

Proyecto: Entrega I

Sistemas Transaccionales 2024-20

Estudiante 1:	Johan Camilo Suarez Sinisterra
Estudiante 2:	Kalia Angelica Gonzalez Jimenez
Estudiante 3:	Juan Miguel Delgado

Contenido

Análisis Modelo de Datos	Documentación Aplicación
Revisión Modelo de Datos	Documentación Arquitectura
Diseño de la BD	Usuario Oracle

Análisis Modelo de Datos

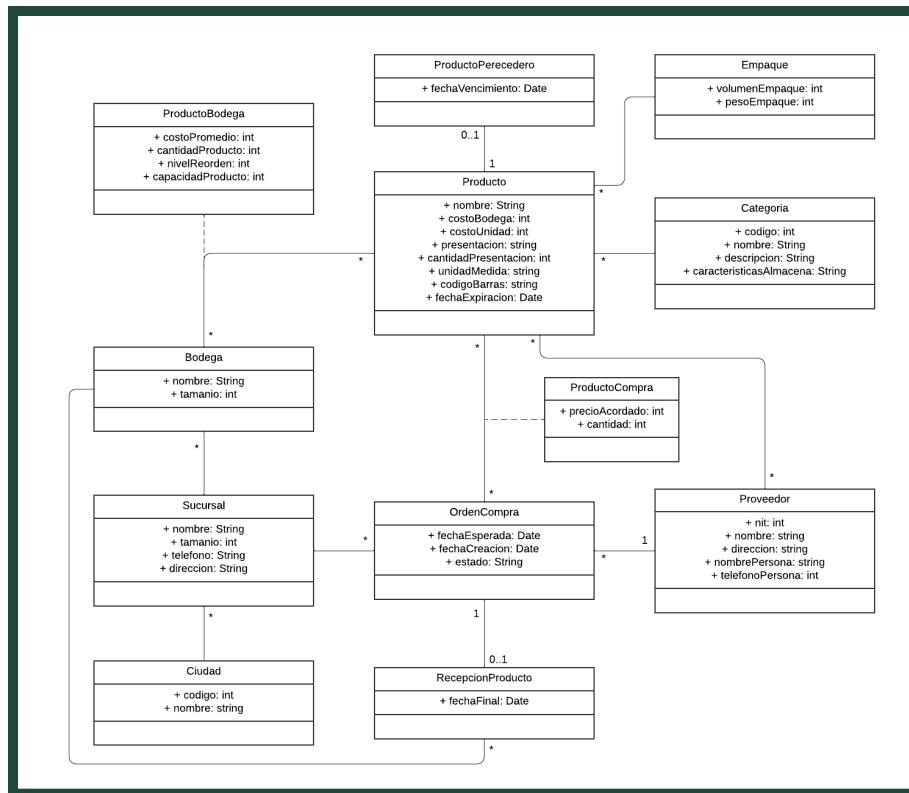
Gracias a la retroalimentación dada por nuestra monitora Sara, logramos encontrar mejoras en nuestros modelos de datos. Con este profundo análisis que realizamos, observamos que se puede llevar a cabo la realización de una base de datos adecuada, completa y desacoplada que cumpla con las expectativas esperadas para el cumplimiento de todos los requerimientos funcionales y de consulta. Es por eso que decidimos tener en cuenta el modelo sugerido y así

realizar las modificaciones necesarias a todos nuestros modelos de datos (UML, E/R y Relacional).

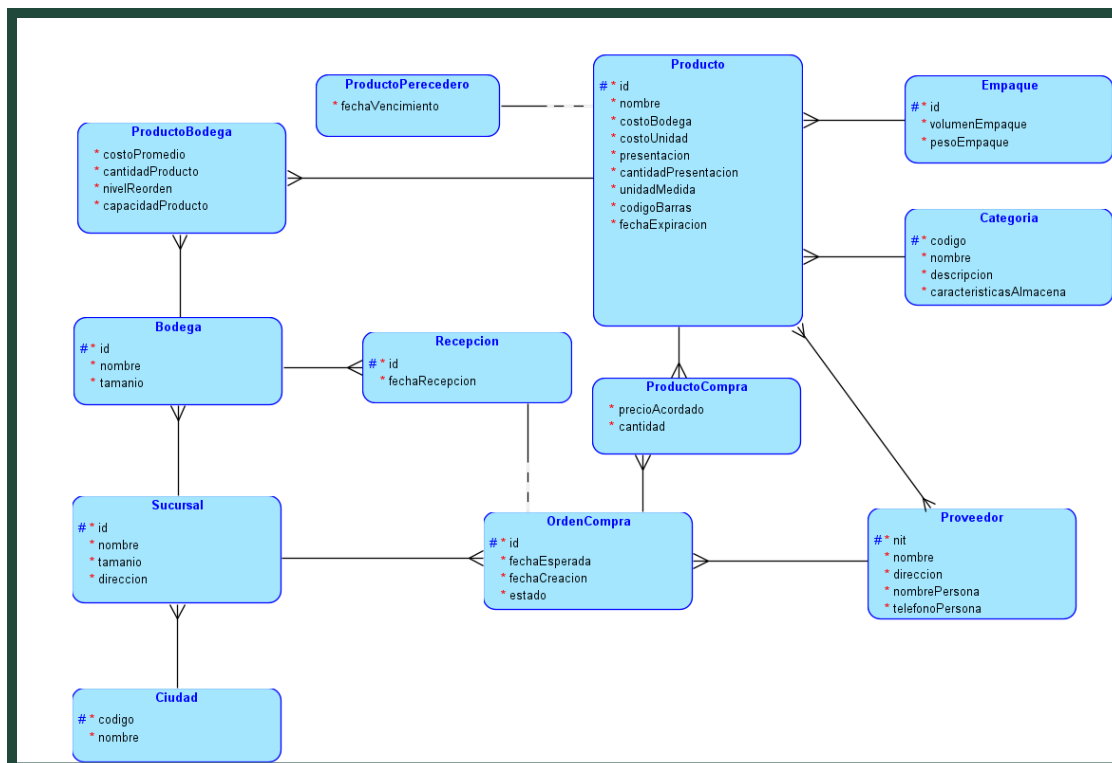
Con todo el contexto anterior, y respecto al Modelo UML, nos dimos a la tarea de agregar/modificar diferentes clases para obtener un modelo mucho más completo. Estas son:

- **Empaque:** clase nueva que representa la especificación del empaque de cada producto. Esta tendrá los siguientes atributos: volumenEmpaque, en centímetros cúbicos y pesoEmpaque, en gramos. Así mismo tendrá una relación de 1 a muchos con la clase Producto, representando que un Producto tiene un empaque asignado, y la especificación de un empaque puede ser la misma para muchos productos.
- **ProductoPerecedero:** clase que representa los productos que tienen una fecha de vencimiento. Las modificaciones que se realizaron fueron: el cambio de su atributo fechaExpiración por un atributo fechaVencimiento. Lo anterior debido a que todos los Productos tienen fecha de expiración, pero sólo los productos perecederos tienen una característica sobre su fecha de vencimiento. Igualmente, se cambió la relación de herencia, por una relacion de 1 a 0..1 entre Producto y ProductoPerecedero, que representa que un producto puede ser o no un producto perecedero que tiene una fecha de vencimiento.
- **Producto:** clase que representa todos los productos ofrecidos por SuperAndes. La modificación que se realizó a esta clase fue agregarle un nuevo atributo fechaExpiración. Así mismo, se eliminaron los atributos volumenEmpaque y pesoEmpaque para asignarlos a la clase Empaque anteriormente mencionada.
- **ProductoBodega y ProductoCompra:** clases que definen el detalle de cada una de las relaciones entre Producto con Bodega y Producto con OrdenCompra respectivamente. La modificación que se realizó fue el cambio en la visualización como clases independientes, y que pasaran a ser clases de asociación entre cada una de las relaciones mencionadas.

Modelo UML: Modificado



Modelo E/R: Modificado



Modelo Relacional: Modificado

Link del Excel con el modelo relacional:

[ModeloRelacional.xlsx](#)

Revisión Modelo de Datos

A continuación, se presentará la revisión del modelo relacional. Esto con el objetivo de verificar que todas las relaciones se encuentren en Forma Normal de Boyce-Codd (BCNF). En este análisis, para cada relación, se mostrarán las llaves candidatas, las dependencias entre atributos y una breve explicación de por qué se encuentran en 1NF, 2NF, 3NF y posteriormente en BCNF.

- **Empaque:** la definición de relación y de sus dependencias funcionales es la siguiente:
 - Empaque {id, volumenEmpaque, pesoEmpaque}
 - Llave candidata: Luego de analizar los atributos de la clase, se llega a la siguiente llave:
 - {id}
 - **Primera Forma Normal (1NF):** la relación se encuentra en primera forma normal ya que esta no cuenta con atributos multivariados. Es decir que contengan listas o algún tipo de estructura similar.
 - **Segunda Forma Normal (2NF):** la relación como ya se encuentra en 1NF, prosigue a ser evaluada para mirar si no se tienen dependencias parciales desde el atributo primo (id). Analizando la clase, un empaque al tener un id que lo identifique y distingue de los demás empaques, muestra la dependencia de la otra información que se necesita saber. Por ejemplo, para saber el volumen de un empaque x, se necesita saber su id para identificar cual es dicho empaque y que sea la correcta. Por lo tanto, se puede afirmar que la relación está en 2NF
 - **Tercera Forma Normal (3NF):** como la relación cumple con las dos reglas anteriores, se prosigue a analizar si está en la regla 3NF. En esta se evalúa que no exista alguna dependencia transitiva. Con la llave primaria, un atributo NO determina otro primo. Analizando todos los atributos junto con la llave primaria, no se ve otra dependencia o relación que puedan existir entre los demás atributos ya que estos son solo características de la entidad, mas no son los que guardan la información de ella. Por ejemplo, no se puede determinar el peso de un empaque con una llave diferente al id.
 - **Forma Normal Boyce-Codd:** la relación está en 3NF, y adicionalmente las llaves son simples y no dependen de otra llave, se puede decir que se encuentra en BCNF.
- **ProductoPerecedero:** la definición de relación y de sus dependencias funcionales es la siguiente:
 - ProductoPerecedero {id, fechaVencimiento}
 - Llave candidata: Luego de analizar los atributos de la clase, se llega a la siguiente llave:
 - {id}
 - **Primera Forma Normal (1NF):** la relación se encuentra en primera forma normal ya que esta no cuenta con atributos multivariados. Es decir que contengan listas o algún tipo de

estructura similar.

- **Segunda Forma Normal (2NF):** la relación como ya se encuentra en 1NF, prosigue a ser evaluada para mirar si no se tienen dependencias parciales desde el atributo primo (id). Analizando la clase, un producto perecedero al tener un id que lo identifique y distinga de los demás productos, muestra la dependencia de la otra información que se necesita saber. Por ejemplo, para saber la fecha de vencimiento de el producto x, se necesita saber su id para identificar cual es dicho producto y que sea el correcto. Por lo tanto, se puede afirmar que la relación está en 2NF
- **Tercera Forma Normal (3NF):** como la relación cumple con las dos reglas anteriores, se prosigue a analizar si está en la regla 3NF. En esta se evalúa que no exista alguna dependencia transitiva. Con la llave primaria, un atributo NO determina otro primo. Analizando todos los atributos junto con la llave primaria, no se ve otra dependencia o relación que puedan existir entre los demás atributos ya que estos son solo características de la entidad, mas no son los que guardan la información de ella. Como en este caso solo tenemos como atributo no primo la fecha de vencimiento, si o si para conocer la información de este de un producto perecedero específico se necesita saber su id.
- **Forma Normal Boyce-Codd:** la relación está en 3NF, y adicionalmente las llaves son simples y no dependen de otra llave, se puede decir que se encuentra en BCNF.

● **ProveedorProducto:** La definición de relación y de sus dependencias funcionales es la siguiente:

- ProveedorProducto{id, CantidadOfrecida}
- Llave candidata: Luego de analizar los atributos de la clase, se llega a la siguiente llave:
 - {id}
- **Primera Forma Normal (1NF):** la relación se encuentra en primera forma normal ya que esta no cuenta con atributos multivariados. Es decir que contengan listas o algún tipo de estructura similar.
- **Segunda Forma Normal (2NF):** La relación como ya se encuentra en 1NF, prosigue a ser evaluada para mirar si no se tienen dependencias parciales desde el atributo primo (id). Analizando la clase, un proveedor producto al tener un id que lo identifique y distinga de los demás productos de este tipo, muestra la dependencia de la otra información que se necesita saber. Por ejemplo, para saber la cantidad ofrecida de un proveedor producto x, hay que saber su id para identificar cuál es el producto y cuál es el correcto. Por lo tanto, se puede afirmar que la relación está en 2NF.
- **Tercera Forma Normal (3NF):** Como la relación cumple con las dos reglas anteriores, se prosigue a analizar si está en la regla 3NF. En esta se evalúa que no exista alguna dependencia transitiva. Con la llave primaria, un atributo NO determina otro primo. Analizando todos los atributos junto con la llave primaria, no se ve otra dependencia o relación que puedan existir entre los demás atributos ya que estos son solo características de la entidad, mas no son los que guardan la información de ella. Como en este caso solo tenemos como atributo no primo la cantidad Ofrecida, si o si para

conocer la información de este de un proveedor producto específico se necesita saber su id.

- **Forma Normal Boyce-Codd:** la relación está en 3NF, y adicionalmente las llaves son simples y no dependen de otra llave, se puede decir que se encuentra en BCNF.

- **RecepcionProducto:** La definición de relación y de sus dependencias funcionales es la siguiente:

- RecepcionProducto{id, fechaFinal}
- Llave candidata: Luego de analizar los atributos de la clase, se llega a la siguiente llave:
 - {id}
- **Primera Forma Normal (1NF):** la relación se encuentra en primera forma normal ya que esta no cuenta con atributos multivariados. Es decir que contengan listas o algún tipo de estructura similar.
- **Segunda Forma Normal (2NF):** La relación como ya se encuentra en 1NF, prosigue a ser evaluada para mirar si no se tienen dependencias parciales desde el atributo primo (id). Analizando la clase, una recepción de producto al tener un id que lo identifique y distingue de las demás recepciones de producto, muestra la dependencia de la otra información que se necesita saber. Por ejemplo, para saber la fecha esperada de una recepción de compra x, se necesita saber su id para identificar la recepción y que sea la correcta. Por lo tanto, se puede afirmar que la relación está en 2NF.
- **Tercera Forma Normal (3NF):** Como la relación cumple con las dos reglas anteriores, se prosigue a analizar si está en la regla 3NF. En esta se evalúa que no exista alguna dependencia transitiva. Con la llave primaria, un atributo NO determina otro primo. Analizando todos los atributos junto con la llave primaria, no se ve otra dependencia o relación que puedan existir entre los demás atributos ya que estos son solo características de la entidad, mas no son los que guardan la información de ella. Por ejemplo, no se puede determinar la fecha final de una recepción con una llave diferente al id.
- **Forma Normal Boyce-Codd:** la relación está en 3NF, y adicionalmente las llaves son simples y no dependen de otra llave, se puede decir que se encuentra en BCNF.

- **ProductoCompra:** La definición de relación y de sus dependencias funcionales es la siguiente:

- ProductoCompra {id, precioAcordado, cantidad}
- Llave candidata: Luego de analizar los atributos de la clase, se llega a la siguiente llave:
 - {id}
- **Primera Forma Normal (1NF):** la relación se encuentra en primera forma normal ya que esta no cuenta con atributos multivariados. Es decir que contengan listas o algún tipo de estructura similar.

