Programación de Aplicaciones Telemáticas

TEMA 10: ACCESO A BASE DE DATOS RELACIONALES (JDBC)

AGENDA

SESSIÓN 1

- Introducción
- Estandares
- Spring Data
- Conceptos
- Repositorios
- Relaciones
- Transacciones
- Schema de base de datos
- Testing

SESSIÓN 1

INTRODUCCIÓN: JDBC

Java Database Connectivity

- Proporciona una API completa para poder conectarnos y trabajar con cualquier SGBD relacional
- Utiliza un driver específico para cada SGBD, este sí es diferente para cada motor. PostgreSQL, MariaDB, MySQL, DB2…

UTILIDAD DE JDBC

- Abrir y cerrar conexiones contra la BD.
- Operaciones de gestión (crear y borrar tablas…)
- Operaciones CRUD (insertar, actualizar, borrar y leer)
- Transacciones

EJEMPLO JDBC LEGACY POSTGRESQL

ESTÁNDARES: ORM

Object-Relational mapping es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia

En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional

Ejemplos: Hibernate, JPA, JDO, iBatis…

ESTÁNDARES: DDD (DOMAIN-DRIVEN DESIGN)

- Diseño de la aplicación en base al dominio
- El dominio es la parte de la realidad que expresamos mediante programación orientada a objetos
- Su representación en objetos se identifica como el Modelo de la aplicación

DDD: CORE PRINCIPLES

- Focus on the core domain and domain logic
- Base complex designs on models of the domain
- Constantly collaborate with domain experts, in order to improve the application model and resolve any emerging domain-related issues

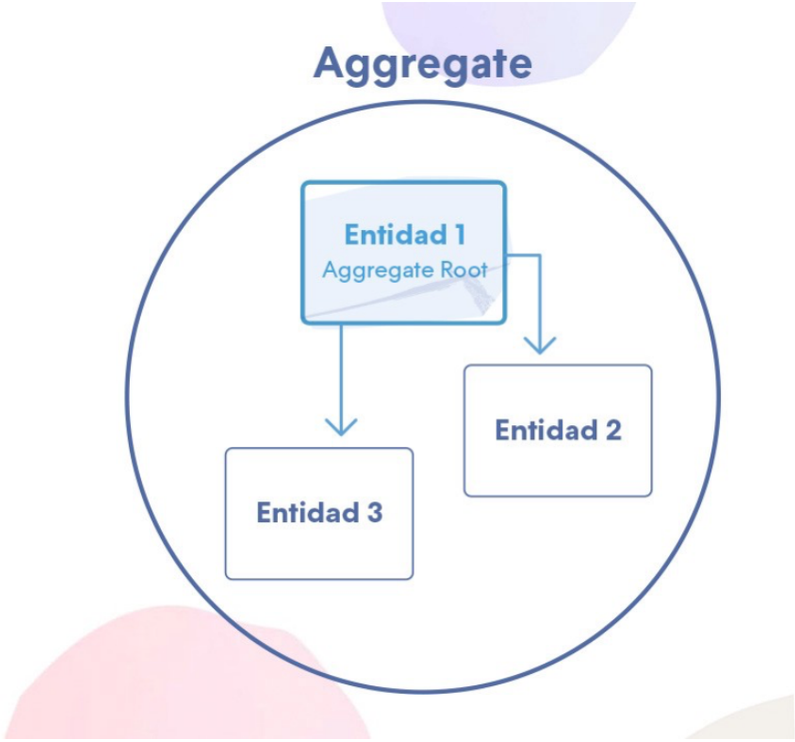
DDD: CORE PRINCIPLES

- **Entity**: Son objetos que tienen identidad propia en el sistema y donde sus atributos o propiedades no identifican quién es. Por ejemplo, un User identificado por un user_id único
- **Value Object**:no tienen identidad ninguna, solo nos interesan sus atributos, ya que complementan la descripción del dominio, pero no se identifican por sí mismos

DDD: CORE PRINCIPLES

- **Aggregate**:grupos de entidades que se relacionan entre sí donde se define la dependencia entre ellas. En dichos agregados hay que definir cuál es la Entidad padre (root) y cuál es la frontera
- **Reference**:Se utilizan en los agregados. En vez de referenciar la entidad completa, se referencia el id de la entidad

