NOTES

Spring Boot

CodeForSuccess.in

Spring Boot

- Spring Boot is a powerful framework for building Java-based web applications and microservices.
- It is a module of Spring through which we speed up the development processes using Java.
- It provides a streamlined way to create stand-alone, production-grade Spring-based applications with minimal configuration.
- It also provides a faster & easier way to setup, configure and execute the applications.
- It simplifies the development of Java-based web applications and microservices by providing a powerful, opinionated framework with sensible defaults and streamlined setup.
- It's widely adopted in the industry for its productivity, scalability, and ease of use.
- Spring Boot provides a convention-over-configuration software design style, which means developers don't need to worry about configurations, and their efforts are decreased because all configurations are done automatically.
- Spring boot scans the class path and then find the dependency to configure the project automatically.

Key concepts and features of Spring Boot

- **Convention over Configuration:** Spring Boot follows the principle of convention over configuration, which means it provides sensible defaults and reduces the need for manual configuration. This allows developers to focus more on writing application logic rather than boilerplate setup.
- **Embedded Servers:** Spring Boot includes embedded servers like Tomcat, Jetty, and Undertow, allowing applications to be run as stand-alone JAR files without needing to deploy them to a separate application server.
- **Auto-Configuration:** Spring Boot automatically configures various components based on the dependencies present in the classpath. It scans the project's dependencies and configures beans accordingly, greatly simplifying setup.
- **Starter Dependencies:** Spring Boot provides a wide range of starter dependencies, which are curated sets of dependencies that simplify the process of adding functionality to your application. For example, spring-boot-starter-web includes everything needed to build web applications, including Spring MVC and embedded Tomcat.

- **Actuators:** Spring Boot Actuator provides built-in endpoints that expose information about your application, such as health, metrics, and environment details. This is particularly useful for monitoring and managing applications in production.
- **Spring Boot CLI:** The Spring Boot Command Line Interface (CLI) allows you to quickly prototype and develop Spring Boot applications using Groovy scripts.
- **Spring Initializr:** Spring Initializr is a web-based tool for quickly generating Spring Boot projects with your desired dependencies. It provides a simple interface for customizing project settings and dependencies.
- **Spring Boot DevTools:** DevTools is a set of tools that streamline the development process by providing features like automatic application restarts, live reloading, and remote debugging.
- **Integration with Spring Ecosystem:** Spring Boot seamlessly integrates with other Spring projects like Spring Data, Spring Security, Spring Cloud, etc., making it easy to leverage the full power of the Spring ecosystem.
- **Production-Ready Features:** Spring Boot includes features like health checks, externalized configuration, logging, and profile support, which are essential for building robust, production-ready applications.

Use-Cases

Spring Boot is a versatile framework widely used across various industries and for different types of applications.

Here are some common use cases where Spring Boot shines:

- **Web Applications:** Spring Boot is extensively used for building web applications, ranging from simple websites to complex enterprise-level applications. Its embedded server support and auto-configuration make it easy to create and deploy web applications quickly.
- **Microservices Architecture:** Spring Boot's lightweight nature and ease of development make it well-suited for building microservices-based architectures. Each microservice can be developed as a standalone Spring Boot application, allowing teams to independently develop, deploy, and scale services.
- **RESTful APIs:** Spring Boot provides excellent support for building RESTful APIs. With Spring MVC or Spring WebFlux, developers can quickly create endpoints for handling HTTP requests and responses. Spring Boot's auto-configuration also

simplifies the setup of features like JSON serialization, content negotiation, and error handling.

- **Batch Processing:** Spring Batch, a module of the Spring ecosystem, integrates seamlessly with Spring Boot for building batch processing applications. This is particularly useful for tasks like data extraction, transformation, and loading (ETL), scheduled jobs, and large-scale data processing.
- **Real-time Data Processing:** Spring Boot combined with Spring Integration or Spring Cloud Stream is suitable for building real-time data processing applications. It allows developers to integrate with messaging systems like Apache Kafka or RabbitMQ, enabling the processing of streams of data in real-time.
- **Event-Driven Architectures:** Spring Boot can be used to build event-driven architectures using tools like Spring Cloud Stream or Spring Cloud Event. These enable communication and processing of events between different services or components in a distributed system.
- **Enterprise Applications:** Spring Boot is widely adopted in building enterprise-grade applications due to its robustness, scalability, and integration capabilities. It seamlessly integrates with enterprise technologies like relational databases (using Spring Data JPA), messaging systems, security frameworks (using Spring Security), and more.
- **IoT (Internet of Things) Applications:** Spring Boot can be used to develop IoT applications, leveraging its support for asynchronous programming, messaging systems, and integration with IoT protocols like MQTT. This allows developers to build scalable and reliable IoT solutions.
- **Cloud-Native Applications:** Spring Boot, when combined with Spring Cloud, provides features for building cloud-native applications. This includes service discovery, distributed configuration management, circuit breakers, and centralized logging, which are essential for building resilient and scalable applications in cloud environments.
- **Prototyping and Rapid Development:** Spring Boot's ease of setup, starter dependencies, and auto-configuration make it an excellent choice for prototyping and rapid development of Java-based applications. Developers can quickly get started with building applications without spending time on boilerplate setup and configuration.

These are just a few examples of the many use cases where Spring Boot can be applied effectively. Its versatility, coupled with the extensive features of the Spring ecosystem, makes it a popular choice for a wide range of applications and industries.

Advantages & Disadvantages of Spring Boot

Spring Boot comes with several advantages, making it a popular choice for building Java-based applications. However, it also has some disadvantages that developers should be aware of.

Advantages of Spring Boot:

- **Rapid Development:** Spring Boot provides a streamlined development experience with auto-configuration and starter dependencies, allowing developers to quickly bootstrap projects and focus on application logic rather than infrastructure setup.
- **Convention over Configuration:** Spring Boot follows the principle of convention over configuration, reducing the need for manual configuration by providing sensible defaults. This results in cleaner and more concise code.
- **Embedded Servers:** Spring Boot includes embedded servers like Tomcat, Jetty, or Undertow, enabling applications to be packaged as standalone JAR files without requiring a separate application server installation.
- **Microservices Support:** With its lightweight nature and modularity, Spring Boot is well-suited for building microservices-based architectures. Each microservice can be developed and deployed independently, facilitating scalability and maintainability.
- **Integration with Spring Ecosystem:** Spring Boot seamlessly integrates with other Spring projects like Spring Data, Spring Security, Spring Batch, etc., providing a comprehensive ecosystem for building enterprise-grade applications.
- **Community Support:** Spring Boot has a large and active community of developers, providing abundant resources, tutorials, and support forums. This makes it easier for developers to find solutions to their problems and stay updated with best practices.
- **Production-Ready Features:** Spring Boot offers features like health checks, metrics, centralized logging, and externalized configuration out-of-the-box, making applications production-ready with minimal effort.
- **Cloud-Native Capabilities:** Spring Boot integrates seamlessly with Spring Cloud, providing features for building cloud-native applications, such as service discovery, distributed configuration management, and circuit breakers.
- **Testing Support:** Spring Boot provides support for testing with tools like JUnit and Mockito, enabling developers to write comprehensive unit tests, integration tests, and end-to-end tests for their applications.

- **Security:** Spring Boot offers robust security features through Spring Security, allowing developers to implement authentication, authorization, and other security measures easily.

Disadvantages of Spring Boot

- **Learning Curve:** While Spring Boot simplifies many aspects of application development, it still has a learning curve, especially for beginners who may need to understand underlying Spring concepts and configurations.
- **Overhead:** Spring Boot applications may have a larger memory footprint compared to plain Java applications due to the inclusion of the Spring framework and other dependencies. However, the overhead is often negligible in most scenarios.
- **Complexity with Customization:** While Spring Boot's auto-configuration is convenient, customizing certain behaviors or overriding defaults may require a deeper understanding of Spring internals, which can be challenging for some developers.
- **Opinionated Approach:** Spring Boot follows certain conventions and opinions, which may not always align with the preferences or requirements of all developers or projects. This could lead to friction in certain scenarios.
- **Runtime Performance:** Although Spring Boot applications are generally efficient, there might be some overhead associated with runtime performance due to reflection and dynamic proxy usage, especially in highly performance-sensitive applications.
- **Potential Version Conflicts:** Managing dependencies and versions, especially in larger projects with multiple modules, can sometimes lead to conflicts or compatibility issues, requiring careful dependency management.
- **Limited Control Over Embedded Servers:** While embedded servers provide convenience, developers may have limited control over server configurations compared to standalone servers, which could be a concern in certain scenarios.
- **Increased Startup Time:** Spring Boot applications may have longer startup times compared to traditional Java applications, particularly when dealing with a large number of dependencies or complex auto-configuration.

Overall, while Spring Boot offers numerous advantages for building Java-based applications, developers should carefully evaluate their requirements and consider both the advantages and disadvantages before choosing it for their projects.

Working Flow of Spring Boot

Spring Boot simplifies Java application development by providing sensible defaults, reducing boilerplate code, and streamlining the configuration process, allowing developers to focus more on building their application's business logic.

Let’s go through with the below points to understand the working flow of Spring Boot:

- **Starters and Dependencies:** In a Spring Boot project, you typically start by adding dependencies to your project build configuration (such as Maven pom.xml or Gradle build.gradle). These dependencies are often provided as "starters" that encapsulate common sets of libraries and configurations for specific tasks, like web development, database access, security, etc.
- **Autoconfiguration:** Once you have defined your dependencies, Spring Boot's autoconfiguration mechanism kicks in. Spring Boot scans the classpath to detect the presence of various libraries and automatically configures the application based on what it finds. This includes setting up default beans, configuring infrastructure components like data sources and servlet containers, and enabling features based on detected dependencies.
- **Embedded Server:** Spring Boot includes an embedded servlet container (such as Tomcat, Jetty, or Undertow) by default. This means you don't need to deploy your application to a separate server; the servlet container is packaged with your application, making it self-contained and easily deployable.
- **Application Properties:** Spring Boot allows you to configure your application using properties files (typically application.properties or application.yml), environment variables, or command-line arguments. These properties can override the default configurations provided by Spring Boot's autoconfiguration, giving you fine-grained control over your application's behavior.
- **Component Scanning and Dependency Injection:** Spring Boot leverages Spring Framework's core features, such as component scanning and dependency injection. Components like controllers, services, repositories, etc., are automatically discovered and wired together using annotations like @RestController, @Service, @Repository, etc.
- **Web Layer:** In a typical web application, requests are handled by controllers, which process incoming requests, perform business logic, and generate responses. Spring Boot simplifies the creation of RESTful APIs or web applications by providing annotations like @RestController, @RequestMapping, @GetMapping, @PostMapping, etc., which map HTTP requests to controller methods.

- **Actuators:** Spring Boot Actuators provide endpoints that expose useful information about your application, such as health checks, metrics, environment details, etc. These endpoints can be accessed via HTTP, making it easy to monitor and manage your application in production.
- **Profile-based Configuration:** Spring Boot supports the concept of profiles, allowing you to define different sets of configurations for different environments (e.g., development, testing, production). You can use profile-specific properties files (application-{profile}.properties) to tailor your application's behavior for each environment.
- **Spring Boot CLI:** Although not always used in production, the Spring Boot Command Line Interface (CLI) provides a convenient way to prototype and develop Spring Boot applications from the command line. It supports features like auto-restarting the application on code changes and quickly generating new projects.

Starter POM

- In Spring Boot, a starter POM (Project Object Model) is a specialized kind of Maven dependency that simplifies the setup of various functionalities in your Spring Boot application.
- These starters are essentially dependency descriptors that pull in all the necessary dependencies, configurations, and transitive dependencies needed to work with a specific feature or technology.
- The starter POM concept in Spring Boot greatly simplifies dependency management and configuration, allowing developers to focus more on building their application's business logic rather than managing infrastructure and dependencies.

Examples of commonly used Spring Boot starters include:

- **spring-boot-starter-web:** for building web applications using Spring MVC.
- **spring-boot-starter-data-jpa:** for working with relational databases using Spring Data JPA.
- **spring-boot-starter-security:** for adding security features to your application.
- **spring-boot-starter-test:** for including testing frameworks and libraries like JUnit, Mockito, etc.

‘spring.factories’ file

- The spring.factories file plays a crucial role in the Spring Framework and Spring Boot applications. It's a resource file used for automatic configuration and component discovery.
- The spring.factories file serves as a mechanism for automatic configuration, component discovery, and extensibility in Spring Boot applications.
- It enables the dynamic registration of configurations, extensions, and autoconfiguration classes, facilitating a more flexible and modular architecture.

Let’s discuss about the explanation of various roles of spring.factories file:

- **Automatic Configuration:** It allows developers and framework authors to declare configuration classes or configuration properties files that should be automatically picked up and processed by the Spring ApplicationContext during application startup.
- **SPI (Service Provider Interface):** The spring.factories file follows the SPI pattern. In Java, the SPI pattern allows service providers to register their implementations dynamically. Similarly, in Spring Boot, framework components and third-party libraries can register their configurations or extensions dynamically using this file.
- **Location:** The spring.factories file is typically located in the META-INF directory of a project's classpath.
- **Format:** It consists of key-value pairs, where the key represents the fully qualified name of an interface or abstract class, and the value is a comma-separated list of fully qualified class names that implement or extend the interface or abstract class.
- **Spring Boot Autoconfiguration:** One of the most common uses of the spring.factories file is for defining autoconfiguration classes in Spring Boot. These autoconfiguration classes are responsible for configuring beans and components based on certain conditions or the presence of specific dependencies. By listing these autoconfiguration classes in the spring.factories file under the org.springframework.boot.autoconfigure.EnableAutoConfiguration key, Spring Boot automatically processes them during application startup.
- **Extensibility:** The spring.factories file allows developers to extend the functionality of Spring Boot or override default configurations by providing their own implementations of certain interfaces or classes. By registering these implementations in the spring.factories file, they can seamlessly integrate with the Spring Boot application context.

Spring Boot Project Creation

There are several ways to create a Spring Boot project, depending on your preferences and the tools you want to use.

Here are some common methods:

1. **Using Maven Project in Eclipse IDE**
   Here's how you can create a Spring Boot project using Maven Project in Eclipse IDE:

- Open Eclipse: Launch Eclipse IDE.
- Create a Maven Project:
    - Go to File -> New -> Project.
    - Select Maven -> Maven Project and click Next.
    - Choose Create a simple project and click Next.
    - Enter the Group Id and Artifact Id for your project and click Finish.
- Add Spring Boot Starter Dependencies:
    - Open the pom.xml file of your Maven project.
    - Inside the <dependencies> section, add the desired Spring Boot starters as dependencies. For example, to include web and data JPA starters, add the following lines:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

    - Save the pom.xml file.
- Update Maven Project: After adding dependencies, Maven needs to update the project to download the required dependencies. Right-click on the project folder, go to Maven -> Update Project. Make sure "Force Update of Snapshots/Releases" is checked and click OK.
- Start Coding: Now you can start writing your Spring Boot application code. You can create Java classes, controllers, repositories, etc., and use the features provided by the added dependencies.

2. **Using Spring Initializr (Recommended)**
   Spring Initializr is a web-based tool provided by the Spring team to bootstrap Spring Boot projects quickly. You can visit the Spring Initializr website and customize your project by selecting the desired dependencies, project metadata, and build system (Maven or Gradle). Once configured, you can download the generated project as a ZIP file and import it into your preferred IDE.

   Here's how you can create a Spring Boot project using Spring Initializr:

- Visit the Spring Initializr Website: Go to the Spring Initializr website.
- Configure Your Project:
    - Choose your project's Project Metadata, including Group, Artifact, and Name.
    - Select the desired Java version for your project.
    - Choose your preferred build tool (Maven or Gradle).
    - Select the Spring Boot version you want to use (usually the latest stable version is recommended).
    - Specify the dependencies you need for your project by clicking on the Add Dependencies button. Common dependencies include:
    - Spring Web for building web applications.
    - Spring Data JPA for accessing relational databases.
    - Spring Security for adding security features.
    - Other dependencies for specific requirements like messaging, caching, etc.
- Generate Your Project:
    - Once you have configured your project settings and dependencies, click on the Generate button.
    - This will generate a ZIP file containing your Spring Boot project with the specified configurations and dependencies.
- Extract and Import the Project:
    - Extract the downloaded ZIP file to a location on your computer.
    - Open your preferred IDE (e.g., IntelliJ IDEA, Eclipse).
    - Import the extracted project into your IDE as an existing Maven or Gradle project, depending on the build tool you selected during project generation.

- Start Coding:
    - Once the project is imported into your IDE, you can start writing your Spring Boot application code.
    - Create controllers, services, repositories, etc., as needed for your application.
    - Configure application properties (application.properties or application.yml) to customize your application settings.
- Run the Application:
    - After writing your application code, you can run the Spring Boot application by executing the main class (typically annotated with @SpringBootApplication).
    - This will start the embedded Tomcat server (or any other embedded server you may have chosen) and deploy your Spring Boot application.

That's it! You've now created a Spring Boot project using Spring Initializr and can start developing your application. Spring Initializr simplifies the project setup process by providing a convenient web interface for configuring project metadata and dependencies.