- **Segunda Forma Normal (2NF):** La relación como ya se encuentra en 1NF, prosigue a ser evaluada para mirar si no se tienen dependencias parciales desde el atributo primo (id). Analizando la clase, un producto de compra al tener un id que lo identifique y distinga de los demás productos de compra, muestra la dependencia de la otra información que se necesita saber. Por ejemplo, para saber la cantidad y el precio acordado de los productos que tiene una compra x, se necesita saber su id para identificarlos uno por uno. Por lo tanto, se puede afirmar que la relación está en 2NF.
 - **Tercera Forma Normal (3NF):** Como la relación cumple con las dos reglas anteriores, se prosigue a analizar si está en la regla 3NF. En esta se evalúa que no exista alguna dependencia transitiva. Con la llave primaria, un atributo NO determina otro primo. Analizando todos los atributos junto con la llave primaria, no se ve otra dependencia o relación que puedan existir entre los demás atributos ya que estos son solo características de la entidad, mas no son los que guardan la información de ella. Por ejemplo, no se puede determinar el precio acordado del producto con una llave diferente al id.
 - **Forma Normal Boyce-Codd:** la relación está en 3NF, y adicionalmente las llaves son simples y no dependen de otra llave, se puede decir que se encuentra en BCNF.
-
- **ProductoBodega:** La definición de relación y de sus dependencias funcionales es la siguiente:
 - ProductoBodega{id, costoPromedio, cantidadProducto, nivelReorden, capacidadProducto}
 - Llave candidata: Luego de analizar los atributos de la clase, se llega a la siguiente llave:
 - {id}
 - **Primera Forma Normal (1NF):** la relación se encuentra en primera forma normal ya que esta no cuenta con atributos multivariados. Es decir que contengan listas o algún tipo de estructura similar.
 - **Segunda Forma Normal (2NF):** La relación como ya se encuentra en 1NF, prosigue a ser evaluada para mirar si no se tienen dependencias parciales desde el atributo primo (id). Analizando la clase, un producto de bodega al tener un id que lo identifique y distinga de los demás productos de bodegas, muestra la dependencia de la otra información que se necesita saber. Por ejemplo, para saber el costo, cantidad y capacidad de los productos que tiene una bodega x, se necesita saber su id para identificarlos uno por uno. Por lo tanto, se puede afirmar que la relación está en 2NF.
 - **Tercera Forma Normal (3NF):** Como la relación cumple con las dos reglas anteriores, se prosigue a analizar si está en la regla 3NF. En esta se evalúa que no exista alguna dependencia transitiva. Con la llave primaria, un atributo NO determina otro primo. Analizando todos los atributos junto con la llave primaria, no se ve otra dependencia o relación que puedan existir entre los demás atributos ya que estos son solo características de la entidad, mas no son los que guardan la información de ella. Por ejemplo, no se puede determinar el costo promedio del producto con una llave diferente al id.

- **Forma Normal Boyce-Codd:** la relación está en 3NF, y adicionalmente las llaves son simples y no dependen de otra llave, se puede decir que se encuentra en BCNF.

- **OrdenCompra:** La definición de relación y de sus dependencias funcionales es la siguiente:
 - OrdenCompra {id,fechaEsperada,fechaCreacion,estado}
 - Llave candidata: Luego de analizar los atributos de la clase, se llega a la siguiente llave:
 - {id}
 - **Primera Forma Normal (1NF):** la relación se encuentra en primera forma normal ya que esta no cuenta con atributos multivariados. Es decir que contengan listas o algún tipo de estructura similar.
 - **Segunda Forma Normal (2NF):** La relación como ya se encuentra en 1NF, prosigue a ser evaluada para mirar si no se tienen dependencias parciales desde el atributo primo (id). Analizando la clase, una orden de compra al tener un id que lo identifique y distinga de los demás órdenes de compra, muestra la dependencia de la otra información que se necesita saber. Por ejemplo, la fecha esperada está determinada por cual fue la orden de compra ya que en esa fecha se realiza varias órdenes de compra, Por lo tanto, se puede afirmar que la relación está en 2NF.
 - **Tercera Forma Normal (3NF):** Como la relación cumple con las dos reglas anteriores, se prosigue a analizar si está en la regla 3NF. En esta se evalúa que no exista alguna dependencia transitiva. Con la llave primaria, un atributo NO determina otro primo. Analizando los atributos junto con la llave primaria, no se ve otra dependencia o relación existente entre los demás, ya que son características de la orden de compra, sino los que guardan la información de ella. Por ejemplo, no se puede determinar el estado de una orden de compra sin saber cuál es, que este lo determina el id.
 - **Forma Normal Boyce-Codd:** la relación está en 3NF, y adicionalmente las llaves son simples y no dependen de otra llave, se puede decir que se encuentra en BCNF.

- **Ciudad:** La definición de relación y de sus dependencias funcionales es la siguiente:
 - Ciudad {código, nombre}
 - Llave candidata: Luego de analizar los atributos de la clase, se llega a la siguiente llave:
 - {codigo}
 - **Primera Forma Normal (1NF):** la relación se encuentra en primera forma normal ya que esta no cuenta con atributos multivariados. Es decir que contengan listas o algún tipo de estructura similar.
 - **Segunda Forma Normal (2NF):** La relación como ya se encuentra en 1NF, prosigue a ser evaluada para mirar si no se tienen dependencias parciales desde el atributo primo (código). Analizando la clase, una ciudad al tener un código que lo identifique y distinga de las demás ciudades, muestra la dependencia de la otra información que se necesita saber. Por lo tanto, se puede afirmar que la relación está en 2NF.

- **Tercera Forma Normal (3NF):** Como la relación cumple con las dos reglas anteriores, se prosigue a analizar si está en la regla 3NF. En esta se evalúa que no exista alguna dependencia transitiva. Con la llave primaria, un atributo NO determina otro primo. Analizando los atributos junto con la llave primaria, no se ve otra dependencia o relación entre los demás, ya que son características de la ciudad más no son los que guardan la información de ella. Por ejemplo, el código puede diferenciar una ciudad de otra.
 - **Forma Normal Boyce-Codd:** la relación está en 3NF, y adicionalmente las llaves son simples y no dependen de otra llave, se puede decir que se encuentra en BCNF.
-
- **Sucursal:** La definición de relación y de sus dependencias funcionales es la siguiente:
 - Sucursal {id, nombre, tamaño, teléfono, dirección}
 - Llave candidata: Luego de analizar los atributos de la clase, se llega a la siguiente llave:
 - {id}
 - **Primera Forma Normal (1NF):** la relación se encuentra en primera forma normal ya que esta no cuenta con atributos multivariados. Es decir que contengan listas o algún tipo de estructura similar.
 - **Segunda Forma Normal (2NF):** La relación como ya se encuentra en 1NF, prosigue a ser evaluada para mirar si no se tienen dependencias parciales desde el atributo primo (id). Analizando la clase, una sucursal al tener un id que lo identifique y distingue de las demás sucursales, a pesar de tener mismo nombre, muestra la dependencia de la otra información que se necesita saber. Por lo tanto, se puede afirmar que la relación está en 2NF.
 - **Tercera Forma Normal (3NF):**
 - Como la relación cumple con las dos reglas anteriores, se prosigue a analizar si está en la regla 3NF. En esta se evalúa que no exista alguna dependencia transitiva. Con la llave primaria, un atributo NO determina otro primo. Analizando todos los atributos junto con la llave primaria, no se ve otra dependencia o relación que puedan existir entre los demás atributos ya que estos son solo características de la categoría, mas no son los que guardan la información de ella. Por ejemplo, no se puede determinar el nombre de la sucursal sin su id. Pueden que haya sucursales con mismo nombre, pero id diferente.
 - **Forma Normal Boyce-Codd:** la relación está en 3NF, y adicionalmente las llaves son simples y no dependen de otra llave, se puede decir que se encuentra en BCNF.

 - **Proveedor:** La definición de relación y de sus dependencias funcionales es la siguiente:
 - Proveedor {nit, nombre,direccion,nombrePersona,telefonoPersona }
 - Llave candidata: Luego de analizar los atributos de la clase, se llega a la siguiente llave:
 - {nit}
 - **Primera Forma Normal (1NF):** la relación se encuentra en primera forma normal ya que esta no cuenta con atributos multivariados. Es decir que contengan listas o algún tipo de estructura similar.

- **Segunda Forma Normal (2NF):** La relación como ya se encuentra en 1NF, prosigue a ser evaluada para mirar si no se tienen dependencias parciales desde el atributo primo (nit). Analizando la clase, un proveedor al tener un nit que lo identifique y distingue de los demás proveedores, a pesar de tener mismo nombre, muestra la dependencia de la otra información que se necesita saber. Por lo tanto, se puede afirmar que la relación está en 2NF.
 - **Tercera Forma Normal (3NF):** Como la relación cumple con las dos reglas anteriores, se prosigue a analizar si está en la regla 3NF. En esta se evalúa que no exista alguna dependencia transitiva. Con la llave primaria, un atributo NO determina otro primo. Analizando todos los atributos junto con la llave primaria, no se ve otra dependencia o relación que puedan existir entre los demás atributos ya que estos son solo características de la categoría, mas no son los que guardan la información de ella. Por ejemplo, no se puede determinar el nombre del proveedor sin su nit.
 - **Forma Normal Boyce-Codd:** la relación está en 3NF, y adicionalmente las llaves son simples y no dependen de otra llave, se puede decir que se encuentra en BCNF.
-
- **Categoría:** La definición de relación y de sus dependencias funcionales es la siguiente:
 - Categoría {id, nombre, descripción, caracterisicasAlmac}
 - Llave candidata: Luego de analizar los atributos de la clase, se llega a la siguiente llave:
 - {id}
 - **Primera Forma Normal (1NF):** la relación se encuentra en primera forma normal ya que esta no cuenta con atributos multivariados. Es decir que contengan listas o algún tipo de estructura similar.
 - **Segunda Forma Normal (2NF):** La relación como ya se encuentra en 1NF, prosigue a ser evaluada para mirar si no se tienen dependencias parciales desde el atributo primo (id). Analizando la clase, una categoría al tener un id que los identifique y distingue de las demás categorías a pesar de tener mismo nombre, muestra la dependencia de la otra información que se necesita saber. Por lo tanto, se puede afirmar que la relación está en 2NF.
 - **Tercera Forma Normal (3NF):** Como la relación cumple con las dos reglas anteriores, se prosigue a analizar si está en la regla 3NF. En esta se evalúa que no exista alguna dependencia transitiva. Con la llave primaria, un atributo NO determina otro primo. Analizando todos los atributos junto con la llave primaria, no se ve otra dependencia o relación que puedan existir entre los demás atributos ya que estos son solo características de la categoría, mas no son los que guardan la información de ella. Por ejemplo, no se puede determinar una descripción de una categoría sin saber su id.
 - **Forma Normal Boyce-Codd:** la relación está en 3NF, y adicionalmente las llaves son simples y no dependen de otra llave, se puede decir que se encuentra en BCNF.
-
- **Bodega:** La definición de relación y de sus dependencias funcionales es la siguiente:

- Bodega{id,nombre,tamano,capacidad}
 - Llave candidata: Luego de analizar los atributos de la clase, se llega a la siguiente llave:
 - {id}
 - **Primera Forma Normal (1NF):** la relación se encuentra en primera forma normal ya que esta no cuenta con atributos multivariados. Es decir que contengan listas o algún tipo de estructura similar.
 - **Segunda Forma Normal (2NF):** La relación como ya se encuentra en 1NF, prosigue a ser evaluada para mirar si no se tienen dependencias parciales desde el atributo primo id. Analizando la clase, una bodega tiene un identificador único (conocido como id), donde contiene la información detallada de la bodega. Adicional a ello, ningún otro atributo no primo puede ser conocido sin saber la llave primaria seleccionada, ya que esta, como se dijo anteriormente, brinda todos los detalles del lugar. Por lo tanto, se puede afirmar que la relación está en 2NF.
 - **Tercera Forma Normal (3NF):** Como la relación cumple con las dos reglas anteriores, se prosigue a analizar si está en la regla 3NF. En esta se evalúa que no exista alguna dependencia transitiva. Con la llave primaria, un atributo NO determina otro primo. Analizando todos los atributos junto con la llave primaria, no se ve otra dependencia o relación que puedan existir entre los demás atributos ya que estos son solo características de la bodega, mas no son los que guardan la información de ella. Por ejemplo, no se puede determinar el tamaño de la bodega con una llave diferente a la primaria (id).
-
- **Producto:** La definición de relación y de sus dependencias funcionales es la siguiente:
 - Producto {codigoBarra, nombre, costoBodega, costoUnidad, presentación, cantidadPresentacion, unidadMedia}
 - Llave candidata: Luego de analizar los atributos de la clase, se llega a la siguiente llave:
 - {codigoBarra}
 - **Primera Forma Normal (1NF):** la relación se encuentra en primera forma normal ya que esta no cuenta con atributos multivariados. Es decir que contengan listas o algún tipo de estructura similar.
 - **Segunda Forma Normal (2NF):** La relación como ya se encuentra en 1NF, prosigue a ser evaluada para mirar si no se tienen dependencias parciales desde el atributo primo (codigoBarra). Analizando la clase, el código de barras básicamente es el encargado de guardar la información detallada de cada producto. Entonces, al saber cuál es este código, puedo saber su precio, nombre, presentación y demás atributos. Por lo tanto, se puede afirmar que la relación está en 2NF.
 - **Tercera Forma Normal (3NF):** Como la relación cumple con las dos reglas anteriores, se prosigue a analizar si está en la regla 3NF. En esta se evalúa que no exista alguna dependencia transitiva. Con la llave primaria, un atributo NO determina otro primo. Analizando todos los atributos junto con la llave primaria, no se ve otra dependencia o

relación que puedan existir entre los demás atributos ya que estos son solo características del producto, mas no son los que guardan la información de él.

Diseño de la Base de Datos

Ahora, entraremos en el concepto del diseño de la base de datos de nuestra aplicación. En este, se llevará a cabo un proceso de creación de tablas, relaciones y sus respectivas restricciones. Así mismo, se podrán encontrar las reglas de diseño que logramos identificar gracias al Modelo Relacional, en donde se asignaron las FK's (Foreign Keys) a cada una de las tablas que las requerían según la multiplicidad de sus relaciones con otras tablas.

A continuación, se presenta una tabla con cada una de las entidades de nuestro proyecto, la explicación de cómo se diseñó en la base de datos y su respectiva argumentación:

Bodega:

En esta tabla se encuentran columnas como el nombre, el tamaño (tamaño en nuestra BD), y el id de una sucursal. Ninguna de estas debe ser NULL, debido a que es muy importante que al momento de la creación de la tabla Bodega pueda tener todos sus atributos especificados.

Respecto a otras restricciones de las columnas, cada tupla de la tabla va a tener una columna con el 'id', que será la PK (Primary Key) de esta, que será la primera restricción. Por otro lado, se agregó una restricción CHECK para que el 'tamaño' (en m^2) de la Bodega sea si o si mayor a $0m^2$.

Así mismo, teniendo en cuenta que una sucursal tiene muchas bodegas, esta última tendrá una columna 'sucursal_id', que representará el id de la sucursal que tiene asignada cada bodega. Esta columna tendrá dos restricciones, una FK que se referencia a la columna 'id' de la tabla Sucursal, y otra NN (Not Null) que obliga a la tupla a tener una sucursal asignada.

Categoría

En esta tabla se encuentran columnas como el código, nombre, descripción y característicasalmacenaje. Ninguna de estos atributos deben ser NULL ya que son muy importantes a la hora de crear la tabla Categoría y así obtener información relevante de esta.

Respecto a otras restricciones de las columnas, cada tupla de la tabla va a tener una columna con el "código" que será la PK (Primary Key) de esta, que será la primera restricción. Por otro lado, se agregó una restricción UNIQUE para que el nombre de la categoría sea único.

