

# PROYECTO IV

NTTDATA Bootcamp Tech Girls Power



YULY SANDRA CHOQUE RAMOS

## Contenido

<b>NTTDATA Bootcamp Tech Girls Power</b> .....	0
<b>PROYECTO IV</b> .....	2
<b>Descripción</b> .....	2
<b>Características</b> .....	2
<b>Microservicios Implementados</b> .....	3
<b>Tecnologías Utilizadas</b> .....	4
<b>Mejora Continua</b> .....	6
<b>CustomerMS</b> .....	7
Junit y Mockito:.....	7
Cálculo de cobertura de código (jacoco) .....	9
Checkstyle .....	10
SOLID y patrones de diseño .....	11
<b>AccountMS</b> .....	14
Junit y Mockito:.....	14
Cálculo de cobertura de código (jacoco) .....	16
Checkstyle .....	17
SOLID y patrones de diseño .....	18

# PROYECTO IV

## Descripción

Este proyecto involucra la implementación de dos microservicios que interactúan con bases de datos relacionales: uno para gestionar **Clientes** y otro para gestionar **Cuentas**. Estos microservicios están diseñados para manejar de manera eficiente la información de los clientes y sus cuentas asociadas, siguiendo los principios de arquitectura de microservicios para garantizar escalabilidad, mantenibilidad y alto rendimiento.

## Características

### Pruebas Unitarias (JUnit)

Se han implementado pruebas unitarias para todos los métodos clave de las clases principales del proyecto. Estas pruebas están diseñadas para asegurar el correcto funcionamiento de los microservicios, verificando que se cumplan los comportamientos esperados en diversas condiciones. Cada método clave tiene pruebas unitarias correspondientes, ayudando a identificar posibles problemas de manera temprana en el proceso de desarrollo.

### Mocks con Mockito

Para simular el comportamiento de dependencias externas, como las interacciones con la base de datos, se ha utilizado **Mockito** para crear mocks. Esto permite tener un entorno de pruebas controlado y aislado, asegurando que las pruebas se enfoquen exclusivamente en la lógica de negocio de los microservicios sin depender de conexiones reales a bases de datos o sistemas externos.

### Cobertura de Código (JaCoCo)

Se ha integrado un informe de cobertura de código utilizando **JaCoCo**. Esta herramienta proporciona métricas detalladas sobre el porcentaje de código cubierto por las pruebas

unitarias. El informe ayuda a evaluar la efectividad de las pruebas y la calidad general del código, asegurando que las áreas críticas de la aplicación estén bien probadas.

### **Aplicación de Checkstyle**

Se ha aplicado **Checkstyle** a lo largo del proyecto para hacer cumplir las mejores prácticas de codificación y garantizar que el código mantenga un estilo consistente. Esta herramienta ayuda a mejorar la legibilidad y mantenibilidad del código, facilitando la colaboración entre desarrolladores y asegurando que el proyecto siga un estándar común.

### **Evaluación de SOLID y Patrones de Diseño**

El código ha sido evaluado para garantizar el cumplimiento de los principios **SOLID**, los cuales se enfocan en mejorar el diseño, la mantenibilidad y la escalabilidad del software. Además, se han implementado **patrones de diseño** donde es aplicable, para resolver problemas comunes de diseño de manera eficiente. Esto ayuda a hacer el código más extensible y robusto. También se han identificado áreas de mejora y se han propuesto refactorizaciones para optimizar la estructura del código en términos de rendimiento y facilidad de mantenimiento.

## **Microservicios Implementados**

### **Microservicio de Clientes**

El **Microservicio de Clientes** se encarga de gestionar las operaciones relacionadas con los clientes, incluyendo:

- **Crear** nuevos clientes
- **Listar** todos los clientes
- **Obtener detalles** de un cliente a través de su ID único
- **Actualizar** la información de un cliente
- **Eliminar** un cliente

Este servicio asegura que los datos de los clientes sean correctamente persistidos en la base de datos y maneja de manera eficiente todas las interacciones relacionadas con la gestión de clientes.

## Microservicio de Cuentas

El **Microservicio de Cuentas** gestiona las cuentas bancarias asociadas a los clientes. Este servicio maneja:

- **Crear** nuevas cuentas
- **Depositar** dinero en las cuentas
- **Realizar retiros** de las cuentas
- **Eliminar** cuentas
- **Listar** todas las cuentas
- **Obtener detalles** de una cuenta mediante su ID

El servicio aplica las siguientes reglas de negocio para la gestión de cuentas:

1. **Validaciones de Cliente:**
  - Cada cliente debe tener un ID único.
  - No se permite eliminar a un cliente si tiene cuentas activas.
2. **Validaciones de Cuentas Bancarias:**
  - El saldo inicial de una cuenta debe ser mayor que 0.
  - No se pueden hacer retiros que dejen el saldo en negativo en las cuentas de ahorro.
  - Las cuentas corrientes pueden tener un descubierto de hasta -500.

## Tecnologías Utilizadas

### Java 17:

Se utiliza **Java 17** para la implementación de los microservicios. Proporciona soporte a largo plazo y nuevas características del lenguaje que mejoran el rendimiento y la mantenibilidad de la aplicación.

### Spring Boot:

Se emplea **Spring Boot** para la creación y gestión de los microservicios. Simplifica el desarrollo de aplicaciones listas para producción con una configuración mínima y setup reducido.

### **JPA (Java Persistence API):**

**JPA** se utiliza para gestionar la persistencia de datos en bases de datos relacionales. Proporciona una forma simple y consistente de interactuar con la base de datos, asegurando una operación eficiente en las consultas y transacciones.

### **MySQL:**

**MySQL** es la base de datos relacional utilizada en el proyecto. Es ampliamente utilizada y ofrece un rendimiento confiable para manejar grandes volúmenes de datos. Soporta transacciones ACID, lo cual es crucial para mantener la integridad de los datos.

### **Lombok:**

**Lombok** se ha integrado en el proyecto para reducir el código repetitivo. Genera automáticamente métodos comunes como `getters`, `setters`, `toString()`, `hashCode()`, y `equals()` mediante anotaciones, lo que mejora la claridad del código y reduce la redundancia.

### **JUnit:**

Se utiliza **JUnit** para las pruebas unitarias. Permite la creación de casos de prueba para verificar que los microservicios funcionen como se espera y garantizar que los cambios futuros no rompan funcionalidades existentes.