DDD: CORE PRINCIPLES



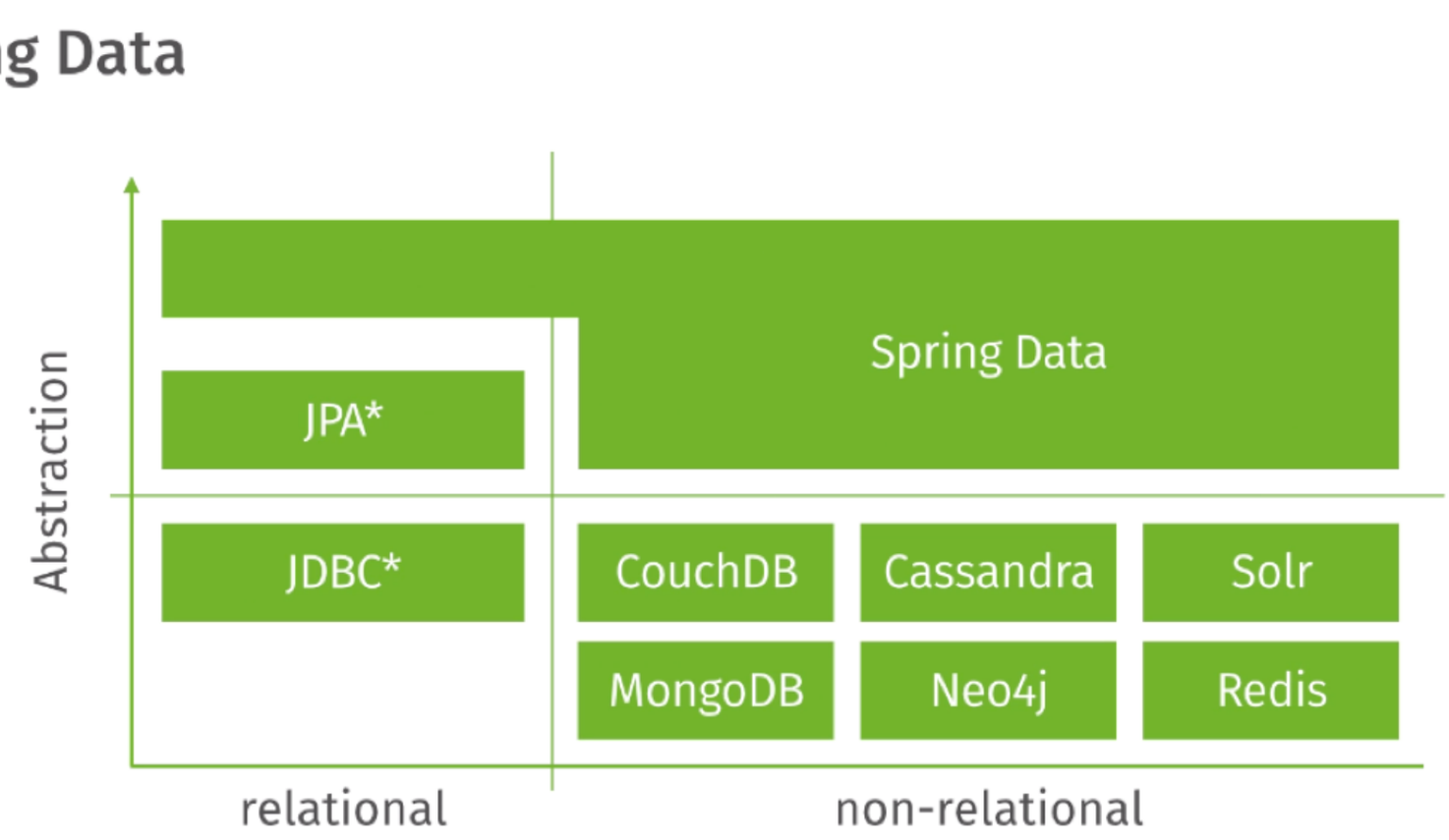
DDD: CORE PRINCIPLES

- **Service**:servicios de dominio. todos aquellos comportamientos que debemos tener en nuestra aplicación y que no pertenezcan a ninguna entidad. No tienen estado, y modifican una o varias entidades de dominio, pero que no son propias de la entidad (cargar dinero en la cuenta de un Usuario)
- **Repository**:clases que se encargan de persistir y recuperar los objetos de dominio que necesitamos que pervivan en el tiempo. Normalmente, se crea un repositorio por Entidad raíz de una Agregación

