

Sistema de Gestión de Tarjetas Bancarias - API REST

Descripción del Proyecto

API REST desarrollada con Spring Boot para la gestión de tarjetas bancarias, implementando operaciones CRUD básicas y activación de tarjetas. El proyecto utiliza JPA/Hibernate para la persistencia de datos con SQL Server.

Arquitectura del Proyecto

Estructura de Paquetes

```
com.prueba_tecnica.prueba_fredy
├── controller/           # Capa de presentación (REST Controllers)
│   └── CardController.java
├── service/              # Capa de lógica de negocio
│   ├── CardService.java
│   └── CardServiceImpl.java
├── repository/           # Capa de acceso a datos
│   └── CardRepository.java
└── entity/               # Capa de modelo de datos
    └── Card.java
```

Interacción entre Capas

El proyecto sigue una arquitectura en capas bien definida:

1. Controller → Service → Repository → Database

Cliente HTTP → CardController → CardService → CardRepository → SQL Server

Flujo de datos:

- El Controller recibe las peticiones HTTP y delega la lógica de negocio al Service
- El Service implementa las reglas de negocio y utiliza el Repository para persistencia
- El Repository abstrae el acceso a datos utilizando JPA
- La Entity representa el modelo de dominio y la tabla en base de datos

Configuración de la Aplicación

Archivo `application.properties`

```
spring.application.name=prueba-fredy

# Configuración de conexión a SQL Server
spring.datasource.url =
jdbc:sqlserver://localhost:1433;databaseName=PruebaFredy;encrypt=true;trustServerCertificate=true
```

```
spring.datasource.username = sa
spring.datasource.password = root
spring.datasource.driver-class-name = com.microsoft.sqlserver.jdbc.SQLServerDriver

# Configuración de JPA/Hibernate
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.SQLServerDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.sql-show=true
```

Explicación de Propiedades:

- spring.application.name: Nombre de la aplicación
- spring.datasource.url: URL de conexión a SQL Server
 - o **local host: 1433**: Host y puerto del servidor
 - o **databaseName=PruebaFredy**: Nombre de la base de datos
 - o **encrypt=true**: Habilita encriptación
 - o **trustServerCertificate=true**: Confía en certificados autofirmados
- spring.datasource.username/password: Credenciales de acceso
- spring.datasource.driver-class-name: Driver JDBC de SQL Server
- spring.jpa.hibernate.dialect: Dialecto SQL específico de SQL Server
- spring.jpa.hibernate.ddl-auto=update: Actualiza el esquema automáticamente
- spring.jpa.sql-show=true: Muestra las consultas SQL en consola

Cambiar a Otra Base de Datos

Para MySQL:

```
spring.datasource.url=jdbc:mysql://localhost:3306/PruebaFredy
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

Dependencia Maven requerida:

```
<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <scope>runtime</scope>
</dependency>
```

Para PostgreSQL:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/PruebaFredy
spring.datasource.username=postgres
spring.datasource.password=root
```

```
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

Dependencia Maven requerida:

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
```

Para H2 (Base de datos en memoria - Testing):

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.h2.console.enabled=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
```



Modelo de Datos

Entidad Card

```
@Entity
@Table(name="cards")
public class Card {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(unique=true, length=16, nullable=false)
    private Long cardNumber;

    @Column(length=200, nullable=false)
    private String cardName;

    @Column(length=7, nullable=false)
    private String cardExpireDate;

    @Column(length=10, nullable=false)
    private String status;

    @Column(length=10, nullable=false)
    private Long cash;
}
```

Campos:

- id: Identificador único autogenerado
- cardNumber: Número de tarjeta (16 dígitos, único)
- cardName: Nombre del titular
- cardExpireDate: Fecha de expiración (formato: MM/YYYY)
- status: Estado de la tarjeta (Inactive/Active)

- cash: Saldo disponible

Endpoints de la API

1. Generar Nueva Tarjeta

Endpoint: **POST /card/number**

Descripción: Crea una nueva tarjeta con número generado automáticamente.

Request Body:

```
{
  "cardName": "Juan Pérez"
}
```

Response:

```
{
  "id": 1,
  "cardNumber": 2233449876543211,
  "cardName": "Juan Pérez",
  "cardExpi reDate": "10/2028",
  "status": "Inacti ve",
  "cash": 0
}
```

Lógica de Generación:

- Número de producto fijo: **223344**
- Número aleatorio de 9 dígitos
- Dígito final: **1**
- Formato final: **223344XXXXXXXXX1** (16 dígitos)
- Fecha de expiración: 3 años desde la fecha actual
- Estado inicial: **Inacti ve**
- Saldo inicial: **0**

Ejemplo de consumo con cURL:

```
curl -X POST http://localhost:8080/card/number \
-H "Content-Type: appl i cati on/j son" \
-d '{"cardName": "Juan Pérez"}'
```

Ejemplo con Postman:

- Method: POST
- URL: **http://localhost:8080/card/number**
- Headers: **Content-Type: appl i cati on/j son**
- Body (raw - JSON):