### **Mockito:**

**Mockito** se usa para crear mocks y stubs que simulan el comportamiento de dependencias externas. Esto asegura que las pruebas unitarias estén aisladas de servicios externos como bases de datos y APIs de terceros, resultando en pruebas más confiables y mantenibles.

### **JaCoCo:**

**JaCoCo** está integrado para medir la cobertura de código. Proporciona información detallada sobre la efectividad de las pruebas unitarias, asegurando que las áreas críticas del código estén bien probadas.

### **Checkstyle:**

**Checkstyle** se aplica para hacer cumplir las convenciones de codificación y garantizar que el código sea consistente y legible. Ayuda a identificar posibles problemas como formateo incorrecto, convenciones de nombres y violaciones de la estructura del código.

### **Maven:**

**Maven** se utiliza como el gestor de dependencias y construcción del proyecto. Ayuda a gestionar bibliotecas y dependencias, automatizando el proceso de construcción y asegurando que el proyecto esté estructurado de manera estándar.

## **Mejora Continua**

Este proyecto sigue un enfoque de **mejora continua**, lo que significa que las implementaciones son revisadas y refactorizadas regularmente para mejorar la calidad del código. Esto incluye mejorar la cobertura de pruebas, optimizar el diseño mediante la adherencia a los principios **SOLID** y refactorizar el código donde sea necesario para optimizar la estructura del código y el rendimiento.

# Funcionalidades Obligatorias

## CustomerMS

### Junit y Mockito:

```
[INFO] Tests run: 15, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.761 s -- in com.corebankingsystem.CustomerMs.CustomerMsApplicationTests
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 15, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] --- jacoco:0.8.8:report (report) @ CustomerMs ---
[INFO] Loading execution data file C:\TGP - ntt data\IV proyecto\NTT-DATA-PROYECTO-IV\CustomerMs\target\jacoco.exec
[INFO] Analyzed bundle 'CustomerMs' with 12 classes
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 13.785 s
[INFO] Finished at: 2024-11-27T16:50:22-05:00
[INFO] -----
Process finished with exit code 0
```

Se realizó la implementación de pruebas unitarias de los métodos y diferentes validaciones según la lógica de los métodos.

- **Crear nuevos clientes**

```
@Test
public void testValidateAndCreateCustomer_Success() {
    // Usando el cliente con datos válidos
    Customer customer = customersArray[0];
    customer.setFirstName("Marcos");
    customer.setLastName("Martinez");
    customer.setDni("78580311");
    customer.setEmail("jmartinez@gmail.com");

    // Simulamos que el DNI no existe en la base de datos
    when(customerRepository.findByDni("78580311")).thenReturn(Optional.empty());
    when(customerRepository.save(customer)).thenReturn(customer);

    ResponseEntity<Object> response = customerServiceImpl.createCustomer(customer);

    // Validamos que la respuesta tenga un código 201 (Created) y el cliente creado
    assertEquals("expected: 201", response.getStatusCodeValue());
    assertEquals(customer, response.getBody());
}
```

- **Listar todos los clientes**

```
@Test
public void testGetCustomers() {
    when(customerRepository.findAll()).thenReturn(Arrays.asList(customersArray));

    List<Customer> customers = customerServiceImpl.getCustomers();

    assertNotNull(customers);
    assertEquals("expected: 3", customers.size());
    assertEquals("expected: Marcos", customers.get(0).getFirstName());
}
```



- **Obtener detalles** de un cliente a través de su ID único

```
@Test
public void testGetCustomerById_Found() {
    when(customerRepository.findById(1L)).thenReturn(Optional.of(customersArray[0]));

    Optional<Customer> customer = customerServiceImpl.getCustomerById(1L);

    assertTrue(customer.isPresent());
    assertEquals( expected: "Marcos", customer.get().getFirstName());
}
```

- **Actualizar** la información de un cliente

```
@Test
public void testUpdateCustomer_Success() {
    Customer updatedCustomer = new Customer( id: 1L, firstName: "Marcos", lastName: "Martinez", dni: "78580311", email: "marcos_updated@gmail.com");

    when(customerRepository.findById(1L)).thenReturn(Optional.of(customersArray[0]));
    when(customerRepository.save(any(Customer.class))).thenReturn(updatedCustomer);

    ResponseEntity<Customer> response = customerServiceImpl.updateCustomer( id: 1L, updatedCustomer);

    assertEquals( expected: 200, response.getStatusCodeValue());
    assertEquals( expected: "marcos_updated@gmail.com", response.getBody().getEmail());
}
```

- **Eliminar** un cliente

```
@Test
public void testUpdateCustomer_NotFound() {
    Customer updatedCustomer = new Customer( id: 5L, firstName: "Marcos", lastName: "Martinez", dni: "78580311", email: "marcos_updated@gmail.com");

    when(customerRepository.findById(5L)).thenReturn( Optional.empty());
    ResponseEntity<Customer> response = customerServiceImpl.updateCustomer( id: 5L, updatedCustomer);
    assertEquals( expected: 404, response.getStatusCodeValue());
}

// Test de la eliminacion de un Customer que si existe
@Test
public void testDeleteCustomer_ExistingCustomer() {
    Long customerId = 1L;
    Customer customer = customersArray[0];
    when(customerRepository.findById(customerId)).thenReturn(Optional.of(customer));
    customerServiceImpl.deleteCustomer(customerId);
    //verify(customerRepository, times(1)).deleteById(customerId);
}

// Test del caso de un Customer que no existe
@Test
public void testDeleteCustomer_CustomerNotFound() {
    Long customerId = 99L;

    when(customerRepository.findById(customerId)).thenReturn( Optional.empty());
    customerServiceImpl.deleteCustomer(customerId);
    verify(customerRepository, times( wantedNumberOfInvocations: 0)).deleteById(customerId);
}
```

# Cálculo de cobertura de código (jacoco)

## Cobertura del proyecto CustomerMS

### CustomerMs

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.example.demo.model	<div></div>	0 %	<div></div>	0 %	50	50	84	84	34	34	2	2
com.example.demo.api	<div></div>	0 %	<div></div>	0 %	30	30	57	57	22	22	6	6
com.corebankingsystem.CustomerMs.controller	<div></div>	7 %	<div></div>	0 %	9	10	19	21	6	7	0	1
com.corebankingsystem.CustomerMs	<div></div>	37 %	<div></div>	n/a	1	2	2	3	1	2	0	1
com.corebankingsystem.CustomerMs.service.impl	<div></div>	100 %	<div></div>	95 %	1	19	0	35	0	7	0	1
com.corebankingsystem.CustomerMs.model	<div></div>	100 %	<div></div>	n/a	0	2	0	9	0	2	0	1
Total	713 of 920	22 %	55 of 78	29 %	91	113	162	209	63	74	8	12

