

TRABAJO FINAL

PERSISTENCIA RELACIONAL Y
NO RELACIONAL DE DATOS

Máster de TWCAM, ETSE-UV
Pablo Gómez Bolós y Diego Ruiz Sierra

Índice

1. Introducción.....	3
2. Apartado Relacional.....	4
2.1. Parking.....	4
2.1.1. Modelo de datos SQL.....	5
2.1.2. Mapeo con JPA.....	6
2.1.3. Repositorio de JPA.....	6
2.1.4. Inicialización de la base de datos.....	6
2.2. Station.....	7
2.2.1. Modelo de datos SQL.....	7
2.2.2. Mapeo con JPA.....	8
2.2.3. Repositorio de JPA.....	9
2.2.4. Inicialización de la base de datos.....	9
3. Apartado No relacional.....	10
3.1. Eventos de bicicletas.....	10
3.1.1. Modelo de datos NoSQL.....	10
3.1.2. Mapeo con Spring Data.....	11
3.1.3. Repositorio de Spring Data.....	11
3.1.4. Inicialización de los documentos.....	12
3.2. Lecturas de polución.....	13
3.2.1. Modelo de datos NoSQL.....	13
3.2.2. Mapeo con Spring Data.....	13
3.2.3. Repositorio de Spring Data.....	14
3.2.4. Inicialización de los documentos.....	15
3.3. Datos agregados para el ayuntamiento.....	15
3.3.1. Modelo de datos NoSQL.....	15
3.3.2. Mapeo con Spring Data.....	16
3.3.3. Repositorio de Spring Data.....	17
3.3.4. Inicialización de los documentos.....	17
3.3.5. Proyecto shared.....	17
4. Resultados.....	18
4.1. Ejecución del código.....	18
4.2. Conclusiones.....	19

1. Introducción

Para este proyecto se pide realizar la simulación de un ayuntamiento que subcontrata dos empresas:

- Una que provee aparcamientos de bicicletas por la ciudad.
- Otra que coloca estaciones con sensores que miden la polución en el ambiente.

Así, el ayuntamiento debe poder acceder a los datos que recogen estas dos empresas para analizarlos posteriormente. En esta memoria se detallan las bases de datos utilizadas para conseguir este objetivo. Se usa para ello:

- Dos bases de datos SQL gestionadas con MySQL Workbench que guardan, respectivamente:
 - Los datos referidos a los identificadores y ubicación de los aparcamientos.
 - Los datos referidos a los identificadores y ubicación de los sensores.
- Tres bases de datos NoSQL que utilizan MongoDB para persistir, cada una:
 - Las operaciones o eventos realizados en los aparcamientos.
 - Los datos obtenidos por las lecturas de los sensores.
 - Las agregaciones de los datos de los aparcamientos junto con los de los sensores.

De esta forma en esta memoria se especifica, para cada tipo de base de datos, las clases o documentos utilizados junto con sus modelos de datos, el mapeo de estas clases en el proyecto en Spring y los repositorios utilizados en la aplicación para manejar estos datos. También se especifica cómo se inicializan las bases de datos tanto para el modelo relacional como el no relacional.

2. Apartado Relacional

Todo lo explicado en este apartado se puede encontrar en el proyecto Spring en diferentes carpetas. Cada API está formada por dos proyectos: el servicio (relevante para la asignatura de “Desarrollo de componentes distribuidos y servicios”) y la capa de acceso de datos (son las carpetas cuyo nombre acaba en “-data”). Estas capas de acceso son las que interesan para esta memoria y en las que se deben buscar las clases que aquí se detallan. Así:

- Las clases que mapean con JPA la información de la base de datos se encuentran en la carpeta *model*.
- Los repositorios se encuentran en la carpeta *repository*.
- El documento que contiene los datos iniciales se encuentra en */resources/data.sql*.
- Los controladores que llaman a las clases del repositorio (sólo relevantes para la asignatura de DCDBS) se encuentran en la carpeta *controllers*.

Sabiendo esto, se explica ahora el apartado relacional.