Así mismo, se tiene una relación entre categoría y producto que se explicará posteriormente.

Ciudad

En esta tabla se encuentran columnas como el código y nombre. Ninguna de estos atributos deben ser NULL ya que son muy importantes a la hora de crear la tabla Ciudad y así obtener información relevante de esta.

Respecto a otras restricciones de las columnas, cada tupla de la tabla va a tener una columna con el "código" que será la PK (Primary Key) de esta, que será la primera restricción. Por otro lado, se agregó una restricción UNIQUE para que el nombre de la ciudad sea único.

Adicionalmente, se tiene que el nombre de la ciudad este dentro de la mayoría de ciudades de Colombia, esto usando la restricción CHECK.

Empaque

En esta tabla se encuentran columnas como el id, volumenempaque y pesoempaque. Ninguna de estos atributos debe ser NULL ya que son muy importantes a la hora de crear la tabla de Empaque y así obtener información relevante de esta.

Respecto a otras restricciones de las columnas, cada tupla de la tabla va a tener una columna con el "id" que será la PK (Primary Key) de esta, que será la primera restricción. Por otro lado, los atributos de volumenempaque y pesoempaque deben ser mayor a 0, esto se verifica usando la restricción CHECK.

OrdenCompra

En esta tabla se encuentran columnas como el id, fechaesperada, fechacreacion, estado, sucursal_id y proveedor_nit. Ninguna de estos atributos debe ser NULL ya que son muy importantes a la hora de crear la tabla de OrdenCompra y así obtener información relevante de esta.

Respecto a otras restricciones de las columnas, cada tupla de la tabla va a tener una columna con el "id" que será la PK (Primary Key) de esta,

que será la primera restricción. Por otro lado, el estado de la orden de compra debe estar en Enviada , Recibida, Pagada o Anulada utilizando la restricción CHECK. Al mismo tiempo, se debe revisar que la fecha esperada sea mayor a la fecha de creación de la orden. También usando CHECK.

Así mismo, teniendo en cuenta que una orden de compra tiene un solo proveedor, esta última tendrá una columna 'proveedor_nit', que representará el nit del proveedor que tiene asignada cada orden de compra. Esta columna tendrá dos restricciones, una FK que se referencia a la columna 'nit' de la tabla Proveedor, y otra NN (Not Null) que obliga a la tupla a tener un proveedor asignado. Luego tenemos que varias ordenes de compra se realizan en una sucursal, para ello se establece como la sucursal_id como FK con esta tabla.

Producto

En esta tabla se encuentran columnas como codigobarras,nombre,costobodega,costounidad,presentacion,cantidadpresentacion,unimedida,fechaexpiracion,categoria_codigo y empaque_id. Ninguna de estos atributos debe ser NULL ya que son muy importantes a la hora de crear la tabla de Producto y así obtener información relevante de esta.

Respecto a otras restricciones de las columnas, cada tupla de la tabla va a tener una columna con el "codigobarras" que será la PK (Primary Key) de esta, que será la primera restricción. Por otro lado, el costobodega, costounidad y cantidadpresentacion deben ser mayor a 0 donde esto se comprueba utilizando la restricción CHECK.

Así mismo, teniendo en cuenta que varios productos tienen una categoria, esta última tendrá una columna 'categoria_codigo', que representará el código de la categoria que tiene asignado cada producto. Esta columna tendrá

dos restricciones, una FK que se referencia a la columna 'codigo' de la tabla Categoria, y otra NN (Not Null) que obliga a la tupla a tener una categoria asignada. Luego tenemos que varios productos estan en un empaque , para ello se establece como empaque_id como FK con esta tabla.

ProductoBodega

En esta tabla se encuentran columnas como costopromedio,cantidadproducto,nivelorden,capacidadproducto,producto_codigobarras y bodega_id. Ninguna de estos atributos debe ser NULL ya que son muy importantes a la hora de crear la tabla de Producto bodega y así obtener información relevante de esta.

Así mismo, teniendo en cuenta que varios productos bodegas estan en una bodega, esta última tendrá una columna 'bodega_id', que representará el id de la bodega que tiene asignado cada producto. Esta columna tendrá dos restricciones, una FK que se referencia a la columna 'id' de la tabla Bodega, y otra NN (Not Null) que obliga a la tupla a tener una bodega asignada. Luego tenemos la relacion con el producto, para ello se estable como llave foranea "producto_codigobarras" donde hace referencia al atributo codigobarras de la tabla Producto .

ProductoCompra

En esta tabla se encuentran columnas como precioacordado,cantidad,ordencompra_id y producto_codigobarras. Ninguna de estos atributos debe ser NULL ya que son muy importantes a la hora de crear la tabla de Producto compra y así obtener información relevante de esta.

Así mismo, teniendo en cuenta que varios productos de compra estan en varias ordenes de compra, esta última tendrá una columna 'ordencompra_id', que representará el id de la orden de compra que tiene asignado cada

producto. Esta columna tendrá dos restricciones, una FK que se referencia a la columna 'id' de la tabla Orden compra, y otra NN (Not Null) que obliga a la tupla a tener una orden de compra asignada. Luego tenemos la relación con el producto, para ello se establece como llave foránea "producto_codigobarras" donde hace referencia al atributo codigobarras de la tabla Producto .

Producto Perecedero

En esta tabla se encuentran columnas como codigobarras y fechavencimiento. Ninguna de estos atributos debe ser NULL ya que son muy importantes a la hora de crear la tabla de Producto perecedero y así obtener información relevante de esta.

Respecto a otras restricciones de las columnas, cada tupla de la tabla va a tener una columna con el "codigobarras" que será la PK (Primary Key) de esta.

Luego tenemos la relación con el producto, para ello se establece como llave foránea "codigobarras" donde hace referencia al atributo codigobarras de la tabla Producto .

Proveedor

En esta tabla se encuentran columnas como nit, nombre, dirección, nombre persona y telefonopersona. Ninguna de estos atributos debe ser NULL ya que son muy importantes a la hora de crear la tabla de Proveedor y así obtener información relevante de esta.

Respecto a otras restricciones de las columnas, cada tupla de la tabla va a tener una columna con el "nit" que será la PK (Primary Key) de esta. Adicionalmente, el atributo de telefonopersona tiene dos restricciones. Con CHECK se revisa que los datos de este sean mayores a 0, y que el teléfono sea único usando UNIQUE.

Recepcion

En esta tabla se encuentran columnas como id, fecharecepcion, bodega_id y ordencompra_id. Ninguna de estos atributos debe ser NULL ya que son muy importantes a la hora de crear la tabla de Recepcion y así obtener información relevante de esta.

Respecto a otras restricciones de las columnas, cada tupla de la tabla va a tener una columna con el "id" que sera la PK (Primary Key) de esta. En este caso, utilizamos el UNIQUE INDEX para representar que la orden por cada recepcion sea unica y no se repita.

Así mismo, teniendo en cuenta que una recepcion tiene una orden de compra, esta última tendrá una columna 'ordencompra_id', que representará el id de la orden de compra que tiene asignado cada producto. Esta columna tendrá dos restricciones, una FK que se referencia a la columna 'id' de la tabla Orden compra, y otra NN (Not Null) que obliga a la tupla a tener una orden de compra asignada. Luego tenemos la relacion con bodega, para ello se estable como llave foranea "bodega_id" donde hace refrencia al atributo id de la tabla Bodega .

Producto Proveedor

En esta tabla se encuentran columnas como producto_codigobarras y proveedor_nit. Ninguna de estos atributos debe ser NULL ya que son muy importantes a la hora de crear la tabla de Productoproveedor y así obtener información relevante de esta.

Respecto a otras restricciones de las columnas, cada tupla de la tabla va a tener dos columnas con "producto_codigobarras" y "proveedor_nit" que seran las PK (Primary Key) de esta.

Luego tenemos la relacion con proveedor y producto. Para ello se estable como llaves foraneas "producto_codigobarras" y "proveedor_nit" donde uno hace refrencia al

Sucursal

atributo codigobarras de la tabla Producto y el otro el nit de la tabla Proveedor .

En esta tabla se encuentran columnas como id, nombre, tamano,direccion y ciudad_codigo. Ninguna de estos atributos debe ser NULL ya que son muy importantes a la hora de crear la tabla de Sucursal y así obtener información relevante de esta.

Respecto a otras restricciones de las columnas, cada tupla de la tabla va a tener una columna con el "id" que sera la PK (Primary Key) de esta.Ademas, utilizando CHECK , se va a revisar que el tamano de la sucursal sea mayor a $0m^2$.

Así mismo, teniendo en cuenta que varias sucursales estan en una ciudad, esta última tendrá una columna 'ciudad_codigo', que representará el codigo de la ciudad donde estan las sucursales. Esta columna tendrá dos restricciones, una FK que se referencia a la columna 'codigo' de la tabla Ciudad, y otra NN (Not Null) que obliga a la tupla a tener una ciudad asignada.

Documentación Aplicación

La Base de Datos ya se encuentra estructurada y diseñada. Ahora, debemos diseñar una aplicación con Java Spring, en donde logremos implementar todos los requerimientos funcionales y de consulta. Para esto, creamos las distintas entidades con sus respectivos repository's y controller's para cumplir con los objetivos de la entrega. Para poder hacer lo anterior, nos guiamos exclusivamente de nuestros modelos de datos (UML, E/R y Relacional).

A continuación, se mostrará cada clase 'Java' implementada en nuestra aplicación:

- **Bodega:** Entidad que representa una bodega de SuperAndes donde se almacenan diferentes productos. Cada bodega pertenece solo a una sucursal.

Bodega.java

Clase que representa la entidad Bodega de nuestros modelos de datos. En esta se encuentran los atributos propios como 'id' con etiqueta @Id que sigue una estrategia de secuencia y que referencia la columna "id" de la tabla Bodega en nuestra base de datos, 'nombre' que referencia la columna "nombre" de la misma tabla y 'tamaño' que referencia la columna "tamaño". Así mismo, esta entidad tiene una relación de muchos a 1 con la entidad Sucursal. Es por eso que en esta clase Java se especifica un atributo 'sucursal' con una etiqueta @ManyToOne que representa que muchas bodegas pertenecen a una sola sucursal y que representa la columna 'sucursal_id' que referencia a la columna 'id' de la tabla Sucursal en nuestra tabla de datos. No menos importante, se incluyen dos constructores, uno vacío (por especificación de Hibernate) y otro que recibe el nombre y el tamaño por parámetro. Finalmente se incluyen los Getters y Setters para cada uno de los atributos de la clase.

BodegaRepository.java

Interface que extiende de JpaRepository, que usa la entidad Bodega y un Integer (id de la bodega). En esta interface se implementan los métodos del CRUD (Create, Read, Update, Delete). Específicamente se crean los siguientes métodos:

1. **darBodegas()** que devuelve una Collection de todas las bodegas que se encuentran en la base de datos, usando una etiqueta @Query que tiene una sentencia SQL con valor "SELECT * FROM Bodega",
2. **insertarBodega(String nombre, Integer tamaño, Integer sucursal_id)** permite insertar una nueva bodega en la base de datos. Se utiliza la anotación @Modifying para indicar

que la consulta modifica los datos, y `@Transactional` para asegurar la transacción. El método utiliza una etiqueta `@Query` con la sentencia SQL "INSERT INTO Bodega (id, nombre, tamaño, sucursal_id) VALUES(bodega_sequence.NEXTVAL,:nombre, :tamaño, :sucursal_id)".

3. **eliminarBodega(int id)** método que elimina una bodega de la base de datos basada en su id. También utiliza `@Modifying` y `@Transactional`, y la etiqueta `@Query` contiene la sentencia SQL "DELETE FROM Bodega WHERE id=:id"

BodegaController.java

Clase controladora que expone una API REST para la entidad Bodega. La clase utiliza el repositorio `BodegaRepository` para gestionar las operaciones CRUD de Bodega. A continuación, se describen los endpoints definidos en esta clase:

1. **/bodega (@GetMapping)**: Este endpoint devuelve una `Collection` con todas las bodegas almacenadas en la base de datos. Para obtener estos datos, se llama al método `darBodegas()` del repositorio `BodegaRepository`.
2. **/bodega/new/save (@PostMapping)**: Este endpoint permite crear una nueva Bodega. Recibe un objeto Bodega en el cuerpo de la petición (anotado con `@RequestBody`) y utiliza el método `insertarBodega` del repositorio `BodegaRepository` para guardar la nueva bodega en la base de datos. Si la operación se completa con éxito, se devuelve un `ResponseEntity` con el mensaje "Bodega creada exitosamente" y un estado `HttpStatus.CREATED`. Si ocurre un error, se devuelve un mensaje de error y un estado `HttpStatus.INTERNAL_SERVER_ERROR`.
3. **/bodega/{id}/edit/save (@PostMapping)**: Este endpoint

permite actualizar una Bodega existente en la base de datos. Recibe el id de la bodega como variable de la URL (anotada con `@PathVariable`) y un objeto Bodega en el cuerpo de la petición (`@RequestBody`). Llama al método `actualizarBodega` del repositorio `BodegaRepository` para actualizar los datos de la bodega correspondiente. Si la operación es exitosa, se devuelve un `ResponseEntity` con el mensaje "Bodega actualizada exitosamente" y un estado `HttpStatus.OK`. Si ocurre un error, se devuelve un mensaje de error y un estado

`HttpStatus.INTERNAL_SERVER_ERROR`.

4. **`/bodega/{id}/delete`** (`@GetMapping`):

Este endpoint permite eliminar una Bodega específica de la base de datos, tomando su id como parámetro en la URL (`@PathVariable`). Utiliza el método `eliminarBodega` del repositorio `BodegaRepository` para eliminar la bodega correspondiente. Si la operación se completa con éxito, se devuelve un `ResponseEntity` con el mensaje "Bodega eliminada exitosamente" y un estado `HttpStatus.OK`. Si ocurre un error, se devuelve un mensaje de error y un estado

`HttpStatus.INTERNAL_SERVER_ERROR`.

- **Categoria** : entidad que representa una Categoria de SuperAndes donde diferentes productos estan clasificados en dicha categoria.

Categoria.java

Clase que representa la entidad Categoria de nuestros modelos de datos. En esta se encuentran los atributos propios como 'id' con etiqueta @Id que sigue una estrategia de secuencia y que referencia la columna "id" de la tabla Categoria en nuestra base de datos, 'nombre' que referencia la columna "nombre" de la misma tabla, 'codigo' que referencia la columna "codigo", "descripcion" que referencia la columna "descripcion" y "caracteristicasAlmacen" que referencia la columna "caracteristicasalmacen". No menos importante, se incluyen dos constructores, uno vacío (por especificación de Hibernate) y otro que recibe los parámetros de la entidad. Finalmente se incluyen los Getters y Setters para cada uno de los atributos de la clase.

CategoriaRepository.java

Interfaz que extiende de JpaRepository, que usa la entidad Categoria y un Integer (id de la categoría). En esta interfaz se implementan los métodos del CRUD (Create, Read, Update, Delete). Específicamente se crean los siguientes métodos:

1. **darCategorias()**: Este método devuelve una Collection de todas las categorías que se encuentran en la base de datos. Usa una etiqueta @Query con la sentencia SQL "SELECT * FROM Categoria" para obtener todos los registros de la tabla Categoria.
2. **darCategoriaPorCodigo(int codigo)**: Este método devuelve una instancia de Categoria que corresponde a un codigo específico en la base de datos. Utiliza la etiqueta @Query con la sentencia SQL "SELECT * FROM Categoria WHERE codigo = :codigo"

para recuperar la categoría correspondiente al código proporcionado.