SPRING DATA

- Spring Data’s mission is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store
- It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services. This is an umbrella project which contains many subprojects that are specific to a given database. The projects are developed by working together with many of the companies and developers that are behind these exciting technologies

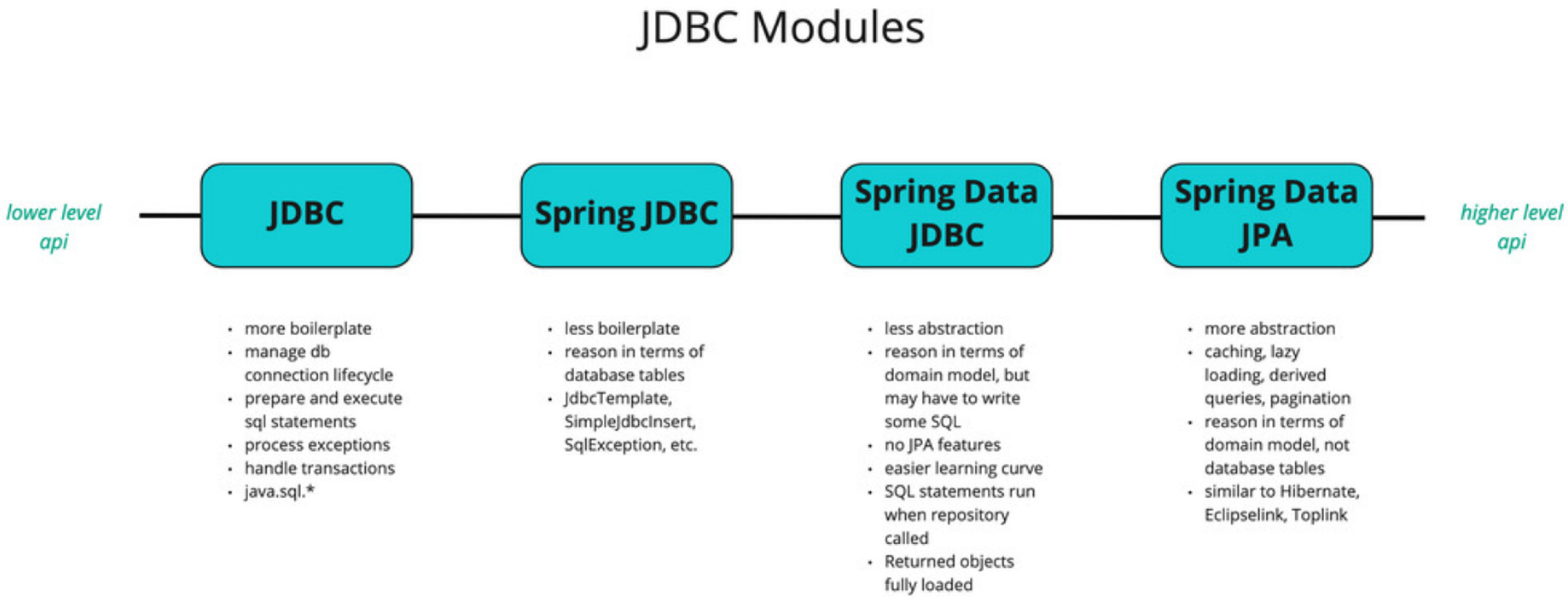
SPRING DATA: TECHNOLOGIES



SPRING DATA JDBC

- ORM simple y limitado
- Permite implementar repositorios basados en JDBC
- Inspirado en DDD
- Integración con Spring

SPRING DATA JDBC



SPRING DATA JDBC: MAVEN DEPENDENCIES

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

SPRING DATA JDBC: DATASOURCE

- Lo primero que se debe configurar para poder hacer uso se Spring Data JDBC es un objeto DataSource
- Este objeto representa la conexión con la fuente de datos (DB)
- Se debe configurar la URL, el driver de la JDBC, usuario y contraseña
- Se puede configurar mediante application.properties o programaticamente