## Cobertura de CostumerServiceImpl donde se encuentra la lógica del negocio

### CustomerServiceImpl

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
createCustomer(Customer)	<div></div>	100 %	<div></div>	100 %	0	9	0	20	0	1
updateCustomer(Long, Customer)	<div></div>	100 %	<div></div>	100 %	0	2	0	6	0	1
deleteCustomer(Long)	<div></div>	100 %	<div></div>	100 %	0	2	0	5	0	1
isNullOrEmpty(String)	<div></div>	100 %	<div></div>	75 %	1	3	0	1	0	1
getCustomerById(Long)	<div></div>	100 %	<div></div>	n/a	0	1	0	1	0	1
getCustomers()	<div></div>	100 %	<div></div>	n/a	0	1	0	1	0	1
CustomerServiceImpl()	<div></div>	100 %	<div></div>	n/a	0	1	0	1	0	1
Total	0 of 175	100 %	1 of 24	95 %	1	19	0	35	0	7

## Revisión detallada de la cobertura

CustomerMs > com.corebankingsystem.CustomerMs.service.impl > CustomerServiceImpl.java

### CustomerServiceImpl.java

```
1. package com.corebankingsystem.CustomerMs.service.impl;
2.
3. import com.corebankingsystem.CustomerMs.model.Customer;
4. import com.corebankingsystem.CustomerMs.repository.CustomerRepository;
5. import com.corebankingsystem.CustomerMs.service.CustomerService;
6. import org.springframework.beans.factory.annotation.Autowired;
7. import org.springframework.http.ResponseEntity;
8. import org.springframework.stereotype.Service;
9.
10. import java.util.HashMap;
11. import java.util.List;
12. import java.util.Map;
13. import java.util.Optional;
14.
15.
16. @Service
17. public class CustomerServiceImpl implements CustomerService {
18.     @Autowired
19.     private CustomerRepository customerRepository;
20.
21.     @Override
22.     public ResponseEntity<Object> createCustomer(Customer customer) {
23.         Map<String, String> errors = new HashMap<>();
24.         if (isEmpty(customer.getFirstName())) { errors.put("firstName", "El nombre es obligatorio"); }
25.         if (isEmpty(customer.getLastName())) { errors.put("lastName", "El apellido es obligatorio"); }
26.         if (isEmpty(customer.getDni())) {
27.             errors.put("dni", "El DNI es obligatorio");
28.         } else if (!customer.getDni().matches("\\d{8}")) {
29.             errors.put("dni", "El DNI debe tener 8 dígitos");
30.         }
31.         if (isEmpty(customer.getEmail())) {
32.             errors.put("email", "El email es obligatorio");
33.         } else if (!customer.getEmail().matches("[A-Za-z0-9+_.-]+@(.+)$")) {
34.             errors.put("email", "Formato de correo electrónico inválido");
35.         }
36.         if (!errors.isEmpty()) {
37.             return ResponseEntity.badRequest().body(errors);
38.         }
39.         Optional<Customer> dni_already_exists = customerRepository.findByDni(customer.getDni());
40.         if (dni_already_exists.isPresent()) {
41.             Map<String, String> response = new HashMap<>();
42.             response.put("message", "El DNI ya existe");
43.             return ResponseEntity.status(409).body(response);
44.         }
45.         Customer customerCreated = customerRepository.save(customer);
46.         return ResponseEntity.status(201).body(customerCreated);
47.     }
48.
49.     @Override
50.     public List<Customer> getCustomers() {
51.         return customerRepository.findAll();
52.     }
53.
54.     @Override
55.     public Optional<Customer> getCustomerById(Long id) {
56.         return customerRepository.findById(id);
57.     }
58.
59.     @Override
60.     public ResponseEntity<Customer> updateCustomer(Long id, Customer customer) {
61.         Optional<Customer> existingCustomer = getCustomerById(id);
62.         if (existingCustomer.isPresent()) {
63.             customer.setId(id);
64.             Customer updatedCustomer = customerRepository.save(customer);
65.             return ResponseEntity.status(200).body(updatedCustomer);
66.         }
67.     }
68. }
```

## Revisión del COVERAGE en IntelliJ IDEA

```

public class CustomerServiceImpl implements CustomerService {
    private CustomerRepository customerRepository;

    @Override
    public ResponseEntity<Object> createCustomer(customer customer) {
        Map<String, String> errors = new HashMap<>();
        if (isNullOrEmpty(customer.getFirstname())) { errors.put("firstName", "El nombre es obligatorio"); }
        if (isNullOrEmpty(customer.getLastname())) { errors.put("lastName", "El apellido es obligatorio"); }
        if (isNullOrEmpty(customer.getDni())) { errors.put("dni", "El DNI es obligatorio"); }
        } else if (!customer.getDni().matches( regex: "\\d{8}" )) { errors.put("dni", "El DNI debe tener 8 digitos"); }
        }
        if (isNullOrEmpty(customer.getEmail())) { errors.put("email", "El email es obligatorio"); }
        } else if (!customer.getEmail().matches( regex: "[A-Za-z0-9+_-]+@[.+]+$" )) { errors.put("email", "Formato de correo electrónico inválido"); }
        }
        if (errors.isEmpty()) { return ResponseEntity.badRequest().body(errors); }
        }

        Optional<Customer> dni_already_exists = customerRepository.findByDni(customer.getDni());
        if (dni_already_exists.isPresent()) { Map<String, String> response = new HashMap<>(); response.put("message", "El DNI ya existe"); return ResponseEntity.status(409).body(response); }
        }

        Customer customerCreated = customerRepository.save(customer);
        return ResponseEntity.status(201).body(customerCreated);
    }

    @Override
    public List<Customer> getCustomers() { return customerRepository.findAll(); }
}