3. **darCategoriaPorNombre(String nombre):** Este método permite obtener una instancia de Categoría a partir de su nombre. Utiliza una etiqueta @Query con la sentencia SQL "SELECT * FROM Categoría WHERE nombre = :nombre" para buscar una categoría por su nombre.
4. **insertarCategoria(String nombre, String descripcion, String característicasAlmacenaje):** Este método inserta una nueva categoría en la base de datos. Utiliza las anotaciones @Modifying y @Transactional para marcar que la consulta modifica los datos y debe ser gestionada en una transacción. La etiqueta @Query contiene la sentencia SQL "INSERT INTO Categoría (codigo, nombre, descripcion, característicasalmacenaje) VALUES(categoría_sequence.NEXTVAL, :nombre, :descripcion, :característicasAlmacenaje)" para realizar la inserción.
5. **actualizarCategoria(int código, String nombre, String descripcion, String característicasAlmacenaje):** Este método permite actualizar una categoría existente. Al igual que en el método anterior, utiliza las anotaciones @Modifying y @Transactional. La consulta SQL está definida en la etiqueta @Query con la sentencia "UPDATE Categoría SET nombre=:nombre, descripcion=:descripcion, característicasalmacenaje=:característicasAlmacenaje WHERE código=:código".

CategoriaController.java

6. **eliminarCategoria(int codigo):** Este método elimina una categoría de la base de datos en función de su código. Utiliza las anotaciones `@Modifying` y `@Transactional` para indicar que es una operación que modifica datos y que debe ejecutarse dentro de una transacción. La consulta SQL está definida en la etiqueta `@Query` con la sentencia "DELETE FROM Categoria WHERE codigo=:codigo".

Clase controladora que expone una API REST para la entidad Categoria. Utiliza el repositorio `CategoriaRepository` para gestionar las operaciones CRUD de Categoria. A continuación, se describen los endpoints definidos en esta clase:

1. **/categoria (@GetMapping):** Este endpoint devuelve una Collection con todas las categorías almacenadas en la base de datos. Para obtener estos datos, se llama al método `darCategorias()` del repositorio `CategoriaRepository`.
2. **/categoria/codigo/{codigo} (@GetMapping):** Este endpoint permite obtener una categoría específica basada en su código (id). Recibe el código como variable de la URL (anotada con `@PathVariable`) y llama al método `darCategoriaPorCodigo()` del repositorio `CategoriaRepository`. Si la categoría es encontrada, devuelve la categoría con un `HttpStatus.OK`. Si no, devuelve un `HttpStatus.NOT_FOUND`. En caso de error, retorna un `HttpStatus.INTERNAL_SERVER_ERROR`.
3. **/categoria/nombre/{nombre} (@GetMapping):** Este endpoint permite obtener una categoría

específica basada en su nombre. Recibe el nombre como variable de la URL (@PathVariable) y utiliza el método darCategoriaPorNombre() del repositorio CategoriaRepository para recuperar la categoría. Si es encontrada, devuelve la categoría con un HttpStatus.OK. Si no, devuelve un HttpStatus.NOT_FOUND. En caso de error, retorna un HttpStatus.INTERNAL_SERVER_ERROR.

4. **/categoria/new/save** (@PostMapping): Este endpoint permite crear una nueva Categoria. Recibe un objeto Categoria en el cuerpo de la petición (anotado con @RequestBody) y llama al método insertarCategoria() del repositorio CategoriaRepository para guardar la nueva categoría en la base de datos. Si la operación se completa con éxito, se devuelve un ResponseEntity con el mensaje "Categoría creada exitosamente" y un estado HttpStatus.CREATED. Si ocurre un error, se devuelve un mensaje de error y un estado HttpStatus.INTERNAL_SERVER_ERROR.
5. **/categoria/{codigo}/edit/save** (@PostMapping): Este endpoint permite actualizar una Categoria existente. Recibe el código de la categoría como variable de la URL (@PathVariable) y un objeto Categoria en el cuerpo de la petición (@RequestBody). Llama al método actualizarCategoria() del repositorio CategoriaRepository para actualizar los datos de la categoría. Si la operación es exitosa, se devuelve un ResponseEntity con el mensaje "Categoría actualizada exitosamente" y un estado HttpStatus.OK. Si ocurre un error, se devuelve un mensaje de

error y un estado
HttpStatus.INTERNAL_SERVER_ERROR.

6. **/categoria/{codigo}/delete**

(@GetMapping): Este endpoint permite eliminar una Categoría específica de la base de datos, tomando su código como parámetro en la URL (@PathVariable). Utiliza el método `eliminarCategoría()` del repositorio `CategoríaRepository` para eliminar la categoría correspondiente. Si la operación se completa con éxito, se devuelve un `ResponseEntity` con el mensaje "Categoría eliminada exitosamente" y un estado `HttpStatus.OK`. Si ocurre un error, se devuelve un mensaje de error y un estado `HttpStatus.INTERNAL_SERVER_ERROR`.

- **Ciudad** : entidad que representa una Ciudad de SuperAndes donde diferentes sucursales están en una ciudad.

Ciudad.java

Clase que representa la entidad Ciudad de nuestros modelos de datos. En esta se encuentran los atributos propios como 'id' con etiqueta @Id que sigue una estrategia de secuencia y que referencia la columna "id" de la tabla Ciudad en nuestra base de datos, 'nombre' que referencia la columna "nombre" de la misma tabla y 'codigo' que referencia la columna "codigo". No menos importante, se incluyen dos constructores, uno vacío (por especificación de Hibernate) y otro que recibe el nombre y el tamaño por parámetro. Finalmente se incluyen los Getters y Setters para cada uno de los atributos de la clase.

CiudadRepository.java

Interfaz que extiende de JpaRepository, que usa la entidad Ciudad y un Integer (id de la ciudad). En esta interfaz se implementan los métodos del CRUD (Create, Read, Update, Delete). Específicamente se crean los siguientes métodos:

1. **darCiudades()**: Este método devuelve una Collection de todas las ciudades que se encuentran en la base de datos. Usa una etiqueta @Query con la sentencia SQL "SELECT * FROM ISIS2304A28202420.ciudad" para obtener todos los registros de la tabla Ciudad.
2. **darCiudad(int codigo)**: Este método devuelve una instancia de Ciudad que corresponde a un código específico en la base de datos. Utiliza la etiqueta @Query con la sentencia SQL "SELECT * FROM Ciudad WHERE codigo = :codigo" para recuperar la ciudad correspondiente al código proporcionado.

3. **insertarCiudad(String nombre):** Este método inserta una nueva ciudad en la base de datos. Utiliza las anotaciones `@Modifying` y `@Transactional` para marcar que la consulta modifica los datos y debe ser gestionada en una transacción. La etiqueta `@Query` contiene la sentencia SQL `"INSERT INTO Ciudad (codigo, nombre) VALUES(ciudad_sequence.NEXTVAL, :nombre)"` para realizar la inserción.
4. **actualizarCiudad(int codigo, String nombre):** Este método permite actualizar una ciudad existente. Al igual que en el método anterior, utiliza las anotaciones `@Modifying` y `@Transactional`. La consulta SQL está definida en la etiqueta `@Query` con la sentencia `"UPDATE Ciudad SET nombre=:nombre WHERE codigo=:codigo"`.
5. **eliminarCiudad(int codigo):** Este método elimina una ciudad de la base de datos en función de su código. Utiliza las anotaciones `@Modifying` y `@Transactional` para indicar que es una operación que modifica datos y que debe ejecutarse dentro de una transacción. La consulta SQL está definida en la etiqueta `@Query` con la sentencia `"DELETE FROM Ciudad"`


CiudadController.java

Clase controladora que expone una API REST para la entidad Ciudad. Utiliza el repositorio `CiudadRepository` para gestionar las operaciones CRUD de Ciudad. A continuación, se describen los endpoints definidos en esta clase:

1. **/ciudad (@GetMapping):** Este endpoint devuelve una `Collection` con todas las ciudades almacenadas en la base de datos. Llama al método `darCiudades()` del repositorio

CiudadRepository para recuperar todos los registros.

2. **/ciudad/new/save (@PostMapping):**
Este endpoint permite crear una nueva Ciudad. Recibe un objeto Ciudad en el cuerpo de la petición (@RequestBody) y llama al método insertarCiudad() del repositorio CiudadRepository para guardar la nueva ciudad en la base de datos. Si la operación se completa con éxito, se devuelve un ResponseEntity con el mensaje "Ciudad creada exitosamente" y un estado HttpStatus.CREATED. Si ocurre un error, se devuelve un mensaje de error y un estado HttpStatus.INTERNAL_SERVER_ERROR.
3. **/ciudad/{codigo}/edit/save (@PostMapping):** Este endpoint permite actualizar una Ciudad existente. Recibe el código de la ciudad como variable en la URL (@PathVariable) y un objeto Ciudad en el cuerpo de la petición (@RequestBody). Llama al método actualizarCiudad() del repositorio CiudadRepository para actualizar los datos de la ciudad. Si la operación es exitosa, se devuelve un ResponseEntity con el mensaje "Ciudad actualizada exitosamente" y un estado HttpStatus.OK. Si ocurre un error, se devuelve un mensaje de error y un estado HttpStatus.INTERNAL_SERVER_ERROR.
4. **/ciudad/{codigo}/delete (@GetMapping):** Este endpoint permite eliminar una Ciudad de la base de datos, tomando su código como parámetro en la URL (@PathVariable). Llama al método eliminarCiudad() del repositorio CiudadRepository para realizar la eliminación. Si la operación es exitosa,



se devuelve un ResponseEntity con el mensaje "Ciudad eliminada exitosamente" y un estado HttpStatus.OK. Si ocurre un error, se devuelve un mensaje de error y un estado HttpStatus.INTERNAL_SERVER_ERROR.

- **Empaque** : entidad que representa un Empaque de SuperAndes donde diferentes sucursales estan en una ciudad.

Empaque.java

Clase que representa la entidad Empaque de nuestros modelos de datos. En esta se encuentran los atributos propios como 'id' con etiqueta @Id que sigue una estrategia de secuencia y que referencia la columna "id" de la tabla Empaque en nuestra base de datos, 'voumenEmpaque' que referencia la columna "volumenempaque" de la misma tabla y 'pesoEmpaque' que referencia la columna "pesoempaque". No menos importante, se incluyen dos constructores, uno vacío (por especificación de Hibernate) y otro que recibe el nombre y el tamaño por parámetro. Finalmente se incluyen los Getters y Setters para cada uno de los atributos de la clase.

EmpaqueRepository.java

Interfaz que extiende de JpaRepository, que utiliza la entidad Empaque y un Integer (id del empaque). En esta interfaz se implementan los métodos del CRUD (Create, Read, Update, Delete). Específicamente se crean los siguientes métodos:


1. **darEmpaques():** Este método devuelve una Collection de todos los empaques que se encuentran en la base de datos. Utiliza una etiqueta @Query con la sentencia SQL "SELECT * FROM Empaque" para obtener todos los registros de la tabla Empaque.
2. **darEmpaque(int id):** Este método devuelve una instancia de Empaque correspondiente a un id específico. Utiliza la etiqueta @Query con la sentencia SQL "SELECT * FROM Empaque WHERE id = :id" para recuperar el empaque correspondiente al id proporcionado.

3. **insertarEmpaque(int volumenEmpaque, int pesoEmpaque):**
Este método inserta un nuevo empaque en la base de datos. Utiliza las anotaciones `@Modifying` y `@Transactional` para indicar que se trata de una operación de escritura y que debe ejecutarse dentro de una transacción. La consulta SQL está definida en la etiqueta `@Query` con la sentencia "INSERT INTO Empaque (id, volumenempaque, pesoempaque) VALUES(empaque_sequence.NEXTVAL, :volumenEmpaque, :pesoEmpaque)".
4. **actualizarEmpaque(int id, int volumenEmpaque, int pesoEmpaque):**
Este método permite actualizar los datos de un empaque existente. Utiliza las anotaciones `@Modifying` y `@Transactional` para las operaciones de escritura dentro de una transacción. La consulta SQL se define en la etiqueta `@Query` con la sentencia "UPDATE Empaque SET volumenempaque=:volumenEmpaque, pesoempaque=:pesoEmpaque WHERE id=:id".
5. **eliminarEmpaque(int id):** Este método elimina un empaque de la base de datos según su id. Utiliza las anotaciones `@Modifying` y `@Transactional` para operaciones de escritura dentro de una transacción. La consulta SQL se define en la etiqueta `@Query` con la sentencia "DELETE FROM Empaque WHERE id=:id".

EmpaqueController.java

Clase que actúa como controlador para gestionar las operaciones CRUD (Create, Read, Update, Delete) relacionadas con la entidad Empaque. Se implementan los siguientes métodos:

1. **empaques()**: Método que maneja peticiones HTTP GET en la ruta `"/empaque"`. Este método devuelve una Collection de todos los empaques presentes en la base de datos, utilizando el método `darEmpaques()` del repositorio `EmpaqueRepository`.
2. **guardarEmpaque(Empaque empaque)**: Método que maneja peticiones HTTP POST en la ruta `"/empaque/new/save"`. Este método guarda un nuevo empaque en la base de datos utilizando los parámetros `volumenEmpaque` y `pesoEmpaque` del objeto `Empaque` recibido en el cuerpo de la petición. Si la operación es exitosa, se retorna un mensaje de éxito con el estado HTTP 201 Created; en caso de error, se retorna un mensaje de error con el estado HTTP 500 Internal Server Error.
3. **editarGuardarEmpaque(int id, Empaque empaque)**: Método que maneja peticiones HTTP POST en la ruta `"/empaque/{id}/edit/save"`. Este método actualiza la información de un empaque existente en la base de datos, identificando el registro por el parámetro `id` y actualizando los valores de `volumenEmpaque` y `pesoEmpaque` proporcionados en el cuerpo de la petición. Si la operación es exitosa, se retorna un mensaje de éxito con el estado HTTP 200 OK; en caso de error, se retorna un mensaje de error con el estado HTTP 500 Internal Server Error.
4. **eliminarEmpaque(int id)**: Método que maneja peticiones HTTP GET en la ruta `"/empaque/{id}/delete"`. Este método elimina un empaque de la base de datos basado en el parámetro `id` proporcionado en la URL. Si la



operación es exitosa, se retorna un mensaje de éxito con el estado HTTP 200 OK; en caso de error, se retorna un mensaje de error con el estado HTTP 500 Internal Server Error.

- **OrdenCompra:** entidad que representa una Orden de Compra de SuperAndes donde diferentes ordenes de compra tienen varios productos, se realizan en una sucursal, tienen un proveedor y tiene o no una recepción de compra.

OrdenCompra.java

Clase que representa la entidad OrdenCompra de nuestros modelos de datos. En esta se encuentran los atributos propios como 'id' con etiqueta @Id que sigue una estrategia de secuencia y que referencia la columna "id" de la tabla OrdenCompra en nuestra base de datos, 'fechaEsperada' que referencia la columna "fechaesperada" de la misma tabla, 'fechaCreacion' que referencia la columna "fechacreacion" y "estado" que hace referencia a la columna "estado". Así mismo, esta entidad tiene una relacion de muchos a 1 con la entidad Sucursal. Es por eso que en esta clase Java se especifica un atributo 'sucursal' con una etiqueta @ManyToOne que representa que muchas ordenes de compra se realizan en una sola sucursal y que representa la columna 'sucursal_id' que referencia a la columna 'id' de la tabla Sucursal en nuestra tabla de datos. Tambien esta entidad tiene una relacion de muchos a 1 con la entidad Proveedor. Es por eso que en esta clase Java se especifica un atributo 'proveedor' con una etiqueta @ManyToOne que representa que muchas ordenes de compra las realiza un solo proveedor y que representa la columna 'proveedor_nit' que referencia a la columna 'nit' de la tabla Proveedor en nuestra tabla de datos. No menos importante, se incluyen dos constructores, uno vacío (por especificación de Hibernate) y otro que recibe el nombre y el tamaño por parámetro. Finalmente se incluyen los Getters y Setters para cada uno de los atributos de la clase.

