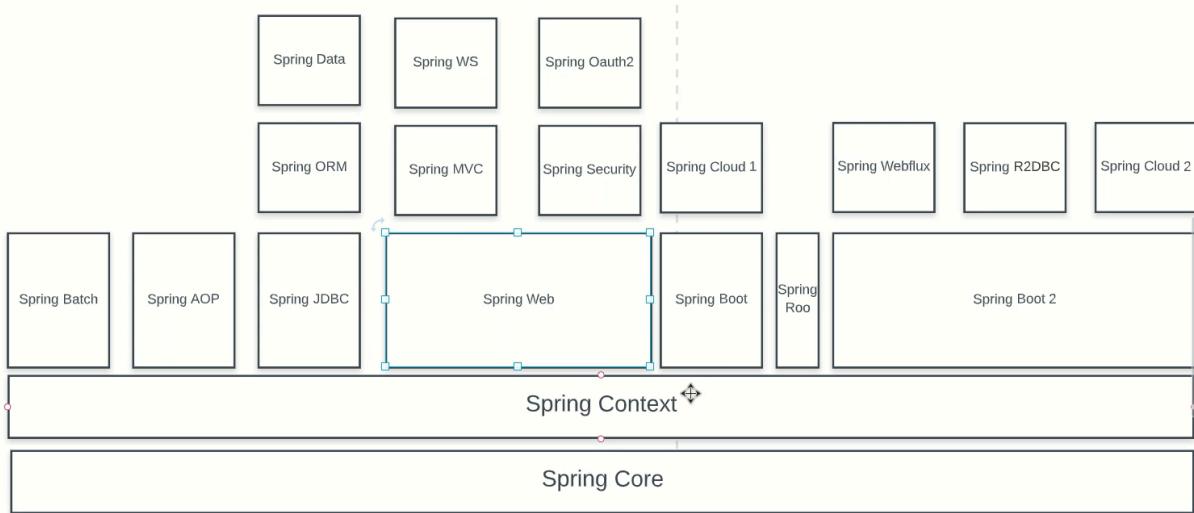


# Spring Boot 3 + Spring framework 6

## SECCIÓN 1 :INTRODUCCIÓN



**Spring web**- es el que hace los métodos POST, GET, etc..., es decir, necesitamos las dependencias de spring web, para poder hacer @PostMapping (método POST, etc..) y resto de métodos.

Las dependencias de Spring Web, contienen las dependencias de Spring Context y Spring Core.

**SPRING CORE:** El núcleo de todas clases de Spring, contiene todas las clases base para trabajar.

**SPRING CONTEXT:** Se encarga de hacer la inyección de dependencia y la inversión de control. Cuando hacemos un @Autowired o un @Component estamos usando Spring Context.

**SPRING BATCH:** No es muy usado. Más usado para el procesamiento por lotes, Apache Spark.

**SPRING AOP:** Para la programación Orientada a Aspectos. “La programación de aspectos es básicamente cuando tenemos un objeto que tiene el mismo proceso pero con diferentes objetos.”

**SPRING JDBC:** Usado para mapear la BBDD y nos daba el DataSource. El DataSource de Spring JDBC es Hikari (es muy eficiente).

**SPRING ORM:** Es el que se encarga de hacer un ORM. Hibernate es el más conocido.

**SPRING DATA:** Usado para cualquier tipo de base de datos, incluida las NO RELACIONALES!!.

**SPRING WEB:** Es el que se encarga de exponer nuestros endpoint a través del protocolo de HTTP. Con Spring Web, podemos hacer REST y SOAP.

**SPRING MVC:** No muy usado ya, usado para usar thymeleaf (el twig de Spring). Conecta nuestra vista con el controlador mediante el thymeleaf. Solo para uso de vista con Thymeleaf.

**SPRING WS (Web Services):** No se usa mucho en la actualidad. Se usaba para consumir servicios SOAP. (Ya no se usa tanto SOAP).

**SPRING SECURITY:** Utilizado para que nuestra aplicación web tenga seguridad. Al poner las dependencias, el proyecto automáticamente ya tiene seguridad, pero hay que configurarlo.

**SPRING Oauth2:** Este trabaja sobre el módulo Security. Nos sirve/ lo usamos para trabajar con el protocolo de autenticación por medio de Tokens. !! Ej: JSON Web Token (JWT)

**SPRING ROO:** Usado para automatizar todo. Creaba el CRUD (capas de servicios, repositorio, vistas, controlador) No tuvo éxito, porque a veces tienes lógicas de negocios muy específicas.

**SPRING BOOT:** Depende de Spring Context y Spring Core pero no depende de Spring Web, ni de Spring MVC. Spring Boot lo que hace es que sea todo **autoconfigurable**. No tenemos que crear el archivo Web XML Application Context para la información web del proyecto. Además Spring Boot te configura un servidor tomcat automáticamente (JAR).

Pregunta típica de exámenes sobre Spring Boot:

- **@SpringBootConfiguration:** Configuración básica de Spring Boot
- **@EnableAutoConfiguration:** Configuración que configura automáticamente nuestro servidor y todas las dependencias.
- **@ComponentScan:** Dice que paquetes van a recibir la inyección de dependencias.

**SPRING CLOUD 1:** Está desarrollado junto con Netflix. Nos ofrece cosas para crear microservicios. Ej: Discovery Server. Cada servicio tiene un DNI que nos sirve para el balanceo de carga entre microservicios a través del DNI o directamente de la url. También nos ofrece un cliente HTTP, era como implementar JPA pero en un cliente HTTP y nos ofrece configuración centralizada como config server. Está en desuso, está Spring CLOUD 2.

**SPRING BOOT 2:** Ofrece trabajar con el paradigma Reactivo. Tanto Spring Webflux, como Spring R2DBC y Spring Cloud 2 son reactivos.

**SPRING Webflux:** Parte de Spring Boot 2, necesita también de Spring Web pero no es SPRING WEB, este nos ofrece funcionalidades reactivas. Ejemplo usa router functions, que es una forma de crear funciones de ruta sin necesidad de un controlador. Nos ofrece un nuevo cliente http reactivo y además no bloquea el hilo como en Spring web (Spring Web: 500 peticiones y 500 hilos, entonces se bloquea, spring webflux: con 500 hilos y 1000 peticiones no se va a bloquear los hilos) (Ventaja + importante de Spring Webflux).

**SPRING R2DBC:** Con soporte para MongoDB y BBDD noSQL. Nos ofrece un Spring Data Reactivo (haciendo que no se bloquee el hilo al ser reactivo al usar repositorios y entidades, a diferencia de Spring Data). En cambio, no implementa el ORM de Hibernate!!.

**SPRING CLOUD 2:** Incluye muchas librerías, es un spring cloud reactivo. Incluye la librería config server para tener un servidor de configuración centralizado. También ofrece Discovery Client para crear microservicios y hacer cargas a través de DNI.

HAY MÓDULOS QUE SON DEPENDIENTES DE OTROS COMO SPRING WEB DE SPRING CONTEXT O SPRING CORE Y OTROS QUE SON INDEPENDIENTES COMO SPRING AOP DE SPRING WEB.

---

## SECCIÓN 2 :INTRODUCCIÓN A SPRING

### PREGUNTA DE EXAMEN O ENTREVISTA DE TRABAJO: CUALES SON LAS ANOTACIONES DE SPRING BOOT?

SPRING BOOT tiene 3 anotaciones principales:

- **@SpringBootConfiguration**
  - **@EnableAutoConfiguration**
  - **@ComponentScan**
- **@EnableAutoConfiguration** tiene la propiedad **spring.boot.enableautoconfiguration** es decir, que la aplicación de spring se va a autoconfigurar, autogestionar sola.
- **@ComponentScan** significa que todos los paquetes y las clases van a recibir la inyección de dependencias siempre que estén dentro del paquete raíz (donde está la aplicación de java para ejecutar.).
- 

**Inversión de Control:** Es cuando nosotros delegamos nuestras dependencias/ nuestros objetos a un framework que se encarga de gestionarlos mediante un contenedor, en este caso Spring Container. Es decir si utilizamos una clase necesitamos que forme parte de spring para poder utilizarlo por lo que es necesario ponerle una anotación para que sea un bean de Spring (frijolito aparece en intelliJ) y forme parte del Contenedor de Spring y así poder usarlo en la aplicación. Si no está dentro no podemos usarlo. Usamos para ello las anotaciones **@Service**, **@repository** o **@Controller** o **@Component** (+general).

**Inyección de dependencias:** Lo usamos justo antes de declarar un objeto de otra clase. Para ello usamos la anotación **@Autowired**.

