SPRING BOOT PROJECT - CREDIT CARD MANAGEMENT SYSTEM

```
/credit-card-management-system
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com
│   │   │       └── creditcard
│   │   │           ├── controller
│   │   │           │   └── CardController.java
│   │   │           ├── model
│   │   │           │   ├── Card.java
│   │   │           │   └── User.java
│   │   │           ├── repository
│   │   │           │   ├── CardRepository.java
│   │   │           │   └── UserRepository.java
│   │   │           ├── service
│   │   │           │   └── CardService.java
│   │   │           ├── exception
│   │   │           │   └── ResourceNotFoundException.java
│   │   ├── resources
│   │   │   ├── application.properties
│   │   │   └── static
│   │   └── test
│   │       └── java
│   │           └── com
│   │               └── creditcard
```

```
│   │               └── CardServiceTest.java
├── build.gradle   ← Gradle dependencies and configurations
└── build          ← Generated folder after Gradle builds the project
```

Gradle Plugin Configuration:

```
plugins {
    id 'java' // Java plugin to compile Java code
    id 'org.springframework.boot' version '3.4.1' // Spring Boot plugin for buil
ding Spring applications
    id 'io.spring.dependency-management' version '1.1.7' // For managing depende
ncies in a consistent way
}
```

Java Toolchain Configuration:

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(21) // Specifies Java version 2
1 for the project
    }
}
```

Configurations:

```
configurations {
    compileOnly {
        extendsFrom annotationProcessor // Enables annotation processing for Lom
bok
    }
}
```

Repositories:

```
groovy
Copy code
repositories {
```

```
    mavenCentral() // Using Maven Central for dependencies
}
```

Dependencies:

```
groovy
Copy code
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa' // JP
A starter for database interaction
    implementation 'org.springframework.boot:spring-boot-starter-web' // Spring
Web starter for REST APIs
    compileOnly 'org.projectlombok:lombok' // Lombok for generating boilerplate
code (getters/setters)
    developmentOnly 'org.springframework.boot:spring-boot-devtools' // For devel
opment-time features like auto-restart
    runtimeOnly 'org.postgresql:postgresql' // PostgreSQL JDBC driver for databa
se connection
    annotationProcessor 'org.projectlombok:lombok' // Enables Lombok annotation
processing
    testImplementation 'org.springframework.boot:spring-boot-starter-test' // Te
st dependencies
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher' // JUnit test r
untime
}
```

Testing Configuration:

```
groovy
Copy code
tasks.named('test') {
    useJUnitPlatform() // Ensures JUnit 5 is used for testing
}
```

Database Configuration:
You need to configure the PostgreSQL connection in `src/main/resources/application.properties`