OrdenCompraRepository.java

Es una interfaz que extiende de JpaRepository, utilizando la entidad

OrdenCompra y un Integer como identificador (id de la orden de compra). En esta interfaz se implementan los métodos para las operaciones CRUD (Create, Read, Update, Delete). A continuación, se describen los métodos específicos:

1. **darOrdenesCompra()**: Este método devuelve una colección de todas las órdenes de compra que se encuentran en la base de datos. Utiliza una consulta SQL nativa para seleccionar todas las filas de la tabla OrdenCompra.
2. **darOrdenCompra(int id)**: Devuelve una orden de compra específica que corresponde al id proporcionado. Usa una consulta SQL nativa para buscar la orden de compra por su identificador.
3. **insertarOrdenCompra(Date fechaEsperada, Date fechaCreacion, String estado, Integer sucursal_id, Integer proveedor_nit)**: Inserta una nueva orden de compra en la base de datos. El id de la orden de compra se genera automáticamente utilizando una secuencia. Los parámetros que se necesitan para crear la orden incluyen la fecha esperada, la fecha de creación, el estado de la orden, el id de la sucursal y el número de identificación tributaria del proveedor.
4. **actualizarOrdenCompra(int id, String estado)**: Actualiza el estado de una orden de compra existente. La orden se identifica por su id, y se actualiza el campo de estado según el valor proporcionado.
5. **eliminarOrdenCompra(int id)**: Elimina una orden de compra de la base de datos utilizando su id como parámetro. Este método borra el

registro correspondiente en la tabla de órdenes de compra.

OrdenCompraController.java

Este controlador maneja las solicitudes HTTP relacionadas con la entidad OrdenCompra. Está anotado con `@RestController` para que Spring lo detecte como un controlador REST. El controlador utiliza un repositorio `OrdenCompraRepository` para interactuar con la base de datos y realizar las operaciones CRUD.

1. **ordenesCompra():** Maneja las solicitudes GET para obtener una lista de todas las órdenes de compra. Devuelve una colección de `OrdenCompra` consultada desde la base de datos mediante el repositorio `ordenCompraRepository`.
2. **guardarOrdenCompra():** Este método maneja las solicitudes POST para crear una nueva orden de compra. Recibe un objeto `OrdenCompra` como cuerpo de la solicitud. Almacena una nueva orden de compra en la base de datos utilizando el método `insertarOrdenCompra()` del repositorio. El estado inicial de la orden es "Vigente", y la fecha de creación se establece como la fecha actual del sistema. Si la operación tiene éxito, devuelve un mensaje indicando que la orden de compra fue creada exitosamente. En caso de error, responde con un mensaje de error y un estado HTTP 500 (Error Interno del Servidor).
3. **editarGuardarOrdenCompra():** Maneja las solicitudes POST para actualizar una orden de compra existente. El método toma el id de la orden en la URL y un objeto `OrdenCompra` en el

cuerpo de la solicitud. Antes de actualizar, el método realiza validaciones:

- Si la orden está en estado "Entregada", no se permite modificarla.
- Si la orden se desea anular y no está en estado "Vigente", la anulación no es válida. Si se cumplen las condiciones, el método actualiza el estado de la orden de compra. Devuelve un mensaje indicando el éxito de la operación o un error en caso de que la orden no se pueda actualizar.

4. **eliminarOrdenCompra()**: Maneja las solicitudes GET para eliminar una orden de compra. Toma el id de la orden en la URL y, si la operación es exitosa, responde con un mensaje que indica que la orden de compra fue eliminada correctamente. Si ocurre algún error, responde con un mensaje de error y un estado HTTP 500.

- **Producto**: entidad que representa un Producto de SuperAndes donde diferentes productos tiene varias ordenes de compra, estan en varios empaque, puede ser o no un producto perecedero, varios productos estan en varias bodegas, varios proveedores pueden ofrecer varios productos y varios de estos pueden pertenecer a una categoria.

Producto.java

Clase que representa la entidad Producto de nuestros modelos de datos. En esta se encuentran los atributos propios como 'id' con etiqueta @Id que sigue una estrategia de secuencia y que referencia la columna "id" de la tabla Producto en nuestra base de datos, 'nombre' que referencia la columna "nombre" de la misma tabla, 'costoBodega' que referencia la columna "costobodega", "costoUnidad" que hace referencia a la columna "costounidad", "presentacion" que hace referencia a "presentacion", "cantidadPresentacion" que hace referencia a "cantidadpresentacion", "unidadMedida" que hace referencia "unidadmedida" y "fechaExpiracion" que hace referencia a "fechaexpiracion". Así mismo, esta entidad tiene una relacion de muchos a 1 con la entidad Categoria. Es por eso que en esta clase Java se especifica un atributo 'categoria' con una etiqueta @ManyToOne que representa que muchos productos estan en una categoria y que representa la columna 'categoria_codigo' que referencia a la columna 'codigo' de la tabla Categoria en nuestra tabla de datos. Tambien esta entidad tiene una relacion de muchos a 1 con la entidad Empaque. Es por eso que en esta clase Java se especifica un atributo 'empaque' con una etiqueta @ManyToOne que representa que muchos productos pueden estar en un empaque y representa la columna 'empaque_id' que referencia a la columna 'id' de la tabla Empaque en nuestra tabla de datos. No menos importante, se incluyen dos constructores, uno vacío (por especificación de Hibernate) y otro que recibe el nombre y el tamaño por parámetro. Finalmente se incluyen los

Getters y Setters para cada uno de los atributos de la clase.

ProductoRepository.java

Esta interfaz es un repositorio para la entidad Producto, extendiendo JpaRepository y proporcionando métodos para realizar las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los productos en la base de datos.

1. **darProductos():** Este método está anotado con @Query, y se utiliza para recuperar todos los productos de la base de datos. La consulta SQL que se ejecuta es "SELECT * FROM Producto", y devuelve una colección de objetos Producto.
2. **darProductoCodigo():** Este método también utiliza una anotación @Query para buscar un producto específico en la base de datos mediante su código de barras. La consulta SQL busca el producto cuyo campo codigobarras coincide con el valor proporcionado. Devuelve un objeto Producto.
3. **darProductoNombre():** Similar al método anterior, este método busca un producto en la base de datos utilizando su nombre. Utiliza @Query para ejecutar una consulta SQL que busca el producto cuyo campo nombre coincide con el valor dado.
4. **darProducto():** Este método busca un producto en la base de datos utilizando su código de barras, como en el método darProductoCodigo(), y también devuelve un objeto Producto.
5. **insertarProducto():** Este método se utiliza para crear un nuevo producto en la base de datos. Está anotado con @Modifying y @Transactional, lo que indica que modifica los datos en la base de datos. La consulta SQL inserta

un nuevo registro en la tabla Producto con los valores proporcionados en los parámetros del método.

6. **actualizarProducto()**: Este método se utiliza para actualizar un producto existente en la base de datos. Al igual que insertarProducto(), está anotado con @Modifying y @Transactional. La consulta SQL actualiza los campos del producto que coincide con el código de barras especificado.
7. **eliminarProducto()**: Este método se utiliza para eliminar un producto de la base de datos mediante su código de barras. También está anotado con @Modifying y @Transactional, lo que indica que modifica los datos en la base de datos. La consulta SQL elimina el registro de la tabla Producto cuyo campo codigobarras coincide con el valor proporcionado.

ProductoController.java

Esta clase es un controlador para manejar las operaciones relacionadas con la entidad Producto. Utiliza la anotación @RestController, lo que indica que se trata de un controlador de tipo REST en una aplicación Spring.

1. **Inyección de Dependencias**: Se utiliza la anotación @Autowired para inyectar una instancia de ProductoRepository, que permite interactuar con la base de datos y realizar operaciones CRUD sobre los productos.
2. **productos()**: Este método está mapeado a la URL /producto mediante la anotación @GetMapping. Cuando se invoca, se llama al método darProductos() del repositorio, que devuelve una colección de todos los productos en la base de datos.

3. **darProductoCodigo()**: Este método permite buscar un producto específico por su código de barras. Está mapeado a la URL `/producto/codigobarras/{codigoBarras}`. Si se encuentra el producto, se devuelve con un estado HTTP 200 (OK); de lo contrario, se devuelve un estado HTTP 404 (NOT FOUND).
4. **darProductoNombre()**: Similar al método anterior, este método busca un producto por su nombre. Está mapeado a la URL `/producto/nombre/{nombre}` y devuelve el producto encontrado o un estado HTTP 404 si no se encuentra.
5. **guardarProducto()**: Este método se utiliza para crear un nuevo producto. Está mapeado a la URL `/producto/new/save` mediante la anotación `@PostMapping`. Toma un objeto `Producto` en el cuerpo de la solicitud, llama al método `insertarProducto()` del repositorio y devuelve un estado HTTP 201 (CREATED) si la creación es exitosa; en caso contrario, devuelve un estado HTTP 500 (INTERNAL SERVER ERROR).
6. **editarGuardarProducto()**: Este método se utiliza para actualizar un producto existente. Está mapeado a la URL `/producto/{codigoBarras}/edit/save`. Se busca el producto mediante su código de barras y, si se encuentra, se actualizan sus detalles. Si la actualización es exitosa, devuelve un estado HTTP 200; de lo contrario, devuelve un estado HTTP 500.
7. **eliminarProducto()**: Este método permite eliminar un producto de la base de datos mediante su código de barras. Está mapeado a la URL `/producto/{codigoBarras}/delete`. Si la

eliminación es exitosa, devuelve un

- **ProductoBodega:** entidad que representa un Producto bodega de SuperAndes donde muchos de estos estan en varias bodegas.

ProductoBodega.java

Clase que representa la entidad ProductoBodega de nuestros modelos de datos. En esta se encuentran los atributos propios como , 'costoPromedio' que referencia la columna "costopromedio" de la misma tabla , 'cantidadProducto' que referencia la columna "cantidadproducto", "nivelReorden" que hace referencia a la columna "nivelreorden" y "capacidadProducto" que hace referencia a "capacidadproducto". No menos importante, se incluyen dos constructores, uno vacío (por especificación de Hibernate) y otro que recibe el nombre y el tamaño por parámetro. Finalmente se incluyen los Getters y Setters para cada uno de los atributos de la clase.

ProductoBodegaRepository.java

La interfaz ProductoBodegaRepository extiende de JpaRepository, que utiliza la entidad ProductoBodega. En esta interfaz se implementan los métodos del CRUD (Create, Read, Update, Delete) de la siguiente manera:

1. **darProductosBodega():** Este método devuelve una Collection de todos los productos bodegas que se encuentran en la base de datos, usando una etiqueta @Query con la sentencia SQL `SELECT * FROM productobodega`.
2. **darProductoBodega(int productoCodigo, int bodegaid):** Este método devuelve un objeto ProductoBodega específico que corresponde a un código de barras de producto y un ID de bodega, utilizando una etiqueta @Query con la sentencia SQL `SELECT * FROM productobodega WHERE producto_codigobarras =`

:productoCodigo AND bodega_id = :bodegalId.

3. **insertarProductoBodega(int productoCodigo, int bodegalId, int costoPromedio, int cantidadProducto, int nivelReorden, int capacidadProducto)**: Este método inserta un nuevo producto bodega en la base de datos, utilizando una etiqueta @Modifying, @Transactional, y una etiqueta @Query con la sentencia SQL INSERT INTO productobodega (producto_codigobarras, bodega_id, costopromedio, cantidadproducto, nivelreorden, capacidadproducto) VALUES(:productoCodigo, :bodegalId, :costoPromedio, :cantidadProducto, :nivelReorden, :capacidadProducto)
4. **actualizarProductoBodega(int productoCodigo, int bodegalId, int costoPromedio, int cantidadProducto, int nivelReorden, int capacidadProducto)**: Este método actualiza los datos de un producto bodega existente en la base de datos, utilizando una etiqueta @Modifying, @Transactional, y una etiqueta @Query con la sentencia SQL UPDATE productobodega SET costopromedio=:costoPromedio, cantidadproducto=:cantidadProducto, nivelreorden=:nivelReorden, capacidadproducto=:capacidadProducto WHERE producto_codigobarras=:productoCodigo AND bodega_id=:bodegalId
5. **eliminarProductoBodega(int productoCodigo, int bodegalId)**: Este método elimina un producto bodega de la base de datos según el código de barras del producto y el ID de la bodega, utilizando una etiqueta

@Modifying, @Transactional, y una etiqueta @Query con la sentencia SQL DELETE FROM productobodega WHERE

ProductoBodegaController.java

La clase ProductoBodegaController es un controlador REST que gestiona las operaciones relacionadas con la entidad ProductoBodega. A continuación se describen sus métodos:

1. **productosBodega()**: Este método devuelve una colección de todos los productos en bodega disponibles en la base de datos. Se accede a través de una solicitud GET en el endpoint /productobodega, utilizando el método darProductosBodega() del repositorio ProductoBodegaRepository para obtener la información.
2. **guardarProductoBodega(ProductoBodega productoBodega)**: Este método se utiliza para guardar un nuevo producto en bodega. Se invoca mediante una solicitud POST en el endpoint /productobodega/new/save, y recibe un objeto ProductoBodega en el cuerpo de la solicitud. Utiliza el repositorio para llamar al método insertarProductoBodega() y guardar el nuevo producto en la base de datos. Si la operación es exitosa, devuelve un mensaje de éxito con el estado HTTP 201 Created; en caso contrario, captura cualquier excepción y devuelve un mensaje de error con el estado 500 Internal Server Error.
3. **editarGuardarProductoBodega(int producto_codigobarras, int bodega_id, ProductoBodega productoBodega)**: Este método permite actualizar un producto en bodega existente. Se accede a través

de una solicitud POST en el endpoint `/productobodega/{producto_codigobarras}/{bodega_id}/edit/save`, donde se pasan el código de barras del producto y el ID de la bodega como variables de ruta. Llama al método `actualizarProductoBodega()` del repositorio para realizar la actualización. Si la operación es exitosa, devuelve un mensaje de éxito con el estado 200 OK; de lo contrario, maneja cualquier excepción y devuelve un mensaje de error con el estado 500 Internal Server Error.

4. **`eliminarProductoBodega(int producto_codigobarras, int bodega_id)`**: Este método se utiliza para eliminar un producto en bodega específico. Se activa mediante una solicitud GET en el endpoint `/productobodega/{producto_codigobarras}/{bodega_id}/delete`. Utiliza el repositorio para llamar al método `eliminarProductoBodega()`, pasando el código de barras del producto y el ID de la bodega. Si la eliminación es exitosa, devuelve un mensaje confirmando la eliminación con el estado 200 OK; si ocurre un error, devuelve un mensaje de error con el estado 500 Internal Server Error.

- **ProductoCompra**: entidad que representa un Producto de Compra de SuperAndes donde diferentes productos de compra estan en varias ordenes de compra .

ProductoCompra.java

Clase que representa la entidad ProductoCompra de nuestros modelos de datos. En esta se encuentran los atributos propios como , 'precioAcordado' que referencia la columna "precioacordado" de la misma tabla ,y 'cantidad' que referencia la columna "cantidad". No menos importante, se incluyen dos constructores, uno vacío (por especificación de Hibernate) y otro que recibe el nombre y el tamaño por parámetro. Finalmente se incluyen los Getters y Setters para cada uno de los atributos de la clase.

