**Tutorial Uso de JPA y Spring**

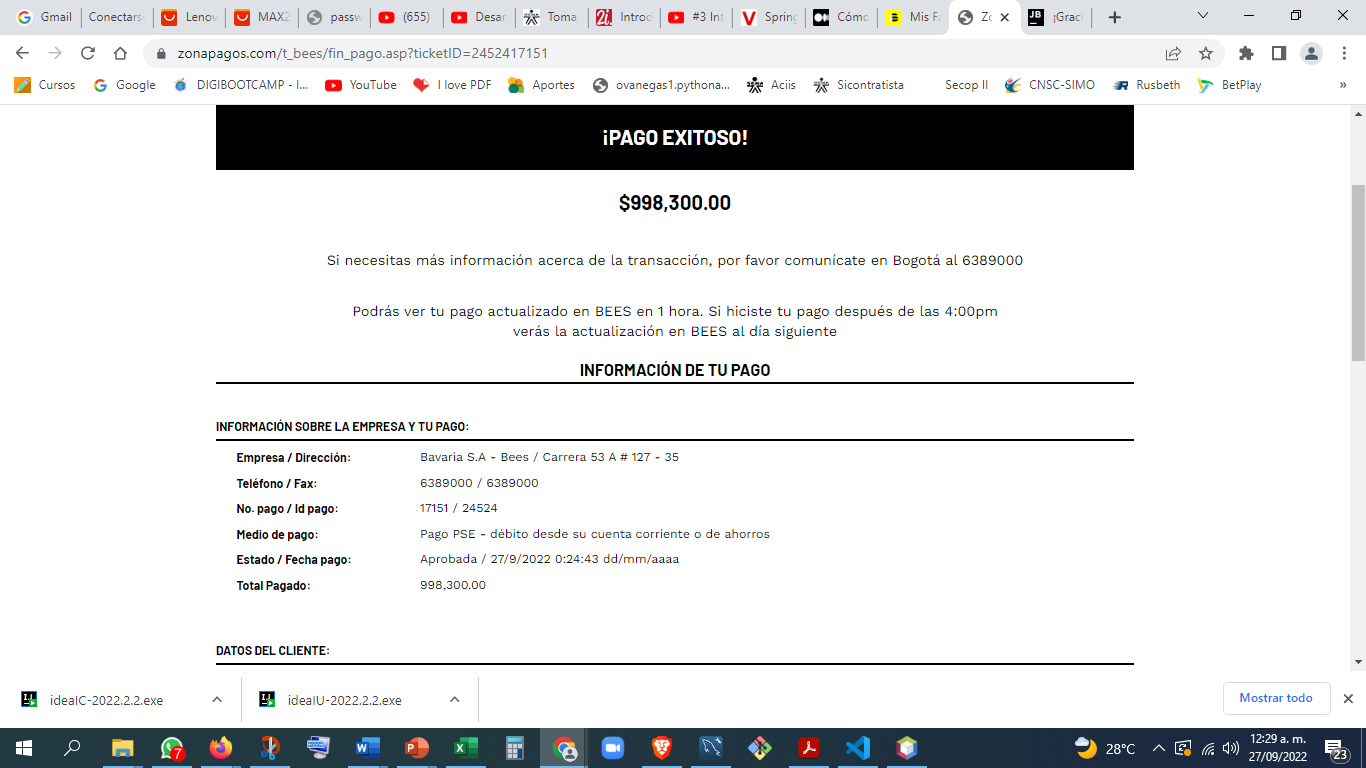
Ciclo 3: Desarrollo de Software

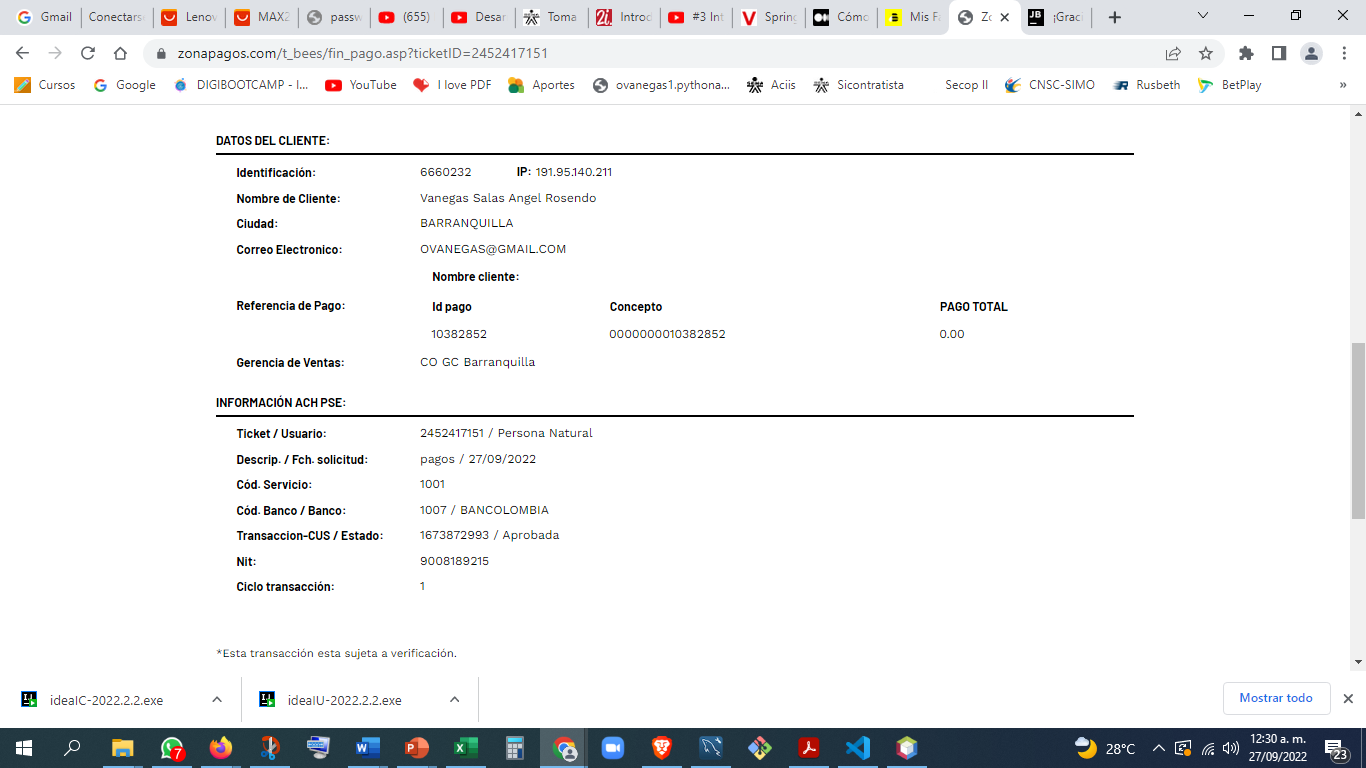
AÑO 2022

# **Introducción**

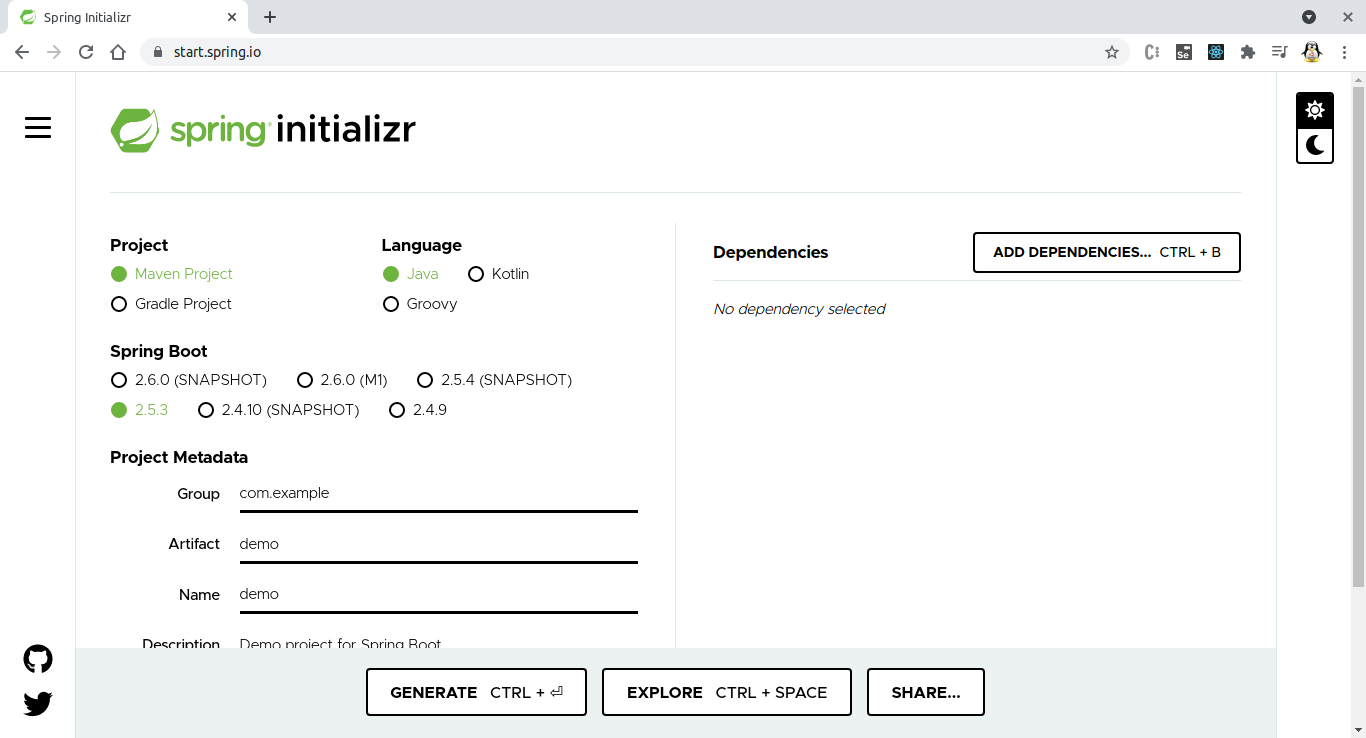
El presente documento se ha elaborado con el fin de facilitar el proceso de aprendizaje en torno al API de Persistencia de Java (JPA). Se espera que al finalizar el documento el estudiante esté en la capacidad de representar de manera correcta un modelo de bases de datos, en una estructura de clases que las representan o mapean. Se propone la realización de un ejemplo que habilite al estudiante a realizar una aplicación web basada en Spring boot.

