La BOGOTÁ que estamos construyendo

UNIVERSIDAD SERGIO ARBOLEDA

TODOS A LA U
Fórmate digital

MasterClass Unidad 3

Desarrollo de Software

Qué es un ambiente de trabajo

- Muchas veces, cuando estamos desarrollando utilizamos algunos valores que dependen EXCLUSIVAMENTE de nuestra máquina.
- La máquina en la que trabajamos, tiene un sistema operativo, puede tener un servidor de base de datos y nuestra configuración puede ser distinta. Por ejemplo, puede que hayamos establecido una contraseña o un usuario distinto para la base de datos.
- A este conjunto de configuraciones dependientes de la máquina es a lo que llamamos AMBIENTE.

Necesidad de separar ambientes

Muchas veces en producción se generan errores debido a que no se modificó el valor de una variable, Una URL apunta a una máquina local o configuraciones semejantes. La solución inicial es dejar escritas todas las posibilidades y comentar aquellas no se utilizan.

```
spring.datasource.url=jdbc:mysql://localhost:3306/carsDB
spring.datasource.username=root
spring.datasource.password=micontraseñasecreta!
server.port=8080
#spring.datasource.url=jdbc:mysql://10.0.0.37:3306/carsDB
#spring.datasource.username=admin
#spring.datasource.password=lacontraseñadeproduccion!
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto = update
```

Cómo separar ambientes

Para soportar perfiles, la configuración y variables de entorno, crearemos tantos archivos application***.properties*** como sea necesarios. Este archivo que es donde reside la configuración se encuentra en la carpeta ***src/main/resources***
Así se verán los archivos, cuyo nombre inicia con application y luego de un guión escribimos elnombre del ambiente, terminamos con ***.properties.***
La configuración común a todos los ambientes se puede quedar en el archivo inicial application.properties

application-dev.properties
application-qa.properties
application-prod.properties
.
.
.
application-myenv.properties

Cómo separar ambientes

application.properties

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.mode
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto = update
```

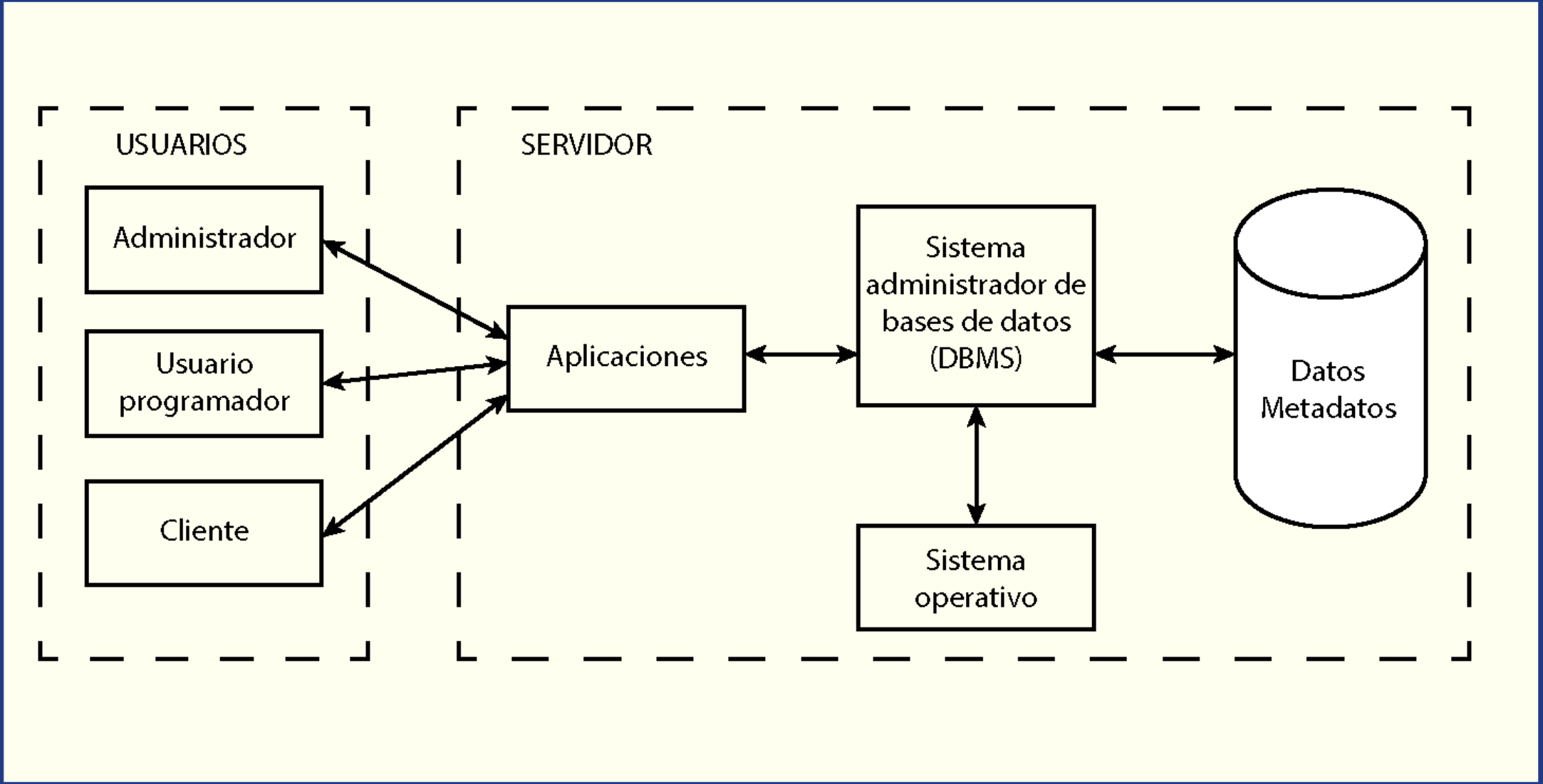
application-dev.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/carsDB
spring.datasource.username=root
spring.datasource.password=micontraseñasecreta!
server.port=8080
```