ProductoCompraRepository.java

La interfaz ProductoCompraRepository extiende de JpaRepository, utilizando la entidad ProductoCompra. En esta interfaz se implementan los métodos del CRUD (Create, Read, Update, Delete) de la siguiente manera:

1. **darProductosCompra():** Este método devuelve una Collection de todos los productos de compra que se encuentran en la base de datos, usando una etiqueta @Query que contiene la sentencia SQL `SELECT * FROM productocompra`.
2. **darProductoCompra(int productoCodigo, int ordenCompraId):** Este método devuelve un objeto ProductoCompra específico que corresponde a un código de barras de producto y un ID de orden de compra, utilizando una etiqueta @Query con la sentencia SQL `SELECT * FROM productocompra WHERE producto_codigobarras = :productoCodigo AND ordencompra_id = :ordenCompraId`.

3. **insertarProductoCompra(int productoCodigo, int ordenCompralId, int precioAcordado, int cantidad):** Este método inserta un nuevo producto de compra en la base de datos. Utiliza una etiqueta @Modifying, @Transactional, y una etiqueta @Query con la sentencia SQL INSERT INTO productocompra (producto_codigobarras, ordencompra_id, precioacordado, cantidad) VALUES(:productoCodigo, :ordenCompralId, :precioAcordado, :cantidad).
4. **actualizarProductoCompra(int productoCodigo, int ordenCompralId, int precioAcordado, int cantidad):** Este método actualiza los datos de un producto de compra existente en la base de datos. Utiliza una etiqueta @Modifying, @Transactional, y una etiqueta @Query con la sentencia SQL UPDATE productocompra SET precioacordado=:precioAcordado, cantidad=:cantidad WHERE producto_codigobarras=:productoCodigo AND ordencompra_id=:ordenCompralId.
5. **eliminarProductoCompra(int productoCodigo, int ordenCompralId):** Este método elimina un producto de compra de la base de datos según el código de barras del producto y el ID de la orden de compra. Utiliza una etiqueta @Modifying, @Transactional, y una etiqueta @Query con la sentencia SQL DELETE FROM productocompra WHERE producto_codigobarras=:productoCodigo AND

operaciones relacionadas con la entidad ProductoBodega. A continuación se describen sus métodos:

1. **productosBodega()**: Este método devuelve una colección de todos los productos en bodega disponibles en la base de datos. Se accede a través de una solicitud GET en el endpoint `/productobodega`, utilizando el método `darProductosBodega()` del repositorio `ProductoBodegaRepository` para obtener la información.
2. **guardarProductoBodega(ProductoBodega productoBodega)**: Este método se utiliza para guardar un nuevo producto en bodega. Se invoca mediante una solicitud POST en el endpoint `/productobodega/new/save`, y recibe un objeto `ProductoBodega` en el cuerpo de la solicitud. Utiliza el repositorio para llamar al método `insertarProductoBodega()` y guardar el nuevo producto en la base de datos. Si la operación es exitosa, devuelve un mensaje de éxito con el estado HTTP 201 Created; en caso contrario, captura cualquier excepción y devuelve un mensaje de error con el estado 500 Internal Server Error.
3. **editarGuardarProductoBodega(int producto_codigobarras, int bodega_id, ProductoBodega productoBodega)**: Este método permite actualizar un producto en bodega existente. Se accede a través de una solicitud POST en el endpoint `/productobodega/{producto_codigobarras}/{bodega_id}/edit/save`, donde se pasan el código de barras del producto y el ID de la bodega como variables de ruta. Llama al método `actualizarProductoBodega()` del repositorio para realizar la

actualización. Si la operación es exitosa, devuelve un mensaje de éxito con el estado 200 OK; de lo contrario, maneja cualquier excepción y devuelve un mensaje de error con el estado 500 Internal Server Error.

4. **eliminarProductoBodega(int producto_codigobarras, int bodega_id)**: Este método se utiliza para eliminar un producto en bodega específico. Se activa mediante una solicitud GET en el endpoint `/productobodega/{producto_codigobarras}/{bodega_id}/delete`. Utiliza el repositorio para llamar al método `eliminarProductoBodega()`, pasando el código de barras del producto y el ID de la bodega. Si la eliminación es exitosa, devuelve un mensaje confirmando la eliminación con el estado 200 OK; si ocurre un error, devuelve un mensaje de error con el estado 500 Internal Server Error.

- **ProductoPerecedero**: entidad que representa un Producto perecedero de SuperAndes donde un producto puede ser o no un producto perecedero .

ProductoPerecedero.java

Clase que representa la entidad ProductoPerecedero de nuestros modelos de datos. En esta se encuentran los atributos propios como , 'fechaVencimiento' que representa a la columna "fechavencimiento" y su pk que es "codigobarras". No menos importante, se incluyen dos constructores, uno vacío (por especificación de Hibernate) y otro que recibe el nombre y el tamaño por parámetro. Finalmente se incluyen los Getters y Setters para cada uno de los atributos de la clase.

ProductoPerecederoRepository.java

La interfaz ProductoPerecederoRepository extiende de JpaRepository, utilizando la entidad ProductoPerecedero. En esta interfaz se implementan los métodos del CRUD (Create, Read, Update, Delete) de la siguiente manera:

1. **darProductosPerecederos():** Devuelve una Collection de todos los productos perecederos que se encuentran en la base de datos. Utiliza una etiqueta @Query con la sentencia SQL: `SELECT * FROM ProductoPerecedero`.
2. **daProductoPerecedero(Integer producto_codigobarras):** Devuelve un objeto ProductoPerecedero específico que corresponde a un código de barras de producto. Utiliza una etiqueta @Query con la sentencia SQL: `SELECT * FROM ProductoPerecedero WHERE producto_codigobarras=:producto_codigobarras`.
3. **insertarProductoPerecedero(Integer producto_codigobarras, Date fechavencimiento):** Inserta un nuevo producto perecedero en la base de


datos.Utiliza una etiqueta @Modifying, @Transactional, y una etiqueta @Query con la sentencia SQL: INSERT INTO ProductoPerecedero (producto_codigobarra, fechavencimiento) VALUES(:producto_codigobarras, :fechavencimiento).

4. **actualizarProductoPerecedero(Integer producto_codigobarras, Date fechavencimiento):**Actualiza la fecha de vencimiento de un producto perecedero existente en la base de datos.Utiliza una etiqueta @Modifying, @Transactional, y una etiqueta @Query con la sentencia SQL: UPDATE ProductoPerecedero SET fechavencimiento=:fechavencimiento WHERE producto_codigobarras=:producto_codigobarras.
5. **eliminarProductoPerecedero(int producto_codigobarras):**Elimina un producto perecedero de la base de datos según su código de barras.Utiliza una etiqueta @Modifying, @Transactional, y una etiqueta @Query con la sentencia SQL: DELETE FROM ProductoPerecedero WHERE producto_codigobarras=:producto_codigobarras.

ProductoBodegaController.java

La clase ProductoPerecederoController es un controlador REST que gestiona las solicitudes relacionadas con los productos perecederos. Esta clase implementa métodos para acceder a los datos de productos perecederos mediante el uso de la interfaz ProductoPerecederoRepository. Los métodos definidos son:

1. **productos():**Método que se mapea a la ruta /productoperecedero.Devuelve



una Collection de todos los productos perecederos que se encuentran en la base de datos.Utiliza el método darProductosPerecederos() de ProductoPerecederoRepository para obtener la colección.La respuesta se serializa en formato JSON automáticamente gracias a la anotación @RestController.

- **ProductoProveedor:** entidad que representa un Producto proveedor de SuperAndes donde un producto puede ser ofrecido por un proveedor .

ProductoProveedor.java

Clase que representa la entidad ProductoProveedor de nuestros modelos de datos. En esta solo se encuentra como pk el código de barras del producto , "producto_codigobarras" y el nit del proveedor,"proveedor_nit". No menos importante, se incluyen dos constructores, uno vacío (por especificación de Hibernate) y otro que recibe el nombre y el tamaño por parámetro. Finalmente se incluyen los Getters y Setters para cada uno de los atributos de la clase.

ProductoProveedorRepository.java

La interfaz ProductoProveedorRepository extiende de JpaRepository, utilizando la entidad ProductoProveedor. En esta interfaz se implementan los métodos del CRUD (Create, Read, Update, Delete) de la siguiente manera:

1. **darProductosProveedores():**Devuelve una Collection de todos los productos proveedores que se encuentran en la base de datos. Utiliza una etiqueta @Query con la sentencia SQL: SELECT * FROM productoproveedor.
2. **darProductoProveedor(int productoCodigo, int proveedorNit):**Devuelve un objeto ProductoProveedor específico que corresponde a un código de barras de producto y un NIT de proveedor.Utiliza una etiqueta @Query con la sentencia SQL: SELECT * FROM productoproveedor WHERE producto_codigobarras = :productoCodigo AND proveedor_nit = :proveedorNit.
3. **insertarProductoProveedor(int productoCodigo, int proveedorNit):**Inserta un nuevo

producto proveedor en la base de datos. Utiliza una etiqueta `@Modifying`, `@Transactional`, y una etiqueta `@Query` con la sentencia SQL: `INSERT INTO productoproveedor (producto_codigobarras, proveedor_nit) VALUES(:productoCodigo, :proveedorNit).`

4. **eliminarProductoProveedor(int productoCodigo, int proveedorNit):** Elimina un producto proveedor de la base de datos según su código de barras y NIT de proveedor. Utiliza una etiqueta `@Modifying`, `@Transactional`, y una etiqueta `@Query` con la sentencia SQL: `DELETE FROM productoproveedor WHERE producto_codigobarras = :productoCodigo AND proveedor_nit = :proveedorNit.`

ProductoProveedorController.java

La clase `ProductoProveedorController` es un controlador REST que gestiona las solicitudes relacionadas con los productos proveedores. Esta clase implementa métodos para acceder y manipular los datos de productos proveedores a través de la interfaz

`ProductoProveedorRepository`. Los métodos definidos son:

1. **productosProveedor():** Método que se mapea a la ruta `/productoproveedor`. Devuelve una `Collection` de todos los productos proveedores que se encuentran en la base de datos. Utiliza el método `darProductosProveedores()` de `ProductoProveedorRepository` para obtener la colección.
2. **guardarProductoProveedor(ProductoProveedor productoProveedor):** Método que se mapea a la ruta

/productoproveedor/new/save. Recibe un objeto ProductoProveedor en el cuerpo de la solicitud para crear un nuevo producto proveedor en la base de datos. Utiliza el método insertarProductoProveedor() de ProductoProveedorRepository para realizar la inserción. Retorna un mensaje de éxito si la operación se completa correctamente, o un mensaje de error en caso de fallo.

3. **eliminarProductoProveedor(int producto_codigo barras, int proveedor_nit):** Método que se mapea a la ruta /productoproveedor/{producto_codigo barras}/{proveedor_nit}/delete. Recibe el código de barras del producto y el NIT del proveedor como parámetros de la ruta para eliminar un producto proveedor específico de la base de datos. Utiliza el método eliminarProductoProveedor() de ProductoProveedorRepository para realizar la eliminación. Retorna un mensaje de éxito si la operación se completa correctamente, o un mensaje de error en caso de fallo.

- **Proveedor:** entidad que representa un Proveedor de SuperAndes donde varios proveedores pueden adquirir varios productos y un proveedor puede realizar varias ordenes de compra .

Proveedor.java

Clase que representa la entidad Proveedor de nuestros modelos de datos. En esta se encuentra los atributos de "nit" para representar la columna "nit" y a su vez sera nuestro id , "nombre" para representar la columna "nombre", "direccion" para representar la columna "direccion", "nombrePersona" para representar la columna "nombrepersona" y "telefonoPersona" para representar la columna "telepersona". No menos importante, se incluyen dos constructores, uno vacío (por especificación de Hibernate) y otro que recibe el nombre y el tamaño por parámetro. Finalmente se incluyen los Getters y Setters para cada uno de los atributos de la clase.

ProveedorRepository.java

La interfaz ProveedorRepository extiende de JpaRepository y está asociada a la entidad Proveedor. Esta interfaz implementa los métodos del CRUD (Crear, Leer, Actualizar, Eliminar) para gestionar los datos de proveedores en la base de datos. Los métodos definidos son:

1. **darProveedores():**Método que devuelve una Collection de todos los proveedores que se encuentran en la base de datos.Utiliza la etiqueta @Query con la sentencia SQL SELECT * FROM proveedor para obtener todos los registros de la tabla proveedor.
2. **darProveedor(int nit):**Método que devuelve un objeto Proveedor específico, basado en el NIT proporcionado.Utiliza la etiqueta @Query con la sentencia SQL SELECT * FROM proveedor WHERE nit = :nit para buscar el proveedor con el NIT dado.El

parámetro nit se pasa a través de @Param.

3. **insertarProveedor(int nit, String nombre, String direccion, String nombrePersona, int telefonoPersona):**Método que permite crear un nuevo proveedor en la base de datos.Utiliza la etiqueta @Modifying y @Transactional para realizar la operación de inserción.Usa la sentencia SQL INSERT INTO proveedor (nit, nombre, direccion, nombrepersona, telefonopersona) VALUES(:nit, :nombre, :direccion, :nombrePersona, :telefonoPersona) para insertar los datos del proveedor.Los parámetros se pasan a través de @Param.
4. **actualizarProveedor(int nit, String nombre, String direccion, String nombrePersona, int telefonoPersona):**Método que permite actualizar un proveedor existente en la base de datos.Utiliza la etiqueta @Modifying y @Transactional para realizar la operación de actualización.Usa la sentencia SQL UPDATE proveedor SET nombre=:nombre, direccion=:direccion, nombrepersona=:nombrePersona, telefonopersona=:telefonoPersona WHERE nit=:nit para modificar los datos del proveedor.Los parámetros se pasan a través de @Param.
5. **eliminarProveedor(int nit):**Método que permite eliminar un proveedor de la base de datos, basado en el NIT proporcionado.Utiliza la etiqueta @Modifying y @Transactional para realizar la operación de eliminación.Usa la sentencia SQL DELETE FROM proveedor WHERE

nit=:nit para eliminar el registro del

ProveedorController.java

La clase ProveedorController es un controlador REST que gestiona las operaciones relacionadas con los proveedores en la aplicación. Utiliza la interfaz ProveedorRepository para realizar las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos. Los métodos definidos son:

1. **proveedores():**Método que maneja las solicitudes GET a la ruta /proveedor.Devuelve una Collection de todos los proveedores disponibles en la base de datos utilizando el método darProveedores() del repositorio ProveedorRepository.
2. **guardarProveedor(@RequestBody Proveedor proveedor):**Método que maneja las solicitudes POST a la ruta /proveedor/new/save.Recibe un objeto Proveedor en el cuerpo de la solicitud y utiliza el método insertarProveedor() del repositorio para crear un nuevo proveedor en la base de datos.Devuelve una respuesta con el estado HTTP 201 Created si la creación es exitosa, o 500 Internal Server Error si ocurre un error.
3. **editarGuardarProveedor(@PathVariable("nit") int nit, @RequestBody Proveedor proveedor):**Método que maneja las solicitudes POST a la ruta /proveedor/{nit}/edit/save, donde {nit} es el NIT del proveedor a actualizar.Recibe un objeto Proveedor en el cuerpo de la solicitud y utiliza el método actualizarProveedor() del repositorio para actualizar los datos del proveedor correspondiente en la base de datos.Devuelve una respuesta con el estado HTTP 200 OK si la

actualización es exitosa, o 500 Internal Server Error si ocurre un error.

4. **eliminarProveedor(@PathVariable("nit") int nit):**Método que maneja las solicitudes GET a la ruta /proveedor/{nit}/delete, donde {nit} es el NIT del proveedor a eliminar.Utiliza el método eliminarProveedor() del repositorio para eliminar el proveedor correspondiente en la base de datos.Devuelve una respuesta con el estado HTTP 200 OK si la eliminación es exitosa, o 500 Internal Server Error si ocurre un error.