SPRING DATA JDBC: DATASOURCE

```java
import javax.sql.DataSource;

import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class DataSourceConfiguration {

    @Bean
    public DataSource getDataSource() {
        return DataSourceBuilder.create().driverClassName("org.h2.Driver").url("jdbc:h2:mem:test").username("sa")
                .password("").build();
    }
}
```

```
# Database configuration

spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```

SPRING DATA JDBC: TEMPLATES

- Gracias a la AutoConfiguration de Spring Boot, al usar Spring Data JDBC, se autoconfiguran dos Beans en base al DataSource definido que podemos usar para realizar operaciones contra la base de datos: JdbcTemplate y NamedParameterJdbcTemplate

```
@Autowired
private NamedParameterJdbcTemplate jdbcTemplate;
```

SPRING DATA JDBC: TEMPLATES

- Su uso es similar al que usamos con JDBC puro

```
@Autowired
private JdbcTemplate jdbcTemplate;
```

SPRING DATA JDBC: TEMPLATES

```
@Autowired
private JdbcTemplate jdbcTemplate;

public int count() {
    return jdbcTemplate.queryForObject("SELECT count(*) FROM USER", Integer.class);
}

public int save(User user) {
    return jdbcTemplate.update(
            "INSERT INTO `USER` (USER_NAME, PASSWORD, CREATED_TIME, UPDATED_TIME, USER_TYPE, DOB)"
                    + " VALUES(?,?,?,?,?,?)",
            user.getUserName(), user.getPassword(), user.getCreatedTime(), user.getUpdatedTime(),
            user.getUserType().toString(), user.getDateofBirth());
}
```

CONCEPTOS: ENTITY

- Clases del modelo que tienen que tener un @Id en la BD
- Cada Entity tiene asociada una tabla de base de datos

```
@Autowired
private NamedParameterJdbcTemplate jdbcTemplate;

public int count() {
    return jdbcTemplate.queryForObject("SELECT count(*) FROM USER", new MapSqlParameterSource(), Integer.class);
}

public int save(User user) {
    MapSqlParameterSource mapSqlParameterSource = new MapSqlParameterSource();
    mapSqlParameterSource.addValue("userName", user.getUserName());
    mapSqlParameterSource.addValue("password", user.getPassword());
    mapSqlParameterSource.addValue("createdTime", user.getCreatedTime());
    mapSqlParameterSource.addValue("userType", user.getUserType().toString());
    mapSqlParameterSource.addValue("dateofBirth", user.getDateofBirth());

    return jdbcTemplate.update("INSERT INTO `USER` (USER_NAME, PASSWORD, CREATED_TIME, USER_TYPE, DOB)"
            + " VALUES(:userName,:password,:createdTime,:userType,:dateofBirth)", mapSqlParameterSource);
}
```

```
@Table("BOOK")
public class Book {

    @Id
    private Long id;

    private String name;

    private String isbn;

    private Double price;

    private LocalDate publishedDate;

}
```

CONCEPTOS: VALUE OBJECT (DTO)

- Uso de @Embedded. No se crea una tabla aparte

```
@Table("PERSON")
public class Person {

    @Id
    private Long id;

    private String first_name;

    private String last_name;

    private String phone;

    private String city;

    private String zipCode;

    private String country;
}
```

```
@Table("PERSON")
public class Person {

    @Id
    private Long id;

    private String first_name;

    private String last_name;

    @Embedded(onEmpty = OnEmpty.USE_EMPTY)
    private ContactAddress contactAddress;
}
```

CONCEPTOS: AGGREGATES

- Entidades que agrupan otras entidades de la base de datos
- La entidad padre o Aggregate Root, es la única entidad que puede ser cargada desde un repository (esquema DDD), y también es la encargada de manejar las entidades hijas

CONCEPTOS: AGGREGATES