# **Aplicación WEB**





Para realizar una aplicación WEB, realizaremos un backend con servicios GET y POST. Iniciaremos haciendo uso de Spring Initializr para descargar la estructura del proyecto junto con el correspondiente archivo pom.xml.



Descargar el proyecto y abrirlo en el IDE.

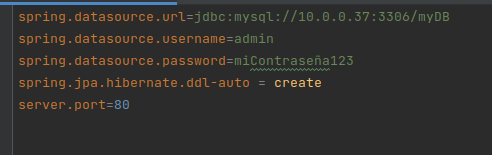
A continuación, iniciaremos estableciendo la conexión de la base de datos. Para ello, es útil crear más de un perfil y de esa manera lograr que la aplicación se ajuste a las condiciones de cada ambiente. Crearemos entonces un archivo que llamaremos application-dev2.properties.

Este archivo contendrá una cadena de conexión para una base de datos basada en archivos, lo cual en entorno de desarrollo es práctico, sin embargo no es recomendado en entorno de producción. Para producción se debe utilizar entonces un perfil que conecte con una base de datos.

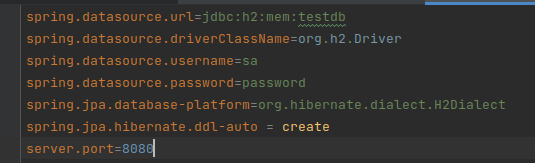
Nuestros archivos quedarán de la siguiente manera:

Archivo de producción:

application-prod.properties:

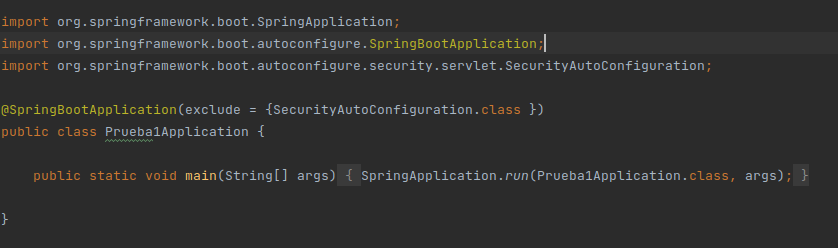


application-dev.properties:



De esa manera, especificamos la conexión y en cada caso la base de datos se creará en el momento de inicializar la aplicación.