```
spring.datasource.url=jdbc:postgresql://localhost:5432/credit_card_db
spring.datasource.username=your_username
spring.datasource.password=your_password
spring.jpa.hibernate.ddl-auto=update
```

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

Here's a detailed explanation of each line in the given Spring Boot `application.properties` configuration for the **Credit Card Service** project:

1. `spring.application.name=Credit_Card_Service`

- **Purpose**: This property sets the name of the Spring Boot application.
- **Explanation**: The application name `Credit_Card_Service` will be used for logging, metrics, and for identifying the service in a microservices environment or a cloud platform.

2. `spring.datasource.url=jdbc:postgresql://localhost:5432/credit_card_db`

- **Purpose**: This specifies the URL of the database Spring Boot should connect to.
- **Explanation**:
  - `jdbc:postgresql://`: The JDBC URL scheme for PostgreSQL.
  - `localhost`: The address of the PostgreSQL database. In this case, it's running locally on the same machine.
  - `5432`: The default port for PostgreSQL.
  - `credit_card_db`: The name of the PostgreSQL database that the application will use.

3. `spring.datasource.username=postgres`

- **Purpose**: This sets the username for connecting to the PostgreSQL database.
- **Explanation**: `postgres` is the default username for PostgreSQL databases. In a production environment, you should replace this with a more secure username.

4. `spring.datasource.password=Anu@9382`

- **Purpose**: This defines the password for the username specified in `spring.datasource.username`.
- **Explanation**: `Anu@9382` is the password for the `postgres` user. This password is required to authenticate the connection to the database.

5. `spring.jpa.hibernate.ddl-auto=update`

- **Purpose**: This property defines the Hibernate's behavior regarding schema generation.
- **Explanation**:
  - `update`: Hibernate will attempt to update the schema (database structure) automatically based on changes in the entity classes. It will add new columns and tables if necessary, but it won't drop existing tables.
  - This is useful during development, but for production environments, it is recommended to use `validate` (which only checks the schema) or `none` (to not modify the schema at all).

6. `spring.jpa.show-sql=true`

- **Purpose**: This enables the logging of SQL statements generated by Hibernate.
- **Explanation**: When set to `true`, every SQL query executed by Hibernate will be shown in the console or logs. This is helpful for debugging and understanding the queries being generated.

7. `spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect`

- **Purpose**: This specifies the Hibernate dialect that tells Hibernate how to generate SQL queries for PostgreSQL.
- **Explanation**:
  - `org.hibernate.dialect.PostgreSQLDialect`: This dialect is optimized for PostgreSQL and ensures that Hibernate generates PostgreSQL-specific SQL syntax.
  - It helps Hibernate map Java types to PostgreSQL-specific data types (e.g., `String` to `VARCHAR`).

Summary:

- **spring.application.name**: Sets the application name for identification.
- **spring.datasource.url**: Specifies the JDBC URL for PostgreSQL.
- **spring.datasource.username** and **spring.datasource.password**: Provide the credentials for connecting to the database.
- **spring.jpa.hibernate.ddl-auto**: Controls Hibernate's schema management behavior (e.g., `update` will automatically adjust the schema).
- **spring.jpa.show-sql**: Enables SQL query logging for debugging.
- **spring.jpa.properties.hibernate.dialect**: Specifies the PostgreSQL-specific dialect for Hibernate to generate correct SQL syntax.

These settings help Spring Boot connect to the PostgreSQL database and configure Hibernate for ORM (Object Relational Mapping) and schema management.

CONTROLLER CLASS, SERVICE CLASS, REPOSITORY CLASS

In a Spring Boot application, the **Controller**, **Service**, and **Repository** layers have distinct roles that help maintain a clean architecture and separation of concerns. Here's what each class does:

1. Controller Class

- **Purpose**: Handles incoming HTTP requests, processes them, and returns the appropriate HTTP responses.
- **Responsibilities**:
  - Receives requests from the client (e.g., web browser or mobile app).
  - Calls the appropriate methods from the service layer to process business logic.
  - Returns data to the client in the form of HTTP responses (often as JSON or HTML).

- **Annotations**:
  - `@RestController`: Marks the class as a controller that handles REST API requests. It automatically includes `@ResponseBody`, meaning it will return data directly (typically in JSON format).
  - `@RequestMapping`, `@GetMapping`, `@PostMapping`: Maps incoming HTTP requests to methods in the controller.
- **Example**:

```
@RestController
@RequestMapping("/api/credit-cards")
public class CreditCardController {

    @Autowired
    private CreditCardService creditCardService;

    @GetMapping("/{id}")
    public ResponseEntity<CreditCard> getCreditCard(@PathVariable Long id) {
        CreditCard creditCard = creditCardService.getCreditCardById(id);
        return ResponseEntity.ok(creditCard);
    }

    @PostMapping
    public ResponseEntity<CreditCard> createCreditCard(@RequestBody CreditCar
d creditCard) {
        CreditCard createdCreditCard = creditCardService.createCreditCard(cre
ditCard);
        return ResponseEntity.status(HttpStatus.CREATED).body(createdCreditCa
rd);
    }
}
```

2. Service Class

- **Purpose**: Contains the business logic of the application. It acts as a bridge between the controller and repository layers.
- **Responsibilities**:
  - Performs the core logic of the application, such as processing data, validating business rules, and interacting with the repository layer.
  - The service class usually delegates the database interaction to the repository and handles any necessary transformation or logic.
- **Annotations**:
  - `@Service`: Marks the class as a service component.
- **Example**:

```
@Service
public class CreditCardService {

    @Autowired
    private CreditCardRepository creditCardRepository;

    public CreditCard getCreditCardById(Long id) {
        return creditCardRepository.findById(id).orElseThrow(() -> new Resour
ceNotFoundException("Credit card not found"));
    }

    public CreditCard createCreditCard(CreditCard creditCard) {
        return creditCardRepository.save(creditCard);
    }
}
```

3. Repository Class

- **Purpose**: Responsible for data access and interactions with the database (typically using JPA/Hibernate).
- **Responsibilities**:
  - Handles CRUD (Create, Read, Update, Delete) operations for entities.
  - Can perform complex database queries or use Spring Data JPA's predefined methods.
  - Acts as an abstraction layer between the application and the database, providing methods to retrieve and store data.
- **Annotations**:
  - `@Repository`: Marks the class as a repository, which is a type of DAO (Data Access Object). This annotation also enables exception translation (converts database-related exceptions into Spring's `DataAccessException`).
- **Example**:

```
@Repository
public interface CreditCardRepository extends JpaRepository<CreditCard, Long>
{
    // Custom query methods can be added here if needed
    Optional<CreditCard> findByCardNumber(String cardNumber);
}
```

Overview of How They Interact:

- **Controller**: Receives HTTP requests from the client. For example, when a user sends a request to get credit card details, the controller invokes the `CreditCardService`.

- **Service**: Processes the business logic and interacts with the repository to fetch, save, or update data. For instance, it might call the repository to retrieve a credit card by ID or save a new credit card.
- **Repository**: Interacts directly with the database to perform operations like fetching or saving data. It provides methods like `findById()`, `save()`, and others.

Simplified Flow:

1. **Client sends an HTTP request** →
2. **Controller receives the request** →
3. **Controller calls a method in the Service** →
4. **Service interacts with the Repository to access or modify the database** →
5. **Repository performs database operations** →
6. **Service processes the results and returns the data to the Controller** →
7. **Controller returns the response to the client**.

This design ensures that:

- **Separation of concerns** is maintained (Controller handles HTTP requests, Service handles business logic, Repository handles data access).
- Code is **modular and maintainable**.
- Each layer can be **independently tested**.

```
CreditCardReportingService/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com.example.creditcardreporting/
│   │   │       ├── controller/
│   │   │       ├── service/
│   │   │       ├── repository/
│   │   │       ├── entity/
│   │   │       └── dto/
│   │   ├── resources/
│   │       ├── application.properties
│   │       └── schema.sql
│   │       └── data.sql
│   └── test/
│       └── java/
│           └── com.example.creditcardreporting/
├── build.gradle
└── settings.gradle
```

Mapping from DTO to Entity:

Typically, when using DTOs, you'll map the data from the DTO (which is meant for data transfer, e.g., JSON or request body) to the entity class (`Company`) that corresponds to the database entity.

For example, your mapping code should look like this:

```
// Assuming company is an instance of Company (entity class)
Company company = new Company();  // Initialize a Company object