```java
@Table("PURCHASE_ORDER")
public class PurchaseOrder {

    private @Id Long id;
    @Column("SHIPPING_ADDRESS")
    private String shippingAddress;
    // Child Entity
    private Set<OrderItem> items = new HashSet<>();

    public void addItem(int quantity, String product) {
        items.add(createOrderItem(quantity, product));
    }

    private OrderItem createOrderItem(int quantity, String product) {

        final OrderItem item = new OrderItem();
        item.product = product;
        item.quantity = quantity;
        return item;
    }

    public void setItems(Set<OrderItem> items) {
        this.items = items;
    }
}
```

```java
@Table("ORDER_ITEM")
public class OrderItem {
    @Id
    Long purchase_order;
    int quantity;
    String product;
}
```

CONCEPTOS: REFERENCES

- Son entidades “intermedias” en una relación entre entidades
- Se utilizan para relacionar entidades de base de datos que están al mismo nivel, es decir, no existe entidad padre ni entidad hija
- Hacen referencia a otra entidad a través de un identificador

CONCEPTOS: REFERENCES

```java
public class Branch {
    @Id
    private Long branchId;
    private String branchName;
    @Column("BRANCH_SHORT_NAME")
    private String branchShortName;
    private String description;
    @MappedCollection(idColumn = "BRANCH_ID")
    private Set<SubjectRef> subjects = new HashSet<>();

    public void addSubject(Subject subject) {
        subjects.add(new SubjectRef(subject.getSubjectId()));
    }
}
```

```java
public class Subject {
    @Id
    private Long subjectId;
    private String subjectDesc;
    private String subjectName;
}
```

```java
@Table("BRANCH_SUBJECT")
@Data
@AllArgsConstructor
public class SubjectRef {
    Long subjectId;
}
```

REPOSITORIOS

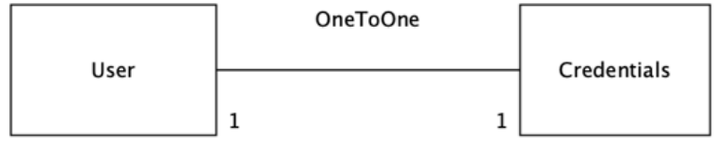
- Interfaz de acceso a base de datos. Desde estos beans se realizan todas las operaciones del CRUD
- Son interfaces que extienden de CrudRepository
- CrudRepository trae ya todos los métodos básicos CRUD: findAll(), delete(), findById(), save()…
- Se puede crear más métodos con el uso de la anotación @Query
- Siguiendo el paradigma DDD, un repositorio por Agreggate Root

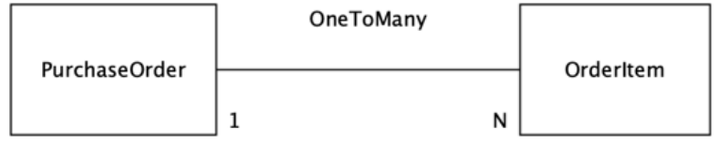
REPOSITORIOS

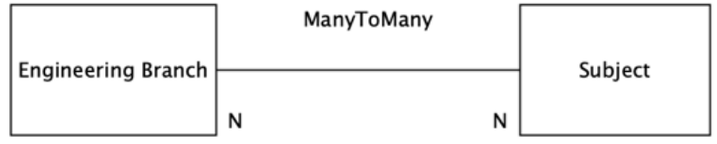
```
interface OrderRepository extends CrudRepository<PurchaseOrder, Long> {

    @Query("select count(*) from order_item")
    int countItems();

    @Query("select * from purchase_order where purchase_order.shipping_address= :address")
    List<PurchaseOrder> findByShippingAddress(@Param("address") String address);
}
```

RELACIONES

- One to One: 1-1
- One to Many: 1-n
- Many to Many: m-n







ONE TO ONE

```java
public class User {
    public enum UserType {
        EMPLOYEE, STUDENT;
    }

    @Id
    private Long id;
    private Date createdTime;
    private Date updatedTime;
    @Column("DOB")
    private Date dateofBirth;
    private UserType userType;

    @MappedCollection(idColumn = "CREDS_ID")
    private Credentials credentials;
}
```

```java
public class Credentials {
    @Id
    private Long credsId;
    private String userName;
    private String password;
}
```

ONE TO MANY