3. **Using IntelliJ IDEA**

IntelliJ IDEA, being one of the most popular IDEs for Java development, has excellent support for generating Spring Boot applications out of the box. You don't need a separate plugin for this purpose.

Here's how you can generate a Spring Boot application using IntelliJ IDEA:

- Open IntelliJ IDEA:
    - Launch IntelliJ IDEA and make sure you have the necessary plugins installed for Java and Spring Boot development. The Spring Boot support is typically included in the Java or Spring plugins.
- Create a New Project:
    - Go to File -> New -> Project.
    - Choose Spring Initializr from the list of options on the left.
- Configure Project:
    - In the Spring Initializr dialog, you can configure your Spring Boot project:
    - Choose the appropriate JDK.

    - Set the Group and Artifact ID for your project.
    - Choose the Spring Boot version.
    - Select the desired dependencies you want to include in your project. You can choose dependencies like Web, JPA, Security, etc., depending on your project requirements.
- Additional Settings:
    - Specify the project name, location, and other settings as needed.
    - Click Next.
- Create:
    - Click Create to generate the project.
- Import Changes:
    - IntelliJ IDEA will then download the necessary dependencies and set up the project for you automatically. It might take a few moments.
- Start Coding:
    - Once the project is set up, you can start coding your Spring Boot application. IntelliJ IDEA provides extensive support for Spring Boot, including code completion, navigation, debugging, and more.

4. **Using Visual Studio Code (VS Code)**

   To create a Spring Boot project using Visual Studio Code (VS Code), you can use the Spring Initializr extension, which allows you to generate Spring Boot projects directly from within the editor.

   Here's how you can do it:

- Install Visual Studio Code:
    - If you haven't already, download and install Visual Studio Code from the official website.
- Install Spring Initializr Extension:
    - Open Visual Studio Code.
    - Go to the Extensions view by clicking on the square icon on the sidebar or pressing Ctrl+Shift+X.
    - Search for "Spring Initializr" in the Extensions Marketplace.
    - Click Install to install the extension.
- Create a New Spring Boot Project:
    - Once the extension is installed, press Ctrl+Shift+P to open the command palette.

- Type "Spring Initializr: Generate a Maven Project" or "Spring Initializr: Generate a Gradle Project" and press Enter based on your preference for Maven or Gradle.
- Fill in the required information such as Group Id, Artifact Id, and dependencies for your Spring Boot project.
- Choose the version of Spring Boot and other dependencies you want to include.
- Select the project language (Java or Kotlin).
- Choose the project location where you want to save the project files.
- Confirm the settings, and the extension will generate the project for you.

- Open the Project:
  - Once the project is generated, you can open it in Visual Studio Code by selecting File -> Open Folder and navigating to the folder where the project was created.
- Start Coding:
  - You can now start coding your Spring Boot application in Visual Studio Code. The IDE provides features like syntax highlighting, IntelliSense, debugging, and more to enhance your development experience.

5. **Spring Boot CLI**

   Spring Boot CLI (Command Line Interface) allows you to create and run Spring Boot applications from the command line. You can use commands like spring init to generate a new Spring Boot project with the desired dependencies and configurations.

   Here's how you can do it:

- Install Spring CLI:
  - If you haven't already installed the Spring CLI, you can download it from the official Spring website or install it using a package manager like SDKMAN or Homebrew.
    https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html#getting-started.installing.cli

- Open Terminal or Command Prompt:
    - Open your terminal or command prompt on your computer.

- Generate a Spring Boot Project:
    - Use the spring init command followed by the project details and dependencies to generate a Spring Boot project.
      For example:
      Gradle Project
      spring init --dependencies=web,data-jpa my-spring-project
      Maven Project
      spring init --build=maven --dependencies=web,data-jpa my-spring-project
    - Replace my-spring-project with the name of your project. You can specify additional dependencies separated by commas based on your project requirements.
- Customize Project Settings:
    - You can customize the project settings by specifying various options such as the project name, group ID, artifact ID, packaging format, Java version, and more.
- Navigate to the Project Directory:
    - After the project is generated, navigate to the project directory using the cd command:
      cd my-spring-project
- Start Coding:
    - You can now start coding your Spring Boot application. The project structure and necessary dependencies will be set up for you automatically.

The Spring CLI provides a convenient way to quickly generate Spring Boot projects from the command line without the need for an IDE. It's particularly useful for creating projects on the fly or integrating with scripts for automation purposes.

Java Persistence API (JPA)

- Java Persistence API (JPA) is a Java specification for accessing, persisting, and managing data between Java objects and a relational database.
- JPA provides a standardized way to work with object-relational mapping (ORM) in Java applications.
- JPA simplifies the development of data-driven applications in Java by providing a high-level abstraction over database interactions, allowing developers to focus more on application logic rather than low-level database operations.

Key concepts and features of JPA

- **Entity:** In JPA, an entity represents a persistent object that is stored in a database. An entity is typically a Java class that is annotated with @Entity. Each instance of an entity class represents a row in the corresponding database table.

- **EntityManager:** EntityManager is the central interface in JPA for performing CRUD (Create, Read, Update, Delete) operations on entities. It manages the lifecycle of entities, including persisting, querying, and removing them from the database. EntityManagerFactory is used to create EntityManager instances.

- **Persistence Unit:** A persistence unit is a collection of entity classes that are managed together as a group. It's defined in the persistence.xml file, which is a configuration file used to configure JPA settings for an application. The persistence unit specifies the database connection details, entity classes, and other JPA settings.

- **Entity Lifecycle:** Entities in JPA have a lifecycle consisting of several states:
    - **New (Transient):** The entity object is newly created and not yet associated with any persistence context.
    - **Managed:** The entity is associated with an EntityManager and is being managed. Changes made to managed entities are automatically synchronized with the database.
    - **Detached:** The entity was once managed but is no longer associated with any persistence context. Modifications to detached entities are not automatically synchronized with the database.
    - **Removed:** The entity is marked for deletion from the database.

- **Relationships:** JPA supports defining relationships between entities, such as one-to-one, one-to-many, many-to-one, and many-to-many relationships. These relationships are defined using annotations such as @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany.

- **JPQL (Java Persistence Query Language):** JPQL is a query language similar to SQL but operates on entities and their persistent fields rather than database tables and columns. It allows developers to write database queries in a platform-independent manner. JPQL queries are translated to native SQL queries by the JPA provider at runtime.

- **Criteria API:** JPA provides a Criteria API for building dynamic queries using a type-safe and object-oriented approach. Criteria queries are defined programmatically using a set of builder classes and methods provided by the Criteria API. This approach can be more flexible and less error-prone than JPQL for complex queries.

- **Validation:** JPA supports bean validation through annotations such as @NotNull, @Size, @Email, etc. These annotations can be used to enforce constraints on entity attributes, and JPA providers can automatically validate entities based on these annotations before persisting them to the database.

- **Caching:** JPA implementations typically provide caching mechanisms to improve performance by reducing the number of database queries. Entities and query results can be cached in memory to avoid repeated database access for frequently accessed data.

JPA – An ORM Tool?

Yes, JPA (Java Persistence API) is an ORM (Object-Relational Mapping) tool. It provides a standardized way for Java developers to map Java objects to relational database tables and perform database operations using an object-oriented approach.

JPA defines a set of interfaces and annotations that ORM frameworks can implement to provide ORM capabilities in Java applications. These capabilities include mapping Java classes to database tables, defining relationships between entities, executing queries, and managing entity lifecycle.

While JPA itself is a specification and does not provide its own implementation, there are several ORM frameworks that implement the JPA specification, such as Hibernate, EclipseLink, and OpenJPA. Developers can choose the ORM framework that best suits their needs and preferences while still using the standard JPA API for database interactions.

JPA and Hibernate

JPA (Java Persistence API) and Hibernate are closely related technologies in the Java ecosystem, but they serve different purposes and have distinct characteristics.

Let's have a look:

1. JPA (Java Persistence API)
    - JPA is a Java specification that defines a standard interface for ORM (Object-Relational Mapping) frameworks to interact with relational databases in Java applications.
    - It provides a set of interfaces and annotations for mapping Java objects to relational database tables and vice versa.
    - JPA is a specification and does not provide its own implementation. Instead, it serves as a standard API that can be implemented by various ORM frameworks, including Hibernate, EclipseLink, and OpenJPA.
2. Hibernate
    - Hibernate is a popular open-source ORM framework that implements the JPA specification.
    - It provides a powerful and flexible ORM solution for Java applications, allowing developers to map Java objects to relational database tables and perform database operations using an object-oriented approach.
    - Hibernate offers additional features beyond the JPA specification, such as support for caching, lazy loading, inheritance mapping strategies, and native SQL queries.

While Hibernate is a powerful and feature-rich ORM framework, JPA provides standardization, portability, and vendor neutrality, making it a preferred choice for many Java developers, especially in enterprise environments. However, the choice between JPA and Hibernate ultimately depends on the specific requirements and constraints of the project.

Difference between JPA and Hibernate

- **Standardization:** JPA is a standard Java specification, while Hibernate is an ORM framework that implements the JPA specification. Hibernate provides additional features and functionalities beyond the standard JPA specification.

- **Flexibility:** Hibernate offers more flexibility and control over database interactions compared to JPA. Developers can leverage Hibernate-specific features and configurations for advanced use cases.

- **Vendor Neutrality:** JPA promotes vendor neutrality by providing a standard API that can be implemented by different ORM frameworks. Using JPA allows developers to switch between different JPA implementations (such as Hibernate, EclipseLink, or OpenJPA) without changing application code.

- **Learning Curve:** Hibernate has a steeper learning curve compared to JPA, especially for beginners. JPA provides a simpler and more standardized approach to ORM, making it easier for developers to get started with database interactions in Java applications.

Advantages of JPA over Hibernate

- **Standardization:** JPA promotes standardization and vendor neutrality by providing a common API for ORM in Java applications. Using JPA allows applications to be more portable across different JPA implementations.

- **Ecosystem:** JPA has a larger ecosystem compared to Hibernate alone. It is part of the Java EE/Java SE platform, and many Java frameworks and libraries support JPA out of the box.

- **Flexibility:** JPA allows developers to switch between different ORM frameworks or JPA implementations without significant code changes. This flexibility is beneficial for applications that require portability and vendor independence.

- **Compliance:** Using JPA ensures compliance with Java EE standards and best practices. Applications built with JPA are more likely to adhere to industry standards and benefit from community support and contributions.

JPA and Hibernate : Use Cases

- Both JPA and Hibernate are powerful tools for working with relational databases in Java applications.
- JPA and Hibernate are both valuable tools for working with relational databases in Java applications, each with its own strengths and use cases.
- JPA provides standardization and portability, while Hibernate offers advanced features and customization options for complex ORM scenarios.
- The choice between JPA and Hibernate depends on the specific requirements and constraints of the project.

Let's discuss some common use cases for each

Use Cases of JPA

- Enterprise Applications: JPA is widely used in enterprise applications where standardization, portability, and vendor neutrality are important. By using JPA, developers can write database code that is independent of any specific ORM framework, allowing for easier migration and integration with different JPA implementations.
- Java EE/Java SE Applications: JPA is an integral part of the Java EE (Enterprise Edition) platform, making it well-suited for building enterprise applications in Java. It is also commonly used in Java SE (Standard Edition) environments for desktop and standalone applications.
- Microservices Architecture: In microservices architecture, where each service typically has its own database, JPA provides a standardized way to interact with databases across different services. Using JPA allows developers to maintain consistency and portability across microservices.
- Prototyping and Rapid Development: JPA simplifies database access and reduces boilerplate code, making it ideal for prototyping and rapid development of Java applications. Developers can quickly map Java objects to database tables and perform CRUD operations without writing complex SQL queries.
- Integration with Existing ORM Frameworks: JPA can be integrated with existing ORM frameworks that implement the JPA specification, such as Hibernate, EclipseLink, and OpenJPA. This allows developers to leverage the features and capabilities of these ORM frameworks while still using the standard JPA API.

Use Cases of Hibernate

- Complex Mapping and Querying: Hibernate is well-suited for scenarios that require complex object-relational mapping or querying capabilities. It provides advanced features for mapping inheritance hierarchies, associations, and custom data types to database tables.
- Performance Optimization: Hibernate offers various performance optimization techniques, such as caching, lazy loading, and batch processing, to improve the performance of database operations. These features are especially useful in applications with high data volume or complex data access patterns.
- Legacy Systems Integration: Hibernate can be used to integrate Java applications with existing legacy systems or databases. It provides support for mapping legacy database schemas to Java objects and performing database operations using modern ORM techniques.
- Batch Processing and ETL (Extract, Transform, Load): Hibernate's batch processing capabilities make it suitable for ETL (Extract, Transform, Load) tasks and batch processing jobs. Developers can efficiently process large volumes of data and perform bulk database operations using Hibernate's batch processing features.
- Customization and Extensibility: Hibernate offers extensive customization and extensibility options, allowing developers to fine-tune the behavior of the ORM framework according to their specific requirements. Developers can customize entity mappings, query execution plans, and transaction management strategies using Hibernate's configuration options and extension points.

JPA with Spring Boot

JPA (Java Persistence API) with Spring Boot is a powerful combination for building robust and scalable Java applications that interact with relational databases. Spring Boot simplifies the setup and configuration of JPA by providing auto-configuration and various utilities.

How JPA works with Spring Boot

- **Dependency Management:** Start by including the necessary dependencies in your project's build configuration file (pom.xml for Maven or build.gradle for Gradle). Spring Boot offers starter dependencies for JPA integration, which automatically configure JPA along with Hibernate as the default JPA provider.

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

- **Entity Classes:** Define your domain model using entity classes. An entity class represents a table in the database. Each entity is typically annotated with @Entity, and properties are mapped to columns using annotations like @Column, @Id, @GeneratedValue, etc.

```
import javax.persistence.Entity;
import  javax.persistence.GeneratedValue;
import  javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private double price;

    // Getters and setters
}
```

- **Repository Interfaces:** Define repository interfaces extending Spring Data CrudRepository or JPA's JpaRepository or related interfaces. These interfaces provide methods for CRUD operations and querying data.
  JpaRepository is an extension of CrudRepository and provides additional JPA-related methods on top of the basic CRUD operations.
  Both CrudRepository and JpaRepository allow developers to define custom query methods by simply declaring method signatures in the repository interfaces. Spring Data JPA will automatically generate the appropriate SQL queries based on the method names. This saves developers from writing boilerplate code for common database operations.

```
import org.springframework.data.jpa.repository.CrudRepository;

public interface ProductRepository extends CrudRepository<Product, Long> {
    // Additional query methods can be declared here
}
```

or

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface ProductRepository extends JpaRepository<Product, Long>
{
    // Additional query methods can be declared here
}
```

- **Configuration:** Spring Boot automatically configures the DataSource, EntityManagerFactory, and TransactionManager based on default settings. However, you can customize these configurations using application.properties or application.yml files.

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
```

- **Transaction Management:** Spring Boot manages transactions using Spring's @Transactional annotation. Simply annotate your service methods with @Transactional, and Spring will handle transaction operations and management for you.

```
import org.springframework.transaction.annotation.Transactional;
@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

    @Transactional
    public Product saveProduct(Product product) {
        return productRepository.save(product);
    }
    // Other service methods...
}
```

- **Using JPA in Controllers or Services:** Finally, you can inject repository interfaces or entity manager directly into your controllers or services to perform database operations.

```
@RestController
public class ProductController {

    @Autowired
    private ProductService productService;

    @PostMapping("/products")
    public ResponseEntity<Product> createProduct(@RequestBody Product
product) {
        Product savedProduct = productService.saveProduct(product);
        return ResponseEntity.ok(savedProduct);
    }

    // Other controller methods...
}
```

By following above steps, you can easily integrate JPA with Spring Boot to build data-driven applications efficiently, leveraging Spring Boot's auto-configuration and dependency management capabilities. JPA provides a high-level abstraction for working with databases, while Spring Boot simplifies the setup and configuration, making development faster and more manageable.

Example : Creating Entity using CrudRepository

1. First create your Spring Starter Project & make sure the internet is available in your system.
2. Then enter group id, artifact id, version, description and package name. Click next
3. Then select the dependency for Spring Data JPA and MySQL Driver. Click Next and then Finish.
4. Now create an Entity class

Employee.java

```
package com.springboot.example;
import jakarta.persistence.Entity;
import  jakarta.persistence.GeneratedValue;
import  jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
@Entity
public class Employee {
	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	int empId;
	String empName;
	String empCity;
	public int getEmpId() {
		return empId;
	}
	public void setEmpId(int empId) {
		this.empId = empId;
	}
	public String getEmpName() {
		return empName;
	}
	public void setEmpName(String empName) {
		this.empName = empName;
	}
	public String getEmpCity() {
		return empCity;
	}
```

```
	public void setEmpCity(String empCity) {
		this.empCity = empCity;
	}
	public Employee() {
		super();
		// TODO Auto-generated constructor stub
	}
	@Override
	public String toString() {
		return "Employee [empId=" + empId + ", empName=" + empName + ", empCity=" + empCity + "]";
	}
}
```

5. Then creating the Repository interface for providing generic CRUD operations for a specific type

EmployeeRepository.java

```
package com.springboot.example;
import org.springframework.data.repository.CrudRepository;
public interface EmployeeRepository extends CrudRepository<Employee, Integer>{
}
```

Where Employee is the entity name and Integer is the type of ID or primary key.