- **Recepcion:** entidad que representa una Recepcion de SuperAndes donde una recepcion puede tener una orden de compra y varias recepciones pueden estar en una bodega .

Recepcion.java

Clase que representa la entidad Recepcion de nuestros modelos de datos. En esta se encuentra los atributos de "id" para representar la columna "id" y "fechaRecepcion" para representar la columna "fecharecepcion". Así mismo, esta entidad tiene una relacion de muchos a 1 con la entidad Bodega. Es por eso que en esta clase Java se especifica un atributo 'bodega' con una etiqueta @ManyToOne que representa que varias recepciones pertenecen a una sola bodega y que representa la columna 'bodega_id' que referencia a la columna 'id' de la tabla Bodega en nuestra tabla de datos. Tambien se tiene que una recepcion pertenece a solo una orden de compra con una etiqueta @OneToOne , donde la nueva columna "ordenCompra" se crea a partir de llamar a la columna "ordencompra_id" de la tabla OrdenCompra y la referencia como "id". No menos importante, se incluyen dos constructores, uno vacío (por especificación de Hibernate) y otro que recibe el nombre y el tamaño por parámetro. Finalmente se incluyen los Getters y Setters para cada uno de los atributos de la clase.

RecepcionRepository.java

La interface RecepcionRepository extiende de JpaRepository y se encarga de gestionar las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para la entidad Recepcion. Los métodos definidos son los siguientes:

1. **darRecepciones():** Método que devuelve una Collection de todas las recepciones que se encuentran en la base de datos. Utiliza la anotación

@Query con la sentencia SQL SELECT * FROM recepcion, permitiendo así recuperar todas las filas de la tabla recepcion.

2. **darRecepcion(@Param("id") int id):**Método que devuelve una única recepción correspondiente al ID especificado.Utiliza la anotación @Query con la sentencia SQL SELECT * FROM recepcion WHERE id = :id, que permite obtener la recepción cuyo ID coincida con el proporcionado como parámetro.
3. **insertarRecepcion(@Param("fechaRecepcion") Date fechaRecepcion):**Método que inserta una nueva recepción en la base de datos.Utiliza la anotación @Modifying y @Transactional, junto con la sentencia SQL INSERT INTO recepcion (id, fecharecepcion) VALUES(recepcion_sequence.NEXTVAL, :fechaRecepcion), que genera un nuevo ID utilizando la secuencia recepcion_sequence.
4. **actualizarRecepcion(@Param("id") int id, @Param("fechaRecepcion") Date fechaRecepcion):**Método que actualiza la fecha de recepción de una recepción existente.Utiliza la anotación @Modifying y @Transactional, junto con la sentencia SQL UPDATE recepcion SET fecharecepcion=:fechaRecepcion WHERE id=:id, que actualiza la fecha de recepción correspondiente al ID especificado.
5. **eliminarRecepcion(@Param("id") int id):**Método que elimina una recepción existente en la base de datos.Utiliza la anotación @Modifying y @Transactional, junto con la sentencia SQL DELETE FROM recepcion WHERE

id=:id, que elimina la recepción cuyo ID coincida con el proporcionado como

RecepcionController.java

El controlador RecepcionController se encarga de gestionar las operaciones relacionadas con la entidad Recepcion. Utiliza la interfaz RecepcionRepository para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre las recepciones. Los métodos definidos son los siguientes:

1. **recepciones():**Método que maneja las solicitudes GET a la ruta /recepcion.Devuelve una colección de todas las recepciones almacenadas en la base de datos mediante el método darRecepciones() del RecepcionRepository.
2. **guardarRecepcion(@RequestBody Recepcion recepcion):**Método que maneja las solicitudes POST a la ruta /recepcion/new/save.Crea una nueva recepción en la base de datos.Utiliza el método insertarRecepcion() del RecepcionRepository para guardar la fecha de recepción proporcionada en el cuerpo de la solicitud.Retorna un ResponseEntity con un mensaje de éxito y un estado HTTP 201 CREATED si la creación es exitosa. En caso de error, devuelve un estado 500 INTERNAL SERVER ERROR.
3. **editarGuardarRecepcion(@PathVariable("id") int id, @RequestBody Recepcion recepcion):**Método que maneja las solicitudes POST a la ruta /recepcion/{id}/edit/save.Actualiza una recepción existente en la base de datos según el ID proporcionado en la URL.Utiliza el método actualizarRecepcion() del RecepcionRepository para actualizar

la fecha de recepción. Retorna un ResponseEntity con un mensaje de éxito y un estado HTTP 200 OK si la actualización es exitosa. En caso de error, devuelve un estado 500 INTERNAL SERVER ERROR.

4. **eliminarRecepcion(@PathVariable("id") int id):**Método que maneja las solicitudes GET a la ruta /recepcion/{id}/delete. Elimina la recepción correspondiente al ID proporcionado en la URL. Utiliza el método eliminarRecepcion() del ReceptionRepository para eliminar la recepción. Retorna un ResponseEntity con un mensaje de éxito y un estado HTTP 200 OK si la eliminación es exitosa. En caso de error, devuelve un estado 500 INTERNAL SERVER ERROR.

- **Sucursal:** entidad que representa una Sucursal de SuperAndes donde varias sucursales pueden estar en una ciudad, en una sucursal se pueden realizar varias ordenes de compra y una sucursal tiene varias bodegas .

Sucursal.java

Clase que representa la entidad Sucursal de nuestros modelos de datos. En esta se encuentra los atributos de "id" para representar la columna "id", "nombre" para representar la columna "nombre", "tamano" para representar la columna "tamano" y "direccion" para representar la columna "direccion". Así mismo, esta entidad tiene una relacion de muchos a 1 con la entidad Ciudad. Es por eso que en esta clase Java se especifica un atributo 'ciudad' con una etiqueta @ManyToOne que representa que varias sucursales pertenecen a una sola ciudad y que representa la columna 'ciudad_codigo' que referencia a la columna 'codigo' de la tabla Ciudad en nuestra tabla de datos. No menos importante, se incluyen dos constructores, uno vacío (por especificación de Hibernate) y otro que recibe el nombre y el tamano por parámetro. Finalmente se incluyen los Getters y Setters para cada uno de los atributos de la clase.

SucursalRepository.java

La interfaz SucursalRepository extiende de JpaRepository y está diseñada para gestionar la entidad Sucursal. En esta interfaz se implementan los métodos del CRUD (Crear, Leer, Actualizar, Eliminar). Los métodos definidos son los siguientes:

1. **darSucursales():** Método que devuelve una colección de todas las sucursales almacenadas en la base de datos. Utiliza la anotación @Query con la sentencia SQL `SELECT * FROM sucursal`, asegurando que se recuperen todos los registros de la tabla sucursal.

2. **darSucursal(@Param("id") int id):**Método que devuelve una sucursal específica basada en su ID.Utiliza la anotación @Query con la sentencia SQL SELECT * FROM sucursal WHERE id = :id, donde :id es un parámetro que se pasa al método.
3. **insertarSucursal(@Param("nombre") String nombre, @Param("tamanio") int tamanio, @Param("direccion") String direccion, @Param("ciudad_codigo") int ciudad_codigo):**Método que inserta una nueva sucursal en la base de datos.Utiliza la anotación @Modifying y @Transactional para realizar la operación de inserción.La consulta SQL INSERT INTO sucursal (id, nombre, tamanio, direccion, ciudad_codigo) VALUES(sucursal_sequence.NEXTVAL, :nombre, :tamanio, :direccion, :ciudad_codigo) se utiliza para crear la nueva sucursal.
4. **actualizarSucursal(@Param("id") int id, @Param("nombre") String nombre, @Param("tamanio") int tamanio, @Param("direccion") String direccion):**Método que actualiza una sucursal existente en la base de datos según el ID proporcionado.Utiliza la anotación @Modifying y @Transactional para realizar la operación de actualización.La consulta SQL UPDATE sucursal SET nombre=:nombre, tamanio=:tamanio, direccion=:direccion WHERE id=:id se utiliza para modificar los campos de la sucursal correspondiente al ID.
5. **eliminarSucursal(@Param("id") int id):**Método que elimina una sucursal de la base de datos basada en su ID.Utiliza la anotación @Modifying y @Transactional para realizar la

operación de eliminación. La consulta SQL DELETE FROM sucursal WHERE

SucursalController.java

El controlador SucursalController está diseñado para gestionar las operaciones relacionadas con la entidad Sucursal. A continuación se detallan los métodos implementados en este controlador:

1. **sucursales():** Método que maneja las solicitudes GET en la ruta `/sucursal`. Devuelve una colección de todas las sucursales que se encuentran en la base de datos, utilizando el método `darSucursales()` del repositorio `SucursalRepository`.
2. **guardarSucursal(@RequestBody Sucursal sucursal):** Método que maneja las solicitudes POST en la ruta `/sucursal/new/save`. Recibe un objeto `Sucursal` en el cuerpo de la solicitud y llama al método `insertarSucursal()` del repositorio `SucursalRepository` para guardar la nueva sucursal en la base de datos. Devuelve una respuesta con el estado HTTP 201 CREATED si la sucursal se crea exitosamente, o 500 INTERNAL SERVER ERROR si ocurre un error durante el proceso.
3. **editarGuardarSucursal(@PathVariable("id") int id, @RequestBody Sucursal sucursal):** Método que maneja las solicitudes POST en la ruta `/sucursal/{id}/edit/save`, donde `{id}` es el identificador de la sucursal a actualizar. Recibe el ID de la sucursal y un objeto `Sucursal` en el cuerpo de la solicitud. Llama al método `actualizarSucursal()` del repositorio para actualizar la información de la sucursal. Devuelve una respuesta con el estado HTTP 200 OK si la sucursal se actualiza exitosamente, o 500

INTERNAL SERVER ERROR si se presenta un error.

4. **eliminarSucursal(@PathVariable("id") int id):**Método que maneja las solicitudes GET en la ruta /sucursal/{id}/delete.Recibe el ID de la sucursal a eliminar y llama al método eliminarSucursal() del repositorio SucursalRepository para eliminar la sucursal de la base de datos.Devuelve una respuesta con el estado HTTP 200 OK si la sucursal se elimina exitosamente, o 500 INTERNAL SERVER ERROR si ocurre un error durante la eliminación.

Ahora, especificaremos que funciones de los repositorys se usaron para cumplir con los RF y lo RFC.

- **RF1 CREAR UNA CIUDAD:** Todas las ciudades están en Colombia, por lo que solo es necesario manejar su nombre y código.
 - Función usada: se usó la función insertarCiudad(@Param("nombre") String nombre) de la clase CiudadRepository.java para cumplir con este requerimiento.
- **RF2 CREAR UNA SUCURSAL:** Debe considerar la información básica de una sucursal. Recuerde que la sucursal está asociada a una sola ciudad existente previamente.
 - Función usada: se usó la función insertarSucursal(@Param("nombre") String nombre, @Param("tamanio") int tamanio, @Param("direccion") String direccion, @Param("ciudad_codigo") int ciudad_codigo) de la clase SucursalRepository.java para cumplir con este requerimiento
- **RF3 CREAR Y BORRAR UNA BODEGA:** Debe considerar la informacion basica de una bodega. Para ello, se usa las siguientes funciones en Bodega.Repository:
 - Primera Función usada: se usó la función insertarBodega((@Param("nombre") String nombre,@Param("tamanio")Integer tamanio ,@Param("sucursal_id")Integer sucursal_id) String nombre) de la clase CiudadRepository.java para cumplir con este requerimiento.
 - Segunda Funcion usada: se uso la funcion eliminarBodega(@Param("id") int id)
- **RF4 CREAR Y ACTUALIZAR PROVEEDORES :** Debe considerar la informacion basica de un PROVEEDOR. Para ello, se usa las siguientes funciones en Proveedor.Repository:
 - Primera Función usada: se usó la función insertarProveedor(@Param("nit") int nit, @Param("nombre") String nombre, @Param("direccion") String direccion, @Param("nombrePersona") String nombrePersona, @Param("telefonoPersona") int telefonoPersona)
 - Segunda Funcion usada: se uso la funcion actualizarProveedor(@Param("nit") int nit, @Param("nombre") String nombre, @Param("direccion") String direccion, @Param("nombrePersona") String nombrePersona, @Param("telefonoPersona") int telefonoPersona)
- **RF5 CREAR Y LEER UNA CATEGORÍA DE PRODUCTO:** Para la creación, se debe registrar la información de las categorías de productos según el enunciado. Para la lectura, se solicita el código o nombre de la categoría, y como resultado, se obtiene la información de dicha categoría.

- Primera función: se usó la función `insertarCategoria(@Param("nombre") String nombre, @Param("descripcion") String descripcion, @Param("caracteristicasAlmacenaje") String caracteristicasAlmacenaje)` para poder crear una Categoría.
- Segunda función: usó la función `actualizarCategoria(@Param("codigo") int codigo, @Param("nombre") String nombre, @Param("descripcion") String descripcion, @Param("caracteristicasAlmacenaje") String caracteristicasAlmacenaje)` para poder actualizar una categoría.
- **RF6 - CREAR, LEER Y ACTUALIZAR UN PRODUCTO:** Para la creación de un producto, debe registrar toda la información de los productos según el enunciado. Recuerde que, en su creación, el producto debe ser asociado a una categoría existente. Recuerde igualmente que una categoría puede tener múltiples productos asociados. Para la lectura, se solicita el código o nombre del producto, y como resultado, se obtiene la información de dicho producto incluyendo la categoría a la que pertenece. Para la Actualización, se solicita el código del producto, y la información que se quiere actualizar, y como resultado, se obtiene la actualización de dicho producto.
 - Primera función: se usó la función `insertarProducto(@Param("nombre") String nombre, @Param("costoBodega") Integer costoBodega, @Param("costoUnidad") Integer costoUnidad, @Param("presentacion") String presentacion, @Param("cantidadPresentacion") Integer cantidadPresentacion, @Param("unidadMedida") String unidadMedida, @Param("fechaExpiracion") Date fechaExpiracion, @Param("categoria_codigo") Integer categoria_codigo, @Param("empaquete_id") Integer empaque_id)` para poder crear un producto.
 - Segunda función: se usaron las funciones `darProductoCodigo(@Param("codigoBarras") Integer codigoBarras)` y `darProductoNombre(@Param("nombre") String nombre)` para poder leer un producto ya sea por el código de barras o el nombre.
 - Tercera función: se usó la función `actualizarProducto(@Param("codigoBarras") Integer codigoBarras, @Param("nombre") String nombre, @Param("costoBodega") Integer costoBodega, @Param("costoUnidad") Integer costoUnidad, @Param("presentacion") String presentacion, @Param("cantidadPresentacion") Integer cantidadPresentacion, @Param("unidadMedida") String unidadMedida, @Param("fechaExpiracion") Date fechaExpiracion, @Param("categoria_codigo") Integer categoria_codigo, @Param("empaquete_id") Integer empaque_id)` para actualizar la información de un producto.
- **RF7 - CREAR UNA ORDEN DE COMPRA PARA UNA SUCURSAL:** Debe considerar toda la información de las órdenes de compra. Para completar este documento, desde un punto de vista gráfico, el usuario vería dos partes: el encabezado y el detalle. El encabezado contiene:
 - - - - La fecha de creación de la orden (corresponde a la fecha del día, la cual aparece automáticamente y el usuario no la puede modificar). La sucursal, la cual es escogida por el usuario de la lista de sucursales. El proveedor al que se le quiere comprar, el cual es escogido de la lista de proveedores. La fecha esperada de entrega de los productos, la cual es ingresada por el usuario. El detalle por su lado contiene:
 - - - Uno o más productos que se quieren comprar, los cuales se escogen de los productos existentes. Las cantidades de los productos seleccionados, los cuales son ingresados por el usuario. Los precios de los productos, los cuales son ingresados por el usuario. Por último, al ser creada, la orden de compra queda en estado "vigente".
- Primera función: se usó la función `insertarOrdenCompra(@Param("fechaEsperada") Date fechaEsperada, @Param("fechaCreacion") Date fechaCreacion, @Param("estado") String estado, @Param("sucursal_id") Integer sucursal_id, @Param("proveedor_nit") Integer proveedor_nit)` para poder crear una Orden de Compra.