2.1. Parking

Para gestionar los aparcamientos de las bicicletas se crea una base de datos llamada “aparcamiento”, de la siguiente manera:

```
CREATE DATABASE `aparcamiento`
```

La conexión con MySQL Workbench y la configuración de JPA se establece en el fichero *properties* de *bicicletasdata* en el *config-server*. También se configuran otros elementos como la utilización de ficheros de inicialización de datos al desplegar la aplicación. Así:

```
#sql
spring.datasource.url=jdbc:mysql://localhost:3306/aparcamiento?serverTimezone=Europe/Madrid
spring.datasource.username=root
spring.datasource.password=root

#jpa
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.sql.init.mode=always
```

Se detalla ahora el modelo de datos, el mapeo y el repositorio utilizados para esta base de datos. También se muestra cómo se añaden los datos de prueba iniciales.

2.1.1. Modelo de datos SQL

Para la base de datos del aparcamiento solo se necesita una clase, llamada Parking. En el modelo de datos de Parking se representa un aparcamiento de bicicletas con su identificador y sus datos básicos de ubicación, además de su capacidad para guardar bicicletas. Este modelo fue proporcionado en el enunciado y se resume en la siguiente imagen:

Parking	
string	ID
string	direction
int	bikesCapacity
float	latitude
float	longitude

Se explica con una tabla cómo se interpreta el modelo en la base de datos:

Columna	Tipo de dato en SQL	Consideraciones adicionales
ID	VARCHAR(255)	PRIMARY KEY (por lo tanto NOT NULL)
direccion	VARCHAR(255)	NOT NULL
bikes_capacity	int	NOT NULL
longitud	float	NOT NULL
latitud	float	NOT NULL

El script usado para introducir la tabla en la base de datos es el siguiente:

```
CREATE TABLE `parking` (
  `idparking` varchar(255) NOT NULL,
  `direction` varchar(255) NOT NULL,
  `bikes_capacity` int NOT NULL,
  `latitude` float NOT NULL,
  `longitude` float NOT NULL,
  PRIMARY KEY (`idparking`)
);
```

2.1.2. Mapeo con JPA

Para obtener los datos de la tabla en el programa se utiliza JPA en la clase *Parking.java* dentro de la carpeta *model*. Se configura de la siguiente forma, junto con sus métodos *get* y *set*:

```
@Entity
@Table(name = "parking")
public class Parking {

    @Id
    private String idparking;

    @Column(nullable = false)
    private String direction;

    @Column(name = "bikes_capacity", nullable = false)
    private Integer bikesCapacity;

    @Column(nullable = false)
    private float latitude;

    @Column(nullable = false)
    private float longitude;

    public String getIdparking() {
        return idparking;
    }
}
```

2.1.3. Repositorio de JPA

No se han necesitado operaciones especiales para operar con los datos de Parking, por lo que no se ha añadido ningún método extra al repositorio:

```
@Repository
public interface ParkingRepository extends JpaRepository<Parking, String> {
}
```

2.1.4. Inicialización de la base de datos

Para inicializar unos datos de prueba se han incluido estas órdenes en un fichero *data.sql* dentro de las carpetas *main\resources* del servicio del parking:

```
USE aparcamiento;

CREATE TABLE IF NOT EXISTS parking (
    idparking VARCHAR(255) PRIMARY KEY,
    direction VARCHAR(255) NOT NULL,
    bikes_capacity INT NOT NULL,
    latitude FLOAT NOT NULL,
    longitude FLOAT NOT NULL
);
```

```
DELETE FROM parking;

INSERT INTO parking (idparking, direction, bikes_capacity, latitude,
longitude)
VALUES
('1', 'Avinguda de la Universitat, 46100 Burjassot, Valencia', 25, 39.5127,
-0.42462),
('2', 'Carrer del Catedràtic José Beltrán Martínez, 2, 46980 Paterna,
Valencia', 20, 39.5136, -0.425218),
('3', 'Plaça de l'Ajuntament, 1, 46002 València, Valencia', 15, 39.4698,
-0.37640);
```

2.2. Station

Para gestionar las estaciones de medición de polución se crea una base de datos llamada “polución”, de la siguiente manera:

```
CREATE DATABASE `polucion`
```

Igual que con Parking, la conexión con MySQL Workbench y la configuración de JPA se puede encontrar en el fichero *properties* de polución en el *config-server*. Así:

```
# SQL
spring.datasource.url=jdbc:mysql://localhost:3306/polucion?serverTimezone=Europe/Madrid
spring.datasource.username=root
spring.datasource.password=root

# JPA
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.sql.init.mode=always
```

Se detalla ahora el modelo de datos, el mapeo y el repositorio utilizados para esta base de datos. También se muestra cómo se añaden los datos de prueba iniciales.