6. Defining the configuration

Application.properties

```
spring.application.name=springbootjpa
spring.datasource.name=springbootdb
spring.datasource.url=jdbc:mysql://localhost:3306/springbootdb?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=android
spring.datasource.dbcp2.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
```

7. Finalizing the SpringBootApplication class

SpringbootjpaApplication.java

```
package com.springboot.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class SpringbootjpaApplication {

	public static void main(String[] args) {
		ConfigurableApplicationContext
context=SpringApplication.run(SpringbootjpaApplication.class, args);
		EmployeeRepository
employeeRepository=context.getBean(EmployeeRepository.class);

		Employee employee=new Employee();
		employee.setEmpName("Aviral Pratap Singh");
		employee.setEmpCity("Lucknow");

		Employee emp=employeeRepository.save(employee);
		System.out.println(emp);
	}

}
```

8. Execute the project.

Note: make sure that MySQL server should be available in your computer and must have database available, named “springbootdb”.

Example: Inserting multiple records in MySQL Table

SpringbootjpaApplication.java

```
package com.springboot.example;
import java.util.ArrayList;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;
@SpringBootApplication
public class SpringbootjpaApplication {
	public static void main(String[] args) {
		ConfigurableApplicationContext
context=SpringApplication.run(SpringbootjpaApplication.class, args);
		EmployeeRepository
employeeRepository=context.getBean(EmployeeRepository.class);

		Employee employee1=new Employee();
		employee1.setEmpName("Garima Choudhary");
		employee1.setEmpCity("Jaipur");
		Employee employee2=new Employee();
		employee2.setEmpName("Shweta Sharma");
		employee2.setEmpCity("Agra"); Employee
		employee3=new Employee();
		employee3.setEmpName("Vikas Sharma");
		employee3.setEmpCity("Noida");

		ArrayList<Employee> employees=new ArrayList<Employee>();
		employees.add(employee1);
		employees.add(employee2);
		employees.add(employee3);
		employeeRepository.saveAll(employees);
	}
}
```

Employee.java, EmployeeRepository.java, application.properties

Remains same

Example : Reading or Finding records from MySQL Table

SpringbootjpaApplication.java

```
package com.springboot.example;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;
@SpringBootApplication
public class SpringbootjpaApplication {

	public static void main(String[] args) {
		ConfigurableApplicationContext
context=SpringApplication.run(SpringbootjpaApplication.class, args);
		EmployeeRepository
employeeRepository=context.getBean(EmployeeRepository.class);

		Iterable<Employee> employees=employeeRepository.findAll();
		System.out.println("available records in the database : ");
		for(Employee employee:employees)
			System.out.println(employee);
	}
}
```

To find a specific record use the following code snippet:

```
Optional<Employee> employeeOptional = employeeRepository.findById(4);
	if (employeeOptional.isPresent()) {
		Employee employee = employeeOptional.get();
		System.out.println(employee);
	} else {
		System.out.println("No Employee exist of the given ID");
	}
```

Where, Optional is a container object that may or may not contain a non-null value. It is a part of the Java 8 API introduced to handle situations where a method may or may not return a value.

Example : Updating the Record in MySQL Table

SpringbootjpaApplication.java

```
package com.springboot.example;

import java.util.Optional;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class SpringbootjpaApplication {

	public static void main(String[] args) {
		ConfigurableApplicationContext
context=SpringApplication.run(SpringbootjpaApplication.class, args);
		EmployeeRepository
employeeRepository=context.getBean(EmployeeRepository.class);

		Optional<Employee>		employeeOptional		=
employeeRepository.findById(4);
		Employee employee = employeeOptional.get();
		employee.setEmpName("Karan Singh");
		employeeRepository.save(employee);
		System.out.println("updated employee details are : "+employee);
	}
}
```

If you want to perform delete operation then you can use:

```
	employeeRepository.delete(employee);	//to delete a single employee record
or
	employeeRepository.deleteAll()		// to delete all the employee records
```

Finder Methods

- In Spring Boot, finder methods refer to methods defined in Spring Data repositories that automatically generate queries based on the method name.
- These methods are used to query entities from the underlying data source (such as a database) without having to write custom queries explicitly.
- Spring Data repositories support various query creation strategies, including method name conventions, @Query annotation, and Querydsl.

How you can use finder methods in Spring Boot

- **Method Name Query Creation**
  Spring Data JPA repositories provide query methods based on the method name. By following a specific naming convention, Spring Data can derive queries from method names.
  For example, if you have an entity User with a field username, you can define a method in your repository interface like this:

```
public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(String username);
}
```

  This method findByUsername(String username) will automatically generate a query to find a User entity by its username field.

- **Using Query Annotation**
  You can also define custom queries using the @Query annotation. This allows you to write JPQL (Java Persistence Query Language) or native SQL queries directly in your repository interface.
  For example:

```
public interface UserRepository extends JpaRepository<User, Long> {
        @Query("SELECT u FROM User u WHERE u.age > :age")
        List<User> findByAgeGreaterThan(@Param("age") int age);
}
```

  Here, the findByAgeGreaterThan method uses a JPQL query to find users with an age greater than a specified value.

- **Querydsl**
  Querydsl is a framework that allows you to write type-safe queries in Java. Spring Data JPA integrates with Querydsl, enabling you to use Querydsl predicates in repository methods.
  For example:

```
public interface UserRepository extends JpaRepository<User, Long>,
        QuerydslPredicateExecutor<User> {
            List<User> findAll(Predicate predicate);
}
```

  With this setup, you can create complex queries using Querydsl predicates and pass them to the findAll method.

By leveraging these finder methods, you can easily define and execute queries without writing boilerplate SQL code, making your codebase more readable and maintainable.

Example : Method Name Query Creation

EmployeeRepository.java

```
package com.springboot.example;
import java.util.ArrayList;
import org.springframework.data.repository.CrudRepository;
public interface EmployeeRepository extends CrudRepository<Employee, Integer>{
        public ArrayList<Employee> findByEmpName(String empName);
}
```

SpringbootjpaApplication.java

```
package com.springboot.example;

import java.util.ArrayList;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class SpringbootjpaApplication {
```

```
	public static void main(String[] args) {
		ConfigurableApplicationContext
context=SpringApplication.run(SpringbootjpaApplication.class, args);
		EmployeeRepository
employeeRepository=context.getBean(EmployeeRepository.class);

		String empName="Shweta Sharma";
		ArrayList<Employee>
employees=employeeRepository.findByEmpName(empName);
		System.out.println("Available records with name = "+empName);
		for(Employee employee:employees)
			System.out.println(employee);

	}
}
```

Example : Using Query Annotation

EmployeeRepository.java

```
package com.springboot.example;

import java.util.ArrayList;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;

public interface EmployeeRepository extends CrudRepository<Employee, Integer>{
	public ArrayList<Employee> findByEmpName(String empName);
		@Query("SELECT u FROM Employee u WHERE u.empId > :id")
		 ArrayList<Employee> findByEmpIdGreaterThan(@Param("id") int empId);
}
```

SpringbootjpaApplication.java

```
package com.springboot.example;

import java.util.ArrayList;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class SpringbootjpaApplication {

	public static void main(String[] args) {
		ConfigurableApplicationContext
context=SpringApplication.run(SpringbootjpaApplication.class, args);
		EmployeeRepository
employeeRepository=context.getBean(EmployeeRepository.class);

		int empId=1;
		ArrayList<Employee>
employees=employeeRepository.findByEmpIdGreaterThan(empId);
		System.out.println("Available records with Id greater than = "+empId);
		for(Employee employee:employees)
			System.out.println(employee);

	}

}
```

Example : Using Querydsl

Homework

If you want to explore more then kindly go through with the below link:

https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

Example : Connecting Spring Boot with MongoDB

1. First create your Spring Starter Project & make sure that internet is available in your system.
2. Then enter group id, artifact id, version, description and package name. Click next
3. Then select the dependency for Spring Data JPA and Spring Data MongoDB. Click Next and then Finish.
4. Now create an document class

Employee.java

```
package com.springboot.example;

import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection="employee")
public class Employee {

 private int empId;
 private String empName;
private String empCity;
public int getEmpId() {
	return empId;
}
public void setEmpId(int empId) {
	this.empId = empId;
}
public String getEmpName() {
	return empName;
}
public void setEmpName(String empName) {
	this.empName = empName;
}
public String getEmpCity() {
	return empCity;
}
public void setEmpCity(String empCity) {
	this.empCity = empCity;
}
```

```
public Employee() {
		super();
		// TODO Auto-generated constructor stub
}
@Override
public String toString() {
		return "Employee [empId=" + empId + ", empName=" + empName + ", empCity="
+ empCity + "]";
}
}
```

5. Create Repository interface.

EmployeeRepository.java

```
package com.springboot.example;

import org.springframework.data.mongodb.repository.MongoRepository;

public interface EmployeeRepository extends MongoRepository<Employee, Integer>{

}
```

6. Define Spring Boot Application class

SpringbootmongoApplication.java

```
package com.springboot.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;
import org.springframework.context.ConfigurableApplicationContext;

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

```
	public static void main(String[] args) {
		ConfigurableApplicationContext
context=SpringApplication.run(SpringbootmongoApplication.class, args);
		EmployeeRepository
employeeRepository=context.getBean(EmployeeRepository.class);

		Employee employee=new Employee();
		employee.setEmpId(47);
		employee.setEmpName("Aviral Pratap Singh");
		employee.setEmpCity("Lucknow");

		Employee emp=employeeRepository.save(employee);
		System.out.println(emp);
	}
}
```

Note : Spring Boot expects a DataSource configuration by default, but if you're using MongoDB which doesn't require a DataSource then you need to tell Spring Boot not to expect a DataSource configuration. You can do this by excluding DataSourceAutoConfiguration like:
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
By excluding DataSourceAutoConfiguration.class, you're telling Spring Boot not to expect a DataSource configuration.

7. Define configuration in “application.properties” file

```
spring.application.name=springbootmongo
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=empdata
```

8. Now run your application but make sure than the database (named ‘empdata’) must be available in your mongodb database.

Example : Connecting Spring Boot with MongoDB using Spring Web

1. First create your Spring Starter Project & make sure that internet is available in your system.
2. Then enter group id, artifact id, version, description and package name. Click next
3. Then select the dependency for Spring Web and Spring Data MongoDB. Click Next and then Finish.
4. Now create an Entity class (Employee.java – As same as previous)
5. Create Repository interface (EmployeeRepository.java – As same as previous).
6. Create Controller

AppController.java

```
package com.springboot.example;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
@RequestMapping("/employee")
public class AppController {
	@Autowired
	private EmployeeRepository employeeRepository;
	@PostMapping("/")
	public ResponseEntity<?> addEmployee(@RequestBody Employee employee)
	{
		Employee emp=this.employeeRepository.save(employee);
		return ResponseEntity.ok(emp);
	}
	@GetMapping("/")
	public ResponseEntity<?> viewEmployees()
	{
		return ResponseEntity.ok(this.employeeRepository.findAll());
	}
}
```

7. Define configuration in “application.properties” file

```
spring.application.name=springbootmongo
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=empdata
```

8. Now run your application and use postman for adding & getting the employee details.

For adding the details you need to pass the following JSON data in the request body:

```
{
   "empId":"12",
   "empName":"Shweta Sharma",
   "empCity":"Jaipur"
}
```

You can download Postman using link https://www.postman.com/

Using Postman

Step 1: Install Postman
you can download it from the official website:
https://www.postman.com/downloads/

Step 2: Start Postman and Create a New Request

- Open Postman after installation.
- Click on the "New" button in the top left corner to create a new request.
- Choose the HTTP method you want to use (e.g., GET or POST).
- Enter the URL of your API endpoint in the address bar.

Step 3: Send Requests and View Responses

- Testing GET Request
    - Select the GET method.
    - Enter the URL for retrieving all employees, for example: http://localhost:3000/employees.
    - Click the "Send" button to send the request.
    - Postman will display the response from your server, showing the list of employees.

- Testing POST Request
    - Select the POST method.
    - Enter the URL for adding a new employee, for example: http://localhost:3000/employees.
    - In the "Body" tab, select the "raw" option.
    - Choose JSON from the dropdown menu.
    - Enter the JSON data for a new employee.

For example:

```
{
  "name": "emp_name",
  "city": "emp_city"
}
```

Click the "Send" button to send the request.
Postman will display the response from your server, showing the newly added employee.

Trouble with Postman?
If you're having trouble writing JSON data in Postman, then go through with the below process:

- **Selecting the Request Method:** Ensure that you've selected the correct request method (e.g., POST) from the dropdown menu next to the request URL.
- **Entering the Request URL:** Enter the URL for the API endpoint you want to send the request to. For example, if you're testing a POST request to add a new employee, the URL might be something like http://localhost:3000/employees.
- **Choosing the Body Type:** In Postman, switch to the "Body" tab located below the request URL input field.
- **Selecting the Body Format:** In the "Body" tab, select the format of the data you're sending. Since you're sending JSON data, choose the "raw" option.
- **Choosing JSON Format:** Once you've selected "raw," a dropdown menu will appear next to it. Click on the dropdown menu and choose "JSON."
- **Writing JSON Data:** Now, you can write your JSON data in the text area provided below.

For example, if you're adding a new employee, your JSON data might look like this:

```
{
  "name": "emp_name",
  "city": "emp_city"
}
```

- **Sending the Request:** After entering the JSON data, click the "Send" button to send the request.

By following the above process steps, you will be able to write JSON data in Postman and send requests to your backend API for testing.

Example : Creating Employee and viewing data using JSP

1. First create your Spring Starter Project & make sure that internet is available in your system.
2. Then enter group id, artifact id, version, description and package name. Click next
3. Then select the dependency for Spring Web and Spring Data MongoDB. Click Next and then Finish.
4. Now, include the tomcat-embed-jasper dependency to can handle JSP pages in your Spring Boot project (pom.xml) using Maven:

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
    <scope>provided</scope>
</dependency>
```

   Also include the JPA dependency:

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

5. Now create an Entity class (Employee.java – As same as previous)
6. Create Repository interface (EmployeeRepository.java – As same as previous).
7. Create Controller

AppController.java

```
package com.springboot.example;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
```

```
@Controller
@RequestMapping("/employee")
public class AppController {

    @Autowired
    private EmployeeRepository employeeRepository;

    @PostMapping("/add")
    public String addEmployee(@RequestParam int id, @RequestParam String name,
@RequestParam String city, Model model) {
        Employee employee = new Employee();
        employee.setEmpId(id);
        employee.setEmpName(name);
        employee.setEmpCity(city);
        employeeRepository.save(employee);
        model.addAttribute("message", "Employee added successfully!");
        return "addEmployee"; // Return addEmployee.jsp
    }
    @GetMapping("/add")
    public String showAddEmployeeForm() {
        return "addEmployee"; // Return addEmployee.jsp
    }
    @GetMapping("/view")
    public String viewEmployees(Model model) {
        List<Employee> employees = employeeRepository.findAll();
        model.addAttribute("employees", employees);
        return "viewEmployees"; // Return viewEmployees.jsp
    }
}
```

8. Now, modifying the project facets then select the Dynamic Web Module and click on “Apply & Close” for getting the project structure according to web requirements.
9. Create a folder with name ‘views’ inside src/main/webapp/WEB-INF to create/store the jsp pages.
10. Use the given codes for creating jsp pages, SpringBootApplication class and configuring project through application.properties file.

addEmployee.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8" %>
<%@ page isELIgnored="false" %>
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Add Employee</title>
</head>
<body>
    <h2>Add Employee</h2>
    <form action="/employee/add" method="post">
            <label for="id">Employee ID:</label><br>
        <input type="text" id="id" name="id"><br>
        <label for="name">Employee Name:</label><br>
        <input type="text" id="name" name="name"><br>
        <label for="city">Employee City:</label><br>
        <input type="text" id="city" name="city"><br><br>
        <input type="submit" value="Submit">
    </form>
    <p>${message}</p> <!-- Display success or error message -->
</body>
</html>
```

viewEmployees.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ page isELIgnored="false" %>
<%@ page import="com.springboot.example.Employee,java.util.List" %>
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>View Employees</title>
</head>
```

```
<body>
   <h2>Employees List</h2>
   <table border="1">
      <thead>
         <tr>
            <th>ID</th>
            <th>Name</th>
            <th>City</th>
         </tr>
      </thead>
      <tbody>
         <%
         List<Employee> employees = (List<Employee>)
request.getAttribute("employees");
         if (employees != null) {
            for (Employee employee : employees) {
         %>
         <tr>
            <td><%= employee.getEmpId() %></td>
            <td><%= employee.getEmpName() %></td>
            <td><%= employee.getEmpCity() %></td>
         </tr>
         <%
            }          }
         %>
      </tbody>
   </table>
</body>
</html>
```

Application.properties

```
spring.application.name=springbootmongo
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=empdata
spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp
```

SpringbootmongoApplication.java

```
package com.springboot.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;

@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
public class SpringbootmongoApplication {

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

}
```

Now, run your springbootapplication class and execute the project functionality.

