HIBERNATE
==========

Hibernate is an open-source Java framework that provides an object-relational mapping (ORM) solution. It simplifies database access for Java applications by mapping Java objects to database tables, enabling developers to work with databases using Java objects rather than writing raw SQL queries. Here are some key aspects of Hibernate:

Features of Hibernate:

1. **Object-Relational Mapping (ORM)**: Hibernate allows you to map Java objects to database tables, making it easier to work with relational databases.

2. **Database Independence**: Hibernate abstracts the underlying database, allowing you to write database-agnostic code. You can switch between different databases without changing your application's code.

3. **Automatic Table Generation**: Hibernate can automatically generate database tables based on your Java entities.

4. **Caching**: Hibernate provides caching mechanisms to improve performance, reducing the number of database queries.

5. **Query Language (HQL)**: Hibernate offers its query language called HQL (Hibernate Query Language) for querying the database. HQL is similar to SQL but operates on Java objects.

6. **Lazy Loading**: Hibernate supports lazy loading, which means that it loads related data from the database only when it is explicitly requested.

7. **Transaction Management**: It provides built-in support for managing database transactions.

8. **Association Mapping**: Hibernate supports various types of associations between entities, such as one-to-one, one-to-many, and many-to-many.

Differences between Hibernate and JDBC:

1. **Abstraction Level**: Hibernate provides a higher-level abstraction for database access, while JDBC is a lower-level API for interacting with databases. With Hibernate, you work with Java objects, whereas with JDBC, you write SQL queries and work with result sets directly.

2. **SQL vs. HQL**: With JDBC, you write SQL queries explicitly. In Hibernate, you can use HQL, which is a higher-level query language that operates on Java objects.

3. **Mapping**: In Hibernate, you define entity classes to map to database tables, while in JDBC, you need to write code to map Java objects to SQL and vice versa.

4. **Portability**: Hibernate provides database independence, allowing you to switch between different databases with minimal code changes. JDBC code may be tightly coupled to a specific database, making it less portable.

Examples:

JDBC Example:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class JDBCDemo {
    public static void main(String[] args) {
        try {
            // Load the JDBC driver
            Class.forName("com.mysql.jdbc.Driver");

            // Establish a connection to the database
            Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb",
"username", "password");

            // Create a SQL statement
            Statement statement = connection.createStatement();

            // Execute a query
            ResultSet resultSet = statement.executeQuery("SELECT * FROM employees");

            // Process the result set
            while (resultSet.next()) {
                System.out.println(resultSet.getString("name"));
            }

            // Close resources
            resultSet.close();
            statement.close();
            connection.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Hibernate Example:

```java
@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private int id;
    private String name;

    // Getters and setters
}

public class HibernateDemo {
    public static void main(String[] args) {
        SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();

        // Create a new employee
        Employee employee = new Employee();
        employee.setName("John Doe");

        // Save the employee to the database
        session.beginTransaction();
        session.save(employee);
        session.getTransaction().commit();

        // Retrieve an employee
        Employee retrievedEmployee = session.get(Employee.class, 1);
        System.out.println(retrievedEmployee.getName());

        session.close();
        sessionFactory.close();
    }
}
```

In the Hibernate example, you define an `Employee` entity, and Hibernate takes care of the database interaction for you. This code is more abstract and portable compared to the low-level JDBC code.

SPRING
==========
Spring is a widely used framework for building enterprise-level Java applications. It provides comprehensive infrastructure support for developing Java-based applications, making it easier to create robust, maintainable, and scalable software. Spring offers various modules for different purposes, such as dependency injection, aspect-oriented programming, data access, messaging, and more. In a Spring application, you can develop components as POJOs (Plain Old Java Objects) and use Spring to manage these components, their dependencies, and other aspects of the application.

Here's a simplified example of a traditional Spring application (non-Spring Boot) to help you understand the fundamental concepts. We'll create a simple Spring application that manages a list of users.