2.2.1. Modelo de datos SQL

Para la base de datos de polución solo se necesita una clase, llamada Station. En el modelo de datos de Station se representa una estación de medición de la polución con su identificador y sus datos básicos de ubicación. Este modelo fue proporcionado en el enunciado y se resume en la siguiente imagen:

Station	
String	ID
String	direction
float	latitude
float	longitude

Se explica con una tabla cómo se interpreta el modelo en la base de datos:

Columna	Tipo de dato en SQL	Consideraciones adicionales
ID	VARCHAR(255)	PRIMARY KEY (por lo tanto NOT NULL)
direccion	VARCHAR(255)	NOT NULL
longitud	float	NOT NULL
latitud	float	NOT NULL

El script usado para introducir la tabla en la base de datos es el siguiente:

```
CREATE TABLE `estacion` (
  `id` varchar(255) NOT NULL,
  `direccion` varchar(255) NOT NULL,
  `longitud` float NOT NULL,
  `latitud` float NOT NULL,
  PRIMARY KEY (`id`)
)
```

2.2.2. Mapeo con JPA

Para obtener los datos de la tabla en el programa se utiliza JPA en la clase *Estacion.java* dentro de la carpeta *model*. Se configura de la siguiente forma, junto con sus métodos *get* y *set*:


```
@Entity
@Table(name = "estacion")
public class Estacion {
    @Id
    private String id;

    @Column(nullable = false)
    private String direccion;

    @Column(nullable = false)
    private float latitud;

    @Column(nullable = false)
    private float longitud;
}
```

2.2.3. Repositorio de JPA

No se han necesitado operaciones especiales para operar con los datos de Estacion, por lo que no se ha añadido ningún método extra al repositorio:

```
@Repository
public interface EstacionRepository extends JpaRepository<Estacion, String> {
}
```

2.2.4. Inicialización de la base de datos

Para inicializar unos datos de prueba se han incluido estas órdenes en un fichero *data.sql* dentro de las carpetas *main/resources* del servicio de polución:

```
USE polucion;

CREATE TABLE IF NOT EXISTS estacion (
    id VARCHAR(255) PRIMARY KEY,
    direccion VARCHAR(255) NOT NULL,
    latitud FLOAT NOT NULL,
    longitud FLOAT NOT NULL
);

DELETE FROM estacion;

INSERT INTO estacion (id, direccion, latitud, longitud) VALUES ('1',
'Avinguda de l'Universitat, 46100 Burjassot, Valencia', 39.512561,
-0.424610);
INSERT INTO estacion (id, direccion, latitud, longitud) VALUES ('2', 'Carrer
del Catedrático José Beltrán Martínez, 2, 46980 Paterna, Valencia',
39.513506, -0.425217);
INSERT INTO estacion (id, direccion, latitud, longitud) VALUES ('3', 'Plaça
de l'Ajuntament, 1, 46002 València, Valencia', 39.46975, -0.37639);
```

3. Apartado No relacional

Como ya se ha comentado para este apartado se utilizará MongoDB. Así, igual que en el apartado relacional, todo lo detallado en este apartado se puede encontrar en el proyecto Spring en diferentes carpetas. En este caso las capas de acceso de datos (“-data”) también son de interés y son en las que se deben buscar las clases que aquí se detallan. Así:

- Las clases que mapean la información de Mongo con Spring Data se encuentran en la carpeta *model*.
- Los repositorios se encuentran en la carpeta *repository*.
- Las clases de importación que se utilizan para inicializar los datos de Mongo se encuentran en la carpeta de *imports*.
- También hay unas clases solamente relevantes para la asignatura de DBCDS en la carpeta *controllers*.

Sabiendo esto, se explica ahora el apartado no relacional.

3.1. Eventos de bicicletas

Para gestionar los diferentes eventos en los aparcamientos de bicicletas se utiliza una base de datos llamada *bicicletas* que se crea y se llama desde el fichero *properties* así:

```
#mongodb
spring.data.mongodb.uri=mongodb://127.0.0.1:27017/bicicletas
```

Se detalla ahora el modelo de datos, el mapeo y el repositorio utilizados para esta base de datos. También se muestra cómo se añaden los datos de prueba iniciales.