Ej:   **@Autowired**  
      private MyService service;

Otra manera de hacer **inyección de dependencias** es crear un **constructor** e instanciarlo, **el atributo debe ser final!!!!!!**.

```

    @SpringBootApplication
    public class BestTravelApplication implements CommandLineRunner {

        private final MyService service;

        public BestTravelApplication(MyService service) {
            this.service = service;
        }
    }

```

Ej 1:

Ej2:

```

private final HotelRepository hotelRepository;
private final FlyRepository flyRepository;
private final TicketRepository repository;
private final ReservationRepository reservationRepository;
private final TourRepository tourRepository;
private final CustomerRepository customerRepository;

public BestTravelApplication(HotelRepository hotelRepository, FlyRepository flyRepository, TicketRepository repository, ReservationRepository
    this.hotelRepository = hotelRepository;
    this.flyRepository = flyRepository;
    this.repository = repository;
    this.reservationRepository = reservationRepository;
    this.tourRepository = tourRepository;

```

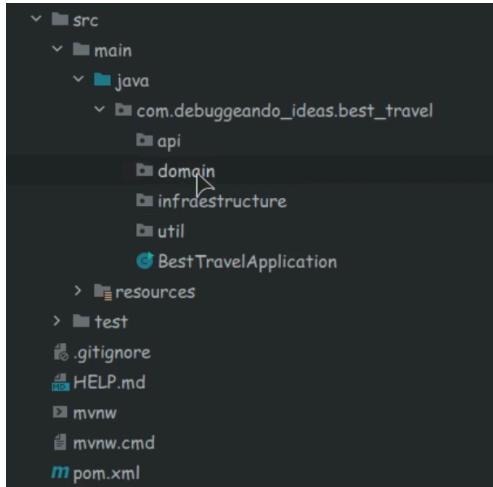
## SECCIÓN 3 :SPRING DATA JPA

-SPRING DATA JPA: Se encarga de los repositorios, trae por defecto la librería de Hibernate.

-Librería de lombok la usamos para los getter y setter y constructores.

Paquetes/carpetas a crear dentro del src del proyecto van a ser:

- **domain**: acceso a datos va a ir dentro del paquete domain, es decir, aquí irán los repositorios y las entidades.
- **infrastructure**: va a tener todos los servicios, irán todas las implementaciones de la capa DAO o repositorio. capa kafka irá aquí también.
- **api**: van a ir todos nuestros controladores y nuestro controller advice .
- **util**: paquete donde va a ir toda la utilería.



La configuración del **dataSource** se hace en el [ApplicationProperties](#):

- Para [spring.datasource](#) hay que incluir:
  - **url** = jdbc:postgresql://localhost:5432/best\_travel  
(jdbc:SGBD://localhost:puerto/nombreDeBBDD)
  - **username**= root
  - **password**= root
  - **hikari.connection-timeout= 20000** :(20 segundos) es un pool de conexiones que lo que hace es cerrar las conexiones que hacemos al hacer modificaciones/inserciones/etc.. en la BBDD, si no lo tenemos la BBDD queda con la conexión abierta, si tenemos muchas conexiones abiertas/sin cerrar, la BBDD cae y no podemos utilizarla. Hikari se encarga de cerrar estas conexiones.
  - **hikari.maximum-pool-size=5** (es el número de conexiones máximas abiertas que puede haber al mismo tiempo a la BBDD)
  - **driver-class-name**=org.postgresql.Driver
- Para Hibernate ([logging.level.org.hibernate](#).):
  - **SQL=DEBUG** (para que nos muestre el SQL de Hibernate en DEBUG)
  - **type.descriptor.sql.BasicBinder=TRACE**

---

## **MAPEANDO ENTIDADES:**

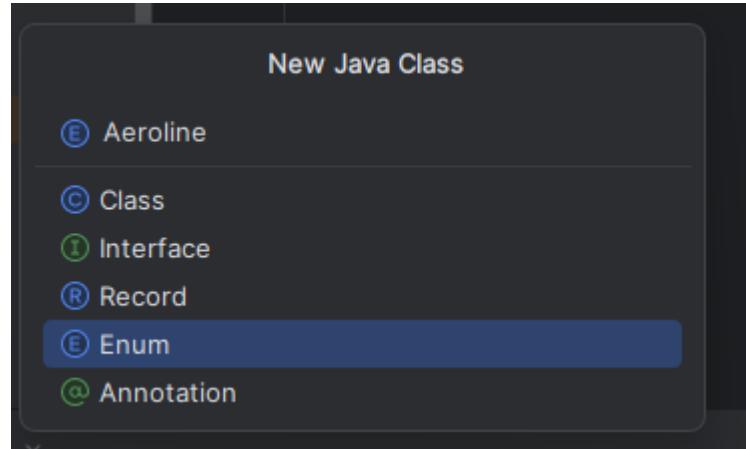
PASOS:

**Importante!!: Las clases se mapean de los menos dependientes a los más dependientes. Los atributos de las relaciones escriben los últimos después de haber generado todas las clases!!!**

1. En la carpeta domain, creamos una **subcarpeta** llamada **entities**.
2. Dentro de entities, creamos una **clase** llamada **FlyEntity** (Entidad Vuelo, el nombre se adapta al proyecto concreto y a su entidad)
3. **implementamos la interfaz Serializable** (opcional!!). Esta significa que todo objeto que implemente esta interfaz va a poder ser serializado a través de la red, es decir, se va a convertir en bytes para que pueda ser transmitido a través de ella, ya sea http o a una BBDD. Ya no es necesario, porque se entiende que todas las entidades van a ser serializables y por tanto lo hace internamente. Por lo que podemos quitarlo si queremos, es OPCIONAL.
4. Le ponemos la **anotación @Entity** para indicar que es una entidad e importamos sus librerías.
  - 4.1. Como nuestra **entidad en la BBDD no se llama igual** le **añadimos** a la notación (**name=fly**)

```
@Entity(name = "fly")
public class FlyEntity implements Serializable { }
```

5. **Declaramos el atributo de clave primaria**. Los atributos siempre se declaran como **privados** y con su **tipo**!! Le ponemos private Long id;
  - 5.1. Añadimos sus anotaciones:
    - 5.1.1. **@Id** (para indicar que atributo es **clave primaria**)
    - 5.1.2. **@GeneratedValue(strategy=GenerationType.IDENTITY)**  
Para indicar que el **atributo es autoincremental**.
6. Declaramos el **resto de atributos**:
  - 6.1. Para tipo de datos de dinero(PRECIO) en **base de datos** deben estar como **float** pero en la entidad, lo mejor es usar el tipo **BigDecimal**(buena práctica, pertenece a la clase java.math)
  - 6.2. Para tipo de datos **ENUM** (enumeración), creamos un **enum** de nombre el atributo (Aeroline (aerolínea)) en el **paquete util**.



Ponemos los **valores del enum**, en nuestro caso, aero\_gold y blue\_sky.

```
Rename usages
public enum AeroLine {aero_gold, blue_sky}
|
```

Declaramos una **nueva variables de tipo AeroLine** en nuestra entidad **y** le añadimos la **anotación @Enumerated y su tipo**, en este caso **EnumType.STRING**.

La anotación **@Column** no es necesaria ponerla en todos los atributos. Podemos ponerla para (ENTRAR/PULSAR DENTRO DE ANOTACIÓN: 2º botón, go to, declaration):

- ❖ Especificar el **máximo de longitud** con:  
    `@Column(length = 20)`
- ❖ Para decir que es **único**:  
    `@Column(unique=true)`
- ❖ Para especificar que es o no **nulo**:  
    `@Column(nullable=true)`
- ❖ Para poner el **nombre** de la columna de la BBDD y que **relacione correctamente** columna con atributo.  
    `@Column(name="")`

- 6.3. Para **tipo de dato UUID**: Un UUID es un **Identificador único Universal** de 32 dígitos. Es lo que usa MongoDB y otras BBDD. Hay que importar paquete para usarlo desde java.util. Abría que incluirle, si es clave primaria, la anotación `@Id`. Este Id se puede generar por nosotros o por la base de datos automáticamente (ambas posibilidades, postgre tiene función para crear UUID) El id de Hotel y Fly si están creados por BBDD, el de Reservation lo creamos nosotros con el UUID.

- 6.4. Tipos de datos de **fechas y horas**.

**6.4.1. Fecha y hora: LocalDateTime**

**6.4.2. Fecha: LocalDate**

7. Uso de **anotaciones Lombok en la clase entidad**, para **getters y setter, constructores** sin argumentos y con argumentos y el **patrón builder**.

7.1. **@NoArgsConstructor**

7.2. **@AllArgsConstructor** (constructor con todos los argumentos)

7.3. **@Data** (incluye getters y setters, `toString` y `EqualsAndHashCode`)

7.4. **@Builder** (patrón de diseño que hace que construyas los objetos con un número de atributos obligatorios y otros opcionales, creas objetos complejos con los atributos que quieras, no necesario todos) Este patrón requiere del **@AllArgsConstructor**.

8. Mapeamos las relaciones entre clases.

8.1. **Identificamos la relación que se da entre entidades:** cual es la entidad de One( 1 ) y cuál la de Many(muchos).Ej: Un vuelo puede tener muchos tickets, pero un ticket solo puede estar asociado a un vuelo.

8.2. **Creamos los atributos tipo la otra entidad** para cada entidad:

8.2.1. **En la clase/entidad Many**(ej: `Ticket(TicketEntity)`) creamos el atributo de tipo la otra entidad (Ej: `FlyEntity(vuelo)`). Ej: `private FlyEntity fly;`

8.2.2. **En la clase/entidad One** (ej:`FlyEntity`) creamos el atributo (`TicketEntity`), ya que son muchos los que tiene asociados, mediante **Set**(Teoría de Conjuntos/modela conjuntos) o mediante **List**.

Ej: `private Set<TicketEntity> tickets;`

8.3. Ponemos las **anotaciones** correspondientes **para** cada lado de **la relación**.

8.3.1. **Para la entidad Many:** Escribimos la anotación `@ManyToOne` y le añadimos la anotación `@JoinColumn` para especificar cual es la columna que une las 2 tablas. Importante: esta se pone en el Many. Le añadimos además el nombre de la columna a la anotación

```
@JoinColumn
```

```
    @ManyToOne
```

```
    @JoinColumn(name = "fly_id")
```

```
    private FlyEntity fly;
```



Además podemos añadirle, si se da el caso, un **nullable = true** o **false**, para cuando tienes a tu entidad relacionada con varias entidades y no pueden darse estas relaciones a la vez en un mismo registro. Ej: un tour puede tener varios tickets o varias reservas pero **no las 2 al mismo tiempo (no por cada registro, si para registros distintos!!**, un registro de tour contiene varias reservas y otro registro de tour contiene varios tickets pero no hay un registro de la entidad tour que contenga tickets y reservas a la vez, para ello especificamos nullable = true).