```

# Checkstyle

## Implementacion del archivo CheckStyle.xml

```

1 <?xml version="1.0" ?>
2 <!DOCTYPE module PUBLIC
3     "-//Checkstyle//DTD Checkstyle Configuration 1.2//EN"
4     "https://checkstyle.org/dtds/configuration_1_2.dtd">
5
6     <module name="Checker">
7
8         <module name="TreeWalker">
9
10             <module name="PackageName">
11                 <property name="format"
12                     value="com.corebankingsystem.CustomerMs"/>
13             </module>
14
15             <module name="MethodLength">
16                 <property name="max" value="30"/>
17             </module>
18
19             <module name="Indentation">
20                 <property name="tabWidth" value="4"/>
21             </module>
22
23             <module name="OneTopLevelClass"/>
24
25             <module name="UnusedImports"/>
26
27             <module name="AvoidInlineConditionals"/>
28
29             <module name="BooleanExpressionComplexity">
30                 <property name="max" value="3"/>
31             </module>
32
33             <module name="ClassDataAbstractionCoupling">
34                 <property name="max" value="3"/>
35             </module>
36
37             <module name="CyclomaticComplexity">
38                 <property name="max" value="10"/>
39             </module>
40         </module>
41     </module>
42 </>

```

## Revisión del del proyecto con CheckStyle

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-Dmaven.multiModuleProjectDirectory=C:\T6P - ntt data\IV proyecto\NTT-DATA-PROYECTO-IV\CustomerMs"
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.corebankingsystem:CustomerMs >-----
[INFO] Building CustomerMs 0.0.1-SNAPSHOT
[INFO]    from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- checkstyle:3.6.0:check (default-cli) @ CustomerMs ---
[INFO] You have 0 Checkstyle violations.
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  1.677 s
[INFO] Finished at: 2024-11-27T16:59:19-05:00
[INFO] -----

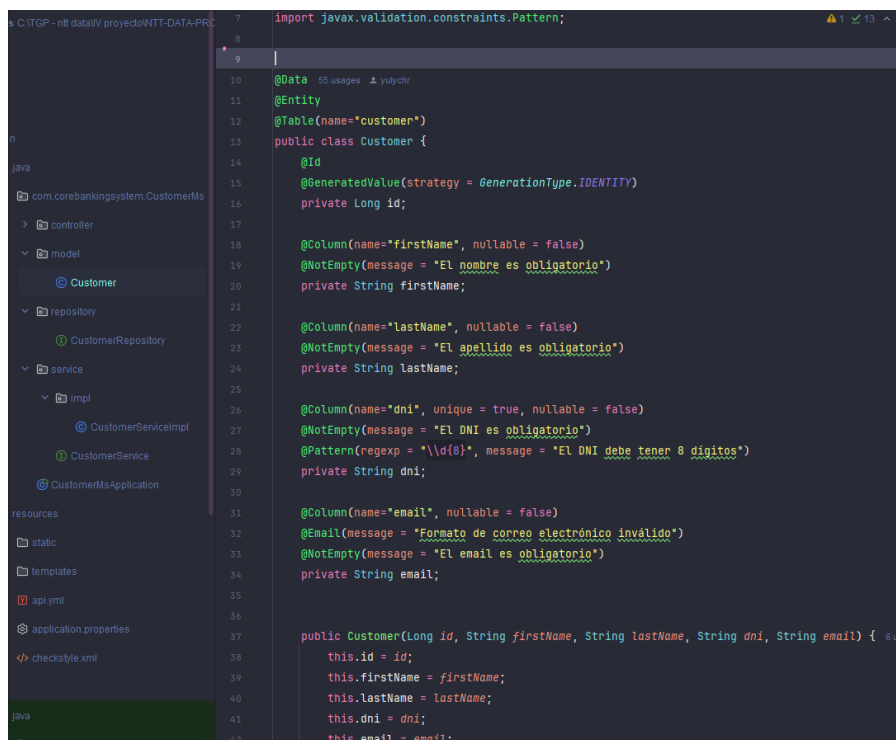
Process finished with exit code 0
```

## SOLID y patrones de diseño

En la clase **Customer**:

### Principios SOLID

- SRP: Cumple bien con el principio, ya que la clase está enfocada en representar a un cliente.
- OCP: Cumple, aunque siempre puedes refactorizar para extender la clase sin modificarla.

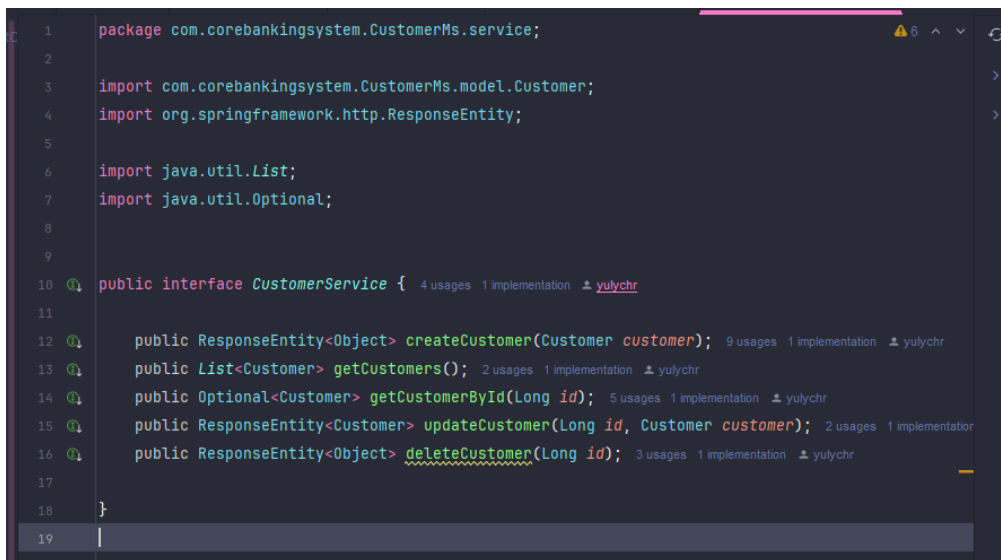