Step 1: Set Up the Project

Create a Maven project and add the necessary dependencies to your `pom.xml`.

pom.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>spring-user-app</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.10.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

Step 2: Create a User Entity

Create a `User` entity class to represent users:

```java
public class User {
    private long id;
    private String username;
    private String email;

    // Getters and setters
}
```

Step 3: Create a UserRepository

Create a repository class to manage user data:

```java
public class UserRepository {
    private List<User> users = new ArrayList<>();

    public void addUser(User user) {
        users.add(user);
    }

    public List<User> getUsers() {
        return users;
    }
```

}
```

**Step 4: Create a Controller**

Create a controller class to handle user-related operations:

```java
public class UserController {
    private UserRepository userRepository;

    public UserController(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public void addUser(User user) {
        userRepository.addUser(user);
    }

    public List<User> getUsers() {
        return userRepository.getUsers();
    }
}
```

Step 5: Create the Spring Configuration

Create a Spring configuration XML file to define Spring beans and wire them together. Let's call it `applicationContext.xml`:

applicationContext.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/
schema/beans/spring-beans.xsd">

    <bean id="userRepository" class="com.example.UserRepository" />
    <bean id="userController" class="com.example.UserController">
        <constructor-arg ref="userRepository" />
    </bean>
</beans>
```

Step 6: Create the Main Application

Create a Java class to load the Spring application context and use the defined beans:
```

```java
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

        UserController userController = context.getBean(UserController.class);

        // Use userController to handle user operations
    }
}
```

In this example, we manually configure Spring beans and their dependencies using an XML configuration file. This demonstrates the basics of a traditional Spring application. You can expand this foundation to build more complex applications.

SPRING AND SPRINGBOOT
==========

Spring and **Spring Boot** are two frameworks in the Spring ecosystem for building Java applications. Here's an overview of each and the key differences between them:

Spring:

1. **Core Container:** The Spring Framework, often referred to as "Spring," provides a comprehensive programming and configuration model for building enterprise applications. It includes modules for managing application configuration, handling dependency injection, and providing core features like AOP (Aspect-Oriented Programming) and transactions.

2. **XML Configuration:** Spring applications are often configured using XML files or Java-based configurations. Configuration details, such as bean definitions and dependency injection, are explicitly defined in these configuration files.

3. **Fine-Grained Control:** Spring offers a high degree of customization and fine-grained control over application components and their behavior. Developers have the flexibility to configure and define application components in a highly customizable way.

4. **Requires Boilerplate Code:** Developing Spring applications typically involves writing more boilerplate code, such as XML configuration or Java-based configuration classes, to set up the application context and configure beans.

5. **Flexible for Complex Applications:** Spring is well-suited for complex enterprise applications where custom configuration and extensive control are required.

Spring Boot:

1. **Opinionated Framework:** Spring Boot is an opinionated framework built on top of the Spring Framework. It simplifies the setup and development of Spring applications by providing a set of defaults and conventions.

2. **Auto-Configuration:** Spring Boot introduces auto-configuration, which automatically configures many aspects of the application based on classpath dependencies. Developers can override these defaults when needed.

3. **Annotations and Defaults:** Spring Boot encourages the use of annotations and sensible defaults. This leads to reduced configuration and less boilerplate code.

4. **Microservices-Ready:** Spring Boot is popular for building microservices due to its ease of development and deployment. It's optimized for creating stand-alone, production-ready Spring-based applications with minimal effort.

5. **Quick Start:** Spring Boot provides a quick and easy way to start new projects. Developers can create a new Spring Boot application with minimal configuration and immediately start building features.

Key Differences:

1. **Configuration:** Spring applications typically require explicit configuration through XML files or Java-based configuration classes, while Spring Boot encourages convention over configuration and relies on auto-configuration.

2. **Boilerplate Code:** Spring applications often involve writing more boilerplate code for setting up the application context and defining beans. Spring Boot reduces the need for boilerplate code.

3. **Complexity:** Spring offers a high degree of customization and is suitable for complex, custom enterprise applications. Spring Boot is designed to simplify development and is well-suited for microservices and rapid application development.

4. **Default Settings:** Spring Boot provides default settings and opinions to get started quickly, while Spring leaves many configuration decisions to the developers.

In summary, Spring is a comprehensive framework that offers fine-grained control and flexibility, while Spring Boot is an opinionated framework that simplifies Spring application development by providing defaults and conventions. The choice between Spring and Spring Boot depends on the specific project requirements and the development team's preferences. Spring Boot is often preferred for new projects, prototypes, and microservices.

Microservice Architecture with Spring Boot:
==========

Microservices is an architectural approach where a software application is divided into a collection of small, independent, and loosely coupled services, each responsible for specific functionality. Spring Boot, a popular Java framework, is often used to build microservices due to its ease of development and the tools it provides.

Microservice in Spring Boot:

A **microservice in Spring Boot** is a standalone, independently deployable application that performs a specific function within a larger software system. Each microservice typically has its own database and communicates with other microservices through well-defined APIs (often using HTTP/REST).

Uses of Microservices with Spring Boot:

- **Scalability:** Microservices can be independently scaled, allowing you to allocate more resources to specific services that need it, while leaving others unaffected.

- **Maintainability:** Smaller codebases are easier to manage and update. Changes in one service don't affect others, making maintenance more manageable.

- **Faster Development:** Smaller teams can develop and deploy microservices independently, accelerating development cycles.

- **Resilience:** Isolating services reduces the impact of failures. If one service fails, it doesn't bring down the entire system.

- **Technology Diversity:** Different microservices can use different technologies and databases to choose the right tools for each task.

Example of a Microservice in Spring Boot:

Consider an e-commerce platform. You might have separate microservices for user authentication, product catalog, order processing, and payment handling.

Here's a simplified example of a product catalog microservice using Spring Boot:

ProductService.java:
```java
@RestController
@RequestMapping("/products")
public class ProductService {