Application Programming Interface

- API (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with each other.
- It defines the methods and data formats that applications can use to request and exchange information.
- APIs are commonly used in web development to enable communication between a web server and client-side applications.

Concept of API

APIs allow developers to access the functionality of a system or service without having to understand its internal workings. They provide a level of abstraction that simplifies the development process and promotes code reusability. APIs can be used to perform a wide range of tasks, such as retrieving data from a database, sending emails, or interacting with external services.

Types of APIs

- **Web APIs (HTTP APIs)**
  - These APIs are accessed over the internet using the HTTP protocol.
  - They are commonly used to enable communication between web servers and client-side applications.
  - Web APIs are often used to retrieve or manipulate data, such as fetching weather information, sending messages, or accessing a database.

  Example: The OpenWeatherMap API allows developers to retrieve weather data for a specific location by sending an HTTP request to their API endpoint.

- **Library APIs**
  - These APIs are provided by software libraries and allow developers to access the functionality of the library in their own code.
  - Library APIs are used to perform specific tasks, such as manipulating images, parsing JSON data, or encrypting data.

  Example: The axios library in Node.js provides an API for making HTTP requests. Developers can use the axios.get() method to send a GET request to a server and retrieve data.

- **Operating System APIs**
    - These APIs are provided by operating systems and allow developers to access system resources, such as files, processes, and network connections.
    - Operating system APIs are used to perform low-level tasks that require interaction with the underlying operating system.

    Example: The Windows API provides a set of functions that allow developers to create and manage windows, handle user input, and perform other tasks related to the Windows operating system.

Common types of Web API's

There are several types of APIs, each with its own set of characteristics and use cases. The choice of API type depends on the specific requirements of your application, including the desired level of complexity, scalability, and performance.

Some of the most common types include:

- **RESTful API (Representational State Transfer)**
    - REST is an architectural style for designing networked applications. RESTful APIs are designed to be stateless and utilize HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources.
    - They typically use JSON or XML as the data format. RESTful APIs are widely used for web and mobile applications due to their simplicity and scalability.

- **SOAP API (Simple Object Access Protocol)**
    - SOAP is a protocol for exchanging structured information in the implementation of web services.
    - SOAP APIs use XML as the data format and typically require a more complex setup compared to RESTful APIs.
    - They are often used in enterprise environments where a high level of security and reliability is required.

- **GraphQL API**
    - GraphQL is a query language for APIs that allows clients to request only the data they need.
    - Unlike RESTful APIs, which expose a fixed set of endpoints, GraphQL APIs have a single endpoint and allow clients to specify the structure of the response.
    - This makes GraphQL APIs more flexible and efficient for fetching data in complex applications.

- **WebSocket API**
    - WebSocket is a communication protocol that provides full-duplex communication channels over a single TCP connection.
    - WebSocket APIs allow for real-time, bidirectional communication between clients and servers, making them ideal for applications that require low-latency updates, such as chat applications and online gaming platforms.

- **RPC API (Remote Procedure Call)**
    - RPC is a protocol that allows a program to execute code on a remote server.
    - RPC APIs enable applications to call functions or procedures on a remote server as if they were local, making them useful for distributed systems and microservices architectures.

- **gRPC (Google Remote Procedure Call)**
    - gRPC is a high-performance RPC framework developed by Google.
    - It uses HTTP/2 for transport and Protocol Buffers as the interface description language.
    - gRPC APIs are often used in microservices architectures and distributed systems where performance and efficiency are critical.

API Use Cases

- **Social Media APIs:** Social media platforms like Facebook, Twitter, and Instagram provide APIs that allow developers to access their platform's features, such as posting updates, retrieving user information, and interacting with friends.

- **Payment Gateway APIs:** Payment gateway services like PayPal, Stripe, and Braintree provide APIs that allow developers to integrate payment processing into their applications. Developers can use these APIs to accept payments, issue refunds, and manage transactions.

- **Maps APIs:** Mapping services like Google Maps and Mapbox provide APIs that allow developers to integrate mapping functionality into their applications. Developers can use these APIs to display maps, get directions, and perform geocoding tasks.

- **Weather APIs:** Weather services like OpenWeatherMap and WeatherAPI provide APIs that allow developers to retrieve weather information for a specific location. Developers can use these APIs to display current weather conditions, forecasts, and historical data.

APIs play a crucial role in modern software development by enabling interoperability between different systems and services. They provide a standardized way for applications to communicate and exchange data, making it easier for developers to build complex and feature-rich applications.

REST API

- REST (Representational State Transfer) is an architectural style for designing networked applications.
- It is commonly used in web services development, particularly in the context of APIs (Application Programming Interfaces) for web-based applications.
- The fundamental idea behind REST is to treat resources as unique entities that can be manipulated using a standard set of operations.
- REST APIs provide a standardized way for systems to communicate over the web, enabling interoperability between different applications and services.

Key concepts of REST API

- **Resources:** In REST, everything is a resource. A resource can be any entity that can be uniquely identified, such as a user, a blog post, or an image. Resources are typically identified by URIs (Uniform Resource Identifiers).

- **HTTP Methods:** RESTful APIs use HTTP methods to perform actions on resources. The most commonly used HTTP methods in RESTful APIs are:

    - **GET:** Retrieves a representation of the resource identified by the URI. It should not have any side-effects and should be idempotent (repeated requests should have the same effect as a single request).

    - **POST:** Creates a new resource. It often involves sending data in the request body, which will be used to create the resource.

    - **PUT:** Updates an existing resource or creates a new resource if it doesn't exist already. The entire resource is replaced with the new representation sent in the request body.

    - **PATCH:** Partially updates an existing resource. Unlike PUT, which replaces the entire resource, PATCH only updates the specified fields of the resource.

    - **DELETE:** Removes the resource identified by the URI.

- **Uniform Interface:** REST APIs should have a uniform interface, meaning that the same set of HTTP methods should be applicable to all resources. This simplifies the API and makes it more predictable.

- **Statelessness:** Each request from a client to the server must contain all the information necessary to understand the request. The server should not store any client state between requests. This makes the system more scalable and reliable.

- **Representation:** Resources are represented in a format such as JSON or XML. Clients and servers communicate by exchanging representations of resources.

- **Hypermedia as the Engine of Application State (HATEOAS):** This is an optional constraint in REST. It means that the API should provide links dynamically within the response for the client to navigate to related resources. This makes the API more self-descriptive and allows clients to interact with the API without prior knowledge of all available endpoints.

Additional aspects of REST APIs

- **Caching:** REST APIs can leverage HTTP caching mechanisms to improve performance and reduce server load. By including appropriate cache-control headers in responses, clients and intermediary caches can store responses and reuse them for subsequent requests, reducing the need for the server to regenerate the same response repeatedly.

- **Versioning:** As APIs evolve over time, it's often necessary to introduce changes that may not be backward compatible. Versioning allows clients to specify which version of the API they wish to interact with, enabling them to adapt to changes at their own pace. Versioning can be done in various ways, such as including the version number in the URI (e.g., /v1/resource) or using custom request headers.

- **Security:** REST APIs need to implement appropriate security measures to protect resources and prevent unauthorized access. This may involve authentication mechanisms such as API keys, OAuth, or JWT (JSON Web Tokens), as well as authorization mechanisms to control what actions users are allowed to perform on resources.

- **Error Handling:** APIs should provide informative error responses to help clients diagnose and resolve issues. This includes using appropriate HTTP status codes (e.g., 4xx for client errors, 5xx for server errors) and including detailed error messages in the response body.

- **Pagination:** When dealing with large collections of resources, it's common for APIs to support pagination to limit the amount of data returned in each response. Pagination parameters (e.g., page and per_page) allow clients to request specific subsets of the data, improving performance and reducing bandwidth usage.

- **Content Negotiation:** REST APIs can support content negotiation to allow clients to specify the desired representation format (e.g., JSON, XML) for responses. This is typically done using the Accept header in the request, and servers can use the Content-Type header in responses to indicate the format of the returned data.

- **Cross-Origin Resource Sharing (CORS):** CORS is a mechanism that allows web browsers to make cross-origin requests to APIs hosted on different domains. APIs can use CORS headers to specify which origins are allowed to access the resources, helping to prevent unauthorized access from malicious websites.

- **Rate Limiting:** To prevent abuse and ensure fair usage of resources, APIs often implement rate-limiting mechanisms to restrict the number of requests that clients can make within a certain time period. Rate limits can be enforced globally or on a per-user basis, and clients are typically notified of rate limit exceeded errors when they exceed their allowed quota.

These additional aspects play crucial roles in designing robust, secure, and efficient REST APIs that meet the needs of both clients and servers.

Spring Boot DevTools

- Spring Boot DevTools is a set of tools designed to improve the development experience when working with Spring Boot applications.
- It provides several features that streamline the development process, enhance productivity, and enable faster iteration cycles.

- To use DevTools in a Spring Boot project, you typically need to include the spring-boot-devtools dependency in your project's pom.xml (if using Maven) or build.gradle (if using Gradle).
- To add DevTools to a Maven-based Spring Boot project, you need to open your project's pom.xml file and Add the spring-boot-devtools dependency within the <dependencies> section:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
</dependency>
```

  Save the pom.xml file.
- Once you've added the dependency, DevTools will be automatically enabled when you run your Spring Boot application in development mode (spring-boot:run). You can then take advantage of features such as automatic restart, live reload, and property defaults to streamline your development workflow.

Advantages of using Spring Boot DevTools

- **Automatic Application Restart:** DevTools monitors the classpath for changes, including changes to Java classes, resources, and static content. When a change is detected, it automatically restarts the application, allowing developers to see their changes instantly without having to manually redeploy the application.
- **Live Reload:** DevTools integrates with browser plugins (e.g., LiveReload) to automatically reload web pages whenever changes are made to static resources (HTML, CSS, JavaScript). This enables developers to see the effects of their changes in real-time without needing to refresh the browser manually.
- **Remote Development:** DevTools supports remote development scenarios where the development environment is separate from the application server. This is useful for debugging applications deployed to remote servers or virtual machines, as DevTools can automatically synchronize changes between the local development environment and the remote server.

- **Property Defaults:** DevTools provides default configurations for common development properties, such as disabling caching, enabling debug logging, and configuring embedded servers. These defaults help ensure consistent behavior across development environments and reduce the need for manual configuration.
- **Developer-Friendly Features:** DevTools includes several developer-friendly features, such as logging improvements, stack trace enhancements, and error page customization. These features make it easier for developers to diagnose issues and troubleshoot problems during development.

Note : You can also customize DevTools behavior by configuring properties in your application.properties or application.yml file.
For example, you can disable specific features or customize the classpath scanning behavior. Refer to the official Spring Boot documentation for more information on configuring and using Spring Boot DevTools Documentation
https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.devtools

Lombok

- Lombok is a library for Java that helps reduce boilerplate code in projects.
- In the context of Spring Boot, Lombok is often used to simplify the creation of Java classes, particularly data classes such as entities, DTOs (Data Transfer Objects), and POJOs (Plain Old Java Objects).

Lombok : key features

- **Annotation-Based:** Lombok provides a set of annotations that you can apply to your Java classes. These annotations instruct the Lombok compiler plugin to generate code during compilation, reducing the need for manual code writing.
- **Reduced Boilerplate Code:** By using Lombok annotations, you can eliminate common repetitive tasks such as writing getters, setters, constructors, equals/hashCode methods, and toString methods. Lombok generates this code for you at compile time, resulting in cleaner and more concise classes.

- **Integration with IDEs:** Lombok seamlessly integrates with popular IDEs such as IntelliJ IDEA, Eclipse, and NetBeans. IDE plugins automatically recognize Lombok annotations and provide code assistance, navigation, and refactoring support as if the code were written manually.
- **Immutable Objects:** Lombok provides annotations such as @Value and @Builder to create immutable objects and builder patterns, respectively. Immutable objects are thread-safe and can simplify concurrent programming.
- **Logging:** Lombok offers annotations like @Slf4j and @Log to automatically generate logger fields in classes, reducing the need to declare logger instances explicitly.
- **Cleaner Code:** By reducing boilerplate code, Lombok helps improve code readability and maintainability. Developers can focus on writing business logic rather than repetitive code structures.

To use Lombok in a Java project, including a Spring Boot application, you'll need to follow a few simple steps:

- **Step 1: Add Lombok Dependency to Your Project**
  If you're using Maven, add the Lombok dependency to your pom.xml:

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.20</version> <!-- Use the latest version -->
    <scope>provided</scope>
</dependency>
```

  If you're using Gradle, add the Lombok dependency to your build.gradle:

```
dependencies {
    compileOnly 'org.projectlombok:lombok:1.18.20' // Use the latest version
    annotationProcessor 'org.projectlombok:lombok:1.18.20' // Use the latest
    version
}
```

- **Step 2: Install Lombok Plugin in Your IDE**
  To enable Lombok support in your IDE (e.g., IntelliJ IDEA, Eclipse, or NetBeans), you'll need to install the Lombok plugin. Instructions for installing the plugin can be found on the Lombok website.
  Go to help>install new software> then install the Lombok plugin using the url:

```
https://projectlombok.org/p2
```

- **Step 3: Start Using Lombok Annotations**
  Once you've added the Lombok dependency and installed the plugin, you can start using Lombok annotations in your Java classes to reduce boilerplate code.

Examples of how to use Lombok annotations:

- **@Data:** Generates getters, setters, equals, hashCode, and toString methods for all fields in the class.

```
import lombok.Data;
@Data
public class User {
    private Long id;
    private String username;
    private String email;
}
```

- **@Getter / @Setter:** Generates getter or setter methods for fields.

```
import lombok.Getter;
import lombok.Setter;
 @Getter @Setter
public class User {
    private Long id;
    private String username;
    private String email;
}
```

- **@NoArgsConstructor / @AllArgsConstructor:** Generates constructors with no arguments or with all arguments for the class.

```
import lombok.NoArgsConstructor;
import lombok.AllArgsConstructor;
@NoArgsConstructor
@AllArgsConstructor
public class User {
    private Long id;
    private String username;
    private String email;       }
```

- **@Builder:** Generates a builder pattern for the class, allowing for fluent object creation.

```
import lombok.Builder;
import lombok.Data;
@Data
@Builder
public class User {
    private Long id;
    private String username;
    private String email;
}
```

- **@Slf4j:** Generates a logger field for the class using SLF4J.

```
import lombok.extern.slf4j.Slf4j;
@Slf4j
public class MyClass {
    public void doSomething() {
        log.info("Doing something...");
    }
}
```

Class Assignment : Create REST API's for Update & Delete Operation.

Example : Create Rest API with SpringBoot, JPA and MySQL

1. First create a spring starter project with three dependencies – Spring Web, Spring Data JPA and MySQL Driver.
2. Add the Lombok Dependency and Spring DevTools Dependency to your project pom.xml file

```
<dependency>
        <groupId>org.projectlombok</groupId>
         <artifactId>lombok</artifactId>
        <version>1.18.20</version> <!-- Use the latest version -->
        <scope>provided</scope>
</dependency>

<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
</dependency>
```

3. Define the Model or Entity

Employee.java

```
package com.springbootapi.example;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import lombok.Data;

@Entity
@Data
public class Employee {
        @Id
        private int empId;
        private String empName;
        private String empDept;
        private String empCity;
}
```

4. Define the Repository

EmployeeRepository.java

```
package com.springbootapi.example;

import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepository extends JpaRepository<Employee,
        Integer>{

}
```

5. Define the Service Layer or Business Logic

EmployeeService.java

```
package com.springbootapi.example;

import java.util.List;

import org.springframework.stereotype.Service;

@Service
public class EmployeeService {
        EmployeeRepository employeeRepository;
        public EmployeeService(EmployeeRepository employeeRepository)
        {
                this.employeeRepository=employeeRepository;
        }
                public List<Employee> getAllEmployee()
        {
                return employeeRepository.findAll();
        }
                public Employee getEmployee(int empId)
        {
                return employeeRepository.findById(empId).get();
        }
```

```
	public String addEmployee(Employee emp)
	{
		employeeRepository.save(emp);
		return "Employee Details Added Successfully";
	}
	public String updateEmployee(Employee emp)
	{
		employeeRepository.save(emp);
		return "Employee Details Updated Successfully";
	}
	public String deleteEmployee(int empId)
	{
		employeeRepository.deleteById(empId);
		return "Employee Details Deleted Successfully";
	}
}
```

6. Define the Controller

AppController.java

```
package com.springbootapi.example;

import java.util.List;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class AppController {
```

```
EmployeeService employeeService;

public AppController(EmployeeService employeeService) {
	this.employeeService=employeeService;
}

@GetMapping("/employee")
public List<Employee> getAllEmployeeDetails()
{
	return employeeService.getAllEmployee();
}
@GetMapping("/employee/{empId}")
public Employee getEmployeeDetails(@PathVariable("empId") int
empId)
{
	return employeeService.getEmployee(empId);
}

@PostMapping("/employee")
public String addEmployeeDetails(@RequestBody Employee
employee)
{
	String message=employeeService.addEmployee(employee);
	return message;

}

@PutMapping("/employee")
public String updateEmployeeDetails(@RequestBody Employee
employee)
{
	String
message=employeeService.updateEmployee(employee);
	return message;

}
```

```
	@DeleteMapping("/employee/{empId}")
	public String deleteEmployeeDetails(@PathVariable("empId") int
	empId)
	{
		String message=employeeService.deleteEmployee(empId);
		return message;

	}
}
```