8.3.2. **Para la entidad One:** Escribimos primero las anotaciones **@ToString.Exclude** y **@EqualsAndHashCode.Exclude** para que no se produzca un bucle en el que el metodo **toString** llama al metodo **toString** de la otra clase y viceversa (bucle infinito de llamadas). Además ponemos la anotación **@OneToMany** y abrimos **parentesis** y le escribimos las siguientes claves/valor:

- ❖ **cascade** =para decir hacer con los datos asociados a una entidad al borrar el registro de la otra entidad. Ej: Si borras un vuelo debes o no borrar los tickets asociados a ese vuelo?. Los valores serían:
  - **CascadeType.REMOVE** : en este caso se borrarán los datos asociados(los tickets) al borrar el registro del vuelo.
  - **CascadeType.PERSIST**: se da cuando al insertar un vuelo guardas también los tickets que van con ese vuelo.
  - **CascadeType.ALL**: Contiene todas las operaciones en cascada que hay.
- ❖ **mappedBy** = Para especificar por quien está mapeado, en este caso sería por la variable **fly**.
- ❖ **fetch** = Usado para especificar cómo se va a cargar el objeto, ej: cuando vayas a cargar el

objeto vuelo, vas a cargar con todos los tickets o sin los tickets. Sus valores pueden ser solo:

- **FetchType.EAGER** : Cargas todo el objeto con los tickets incluidos. En consulta SQL sería: select \* from fly f join ticket t on f.id = t.id;
- **FetchType.LAZY** : Carga perezosa (No cargamos los tickets) En consulta SQL sería: select \* from fly;
- ❖ **orphanRemoval** = Para decir si hay que remover un objeto que se ha quedado sin su clave foranea y por tanto ya no se puede encontrar y no se puede utilizar. Si ponemos **true**, se elimina, si ponemos **false**, se mantiene.

```
@OneToOne(
    cascade = CascadeType.ALL,
    fetch = FetchType.EAGER,
    orphanRemoval = true,
    mappedBy = "fly"
)
```

## CREACIÓN DE REPOSITORIOS:

### 9. Hay 3 tipos de repositorios:

#### 9.1.1. **CrudRepository**: Es el más simple de los 3 y nos ofrece operaciones para el crud.

Modifier and Type	Method	Description
long	<code>count()</code>	Returns the number of entities available.
void	<code>delete(T entity)</code>	Deletes a given entity.
void	<code>deleteAll()</code>	Deletes all entities managed by the repository.
void	<code>deleteAll(Iterable &lt;? extends T&gt; entities)</code>	Deletes the given entities.
void	<code>deleteAllById(Iterable &lt;? extends ID&gt; ids)</code>	Deletes all instances of the type T with the given IDs.
void	<code>deleteById(ID id)</code>	Deletes the entity with the given id.
boolean	<code>existsById(ID id)</code>	Returns whether an entity with the given id exists.
Iterable <T>	<code>findAll()</code>	Returns all instances of the type.
Iterable <T>	<code>findAllById(Iterable &lt;ID&gt; ids)</code>	Returns all instances of the type T with the given IDs.
Optional <T>	<code>findById(ID id)</code>	Retrieves an entity by its id.
<S extends T> S	<code>save(S entity)</code>	Saves a given entity.
<S extends T> Iterable <S>	<code>saveAll(Iterable &lt;S&gt; entities)</code>	Saves all given entities.

9.1.2. **PagingAndSortingRepository**: Nos ofrece un método `findAll` de tipo `Page`.

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method	Description
<code>Page&lt;T&gt;</code>	<code>findAll(Pageable pageable)</code>	Returns a <code>Page</code> of entities meeting the paging restriction provided in the <code>Pageable</code> object.
<code>Iterable &lt;T&gt;</code>	<code>findAll(Sort sort)</code>	Returns all entities sorted by the given options.

9.1.3. **JpaRepository**: Es el que tiene más funcionalidad (métodos) de los 3, pero no siempre vamos a necesitarlo puesto que a veces con un `crudRepository` nos es suficiente. (Depende de lo que necesitemos)

Los repositorios se encargan de gestionar el `EntityManager` de `Hibernate`. El `EntityManager` es el objeto que se encarga de gestionar todas las entidades y de transformarlas a consultas y hacer toda la capa de datos. El `EntityManager` se encarga de transformar la entidad en una consulta legible en `sql` y hacer la operación. Al usar `Hibernate` sin `Spring` hay que trabajar con el `EntityManager` y hacer las operaciones que ofrece pero cuando usamos un repositorio de `Spring`, es `Spring` el que se encarga de gestionar este `EntityManager` como bean propio de `Spring` y nosotros no interactuamos casi nunca directamente con el `EntityManager`.

-Interfaz **CrudRepository**, interfaz genérica en la que entran 2 parámetros( El tipo de la Entidad y el tipo de dato del ID).

-Interfaz **PagingAndSortingRepository** : Tiene los mismos métodos que el `Crud Repository`, más además, tiene:

- **findAll de tipo page**. Es decir, cada pagina será de X objetos y nos va regresando de X en X (Ej: Cada página tiene 10 obj. y nos regresa de 10 en 10.) El objeto `page` es como una lista que va regresando objetos en un rango (De x en X, ej: De 10 en 10).
- **findAll de tipo Iterable**. Ordena los datos por alguna condición personalizada (ej: por orden alfabético).

-**JpaRepository**: Tiene los métodos del `CrudRepository` y los del `PagingAndSortingRepository`, más además otros como:

- `deleteAllByIdInBatch`: Para cuando tenemos que borrar por lotes (másivamente). ect..

A nivel de código **PASOS**:

1. Creamos una carpeta dentro de la carpeta `domain`, llamada `repositories`.
2. Creamos la interfaz de nombre según entidad seguido de `repository`, ej: (`FlyRepository`)

3. extendemos de alguno de los repositorios y con las llaves de diamantes, ponemos como primer parametro el tipo de la entidad(FlyEntity) y como segundo parametro el tipo de dato de la clave primaria (Ej: Long).

```
public interface FlyRepository extends JpaRepository<FlyEntity, Long> { }
```

4. Importante, NO ES NECESARIO PONER UNA ANOTACIÓN @Repository, ya que Spring detecta automáticamente que es un repositorio y lo carga como bean en Spring.
5. EXTRA: Ejecutar el CommandLineRunner para probar que funciona:
  - Para el ejemplo prueba, inyectamos los 2 repositorios mediante la anotación @Autowired seguida de las variables en privado en la **aplicación principal**.

```
@Autowired
private HotelRepository hotelRepository;
@Autowired
private FlyRepository flyRepository;
```

- Añadimos como **implements** el **CommandLineRunner** en la firma del método.
- Implementamos/**sobreescrivimos** el método **run** mediante pruebas para ver si funciona el repositorio:

```
@Override
public void run(String... args) throws Exception {
    var fly = flyRepository.findById(15L).get();
}
```

**Creamos nuestras variables** que obtengan apartir de cada repositorio el registro concreto que queremos.

Usamos el **get()** porque el método **findById** nos devuelve un **Optional**, entonces necesitamos el **get** para acceder/conseguir el objeto.

- Usamos la anotación **@Slf4j** de Lombok para el logging y dentro del método run usamos el método **log.info**, que contiene como parámetro la variable creada convertida(conversión de tipo) a String.

```
@SpringBootApplication
@Slf4j
public class BestTravelApplication implements CommandLineRunner {
```

```

@Override
public void run(String... args) throws Exception {
    var fly = flyRepository.findById(15L).get();
    var hotel = hotelRepository.findById(7L).get();

    log.info(String.valueOf(fly));
    log.info(String.valueOf(fly));
}

```

### CREACIÓN DE CONSULTAS JPQL:

Hay 3 tipos de consultas/querys implementaciones:

- **Query Nativo:** Consulta tal cual la escribimos en pqql.
- **Query JPQL:** Lenguaje de consultas de **Jpa**
- **Query de Hibernate(HPQL):** Lenguaje de consultas con **Hibernate**.

Es preferible usar **JPQL** con respecto al resto.

**Motivo:** Podemos usar Jpa o Hibernate, pero Hibernate nos obliga a una implementación concreta (puede volverse de pago y tienes que migrar los datos) con Jpa esto no pasa, es una implementación abstracta. Por ello, por defecto, se utiliza JPQL y no HPQL.

### PASOS:

1. En el repositorio/dentro, creamos un set o list del tipo de entidad concreto y le damos nombre a la consulta.
2. Incluimos la anotación **@Query**, que te permite poner entre **paréntesis** un **string**, en concreto el nuestro será una **consulta de jpql** que se realizará **sobre la propia entidad** y no sobre la BBDD por tanto, la tabla tendrá el **nombre de la variable** del tipo de **entidad** concreto.
3. Incluimos en nuestra **anotación** y en el **método** los **parámetros** que van a entrar.
  - a. Para la **anotación @Query** deberemos especificar que es una **variable** mediante el **uso de los dos puntos (:)**
  - b. Para el **método** tenemos **2 opciones**:
    - i. **Renombrando la variable de entrada:** Usar una anotación **@Param** con el nombre que le vamos a dar a la variable y que debe coincidir con el nombre de la variable que usemos en la anotación **@Query** y, además ponemos el tipo y el nombre de la

variable (este nombre **se renombrará** al de la anotación @param). NO RECOMENDADO.

```
@Query("select f from fly f where f.price < :price")
Set<FlyEntity> selectLessPrice(@Param("price") BigDecimal name);
```

- ii. **Sin renombrar la variable de entrada.** La variable de entrada/parametro **debe coincidir en nombre**, directamente, con la **variable** que se use en el @Query.