application-prod.properties

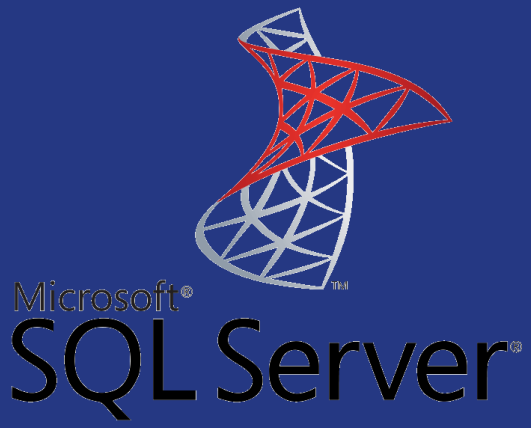
```
spring.datasource.url=jdbc:mysql://10.0.0.37:3306/carsDB
spring.datasource.username=admin
spring.datasource.password=lacontraseñadeproduccion!
server.port=80
```

Base de datos



Base de datos: DBMS

PostgreSQL



ORACLE

DATABASE

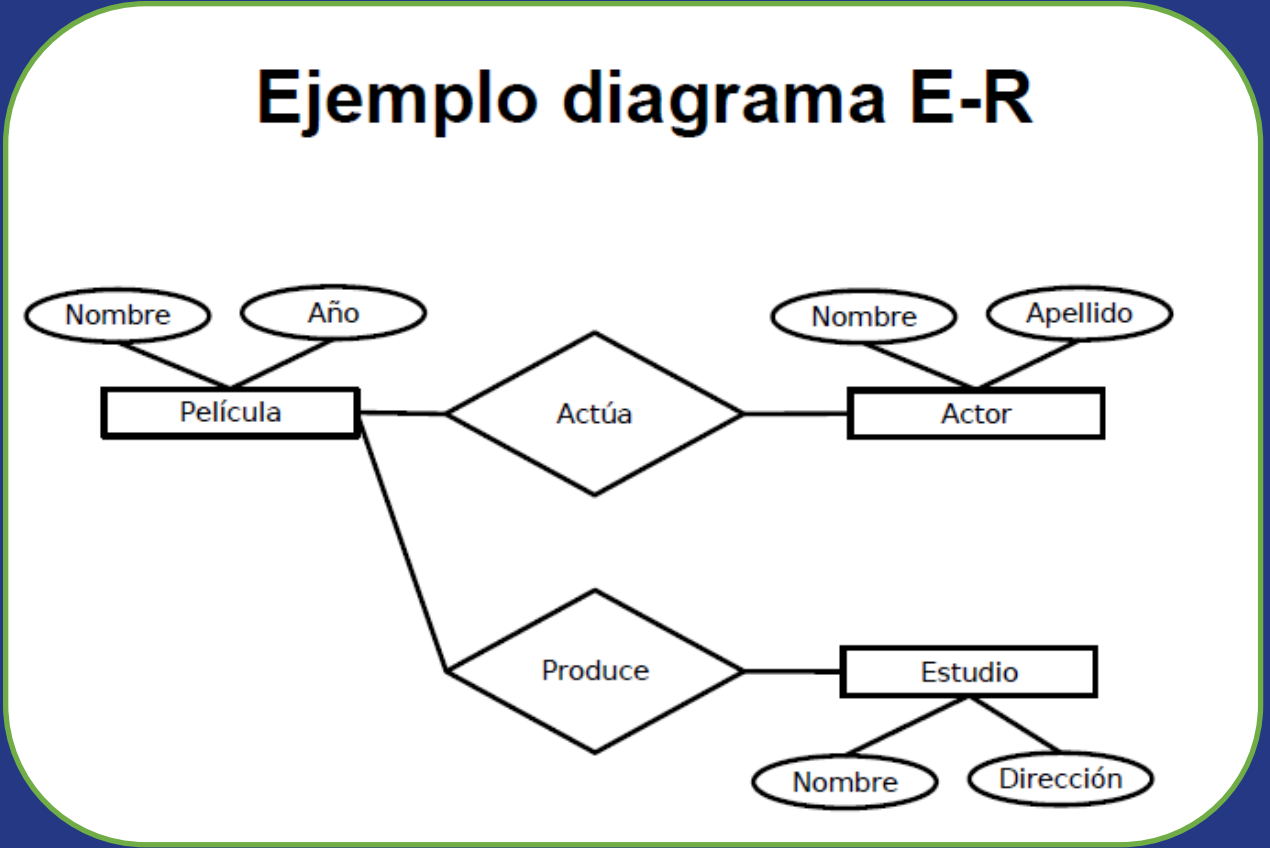
Modelo E/R

Representación del modelado relacional de datos.

Propuesto por Peter Chen en 1976.

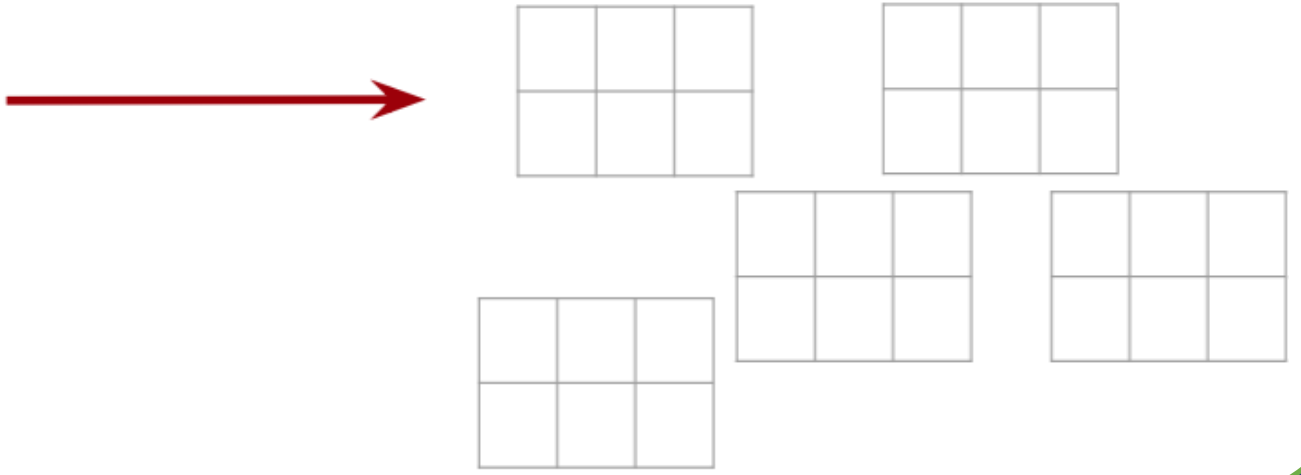
Elementos:

- Entidad
- Atributos
- Relaciones
- Restricciones



Modelo E/R y modelo Relacional

E/R		Relacional
Entidad	→	Relación (tabla)
Relaciones	→	Relación/Llaves foráneas



Modelo E/R y modelo Relacional

Diagrama E/R	Modelo Relacional
Relación 1 a 1	Atributos de una entidad pasan a ser atributos de otra.
Relación 1 a muchos	Llave primaria de la entidad con cardinalidad 1 pasa a ser llave foránea en la entidad con cardinalidad múltiple.
Relación muchos a muchos	Cada llave primaria de una entidad está presente como llave foránea en la otra entidad.

SQL – Creación de tablas

SQL para crear una tabla:

```
CREATE TABLE nombre_tabla (
        nombre_columna1 datatype NULL,
        nombre_columna2 datatype NOT NULL,
        nombre_columna3 datatype NULL,
        ...
);
```

MySQL – Creación de tablas

MySQL para crear una tabla ejemplo:
Para definir la llave primaria.

```
CREATE TABLE Cliente (
        cedula      int(10)          NOT NULL   PRIMARY KEY,
        nombre      varchar(30)      NOT NULL,
        apellido    varchar(30)      NOT NULL,
        email       varchar(30)      NULL,
        cargo       varchar(15)      NOT NULL
);
```

MySQL – Valores por defecto y listas de valores

MySQL para crear un valor por defecto y una lista de posibles valores:

```
CREATE TABLE pais (
        id              int(10)         AUTOINCREMENT           PRIMARY KEY,
        nombre          varchar(30)             NOT NULL    DEFAULT '',
        continente      enum('Asia', 'Europa`,`Oceania`,`America`,`Antartica`)
                        NOT NULL    DEFAULT  'Asia',
);
```

SQL – Borrar tablas

SQL para borrar una tabla:

```
DROP TABLE nombre_tabla
```

Mucho cuidado con este comando, es complicado recuperar información borrada.

SQL – Adicionar Columnas

SQL para agregar una columna a una tabla que ya existe:

Agregar una columna:

```
ALTER TABLE nombre_tabla
Add nombre_columna datatype NULL
```

Agregar varias columnas:

```
ALTER TABLE nombre_tabla
Add nombre_columna     datatype NULL,
      nombre_columna2  datatype NULL
```

Toda columna que se agregue debe tener la propiedad NULL.

SQL – Adicionar un valor por defecto

SQL para agregar un valor por defecto a una columna que ya existe:

```
ALTER TABLE nombre_tabla
ALTER nombre_columna  SET DEFAULT ‘valor por defecto’;
```

SQL – Inserción de datos (filas)

Sintaxis simplificada:

```
INSERT INTO nombre_tabla (nombre_atributo1, nombre_atributo2, …)
        VALUES (valor_atributo1, valor_atributo2, …);
```

Esta sintaxis se puede usar cuando los valores de los atributos se pasan en el orden en que están en la tabla.

```
INSERT INTO nombre_tabla
        VALUES (valor_atributo1, valor_atributo2, …);
```

SQL – Borrar datos

SQL para borrar una tabla:

```
DELETE FROM nombre_tabla;
```

SQL para borrar todos los registros de una tabla que cumplen con una condición:

```
DELETE FROM cliente WHERE país = ‘Francia’;
```

SQL – Actualizar datos

SQL para actualizar un registro de una tabla:

```
UPDATE nombre_tabla
SET atributo1 = valor1, atributo2 = valor 2, …
WHERE atributoN = valorN;
```

Ejemplo: Actualizar en la base de datos de clientes, aquellos que tenían en nacionalidad ‘Holanda’ por ‘Paises Bajos’.

```
UPDATE clientes
SET nacionalidad = ‘Paises Bajos’
WHERE nacionalidad = ‘Holanda’;
```

Spring Data

- El objetivo principal de Spring Data es gestionar tareas desde java en las bases de datos, que son repetitivas y que se pueden hacer mucho más fácil.
- Las tareas cotidianas o repetitiva con bases de datos pueden ser optimizadas gracias a los repositorios sin código.
- Los repositorios sin código de SpringData permiten ejecutar acciones de creación, consulta, actualización y borrado sin escribir una sola línea de código.

