

**ACADEMIA XIDERAL**  
**Sistema Bancario Digital**

**2:A Explicación de proyecto final**  
**“Sistema Bancario”**

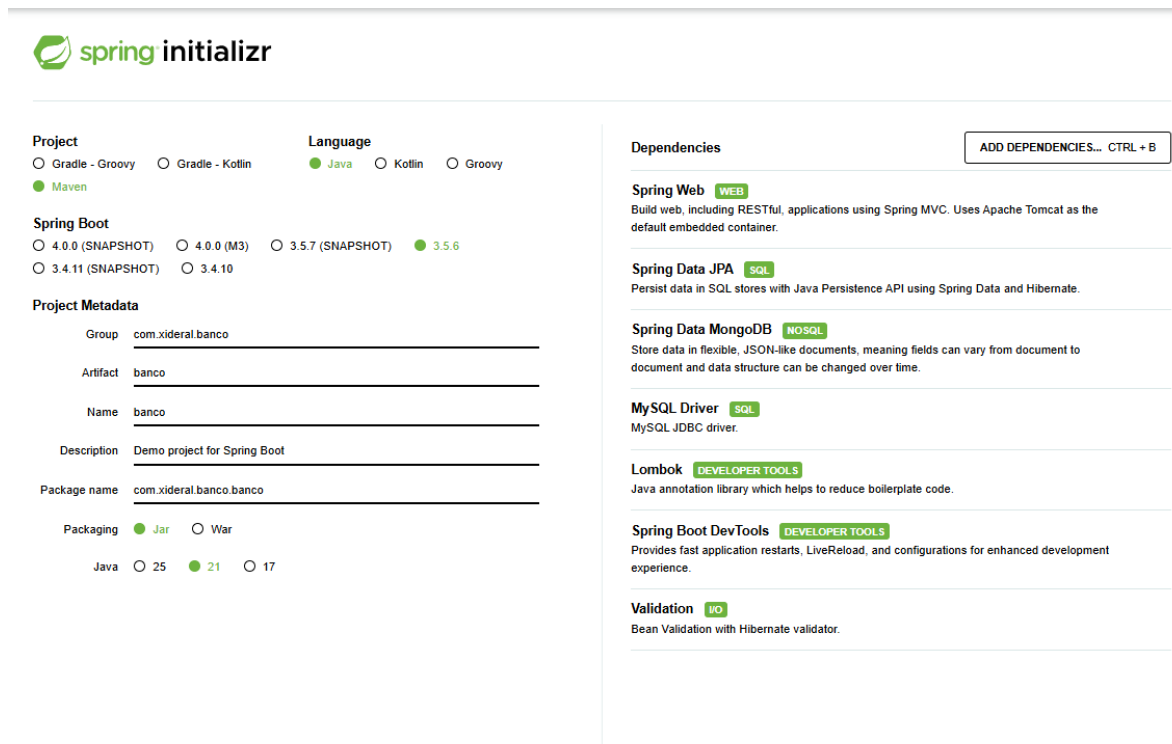
**Eduardo Rosales Alvarado**

## Objetivo

Desarrollar proyecto haciendo uso de todo lo aprendido a lo largo de la academia basándose en una arquitectura modular, aplicando inyección de dependencias para configurar bases de datos híbridas haciendo uso de APIs REST y demostrar el polimorfismo.

## Día 1

### 1. En el primer día se crea el proyecto desde spring initializr



The screenshot shows the Spring Initializr web application. It has a header with the 'spring initializr' logo. The main content is divided into two columns. The left column contains sections for 'Project' (with radio buttons for Gradle - Groovy, Gradle - Kotlin, Java (selected), and Maven), 'Spring Boot' (with radio buttons for various versions, with 3.5.6 selected), and 'Project Metadata' (with input fields for Group, Artifact, Name, Description, and Package name, and radio buttons for Packaging and Java version). The right column is titled 'Dependencies' and has a button 'ADD DEPENDENCIES... CTRL + B'. Below this, there are several dependency cards: 'Spring Web' (WEB), 'Spring Data JPA' (SQL), 'Spring Data MongoDB' (NO SQL), 'MySQL Driver' (SQL), 'Lombok' (DEVELOPER TOOLS), 'Spring Boot DevTools' (DEVELOPER TOOLS), and 'Validation' (JSO).

Donde seleccionamos el lenguaje, el tipo de proyecto, la versión de Spring Boot y agregamos las dependencias a nuestro proyecto.