Ejs:

```
@Query("select f from fly f where f.price < :price")
Set<FlyEntity> selectLessPrice(BigDecimal price);
```

```
@Query("select f from fly f where f.price between :min and :max")
Set<FlyEntity> selectBetweenPrice(BigDecimal min, BigDecimal max);
```

```
@Query("select f from fly f where f.originName = :origin and f.destinyName = :destiny")
Set<FlyEntity> selectOriginDestiny(String origin, String destiny);
```

Si quisiéramos hacer una consulta nativa en vez de una con jpql, deberíamos añadir en el @Query después de la consulta nativequery = true. NO ES RECOMENDABLE USAR CONSULTA NATIVA, MEJOR CON JPQL (por defecto nativequery = false).

```
@Query(value = "select f from fly f where f.price < :price", nativeQuery = true)
Set<FlyEntity> selectLessPrice(BigDecimal price);
```

**-PARA PROBAR nuestras CONSULTAS:** hacemos en nuestro **método run** (CommandLineRunner) **referencia al propio objeto** en la clase aplicación, **al repositorio concreto**, a la **consulta/método concreta** y como **parametro** de entrada **convertimos**, para asegurarnos, al tipo de dato concreto el valor que sea y hacemos un **foreach** sobre el set/**conjunto o lista** resultado y nos lo **imprima**.

Para imprimir hay **2 opciones**:

1. Usar una expresión **lambda**

```
this.flyRepository.selectLessPrice(BigDecimal.valueOf(20)).forEach(System.out::println);
```

2. Hacer función **flecha**.

```
this.flyRepository.selectLessPrice(BigDecimal.valueOf(20)).forEach(f->System.out.println(f));
```

Ej2:

```
this.flyRepository.selectBetweenPrice(BigDecimal.valueOf(10), BigDecimal.valueOf(15)).forEach(System.out::println);
```

Ej3:

```
this.flyRepository.selectOriginDestiny( origin: "Grecia", destiny: "Mexico").forEach(System.out::println);
```

### Consultas con JOIN:

Hay varias formas de realizarlas:

1. Usando el `findById` del propio repositorio ya incluido para la consulta join "select":
  - a. Para ello obtenemos el registro/objeto y, posteriormente, conseguimos a través de **get del atributo de la relación** (ej:`getTickets`) y de un `forEach` que imprima todos los registros de la otra tabla asociados a ese registro concreto. Conseguimos con ello un JOIN de las 2 tablas en función del valor del campo/atributo de la relación que tienen en común (clave foranea).

```
var fly = flyRepository.findById(1L).get();

fly.getTickets().forEach(ticket -> System.out.println(t));
```

- b. Mediante una `@Query` en el repositorio donde está la clave foránea: Usamos un `Optional` de tipo Entidad y que entre como parámetro el id/clave primaria y foranea de la relación. En nuestro `@Query`, hacemos un `join fetch` (unión/busqueda) a partir del atributo/campo que es clave foránea y le damos un sobrenombre(ej: `t`) y hacemos un `where` (condicional) para que nos busque por el id que le vamos a pasar como parametro.

```
@Query("select f from fly f join fetch f.tickets t where t.id = :id")
Optional<FlyEntity> findByTicketId(UUID id);
```

Probamos de la siguiente manera el método (también se puede probar como se mencionó anteriormente):

```
var fly = flyRepository.findByTicketId(UUID.fromString("12345678-1234-5678-2236-567812345678"));

System.out.println(fly);
```

## 2. Web: Palabras Clave para Crear Queries:

<https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/jpa.repositories.html>

Keyword	Sample	JPAQL snippet
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Is_Equals	findByFirstname, findByFirstnameIs, findByFirstnameEquals	... where x.firstname = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	... where x.age <= ?1
Greater Than	findByAgeGreaterThan	... where x.age > ?1
Greater Than Equal	findByAgeGreaterThanEqual	... where x.age >= ?1
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1
IsNull	findByAgeIsNull	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	... where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	... where x.age not in ?1
True	findByActiveTrue()	... where x.active = true
False	findByActiveFalse()	... where x.active = false

Podemos usar esas tipos de queries, cambiando el nombre de la variable para adaptarlo al caso e incluimos sus parámetros de entrada. Ej: En vez de findByAgeLessThan

```
Set<HotelEntity> findByPriceLessThan(BigDecimal price);
```

Para probar el código usamos el método a partir del repositorio, incluimos su parámetro y hacemos un forEach puesto que en este caso, nos puede devolver varios resultados (lista de hoteles donde el precio es menor que el valor dado por parámetro):

```
hotelRepository.findByPriceLessThan(BigDecimal.valueOf(100)).forEach(System.out::println);
```

Otros ejemplos:

En el repositorio:

```
• Set<HotelEntity> findByPriceIsBetween(BigDecimal min, BigDecimal max);
  Set<HotelEntity> findByRatingGreaterThanOrEqual(Integer rating);
```

En la aplicación principal:

```
hotelRepository.findByPriceIsBetween(BigDecimal.valueOf(100),  
    BigDecimal.valueOf(150)).forEach(System.out::println);  
  
hotelRepository.findByRatingGreater Than(3).forEach(System.out::println);
```

**Join** mediante el **Lenguaje inclusivo de Spring**(evita uso del **@Query**):

1. Existen además palabras clave del lenguaje de Spring con la que **no** tenemos porqué hacer el **@Query** y podemos adaptarla a nuestra entidad concreta.
2. Estructura: **findBy** es **palabra clave** para crear consultas, posteriormente **añadimos** la **entidad** y después de esta el **campo** por el que vamos a **identificar**. Añadimos por último los **parámetros** del método de consulta.

```
Optional<HotelEntity> findByReservationId(UUID id);
```

3. Esta forma es lo mismo que poner:

```
@Query("select f from fly f join fetch f.tickets t where t.id = :id")  
Optional<FlyEntity> findByTicketId(UUID id);
```

(cambia la entidad **fly** con **hotelEntity** solo, ejemplo =). Con ello **evitamos** el **@Query** y tener que escribir la **consulta completa**.

PROBANDO ENTIDADES TOUR, VUELOS, RESERVAS, HOTELES, CLIENTES Y TICKETS.

1. Conseguimos y guardamos en variables los objetos/registros de las **entidades menos dependientes usando el **findById**** mediante el **repositorio** concreto, en nuestro caso serían Vuelos(Fly), hoteles y Clientes(customer). Importante, cuando usemos el **findById** debemos ponerle al final un **.orElseThrow()** para **controlar la excepción** en caso de **no encontrar el registro** en la BBDD.
2. A partir de estas entidades **generamos el resto de entidades usando** el patrón de diseño **builder()**,(importante, usamos las propias **entidades** para acceder al **patrón builder** y **no los repositorios**) damos valor a los atributos de relación como el **valor** de las **variables/objetos** creadas **anteriormente** para que nos los **relacione con las otras entidades e** incluimos al final un **.build()** para que nos **construya** el objeto.
3. Podemos además **mostrar el valor de los objetos** para comprobar mediante un **log.info**.

```

var customer = customerRepository.findById("GOTW771012HMRGR087").orElseThrow();
log.info("Client name: " + customer.getFullName());

var fly2 = flyRepository.findById(11L).orElseThrow();
log.info("Vuelo: " + fly2.getOriginName() + "-" + fly2.getDestinyName());

var hotel2 = hotelRepository.findById(3L).orElseThrow();
log.info("Hotel: " + hotel2.getName());

var tour = TourEntity.builder().customer(customer).build();

var ticket = TicketEntity.builder()
    .id(UUID.randomUUID())
    .price(fly2.getPrice().multiply(BigDecimal.TEN))
    .arrivalDate(LocalDate.now())
    .departureDate(LocalDate.now())
    .purchaseDate(LocalDate.now())
    .customer(customer)
    .tour(tour)
    .fly(fly2)
    .build();

var reservation = ReservationEntity.builder()
    .id(UUID.randomUUID())
    .dateTimeReservation(LocalDate.now())
    .dateEnd(LocalDate.now().plusDays(daysToAdd: 2))
    .dateStart(LocalDate.now().plusDays(daysToAdd: 1))
    .hotel(hotel2)
    .customer(customer)
    .tour(tour)
    .totalDays(1)
    .price(hotel2.getPrice().multiply(BigDecimal.TEN))
    .build();

```

## MAPEANDO RELACIONES INVERSAS:

-Cuando creamos un objeto/registro en una entidad que es One y tiene Many en su relación, al crear ese objeto no se relaciona automáticamente con el Set de la otra entidad. Para ello tenemos que crearle unos métodos específicos que nos permitan relacionar ese registro con otro concreto de la otra entidad, borrar la relación o actualizarla. Ej: Creamos un tour, ese tour no está asociado a ningún Set de tickets (a ningún ticket), para ello tendremos 3 operaciones/métodos que nos permitirán: añadir un ticket a ese Set de tickets del tour y relacionarlo (por lo tanto ese ticket llevará el id de ese tour correspondiente), eliminar del Set de tickets de ese tour el ticket relacionado con ese tour que tenga el id concreto (el ticket ya no tiene el id del tour, es decir, ya no pertenece al set de Tour) o actualizar toda la lista o Set de tickets para que pertenezca o se relacionen con este tour. También hemos añadido un

Objects.isNull, para que en caso de que el Set sea nulo, no nos de un NullPointerException. Para ello comprobamos si el Set de reservas asociadas al tour es nulo (no hay) y si es así lo creamos tanto para añadir ticket como para borrarlos, no es necesario para actualizar.