```
7 import javax.validation.constraints.Pattern;
8
9
10 @Data
11 @Entity
12 @Table(name="customer")
13 public class Customer {
14     @Id
15     @GeneratedValue(strategy = GenerationType.IDENTITY)
16     private Long id;
17
18     @Column(name="firstName", nullable = false)
19     @NotEmpty(message = "El nombre es obligatorio")
20     private String firstName;
21
22     @Column(name="lastName", nullable = false)
23     @NotEmpty(message = "El apellido es obligatorio")
24     private String lastName;
25
26     @Column(name="dni", unique = true, nullable = false)
27     @NotEmpty(message = "El DNI es obligatorio")
28     @Pattern(regexp = "\\d{8}", message = "El DNI debe tener 8 dígitos")
29     private String dni;
30
31     @Column(name="email", nullable = false)
32     @Email(message = "Formato de correo electrónico inválido")
33     @NotEmpty(message = "El email es obligatorio")
34     private String email;
35
36
37     public Customer(Long id, String firstName, String lastName, String dni, String email) {
38         this.id = id;
39         this.firstName = firstName;
40         this.lastName = lastName;
41         this.dni = dni;
42         this.email = email;
43     }
44 }
```

En la interface **CustomerService**:

### Principios SOLID

- SRP: La interfaz cumple con este principio ya que tiene una única responsabilidad de definir operaciones relacionadas con el servicio de clientes.
- OCP: Cumple con el principio, ya que puedes extender la interfaz sin modificarla.
- DIP: Cumple con este principio, ya que se está utilizando una abstracción (interfaz) en lugar de depender de implementaciones concretas.
- 



```
1 package com.corebankingsystem.CustomerMs.service;
2
3 import com.corebankingsystem.CustomerMs.model.Customer;
4 import org.springframework.http.ResponseEntity;
5
6 import java.util.List;
7 import java.util.Optional;
8
9
10 public interface CustomerService { 4 usages 1 implementation yulychr
11
12     public ResponseEntity<Object> createCustomer(Customer customer); 9 usages 1 implementation yulychr
13     public List<Customer> getCustomers(); 2 usages 1 implementation yulychr
14     public Optional<Customer> getCustomerById(Long id); 5 usages 1 implementation yulychr
15     public ResponseEntity<Customer> updateCustomer(Long id, Customer customer); 2 usages 1 implementation
16     public ResponseEntity<Object> deleteCustomer(Long id); 3 usages 1 implementation yulychr
17
18 }
19
```

En la clase **CustomerServiceImpl**:

### Principios SOLID

- OCP: Cumple, pero se puede hacer más extensible con más composición en lugar de herencia.
- ISP: Cumple, pero puede beneficiarse de dividir la interfaz en más pequeñas si la aplicación crece.
- DIP: Cumple, ya que la inyección de dependencias de Spring desacopla las clases.

Patrones de diseño: No cumple con ninguno, Aunque algunos patrones podrían aplicarse para mejorar el diseño (como Factory Method o Strategy), en su estado actual no se están utilizando explícitamente

```

16  @Service 2 usages yulychr
17  public class CustomerServiceImpl implements CustomerService{
18      @Autowired 7 usages
19      private CustomerRepository customerRepository;
20
21      @Override 9 usages yulychr
22      public ResponseEntity<Object> createCustomer(Customer customer) {
23          Map<String, String> errors = new HashMap<>();
24          if (isEmpty(customer.getFirstName())){ errors.put("firstName", "El nombre es obligatorio"); }
25          if (isEmpty(customer.getLastName())) { errors.put("lastName", "El apellido es obligatorio"); }
26          if (isEmpty(customer.getDni())) {
27              errors.put("dni", "El DNI es obligatorio");
28          } else if (!customer.getDni().matches( regex: "\\d{8}")) {
29              errors.put("dni", "El DNI debe tener 8 dígitos");
30          }
31          if (isEmpty(customer.getEmail())) {
32              errors.put("email", "El email es obligatorio");
33          } else if (!customer.getEmail().matches( regex: "[A-Za-z0-9+_.-]+@(.+)$")) {
34              errors.put("email", "Formato de correo electrónico inválido");
35          }

```

En la clase **CustomerController**:

### Principios SOLID

- SRP (Responsabilidad Única): El controlador sigue este principio, pero la lógica de negocio en `deleteCustomer()` podría moverse a un servicio.
- OCP (Abierto/Cerrado): Cumple, ya que puedes extender la funcionalidad sin modificar el controlador.
- LSP (Sustitución de Liskov): Cumple, ya que las subclases de `CustomerService` pueden sustituir la implementación sin problemas.
- DIP (Inversión de Dependencias): Cumple con el principio de inversión de dependencias, ya que el controlador depende de la interfaz y no de la implementación concreta.

```

13  import java.util.Optional;
14
15  @RestController no usages yulychr
16  @RequestMapping("/api/")
17  public class CustomerController {
18
19      private final RestTemplate restTemplate = new RestTemplate(); 1 usage
20
21      @Autowired 5 usages
22      private CustomerService customerService;
23
24      //Post - Create a new customer
25      @PostMapping("/clientes") no usages yulychr
26      public ResponseEntity<Object> createCustomer(@Valid @RequestBody Customer customer) {
27          return customerService.createCustomer(customer);
28      }
29
30      //Get - Retrieves a list of all customers in the system.
31      @GetMapping("/clientes") no usages yulychr

```

# AccountMS

## Junit y Mockito:

```
[INFO] Tests run: 13, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 6.243 s -- in com.corebankingsystem.AccountMs.AccountMsApplicationTests
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 13, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jacoco:0.8.8:report (report) @ AccountMs ---
[INFO] Loading execution data file C:\TGP - ntt data\IV proyecto\NTT-DATA-PROYECTO-IV\AccountMs\target\jacoco.exec
[INFO] Analyzed bundle 'AccountMs' with 10 classes
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 13.051 s
[INFO] Finished at: 2024-11-27T17:24:20-05:00
[INFO] -----
Process finished with exit code 0
```

Se realizó la implementación de pruebas unitarias de los métodos y diferentes validaciones según la lógica de los métodos.

- **Crear** nuevas cuentas

```
//Test para Create Account
@Test
public void testCreateAccount() {
    when(accountRepository.existsByAccountNumber(anyString())).thenReturn(true);
    when(accountRepository.save(any(Account.class))).thenReturn(accountsArray[0]);
    Account result = accountServiceImpl.createAccount(balance: 1000.0, Account.TypeAccount.corriente, customerId: 1L);
    assertNotNull(result);
    assertEquals(1000.0, result.getBalance());
}
```

- **Depositar** dinero en las cuentas