```java
@Table("PURCHASE_ORDER")
public class PurchaseOrder {

    private @Id Long id;
    @Column("SHIPPING_ADDRESS")
    private String shippingAddress;
    // Child Entity
    @MappedCollection(keyColumn = "PURCHASE_ORDER", idColumn = "PURCHASE_ORDER")
    private Set<OrderItem> items = new HashSet<>();

    public void addItem(int quantity, String product) {
        items.add(createOrderItem(quantity, product));
    }

    private OrderItem createOrderItem(int quantity, String product) {

        final OrderItem item = new OrderItem();
        item.product = product;
        item.quantity = quantity;
        return item;
    }

    public void setItems(Set<OrderItem> items) {
        this.items = items;
    }
}
```

```java
@Table("ORDER_ITEM")
public class OrderItem {
    int quantity;
    String product;
}
```

MANY TO MANY

```java
public class Branch {
    @Id
    private Long branchId;
    private String branchName;
    @Column("BRANCH_SHORT_NAME")
    private String branchShortName;
    private String description;
    @MappedCollection(idColumn = "BRANCH_ID")
    private Set<SubjectRef> subjects = new HashSet<>();

    public void addSubject(Subject subject) {
        subjects.add(new SubjectRef(subject.getSubjectId()));
    }
}
```

```java
public class Subject {
    @Id
    private Long subjectId;
    private String subjectDesc;
    private String subjectName;
}
```

```java
@Table("BRANCH_SUBJECT")
@Data
@AllArgsConstructor
public class SubjectRef {
    Long subjectId;
}
```

TRANSACCIONES

- Cuando se realizan operaciones de escritura es necesario manejar transacciones para evitar inconsistencias en la base de dato
- En Spring podemos hacer uso de la anotación @Transactional a nivel de método, en repositorios y servicios
- Si se produce alguna excepción dentro de dicho método, Spring hará un rollback de dicha transacción, dejando la BD en el estado original
- Las operaciones de escritura por defecto de los repositorios que extienden de CrudRepository son por defecto, transaccionales

TRANSACCIONES: @TRANSACTIONAL

```java
@Service
public class PersonServiceImpl implements PersonService {

    @Autowired
    private JdbcTemplate jdbcTemplate;


    @Transactional
    public void addBulkData() {
        jdbcTemplate.execute("INSERT INTO PERSON(FIRST_NAME, LAST_NAME) VALUES ('Victor', 'Hugo')");
        jdbcTemplate.execute("INSERT INTO Person(FIRST_NAME, LAST_NAME) VALUES ('Dante', 'Alighieri')");
        jdbcTemplate.execute("INSERT INTO Person(FIRST_NAME, LAST_NAME) VALUES ('Stefan', 'Zweig')");
        jdbcTemplate.execute("INSERT INTO Person(FIRST_NAME, LAST_NAME) VALUES ('Oscar', 'Wilde')");
    }

}
```

TRANSACCIONES: @MODIFYING

- Las queries que sean del tipo UPDATE y DELETE, además de ser transaccionales deberán llevar la anotación @Modifying para poder ser ejecutadas

```
public interface PersonRepository extends CrudRepository<Person, Long> {

    @Transactional
    @Query("UPDATE PERSON SET PERSON.FIRST_NAME= :userName WHERE PERSON.ID= :id ")
    @Modifying
    public int updateUserNameById(@Param("userName") String userName, @Param("id") Long id);