Ej: 1

```
public void addTicket(TicketEntity ticket) {
    if ((Objects.isNull(this.tickets))) this.tickets = new HashSet<>();
    this.tickets.add(ticket);
}

public void removeTicket(UUID id) {
    if ((Objects.isNull(this.tickets))) this.tickets = new HashSet<>();
    this.tickets.removeIf(ticket -> ticket.getId().equals(id));
}

public void updateTickets() {
    this.tickets.forEach(ticket -> ticket.setTour(this));
}
```

Ej: 2

```
public void addReservation(ReservationEntity reservation) {
    if (Objects.isNull(this.reservations)) this.reservations = new HashSet<>();
    this.reservations.add(reservation);
}

public void removeReservation(UUID idReservation) {
    if (Objects.isNull(this.reservations)) this.reservations = new HashSet<>();
    this.reservations.removeIf(r -> r.getId().equals(idReservation));
}

public void updateReservations() {
    this.reservations.forEach(r -> r.setTour(this));
}
```

Los usamos en la aplicación principal de la siguiente manera:

```
tour.addReservation(reservation); //añadir reservas al set
tour.updateReservations(); //para hacer la relación inversa

tour.addTicket(ticket);
tour.updateTickets();
this.tourRepository.save(tour);
```

Comprobamos en el DBeaver con consulta:

```
select * from tour
join reservation r on tour.id = r.tour_id
join hotel h on h.id = r.hotel_id
join customer c on c.dni = r.customer_id;
```

Para borrar los tickets, no usaremos el método creado anteriormente (este se usará para otra cosa más adelante), sino que lo que haremos será **cambiar** el atributo **relación** de la entidad **con los sets, de EAGER a LAZY**, ya que **EAGER te impide borrar** al estar relacionado (Query: select \* from tour t join ticket t on ....) para ello pondremos **LAZY (Query no relacionada)** y **borraremos los registros asociados mediante el orphanRemoval = true**, es decir, borramos el tour concreto y al quedar tickets y reservas asociadas a ese tour borrado como huérfanas se borran también dichos tickets y reservas.

También podemos usar las **anotaciones @PreRemove, @PrePersist, @PreUpdate**. Estas anotaciones deberán ser:

- Únicas, solo puede haber **una por clase** (1 @PreRemove, 1 @PreUpdate y 1 @PrePersist)
- **No pueden devolver**, deben **ser void ni** pueden **llevar parámetros de entrada**.

---

## SECCIÓN 4 :SPRING WEB

1. Incluimos en el .pom, el paquete spring starter web (spring web)

#### **PAQUETE MODELS (DENTRO DE API):**

2. Creamos el paquete **models** con 2 subpaquetes: **responses** y **request**.
3. **PAQUETE REQUEST:** Creamos la clase **TicketRequest** en el paquete **request** y le incluimos las **anotaciones de constructores, setters/getters** de las entidades (**@AllArgsConstructor, @NoArgsConstructor, @Data, @Builder**) e implementamos la interfaz **Serializable**.
  - a. **Creamos los atributos** que necesitamos que nos dé el cliente. Si los datos podemos conseguirlos a partir de otros que ya tenemos o podemos generarlos nosotros, no creamos ese atributo. Solo para datos que **nos falten y no puedan ser generados** por nosotros y **necesiten ser pedidos al cliente**.

```
    @NoArgsConstructor  
    @AllArgsConstructor  
    @Data  
    @Builder  
    public class TicketRequest implements Serializable {  
  
        private String idClient;  
        private Long idFly;  
    }
```

Ej: En este caso, solo vamos a necesitar el id del cliente y el id del vuelo. Todo lo demás o es autogenerable o se puede sacar a partir de los datos del cliente y del vuelo.

4. **PAQUETE RESPONSES:** Creamos la clase **TicketResponse** en el paquete Response.
  - a. le **incluimos las anotaciones y el serializable** de la clase anterior.
  - b. Le incluimos los **atributos de la entidad** y para el **atributo fly de la relación** lo tomamos de **FlyResponse**, como **FlyResponse** no existe lo **Creamos** y le añadimos anotaciones, serializable y atributos de FlyEntity.

5. **TEORÍA:** Carpetas/paquetes Request y Response son nuestros DTOs, vamos a transformar nuestras respuestas en un objeto Response y nuestros objetos Request (peticiones) en un objeto que pueda persistir en nuestra Base de Datos.

```

6. FlyResponse.java
15 import lombok.NoArgsConstructor;
16 import lombok.AllArgsConstructor;
17 import lombok.Data;
18 import lombok.Builder;
19 public class FlyResponse implements Serializable {
20     private Long id;
21     private Double originLat;
22     private Double originLng;
23     private Double destinyLat;
24     private Double destinyLng;
25     private String originName;
26     private String destinyName;
27     private BigDecimal price;
28     private AeroLine aeroline;
29 }
30 }

TicketRequest.java
3 import lombok.NoArgsConstructor;
4 import lombok.Builder;
5 import lombok.Data;
6 import lombok.NoArgsConstructor;
7
8 import java.io.Serializable;
9
10 @NoArgsConstructor no usages
11 @AllArgsConstructor
12 @Data
13 @Builder
14 public class TicketRequest implements Serializable {
15     private String idClient;
16     private Long idFly;
17 }
18

TicketResponse.java
1 / import lombok.NoArgsConstructor;
2 import lombok.AllArgsConstructor;
3 import lombok.Data;
4 import lombok.Builder;
5 import java.io.Serializable;
6 import java.math.BigDecimal;
7 import java.time.LocalDate;
8 import java.util.UUID;
9
10 @NoArgsConstructor no usages
11 @AllArgsConstructor
12 @Data
13 @Builder
14 public class TicketResponse implements Serializable {
15     private UUID id;
16     private LocalDate departureDate;
17     private LocalDate arrivalDate;
18     private LocalDate purchaseDate;
19     private BigDecimal price;
20     private FlyResponse fly;
21 }
22

```

## PAQUETE INFRAESTRUCTURE:

### SUBPAQUETE ABSTRACT\_SERVICE:

1. **Creamos** una interfaz llamada **CrudService** (Se utilizará para Fly, Hotel y Tour) mientras que para las entidades Tickets y Reservations utilizaremos Catálogos(solo consultas).
2. Entran **3 parámetros usando simbolos de diamante**, que serán: **RQ, RS, ID** (simbolizan request, response y id, puedes denominarlo con el nombre que quieras). Son parametros genéricos, es decir, el id puede ser y será de distinto tipo (tener en cuenta que entidades su id puede ser de distinto tipo).
3. Creamos los **métodos** de la **interfaz CrudService (general)**:
  - a. Para el **create**, **devuelve RS**(una respuesta y **entra** como parametro un **request** (una petición))
  - b. Para el **read** (lectura/get), **devuelve RS** (respuesta) y **entra** como parámetro un **id**(identificador para buscar y leer)
  - c. Para el **update** (actualizar): **devuelve un RS** y **entra** como parámetro un **RQ** (una petición) y un **id** (identificador).
  - d. Para el **delete**(eliminar): **no devuelve nada (void)** y **entra** como parametro un **id** (identificador).

```

public interface CrudService<RQ, RS, ID> {

    RS create(RQ request);

    RS read(ID id);

    RS update(RQ request, ID id);      []

    void delete(ID id);

}

```

4. Creamos una **interfaz** para las entidades en las que se va a hacer el crud y extendemos del CrudService (herencia entre interfaces) y le incluimos sus parámetros específicos para esa entidad concreta. Ej: **ITicketService**.

```

public interface ITicketService extends CrudService<TicketRequest, TicketResponse, UUID> {
}

```

EXTRA: Creación

#### **SUBPAQUETE SERVICE:**

1. Creamos una **carpeta en infraestructure** llamada **service**.
2. Creamos la **clase TicketService** para implementar los servicios.
3. Le **implementamos la interfaz ITicketService y todos sus métodos no implementados automáticamente!!**
4. Le añadimos la **notación @Transaccional** (De Spring!!, no usar la de Jakarta) para nuestras transacciones (Operaciones a la Base de datos en las que se puede hacer un commit o un rollback).
5. Le añadimos la anotación **@Service** para cargarlo al contenedor de Spring.
6. Añadimos notación **@Slf4j** (para logging, log.info).
7. Nos traemos nuestros **repositorios** y los **cargamos como variables**. **Generamos un constructor** en vez de usar anotación Autowired(no recomendada actualmente/ en desuso o deprecada). Con segundo botón nos crea automáticamente el constructor o **mediante** la anotación **@AllArgsConstructor** (Recomendada esta última).

```

@Transational
@Service
@Slf4j
@AllArgsConstructor
public class TicketService implements ITicketService {

    private final FlyRepository flyRepository;
    private final CustomerRepository customerRepository;
    private final TicketRepository ticketRepository;

    @Override
    public TicketResponse create(TicketRequest request) {
        return null;
    }
}