3.1.1. Modelo de datos NoSQL

Cuando se realiza un evento en uno de los parkings debe registrarse el tipo de evento (alquiler de una bici, devolución, alquiler múltiple o devolución múltiple), las bicis que quedan disponibles, los huecos que hay, el identificador de la estación donde ha ocurrido el evento y el momento en el que ha sucedido. Cada documento de la tabla, entonces, tendría el siguiente formato, según se ha definido en el enunciado:

```
{
  "id": (int),
  "operation": (String),
  "bikesAvailable": (int),
  "freeParkingSpots": (int),
  "timestamp": (LocalDateTime)
}
```

3.1.2. Mapeo con Spring Data

Para obtener los datos del documento en el programa se utiliza Spring Data en la clase *Evento.java* dentro de la carpeta *model*. Se configura de la siguiente forma, junto con sus métodos *get* y *set*:

```
@Document(collection = "eventos")
public class Evento {
    @Id
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    @JsonIgnore
    private String mongoId;

    @JsonProperty("parkingId")
    private String parkingId;

    @JsonProperty("operation")
    private String operation;

    @JsonProperty("bikesAvailable")
    private Integer bikesAvailable;

    @JsonProperty("freeParkingSpots")
    private Integer freeParkingSpots;

    @JsonProperty("timestamp")
    private LocalDateTime timestamp;
}
```

Como se puede ver en la imagen:

- Se ha configurado la colección de la base de datos con el nombre de *eventos*.
- Se ha anotado cada atributo de la clase con *@JsonProperty* para conseguir un mapeo correcto.
- Se ha configurado el id de mongo como *@Id* y se ha anotado con *@JsonIgnore* para que no lo tenga en cuenta como elemento de los documentos.

3.1.3. Repositorio de Spring Data

El repositorio, que extiende de *MongoRepository*, contiene tres métodos que se utilizan en diferentes endpoints de la API:

```
@Repository
public interface EventoRepository extends MongoRepository<Evento, String> {
    List<Evento> findByParkingIdAndTimestampBetween(String parkingId, LocalDateTime from, LocalDateTime to);

    Optional<Evento> findFirstByParkingIdOrderByTimestampDesc(String parkingId);

    List<Evento> findByTimestampLessThanEqual(LocalDateTime fecha);
}
```

- *findByParkingIdAndTimestampBetween*:
 - Devuelve una lista de los Eventos producidos en un parking específico (*findByParkingId...*) con el identificador pasado por parámetro. Estos eventos se filtran por el momento en el que ocurrió el evento (*...andTimestampBetween*) entre las fechas *from* y *to* del argumento de la función.
 - Se utiliza en el endpoint */aparcamiento/{id}/status* con parámetros, que muestra los cambios de estado de una parada en un cierto espacio de tiempo.

- *findFirstByParkingIdOrderByTimestampDesc*:
 - Devuelve el primer evento obtenido de una lista (*findFirstByParkingId...*) que se encuentra ordenada por fecha en orden descendente (*...OrderByTimestampDesc*), por lo que devuelve el evento más reciente de la lista para un parking concreto. Devuelve un *Optional*, por lo que puede que devuelva un objeto vacío si no hay datos.
 - Se utiliza en el endpoint */aparcamiento/{id}/status* sin parámetros, que muestra el último estado registrado de un parking.
- *findByTimestampLessThanEqual*:
 - Recupera todos los eventos que ocurrieron en o antes de la *fecha* pasada por parámetro.
 - Se utiliza en el endpoint */aparcamiento/available*, que muestra los 10 parkings con más bicis disponibles en este momento.

3.1.4. Inicialización de los documentos

Para la inicialización de documentos en la base de datos se crea una clase llamada *ImportServiceMongo* en la carpeta *imports*. Esta clase, si no detecta que ya hay documentos en la colección, obtiene el archivo con los datos (llamado *evento.txt*), lo lee línea por línea y mapea cada Evento, guardándolo después en el repositorio. En esta aplicación hay una clase de este tipo para cada base de datos de Mongo:

```
@Service
public class ImportServiceMongo {

    @Autowired
    private EventoRepository eventoRepository;

    public void importarEventos(Resource resource) {
        if (eventoRepository.count() == 0) {
            ObjectMapper mapper = new ObjectMapper();
            mapper.registerModule(new JavaTimeModule());

            try (Scanner scanner = new Scanner(resource.getFile())) {
                while (scanner.hasNextLine()) {
                    String json = scanner.nextLine();
                    Evento evento = mapper.readValue(json, Evento.class);
                    eventoRepository.save(evento);
                }
                System.out.println(x:"Eventos insertados correctamente.");
            } catch (Exception e) {
                System.err.println("Error al importar eventos: " + e.getMessage());
            }
        }
    }
}
```