7. Define the configuration in application.properties

```
spring.application.name=springbootapi
spring.datasource.name=springbootdb
spring.datasource.url=jdbc:mysql://localhost:3306/springbootdb?serverTi
mezone=UTC
spring.datasource.username=root
spring.datasource.password=android
spring.datasource.dbcp2.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
```

8. SpringBootApplication Class

```
package com.springbootapi.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

To Check the API Functionality, you need to use postman and make sure that the database with name “springbootdb” should be available at your MySQL Server.

- To Add Employee Details, you need to use url http://localhost:8080/api/employee with method “POST” and pass the following data in request body:

```
{
        "empId":"123", "empName":"Neha
        Rathore", "empDept":"front end
        development",
        "empCity":"Ghaziabad"
}
```

- To view the details of all the employees, you need to use the url http://localhost:8080/api/employee with method “GET”
- To view the details of any specific employee, you need to use the url http://localhost:8080/api/employee/{empId} with method “GET”
- To update the existing Employee Details (with empId as primary key), you need to use url http://localhost:8080/api/employee with method “PUT” and pass the following data in request body:

```
{
        "empId":"123", "empName":"Neha
        Rathore", "empDept":"back end
        development",
        "empCity":"Ghaziabad"
}
```

- To delete the details of any specific employee, you need to use the url http://localhost:8080/api/employee/{empId} with method “DELETE”

Exception handling in Spring Boot

- Exception handling in Spring Boot follows a robust and customizable approach to manage errors and exceptions that occur during the execution of an application.
- By effectively managing exceptions, Spring Boot applications can maintain stability, provide a better user experience, and simplify the debugging process for developers.
- It ensures graceful handling of errors, providing meaningful responses to users and preventing application crashes.

Terms & Concepts

- **Global Exception Handling:** Spring Boot provides mechanisms to handle exceptions globally across the application. This means you can define a centralized place to catch and process exceptions that occur anywhere in your application.
- **@ControllerAdvice:** Spring Boot leverages the @ControllerAdvice annotation to define global exception handling. Classes annotated with @ControllerAdvice can contain methods to handle exceptions thrown by controllers within the application.
- **Exception Handling Methods:** Within classes annotated with @ControllerAdvice, you can define methods annotated with @ExceptionHandler. These methods take the specific exception types as parameters and define how to handle those exceptions.
- **ResponseEntity:** Exception handling methods typically return ResponseEntity objects, allowing you to customize the HTTP response sent to the client. This includes setting the HTTP status code, response body, headers, etc.
- **Custom Exception Classes:** Spring Boot allows you to define custom exception classes representing different error scenarios in your application. These custom exceptions can extend from RuntimeException or any other appropriate superclass.
- **Custom Error Responses:** You can define custom error response payloads to be returned to clients when exceptions occur. These responses can include error codes, messages, timestamps, and any other relevant information for debugging or user feedback.
- **Default Error Handling:** Spring Boot also provides default error handling mechanisms for common exceptions like 404 (Not Found), 500 (Internal Server Error), etc. These can be customized to provide application-specific error messages or behaviors.

- **Logging:** Proper logging of exceptions is crucial for debugging and monitoring applications. Spring Boot integrates seamlessly with logging frameworks like Logback or Log4j, allowing you to log exception details at different severity levels.
- **Testing Exception Handling:** Just like any other component of your application, it's essential to test exception handling mechanisms. Spring Boot provides tools and utilities for testing exception handling logic to ensure that exceptions are handled correctly under different scenarios.

Implementing exception handling in Spring Boot

Implementing exception handling in Spring Boot involves several steps:

- **Define Custom Exception Classes:** Create custom exception classes to represent different error scenarios in your application. These classes can extend from RuntimeException or any other appropriate superclass.

```
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}
```

- **Define Custom Error Response Payload:** Create a custom class to represent the error response payload. This class will contain details such as error code, message, timestamp, etc.

```
public class ErrorResponse {
    private HttpStatus status;
    private String message;
    private LocalDateTime timestamp;

    // Constructor, getters, and setters
}
```

- **Create Exception Handling Classes:** Create classes annotated with @ControllerAdvice to handle exceptions globally or for specific controllers. Define methods annotated with @ExceptionHandler to handle specific exceptions.

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<Object>
handleResourceNotFoundException(ResourceNotFoundException ex) {
        // Create custom error response
        ErrorResponse errorResponse = new
ErrorResponse(HttpStatus.NOT_FOUND, ex.getMessage(),
LocalDateTime.now());
        return new ResponseEntity<>(errorResponse,
HttpStatus.NOT_FOUND);
    }
    @ExceptionHandler(Exception.class)
    public ResponseEntity<Object> handleGlobalException(Exception ex) {
        // Create custom error response for other exceptions
        ErrorResponse errorResponse = new
ErrorResponse(HttpStatus.INTERNAL_SERVER_ERROR, ex.getMessage(),
LocalDateTime.now());
        return new ResponseEntity<>(errorResponse,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

- **Use Exception Handling in Controllers:** In your controller methods, throw custom exceptions when necessary.

```
@RestController
@RequestMapping("/api")
public class MyController {
    @GetMapping("/resource/{id}")
    public ResponseEntity<Object> getResource(@PathVariable Long id) {
        // Logic to fetch resource by id
        Resource resource = resourceService.getResourceById(id);
        if (resource == null) {
            throw new ResourceNotFoundException("Resource with id " + id +
" not found");
        }
        return new ResponseEntity<>(resource, HttpStatus.OK);
    }}
```

- **Logging:** Ensure that exceptions are properly logged for debugging and monitoring purposes.

```
@ControllerAdvice
public class GlobalExceptionHandler {

    private static final Logger logger =
LoggerFactory.getLogger(GlobalExceptionHandler.class);

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<Object>
handleResourceNotFoundException(ResourceNotFoundException ex) {

 // Log the exception
        logger.error("Resource not found exception: {}", ex.getMessage());

// Create custom error response
        ErrorResponse errorResponse = new
ErrorResponse(HttpStatus.NOT_FOUND, ex.getMessage(),
LocalDateTime.now());

return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }

    // Other exception handlers...
}
```

With these steps, you can effectively implement exception handling in your Spring Boot application, ensuring graceful error handling and appropriate responses to clients.

Example : Implementing exception handling in a Spring Boot application for an Employee entity

1. **Create a New Spring Boot Project:**
   Open Spring Tool Suite and navigate to File -> New -> Spring Starter Project.
   Enter the project details such as Name, Group, and Artifact, then click Next.
   Select the required dependencies such as Spring Web and Spring Data JPA (if you plan to use a database), and click Finish.

2. **Add Employee Entity Class**

Employee.java

```
package com.example.demo;
import lombok.Data;
@Data
public class Employee {
   private int empId;
   private String empName;
   private String empCity;

   public Employee(int empId, String empName, String empCity) {
            super();
            this.empId = empId;
            this.empName = empName;
            this.empCity = empCity;
      }
}
```

3. **Add Custom Exception Class**
   Create a new Java class named EmployeeNotFoundException which extends RuntimeException.
   Custom Exception Class – EmployeeNotFoundException.java

```
package com.example.demo;

public class EmployeeNotFoundException extends RuntimeException {
   public EmployeeNotFoundException(String message) {
      super(message);
   }
}
```

4. **Create Controller Class**

Create a new Java class named EmployeeController annotated with @RestController. Implement controller methods to handle HTTP requests related to employee data.

Controller – EmployeeController.java

```
package com.example.demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @GetMapping("/{id}")
    public ResponseEntity<Employee> getEmployeeById(@PathVariable int
id) {
        Employee employee = employeeService.getEmployeeById(id);
        if (employee == null) {
            throw new EmployeeNotFoundException("Employee with ID " + id
+ " not found");
        }
        return new ResponseEntity<>(employee, HttpStatus.OK);
    }
}
```

5. **Create Service Class**

Create a new Java class named EmployeeService annotated with @Service. Implement service methods to handle business logic related to employee data.

Service – EmployeeService.java

```
package com.example.demo;

import java.util.ArrayList;
import java.util.List;

import org.springframework.stereotype.Service;

@Service
public class EmployeeService {

    // Mock data for demonstration
    private static final List<Employee> employees = new ArrayList<>();

    static {
        employees.add(new Employee(1, "Sumita Garg", "Jaipur"));
        employees.add(new Employee(2, "Tushar Gupta", "Ghaziabad"));
    }

    public Employee getEmployeeById(int id) {

        for (Employee employee : employees) {

            if (employee.getEmpId() == id) {
                return employee;
            }
        }
        return null; // Employee not found
    }
}
```

6. **Global Exception Handler**

   Create a new Java class annotated with @ControllerAdvice to handle global exceptions. Implement methods annotated with @ExceptionHandler to handle specific exceptions.

Global Exception Handler – GlobalExceptionHandler.java

```
package com.example.demo;

import java.time.LocalDateTime;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class GlobalExceptionHandler {

   @ExceptionHandler(EmployeeNotFoundException.class)
   public ResponseEntity<Object>
handleEmployeeNotFoundException(EmployeeNotFoundException ex) {

      ErrorResponse errorResponse = new
ErrorResponse(HttpStatus.NOT_FOUND, ex.getMessage(),
LocalDateTime.now());

      return new ResponseEntity<>(errorResponse,
HttpStatus.NOT_FOUND);
   }

   @ExceptionHandler(Exception.class)
   public ResponseEntity<Object> handleGlobalException(Exception ex) {

      ErrorResponse errorResponse = new
ErrorResponse(HttpStatus.INTERNAL_SERVER_ERROR, ex.getMessage(),
LocalDateTime.now());

      return new ResponseEntity<>(errorResponse,
HttpStatus.INTERNAL_SERVER_ERROR);
   }
}
```

7. **Error Response Class**

   Create a new Java class to represent the error response payload.

   Error Response Class – ErrorResponse.java

```
package com.example.demo;

import java.time.LocalDateTime;

import org.springframework.http.HttpStatus;

import lombok.Data;

@Data
public class ErrorResponse {
   private HttpStatus status;
   private String message;
   private LocalDateTime timestamp;

        public ErrorResponse(HttpStatus status, String message,
LocalDateTime timestamp) {
                super();
                this.status = status;
                this.message = message;
                this.timestamp = timestamp;
        }

}
```

8. **Application Properties**

   Configure the application properties such as server port, database configuration (if required), etc., in the application.properties file.

Response Handling in Spring Boot

- In Spring Boot, response handling refers to the process of managing and shaping HTTP responses sent back to clients from your application.
- This includes formatting the response data, status codes, setting appropriate HTTP headers, handling errors, and customizing the response based on various conditions.
- By leveraging controllers, interceptors, filters, and advice, you can customize the behavior of your application's responses to meet specific requirements and provide a better experience for clients.

Response Handling in Spring Boot – Key Concepts

- **Controller Methods**
    - In Spring MVC, controllers are responsible for handling HTTP requests and generating HTTP responses.
    - Controller methods typically return objects that represent the data to be sent back in the response.
    - Spring Boot automatically serializes these objects into the appropriate format (e.g., JSON, XML) based on the Accept header in the request.
    - You can annotate controller methods with @ResponseBody to indicate that the return value should be serialized and included in the response body.

- **HTTP Status Codes**
    - Controllers can set the HTTP status code of the response using the @ResponseStatus annotation or by returning a ResponseEntity with the desired status code.
    - Spring Boot provides constants (e.g., HttpStatus.OK, HttpStatus.NOT_FOUND) for commonly used status codes.

- **Error Handling**
    - Spring Boot offers several mechanisms for handling errors and exceptions that occur during request processing.
    - You can define global exception handlers using @ControllerAdvice annotated classes or specific exception handlers within individual controllers.
    - Error handlers can return custom error responses, log errors, or perform other actions based on the type of error.

- **Interceptors**
    - Interceptors in Spring MVC allow you to intercept and modify both requests and responses before they reach the controller or after they leave the controller.
    - You can create custom interceptors by implementing the HandlerInterceptor interface and registering them using the WebMvcConfigurer interface.
    - Interceptors provide a centralized location for common response handling tasks such as adding headers, logging, or modifying the response body.

- **Filters**
    - Filters are another mechanism for intercepting and modifying HTTP requests and responses.
    - Unlike interceptors, filters operate at a lower level in the servlet container and can intercept requests and responses before they reach the Spring MVC dispatcher servlet.
    - Filters are ideal for tasks such as content transformation, request/response logging, or security enforcement.

- **Custom Response Formats**
    - Spring Boot allows you to customize the format of HTTP responses to meet specific requirements.
    - You can create custom response DTOs (Data Transfer Objects) and use them to encapsulate response data along with metadata such as status codes and messages.
    - By implementing custom serialization/deserialization logic or using libraries like Jackson, you can control how response objects are converted to JSON, XML, or other formats.

- **Content Negotiation**
    - Spring Boot supports content negotiation, allowing clients to specify the desired representation format (e.g., JSON, XML) using the Accept header.
    - You can configure content negotiation settings in Spring Boot to choose the appropriate response format based on the client's preferences.

- **HTTP Headers**
  - HTTP headers provide additional metadata about the response, such as content type, caching directives, and security policies.
  - Controllers, interceptors, and filters can set HTTP headers in the response using methods provided by the HttpServletResponse object or using convenience methods provided by Spring Boot.

Example : JSON Response Handling using Spring Boot REST API

Employee.java

```
package com.springbootapi.example;
import lombok.Data;
@Data
public class Employee {
 private int empId;
 private String empName;
 private String empCity;
 public Employee(int empId, String empName, String empCity) {
                super();
                this.empId = empId;
                this.empName = empName;
                this.empCity = empCity;
}
}
```

EmployeeService.java

```
package com.springbootapi.example;
import java.util.ArrayList;
import java.util.List;
import org.springframework.stereotype.Service;
@Service
public class EmployeeService {
 // Mock data for demonstration
 private static final List<Employee> employees = new ArrayList<>();
        static {
        employees.add(new Employee(1, "Sumita Garg", "Jaipur"));
        employees.add(new Employee(2, "Tushar Gupta", "Ghaziabad"));      }
```

```
        public Employee getEmployeeById(int id) {
            for (Employee employee : employees) {
                if (employee.getEmpId() == id) {
                    return employee;
                }
            }
            return null; // Employee not found
        }
    }
```

AppController.java

```
    package com.springbootapi.example;
    import org.springframework.beans.factory.annotation.Autowired;
    import org.springframework.http.HttpStatus;
    import org.springframework.http.ResponseEntity;
    import org.springframework.web.bind.annotation.GetMapping;
    import org.springframework.web.bind.annotation.PathVariable;
    import org.springframework.web.bind.annotation.RequestMapping;
    import org.springframework.web.bind.annotation.RestController;
    @RestController
    @RequestMapping("/employee")
    public class AppController {
        @Autowired
        private EmployeeService employeeService;

        @GetMapping("/{id}")
        public ResponseEntity<Object> getEmployeeById(@PathVariable int id) {
            return ResponseHandler.responseBuilder("Employee details has
    been fetched",
                    HttpStatus.OK, employeeService.getEmployeeById(id)
    );
        }
    }
```

Now, you can check the response using PostMan.

Upload Static File using REST API

1. First create project with dependency spring web, Lombok & spring dev tools.
2. Then define the controller

AppController.java

```
package com.springbootapi.example;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;

@RestController
public class AppController {

	public static final String
dirPathToUpload="C:\\Users\\Administrator\\Documents\\workspace-
spring-tool-suite-4-
4.22.0.RELEASE\\springbootappp\\src\\main\\resources\\static";

	@PostMapping("/upload")
	public ResponseEntity<String> fileUpload(@RequestParam("file")
MultipartFile file ){
		if(file.isEmpty()) {
			return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("No file
is available to upload");
		}
		boolean status=uploadFile(file);
		if(status==true)
```