```

8. Implementamos los métodos pero antes necesitamos **crear un método**, que será **usado en los 3 métodos del crud, para que nos devuelva un TicketResponse**. Para ello:
  - a. creamos variable response y creamos el objeto new TicketResponse();
  - b. Usamos la librería de Spring **BeanUtils.copyProperties()**; Esta librería se encarga de **emparejar** el id del TicketEntity con el id del TicketResponse y de emparejar de igual manera con el resto de **atributos**. Va a **hacer el mapeo** de asignarle el **valor** de las variables (si en una entidad cambia en la otra también por el mismo valor). (Cambiemos para usar esta librería, a LocalDateTime). **Como parámetros** en esta librería ponemos **entity y response**, para que todos los **datos de la entidad** nos los matchee (**empareje**) **con los de response**.
  - c. Hacemos **lo mismo** pero **con el FlyResponse** y despues se lo añadimos a nuestra respuesta de tipo ticketResponse, puesto que esta lleva un atributo de tipo FlyResponse.

```

private TicketResponse entityToResponse(TicketEntity entity) {
    var response = new TicketResponse();
    BeanUtils.copyProperties(entity, response);
    var flyResponse = new FlyResponse();
    BeanUtils.copyProperties(entity.getFly(), flyResponse);
    response.setFly(flyResponse);
    return response;
}

```

9. **Implementamos los métodos:**
  - a. Método **CREATE (inserciones)**:

- i. Creamos una **variable** que **guarda** el **vuelo** concreto a **partir** del **id (request)** mandado por el cliente (**findById(request.getIdFly())**). Importante **usar** el **orElseThrow()** en caso de no encontrar el vuelo.
- ii. **Creamos** un **customer** apartir, **igual** que el vuelo del request mandado por el cliente (**findById(request.getIdCustomer())**), y añadimos su **orElseThrow**.
- iii. Una vez creadas las variables de tipo otra entidad que están dentro de nuestro ticket, pasamos a **crear** nuestro **ticket concreto usando** una **variables**, la propia **entidad** y el patrón **builder**. Los **valores** para **fly** y para **customer** serán las **variables** creadas **anteriormente**.

```
@Override
public TicketResponse create(TicketRequest request) {
    var fly = flyRepository.findById(request.getIdFly()).orElseThrow();
    var customer = customerRepository.findById(request.getIdClient()).orElseThrow();

    var ticketToPersist = TicketEntity.builder()
        .id(UUID.randomUUID())
        .fly(fly)
        .customer(customer)
        .price(fly.getPrice().multiply(BigDecimal.valueOf(0.25)))
        .purchaseDate(LocalDate.now())
        .arrivalDate(LocalDateTime.now())
        .departureDate(LocalDateTime.now())
        .build();
```

- iv. Creamos la variable que guarda el objeto en la Base de Datos y la persiste.
- v. Hacemos log para visualizar ese guardado. Las llaves {} se sustituyen por el 2º parámetro.
- vi. Devolvemos la variable que guarda el objeto mediante el método creado para devolver tipos response.

```
var ticketPersisted = this.ticketRepository.save(ticketToPersist);

log.info("Ticket saved with id: {}", ticketPersisted.getId());
TicketEntity entity
```

return this.entityToResponse(ticketPersisted);

- b. **Método READ (lectura/get):**

- i. Creamos una variable que contenga nuestro registro concreto, buscando por el id que entra como parámetro en el método.
- ii. Devolvemos un tipo response en el que entra por parámetro la variable anterior para que nos devuelva el mismo registro pero transformado a response de ese tipo.

```
@Override
public TicketResponse read(UUID id) {
    var ticketFromDB = this.ticketRepository.findById(id).orElseThrow();
    return this.entityToResponse(ticketFromDB);
}
```

c. **Método UPDATE (actualización/PUT):**

- i. Guardamos en una variable el ticket buscado usando el repositorio, apartir del id que entra como parámetro.
- ii. Guardamos en otra variable el vuelo, sacado apartir del id del vuelo que está en el objeto request que nos entra como parámetro.
- iii. Actualizamos los valores que nos interesen (variable con el registro del vuelo se lo ponemos al idFly del ticket(variable relación))
- iv. Creamos otra variable que actualiza usando el repositorio los datos modificado de esa variable.
- v. Hacemos un log para mostrar que todo ha ido bien.
- vi. Devolvemos un Response de tipo la variable que actualiza .

```
@Override
public TicketResponse update(TicketRequest request, UUID id) {
    var ticketToUpdate = ticketRepository.findById(id).orElseThrow();
    var fly = flyRepository.findById(request.getIdFly()).orElseThrow();

    ticketToUpdate.setFly(fly);
    ticketToUpdate.setPrice(BigDecimal.valueOf(0.25));
    ticketToUpdate.setDepartureDate(LocalDateTime.now());
    ticketToUpdate.setArrivalDate(LocalDateTime.now());

    var ticketUpdated = this.ticketRepository.save(ticketToUpdate);

    log.info("Ticket updated with id {}", ticketUpdated.getId());

    return this.entityToResponse(ticketToUpdate);
}
```

d. **Método DELETE (eliminación):**

- i. Buscamos el registro apartir del id usando el repositorio y lo guardamos en una variable.
- ii. Eliminamos el registro usando el repositorio.

```
@Override  
public void delete(UUID id) {  
    var ticketToDelete = ticketRepository.findById(id).orElseThrow();  
    this.ticketRepository.delete(ticketToDelete);  
}
```

**Creamos el controlador:**

1. En el **applications.properties**, le ponemos el **path del proyecto**, para ello

```
server.servlet.context-path=/best_travel  
escribimos:
```

2. **Creamos el controlador** TicketController y le **añadimos la anotación @RestController** (si sale en rojo (no lo reconoce), le damos a maven y a reload all maven projects) y añadimos su librería.  
La diferencia entre las **anotaciones @RestController y @Controller**, es que **RestController** se usa para **servicios Rest** y **Controller** cuando usamos una **aplicación MVC** que va a tener **vistas y JSP integrados**.
3. Le añadimos la **anotación @RequestMapping** junto con el **atributo path** y el **valor** concreto “ticket”.
4. Le añadimos anotación de constructores **@AllArgsConstructor**

```
@RestController  
 @RequestMapping(path = "/ticket")  
 @AllArgsConstructor  
 public class TicketController {
```

5. **Creamos una variable final de tipo Interfaz Servicio (Ej: ITicketService)**, se usa/inyecta la interfaz, no la clase en concreto, de eso se encarga Spring en instanciar las clases.
6. **Creamos el endpoint:**

- a. Para las **inserciones(POST)**:

Nos va a **devolver un ResponseEntity de tipo TicketResponse** y **entra** como parámetro un **TicketRequest** llamado **request** y le incluimos la **anotación @RequestBody** (para incluir el objeto entero y no un parámetro solo)  
Le incluimos la anotación **@PostMapping**

-Devuelve: **return**

**ResponseEntity.ok(ticketService.create(request))**. ok es el que tiene estatus 200 de HTTP.

```
public class TicketController {  
  
    private final ITicketService ticketService;  
  
    @PostMapping()  
    public ResponseEntity<TicketResponse> post(@RequestBody TicketRequest request) {  
        return ResponseEntity.ok(ticketService.create(request));  
    }  
}
```

b. Para las lecturas/**consultas** de tipo **GET**:

- i. **Devolvemos un ResponseEntity del tipo entidad y entra un @PathVariable de tipo UUID o del tipo que sea como parámetro.**  
Devuelve un ResponseEntity cuando el estatus es 200 de ticketService (el servicio) del método de lectura con el id pasado por parámetro. Importante poner la anotación **@GetMapping** con su parámetro (**path="{id}"**)

```
@GetMapping(path = @v "{id}")  
public ResponseEntity<TicketResponse> get(@PathVariable UUID id) {  
    return ResponseEntity.ok(ticketService.read(id));  
}
```

- ii. Si la **variable** del **@PathVariable** no lleva el mismo nombre que la del **@GetMapping** se puede especificar para conectarlo en el **(@PathVariable(path="nombrePuestoEnGetMapping"))** UUID id).

```
@GetMapping(path = @v "{uuid}")  
public ResponseEntity<TicketResponse> get(@PathVariable(name = "uuid") UUID id) {  
    return ResponseEntity.ok(ticketService.read(id));  
}
```

c. Para las **actualizaciones** de tipo **PUT**:

- i. Entrará, como diferencia de los otros métodos, un request (@ResquestBody) y un id (@PathVariable)). Nos devolverá un ResponseEntity.ok(this.ticketService.update(request, id)); Su anotación será **@PutMapping(path={id})** entrará el parametro id.

```
@PutMapping(path = @v "{id}")  
public ResponseEntity<TicketResponse> put(@PathVariable UUID id, @RequestBody TicketRequest request) {  
    return ResponseEntity.ok(this.ticketService.update(request, id));  
}
```

d. Para las **actualizaciones** de tipo **DELETE**:

El **método devuelve un ResponseEntity** de tipo **void** y **entra** por parámetro un **id**, se hace el servicio de eliminación y devuelve en un return un ResponseEntity sin contenido y construido. Debemos modificar los atributos relaciones ManyToOne hay que cambiarles, para poder borrarlo, su fetch a LAZY.

```
@DeleteMapping(path = "/{id}")
public ResponseEntity<Void> delete(@PathVariable UUID id) {
    this.ticketService.delete(id);
    return ResponseEntity.noContent().build();
}
```

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "customer_id")
private CustomerEntity customer;
```

### PROBANDO CON POSTMAN LOS ENDPOINT:

1. Creamos una nueva colección y le damos nombre
2. Ponemos la petición con la url y el método (ej: POST)
3. Ponemos el JSON con los datos de entrada, si es necesario, en el body (raw->json).
4. Enviamos la petición
5. Comprobamos que los datos, según sea, se han insertado borrado o actualizado en la BBDD con el DBeaver.

---

**Script en Postman para no tener que buscar el id u otros parametros** de la entidad a la hora de hacer pruebas:

1. Le damos a la pestaña test de nuestra petición concreta(a la der. del body).
2. Creamos una variable llamada response que guardará el json del cuerpo de la respuesta (el ultimo dado)
3. Damos doble click en la colección, vamos a donde dice Variables y creamos la variable id\_ticket sin ningún valor por defecto(resto de campo en blanco).
4. Volvemos a test en nuestra petición y usamos del objeto pm su colección de variables y añadimos nuestra variable id\_ticket y nuestro id de la variable response para que nos la conecte.
5. Para ver si la variable tiene el valor asignado correctamente usamos un console.log.