company.setName(companyDTO.getName());
company.setEmail(companyDTO.getEmail());
company.setAge(companyDTO.getAge());
```

This is assuming `companyDTO` is an instance of `CompanyDTO`, and `company` is an instance of the `Company` entity.

DEPENDENCY INJECTION

the
`@Enumerated` annotation is used in **Spring Boot** to ensure the mapping of an enum field to a specific type in the database.

`@GeneratedValue`

- This annotation is used to specify that the value of the annotated field should be automatically generated by the underlying database.

The
`@Service` annotation marks a class as a **service** that holds business logic or service-related operations. Typically, services are responsible for handling the business rules and interacting with data access layers (repositories), but they don't directly manage HTTP requests or responses.

The
`@Autowired` annotation in Spring is used for **automatic dependency injection**, which means Spring automatically provides the required dependencies for a class, instead of you having to manually create or pass them. It is part of the **Inversion of Control (IoC)** principle, which is fundamental to the Spring framework.

`@NoArgsConstructor`: Generates a no-argument constructor for the class.

`@AllArgsConstructor`: Generates a constructor with arguments for all fields in the class.

A. Customer Interaction Workflow

Requirements:

- Add functionality for an agent to generate a lead for a customer.

Implementation Steps:

- Create a service method in `LeadService`:
  - Check if the customer exists.
  - Check if the agent exists.
  - Generate a unique reference number.
  - Save the lead to the database with the `PENDING` status.

implement a business logic that evaluates the credit card lead by checking the **customer's civil score** and **bank balance**. Depending on these two factors, the lead will either be:

- **Approved** if both conditions meet.
- **On Hold** if the balance is insufficient but the civil score is acceptable.
- **Rejected** if both conditions fail.

DTO

Using **DTO (Data Transfer Object)** classes is a common practice in modern software development, especially in Spring Boot applications, to improve design, maintainability, and flexibility.

Here's why we use DTOs:

---

1. Separation of Concerns

- **Entities** represent the database structure, while **DTOs** represent the structure of data that flows between layers (e.g., controller and service).
- Keeping entities and DTOs separate allows you to modify one without affecting the other. For instance, if you change a database schema, your API contracts remain intact.

---

2. Encapsulation and Abstraction

- DTOs allow you to expose only the necessary fields to the client, hiding sensitive or irrelevant data (e.g., internal IDs, audit logs, or passwords).
- You can provide only what's needed for a specific use case, reducing data bloat in the API response or request.

---

3. Validation

- You can define constraints (e.g., `@NotNull`, `@Email`, `@Size`) directly in the DTO, ensuring incoming data is valid before processing.

- Example:

```
public class AgentDTO {
    @NotNull
    private String name;

    @Email
    private String email;

    private Integer companyId;
}
```

4. Flexibility and Transformation

- DTOs allow you to create custom representations of your data for specific use cases.
- Example: A DTO for creating an agent might not require `agentId` (which is auto-generated), while a DTO for updating an agent might include it.

5. Decoupling Layers

- By using DTOs, the controller and service layers are decoupled from the persistence layer (entities).
- This improves code reusability and enables easy swapping of technologies or frameworks.

6. Improved Testability

- DTOs make unit testing easier by isolating business logic from persistence concerns.
- You can test business logic without worrying about database interactions.

7. Reduced Data Size

- Avoid transferring large objects with unnecessary fields to minimize payload size and improve performance.

Example: Without and With DTOs

Without DTO:

```
public Agent createAgent(@RequestBody Agent agent) {
    return agentRepository.save(agent);
}
```

- Exposes the entire `Agent` entity, potentially revealing sensitive or unnecessary data.

With DTO:

```
public Agent createAgent(@RequestBody AgentDTO agentDTO) {
    Agent agent = new Agent();
    agent.setName(agentDTO.getName());
    agent.setEmail(agentDTO.getEmail());
    agent.setCompany(companyRepository.findById(agentDTO.getCompanyId()).orElseT
hrow(...));
    return agentRepository.save(agent);
}
```

- You control what fields are passed and avoid directly exposing database structures.

Summary

Using DTOs leads to better:

- **Security**: By limiting exposed data.
- **Maintainability**: By separating API contracts from database schemas.
- **Flexibility**: By transforming data for specific use cases.
- **Testability**: By isolating data structures from the persistence layer.

This pattern ensures that your application is more scalable, modular, and easier to maintain over time.

Enterprice level application using kafka for customer.

docker desktop

akHQ.io