Incluir SpringData

- SpringData se agrega como una dependencia en maven. También desde Spring Initializr se puede elegir que queremos utilizar Spring Data.

```
    <version>0.0.1-SNAPSHOT</version>
    <name>RentCar</name>
    <description>RentCar</description>
    <properties>
        <java.version>11</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
```

Conectar la aplicación a Base de datos

Como veíamos en la clase en la que se trabajaban ambientes, es recomendable utilizar diferentes ambientes para establecer las diferentes conexiones.

```
spring.datasource.url=jdbc:mysql://localhost:3306/carsDB
spring.datasource.username=root
spring.datasource.password=micontraseñasecreta!
server.port=8080
```

Creación de las entidades

Las entidades serán creadas para representar los elementos de las tablas. Estas clases deberán ser decoradas con las anotaciones que se hicieron previamente en el ciclo anterior.
Muy importante generar los métodos getters y setters.

```java
@Entity
@Table(name = "clients")
public class Client implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer idClient;
    private String email;
    private String password;
    private String name;
    private Integer age;
```

Relaciones entre clases

- En este ejemplo, un carro, tiene una gama.
- A una misma gama pertenecen varios carros.
- La relación de Carros con gama entonces es MUCHOS A UNO.
- La relación de Gama con Carros es UNO A MUCHOS.
- Cuando una clase tiene una relación con muchos, los guarda en una estructura de colección, que puede ser una lista.
- Cuando la relación es con uno solo, lo guarda en un objeto de esa clase

```java
@OneToMany(cascade = {CascadeType.PERSIST},mappedBy="gama")
@JsonIgnoreProperties("gama")
private List<Car> cars;
```

```java
@ManyToOne
@JoinColumn(name="gamaId")
@JsonIgnoreProperties("cars")
private Gama gama;
```

Relaciones entre clases

- Las anotaciones indican las acciones de guardado y borrado qué estrategia utilizará y cuál es el correspondiente en la otra clase.
- Las anotaciones JsonIgnoreProperties son una solución para evitar la redundante recursividad, la cual consiste en:
  - Un carro tiene una gama → la gama tiene una lista de carros → los carros de la gama tienen… gama… y de esa manera se inicia una recursividad infinita.
  - Esta anotación hace que los carros de la gama ignoren el atributo gama y que la gama del carro, ignore la lista de carros. (Se ignoran solamente cuando se muestran)

```
@OneToMany(cascade = {CascadeType.PERSIST},mappedBy="gama")
@JsonIgnoreProperties("gama")
private List<Car> cars;
```

```
@ManyToOne
@JoinColumn(name="gamaId")
@JsonIgnoreProperties("cars")
private Gama gama;
```

JPA crea las tablas

Dentro de las muchas posiblidades que hay, JPA nos ofrece crear las tablas a partir de las clases.
Ya hemos visto como crear las clases automáticamente a partir de la base de datos, ahora podemos configurar la aplicación para que al ejecutarse cree las tablas. Hay que tener presente que si siempre lo dejamos en creación, borrará las tablas si las hay.
Puede que la actualización nos corrompa la información.

Para ello agregaremos lo siguiente en el archivo application.properties

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto = update
```

La segunda línea es para que Spring respete el nombre de las tablas y los atributos.

Interfaz CrudRepository

Permite hacer operaciones de CRUD sin necesidad de escribir código. Basta con definir una interfaz que la herede para definir cuáles son los tipos con los que se trabajará.
En este ejemplo, se hará un crud repository para trabajar con un objeto de tipo Producto. El segundo tipo que se define es Integer, que corresponde al tipo de dato de LA LLAVE PRIMARIA.
Con esto es suficiente para realizar todas las acciones CRUD de producto.