```

1 var response = JSON.parse(responseBody);
2 pm.collectionVariables.set("id_ticket", response.id);
3 console.log(pm.collectionVariables.get("id_ticket"))

```

- Para usar esa variable ya asignada ponemos **dobles corchetes** y el nombre de la variable.

GET | localhost:8080/best\_travel/ticket/{{id\_ticket}}

## AÑADIENDO MÉTODOS/FUNCIONES EXTRAS:

### EN LOS SERVICE APARTE DEL CRUD.

En este ejemplo: Estamos creando una función extra para conocer el precio de un ticket a partir del vuelo (El precio de un ticket será el precio del vuelo + un 0,25%)

- En nuestra Interfaz servicio (Ej: ITicketService) creamos un método sin implementar, e implementamos en nuestro servicio (Ej: TicketService).

```

public interface ITicketService extends CrudService<TicketRequest, TicketResponse, UUID> {
    ...
    BigDecimal findPrice(Long idFly);
}

```

- Para implementarlo, corregimos el error en el servicio(no están implementado el método de la interfaz) y nos lo crea automáticamente. Implementamos el método concreto. (Hemos creado una constante llamada charger\_price\_percentage para aumentar ese 0,25 % al precio del vuelo.

```

private static final BigDecimal charger_price_percentage = BigDecimal.valueOf(0.25);

```

```

@Override
public BigDecimal findPrice(Long idFly) {
    var fly = this.flyRepository.findById(idFly).orElseThrow();
    return fly.getPrice().multiply(charger_price_percentage);
}

```

### EN EL CONTROLADOR.

- Copiamos el método GetMapping, al ser un tipo get (de consulta) en nuestro ejemplo. Entra un RequestParam al ser un parámetro de un objeto/registro

que ya tenemos. Usamos un Map con un string y el precio ya conseguido para que nos lo devuelva.

```
@GetMapping
public ResponseEntity<Map<String, BigDecimal>> getFlyPrice(@RequestParam Long flyId) {
    return ResponseEntity.ok(Collections.singletonMap("flyPrice", this.ticketService.findPrice(flyId)));
}
```

## EN POSTMAN PARA COMPROBAR:

### 2. Verifica la URL en Postman

Basado en tu controlador, aquí están las rutas que deberías probar en Postman:

- Crear ticket (POST): `POST http://localhost:8080/best\_travel/ticket`
- Obtener ticket por ID (GET): `GET http://localhost:8080/best\_travel/ticket/{id}`
- Actualizar ticket (PUT): `PUT http://localhost:8080/best\_travel/ticket/{id}`
- Eliminar ticket (DELETE): `DELETE http://localhost:8080/best\_travel/ticket/{id}`
- Obtener precio del vuelo (GET): `GET http://localhost:8080/best\_travel/ticket?flyId={flyId}`

POST localhost:8080/best\_travel/ticket

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none  form-data  x-www-form-urlencoded  raw

```
1 {  
2   "idClient": "BBMB771012HMCRR022",  
3   "idFly": 4  
4 }
```

GET localhost:8080/best\_travel/ticket/22345678-1234-5678-3235-567812345678

Params Authorization Headers (6) Body Pre-request Script Tests Settings

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL  JSON

```
1 { }
```

En nuestro caso: `localhost:8080/best_travel/ticket?flyId=4` (Recuerda, sin el / barra final en ticket) e igual para el método POST (sin barra final), sino no funciona.

---

### Modificación de fechas para que se muestren en un formato determinado:

Para ello usamos la **anotación @JsonFormat** (librería JsonFormat) e **incluimos el patrón** en el que queremos que se muestre/ vea. **Solo** se pone **en el Response** (la respuesta), puesto que **solo** es para **mostrarlo** al consultarse en ese formato. Como una conversión para la muestra.

```
@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd HH:mm")
private LocalDateTime departureDate;

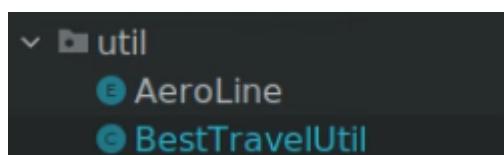
@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd HH:mm")
private LocalDateTime arrivalDate;

@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd")
private LocalDate purchaseDate;
```

---

### Creación de fechas aleatorias para que sea más realista:

1. Creamos una clase de utilería en el paquete Utils.



2. Creamos creamos una **constante random (aleatoria)**,

```
//usado para asignar fechas aleatorias.  
private static final Random random = new Random(); 2 usages
```

3. Creamos **2 métodos estáticos** (**las clases de utilería deben llevar métodos estáticos!!**) que nos van a generar **2 fechas**, una más **cercana** a la fecha actual y otra más **lejana** a la fecha actual para que generemos fecha de salida y de llegada del vuelo. Para ello:

- a. **creamos una variable para saber cuantas horas vas a aumentar a la fecha con hora actual:** para ello se genera un número aleatorio que esté entre 2 valores (el primero el mayor el segundo el menor) y le sumamos a esa cantidad la hora por la que va a comenzar (Ej: Numero aleatorio entre estos 2 valores que empieza a partir de las 2).
- b. A la **fecha-hora actual le sumamos** la cantidad (en horas) **variable** generada.
- c. Repetimos esto para las 2 fechas.

```
@  //Creacion de método que saca fecha random cercana a la nuestra.  
public static LocalDateTime getRandomSoon(){ 2 usages  
    //intervalo random será entre 2 y 5 y empezará en las 2. Ej:  
    var randomHours = random.nextInt( bound: 5 - 2 ) + 2;  
    var now = LocalDateTime.now();  
    return now.plusHours(randomHours);  
}  
  
@  //Creacion de método que saca fecha random lejana a la nuestra.  
public static LocalDateTime getRandomLatter(){ 2 usages  
    //intervalo random será entre 6 y 12 y empezará en las 6.  
    var randomHours = random.nextInt( bound: 12 - 6 ) + 6;  
    var now = LocalDateTime.now();  
    return now.plusHours(randomHours);  
}
```

- d. Implementamos los métodos estáticos en la clase TicketService en nuestro caso, o en las clases Servicio, en los métodos en los que necesitemos generar nuestras fechas-horas aleatorias para nuestros vuelos.

Ej:

```
ticketToUpdate.setFly(fly);  
ticketToUpdate.setPrice(fly.getPrice().add(fly.getPrice().multiply(charge_price_percentage)));  
ticketToUpdate.setDepartureDate(BestTravelUtil.getRandomSoon());  
ticketToUpdate.setArrivalDate(BestTravelUtil.getRandomLatter());
```

Código Reservation (ReservationRequest, ReservationResponse, IReservationService, ReservationService): Muy parecido a Ticket, se añaden explicaciones extra de lo que no esté en ticket.

```
@NoArgsConstructor 8 usages
@NoArgsConstructor
@Data
@Builder
public class ReservationRequest {

    private String idClient;
    private Long idHotel;
    private Integer totalDays;
}
```

```
@NoArgsConstructor 12 usages
@NoArgsConstructor
@Data
@Builder
public class ReservationResponse implements Serializable {
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd HH:mm")
    private UUID id;
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd")
    private LocalDate dateStart;
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd")
    private LocalDate dateEnd;
    private Integer totalDays;
    private BigDecimal price;
    private HotelResponse hotel;
}
```

```
import java.math.BigDecimal;
import java.util.UUID;

public interface IReservationService extends CrudService<ReservationRequest, ReservationResponse, UUID>{
    BigDecimal findPriceHotel(Long hotelId); 1 usage 1 implementation
}
```

```
@Transactional no usages
@Service
@Slf4j
@AllArgsConstructor
public class ReservationService implements IReservationService {

    private final HotelRepository hotelRepository;
    private final CustomerRepository customerRepository;
    private final ReservationRepository reservationRepository;
    private static final BigDecimal charger_price_percentage = BigDecimal.valueOf(0.20); 3 usage

    @Override 1 usage
    public BigDecimal findPriceHotel(Long hotelId) {
        var hotel = this.hotelRepository.findById(hotelId).orElseThrow();
        return hotel.getPrice().add(hotel.getPrice().multiply(charger_price_percentage));
    }
}
```