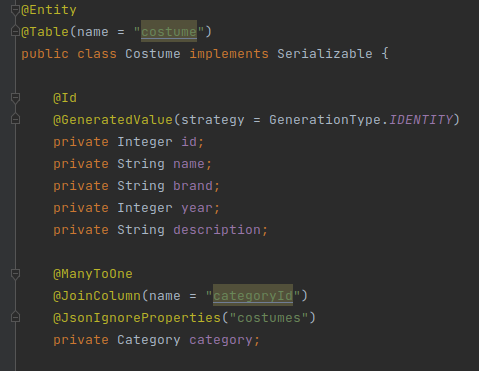
La clase principal en la que se ejecuta el método main tiene lo siguiente:



Si hemos incluido Spring Security en el momento de inicializar el proyecto, debemos poner la etiqueta @SpringBootApplication(exclude={SecurityAutoConfiguration.class}) con el fin de poder ejecutar la aplicación sin tener inconvenientes con la exigencia de inicio de sesión.

A continuación crearemos una estuctura de dos clases que representarán dos tablas. Estas dos tablas tienen una relación de uno a muchos, lo cual trae consigo un reto adicional.

La entidad debe ser representada de la siguiente forma:



La clase es ahora una Clase Entidad gracias a la anotación @Entity. Esta anotación está acompañada de la anotación @Table en la que se especifica el nombre. En este caso hemos especificado el nombre como el mismo nombre de la clase, pero en minúsculas. En caso de no especificarlo, la tabla se llamará igual.

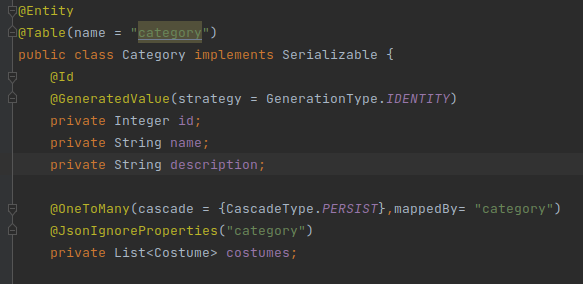
Hemos de recordar que en este ejercicio la estrategia es create (archivos application-dev.properties y application-prod.properties)lo que significa que creará siempre de ceros las tablas. Sin embargo la estrategia recomendada es Update en caso que siempre tengamos control de la base de datos y esta no sea un recurso compartido por terceros.

En esta tabla hemos establecido los campos id, name, brand, year y description con sus respectivos tipos de dato. Id es establecido como clave primaria gracias a la anotación @Id; la anotación @GeneratedValue nos permite trabajar el identificador como un valor auto incremental.

Sin embargo, uno de los atributos que se muestran ahí es category. Este atributo muestra la relación entre las dos tablas Costume y Category. La relación en este ejemplo corresponde a que un elemento de la tabla Costume tiene un elemento de la tabla Category. Un elemento de la tabla Category, puede ser referenciado por muchos de la tabla Costume. Eso hace que visto desde la tabla Costume, la relación sea muchos a uno y desde la tabla Category sea vista como uno a muchos.

La relación en este caso no se muestra como un campo entero en el que se almacena una llave foranea. En este caso se hace caso al paradigma de la programación orientada por objetos y este atributo es un objeto de la clase Category.

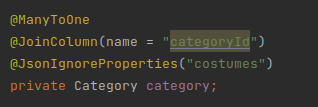
Veamos Category:

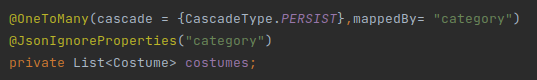
 =

En esta clase, la cual también está decorada para ser una clase entidad, con sus atributos establecidos según sus tipos de datos y el id es una clave primara que aumenta su valor automáticamente.

Sin embargo, en este caso, la clase Category tiene un atributo llamado costumes (en plural), que representa el conjunto de elementos de la clase Costume que referencian al objeto category.

Notar, que en esta clase, la relación marcada por la anotación @oneToMany.





Las imágenes de arriba muestran la manera como se representa la relación uno a muchos y su correspondiente muchos a uno.

La etiqueta JoinColumn de la clase Costume, indica cómo en la base de datos será al almacenado el id de la tabla category.

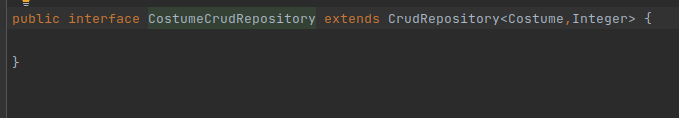
Es importante que coincida el valor de mapped by con el nombre del atributo dado en la otra clase, así el sistema es capaz de reconocer dónde encontrar los elementos que lo referencian.