```
                return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("file
uploading failed");
            else
                return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("file
uploading failed");
        }

        public boolean uploadFile(MultipartFile file)
        {
            boolean status=false;

            try {
                Files.copy(file.getInputStream(),
Paths.get(dirPathToUpload+File.separator+file.getOriginalFilename()),Stan
dardCopyOption.REPLACE_EXISTING);
                status=true;
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            return status;
        }
}
```

3. Using postman to upload / pass the file, send request & check the response

   To pass a file using Postman as a request, you can use the "form-data" or "binary" mode in the body of the request. Here's how to do it:

   a) Open Postman: Launch the Postman application.
   b) Create a New Request: Either create a new request or open an existing one where you want to pass a file.
   c) Select Appropriate Request Type: Make sure you're using a request type that supports file uploads, such as POST or PUT.
   d) Select Body Tab: In the request builder area, select the "Body" tab.

e) Choose Form-Data or Binary:
    a. If you're uploading a file along with other form fields, select "form-data".
    b. If you're only uploading a file without any other form fields, select "binary".
f) Add Key-Value Pair:
    a. For "form-data": Click on "Add" to add a new key-value pair. Set the key to the desired name for the file parameter.
    b. For "binary": Click on "Select Files" to choose the file you want to upload.
g) Select File:
    a. If you chose "form-data", set the value of the key to the file you want to upload by clicking on "Choose Files" and selecting the file from your system.
    b. If you chose "binary", directly select the file by clicking on "Select Files" and choosing the file from your system.
h) Send Request: Once you have added the file, click the "Send" button to send the request.

Postman will then send the request to the specified endpoint with the file attached. Make sure your server-side application is configured to handle file uploads appropriately.

Home Assignment – Need to Submit on 2nd April 2024

1. CRUD Operations:

a. Create a Spring Boot application to manage a simple todo list.

b. Implement functionality to add, retrieve, update, and delete todo items using CRUD operations.

2. REST API Development:

a. Extend the todo list application to expose RESTful APIs for CRUD operations.

b. Implement endpoints for adding, retrieving, updating, and deleting todo items using RESTful principles.

3. Exception Handling:

a. Introduce error handling in the todo list application to deal with common exceptions like ResourceNotFoundException and MethodArgumentNotValidException.

b. Implement custom exception handling to return appropriate error responses with meaningful error messages.

Junit4

- JUnit 4 is a widely used testing framework for Java applications. It provides a simple and effective way to write and execute automated tests for Java code.
- JUnit 4 is designed to support various testing scenarios, including unit tests, integration tests, and more, making it suitable for testing all aspects of Java applications.
- JUnit 4 provides a comprehensive and flexible framework for writing and executing automated tests in Java applications.
- By using its features such as annotations, assertions, parameterized tests, exception testing, and test suites, developers can ensure the reliability and correctness of their Java code through rigorous testing practices.

Key features of JUnit 4

1. **Test Annotations:** JUnit 4 uses annotations to mark methods as test methods and to configure the test environment.
   Example

```
import org.junit.*;

public class MyTest {

   @BeforeClass
   public static void setUpClass() {
      // Code to run once before any test methods in the class
   }
   @AfterClass
   public static void tearDownClass() {
      // Code to run once after all test methods in the class
   }
   @Before
   public void setUp() {
      // Code to run before each test method
   }
   @After
   public void tearDown() {
      // Code to run after each test method
   }
```

```
    @Test
    public void testAddition() {
        int result = Calculator.add(3, 4);
        Assert.assertEquals(7, result);
    }

    // More test methods...
}
```

In the example, @Before, @After, @BeforeClass, and @AfterClass annotations are used to define setup and teardown methods for test classes and test methods. The @Test annotation marks the testAddition method as a test method.

2. **Assertions:** JUnit 4 provides a set of assertion methods that are used to verify expected outcomes in test methods. These assertion methods are part of the org.junit.Assert class.

   Example

```
import org.junit.Assert;
import org.junit.Test;
public class CalculatorTest {
    @Test
    public void testAddition() {
        int result = Calculator.add(3, 4);
        Assert.assertEquals(7, result);
    }
    @Test
    public void testDivision() {
        double result = Calculator.divide(10, 2);
        Assert.assertEquals(5.0, result, 0.001); // Delta value for floating point
comparison
    }
    // More test methods...
}
```

In the example, assertEquals and assertSame methods are used to verify the expected outcomes of addition and division operations.

3. **Parameterized Tests:** JUnit 4 supports parameterized tests, allowing you to run the same test method with different sets of parameters. This is useful for testing the same functionality with multiple input values.

Example

```
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import java.util.Arrays;
import java.util.Collection;

@RunWith(Parameterized.class)
public class CalculatorParameterizedTest {
   private final int a;
   private final int b;
   private final int expected;
   public CalculatorParameterizedTest(int a, int b, int expected) {
      this.a = a;
      this.b = b;
      this.expected = expected;
   }
   @Parameterized.Parameters
   public static Collection<Object[]> data() {
      return Arrays.asList(new Object[][] {
            {1, 1, 2},
            {2, 3, 5},
            {5, 5, 10}       });
   }
   @Test
   public void testAddition() {
      int result = Calculator.add(a, b);
      Assert.assertEquals(expected, result);
   }
}
```

In the example, the CalculatorParameterizedTest class contains a parameterized test method testAddition, which is run multiple times with different sets of parameters specified by the @Parameters annotation.

4. **Exception Testing:** JUnit 4 provides built-in support for testing expected exceptions using the expected attribute of the @Test annotation or using the ExpectedException rule.

   Example

```
import org.junit.Test;

public class ExceptionTest {

    @Test(expected = ArithmeticException.class)
    public void testDivideByZero() {
        int result = Calculator.divide(10, 0);
    }
}
```

   In this example, the testDivideByZero method tests that an ArithmeticException is thrown when dividing by zero.

5. **Test Suites:** JUnit 4 allows you to organize multiple test classes into test suites using the @RunWith and @Suite annotations. Test suites enable you to run multiple tests together as a single test suite.

   Example

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
        CalculatorTest.class,
        CalculatorParameterizedTest.class,
        ExceptionTest.class
})
public class TestSuite {
    // This class is empty; it is only used as a holder for the above annotations
}
```

   In the example, the TestSuite class contains annotations to define a test suite that includes CalculatorTest, CalculatorParameterizedTest, and ExceptionTest classes.

Example : Unit testing with JUnit4

1. First add the JUnit4 dependency to your maven project pom.xml file

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
```

   If the dependency for JUnit is available inside the pom.xml file then only you need to change the version as you are going to use JUnit4.

2. Now create a class inside src/main/java with some functionality or methods on which you need to perform testing

   Calculate.java

```
package com.springmvc.example;
public class Calculate {
        public static int addition(int a, int b)        {
                return a+b;
        }
        public static int subtraction(int a, int b)    {
                return a-b;
        }
        public static int multiply(int a, int b)        {
                return a*b;
        }
        public static int divide(int a, int b) {
                return a/b;
        }
}
```

3. Create a class inside src/main/test to test the above class functionality. Make sure that the classname must contain the name of class which is going to test (Calculate) followed by word “Test”.

CalculateTest.java

```
package com.springmvc.example;
import org.junit.Test;
import org.junit.Assert;
public class CalculateTest {
	@Test
	public void additionTest()
	{
		int result= Calculate.addition(10,20);
		int expectedResult=300;

		Assert.assertEquals(expectedResult, result);
	}
}
```

4. Run the above testclass using “JUnit Test” to check the runs, failures & Errors.

We can also test multiple modules or methods of a class.

CalculateTest.java

```
package com.springmvc.example;
import org.junit.Test;
import org.junit.Assert;
public class CalculateTest {
	@Test
	public void additionTest()   {
		int result= Calculate.addition(10,20);
		int expectedResult=30;
		Assert.assertEquals(expectedResult, result);
	}
	@Test
	public void subtractionTest()        {
		int result= Calculate.subtraction(100,20);
		int expectedResult=130;
		Assert.assertEquals(expectedResult, result);
	}}
```

If we want to perform some operations before & after the execution of all the test cases then

CalculateTest.java

```
package com.springmvc.example;
import org.junit.Test; import
org.junit.AfterClass; import
org.junit.Assert; import
org.junit.BeforeClass; public
class CalculateTest {
        @BeforeClass
        public static void start()      {
                System.out.println("test cases are going to be executed");
        }
        @AfterClass
        public static void end()       {
                System.out.println("test cases have been executed");
        }
        @Test
        public void additionTest()   {
                System.out.println("addition operation");
                int result= Calculate.addition(10,20);
                int expectedResult=30;

                Assert.assertEquals(expectedResult, result);
        }
        @Test

        public void subtractionTest()        {
                System.out.println("subtraction operation");
                int result= Calculate.subtraction(100,20);
                int expectedResult=130;

                Assert.assertEquals(expectedResult, result);
        }
}
```

If we want to perform some operations before & after the execution of each test case then

CalculateTest.java

```
package com.springmvc.example;
import org.junit.Test;
import org.junit.After;
import org.junit.Assert;
import org.junit.Before;

public class CalculateTest {
	@Before
	public void start()	{
		System.out.println("test case is going to be executed");
	}
	@After
	public void end()	{
		System.out.println("test case has been executed");
	}
	@Test
	public void additionTest()	{
		System.out.println("addition operation");
		int result= Calculate.addition(10,20);
		int expectedResult=30;

		Assert.assertEquals(expectedResult, result);
	}
	@Test

	public void subtractionTest()		{
		System.out.println("subtraction operation");
		int result= Calculate.subtraction(100,20);
		int expectedResult=130;

		Assert.assertEquals(expectedResult, result);
	}
}
```

@Test(timeout = milliseconds): Sets a timeout for a test method in milliseconds. If the method takes longer than the specified timeout, it will fail.

CalculateTest.java

```
package com.springmvc.example;

import org.junit.Test;
import org.junit.Assert;

public class CalculateTest {
	@Test (timeout=3000)
	public void additionTest()
	{
		System.out.println("addition operation");
		int result= Calculate.addition(10,20);
		int expectedResult=30;
		try {
			Thread.sleep(4000);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		Assert.assertEquals(expectedResult, result);
	}

	@Test
	public void subtractionTest()
	{
		System.out.println("subtraction operation");
		int result= Calculate.subtraction(100,20);
		int expectedResult=130;

		Assert.assertEquals(expectedResult, result);
	}
}
```

Here are some other commonly used annotations along with their purposes:

- **@Ignore or @Disabled:** Marks a test method to be ignored during test execution.
- **@RunWith:** Specifies a custom test runner to use for executing the tests.
- **@Rule:** Defines a rule to be applied to all test methods in a class or to a specific test method.
- **@Parameters:** Indicates that the test method should be executed multiple times with different parameter values.
- **@Test(expected = Exception.class):** Specifies that the test method should throw a specific exception, otherwise, it will fail.
- **@DisplayName:** Provides a custom display name for the test method.

JUnit5

- JUnit 5 is a powerful and versatile testing framework for Java, designed to make writing and running tests easier and more flexible.
- JUnit 5 builds upon the foundation of JUnit 4 while introducing new features and enhancements to address the evolving needs of modern software development practices.
- Its modular architecture, flexible extension model, and support for new Java features make it a powerful choice for writing and running tests in Java applications.

Unique Features and Enhancements

- **Architecture and Modules:** JUnit 5 is composed of three main modules: JUnit Platform, JUnit Jupiter, and JUnit Vintage. The JUnit Platform serves as the foundation for launching testing frameworks on the JVM. JUnit Jupiter provides the programming model for writing tests and extensions in JUnit 5. JUnit Vintage provides backward compatibility for running JUnit 3 and JUnit 4 tests on the JUnit 5 platform.
- **Extension Model**: JUnit 5 introduces an extension model based on annotations and interfaces, allowing developers to extend the behavior of tests and the test engine. Extensions can be used to add additional functionality such as parameter resolution, lifecycle callbacks, and custom assertions.
- **Parameterized Tests:** Parameterized tests allow developers to run the same test method multiple times with different arguments. In JUnit 5, parameterized tests are more flexible and intuitive compared to JUnit 4, with support for different sources of parameters including method arguments, CSV files, and custom providers.
- **Dynamic Tests:** JUnit 5 introduces dynamic tests, which are generated at runtime by factory methods. This allows for more flexible and expressive test creation, particularly in scenarios where the number of tests or test cases is determined dynamically during execution.
- **Conditional Test Execution:** JUnit 5 introduces conditional test execution based on predefined conditions. Tests can be enabled or disabled dynamically based on conditions such as the operating system, system properties, environment variables, or custom conditions.

- **Assertions:** While assertions are not unique to JUnit 5, it provides enhanced support for assertions through the assertAll() method, which allows grouping multiple assertions within a single test case. This improves readability and helps identify multiple failures within a single test.
- **Test Instance Lifecycle:** In JUnit 5, test instances can be controlled at the class level, allowing developers to specify whether a new instance should be created for each test method or reused across multiple test methods.
- **Tagging and Filtering:** JUnit 5 introduces tagging and filtering of tests, allowing developers to categorize tests using tags and selectively execute tests based on these tags. This feature is particularly useful for organizing and managing large test suites.
- **Support for Java 8 Features:** JUnit 5 leverages Java 8 features such as lambdas and default methods to provide more expressive and concise APIs for writing tests and extensions.
- **Improved IDE Integration:** JUnit 5 offers improved integration with popular IDEs such as IntelliJ IDEA, Eclipse, and NetBeans, providing better support for features such as test discovery, execution, and reporting.

JUnit Modules

JUnit 5 is composed of three main modules, each serving a specific purpose within the testing ecosystem:

- **JUnit Platform**
    - The JUnit Platform serves as the foundation for launching testing frameworks on the JVM (Java Virtual Machine).
    - It provides an API for test engines to discover, execute, and report on tests. The platform is designed to be vendor-neutral and extensible, allowing different testing frameworks to integrate seamlessly.
    - Test engines can be written to support various testing frameworks, enabling developers to choose the framework that best fits their needs while still leveraging the benefits of the JUnit Platform.
    - The test engine implementation is the responsibility of the JUnit Platform.
- **JUnit Jupiter**
    - JUnit Jupiter is the programming model and extension model for writing tests and extensions in JUnit 5.

    - It provides annotations and APIs for defining test classes, test methods, assertions, and extensions.
    - JUnit Jupiter introduces several new features compared to JUnit 4, including support for parameterized tests, nested tests, dynamic tests, and more expressive assertions.
    - It also offers an improved extension model based on annotations and interfaces, allowing developers to extend the behavior of tests and the test engine in a modular and flexible way.
- **JUnit Vintage**
    - JUnit Vintage provides backward compatibility for running JUnit 3 and JUnit 4 tests on the JUnit 5 platform.
    - It serves as a bridge between the old and new versions of JUnit, allowing developers to migrate existing tests gradually without having to rewrite them completely.
    - JUnit Vintage supports running tests written in the JUnit 3 and JUnit 4 style, including the use of annotations such as @Test, @Before, @After, etc.
    - This ensures that legacy tests can be executed alongside tests written in the JUnit Jupiter style within the same test suite.

To use JUnit5 in our java project to write the test cases, you need to inject the following dependency in your maven project pom.xml file

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-api</artifactId>
      <version>5.9.2</version>
      <scope>test</scope>
    </dependency>
```

But the limitation with above dependency is that it only provide the API, not the implementation, so you need to use the junit-jupiter-engine dependency.

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-engine -->
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.9.2</version>
      <scope>test</scope>
    </dependency>
```

For running parameterized test cases, you need to use the following dependency

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-params -->
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-params</artifactId>
      <version>5.9.2</version>
      <scope>test</scope>
    </dependency>
```

Assertions and Annotations

JUnit 5 provides a variety of assertions and annotations to facilitate writing and organizing tests effectively.
Here's a list of commonly used assertions and annotations available in JUnit 5:

Assertions

- assertEquals(expected, actual): Asserts that two objects are equal.
- assertNotEquals(expected, actual): Asserts that two objects are not equal.
- assertTrue(condition): Asserts that a condition is true.
- assertFalse(condition): Asserts that a condition is false.
- assertNull(object): Asserts that an object is null.
- assertNotNull(object): Asserts that an object is not null.
- assertSame(expected, actual): Asserts that two objects refer to the same object.
- assertNotSame(expected, actual): Asserts that two objects do not refer to the same object.
- assertArrayEquals(expectedArray, actualArray): Asserts that two arrays are equal.
- assertIterableEquals(expectedIterable, actualIterable): Asserts that two iterables are equal.
- assertLinesMatch(expectedLines, actualLines): Asserts that two lists of strings are equal, ignoring any differences in line terminators.
- assertThrows(exceptionType, executable): Asserts that an executable throws a specific type of exception.
- assertTimeout(duration, executable): Asserts that an executable completes within a specified duration.
- assertIterableEquals(expected, actual, [comparator]): Asserts that two iterables are equal, optionally using a custom comparator.
- assertAll(executables...): Asserts that all of the provided executable assertions pass.

- assertDoesNotThrow(executable): Asserts that an executable does not throw any exception.
- assertDoesNotThrowAny(ThrowableType, executable): Asserts that an executable does not throw an exception of the specified type.
- assertDoesNotThrowAny(executables): Asserts that none of the provided executables throw any exception.
- assertIterableEquals(expected, actual, [comparator], [messageSupplier]): Asserts that two iterables are equal, optionally using a custom comparator and providing a custom message supplier.

Annotations

- @Test: Marks a method as a test method.
- @BeforeEach: Indicates that the annotated method should be executed before each test method.
- @AfterEach: Indicates that the annotated method should be executed after each test method.
- @BeforeAll: Indicates that the annotated method should be executed once before all test methods in the current class.
- @AfterAll: Indicates that the annotated method should be executed once after all test methods in the current class.
- @DisplayName: Provides a custom display name for a test class or test method.
- @Disabled: Disables a test class or test method from execution.
- @ParameterizedTest: Indicates that the annotated method is a parameterized test.
- @RepeatedTest: Indicates that the annotated method should be repeated a specified number of times.
- @Nested: Allows nesting test classes within another test class.
- @Tag: Tags a test class or test method with one or more labels.
- @Timeout: Specifies a timeout for a test method or test template method.
- @TestFactory: Indicates that the annotated method is a test factory for dynamic tests.
- @DisplayNameGeneration: Specifies a custom display name generator for test classes and test methods.
- @ExtendWith: Registers one or more extensions for a test class or test method.
- @RegisterExtension: Declares a field to be used as an extension instance.
- @TempDir: Provides a temporary directory that is created before each test method execution.
- @TimeoutTest: Specifies a timeout for a specific test method.