```
@Override 2 usages
public ReservationResponse create(ReservationRequest request) {
    var hotel = this.hotelRepository.findById(request.getIdHotel()).orElseThrow();
    var customer = customerRepository.findById(request.getIdClient()).orElseThrow();

    var reservationToPersist = ReservationEntity.builder()
        .id(UUID.randomUUID())
        .hotel(hotel)
        .customer(customer)
        .totalDays(request.getTotalDays())
        .dateTimeReservation(LocalDateTime.now())
        .dateEnd(LocalDate.now().plusDays(request.getTotalDays()))
        .dateStart(LocalDate.now())
        .price(hotel.getPrice().add(hotel.getPrice().multiply(charger_price_percentage)))
        .build();

    var reservationPersisted = this.reservationRepository.save(reservationToPersist);
    log.info("Reservation saved with id: {}", reservationPersisted.getId());
    return this.entityToResponse(reservationPersisted);
}
```

```
@Override 2 usages
public ReservationResponse read(UUID uuid) {
    var response = this.reservationRepository.findById(uuid).orElseThrow();

    return this.entityToResponse(response);
}
```

```

@Override 2 usages
public ReservationResponse update(ReservationRequest request, UUID id) {
    var reservationToUpdate = this.reservationRepository.findById(id).orElseThrow();
    var hotel = this.hotelRepository.findById(request.getIdHotel()).orElseThrow();

    reservationToUpdate.setHotel(hotel);
    reservationToUpdate.setPrice(hotel.getPrice().add(hotel.getPrice().multiply(charge_price_percentage)));
    reservationToUpdate.setDateStart(LocalDate.now());
    reservationToUpdate.setDateEnd(LocalDate.now().plusDays(request.getTotalDays()));
    reservationToUpdate.setTotalDays(request.getTotalDays());}

    var reservationUpdate = this.reservationRepository.save(reservationToUpdate);
    log.info("Reservation update whit id {}", reservationUpdate.getId());

    return entityToResponse(reservationUpdate);
}

```

```

@Override 2 usages
public void delete(UUID id) {
    var ticketToDelete = this.reservationRepository.findById(id).orElseThrow();
    this.reservationRepository.delete(ticketToDelete);
}

@ private ReservationResponse entityToResponse(ReservationEntity entity){ 3 usages
    var response = new ReservationResponse();
    BeanUtils.copyProperties(entity, response);
    var hotelResponse = new HotelResponse();
    BeanUtils.copyProperties(entity.getHotel(), hotelResponse);
    response.setHotel(hotelResponse);
    return response;
}
}

```

```

@RestController no usages
@RequestMapping(path="reservation")
@AllArgsConstructor
public class ReservationController {

    private final IReservationService reservationService;

    @PostMapping no usages
    public ResponseEntity<ReservationResponse> post(@RequestBody ReservationRequest request){
        return ResponseEntity.ok(reservationService.create(request));
    }

    @GetMapping(path="{id}") no usages
    public ResponseEntity<ReservationResponse> get(@PathVariable UUID id){
        return ResponseEntity.ok(reservationService.read(id));
    }
}

```

```

    @PutMapping(path="{id}") no usages
    public ResponseEntity<ReservationResponse> update(@PathVariable UUID id, @RequestBody ReservationRequest request){
        return ResponseEntity.ok(reservationService.update(request, id));
    }

    @DeleteMapping(path="{id}") no usages
    public ResponseEntity<Void> delete(@PathVariable UUID id){
        this.reservationService.delete(id);
        return ResponseEntity.noContent().build();
    }

    @GetMapping no usages
    public ResponseEntity<Map<String, BigDecimal>> getHotelPrice(@RequestParam Long hotelId){
        return ResponseEntity.ok(Collections.singletonMap("hotelPrice", this.reservationService.findPriceHotel(hotelId)));
    }
}

```

Recordar: Si no funciona el método delete en postman, es porque falta en los ManyToOne el fetch a tipo LAZY, cambiar!!!!!! Si no, no permite borrar.

```

ReservationEntity.java

22 public class ReservationEntity implements Serializable {
23
24     @Column(name = "date_end")
25     private LocalDate dateEnd;
26
27     @Column(name = "total_days")
28     private Integer totalDays;
29     private BigDecimal price;
30
31
32     @ManyToOne(fetch = FetchType.LAZY)
33     @JoinColumn(name="hotel_id")
34     private HotelEntity hotel;
35
36     @ManyToOne(fetch = FetchType.LAZY)
37     @JoinColumn(name="tour_id", nullable = true)
38     private TourEntity tour;
39
40     @ManyToOne(fetch = FetchType.LAZY)
41     @JoinColumn(name="customer_id")
42     private CustomerEntity customer;
43
44 }

```

## Creación de Catálogos:

Catálogos de Vuelo:

1. Creamos una interfaz base en la carpeta abstract\_service de nombre CatalogService.
2. Esta interfaz va a recibir un tipo de parámetro Response lo llamamos R
3. Creamos los métodos de la interfaz:
  - a. 1er Método: Mostrar los resultados paginados. Para ello ponemos: Page<R> un page de tipo response, entran 3 parámetros un integer con

el número de la página, otro integer para el tamaño de la página(size) y un parámetro SortType para decirle cómo quieras que se ordene, ese SortType no está creado, lo creamos en la carpeta de Utils.

## Diferencia entre @RequestBody y @RequestParam:

Lo primero es tener bien clara la diferencia de lo que es un parámetro de lo que es el cuerpo de la petición.

En HTTP, una petición (request) es el envío de un mensaje de una aplicación cliente a otra aplicación servidora. Ese mensaje se compone de

- **Cabeceras:** Metadatos del mensaje
- **Cuerpo:** Contenido del mensaje

Ese mensaje se envía a una URL utilizando un método (GET, POST, etc.)

Por lo que tenemos 3 sitios donde se puede incluir información (Cabeceras, Cuerpo y la propia URL de destino del mensaje)

¿Qué es un **parámetro**? Un parámetro es cada uno de los pares clave valor que se especifican en la URL. Si tu URL donde mandas el mensaje es <http://www.miweb.com/index.php?name=john&surname=doe> estás mandando dos parámetros, uno con clave name y valor john y otro con clave surname y valor doe.

¿Qué es el **cuerpo**? El cuerpo es el resto de los datos que van dentro del mensaje, es decir, no van en la URL como el parámetro. Existen varias formas de codificar estos valores. Para especificar la codificación, lo puedes hacer añadiendo una cabecera con clave "Content-Type" y después el tipo de codificación que vas a utilizar.

Por ejemplo, si utilizas: `Content-Type: application/x-www-form-urlencoded` el cuerpo contendría algo así:

```
name=john&surname=doe
```

Pero si utilizases `Content-Type: application/json` el cuerpo contendría algo así:

```
{"name": "john", "surname": "doe"}
```

Como ves la información es la misma, pero codificada de diferente forma.

Ya volviendo a Spring, ¿para qué nos valen estas anotaciones?

- La anotación `@RequestParam` nos sirve para acceder al valor de uno de los parámetros. Por ejemplo si tenemos este método:

```
public String getData(@RequestParam name, @RequestParam surname) { ... }
```

En este caso Spring automáticamente injectaría en la variable `name` el valor del parámetro `name` (recordemos que los parámetros vienen de la URL) y en la variable `surname` el valor parámetro `surname`.

- La anotación `@RequestBody` nos sirve para deserializar un objeto completo a partir del cuerpo de la petición. Es decir si tenemos esta clase:

```
public class Person {  
    private String name;  
    private String surname;  
    // Getters+setters, etc.  
}
```

y tenemos este método:

```
public String getData(@RequestBody Person person) { ... }
```

En este caso Spring nos injectaría en la variable `person` un objeto de la clase `Person` con los atributos `name` y `surname` informados con lo que nos venga en el cuerpo de esa petición.

Aquí Spring hace muchas cosas, porque primero mira el Content-type para determinar el tipo de codificación, después decodifica esos datos, crea una instancia nueva de la clase Person y llama a los setters correspondientes para informar sus atributos.

En general, al mandar un mensaje con GET no se manda cuerpo, por lo que los datos se suelen pasar en la URL, por lo que para obtenerlos se usaría `@RequestParam`.

Para otro tipo de métodos, los datos se suelen mandar en el cuerpo y/o como parámetros. Según lo que queramos obtener, deberemos usar `@RequestParam` o `@RequestBody`.

Si inspeccionas una petición con las herramientas de desarrollo de tu navegador vas a ver algo así:

```
▼ General
  Request URL: http://phpinfo.test/index.php?id=3
  Request Method: POST
  Status Code: 200 OK
  Remote Address: 127.0.0.1:88
  Referrer Policy: no-referrer-when-downgrade
  ▶ Response Headers (7)
  ▶ Request Headers      view source
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
    Accept-Encoding: gzip, deflate
    Accept-Language: es-ES,es;q=0.9,en;q=0.8
    Cache-Control: max-age=0
    Connection: keep-alive
    Content-Length: 21
    Content-Type: application/x-www-form-urlencoded
    Host: phpinfo.test
    Origin: http://phpinfo.test
    Referer: http://phpinfo.test/index.html
    Upgrade-Insecure-Requests: 1
    User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Ubuntu Chromium/83.0.4103.61 Chrome/83.0.4103.61 Safari/537.36
  ▶ Query String Parameters      view source      view URL encoded
    id: 3
  ▶ Form Data      view source      view URL encoded
    name: John
    surname: Doe
```

En los recuadros rojos vas a ver, que se aprecian:

- La URL con sus parámetros
- El método usado
- El Content-Type, una de las posibles cabeceras que se pueden utilizar en la que indicamos la codificación del cuerpo de la petición.
- En Query String Parameters tienes los valores que se pueden extraer con @RequestParam, es decir, los parámetros.
- En Form Data tienes los valores que se pueden extraer con @RequestBody, es decir, el cuerpo de la petición.