Las anotaciones JsonIgnoreProperties son utilizadas para evitar llamados recurrentes en el momento de hacer la conversón a formato JSON. De esa manera se evita el ciclo infinito en el que Costume tiene categoría y categoría tiene una lista de costumes que muestran la misma categoría y este ciclo se repite infinitas veces. Para ello se hace que cuando costume muestre categoría, se ignore el campo costumes de categoría y cuando category muestre sus costumes, se ignorará el campo category. Se puede ignorar más de un atributo.

**REPOSITORIOS**

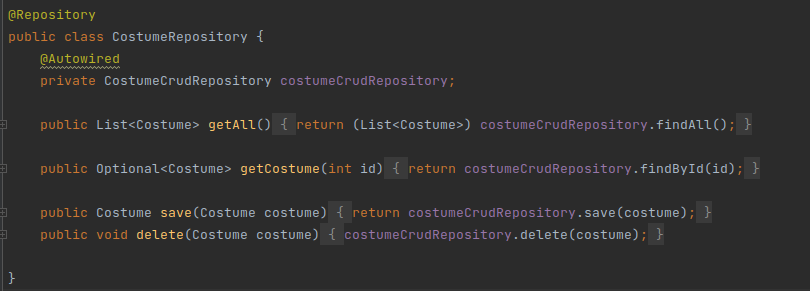
Con el fin de gestionar estas entidades en la base de datos, se crearán componentes de software que gracias al framework Spring, las labores de codificación se reducen a la gestión solamente de la lógica del negocio.

Para ello, tendremos que crear una interfaz que **heredará** de la interfaz CrudRepository de Spring. Esta interfaz recibirá a dos elementos importantes: El tipo de entidad que gestionará y el tipo de dato de la llave primaria. Nuestra interfaz se verá así:



Gracias a Spring, este componente será el encargado de gestionar las acciones CRUD de la entidad en la base de datos.

Para hacer uso de este componente crearemos una clase que utilizará un objeto de tipo CustomeCrudRepository para hacer las operaciones con base de datos. Este objeto será un repositorio, el repositorio se llamará @Repository.



Para que Spring gestione CostumeRepository, le pondremos la anotación @Repository. Esta clase tendrá solo un atributo, el cual es de tipo CostumeCrudRepository.

En este momento nos debemos estar cuestionando cómo podremos tener una instancia de CostumeCrudRepository si solamente lo hemos definido como una interfaz. Pues bien, gracias a que la clase CostumeRepository es un @Repository, nuestro CostumeCrudRepository se registra como @AutoWired y Spring será entonces el encargado de proveernos la instancia sin que sea una preocupación para nosotros. Podremos utilizar los elementos anotados por AutoWired en los entornos configurados como componentes de Spring: @Controller, @Service, @Repository o @RestController.

Los métodos son los necesarios para obtener todas las entidades, obtener un objeto en particular según su id, guardar alguno y borrar según su id. Como se puede ver, no hay método para la actualización, pues en el momento en que traigo un elemento de la base de datos, si lo modifico, lo puedo actualizar utilizando el mismo método de guardado. El método guardar, crea nuevos elementos en la base de datos o actualiza los existentes.

**SERVICIOS**

Ya que tenemos los repositorios y las entidades listas, nos centraremos en la lógica de negocio. Las operaciones lógicas que determinan condiciones y acciones respecto al acceso de los elementos de la base de datos. Este componente será entonces un Servicio. El servicio de este ejemplo, será un servicio básico que contendrá la gestión de las operaciones de creación, borrado, actualización y obtención de elementos, ya sea todos o uno en particular por id.

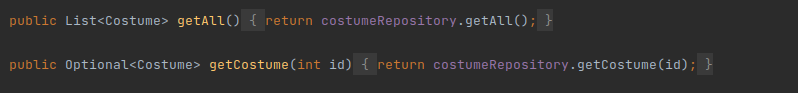
Para eso, lo primero que haremos es crear la clase CostumeService y la anotaremos como @Service.



Esta clase, tendrá un objeto de tipo Repository (el que recién creamos) que se encargará de gestionar la información en la base de datos. Al igual que lo hicimos en la sección anterior, acá Spring se encargará de inicializarlo y crear instancias. Eso implica anotarlo con @Autowired.