Mockito

- Mockito is a popular Java library used for creating and configuring mock objects for testing purposes.
- It allows developers to simulate the behavior of external dependencies or collaborators within their tests, enabling isolated unit testing.
- Mockito is particularly useful for Java and Spring Boot applications because it helps in writing reliable and maintainable tests by mocking external dependencies and focusing on testing individual units of code in isolation.
- Mockito is a valuable tool for Java and Spring Boot developers, providing them with the means to write effective unit tests that are focused, reliable, and maintainable.
- By using Mockito to mock dependencies and isolate units under test, developers can ensure the quality and correctness of their code while promoting best practices in software testing.
- In simple words, a mock object returns a dummy data and avoids external dependencies.
- It uses java reflection API to create objects.

How Mockito is beneficial for Java and Spring Boot applications:

- **Mocking Dependencies:** Mockito enables developers to create mock objects representing dependencies of the component being tested. These mock objects mimic the behavior of real objects but allow developers to specify their behavior and responses during test scenarios.
- **Isolation of Unit Tests:** With Mockito, developers can isolate the unit under test by replacing its dependencies with mock objects. This isolation ensures that each unit test focuses solely on the behavior of the unit being tested, without interference from external components.
- **Flexible Behavior Configuration:** Mockito provides a rich set of methods for configuring the behavior of mock objects. Developers can specify return values, throw exceptions, verify method invocations, and more, allowing them to simulate various scenarios and edge cases in their tests.
- **Verification of Interactions:** Mockito allows developers to verify interactions between the unit under test and its dependencies. This ensures that the unit behaves correctly by making the expected calls to its collaborators with the correct arguments.

- **Integration with Testing Frameworks:** Mockito integrates seamlessly with popular Java testing frameworks like JUnit and TestNG. Developers can easily incorporate Mockito into their existing test suites and leverage its features to write comprehensive and reliable tests.
- **Simplified Test Setup:** By using Mockito, developers can simplify the setup process for unit tests by avoiding the need to create complex object graphs or set up elaborate test environments. Mock objects can be created on-the-fly within test methods, reducing boilerplate code and improving test readability.
- **Enhanced Test Coverage:** Mockito facilitates the creation of comprehensive test suites by allowing developers to simulate various scenarios and edge cases, including error conditions and exceptional behavior. This helps in achieving higher test coverage and uncovering potential bugs or issues early in the development process.

Reflection API

- The Reflection API allows you to examine or modify the runtime behavior of applications running on the Java Virtual Machine (JVM).
- It provides a way to inspect classes, interfaces, fields, and methods at runtime, as well as instantiate objects, invoke methods, and access or modify fields dynamically, without having direct compile-time knowledge of the classes involved.
- This ability to introspect and manipulate classes and objects at runtime makes the Reflection API a powerful and flexible tool, often used in frameworks, libraries, and tools that require dynamic behavior.
- Reflection provides a way to inspect and manipulate classes and objects dynamically, offering flexibility and power but also requiring caution due to its potential impact on performance and security.
- It's commonly used in frameworks like Spring and Hibernate, as well as in various testing and debugging tools.
- Reflection API has the capability to dynamically generate proxies for interfaces and classes. This feature enables the creation of dynamic proxy instances that intercept method calls and provide custom behavior before or after invoking the actual methods of the target object.

Reflection API - key concepts

- **Class Objects:** In the Reflection API, everything starts with Class objects, which represent classes or interfaces in Java. These Class objects can be obtained in various ways, such as using the .class syntax, Class.forName(), or by calling .getClass() on an object instance.
- **Inspecting Classes:** Reflection allows you to inspect the structure of classes at runtime. You can access information about fields, methods, constructors, annotations, and more using methods provided by the Class class.
- **Accessing Fields and Methods:** Reflection enables you to access fields and methods of classes dynamically, even private ones. You can get, set, or invoke fields and methods using Field and Method objects, respectively.
- **Dynamic Instantiation:** Reflection enables you to instantiate objects of classes dynamically, without knowing their type at compile time. You can create new instances using constructors obtained through reflection.
- **Dynamic Invocation:** Reflection allows you to invoke methods of classes dynamically, without knowing their signatures at compile time. You can invoke methods with specific arguments using reflection.

Example : Reflection API

```
package com.java.example;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

class Employee {
   private String name;
   private int age;

   public Employee(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public void getDetails() {
```

```
        System.out.println("Hello, my name is " + name + " and I am " + age + " years old.");
    }
}

public class ReflectionExample {
    public static void main(String[] args) throws Exception {
        // Get the Class object for the Employee class
        Class<?> employeeClass = Employee.class;

        // Instantiate a new Employee object using reflection
        Constructor<?> constructor = employeeClass.getConstructor(String.class, int.class);
        Object employee = constructor.newInstance("Nisha", 30);

        // Access and modify private field 'age' using reflection
        Field ageField = employeeClass.getDeclaredField("age");
        ageField.setAccessible(true);
       // ageField.setInt(employee, 35);

        // Invoke the 'getDetails' method dynamically
        Method getDetailsMethod = employeeClass.getDeclaredMethod("getDetails");
        getDetailsMethod.invoke(employee);
    }
}
```

Example – Execute parameterized method using Reflection API

```
package com.springboot.example;

import java.lang.reflect.Method;

class ParameterizedMethodExample {
    public static void main(String[] args) throws Exception {
        // Get the Class object for the class containing the method
        Class<?> cls = Employee.class;
```

```
        // Get the Method object representing the parameterized method
        Method method = cls.getMethod("getDetails", int.class, String.class);
        Employee employee = new Employee();
        // Define arguments to pass to the method
        int arg1 = 32;
        String arg2 = "Tanya Sharma";
        // Invoke the parameterized method dynamically Object result =
        method.invoke(employee, arg1, arg2); System.out.println("Method
        executed successfully with result: " + result);
    }
}
class Employee {
    public String getDetails(int id, String name) {
        return "Employee Id is : " + id+ ", & his name is :  " + name;
    }
}
```

As Reflection API is capable to dynamically generate proxies for interfaces and classes then here is the overview of dynamic proxies and their benefits:

- **Interface Proxies:** With reflection, you can create a proxy instance that implements one or more interfaces. This proxy intercepts method calls made to those interfaces and allows you to provide custom behavior.
- **Method Invocation Handling:** Dynamic proxies give you the ability to intercept method invocations on the proxy object and perform additional logic before or after invoking the actual method on the target object.
- **Aspect-Oriented Programming (AOP):** Dynamic proxies are commonly used in AOP frameworks to implement cross-cutting concerns such as logging, security, and transaction management. By intercepting method calls, AOP proxies can inject additional behavior into the application without modifying the original source code.
- **Decoupling:** Dynamic proxies help in decoupling the concerns of the application. They allow you to separate the core business logic from cross-cutting concerns, resulting in cleaner and more maintainable code.
- **Lazy Initialization:** Dynamic proxies can be used for lazy initialization of objects. Instead of creating an instance of a heavyweight object upfront, you can defer its creation until the first method call on the proxy, improving performance by avoiding unnecessary object creation.

Example : demonstrating how to create a dynamic proxy using the Reflection API

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
interface Hello {
    void sayHello();
}
class HelloImpl implements Hello {
    public void sayHello() {
        System.out.println("Hello, World!");
    }
}

class MyInvocationHandler implements InvocationHandler {
    private Object target;
    public MyInvocationHandler(Object target) {
        this.target = target;
    }
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        System.out.println("Before invoking method: " + method.getName());
        Object result = method.invoke(target, args);
        System.out.println("After invoking method: " + method.getName());
        return result;
    }
}
public class DynamicProxyExample {
    public static void main(String[] args) {
        Hello hello = new HelloImpl();
        InvocationHandler handler = new MyInvocationHandler(hello);
        Hello proxy = (Hello) Proxy.newProxyInstance(
                Hello.class.getClassLoader(),
                new Class[] { Hello.class },
                handler);
        proxy.sayHello();
    }
}
```

In the example:

- We define an interface Hello and a class HelloImpl that implements the interface.
- We create an InvocationHandler implementation MyInvocationHandler that intercepts method calls on the proxy.
- We create a dynamic proxy instance using Proxy.newProxyInstance(), passing the interface Hello, the class loader, and the InvocationHandler.
- When the sayHello() method is invoked on the proxy, the invoke() method of the InvocationHandler is executed, allowing us to perform custom logic before and after invoking the actual method on the target object.

Dynamic proxies provide a flexible mechanism for adding behavior to objects at runtime, making them a powerful tool for building extensible and modular applications.

Lets, come back to Mockito

So, if we want to perform Unit testing using Mockito then the following dependencies would be required:

```
<!-- https://mvnrepository.com/artifact/org.mockito/mockito-core -->
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.2.0</version>
    <scope>test</scope>
</dependency>
```

If you are using JUnit5 then you also need to inject the following dependency

```
<!-- https://mvnrepository.com/artifact/org.mockito/mockito-junit-jupiter -->
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>5.2.0</version>
    <scope>test</scope>
</dependency>
```

Exercise:

- First create a class Employee

```
public class Employee
{
        private EmployeeDetails employeeDetails;

        public int getEmployeeID()
        {
                int id = this.employeeDetails.getEmployeeDetails();
                System.out.println("The Employee Id is : "+id);
                return id;
        }
}
```

- Then create an interface EmployeeDetails

```
public interface EmployeeDetails{
        public int getEmployeeDetails();
}
```

Now create a test class to test the above classes and interfaces

```
public class EmployeeTest{

    public void getEmployeeIDTest()
    {
            Employee  emp=new  Employee();
            int   result=emp.getEmployeeID();
            Assertions.assertEquals(20,result);
    }
}
```

So, If we run the above code then we will get NullPointerException because we don't have any value to Test.

So to overcome this problem you can create a stub for Testing the module

Now create a constructor inside the Employee class

```
public class Employee
{
	private EmployeeDetails employeeDetails;

	public Employee(EmployeeDetails employeeDetails)	{
		this.employeeDetails=employeeDetails;
	}

	public int getEmployeeID() {
		int id = this.employeeDetails.getEmployeeDetails();
		System.out.println("The Employee Id is : "+id);
		return id;
	}
}
```

& create a stub

```
public class EmployeeTest{
	public void getEmployeeIDTest()
	{
		EmployeeDetails employeeDetails=new
EmployeeDetailsImpl();
		Employee emp=new Employee(employeeDetails);
		int result=emp.getEmployeeID();
		Assertions.assertEquals(20,result);
	}
}
//stub
class EmployeeDetailsImpl implements EmployeeDetails
{
	@Override
	public int getEmployeeDetails(){
		return 20;
	}
}
```

The above concept get failed when you will have some additional functionalities in the EmployeeDetails interface in future so to manage these types of operations would be complex & you need to manually make changes inside the implementation class to provide the functionality

In that case you need to use the Mockito to mock the objects.

Now you can have the Test class like

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

public class EmployeeTest{

	@Test
	public void getEmployeeIDTest()
	{
		//Mocking Object
		EmployeeDetails employeeDetails=Mockito.mock(EmployeeDetails.class);
		Employee emp=new Employee(employeeDetails);
		// To declare that when the getEmployeeDetails method get called then it will return 20.
		Mockito.when(employeeDetails.getEmployeeDetails()).thenReturn(200);
		int result=emp.getEmployeeID();
		Assertions.assertEquals(20,result);
	}

}
```

Mockito Annotations

Mockito is a popular Java framework used for creating and configuring mock objects. Mockito provides several annotations that help in writing unit tests effectively.

In previous example we had used the below statement to create the mock object of the interface EmployeeDetails.

```
EmployeeDetails employeeDetails=Mockito.mock(EmployeeDetails.class);
```

So, if you don't want to do this then you can simply use the @Mock Annotation to create the mock object of the same interface.

& we had also used the below statement to inject the mock object into the Employee

```
Employee emp=new Employee(employeeDetails);
```

So, if you don't want to do this then you can use @InjectMocks Annotation.

To use the Mockito related annotations, you need to initialize the MockitoAnnotations using method MockitoAnnotations.initMocks(). The method must get executed before the execution of every test case or all the test cases of the test class.

Employee.java

```
public class Employee
{
        private EmployeeDetails employeeDetails;
        public Employee(EmployeeDetails employeeDetails)    {
                this.employeeDetails=employeeDetails;
        }
        public int getEmployeeID() {
                int id = this.employeeDetails.getEmployeeDetails();
                System.out.println("The Employee Id is : "+id);
                return id;
        }
}
```

EmployeeDetails.java

```
public interface EmployeeDetails{
        public int getEmployeeDetails();
        }
```

EmployeeTest.java

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.MockitoAnnotations;

public class EmployeeTest{

	@Mock
	EmployeeDetails employeeDetails;
	@InjectMocks
	Employee employee;

	@BeforeEach
	public void init()	{
		MockitoAnnotations.initMocks(this);
	}
	@Test
	public void getEmployeeIDTest()
	{
	Mockito.when(employeeDetails.getEmployeeDetails()).thenReturn(20);
		int result=employee.getEmployeeID();
		Assertions.assertEquals(20,result);
	}
}
```

If you don't want to use MockitoAnnotations.initMocks(this) for initializing annotated fields manually, you can use MockitoExtension along with @ExtendWith annotation. MockitoExtension is provided by Mockito as an extension for JUnit 5, which automatically initializes annotated fields for you.

EmployeeTest.java

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
public class EmployeeTest{
	@Mock
	EmployeeDetails employeeDetails;
	@InjectMocks
	Employee employee;
	@Test
	public void getEmployeeIDTest()  {
	Mockito.when(employeeDetails.getEmployeeDetails()).thenReturn(20);
		int result=employee.getEmployeeID();
		Assertions.assertEquals(20,result);
	}
}
```

With the setup, you don't need to call MockitoAnnotations.initMocks(this) in your setup method. The MockitoExtension will take care of initializing mocks and spies for you. This approach is specifically for JUnit 5. If you're using JUnit 4, then MockitoJUnitRunner or MockitoAnnotations.initMocks(this) are the options available.

Mocking Object of Predefined Java Classes

Mockito can mock interfaces and abstract classes without any issues.

CollectionOperation.java

```
import java.util.List;
import java.util.Map;
import java.util.Set;

public class CollectionOperation {
   public int processList(List<String> list) {
      return list.size();
   }
   public int processSet(Set<String> set) {
      return set.size();
   }
   public int processMap(Map<Integer, String> map) {
      return map.size();
   }
}
```

CollectionOperationTest.java

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.*;
import static org.mockito.Mockito.when;

@ExtendWith(MockitoExtension.class)
public class CollectionOperationTest {

   @Mock
   private List<String> listMock;
```

```
    @Mock
    private Set<String> setMock;

    @Mock
    private Map<Integer, String> mapMock;

    @Test
    public void testProcessList() {
        when(listMock.size()).thenReturn(5);
        CollectionOperation collectionOperation = new  CollectionOperation();
        int result = collectionOperation.processList(listMock);
        Assertions.assertEquals(5, result);
    }

    @Test
    public void testProcessSet() {
        when(setMock.size()).thenReturn(3);
        CollectionOperation collectionOperation = new  CollectionOperation();
        int result = collectionOperation.processSet(setMock);
        Assertions.assertEquals(3, result);
    }

    @Test
    public void testProcessMap() {
        when(mapMock.size()).thenReturn(10);
        CollectionOperation collectionOperation = new  CollectionOperation();
        int result = collectionOperation.processMap(mapMock);
        Assertions.assertEquals(10, result);
    }
}
```

Example : Unit testing of Spring Boot Application using JUnit5

- **Create a Spring Boot Project**
  Create a new Spring Boot project with the Spring Web, Spring Data MongoDB, Lombok & Spring Boot Dev Tools dependencies.

- **Define the Employee Entity**
  Employee.java

```
package com.springbootapi.example;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import lombok.Data;
@Data
@Document(collection = "employees")
public class Employee {
   @Id
   private String id;
   private String name;
   private String city;
}
```

- **Create a Repository Interface for performing CRUD operations on the Employee collection in MongoDB.**
  EmployeeRepository.java

```
package com.springbootapi.example;
import org.springframework.data.mongodb.repository.MongoRepository;
public interface EmployeeRepository extends MongoRepository<Employee,
String> {
}
```

- **Create a Service Class to implement business logic for managing employees**
  EmployeeService.java

```
package com.springbootapi.example;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
@Service
public class EmployeeService {
```

```
    @Autowired
    private EmployeeRepository employeeRepository;
    public Employee save(Employee employee) {
        return employeeRepository.save(employee);
    }
    public List<Employee> findAll() {
        return employeeRepository.findAll();
    }
}
```

- **Create a REST controller EmployeeController to handle HTTP requests**

EmployeeController.java