Las dependencias que nos faltan por agregar las agregaremos a través del archivo pom.xml las cuales son las siguientes Spring Boot Starter Test, JUnit, Mockito, JaCoCo y Swagger.

2. Nos dirigimos al archivo de applications.properties que es donde vamos a realizar las configuraciones de nuestro proyecto y además configuraremos la conexión con la base de datos

Para el caso de MySQL se realiza la siguiente configuración

```
# MySQL Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/banco_db?createDatabaseIfNotExist=true&useSSL=false&serverTimezone=UTC&allowPublicKeyRetrieval=true
spring.datasource.username=root
spring.datasource.password=xideral1234
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

Para el caso de MongoDB se realiza lo siguiente

```
# MongoDB Configuration
spring.data.mongodb.uri=mongodb://admin:xideral4321@localhost:27017/banco_logs?authSource=admin
spring.data.mongodb.database=banco_logs
```

3. Se crea la estructura del proyecto siguiente:

com.xideral.banco/

```
├── customer/
│   ├── controller/
│   ├── service/
│   ├── repository/
│   └── model/
├── account/
│   ├── controller/
│   ├── service/
│   ├── repository/
│   └── model/
├── notification/
│   ├── service/
│   └── model/
├── batch/
├── events/
└── config/
```

4. Se crean entidades bases, donde lombok y JPA nos permitirán evitar escribir todos los códigos repetitivos como son los métodos constructores, getter, setter, toString, hash y equals. Todo esto a través de las anotaciones de las dependencias.

Además de que nos permitirán mapear en nuestras bases de datos que en este caso será para una MySQL.

```
public class Customer {
```

```
@Id
```

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@NotBlank(message = "Name is required")

@Column(nullable = false, length = 100)

private String name;

@NotBlank(message = "Email is required")

@Email(message = "Email should be valid")

@Column(nullable = false, unique = true, length = 100)

private String email;

@NotBlank(message = "Phone is required")

@Pattern(regexp = "^\\d{10}\$", message = "Phone must be 10 digits")

@Column(nullable = false, length = 10)

private String phone;

@Enumerated(EnumType.STRING)

@Column(nullable = false, length = 20)

private CustomerStatus status = CustomerStatus.ACTIVE;

@CreationTimestamp

@Column(name = "created\_at", nullable = false, updatable = false)

private LocalDateTime createdAt;

@UpdateTimestamp

@Column(name = "updated\_at")

private LocalDateTime updatedAt;

```

public enum CustomerStatus {
    ACTIVE,
    INACTIVE
}
}

```

## Día 2

En este día se busca realizar un CRUD a través de una API REST

1. Empezamos definiendo en nuestra capa de repository los métodos

```

@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    Optional<Customer> findByEmail(String email); 2 usages

    boolean existsByEmail(String email); 13 usages

    List<Customer> findByStatus(Customer.CustomerStatus status);

    List<Customer> findByNameContainingIgnoreCase(String name); no usages
}

```

2. Luego pasamos con la capa de servicio donde definimos los métodos de esta

```

public interface CustomerService { 5 usages 1 implementation

    Customer createCustomer(Customer customer); 6 usages 1 implementation

    Customer updateCustomer(Long id, Customer customer); 5 usages 1 implementation

    Customer getCustomerById(Long id); 11 usages 1 implementation

    List<Customer> getAllCustomers(); 4 usages 1 implementation

    List<Customer> getCustomersByStatus(Customer.CustomerStatus status); 4 usages

    void deleteCustomer(Long id); 4 usages 1 implementation

    Customer activateCustomer(Long id); 4 usages 1 implementation

    Customer deactivateCustomer(Long id); 4 usages 1 implementation

    boolean existsByEmail(String email); 1 usage 1 implementation
}

```

3. Continuamos con la implementación de estos métodos además de que hacemos uso de la inyección de dependencias.