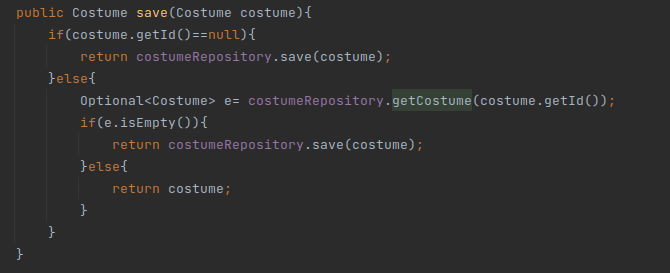
Los métodos que tendrá nuestro servicio están determinados por el objetivo del negocio. Nuestros objetivos en este momento son las operaciones básicas, así que empezaremos a revisar cada método:



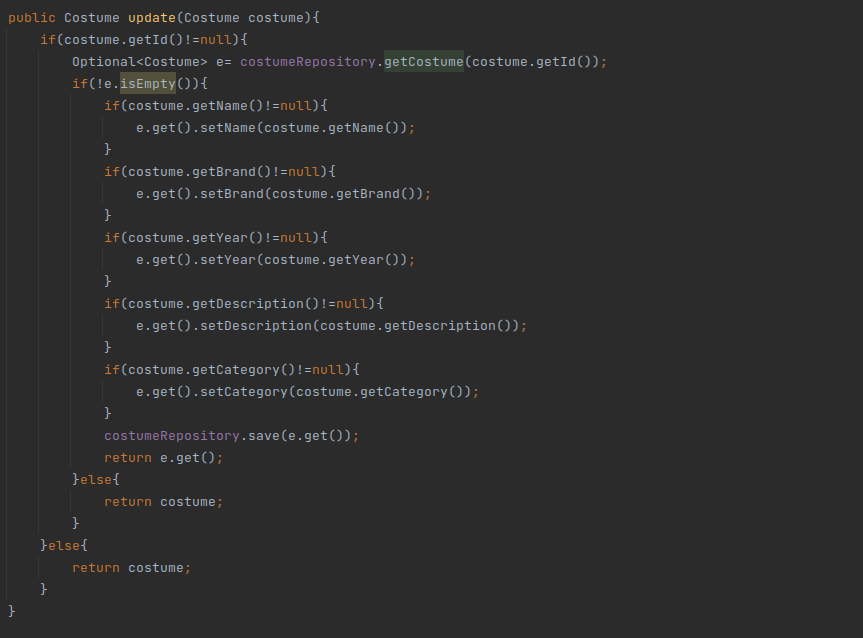
Estos dos métodos serán los encargados de las acciones GET que son las que traerán elementos de la base de datos. El primero es para traer todos los elementos. Los métodos son sencillos, debido a que corresponden simplemente a hacer el llamado de los métodos del repositorio. El repositorio tendrá los métodos getAll() y getCostume(id).

El método getAll retornará siempre una lista con todos los elementos. En caso de no haber ninguno la lista estará vacía.

El método getCostume, que definimos en el repositorio, invoca el método findById y retorno un objeto de tipo Optional. Esto es debido a que puedo buscar por id un elemento y este puede o no estar en la base de datos. Optional es una excelente alternativa a los inconvenientes que genera el trabajo con null. En el caso en que no se encuentre, el objeto Optional estará vacío. Si lo encuentra, el resultado estará dentro del objeto optional.



El método Save puede implementar simplemente el llamado al método save del repositorio, sin embargo acá realizamos una validación para asegurarnos que el método save del servicio no sea utilizado para actualizar y sea empleado solamente para guardar elementos nuevos. Lo que esta lógica implementa es validar que el objeto costume que llega por argumento de la función no tenga un id establecido. Si no lo tiene, lo guardará en la base de datos. Su id será establecido automáticamente al guardar. Sin embargo, si el objeto tiene id lo que hace es buscar un objeto en la base de datos que tenga ese id. Si no lo encuentra, entonces lo almacena sin inconveniente, de encontrarlo, retorna el objeto pero no lo guarda en la base de datos. Esta lógica es utilizada como ejemplo y no debe asumirse como obligatoria o recomendada, pues los requerimientos de almacenamiento dependen de la lógica de negocio y de los requerimientos capturados.