```
package com.springbootapi.example;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
@RequestMapping("/api/employees")
public class EmployeeController {
    @Autowired
    private EmployeeService employeeService;
    @PostMapping
    public ResponseEntity<Employee> createEmployee(@RequestBody
Employee employee) {
        return ResponseEntity.ok(employeeService.save(employee));
    }
    @GetMapping
    public ResponseEntity<List<Employee>> getAllEmployees() {
        return ResponseEntity.ok(employeeService.findAll());
    }
}
```

- **Write unit tests using JUnit5 and Mockito to test the service and controller classes.**

EmployeeServiceTest

```
package com.springbootapi.example;
import java.util.ArrayList;
import java.util.List;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
class EmployeeServiceTest {

   @InjectMocks
   private EmployeeService employeeService;

   @Mock
   private EmployeeRepository employeeRepository;

   @Test
   void saveEmployeeTest() {
      Employee employee = new Employee();
      employee.setName("Neha Mittal");
      employee.setCity("Gurugram");

Mockito.when(employeeRepository.save(Mockito.any(Employee.class))).the
nReturn(employee);

      Employee savedEmployee = employeeService.save(employee);
      Assertions.assertNotNull(savedEmployee);
      Assertions.assertEquals("Neha Mittal", savedEmployee.getName());
      Assertions.assertEquals("Gurugram", savedEmployee.getCity());
   }
```

```
    @Test
    void findAllEmployeesTest() {
        List<Employee> employees = new ArrayList<>();
        employees.add(new Employee());
        employees.add(new Employee());

        Mockito.when(employeeRepository.findAll()).thenReturn(employees);

        List<Employee> allEmployees = employeeService.findAll();

        Assertions.assertNotNull(allEmployees);
        Assertions.assertEquals(2, allEmployees.size());
    }
}
```

- **Manage application.properties file for your Spring Boot application with MongoDB configuration**

  application.properties

```
spring.application.name=springbootapplication
# MongoDB properties
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=empdatabase
```

- **Example of the JSON body you can use in Postman**

```
{
    "id":35,
    "name": "Sagar Agarwal",
    "city": "Agra"
}
```

- **Run the test cases**

  To run the test cases, open the test class containing your unit tests and Right-click on the test class file in the Package Explorer or in the editor window. Then, select "Run As" -> "JUnit Test" from the context menu.

Example : Unit testing of Spring Boot CRUD Application using JUnit5

Employee.java, EmployeeRepository.java & application.properties

Remain Same

EmployeeService

```
package com.springbootapi.example;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.Optional;
@Service
public class EmployeeService {
    @Autowired
    private EmployeeRepository employeeRepository;
    public Employee save(Employee employee) {
        return employeeRepository.save(employee);
    }
    public List<Employee> findAll() {
        return employeeRepository.findAll();
    }
    public Optional<Employee> findById(String id) {
        return employeeRepository.findById(id);
    }
    public Employee updateEmployee(String id, Employee updatedEmployee) {
        Optional<Employee> existingEmployeeOptional = 
employeeRepository.findById(id);
        if (!existingEmployeeOptional.isPresent()) {
            throw new ResourceNotFoundException("Employee not found with id: " + 
id);
        }
        Employee existingEmployee = existingEmployeeOptional.get();
        existingEmployee.setName(updatedEmployee.getName());
        existingEmployee.setCity(updatedEmployee.getCity());
        return employeeRepository.save(existingEmployee);
    }
```

```
    public void deleteEmployee(String id) {
        Optional<Employee> existingEmployeeOptional =
employeeRepository.findById(id);
 if (!existingEmployeeOptional.isPresent()) {
            throw new ResourceNotFoundException("Employee not found with id: " +
id);
        }
        employeeRepository.deleteById(id);
    }

}
```

ResourceNotFoundException.java

```
package com.springbootapi.example;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;
@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}
```

EmployeeController.java

```
package com.springbootapi.example;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import java.util.List;
@RestController
@RequestMapping("/api/employees")
public class EmployeeController {
    @Autowired
    private EmployeeService employeeService;
    @PostMapping
    public ResponseEntity<Employee> createEmployee(@RequestBody Employee
employee) {
```

```
        return ResponseEntity.ok(employeeService.save(employee));
    }
    @GetMapping
    public ResponseEntity<List<Employee>> getAllEmployees() {
        return ResponseEntity.ok(employeeService.findAll());
    }
    @GetMapping("/{id}")
    public ResponseEntity<Employee> getEmployeeById(@PathVariable String id) {
        return ResponseEntity.ok(employeeService.findById(id).orElseThrow(() ->
                new ResourceNotFoundException("Employee not found with id: " + id)));
    }
    @PutMapping("/{id}")
    public ResponseEntity<Employee> updateEmployee(@PathVariable String id,
@RequestBody Employee updatedEmployee) {
        return ResponseEntity.ok(employeeService.updateEmployee(id,
updatedEmployee));
    }
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteEmployee(@PathVariable String id) {
        employeeService.deleteEmployee(id);
        return ResponseEntity.noContent().build();
    }
}
```

EmployeeServiceTest.java

```
package com.springbootapi.example;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.junit.jupiter.MockitoExtension;
import java.util.Optional;
import static org.junit.jupiter.api.Assertions.*;
```

```
@ExtendWith(MockitoExtension.class)
class EmployeeServiceTest {
    @InjectMocks
    private EmployeeService employeeService;
    @Mock
    private EmployeeRepository employeeRepository;
    @Test
    void findEmployeeByIdTest() {
        String id = "1";
        Employee employee = new Employee();
        employee.setId(id);
        employee.setName("Shweta Sharma");
        employee.setCity("Noida");
Mockito.when(employeeRepository.findById(id)).thenReturn(Optional.of(employee));
        Optional<Employee> foundEmployee = employeeService.findById(id);
        assertTrue(foundEmployee.isPresent());
        Assertions.assertEquals("Shweta Sharma", foundEmployee.get().getName());
        Assertions.assertEquals("Noida", foundEmployee.get().getCity());
    }

    @Test
    void updateEmployeeTest() {
        String id = "1";
        Employee existingEmployee = new Employee();
        existingEmployee.setId(id);
        existingEmployee.setName("Shweta Sharma");
        existingEmployee.setCity("Noida");

        Employee updatedEmployee = new Employee();
        updatedEmployee.setId(id);
        updatedEmployee.setName("Tarun Gupta");
        updatedEmployee.setCity("Tundla");
    Mockito.when(employeeRepository.findById(id)).thenReturn(Optional.of(existingE
    mployee));
    Mockito.when(employeeRepository.save(Mockito.any(Employee.class))).thenRetur
    n(updatedEmployee);
        Employee result = employeeService.updateEmployee(id, updatedEmployee);
```

```
        Assertions.assertNotNull(result);
        Assertions.assertEquals("Tarun Gupta", result.getName());
        Assertions.assertEquals("Tundla", result.getCity());
    }

    @Test
    void deleteEmployeeTest() {
        String id = "1";
        Employee employee = new Employee();
        employee.setId(id);
        employee.setName("Shweta Sharma");
        employee.setCity("Agra");
Mockito.when(employeeRepository.findById(id)).thenReturn(Optional.of(employee));
        employeeService.deleteEmployee(id);
        Mockito.verify(employeeRepository, Mockito.times(1)).deleteById(id);
    }

    @Test
    void updateNonExistentEmployeeTest() {
        String id = "1";
        Employee updatedEmployee = new Employee();
        updatedEmployee.setId(id);
        updatedEmployee.setName("Shweta Sharma");
        updatedEmployee.setCity("Noida");
    Mockito.when(employeeRepository.findById(id)).thenReturn(Optional.empty());
        Assertions.assertThrows(ResourceNotFoundException.class, () -> {
            employeeService.updateEmployee(id, updatedEmployee);
        });
    }
}
```

Run the test cases

To run the test cases, open the test class containing your unit tests and Right-click on the test class file in the Package Explorer or in the editor window. Then, select "Run As" -> "JUnit Test" from the context menu.

- **Create Spring Boot Application Class**
  Create a class annotated with @SpringBootApplication. This class serves as the entry point for your Spring Boot application.

  MySpringBootApplication.java

```
import org.springframework.boot.SpringApplication; import
org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class MySpringBootApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

- **Access Thymeleaf Templates**
  Thymeleaf templates will be rendered dynamically based on the data provided by your Spring MVC controllers. You can access these templates by navigating to the corresponding URL mappings in your application.

  AppController.java

```
package com.springboot.example;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
@Controller
public class AppController {
    @GetMapping("/")
    public String homePage(Model model) {
        model.addAttribute("message", "Hello, Dear Students!");
        return "index"; // Thymeleaf template name without extension
    }
}
```

- **Run the Application**
  Right-click on the main application class (MySpringBootApplication in the example).
  Select Run As > Spring Boot App.

- **Expression Language**

  Thymeleaf:

  Thymeleaf provides its own expression language (Thymeleaf Standard Dialect), which offers features like variable access, iteration, conditionals, and more. Thymeleaf expressions are designed to be simple and powerful, allowing for easy manipulation of data within templates.

  JSP:

  JSP uses JavaServer Pages Expression Language (EL), which provides similar functionality to Thymeleaf expressions but with a syntax that is more closely tied to Java.

- **Popularity and Ecosystem**

  Thymeleaf:

  Thymeleaf has gained popularity in recent years due to its ease of use, powerful features, and seamless integration with Spring Boot. It has a growing ecosystem of plugins, extensions, and community support.

  JSP:

  JSP has been a standard part of the Java EE (now Jakarta EE) platform for many years and has a large ecosystem of libraries and frameworks. However, its usage has declined in favor of more modern templating engines like Thymeleaf.

Overall, both Thymeleaf and JSP are capable technologies for server-side rendering in Java web applications, but Thymeleaf is often preferred for its cleaner syntax, better integration with Spring Boot, and more modern approach to templating.

Microservices

- Microservices are a software development approach where a large application is built as a collection of loosely coupled services.
- Each service is focused on a specific business capability and can be developed, deployed, and scaled independently.
- Microservices architecture can help organizations build more scalable, flexible, and resilient applications that can adapt to changing business requirements.
- Microservices play a crucial role in large-scale applications due to their ability to break down complex systems into smaller, more manageable parts.

Key characteristics of microservices:

- **Decomposition:** A monolithic application is broken down into smaller, manageable services, each responsible for a specific function or feature.
- **Independence:** Each microservice can be developed, deployed, and scaled independently. This allows for faster development cycles and easier maintenance.
- **Communication:** Microservices communicate with each other over the network, typically using lightweight protocols such as HTTP or messaging queues.
- **Data Management:** Each microservice can have its own database, chosen to best fit the service's requirements. This can include relational databases, NoSQL databases, or even in-memory stores.
- **Resilience:** Microservices are designed to be resilient to failures. If one service goes down, it should not bring down the entire application.
- **Scalability:** Since each microservice is independent, it can be scaled independently based on the load it receives. This allows for better resource utilization and cost-effectiveness.
- **DevOps:** Microservices are often associated with DevOps practices, where development and operations teams work closely together to automate the deployment and monitoring of services.
- **Challenges:** While microservices offer many benefits, they also come with challenges such as managing the increased complexity of a distributed system, ensuring consistency across services, and handling inter-service communication.

How microservices contribute to the success of large-scale applications?

Microservices enable large-scale applications to be more agile, scalable, and resilient, making them better able to meet the demands of modern software development.

Here are some key ways in which microservices contribute to the success of large-scale applications:

- **Scalability:** Microservices allow different parts of an application to be scaled independently based on their specific needs. This means that resources can be allocated more efficiently, leading to better performance and cost-effectiveness.
- **Flexibility:** Microservices enable teams to work on different parts of an application independently, using different technologies and deployment strategies. This flexibility allows for faster development cycles and easier adaptation to changing requirements.
- **Resilience:** In a microservices architecture, if one service fails, it does not necessarily impact the entire application. This resilience is achieved through the use of techniques such as load balancing, fault tolerance, and graceful degradation.
- **Technology Diversity:** Microservices allow teams to use the best technology for each service's specific requirements. This means that teams can leverage the strengths of different technologies without being constrained by a monolithic architecture.
- **Continuous Delivery:** Microservices make it easier to adopt continuous delivery practices, as each service can be deployed independently. This leads to faster release cycles and quicker time-to-market.
- **Easier Maintenance:** With microservices, it is easier to understand and maintain different parts of an application since each service has a clearly defined scope and responsibility. This can lead to reduced maintenance costs over time.
- **Incremental Updates:** Microservices allow for incremental updates to be made to an application, as changes can be rolled out to individual services without affecting the entire system. This reduces the risk of introducing bugs or breaking changes.

Monolithic Architecture

- Monolithic architecture is a traditional approach to designing software where the entire application is built as a single, indivisible unit.
- In a monolithic architecture, all the components of the application, such as the user interface, business logic, and data access layer, are tightly coupled and run as a single process on a single server.
- Scaling a monolithic application typically involves scaling the entire application, which can be inefficient and costly.
- Monolithic applications are often deployed as a single unit, making it challenging to deploy updates or changes to specific parts of the application without affecting the entire system.

Distributed Architecture

- Distributed architecture is an architectural approach where different parts of the application are distributed across multiple systems or nodes that communicate with each other over a network.
- In a distributed architecture, each component of the application is a separate service that can be developed, deployed, and scaled independently.
- Scaling a distributed application involves scaling individual components or services based on their specific needs, which can lead to better resource utilization and cost-effectiveness.
- Distributed applications are typically more resilient to failures since a failure in one component does not necessarily affect the entire application.

Difference between Monolithic and Distributed Architecture

- **Deployment:** In a monolithic architecture, the entire application is deployed as a single unit, while in a distributed architecture, different parts of the application are deployed independently.
- **Scalability:** Monolithic applications are scaled by replicating the entire application, whereas distributed applications can be scaled by scaling individual components.
- **Development and Deployment Independence:** In a monolithic architecture, development and deployment are tightly coupled, while in a distributed architecture, they can be decoupled, allowing for more flexibility and agility.

- **Fault Tolerance:** Distributed architectures are typically more fault-tolerant than monolithic architectures, as a failure in one component does not necessarily bring down the entire application.
- **Technology Stack:** Monolithic applications usually use a single technology stack for the entire application, while distributed applications can use different technologies for different components.
- **Complexity:** Monolithic applications are generally simpler to develop and deploy compared to distributed applications, which can be more complex due to the distributed nature of the architecture.

How microservices work in a distributed environment?

- In a distributed environment, microservices work by breaking down a large application into smaller, loosely coupled services that communicate with each other over a network.
- Each microservice is responsible for a specific business function and can be developed, deployed, and scaled independently. This approach offers several advantages, such as improved scalability, flexibility, and resilience.

why we can't design microservices for monolithic architecture?

One key reason why microservices are not suitable for a monolithic architecture is that they rely on a distributed system's principles, which are fundamentally different from those of a monolith. A monolithic architecture typically involves a single, tightly integrated codebase that is deployed as a single unit. In contrast, microservices are designed to be distributed, with each service running in its own process and communicating with other services over a network.

Additionally, designing microservices for a monolithic architecture would defeat the purpose of using microservices in the first place. Microservices are intended to address the limitations of monolithic architectures, such as scalability, flexibility, and resilience. Attempting to shoehorn microservices into a monolithic architecture would likely result in a complex and unwieldy system that lacks many of the benefits of a true microservices architecture.

Overall, while microservices offer many advantages in a distributed environment, they are not a one-size-fits-all solution and should be carefully considered based on the specific requirements and constraints of the application being developed.

Developing and Deploying Microservices for a Distributed Architecture

Developing and deploying microservices for a distributed architecture involves below key steps:

- **Define Service Boundaries:** Identify the different parts of your application that can be broken down into separate services. Each service should have a well-defined boundary and responsibility.
- **Choose Communication Protocols:** Decide on the communication protocols that will be used between services. Common options include HTTP/REST, messaging queues (e.g., RabbitMQ, Kafka), and gRPC.
- **Design APIs:** Define the APIs that will be used by each service to communicate with other services. This includes specifying the data formats (e.g., JSON, XML) and the endpoints for each service.
- **Implement Services:** Develop each microservice using the appropriate technologies and frameworks. Each service should be self-contained and responsible for a specific business function.
- **Containerize Services:** Use containerization technology (e.g., Docker) to package each microservice along with its dependencies. This ensures that the service can run consistently across different environments.
- **Deploy Services:** Deploy each microservice to your chosen deployment environment (e.g., on-premises, cloud). Use orchestration tools (e.g., Kubernetes, Docker Swarm) to manage and scale your services.
- **Monitor and Manage:** Implement monitoring and logging to track the performance and health of your microservices. Use tools like Prometheus, Grafana, and ELK stack for monitoring and logging.
- **Handle Failures:** Implement strategies to handle failures gracefully, such as retrying failed requests, circuit breaking, and fallback mechanisms.
- **Scale Services:** Use auto-scaling features provided by your deployment environment to scale your services based on demand.
- **Update Services: Implement** a strategy for updating your services, such as blue-green deployments or canary releases, to minimize downtime and impact on users.
- **Security:** Implement security best practices, such as authentication, authorization, and encryption, to protect your microservices from unauthorized access and attacks.
- **Testing:** Perform thorough testing of your microservices, including unit tests, integration tests, and end-to-end tests, to ensure they work correctly and meet the required quality standards.