```
@Service 1 usage
@RequiredArgsConstructor
@Slf4j
@Transactional
public class CustomerServiceImpl implements CustomerService {

    private final CustomerRepository customerRepository;
    private final ApplicationEventPublisher eventPublisher;

    @Override 6 usages
    public Customer createCustomer(Customer customer) {
        log.debug("Creating customer with email: {}", customer.getEmail());

        if (customerRepository.existsByEmail(customer.getEmail())) {
            throw new IllegalArgumentException("Email already exists: " + customer.getEmail());
        }

        customer.setStatus(Customer.CustomerStatus.ACTIVE);
        Customer savedCustomer = customerRepository.save(customer);
        log.info("Customer created successfully with id: {}", savedCustomer.getId());

        // Publicar evento
        CustomerCreatedEvent event = new CustomerCreatedEvent(
            savedCustomer.getId(),
            savedCustomer.getName(),
            savedCustomer.getEmail(),
            LocalDateTime.now()
        );
        eventPublisher.publishEvent(event);
        log.debug("CustomerCreatedEvent published for customer: {}", savedCustomer.getEmail());

        return savedCustomer;
    }
}
```

```

@Override 5 usages
public Customer updateCustomer(Long id, Customer customer) {
    log.debug("Updating customer with id: {}", id);

    Customer existingCustomer = getCustomerById(id);

    // Verificar si el email cambió y si ya existe
    if (!existingCustomer.getEmail().equals(customer.getEmail()) &&
        customerRepository.existsByEmail(customer.getEmail())) {
        throw new IllegalArgumentException("Email already exists: " + customer.getEmail());
    }

    existingCustomer.setName(customer.getName());
    existingCustomer.setEmail(customer.getEmail());
    existingCustomer.setPhone(customer.getPhone());

    Customer updatedCustomer = customerRepository.save(existingCustomer);
    log.info("Customer updated successfully with id: {}", updatedCustomer.getId());
    return updatedCustomer;
}

```

```

@Override 11 usages
@Transactional(readOnly = true)
public Customer getCustomerById(Long id) {
    log.debug("Getting customer by id: {}", id);
    return customerRepository.findById(id)
        .orElseThrow(() -> new IllegalArgumentException("Customer not found with id: "
    )
}

@Override 4 usages
@Transactional(readOnly = true)
public List<Customer> getAllCustomers() {
    log.debug("Getting all customers");
    return customerRepository.findAll();
}

@Override 4 usages
@Transactional(readOnly = true)
public List<Customer> getCustomersByStatus(Customer.CustomerStatus status) {
    log.debug("Getting customers by status: {}", status);
    return customerRepository.findByStatus(status);
}

```



```

@Override 4 usages
public void deleteCustomer(Long id) {
    log.debug("Deleting customer with id: {}", id);

    Customer customer = getCustomerById(id);
    customer.setStatus(Customer.CustomerStatus.INACTIVE);
    customerRepository.save(customer);
    log.info("Customer soft deleted (deactivated) with id: {}", id);
}

@Override 4 usages
public Customer activateCustomer(Long id) {
    log.debug("Activating customer with id: {}", id);

    Customer customer = getCustomerById(id);
    customer.setStatus(Customer.CustomerStatus.ACTIVE);
    Customer activatedCustomer = customerRepository.save(customer);
    log.info("Customer activated successfully with id: {}", id);
    return activatedCustomer;
}

```

```

@Override 4 usages
public Customer deactivateCustomer(Long id) {
    log.debug("Deactivating customer with id: {}", id);

    Customer customer = getCustomerById(id);
    customer.setStatus(Customer.CustomerStatus.INACTIVE);
    Customer deactivatedCustomer = customerRepository.save(customer);
    log.info("Customer deactivated successfully with id: {}", id);
    return deactivatedCustomer;
}

@Override 1 usage
@Transactional(readOnly = true)
public boolean existsByEmail(String email) { return customerRepository.existsByEmail(email); }
}