    @GetMapping("/{productId}")
    public Product getProductById(@PathVariable Long productId) {
        // Retrieve product information from the database and return it
    }

    @PostMapping
    public Product addProduct(@RequestBody Product product) {
```

```
        // Save the new product to the database and return it
    }

    // Other product-related endpoints
}
```

In this example:

- `ProductService` is a Spring Boot application serving as a microservice for product-related operations.

- It defines endpoints to retrieve a product by ID and add a new product.

- Each operation is independent, and the service can be deployed and scaled on its own.

Differences Between a Normal Spring Boot Application and a Microservices Application:

1. **Monolith vs. Distributed:** A normal Spring Boot application is often a monolith, where all functionalities are tightly integrated into a single codebase. In contrast, a microservices application is distributed, with functionalities divided into separate services.

2. **Size:** Normal Spring Boot applications tend to be larger, with all functionalities in one place. Microservices are smaller, focused on specific tasks.

3. **Complexity:** Microservices applications can be more complex to manage due to the need for service coordination, inter-service communication, and distributed data.

4. **Scalability:** Microservices allow for fine-grained scalability, while a monolith typically scales as a single unit.

5. **Independent Deployment:** Microservices can be deployed independently. In a monolith, changes may require redeploying the entire application.

6. **Technology Stack:** In a monolith, the entire application uses the same technology stack. In microservices, different services can use different stacks.

7. **Maintenance:** Microservices can be easier to maintain as changes in one service don't affect others. In a monolith, changes can be riskier.

The choice between a normal Spring Boot application and a microservices architecture depends on the specific project requirements, scalability needs, development team size, and other factors. Microservices offer advantages in terms of scalability and maintainability but introduce complexity in managing the interactions between services.

Normal Spring Boot Monolithic Application:

In a monolithic Spring Boot application, all functionalities are bundled together in a single application. Here's an example of a simplified monolithic application for a basic e-commerce platform:

MonolithicEcommerceApplication.java:

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

ProductController.java:

```java
@RestController
@RequestMapping("/products")
public class ProductController {
    @GetMapping("/{productId}")
    public Product getProduct(@PathVariable Long productId) {
        // Retrieve and return the product from the database
    }

    @PostMapping
    public Product addProduct(@RequestBody Product product) {
        // Save the new product to the database and return it
    }

    // Other product-related endpoints
}
```

In this monolithic example:

- There's a single Spring Boot application that handles all aspects of the e-commerce platform, including product management.

- The `ProductController` defines endpoints for retrieving and adding products.

Microservices Architecture:

Now, let's look at a simplified microservices architecture for the same e-commerce platform. We'll split the product management into a separate microservice.

ProductService.java:

```java
```

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

ProductController.java (Microservice):

```java
@RestController
@RequestMapping("/products")
public class ProductController {
    @GetMapping("/{productId}")
    public Product getProduct(@PathVariable Long productId) {
        // Retrieve and return the product from the microservice's database
    }

    @PostMapping
    public Product addProduct(@RequestBody Product product) {
        // Save the new product to the microservice's database and return it
    }