```
public interface ProductoCrudRepository extends CrudRepository <Producto, Integer>
{

}
```

Repositorio

Ahora se realiza una clase que sea un repositorio, el cual interactúa con el CrudRepository que creamos previamente. Esta clase hará uso de las funciones del CrudRepository y será solamente allí donde se use el CrudRepository.
La anotación Autowired indica a Spring que se debe encargar de crear implementaciones e instancias.

```
@Repository
public class CarRepository {
    @Autowired
    private CarCrudRepository carCrudRepository;

    public List<Car> getAll() { return (List<Car>) carCrudRepository.findAll(); }
}
```

Contenido

- Introducción a capa de Servicio
- Objetivo de la capa de Servicio en Spring
- La lógica en capa de Servicio

Servicio en Spring

- En Spring la anotación @Service es utilizada solamente en clases e indica que esta clase proveerá un servicio.
- Proveer un servicio significa que esa clase se encargará de funcionalidades de negocio.
- @Service hace que nuestra clase se convierta en un componente de Spring, lo cual permite al contexto administrar sus instancias.
- La lógica del negocio: restricciones, reglas, objetivos, procesos, harán parte de este componente. Es posible que este componente tenga que asociarse con otras clases de la capa de modelo. También se asocia con repositorios para acceder a base de datos.

Servicio en Spring

- Tener esta capa nos permite separar la lógica del negocio de las demás actividades de la aplicación como acceso a datos, gestión de seguridad o acceso de las peticiones.
- Aún dentro de la capa de modelo, la representación de la información y las entidades no están independientes de la lógica.
- Si el objetivo por ejemplo es establecer un reporte, el repositorio trae la información necesaria y el servicio es el encargado de establecer esos datos, trabajarlos y entregarlos como respuesta.
- Las restricciones lógicas en los procesos de acceso de datos también se hacen en la capa de Servicio.

Aplicando la lógica en un servicio

Ejemplo:

Actualizar campos cuando la información que es enviada es incompleta.

Es posible que al hacer la petición de actualización de datos, se indique solamente el identificador y el campo que se quiere modificar con la nueva información, así en la representación de esa información los demás campos serán nulos.

Aplicando la lógica en un servicio

Ejemplo:

Tenemos la entidad Cliente, la cual tiene los atributos:
-id
-email
-name
-password

Al hacer una petición de modificación de nombre, la información enviada suele ser:
{ “id”:2, “name”: “Juan”}
De este modo se pide establecer el nombre a “Juan” del usuario con id 12.

Aplicando la lógica en un servicio

Ejemplo:

El objeto que representa la información de la petición en realidad tiene la siguiente información:
-id:12
-email:null
-name:Juan
-password:null

De esta manera si solicitamos actualizar directamente el contenido del usuario con id=12 en la base de datos, también se sobreescribirán los datos de email y password siendo reemplazados por null.

Aplicando la lógica en un servicio

Ejemplo:

Entonces, en nuestro ejemplo lo que haremos es que en la clase de SERVICIO traeremos el usuario en cuyo id es 12 y actualizaremos solamente los campos que no sean nulos.

*Este proceso es posible generalizarlo para todas las clases, pero para facilitar la comprensión del código, lo haremos con los atributos y métodos específicos de la clase.

Iniciamos mostrando cómo se establece la clase servicio de Clientes.

Aplicando la lógica en un servicio

Ejemplo:

Dentro de nuestro proyecto creamos un paquete que se llame service.

Acá creamos una clase que se llame ClientService.