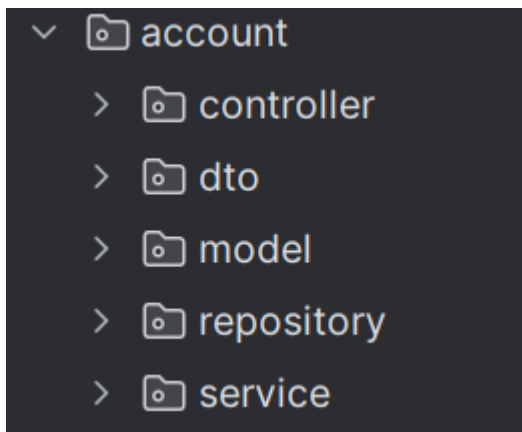
```

4. En la capa controller definimos los métodos de la API REST además de los endpoints para poder consumir la API, se sea haciendo uso de Postman o Swagger. Para este proyecto se hará uso de Swagger

Customer <small>Customer management APIs</small>			^
GET	/api/customers/{id}	Get customer by ID	▼
PUT	/api/customers/{id}	Update customer	▼
DELETE	/api/customers/{id}	Delete customer (soft delete)	▼
GET	/api/customers	Get all customers	▼
POST	/api/customers	Create a new customer	▼
PATCH	/api/customers/{id}/deactivate	Deactivate customer	▼
PATCH	/api/customers/{id}/activate	Activate customer	▼
GET	/api/customers/status/{status}	Get customers by status	▼

Día 3

1. Este día se configura la parte del proyecto de account



2. Al igual que el día anterior se define la capa de model y se hace uso de las dependencias de lombok y JPA para definir las entidades la cuales van a ser mapeadas en la base de datos que son Account y Transaction

Para el caso de Account se mapea en nuestra base de datos de la siguiente manera

id	account_number	account_type	balance	created_at	customer_id	status	updated_at
----	----------------	--------------	---------	------------	-------------	--------	------------

Mientras que Transaction lo hace de la siguiente forma

id	account_number	account_type	balance	created_at	customer_id	status	updated_at
----	----------------	--------------	---------	------------	-------------	--------	------------

En esencia es lo mismo, pero representan diferentes cosas.

3. Se definen las capas de Repository y service donde se hace inyección de dependencias y se definen los métodos y se implementan para posteriormente definir los endpoints y los métodos (API REST) en la capa de controller para poder consumirlos a través de Postman de Swagger.

Account Account management and banking operations APIs			^
GET	/api/accounts/{id}	Get account by ID	▼
PUT	/api/accounts/{id}	Update account	▼
DELETE	/api/accounts/{id}	Delete account (soft delete)	▼
GET	/api/accounts	Get all accounts	▼
POST	/api/accounts	Create a new account	▼
POST	/api/accounts/withdraw	Withdraw money from account	▼
POST	/api/accounts/transfer	Transfer money between accounts	▼
POST	/api/accounts/deposit	Deposit money to account	▼
PATCH	/api/accounts/{id}/close	Close account	▼
PATCH	/api/accounts/{id}/activate	Activate account	▼
GET	/api/accounts/type/{type}	Get accounts by type	▼
GET	/api/accounts/status/{status}	Get accounts by status	▼
GET	/api/accounts/number/{accountNumber}	Get account by account number	▼
GET	/api/accounts/customer/{customerId}	Get all accounts by customer ID	▼
GET	/api/accounts/customer/{customerId}/count	Count accounts by customer ID	▼
GET	/api/accounts/customer/{customerId}/active	Get active accounts by customer ID	▼

## Día 4

En este día se hizo la implementación de spring modulith, donde se hace la comunicación entre los módulos a través de los eventos, uno publica mientras otro se mantiene esperando a que este sea publicado

1.En la capa de servicio del proyecto tanto de Account como de Customer se publican los eventos

```

public Customer createCustomer(Customer customer) {
    log.debug("Creating customer with email: {}", customer.getEmail());

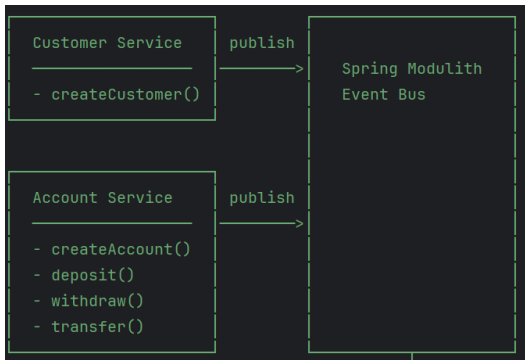
    if (customerRepository.existsByEmail(customer.getEmail())) {
        throw new IllegalArgumentException("Email already exists: " + customer.getEmail());
    }

    customer.setStatus(Customer.CustomerStatus.ACTIVE);
    Customer savedCustomer = customerRepository.save(customer);
    log.info("Customer created successfully with id: {}", savedCustomer.getId());