Para conseguir que esta clase se lea y cumpla su función se implementa la interfaz *CommandLineRunner* en la clase principal del microservicio de bicicletas, lo que permite llamar a la clase *ImportServiceMongo* al inicializar la aplicación. También se le indica a Spring que debe leer el proyecto tanto de *bicicletas* como la capa de acceso a los datos con *scanBasePackages* en *@SpringBootApplication*. Toda esta configuración también se hace para cada base de datos de Mongo en la aplicación, como se irá detallando en esta memoria:

```

@SpringBootApplication(scanBasePackages = {
    "twcam.proyecto.bicicletas",
    "twcam.proyecto.bicicletasdata"
})
@EnableJpaRepositories(basePackages = "twcam.proyecto.bicicletasdata.repository")
@EnableMongoRepositories(basePackages = "twcam.proyecto.bicicletasdata.repository")
@EntityScan(basePackages = "twcam.proyecto.bicicletasdata.model")
@OpenAPIDefinition(info = @Info(title = "API de bicicletas", version = "v1", contact = @Contact(name = "Pablo Gómez/Diego Ruiz", email = "pagobo2@alumni.uv.es/dieruiz4@alumni.uv.es"))
public class BicicletasApplication implements CommandLineRunner {
    @Autowired
    private ImportServiceMongo importServiceMongo;

    @Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(BicicletasApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        importServiceMongo.importarEventos(new ClassPathResource("evento.txt"));
    }
}

```

3.2. Lecturas de polución

Para gestionar las diferentes lecturas de contaminantes en las estaciones de medición se utiliza una base de datos llamada *polucion* que se crea y se llama desde el fichero *properties* así:

```

# MongoDB
spring.data.mongodb.uri=mongodb://localhost:27017/polucion

```

Se detalla ahora el modelo de datos, el mapeo y el repositorio utilizados para esta base de datos. También se muestra cómo se añaden los datos de prueba iniciales.

3.2.1. Modelo de datos NoSQL

Cuando se realiza una lectura se guarda el identificador de la estación en la que se ha medido, el momento en el que ha sucedido y los diferentes niveles de contaminantes que se han encontrado en el aire. Cada documento de la tabla, entonces, tendría el siguiente aspecto, según se ha definido en el enunciado:

```

{
    "id": (int),
    "timestamp": (Instant)
    "nitricOxides": (float),
    "nitrogenDioxides": (float),
    "VOCs_NMHC": (float),
    "PM2_5": (float)
}

```

3.2.2. Mapeo con Spring Data

Para obtener los datos del documento en el programa se utiliza Spring Data en la clase *Lectura.java* dentro de la carpeta *model*. Se configura de la siguiente forma, junto con sus métodos *get* y *set*:

```
@Document(collection = "lecturas")
public class Lectura {
    @Id
    @JsonIgnore
    private String mongoId;

    @JsonProperty("id")
    private int id;

    @JsonProperty("timeStamp")
    private Instant timeStamp;

    @JsonProperty("nitricOxides")
    private float nitricOxides;

    @JsonProperty("nitrogenDioxides")
    private float nitrogenDioxides;

    @JsonProperty("VOCs_NMHC")
    private float vocs_nmhc;

    @JsonProperty("PM2_5")
    private float pm2_5;
}
```

Como se puede ver en la imagen, se ha configurado la colección de la base de datos con el nombre de *lecturas*. También se utilizan las anotaciones *@JsonProperty*, *@Id* y *@JsonIgnore* con el mismo propósito que en el caso de los Eventos.