```
// Test para deposit
@Test
public void testDeposit_Success() {
    when(accountRepository.findById(1L)).thenReturn(java.util.Optional.of(accountsArray[0]));
    ResponseEntity<Object> response = accountServiceImpl.deposit(accountId: 1L, amount: 500.0);
    assertEquals(200, response.getStatusCodeValue());
    Account updatedAccount = (Account) response.getBody();
    assertNotNull(updatedAccount);
    assertEquals(1500.0, updatedAccount.getBalance());
    verify(accountRepository, times(wantedNumberOfInvocations: 1)).save(accountsArray[0]);
}

@Test
public void testDeposit_NegativeAmount() {
    when(accountRepository.findById(1L)).thenReturn(java.util.Optional.of(accountsArray[0]));
    ResponseEntity<Object> response = accountServiceImpl.deposit(accountId: 1L, amount: -500.0);
    assertEquals(400, response.getStatusCodeValue());
    String message = (String) response.getBody();
    assertEquals("Invalid deposit amount. Amount must be positive.", message);
    verify(accountRepository, times(wantedNumberOfInvocations: 0)).save(any(Account.class));
}

@Test
public void testDeposit_AccountNotFound() {
    when(accountRepository.findById(99L)).thenReturn(java.util.Optional.empty());
    Exception exception = assertThrows(RuntimeException.class, () -> {
        accountServiceImpl.deposit(accountId: 99L, amount: 500.0);
    });
    assertEquals("The account ID does not exist or is invalid.", exception.getMessage());
    verify(accountRepository, times(wantedNumberOfInvocations: 0)).save(any(Account.class));
}
```

- **Realizar retiros de las cuentas**

```
// Test para withdraw
@Test
public void testWithdraw_SuccessfulWithdrawal_AhorroAccount() {
    ResponseEntity<Object> response = accountServiceImpl.withdraw(1L, amount: 500.0);
    assertEquals( expected: 200, response.getStatusCodeValue());
    Account updatedAccount = (Account) response.getBody();
    assertNotNull(updatedAccount);
    assertEquals( expected: 500.0, updatedAccount.getBalance(), delta: 0.0);
    verify(accountRepository, times( wantedNumberOfInvocations: 1)).save(updatedAccount);
}

@Test
public void testWithdraw_InsufficientBalance_AhorroAccount() {
    ResponseEntity<Object> response = accountServiceImpl.withdraw(1L, amount: 2000.0);
    assertEquals( expected: 422, response.getStatusCodeValue());
    String message = (String) response.getBody();
    assertEquals( expected: "Withdrawals that result in a negative balance are not allowed for savings accounts.", message);
    verify(accountRepository, times( wantedNumberOfInvocations: 0)).save(any(Account.class));
}

@Test
public void testWithdraw_InsufficientBalance_CorrienteAccount() {
    ResponseEntity<Object> response = accountServiceImpl.withdraw(2L, amount: 2000.0);
}
```

- **Eliminar cuentas**

```
//Test for Delete
@Test
public void testDeleteAccount_Success() {
    when(accountRepository.findById(1L)).thenReturn(Optional.of(accountsArray[0]));
    ResponseEntity<Object> response = accountServiceImpl.deleteAccount(1L);
    assertEquals( expected: 200, response.getStatusCodeValue());
    assertEquals( expected: "Account successfully deleted", response.getBody());
    verify(accountRepository, times( wantedNumberOfInvocations: 1)).deleteById(1L);
}

@Test
public void testDeleteAccount_AccountNotFound() {
    when(accountRepository.findById(999L)).thenReturn(Optional.empty());
    ResponseEntity<Object> response = accountServiceImpl.deleteAccount(999L);
    assertEquals( expected: 404, response.getStatusCodeValue());
    assertEquals( expected: "Account not found", response.getBody());
    verify(accountRepository, times( wantedNumberOfInvocations: 0)).deleteById(999L);
}
}
```

- **Listar todas las cuentas**

```
//Test for get all accounts
@Test
public void testGetAccounts() {
    when(accountRepository.findAll()).thenReturn(Arrays.asList(accountsArray));
    List<Account> result = accountServiceImpl.getAccounts();
    assertNotNull(result);
    assertEquals( expected: 3, result.size());
    assertEquals( expected: "123456", result.get(0).getAccountNumber());
}
}
```

- **Obtener detalles de una cuenta mediante su ID**

```
//Test for get account by id
@Test
public void testGetAccountId() {
    when(accountRepository.findById(1L)).thenReturn(Optional.of(accountsArray[0]));
    Optional<Account> result = accountServiceImpl.getAccountId(1L);
    assertTrue(result.isPresent());
    assertEquals( expected: "123456", result.get().getAccountNumber());
}
}
```



# Cálculo de cobertura de código (jacoco)

## Cobertura del proyecto AccountMS

AccountMs												
AccountMs												
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.example.demo.model		0 %		0 %	81	81	138	138	55	55	5	5
com.example.demo.api		0 %		0 %	35	35	67	67	25	25	6	6
com.corebankingsystem.AccountMs.controller		8 %		n/a	9	10	20	22	9	10	0	1
com.corebankingsystem.AccountMs.service.impl		95 %		81 %	4	19	2	41	1	11	0	1
com.corebankingsystem.AccountMs		37 %		n/a	1	2	2	3	1	2	0	1
com.corebankingsystem.AccountMs.model.entity		100 %		50 %	2	7	0	18	0	5	0	2
Total	989 of 1,248	20 %	77 of 92	16 %	132	154	229	289	91	108	11	16

## Cobertura de AccountServiceImpl donde se encuentra la lógica del negocio

AccountMs > com.corebankingsystem.AccountMs.service.impl > AccountServiceImpl												
AccountServiceImpl												
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods		
getCustomerId(Long)		0 %		n/a	1	1	1	1	1	1		
createAccount(double, Account.TypeAccount, long)		85 %		50 %	1	2	1	5	0	1		
withdraw(Long, Double)		100 %		80 %	2	6	0	15	0	1		
deposit(Long, Double)		100 %		100 %	0	2	0	7	0	1		
deleteAccount(Long)		100 %		100 %	0	2	0	7	0	1		
generateAccountNumber()		100 %		n/a	0	1	0	3	0	1		
getAccountId(Long)		100 %		n/a	0	1	0	1	0	1		
lambda\$withdraw\$1()		100 %		n/a	0	1	0	1	0	1		
lambda\$deposit\$0()		100 %		n/a	0	1	0	1	0	1		
getAccounts()		100 %		n/a	0	1	0	1	0	1		
AccountServiceImpl()		100 %		n/a	0	1	0	1	0	1		
Total	9 of 197	95 %	3 of 16	81 %	4	19	2	41	1	11		

## Revisión detallada de la cobertura