    // Publicar evento
    CustomerCreatedEvent event = new CustomerCreatedEvent(
        savedCustomer.getId(),
        savedCustomer.getName(),
        savedCustomer.getEmail(),
        LocalDateTime.now()
    );
    eventPublisher.publishEvent(event);
    log.debug("CustomerCreatedEvent published for customer: {}", savedCustomer.getEmail());

    return savedCustomer;
}

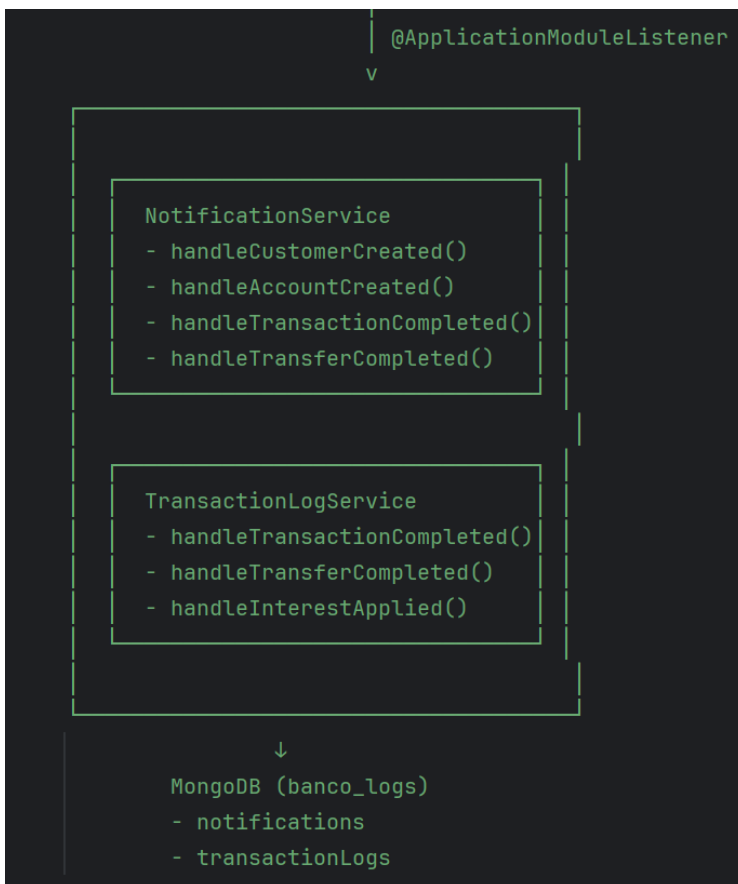
```



Estos eventos son publicados para ser escuchados a través de la anotación `@ApplicationModuleListener` y después sean persistidos en la base de datos MongoDB

```

// Event Listeners
@ApplicationModuleListener 1 usage
public void handleCustomerCreated(CustomerCreatedEvent event) {
    log.debug("Handling CustomerCreatedEvent for customer: {}", event.getEmail());
    notifyCustomerRegistered(event.getCustomerId(), event.getEmail(), event.getFullName());
}
  
```



```
_id: ObjectId('68ded6bdc46a3513bcae1ccf')
customerId: 1
customerEmail: "patrobas.garcia@example.com"
type: "CUSTOMER_REGISTERED"
channel: "EMAIL"
subject: "Bienvenido al Banco Digital"
message: "Bienvenido Patrobas, su registro ha sido exitoso. Puede comenzar a uti..."
status: "SENT"
createdAt: 2025-10-02T19:47:09.513+00:00
sentAt: 2025-10-02T19:47:09.913+00:00
transactionType: "CUSTOMER_REGISTERED"
_class: "com.xideral.banco.notification.model.Notification"
```

## Día 5

En este día se realiza un proceso batch y se actualizan las cuentas que están activas aplicándoles un interés correspondiente, para esto se realiza un trabajo que consta de 2 pasos (steps)

```
@Bean
public Job monthlyInterestJob() {
    JobBuilder jobBuilder = new JobBuilder("monthlyInterestJob", jobRepository);

    // MongoDB listener para auditoría
    if (batchJobExecutionMongoListener != null) {
        jobBuilder.listener(batchJobExecutionMongoListener);
    }

    return jobBuilder
        .start(calculateAndApplyInterestStep()) // STEP 1
        .next(publishEventsStep()) // STEP 2
        .build();
}
```

En el paso 1 se obtiene la información de las cuentas de las bases de datos, se calcula el interés y se aplica y se vuelve a almacenar en la base de datos.

```
@Bean
public Step calculateAndApplyInterestStep() {
    return new StepBuilder("calculateAndApplyInterestStep", jobRepository)
        .<Account, AccountInterestData>chunk(10, transactionManager)
        .reader(accountReader())
        .processor(interestCalculatorProcessor())
        .writer(interestApplierWriter())
        .build();
}
```

El paso 2 se encarga de publicar el evento para que después se persista en la base de datos MongoDB la información.