3.2.3. Repositorio de Spring Data

El repositorio, que extiende de *MongoRepository*, contiene dos métodos que se utilizan en diferentes endpoints de la API:

```
import org.springframework.data.mongodb.repository.MongoRepository;
import twcam.proyecto.polucion.model.mongo.Lectura;
import java.time.Instant;
import java.util.List;

public interface LecturaRepository extends MongoRepository<Lectura, String> {
    List<Lectura> findByEstacionIdOrderByTimeStampDesc(int estacionId);

    List<Lectura> findByEstacionIdAndTimeStampBetweenOrderByTimeStampDesc(int estacionId, Instant desde, Instant hasta);
}
```

- *findByEstacionIdOrderByTimeStampDesc*: igual que en Eventos, devuelve la lectura más reciente de una estación. Se utiliza en el endpoint */estacion/{id}/status* sin parámetros.
- *findByEstacionIdAndTimeStampBetweenOrderByTimeStampDesc*: también como en Eventos, devuelve las lecturas de una estación en un periodo indicado. Se utiliza en el endpoint */estacion/{id}/status* con parámetros.

3.2.4. Inicialización de los documentos

La inicialización de los documentos al desplegar la aplicación funciona igual que en los Eventos, siendo la única diferencia el nombre del archivo de inicialización: *lecturas.txt*.

3.3. Datos agregados para el ayuntamiento

Para gestionar las diferentes operaciones relacionadas con los datos agregados en el ayuntamiento se utiliza una base de datos llamada *ayuntamiento* que se crea y se llama desde el fichero *properties* así:

```
# MongoDB
spring.data.mongodb.uri=mongodb://localhost:27017/ayuntamiento
```

Se detalla ahora el modelo de datos, el mapeo y el repositorio utilizados para esta base de datos. También se muestra cómo se añaden los datos de prueba iniciales y el uso de la carpeta *shared*.

3.3.1. Modelo de datos NoSQL

Cuando se realiza el agregado de los datos de los eventos de las bicicletas y de las lecturas de polución se añade la fecha en la que se han agregado los datos y, para cada parking se guarda su identificador y la media de las bicis disponibles en ese aparcamiento. También se busca la estación de medición más cercana y se guarda la media de contaminantes atmosféricos medidos en esa estación. Así:

```
{
  "timestamp": (Instant)
  "aggregatedData": [
    {
      "id": (int)
      "average_bikesAvailable": (float)
      "air_quality": {
        "nitricOxides": (float)
        "nitrogenDioxides": (float)
        "VOCs_NMHC": (float)
        "PM2_5": (float)
      }
    },
    {
      "id": (int)
      ...
    },
    ...
  ]
}
```

3.3.2. Mapeo con Spring Data

Para obtener los datos del documento en el programa se utilizan tres clases diferentes dentro de la carpeta *model*, estructuradas tal y como se separan los datos en el documento. Estas clases son *AggregatedData.java*, *EstacionAggregatedData.java* y *AirQuality.java*. Se configuran de la siguiente forma, junto con sus métodos *get* y *set*:

```
@Document(collection = "aggregated_data")
public class AggregatedData {

    @JsonIgnore
    private String id;

    @JsonProperty("timeStamp")
    private Instant timeStamp;

    @JsonProperty("aggregatedData")
    private List<EstacionAggregatedData> aggregatedData;
```

```
public class EstacionAggregatedData {

    @JsonProperty("id")
    private int id;

    @JsonProperty("average_bikesAvailable")
    private float average_bikesAvailable;

    @JsonProperty("air_quality")
    private AirQuality air_quality;
```

```
public class AirQuality {

    @JsonProperty("nitricOxides")
    private float nitricOxides;

    @JsonProperty("nitrogenDioxides")
    private float nitrogenDioxides;

    @JsonProperty("VOCs_NMHC")
    private float vocs_nmhc;

    @JsonProperty("PM2_5")
    private float pm2_5;
```

Como se puede ver en la imagen de la clase *AggregatedData.java*, se ha configurado la colección de la base de datos con el nombre de *aggregated_data*. También se utilizan las anotaciones *@JsonProperty*, *@Id* y *@JsonIgnore* con el mismo propósito que en el caso de los Eventos y las Lecturas.