}
```

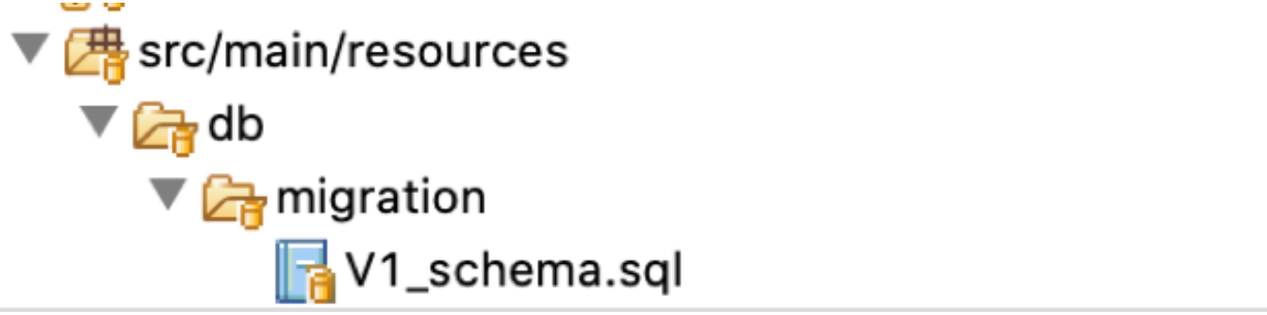
SCHEMA DE BASE DE DATOS

- Cuando levantamos nuestra aplicación, ésta necesita que la base de datos esté activa y que exista el schema
- Para esta tarea, Spring nos proporciona unas properties donde podemos indicarle rutas a scripts SQL que ejecutará al levantar la aplicación

SCHEMA DE BASE DE DATOS

```sql
DROP SCHEMA IF EXISTS BASICS_SCHEMA;

CREATE schema IF NOT EXISTS BASICS_SCHEMA;
USE BASICS_SCHEMA;

DROP TABLE IF EXISTS BOOK;

CREATE TABLE BOOK (
    ID INTEGER IDENTITY PRIMARY KEY,
    NAME VARCHAR(256) NOT NULL,
    ISBN VARCHAR(16) NOT NULL,
    PRICE DECIMAL(10,2) NOT NULL,
    PUBLISHED_DATE DATE NOT NULL
);

DROP TABLE IF EXISTS PERSON;

CREATE TABLE PERSON (
    ID INTEGER IDENTITY PRIMARY KEY,
    FIRST_NAME VARCHAR(256) NOT NULL,
    LAST_NAME VARCHAR(256) NOT NULL
);

DROP TABLE IF EXISTS USER;

CREATE TABLE USER (
    ID INTEGER IDENTITY PRIMARY KEY,
    USER_NAME VARCHAR(45) NOT NULL,
    PASSWORD VARCHAR(45) NOT NULL,
    EMAIL VARCHAR(100) DEFAULT NULL,
    CREATED_TIME DATETIME NOT NULL,
    UPDATED_TIME DATETIME DEFAULT NULL,
    USER_TYPE VARCHAR(45) NOT NULL,
    DOB DATE NOT NULL
);
```

SCHEMA DE BASE DE DATOS




```
INSERT INTO `USER` VALUES
     (1,'PeterM','ABC123abc*','peter@email.com','2020-03-17 07:13:30',NULL,'STUDENT','2020-03-17'),
     (2,'Mike','password','mike@email.com','2020-03-18 14:59:35',NULL,'EMPLOYEE','2020-03-18'),
     (3,'KingPeter','password','kingpeter@email.com','2020-03-19 12:19:15',NULL,'EMPLOYEE','2020-03-18'),
     (4,'PeterH','ABC123abc*','peterh@email.com','2020-03-17 07:13:30',NULL,'STUDENT','2020-03-17'),
     (5,'Kelvin','password','kelvin@email.com','2020-03-18 14:59:35',NULL,'EMPLOYEE','2020-03-18'),
     (6,'PeterLouise','password','peterl@email.com','2020-03-19 12:19:15',NULL,'EMPLOYEE','2020-03-18'),
     (7,'JustinB','ABC123abc*','justin@email.com','2020-03-17 07:13:30',NULL,'STUDENT','2020-03-17'),
     (8,'AshjaA','password','ashja@email.com','2020-03-18 14:59:35',NULL,'EMPLOYEE','2020-03-18'),
     (9,'JenniferH','password','jennifer@email.com','2020-03-19 12:19:15',NULL,'EMPLOYEE','2020-03-18'),
     (10,'DonaldT','ABC123abc*','donald@email.com','2020-03-17 07:13:30',NULL,'STUDENT','2020-03-17'),
     (11,'HilloryK','password','hillory@email.com','2020-03-18 14:59:35',NULL,'EMPLOYEE','2020-03-18'),
     (12,'MartinKing','password','martin@email.com','2020-03-19 12:19:15',NULL,'EMPLOYEE','2020-03-18');
```

TESTING: DATOS DE INICIALIZACIÓN

- En el caso del testing, podemos cargar los datos en la base de datos bien por código o bien mediante un script SQL
- De manera similar al script del Schema, podemos dejarlo en src/main/resources , y hacer uso de una anotación que nos proporciona Spring para cargarlo en la ejecución del test

TESTING: DATOS DE INICIALIZACIÓN

TESTING: DATOS DE INICIALIZACIÓN