```
@Service
public class ClientService {
    @Autowired
    private ClientRepository clientRepository;
```

Aplicando la lógica en un servicio

Ejemplo:

Nuestro Servicio, tendrá entonces acceso a un repositorio de Clientes llamado ClientRepository.

Con la etiqueta @Autowired permitiremos a Spring la gestión de la creación de las instancias.

También crearemos un método que actualice el Cliente. Este método recibirá el objeto Client con la información que se quiere actualizar y retornará un objeto cliente con toda la información con la que quedó el objeto en la base de datos.

Aplicando la lógica en un servicio

Ejemplo:

```
public Client update(Client client){
    if(client.getIdClient()!=null){
        Optional<Client> e= clientRepository.getClient(client.getIdClient());
        if(!e.isEmpty()){
            if(client.getName()!=null){
                e.get().setName(client.getName());
            }
            if(client.getPassword()!=null){
                e.get().setPassword(client.getPassword());
            }
            clientRepository.save(e.get());
            return e.get();
        }else{
            return client;
        }
    }else{
        return client;
    }
}
```

Aplicando la lógica en un servicio

Ejemplo:

En este método, se crea un Optional<Client> que se llama e. Este objeto es inicializado con la petición al repositorio del elemento que tenga el id del client que llega por parámetro

```
Optional<Client> e= clientRepository.getClient(client.getIdClient());
```

El objeto e al ser opcional, puede o no tener un objeto cliente en su interior. Si el objeto con el id existe, lo contendrá, de otro modo estará vacío. Acá viene la primera restricción: No se puede editar algo que no existe.

Aplicando la lógica en un servicio

Ejemplo:

Para saber si el objeto existe, se utiliza la función isEmpty() que retorna un booleano. En este caso se niega para empezar a actualizar en caso que no esté vacío. Para acceder al objeto cliente que contiene e, nos referimos a él como e.get()

```
if(!e.isEmpty()){
        if(client.getName()!=null){
          e.get().setName(client.getName());
        };
```

Ahora, las validaciones que se hacen son propias del ejemplo Client, en las que se preguntan si los atributos name y password NO son null, en ese caso establecemos el valor del cliente de e con el valor de client.

Aplicando la lógica en un servicio

Ejemplo:

Por último, actualizamos en la base de datos el objeto que contiene e.

```
clientRepository.save(e.get());
return e.get();
```

Además, retornamos este objeto como respuesta, así la respuesta será el objeto actualizado.

¿Hemos notado que no se actualizaron todos los atributos de cliente? Pues hemos determinado para este ejemplo que ni el ID ni el correo electrónico sean actualizables, de esta manera aún cuando esta información esté en la petición no se modificará. De esta manera vemos cómo el Servicio atiende a la lógica del negocio.

Aplicando la lógica en un servicio

Ejemplo 2:

Ahora veremos cómo se puede orquestar la lógica de un servicio para que organice la respuesta a un reporte.

Asumamos que tenemos el requerimiento de saber cuántas reservas están completas y cuántas están canceladas.

Para iniciar, se creará un DTO, que es una clase en la que guardaremos la información y la transportaremos a lo largo de la aplicación. Esta clase tiene dos números: completas y canceladas.

Aplicando la lógica en un servicio

Ejemplo 2:

```java
public class StatusAmount {
    private int completed;
    private int cancelled;

    public StatusAmount(int completed, int cancelled) {
        this.completed = completed;
        this.cancelled = cancelled;
    }

    public int getCompleted() { return completed; }
    public void setCompleted(int completed) { this.completed = completed; }
    public int getCancelled() { return cancelled; }
    public void setCancelled(int cancelled) { this.cancelled = cancelled; }
}
```

Aplicando la lógica en un servicio

Ejemplo 2:

Hemos de suponer ahora, que la respuesta será una instancia de este objeto con los valores correspondientes en sus atributos.

Ahora lo que sigue es establecer la consulta de la información. Para ello tendremos que hacer dos consultas:

-Reservas con estado completo
-Reservas con estado cancelado.

Es recomendable que la consulta a la base de datos traiga el número y no los objetos con el objetivo de mejorar el rendimiento.

Aplicando la lógica en un servicio

Ejemplo 2:

A través de SpringJPA y los JPA Query Methods solicitaremos al repositorio que nos entregue las reservas según cada estado.

El servicio almacenará la información de cada consulta y luego, creará la respuesta. Se verá así:

```
public StatusAmount getReservationsStatusReport(){
    List<Reservation>completed=reservationRepository.getReservationsByStatus("completed");
    List<Reservation>cancelled=reservationRepository.getReservationsByStatus("cancelled");
    return new StatusAmount(completed.size(),cancelled.size());

}
```