```
13. import java.util.Random;
14.
15. @Service
16. public class AccountServiceImpl implements AccountService {
17.     @Autowired
18.     private AccountRepository accountRepository;
19.
20.     // Metodo para crear una nueva cuenta
21.     @Override
22.     public Account createAccount(double balance, Account.TypeAccount typeAccount, long customerId) {
23.         String accountNumber = generateAccountNumber();
24.         while (accountRepository.existsByAccountNumber(accountNumber)) {
25.             accountNumber = generateAccountNumber(); // Genera un nuevo número si ya existe
26.         }
27.         Account account = new Account(null, accountNumber, balance, typeAccount, customerId);
28.         return accountRepository.save(account);
29.     }
30.
31.     @Override
32.     public List<Account> getAccounts() {
33.         return accountRepository.findAll();
34.     }
35.     @Override
36.     public Optional<Account> getAccountId(long id) {
37.         return accountRepository.findById(id);
38.     }
39.
40.     @Override
41.     @Transactional
42.     public ResponseEntity<Object> deposit(Long accountId, Double amount) {
43.         Account account = accountRepository.findById(accountId).orElseThrow(() -> new RuntimeException("The account ID does not exist or is invalid."));
44.         if (amount <= 0) {
45.             String message = "Invalid deposit amount. Amount must be positive.";
46.             return ResponseEntity.status(400).body(message);
47.         }
48.         account.deposit(amount);
49.         accountRepository.save(account);
50.         return ResponseEntity.status(200).body(account);
51.     }
52.
53.     @Override
54.     @Transactional
55.     public ResponseEntity<Object> withdraw(Long accountId, Double amount) {
56.         Account account = accountRepository.findById(accountId).orElseThrow(() -> new RuntimeException("The account ID does not exist or is invalid."));
57.         if (amount <= 0) {
58.             String message = "Invalid withdraw amount. Amount must be positive.";
59.             return ResponseEntity.status(400).body(message);
60.         }
61.         if (account.getTypeAccount() == Account.TypeAccount.ahorros) {
62.             if (account.getBalance() - amount < 0) {
63.                 String message = "Withdrawals that result in a negative balance are not allowed for savings accounts.";
64.                 return ResponseEntity.status(422).body(message);
65.             }
66.         } else if (account.getTypeAccount() == Account.TypeAccount.corriente) {
67.             if (account.getBalance() - amount < -500) {
68.                 String message = "Withdrawals exceeding minus 500 below the balance are not allowed for checking account (overdrafts are permitted).";
69.                 return ResponseEntity.status(422).body(message);
70.             }
71.         }
72.         account.withdraw(amount);
73.         accountRepository.save(account);
74.         return ResponseEntity.status(200).body(account);
75.     }
76.
77.     @Override
78.     public ResponseEntity<Object> deleteAccount(Long id) {
79.         Optional<Account> account = getAccountId(id);
80.         if (account.isPresent()) {
```

## Revisión del COVERAGE en IntelliJ IDEA

Element	Class, %	Method, %	Line, %	Branch, %
com.corebankingsystem.AccountMs	60% (3/5)	59% (13/22)	70% (52/74)	75% (15/20)
controller	0% (0/1)	0% (0/7)	0% (0/19)	100% (0/0)
model entity	100% (2/2)	100% (6/6)	100% (14/14)	50% (2/4)
repository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
service	100% (1/1)	87% (7/8)	95% (38/40)	81% (13/16)
impl	100% (1/1)	87% (7/8)	95% (38/40)	81% (13/16)
AccountServiceImpl	100% (1/1)	87% (7/8)	95% (38/40)	81% (13/16)
AccountService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
AccountMsApplication	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)

## Checkstyle

### Implementacion del archivo CheckStyle.xml

```
<?xml version="1.0" ?>
<!-- disable DTD validation in IntelliJ IDEA -->
<!DOCTYPE module PUBLIC
    "-//Checkstyle//DTD Checkstyle Configuration 1.2//EN"
    "https://checkstyle.org/dtds/configuration_1_2.dtd">

<module name="Checker">
    <module name="TreeWalker">
        <module name="PackageName">
            <property name="format"
                value="com.corebankingsystem.AccountMs"/>
        </module>

        <module name="MethodLength">
            <property name="max" value="30"/>
        </module>

        <module name="Indentation">
            <property name="tabWidth" value="4"/>
        </module>

        <module name="OneTopLevelClass"/>

        <module name="UnusedImports"/>

        <module name="AvoidInlineConditionals"/>

        <module name="BooleanExpressionComplexity">
            <property name="max" value="3"/>
        </module>

        <module name="ClassDataAbstractionCoupling">
            <property name="max" value="3"/>
        </module>
    </module>
</module>
```

## Revisión del del proyecto con CheckStyle

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-Dmaven.multiModuleProjectDirectory=C:\TGP - ntt data\IV proyecto\NTT-DATA-PROYECTO-IV\AccountMs"
[INFO] Scanning for projects...
[INFO] -----< com.corebankingsystem:AccountMs >-----
[INFO] Building AccountMs 0.0.1-SNAPSHOT
[INFO]   from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- checkstyle:3.6.0:check (default-cli) @ AccountMs ---
[INFO] You have 0 Checkstyle violations.
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.564 s
[INFO] Finished at: 2024-11-27T17:32:22-05:00
[INFO] -----
Process finished with exit code 0
```

## SOLID y patrones de diseño

En la clase **Account**:

### Principios SOLID

- SRP (Responsabilidad Única): Cumple, ya que Account se enfoca en representar la entidad de la cuenta bancaria y sus operaciones básicas.
- OCP (Abierto/Cerrado): Cumple, ya que se puede extender para agregar nuevos tipos de cuentas o nuevas operaciones sin modificar la clase base.
- DIP (Inversión de Dependencias): No se aplica directamente en la clase Account, pero se puede aplicar si hay servicios que interactúan con cuentas.

```
11 public class Account {
26     private TypeAccount typeAccount;
27
28     @Column(name="customer_id", nullable = false)
29     @NotEmpty(message = "CustomerID type is required")
30     private long customerId;
31
32     public Account(Long id, String accountNumber, double balance, TypeAccount typeAccount, long custo
33         this.id = id;
34         this.accountNumber = accountNumber;
35         this.balance = balance;
36         this.typeAccount = typeAccount;
37         this.customerId = customerId;
38     }
39     public Account(){ no usages 1 yulychr
40
41     }
42
43     public enum TypeAccount { 10 usages 1 yulychr
44         ahorros, 3 usages
45         corriente 3 usages
46     }
47 }
```