```java
@Sql(scripts= "/sql_data.sql")
@Transactional
public class NamedParameterJdbcTemplateTest {

    @Autowired
    private UserNamedParameterJdbcTemplateRepository userRepository;

    @Test
    void given_repository_when_add_user_then_Ok() {

        //Given

        //When
        User user = getUser();
        int created = userRepository.save(user);

        //Then
        then(created).isEqualTo(1);
    }
```

TEST CONTAINERS

- Utilizados en el contexto de Testing
- Nos permiten crear de manera sencilla contenedores a modo de sandbox para ejecutar los tests
- Reemplazan a los servicios reales con los que trabajaría nuestra aplicación
- P.e. Una base de datos

TEST CONTAINERS: POSTGRESQL

```
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
public class DockerPostgreDataSourceInitializer
        implements ApplicationContextInitializer<ConfigurableApplicationContext> {

    public PostgreSQLContainer<?> postgreSQLContainer = new PostgreSQLContainer<>("postgres:13");

    @Override
    public void initialize(ConfigurableApplicationContext configurableApplicationContext) {
        postgreSQLContainer.start();

        TestPropertySourceUtils.addInlinedPropertiesToEnvironment(configurableApplicationContext,
                "spring.datasource.url=" + postgreSQLContainer.getJdbcUrl(),
                "spring.datasource.username=" + postgreSQLContainer.getUsername(),
                "spring.datasource.password=" + postgreSQLContainer.getPassword());
    }
}
```

TEST CONTAINERS: POSTGRESQL

```
@ContextConfiguration(initializers = DockerPostgreDataSourceInitializer.class)
public class BookRepositoryTest {

    @Autowired
    private BookRepository bookRepository;

    @Test
    public void createBookWithAuthor() {

        final Book book = Book.builder().name("My Book").isbn("ISBN1234").publishedDate(LocalDate.of(2018, 12, 12))
                .price(12.99).build();

        bookRepository.save(book);

        final Book savedBook = bookRepository.findById(book.getId()).orElseThrow(RuntimeException::new);
        then(savedBook.getId()).isNotNull();
    }
}
```

TEST CONTAINERS: POSTGRESQL

```
2021-04-11 20:32:06 [main] INFO  o.testcontainers.DockerClientFactory - Ryuk started - will monitor and terminate Testcontainers containers on JVM exit
        ℹ Checking the system...
        ✔ Docker version should be at least 1.6.0
        ✔ Docker environment should have more than 2GB free disk space


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.4.4)

2021-04-11 20:32:06 [main] INFO  🐳 [postgres:13] - Creating container for image: postgres:13
2021-04-11 20:32:06 [main] INFO  🐳 [postgres:13] - Starting container with ID: dfb1275733b78c1a80da71b032c6529946a75b63b451148084f1684d286e5116
2021-04-11 20:32:07 [main] INFO  🐳 [postgres:13] - Container postgres:13 is starting: dfb1275733b78c1a80da71b032c6529946a75b63b451148084f1684d286e5116
2021-04-11 20:32:09 [main] INFO  🐳 [postgres:13] - Container postgres:13 started
```

REFERENCIAS

- https://spring.io/projects/spring-data
- https://spring.io/projects/spring-data-jdbc
- https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html#spring.datasource.url
- https://javabydeveloper.com/
- https://medium.com/ingenier%C3%ADa-en-tiendanube/domain-driven-design-y-el-modelo-de-actores-deae7675a921
- https://airbrake.io/blog/software-design/domain-driven-design
- https://www.paradigmadigital.com/dev/ddd-dominio-implica-crecer-fuerte/
- https://docs.spring.io/spring-data/jdbc/docs/current/reference/html/
- https://www.testcontainers.org/modules/databases/postgres/