3.3.3. Repositorio de Spring Data

El repositorio, que extiende de *MongoRepository*, contiene solamente un método:

```
public interface AggregatedDataRepository extends MongoRepository<AggregatedData, String> {  
    @Query(sort = "{ timeStamp : -1 }")  
    AggregatedData findFirstOrderByTimeStampDesc();  
}
```

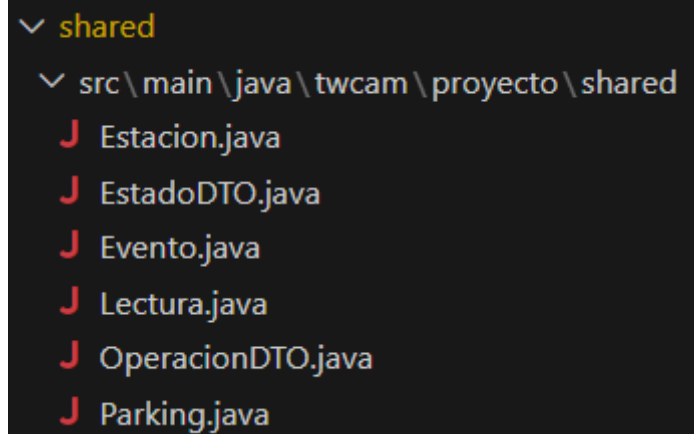
El método es *findFirstOrderByTimeStampDesc*, que obtiene los primeros datos agregados de la base de datos, ordenados de forma descendente. También se ordena de forma descendente mediante la anotación *@Query* para asegurar un correcto funcionamiento en Mongo. Se obtienen así los últimos datos registrados en la base de datos.

3.3.4. Inicialización de los documentos

La inicialización de los documentos al desplegar la aplicación funciona igual que en los Eventos y Lecturas, siendo la única diferencia el nombre del archivo de inicialización: *aggregatedData.txt*.

3.3.5. Proyecto *shared*

Con el objetivo de reutilizar clases de las APIs de bicicletas y de polución se ha creado un proyecto llamado *shared* con clases de interés pero con información reducida. Su principal objetivo es el uso de estas clases mayormente en la API del ayuntamiento sin que sea necesaria la declaración completa de la clase de nuevo. Las clases que se utilizan son las siguientes:



```
▼ shared  
  ▼ src\main\java\twcam\proyecto\shared  
    J Estacion.java  
    J EstadoDTO.java  
    J Evento.java  
    J Lectura.java  
    J OperacionDTO.java  
    J Parking.java
```

4. Resultados

4.1. Ejecución del código

Para poder ejecutar el proyecto se deben seguir los siguientes pasos:

- Para desplegar las bases de datos se puede utilizar el *docker-compose* ubicado en la carpeta *cloud* del proyecto, con la siguiente orden:

```
docker-compose -f docker-compose-mysql-mongo.yml up -d
```

- Ahora se despliegan en Spring, primero *authorization-server* y *config_server*.
- Luego, *bicicletas*, *bicicletas-data*, *polucion* y *polucion-data*.
- Después, *aparcamiento-worker*, *estacion-worker*, *ayuntamiento* y *ayuntamiento-data*.
- Y por último, *servicio-worker*.

Se pueden utilizar los diferentes endpoints de cada API en el navegador gracias a Swagger para poder comprobar el correcto funcionamiento y conexión de las bases de datos, en las siguientes páginas:

- API de bicicletas: [Swagger UI Bicis](#)
- API de polución: [Swagger UI Polución](#)
- API del ayuntamiento: [Swagger UI Ayunt](#)

En este ejemplo se muestra un endpoint que obtiene todas las estaciones de polución, donde se puede ver que se obtienen satisfactoriamente:

GET /estaciones Obtiene todas las estaciones

Obtiene un listado con todas las estaciones de medición de la aplicación junto con sus datos

Parameters

No parameters

Execute Clear

Responses

Code Details

200

Response body

```
{
  "id": "1",
  "direccion": "Avinguda de l'Universitat, 46100 Burjassot, Valencia",
  "latitud": 39.5126,
  "longitud": -0.42461
},
{
  "id": "2",
  "direccion": "Carreer del Catedràtic Josè Beltrà Martínez, 2, 46980 Paterna, Valencia",
  "latitud": 39.5135,
  "longitud": -0.425217
},
{
  "id": "3",
  "direccion": "Plaça de l'Ajuntament, 1, 46002 València, Valencia",
  "latitud": 39.4697,
  "longitud": -0.37639
}
}
```

4.2. Conclusiones

La implementación de la capa de persistencia, tanto la relacional con JPA como la no relacional con Spring Data, ha permitido modelar y almacenar de forma eficiente los datos generados por los aparcamientos de bicicletas y las estaciones de medición de polución.

Así, esta arquitectura híbrida combina las ventajas de ambos enfoques y se ajusta a los requisitos pedidos en el enunciado del proyecto, además de haber utilizado las prácticas aprendidas durante la asignatura de persistencia.