En la interface **AccountService**:

### Principios SOLID

- SRP: La interfaz cumple con este principio ya que tiene una única responsabilidad de definir operaciones relacionadas con el servicio de cuentas.
- OCP: Cumple con el principio, ya que puedes extender la interfaz sin modificarla.
- DIP: Cumple con este principio, ya que se está utilizando una abstracción (interfaz) en lugar de depender de implementaciones concretas.

```
1 package com.corebankingsystem.AccountMs.service;
2
3 import com.corebankingsystem.AccountMs.model.entity.Account;
4 import org.springframework.http.ResponseEntity;
5
6 import java.util.List;
7 import java.util.Optional;
8
9 public interface AccountService { 4 usages 1 implementation yulychr
10
11     public List<Account> getAccounts(); 2 usages 1 implementation yulychr
12     public Optional<Account> getAccountId(Long id); 3 usages 1 implementation yulychr
13     public ResponseEntity<Object> deposit(Long accountId, Double amount); 4 usages 1 implementation yulychr
14     public ResponseEntity<Object> withdraw(Long accountId, Double amount); 6 usages 1 implementation yulychr
15     public ResponseEntity<Object> deleteAccount(Long id); 3 usages 1 implementation yulychr
16     public Account createAccount(double balance, Account.TypeAccount typeAccount, long customerId); 2 usages 1 implementation yulychr
17     public Optional<List<Account>> getCustomerId(Long id); 1 usage 1 implementation yulychr
18 }
```

En la clase **AccountServiceImpl**:

### Principios SOLID

- SRP: Cumple, AccountServiceImpl maneja exclusivamente la lógica de cuentas (crear, depositar, retirar, eliminar), con una única responsabilidad.
- OCP: Cumple, a clase es fácil de extender (se pueden agregar nuevos comportamientos sin modificar la clase actual).
- LPS: Cumple, la clase es extensible sin afectar el comportamiento actual, permitiendo herencia o implementación futura sin problemas.
- ISP: Cumple, la interfaz AccountService está centrada solo en métodos relevantes para cuentas, evitando métodos innecesarios.

Patrones de diseño:

- Factory Method: cumple parcialmente al tener un método para crear cuentas con una lógica interna, pero no está formalizado como un patrón.

```

@Service 2 usages 1 yulchr
public class AccountServiceImpl implements AccountService {
    @Autowired 10 usages
    private AccountRepository accountRepository;

    // Metodo para crear una nueva cuenta
    @Override 2 usages 1 yulchr
    public Account createAccount(double balance, Account.TypeAccount typeAccount, long customerId) {
        String accountNumber = generateAccountNumber();
        while (accountRepository.existsByAccountNumber(accountNumber)) {
            accountNumber = generateAccountNumber(); // Genera un nuevo número si ya existe
        }
        Account account = new Account( id: null, accountNumber, balance, typeAccount, customerId);
        return accountRepository.save(account);
    }

    @Override 2 usages 1 yulchr
    public List<Account> getAccounts() { return accountRepository.findAll(); }
    @Override 3 usages 1 yulchr
    public Optional<Account> getAccountId(Long id) { return accountRepository.findById(id); }

    @Override 4 usages 1 yulchr
    @Transactional
    public ResponseEntity<Object> deposit(Long accountId, Double amount) {
        Account account = accountRepository.findById(accountId).orElseThrow(() -> new RuntimeException("The a
        if (amount <= 0) {
            String message = "Invalid deposit amount. Amount must be positive.";
            return ResponseEntity.status(400).body(message);
        }
        account.deposit(amount);
        accountRepository.save(account);
        return ResponseEntity.status(200).body(account);
    }
}

```

En la clase **AccountController**:

## Principios SOLID

- SRP (Responsabilidad Única): El controlador tiene la responsabilidad de manejar las solicitudes HTTP relacionadas con las cuentas, como la creación, consulta, actualización y eliminación.
- OCP (Abierto/Cerrado): Cumple, la clase está abierta a la extensión (se pueden agregar más endpoints fácilmente), pero no está completamente cerrada a la modificación.
- LSP (Sustitución de Liskov): No hay herencia explícita ni reemplazo de clases, ya que el controlador es una clase concreta que maneja las solicitudes.
- ISP: Cumple, el controlador no está haciendo cosas que no están relacionadas con su tarea principal.
- DIP (Inversión de Dependencias): El controlador depende de la abstracción (AccountService) y no de una implementación concreta. Esto permite que el servicio AccountService sea intercambiable, facilitando la prueba y la extensión del código.

Patrones de Diseño:

- **Adapter:** Cumple parcialmente. El patrón Adapter se usa para convertir una interfaz en otra, pero el uso del RestTemplate para realizar solicitudes a un servicio externo (para validar el customerId) podría considerarse una forma de adaptador.

```
public class AccountController {

    private final RestTemplate restTemplate = new RestTemplate();

    @Autowired
    private AccountService accountService;

    @PostMapping ("/cuentas")
    @ResponseStatus(HttpStatus.CREATED) // Respuesta 201 cuando se crea la cuenta
    public ResponseEntity<Object> createAccount(@Valid @RequestBody Account account) {

        try {
            String url = "http://localhost:8086/api/clientes/" + account.getCustomerId() ;
            restTemplate.getForObject(url, String.class);
        } catch (HttpClientErrorException e) {
            String message = "The customer ID does not exist ";
            return ResponseEntity.status(404).body(message);
        }

        Account localAccount = accountService.createAccount(
            account.getBalance(),
            account.getTypeAccount(),
            account.getCustomerId()
        );
        return ResponseEntity.status(200).body(localAccount);
    }

    @GetMapping ("/cuentas")
    public ResponseEntity<List<Account>> getAllAccounts() {
        List<Account> accounts = accountService.getAccounts();
        return ResponseEntity.status(200).body(accounts);
    }
}
```