jetty, or Undertow, you can customize server-specific settings by including the corresponding starter and configuring properties or beans for the server.

Summary:

- Customize the embedded server in Spring Boot using application properties, programmatic configuration, or specific server libraries. Adjust settings such as port, context path, and server-specific configurations.

Q - 99) How do you implement custom metrics in Spring Boot?

To implement custom metrics in Spring Boot, you can use Micrometer, which is integrated with Spring Boot for metrics collection and monitoring:

1. Add Dependencies:
   - Include `micrometer-core` and any specific registry dependency (e.g., Prometheus) in your `pom.xml` or `build.gradle`.
   - Example for Maven:

```
<dependency>

   <groupId>io.micrometer</groupId>

   <artifactId>micrometer-core</artifactId>

</dependency>

<dependency>

   <groupId>io.micrometer</groupId>

   <artifactId>micrometer-registry-prometheus</artifactId>

</dependency>
```

2. Define Custom Metrics:
   - Use `MeterRegistry` to define and record custom metrics in your service or component.
   - Example:

```
import io.micrometer.core.instrument.MeterRegistry;

import org.springframework.stereotype.Service;



@Service
```

```
public class MyService {

    private final MeterRegistry meterRegistry;

    public MyService(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
    }

    public void process() {
        // Define and record a custom counter
meterRegistry.counter("my_custom_metric").increment();
    }
}
```

3. Expose Metrics Endpoint:
   - Ensure the metrics endpoint is exposed for monitoring tools to scrape metrics.
   - Example in `application.properties`:

     ```properties
management.endpoints.web.exposure.include=*

management.endpoint.metrics.enabled=true

```

Summary:

- Implement custom metrics in Spring Boot using Micrometer by adding the required dependencies, defining custom metrics with `MeterRegistry`, and exposing the metrics endpoint for monitoring tools.

Q - 100) How do you create and use a custom health indicator in Spring Boot Actuator?

To create and use a custom health indicator in Spring Boot Actuator, follow these steps:

1. Add Dependencies:
   - Ensure you have `spring-boot-starter-actuator` in your `pom.xml` or `build.gradle`.
   - Example for Maven:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2. Implement Custom Health Indicator:
   - Implement the `HealthIndicator` interface to create a custom health indicator.
   - Example:

```
import org.springframework.boot.actuate.health.Health;
import
org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;


@Component
public class CustomHealthIndicator implements
HealthIndicator {


    @Override
    public Health health() {
        // Custom logic to determine health status
        boolean isHealthy = checkSomeHealthCondition();
        if (isHealthy) {
            return Health.up().withDetail("custom", 
"Service is healthy").build();
        } else {
            return Health.down().withDetail("custom", 
"Service is not healthy").build();
        }
    }

    private boolean checkSomeHealthCondition() {
        // Custom health check logic
        return true; // Example condition
    }
}
```

3. Access Health Endpoint:
   - Access the `/actuator/health` endpoint to see the status of your custom health indicator.
   - Example in `application.properties`:

```properties
management.endpoints.web.exposure.include=health
```

Summary:

- Create a custom health indicator in Spring Boot Actuator by implementing the `HealthIndicator` interface and defining the health check logic. Ensure the actuator dependency is included and access the health endpoint to view the custom health status.

https://www.youtube.com/@codewitharrays

https://www.instagram.com/codewitharrays/

https://t.me/codewitharrays Group Link: https://t.me/ccee2025notes

+91 8007592194 +91 9284926333

codewitharrays@gmail.com

https://codewitharrays.in/project