    // Other product-related endpoints
}
```

In this microservices example:

- There are two separate Spring Boot applications: the **ProductService** microservice and the **E-commerceGateway** application (not shown here) that routes requests to various microservices.

- The **ProductService** microservice handles product-related operations, just as in the monolithic example. It runs independently and can be scaled separately.

- Other microservices can handle different parts of the e-commerce platform, and they can use various technologies and databases as needed.

The key difference is that in the microservices architecture, the functionalities are divided into smaller, independently deployable services, whereas the monolithic application handles everything in a single codebase. Microservices offer advantages in terms of scalability, maintainability, and technology stack diversity but introduce complexities in terms of service coordination and inter-service communication.

SPRING-BOOT
==========

application.properties file
==========
#changing port number
server.port=8081

spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/practice
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

#Logging the sql queries
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type=TRACE
spring.jpa.show-sql=true

#Write a sql query beautiful on console
spring.jpa.properties.hibernate.format_sql=true

WHAT IS SPRING-BOOT
==========

Spring Boot is a Java-based framework that simplifies the development of standalone, production-ready applications.
It is part of the larger Spring ecosystem, but it focuses on making it easy to create Spring applications with minimal configuration.
Spring Boot provides a set of conventions and defaults that enable developers to build applications quickly and efficiently.
1. Simple web application.
==========

```
String data = "Welcome To Plant %s!";

    @GetMapping("/")
public String message(@RequestParam(value = "name", defaultValue = "Nani") String name){
    return String.format(data,name);
}
```

--> @RequestParam : it is telling Spring to expect a name value in the request, but if itâ s not there, it will use the word "Nani" by default.

http://localhost:8081/?name=Rajesh

output: Welcome To Plant Rajesh!

2. Spring Boot Restful Services:
==========

1. Spring Boot is often used to create RESTful web services, which are web services that use HTTP methods (GET, POST, PUT, DELETE) to perform CRUD (Create, Read, Update, Delete) operations on resources.
2. Spring Boot simplifies the creation of RESTful services by providing annotations like @RestController to define REST endpoints, and it includes libraries to handle HTTP requests and responses.
3. You can easily build RESTful APIs to expose data or services over HTTP.
4. These services are typically used for communication between systems, mobile apps, web applications, and other clients

3. SpringBoot Data JPA
==========

Spring Data JPA is a part of the larger Spring Data project.
It is a framework that simplifies data access in Java applications that use the Java Persistence API (JPA) to interact with relational databases.
JPA is a Java specification for working with databases in an object-oriented way, allowing developers to work with databases using Java classes and objects instead of SQL queries.

1. Automatic Repository Creation: Spring Boot Data JPA can automatically create repository interfaces for your domain models, which significantly reduces the amount of boilerplate code you need to write for basic CRUD operations.

2. Query Methods: It allows you to create custom query methods by simply defining method names following a specific naming convention. Spring Data JPA translates these method names into SQL queries automatically.

3. Pagination and Sorting: Built-in support for paging and sorting of query results.

4. Transactional Support: It simplifies transaction management, ensuring that database operations are executed within a single transaction context.

5. Custom Queries: You can write native SQL queries, JPQL (Java Persistence Query Language) queries, or use the Criteria API to create complex queries.

RequestBod for List<User>
==========


```
[
   {
      "firstName":"NaniBabu",
      "lastName":"Pallapu",
      "loginName":"NPallapu",
      "password":"Hyderabad@369",
      "email":"nanipallapu369@mail.com",
      "phone":"9392590089"
   },
```

```
    {
    "firstName":"Priyanka",
    "lastName":"Bandi",
    "loginName":"PBandi",
    "password":"Priya@369",
    "email":"pryanka369@mail.com",
    "phone":"8008142536"
  },

    {
    "firstName":"Divya",
    "lastName":"Kommirisetti",
    "loginName":"DKommirisetti",
    "password":"Divya@2002",
    "email":"divya2002@mail.com",
    "phone":"9567892345"
  }
]
```

@RequestBody for Single User Object
==========