```
{
  "cardName": "Juan Pérez"
}
```

2. Activar Tarjeta

Endpoint: **POST** /card/enroll

Descripción: Activa una tarjeta existente cambiando su estado a "Active".

Request Body:

```
{
  "id": 1
}
```

Response:

```
{
  "id": 1,
  "cardNumber": 2233449876543211,
  "cardName": "Juan Pérez",
  "cardExpiryDate": "10/2028",
  "status": "Active",
  "cash": 0
}
```

Ejemplo de consumo con cURL:

```
curl -X POST http://localhost:8080/card/enroll \
-H "Content-Type: application/json" \
-d '{"id":1}'
```

3. Eliminar Tarjeta

Endpoint: **DELETE** /card/{id}

Descripción: Elimina una tarjeta por su ID.

Parámetros:

- **id** (path): ID de la tarjeta a eliminar

Response: **200 OK** (sin contenido)

Ejemplo de consumo con cURL:

```
curl -X DELETE http://localhost:8080/card/1
```

Ejemplo con JavaScript (Fetch API):

```
fetch('http://localhost:8080/card/1', {
  method: 'DELETE'
})
.then(response => {
  if(response.ok) {
    console.log('Tarjeta eliminada exitosamente');
  }
})
```

```
})  
.catch(error => console.error('Error: ', error));
```

🎯 Principios SOLID Aplicados

1. Single Responsibility Principle (SRP)

Cada clase tiene una única responsabilidad:

- CardController: Manejo de peticiones HTTP
- CardService/CardServiceImpl: Lógica de negocio
- CardRepository: Acceso a datos
- Card: Representación del modelo de datos

2. Open/Closed Principle (OCP)

- El uso de interfaces (`CardService`) permite extender funcionalidad sin modificar código existente
- Se pueden agregar nuevas implementaciones de `CardService` sin afectar el controller

3. Liskov Substitution Principle (LSP)

- `CardServiceImpl` implementa `CardService` y puede sustituirse por cualquier otra implementación
- El controller depende de la abstracción (`CardService`), no de la implementación concreta

4. Interface Segregation Principle (ISP)

- `CardService` define solo los métodos necesarios para operaciones con tarjetas
- No se fuerza a implementar métodos innecesarios

5. Dependency Inversion Principle (DIP)

- El controller depende de la abstracción `CardService`, no de `CardServiceImpl`
- La inyección de dependencias se maneja mediante el constructor
- Spring gestiona las dependencias automáticamente

Ejemplo de DIP en el código:

```
public class CardController {  
    private final CardService cardService; // Dependencia de abstracción  
  
    public CardController(CardService cardService) {  
        this.cardService = cardService; // Inyección por constructor  
    }  
}
```

🔑 Tecnologías Utilizadas

- Spring Boot: Framework principal

- Spring Data JPA: Persistencia de datos
- Hibernate: ORM (Object-Relational Mapping)
- SQL Server: Sistema de gestión de base de datos
- Maven: Gestión de dependencias

Instalación y Ejecución

Prerrequisitos:

- JDK 17 o superior
- SQL Server instalado y en ejecución
- Maven 3.6+

Pasos:

1. Clonar el repositorio

```
git clone <url-repositorio>  
cd prueba-fredy
```

2. Crear la base de datos

```
CREATE DATABASE PruebaFredy;
```

3. Configurar credenciales Editar `application.properties` con tus credenciales de SQL Server
4. Compilar el proyecto

```
mvn clean install
```

5. Ejecutar la aplicación

```
mvn spring-boot:run
```




La aplicación estará disponible en: `http://localhost:8080`

Consideraciones de Seguridad:

- El número de tarjeta se genera con `Random`, en producción usar `SecureRandom`
- Las credenciales en `application.properties` deberían estar en variables de entorno
- Implementar cifrado para datos sensibles como números de tarjeta

☐ Disclaimer

Este proyecto fue desarrollado completamente de forma manual, SIN el uso de:

-  Inteligencia Artificial (ChatGPT, GitHub Copilot, etc.)
-  Herramientas de generación automática de código
-  Asistentes de código basados en IA

- ✕ Google / Navegador

Todo el código fue escrito línea por línea utilizando únicamente:

- ✓ Conocimientos propios de programación
- ✓ Documentación oficial de Spring Boot
- ✓ Experiencia en desarrollo de APIs REST
- ✓ Buenas prácticas de desarrollo software

Agradecimientos

Un agradecimiento especial a Banco Santander por brindar esta oportunidad y desafío técnico. Este proyecto representa un ejercicio práctico que demuestra habilidades en:

- Desarrollo backend con Spring Boot
- Arquitectura en capas
- Principios SOLID
- Persistencia con JPA/Hibernate
- Diseño de APIs RESTful
- Gestión de bases de datos relacionales

Agradezco la confianza depositada en este proceso y la oportunidad de demostrar competencias técnicas en el desarrollo de aplicaciones empresariales.