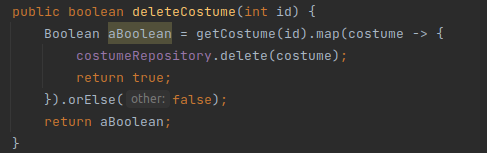


La actualización, en este método es una lógica muy dependiente de la entidad. Es posible realizar una generalización a este método, pero no lo cubriremos en este documento.

Para realizar la actualización de una entidad, lo primero que hacemos es validar que se indique el id, de otro modo será imposible actualizarlo. Acá, separamos la lógica de guardado de la de actualización, por tal razón estamos evitando que el método de creación se utilice para actualizar. Junto con lo anterior, tenemos que hemos diferenciado plenamente las acciones de guardar y actualizar.

En caso que tengamos un id establecido en el argumento, entramos a consultar en la base de datos si existe. En caso de existir, procedemos a actualizar cada uno de los atributos.

Puede suceder que para realizar una actualización, no enviemos en la petición todos los atributos, sino que en lugar de ello enviemos solamente el id y el atributo que queremos modificar con el valor nuevo. Para ello lo que hacemos es validar cada uno de los campos, en caso de ser nulos no los modificaremos porque significa que esa información no se ha enviado. Este modo de actualizar también permite determinar qué campos pueden ser actualizables por ese método y qué campos no. por ejemplo acá, en una tabla de usuarios a menudo se puede encontrar que hay campos claves como el correo electrónico que no se permiten modificar. Por último, el objeto traído de la base de datos y modificado se guarda.



Para la función de borrado, proponemos como ejemplo una función lambda que evalúa la existencia del objeto buscado en la base de datos, y si lo encuentra, le pedirá al repositorio que lo borre. Al borrarlo retorna un verdadero, si no lo encuentra, no podrá hacer nada y retornará un falso. Es de notar que acá se está utilizando el método de la misma clase getCostume.

**CONTROLADOR**

El controlador será la puerta de entrada a nuestro backend. Esta clase es particularmente interesante, pues es ella la que define las rutas a las peticiones, el esquema de las URI y redirige a la lógica las solicitudes del usuario o de otros sistemas.



Esta clase inicia con la anotación @RestController. A continuación se establece la anotación @RequestMapping que establece la URL relativa para acceder a este controlador. El acceso a cada método entrá que ser precedido en la url por el mapeo que se establece allí.

Para que este controlador pueda conectarse con el servicio, se creará un objeto de tipo CostumeService el cual será anotado con Autowired para que Spring gestione sus instancias.

Posteriormente empezaremos a crear los métodos que gestionarán las peticiones correspondientes a los métodos HTTP utilizados, así cada método GET, POST, PUT, DELETE tendrán una correspondiente puerta de entrada que los dirige a la acción del Service (lógica del negocio) pertinente.

El método getCostumes tendrá acceso a través del path “all”. Es decir que para llegar ahí, la url será DIRECCIÓN\_DEL\_SERVIDOR/Costume/all.

Adicional, la anotación GetMapping implica que a esta url se llega a través de un método GET. Este método hace el llamado al método getAll del servicio que hemos definido previamente.

Se crea también otro método anotado con GetMapping el cuál tiene como parámetro en la url un valor de id encerrado en llaves (corchetes). Esto significa que lo que vaya después del caracter /será evaluado como una variable que se llamará id. Esta variable es entregada al método getCostume y este método se encarga de invocar el método del servicio.

Los demás métodos, anotados también con el tipo de petición que reciben: POST, PUT y DELETE y poniendo en la url el nombre de la acción que ejecutan, llaman los métodos del servicio para realizar la acción correspondiente.

Es de destacar que cada método recibe el parámetro según el canal por el que la información le sea entregada: En los métodos POST y PUT la información llega en el cuerpo de la solicitud (Request Body) y por eso el argumento de la función es decordado con @RequestBody. El argumento a su vez es un objeto de tipo Costume, lo que significa que automáticamente se hace una traducción entre el formato JSON de la petición y la creación de instancias.

Por su parte el método delete y el método getCostume reciben la información de la petición desde la url en la varialbe {id}, por eso el parámetro es decorado con @PathVariable(“id”) indicando cuál es la varialbe de la url que es relacionada con el parámetro.

Estos métodos retornan objetos de tipo Costume, List, Optional y booleano. En cualquier caso, serán automáticamente traducidos a JSON para dar respuesta.

De este modo nuestra aplicación web tiene un back end completo que gestiona su base de datos con JPA.