```
{
    "firstName":"Divya",
    "lastName":"Kommirisetti",
    "loginName":"DKommirisetti",
    "password":"Divya@2002",
    "email":"divya2002@mail.com",
    "phone":"9567892345"
  }
```

Dependency Injection (DI)
==========
In Spring Boot is a design pattern and framework feature that helps manage the dependencies between various components in a Spring application.
The primary goal of DI is to achieve the Inversion of Control (IoC) principle, where the control of creating and managing objects is shifted from the components themselves to an external entity or container.
In the context of Spring Boot, this container is the Spring IoC container.

Here's why Dependency Injection is used and its advantages:

1. Loose Coupling:** DI promotes loose coupling between components, making it easier to maintain, extend, and test the application. Components don't need to know how their dependencies are created or configured; they simply rely on the provided interfaces.

2. Reusability:** Components can be reused across different parts of the application, as they are not

tightly bound to specific implementations of their dependencies.

3. Testability:** DI allows for easier unit testing. You can inject mock or stub dependencies during testing to isolate and verify the behavior of individual components.

4. Configurability:** DI enables you to configure the application's components and dependencies, often through configuration files or annotations, without changing the code. This makes the application more adaptable to different environments or configurations.

Here's a simple example of Dependency Injection in a Spring Boot application:

UserService.java:
```java
@Service
public class UserService {
    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User getUserById(Long id) {
        return userRepository.findById(id).orElse(null);
    }
}
```

UserRepository.java:
```java
public interface UserRepository extends JpaRepository<User, Long> {
}
```

In this example:

- `UserService` is a Spring service class that depends on `UserRepository` for fetching user data.

- `UserRepository` is an interface that extends `JpaRepository`. Spring Data JPA will provide the actual implementation of this interface.

- Constructor injection is used in `UserService` to receive an instance of `UserRepository`. The `@Autowired` annotation tells Spring to inject the dependency.

- By using DI, `UserService` is loosely coupled with the concrete implementation of `UserRepository`. The Spring IoC container takes care of creating and providing instances of `UserRepository` to `UserService`.

In your Spring Boot application, you can configure and define beans in various ways, such as through annotations or XML configuration files. Spring Boot's auto-configuration and component scanning capabilities help simplify the DI process, allowing you to focus on your application's logic rather than the details of object

creation and wiring.

Spring Boot MVC (Model-View-Controller)
==========

Spring Boot MVC is an architectural pattern and framework for building web applications in Spring Boot. It provides a structured approach to handling web requests and responses, separating an application into three interconnected components:

1. Model: Represents the application's data and business logic. It contains the information that the application operates on.

2. View: Represents the presentation layer of the application, responsible for rendering the user interface and displaying data from the Model.

3. Controller: Acts as an intermediary between the Model and View. It processes incoming HTTP requests, interacts with the Model to retrieve data, and selects the appropriate View for rendering the response.

Uses of Spring Boot MVC:

Spring Boot MVC is widely used for building web applications, both simple and complex. Some common use cases include:

1. Web Applications: Building web applications with dynamic content and user interaction.

2. RESTful APIs: Creating RESTful web services to expose data and functionality to clients.

3. Authentication and Authorization: Implementing user authentication and authorization for web applications.

4. Form Handling: Handling forms and user input in web applications.

5. View Templating: Rendering HTML or other view templates to generate dynamic web pages.

Here's an example of a simple Spring Boot MVC application to demonstrate the use of the Model, View, and Controller components:

Model: User.java
```java
public class User {
    private String username;
    private String email;

    // Getters and setters
}
```

View: userForm.html

```html
<!DOCTYPE html>
<html>
<head>
    <title>User Registration</title>
</head>
<body>
    <h1>User Registration</h1>
    <form method="post" action="/register">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required><br>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required><br>
        <input type="submit" value="Register">
    </form>
</body>
</html>
```

Controller: UserController.java
```java
@Controller
public class UserController {

    @GetMapping("/registration")
    public String showRegistrationForm(Model model) {
        model.addAttribute("user", new User());
        return "userForm";
    }

    @PostMapping("/register")
    public String registerUser(@ModelAttribute("user") User user) {
        // Process and store the user data
        return "registrationSuccess";
    }
}
```

In this example:

- `User` represents the Model, holding user data.

- The `userForm.html` file is the View, displaying a user registration form.

- `UserController` acts as the Controller, handling the HTTP requests. The `showRegistrationForm` method displays the registration form, and the `registerUser` method processes the submitted form.

This is a simplified example, but it demonstrates the essential components of a Spring Boot MVC application. When a user accesses the `/registration` URL, the registration form is displayed. When the form is submitted, the `registerUser` method processes the user's input, and the response is rendered using a view template.

Spring Boot simplifies the setup and configuration of such applications, making it easier to develop web-based systems.