- Segunda función: se usó la función `insertarProductoCompra(@Param("productoCodigo") int productoCodigo, @Param("ordenCompraId") int ordenCompraId, @Param("precioAcordado") int precioAcordado, @Param("cantidad") int cantidad)` para poder crear el detalle de una orden de compra
- **RF8 ACTUALIZAR UNA ORDEN DE COMPRA CAMBIANDO SU ESTADO A ANULADA:** Para anular una orden de compra, se ingresa el número de la orden. Como resultado, la orden pasa de estado "vigente" a "anulada". Una orden de compra en estado "entregada" no puede ser anulada.
 - Función usada: se usó la función `actualizarOrdenCompra(@Param("id") int id, @Param("estado") String estado)` para poder actualizar una orden de compra a anulada.
- **RF9 MOSTRAR TODAS LAS ORDENES DE COMPRA:** Para mostrar todas las ordenes de compra que se tiene al momento, se usa la siguiente función del `OrdenCompra.Repository` :
 - Función usada: se usó la función `darOrdenesCompra()` , que retorna toda las ordenes de compra que se han realizado son toda su información.

Escenarios de prueba

En este conjunto de pruebas se evaluará la integridad de la base de datos, centrándonos en tablas con restricciones de tipo CHECK (CK) y en diferentes scripts de inserción para validar las claves foráneas (FK). Las pruebas referentes a las claves primarias (PK) no se detallarán para cada tabla individualmente, ya que el resultado fue uniforme en todos los casos. No obstante, se describirá el procedimiento general utilizado para la verificación de las PK de cada tabla.

Pruebas de unicidad

Para verificar la unicidad de las tablas, primero se toma un registro del estado actual de cada tabla. Luego, se selecciona un valor para el identificador (ID) que no esté previamente presente en la tabla. El procedimiento a seguir es el siguiente:

1. Se crea una tupla con valores aleatorios que cumplan con las reglas de negocio establecidas. El ID de esta tupla será el seleccionado previamente.
2. Se inserta la tupla en la tabla utilizando el siguiente comando SQL:
 - **INSERT INTO nombre_tabla VALUES (valores_tupla);**

Si la inserción se realiza correctamente, se repite el proceso, pero en esta ocasión se intenta insertar una nueva tupla utilizando el mismo ID que en la inserción anterior. Si la tabla está correctamente estructurada, el sistema debería generar un error, indicando que no es posible insertar dos tuplas con la misma clave primaria.

```
INSERT INTO bodega
VALUES (5, 'Bodega afuera', 120, '2')
Informe de error -
ORA-00001: restricción única (ISIS2304A28202420.BODEGA_PK) violada
```


El éxito de esta prueba confirma que la estructura de la tabla y la configuración de su clave primaria son correctas, ya que no se permite la duplicación de valores en la PK.

Estas pruebas se realizaron para las tablas de Producto, Bodega, Categoría, Proveedor, Sucursal, Ciudad, OrdenCompra, Recepción, Empaque, y ProductoPerecedero. En todas las tablas, los resultados fueron consistentes y pasaron exitosamente esta primera fase de validación.

Pruebas integridad

- Sucursal:

- Insertaremos una tupla que tiene un FK que ya existe en la tabla ciudad
- Se inserta la tupla (5, 'central', 12, 'aquí', 2). Y se obtiene:

```
1 fila insertadas.
```

- Ahora insertaremos una tupla con un FK que no tiene la tabla ciudad, será la siguiente tupla (6, 'abajo', 13, 'alla', 5). Y se obtiene:

```
Error que empieza en la línea: 2 del comando :
INSERT INTO sucursal
values (6, 'alla' , 13 , 'alli', 5)
Informe de error -
ORA-02291: restricción de integridad (ISIS2304A28202420.SUCURSAL_CIUDAD_FK) violada - clave principal no encontrada
```

- La tabla esta correctamente construida, no permite la entrada de FK que no estén antes definidas en la respectiva tabla.

- Bodega:

- Repetimos el procedimiento, se insertará la tupla (5, 'algo', 45, 4), donde si existe la sucursal asignada. Y obtenemos el resultado de fila insertada correctamente.

```
1 fila insertadas.
```

- Al querer ingresar una fila con un FK no existente obtenemos el siguiente resultado:

```
Error que empieza en la línea: 1 del comando :
insert into bodega values (6, 'algo', 45, 456)
Informe de error -
ORA-02291: restricción de integridad (ISIS2304A28202420.BODEGA_SUCURSAL_FK) violada - clave principal no encontrada
```

- La tabla está construida correctamente.

- Producto:

- Se probará la FK de empaque_id.
- Se insertará la tupla correcta, (1920, 'producto', 45, 45, 'algo', 45, 'g', '15/01/26', 1, 1), y se obtiene.

```
1 fila insertadas.
```

- Al probar con la tupla (1921, 'producto', 45, 45, 'algo', 45, 'g', '15/01/26', 1, 545), la llave 545 no existe en empaque_id, al intentarlo obtenemos:

```
Error que empieza en la línea: 3 del comando :
insert into producto values (1921, 'producto', 45, 45, 'algo', 45, 'g', '15/01/26', 1, 545)
Informe de error -
ORA-02291: restricción de integridad (ISIS2304A28202420.PRODUCTO_EMPAQUE_FK) violada - clave principal no encontrada
```

- Se probará la FK de categoría_codigo
- Se inserta la tupla (1922, 'producto', 45, 45, 'algo', 45, 'g', '15/01/26', 1, 1), donde la FK de categoría_producto existe y se obtiene.

```
1 fila insertadas.
```

- Luego al probar la tupla (1924, 'producto', 45, 45, 'algo', 45, 'g', '15/01/26', 1798, 1), donde la FK 1789 no existe, y obtenemos el siguiente resultado.

```
Error que empieza en la línea: 6 del comando :
insert into producto values (1924, 'producto', 45, 45, 'algo', 45, 'g', '15/01/26', 1798, 1)
Informe de error -
ORA-02291: restricción de integridad (ISIS2304A28202420.PRODUCTO_CATEGORIA_FK) violada - clave principal no encontrada
```

● OrdenCompra:

- Probaremos la FK de proveedor_nit
- Insertaremos la tupla (6, '01/11/24', '30/09/24', 'Vigente', 1, 1), donde las FK están definidas en la otra tabla y obtenemos el siguiente resultado.

```
1 fila insertadas.
```

- Insertaremos la tupla (7, '01/11/24', '30/09/24', 'Vigente', 1, 654), donde la llave 654 no existe.

```
Error que empieza en la línea: 4 del comando :
insert into ordencompra values (7, '01/11/24', '30/09/24', 'Vigente', 1, 654)
Informe de error -
ORA-02291: restricción de integridad (ISIS2304A28202420.ORDENCOMPRA_PROVEEDOR_FK) violada - clave principal no encontrada
```

- Probaremos la FK de sucursal_id
- Insertaremos la tula (8, '01/11/24', '30/09/24', 'Vigente', 1, 1), donde las llaves foráneas son correctas obtenemos.

```
1 fila insertadas.
```

- Finalmente insertaremos la tupla (9, '01/11/24', '30/09/24', 'Vigente', 564, 1), donde la llave 564 no existe en la tabla de referencia y obtenemos el siguiente resultado:

```
Error que empieza en la línea: 7 del comando :
insert into ordencompra values (9, '01/11/24', '30/09/24', 'Vigente', 564, 1)
Informe de error -
ORA-02291: restricción de integridad (ISIS2304A28202420.ORDENCOMPRA_SUCURSAL_FK) violada - clave principal no encontrada
```

● Recepción:

- Verificaremos ordencompra_id
- Insertaremos la siguiente tupla donde la FK existe en la tabla de referencia (6, '01/10/24', 1, 1), y obtenemos el siguiente resultado.

```
1 fila insertadas.
```

- Luego insertaremos la siguiente tupla donde la llave foránea no existe en la tabla de referencia (6, '01/10/24', 1, 47857), y obtenemos el siguiente resultado.

```
Error que empieza en la línea: 3 del comando :
insert into recepcion values (6, '01/10/24', 1, 47857)
Informe de error -
ORA-02291: restricción de integridad (ISIS2304A28202420.RECEPCION_ORDEN_FK) violada - clave principal no encontrada
```

- Verificaremos bodega_id
- Probaremos insertando la tula, donde las llaves están correctas, (6, '01/10/24', 1, 1), y obtenemos el siguiente resultado:

```
1 fila insertadas.
```

- Ahora probaremos con una FK la cual no existe en la tabla de llegada, (6, '01/10/24', 78989, 8979), obteniendo.

```
Error que empieza en la línea: 5 del comando :
insert into recepcion values (6, '01/10/24', 78989, 8979)
Informe de error -
ORA-02291: restricción de integridad (ISIS2304A28202420.RECEPCION_BODEGA_FK) violada - clave principal no encontrada
```

Pruebas integridad CK

- Bodega:

- tamaño > 0

```
Error que empieza en la línea: 3 del comando :
insert into bodega values (6,'algo', -1, 1)
Informe de error -
ORA-02290: restricción de control (ISIS2304A28202420.TAMANIOBOD_CK) violada
```

- Categoría:

- nombre - unique

```
Error que empieza en la línea: 3 del comando :
insert into categoria values (8,'Aseo', 'algo', 'algo')
Informe de error -
ORA-00001: restricción única (ISIS2304A28202420.CATEGORIANOMBRE_UK) violada
```

- Ciudad:

- nombre in listaCiudadesColombia

```
Error que empieza en la línea: 3 del comando :
insert into ciudad values (8,'Aseo')
Informe de error -
ORA-02290: restricción de control (ISIS2304A28202420.CIUDADNOMBRE_CK) violada
```

- nombre unique

```
Error que empieza en la línea: 3 del comando :
insert into ciudad values (8,'Bogota')
Informe de error -
ORA-00001: restricción única (ISIS2304A28202420.CIUDADNOMBRE_UK) violada
```

- Empaque:

- Volumen > 0

```
Error que empieza en la línea: 3 del comando :
insert into empaque values (8,-1,1)
Informe de error -
ORA-02290: restricción de control (ISIS2304A28202420.VOLUMEN_CK) violada
```

- `PesoEmpaque > 0`

```
Error que empieza en la línea: 3 del comando :
insert into empaque values (8,1,-1)
Informe de error -
ORA-02290: restricción de control (ISIS2304A28202420.PESO_CK) violada
```

- Ordencompra:

- `fechaCreacion < fechaEsperada`

```
Error que empieza en la línea: 3 del comando :
insert into ordencompra values (8,'01/11/24','01/11/24', 'Vigente', 1,5)
Informe de error -
ORA-02290: restricción de control (ISIS2304A28202420.ORDENFECHA_CK2) violada
```

- `estado in listaEstados`

```
Error que empieza en la línea: 3 del comando :
insert into ordencompra values (8,'01/11/24','30/09/24', 'algo', 1,5)
Informe de error -
ORA-02290: restricción de control (ISIS2304A28202420.ORDENESTADO_CK) violada
```

- Producto:

- `cantidadpresentacion > 0`

```
Error que empieza en la línea: 3 del comando :
insert into producto values (1,'Jabón Líquido', 1200, 1400, 'Botella', -1, 'ml','15/08/26', 1 ,1)
Informe de error -
ORA-02290: restricción de control (ISIS2304A28202420.PRODUCTOCANTIDAD_CK) violada
```

- `costoUnidad > 0`

```
Error que empieza en la línea: 3 del comando :
insert into producto values (1,'Jabón Líquido', 1200, -1, 'Botella', 1, 'ml','15/08/26', 1 ,1)
Informe de error -
ORA-02290: restricción de control (ISIS2304A28202420.COSTOUNIDAD_CK) violada
```

- `costoBodega > 0`

```
Error que empieza en la línea: 3 del comando :
insert into producto values (1,'Jabón Líquido', -1, 1, 'Botella', 1, 'ml','15/08/26', 1 ,1)
Informe de error -
ORA-02290: restricción de control (ISIS2304A28202420.COSTOBODEGA_CK) violada
```

- Proveedor:

- `telefonoPersona > 0`

```
Error que empieza en la línea: 3 del comando :
insert into proveedor values (1,'Proveedor A' , 'Av. Principal' , 'Juan Pérez' , -1)
Informe de error -
ORA-02290: restricción de control (ISIS2304A28202420.PROVEEDORTELEFONO_CK) violada
```

- telefonoPersona UNUQUE

```
Error que empieza en la línea: 3 del comando :
insert into proveedor values (1,'Proveedor A' , 'Av. Principal' , 'Juan Pérez' , 3005)
Informe de error -
ORA-00001: restricción única (ISIS2304A28202420.PROVEEDOR_PK) violada
```

- Sucursal:

- tamaño > 0

```
Error que empieza en la línea: 3 del comando :
insert into sucursal values (80, 'Sucursal Occidente', -1, 'Calle', 1)
Informe de error -
ORA-02290: restricción de control (ISIS2304A28202420.SUCURSALTAMANIO_CK) violada
```

Requerimientos funcionales

- RF1

Crear:

insert into ciudad values (3, 'Cali');

CODIGO	NOMBRE
1	Bogota
2	Bucaramanga
3	Cali

- RF2

Crear:

insert into sucursal values (5, 'Sucursal Central', 150, 'Calle 5', 2);

ID	NOMBRE	TAMANIO	DIRECCION	CIUDAD_CODIGO
1	Sucursal Occidente	100	Calle 1	1
2	Sucursal Oriente	150	Calle 2	1
3	Sucursal Norte	120	Calle 3	2
4	Sucursal Sur	130	Calle 4	2
5	Sucursal Central	150	Calle 5	2

- RF3

Crear:

insert into bodega values (5, 'Bodega Central', 150 , 2);

ID	NOMBRE	TAMANIO	SUCURSAL_ID
1	Bodega Occidente	50	1
2	Bodega Oriente	60	2
3	Bodega Norte	55	3
4	Bodega Sur	70	4
5	Bodega Central	150	2

Bodega:

delete from bodega where (id = 5)

ID	NOMBRE	TAMANIO	SUCURSAL_ID
1	Bodega Occidente	50	1
2	Bodega Oriente	60	2
3	Bodega Norte	55	3
4	Bodega Sur	70	4

- RF4

Crear:

insert into proveedor values (6, 'Proveedor D', 'Avenida Sexta', 'Juan López',3006);

NIT	NOMBRE	DIRECCION	NOMBREPERSONA	TELEFONOPERSONA
1	Proveedor A	Av. Principal	Juan Pérez	3001
2	Proveedor B	Calle Secundaria	María García	3002
3	Proveedor C	Calle Terciaria	Carlos Rodríguez	3003
4	Proveedor D	Carrera Cuarta	Ana Gómez	3004
5	Proveedor E	Avenida Quinto	Pedro López	3005
6	Proveedor D	Avenida Sexta	Juan López	3006

Update:

update proveedor set nombrepersona = 'Jack Sparow' where (nit = 6);

NIT	NOMBRE	DIRECCION	NOMBREPERSONA	TELEFONOPERSONA
1	Proveedor A	Av. Principal	Juan Pérez	3001
2	Proveedor B	Calle Secundaria	María García	3002
3	Proveedor C	Calle Terciaria	Carlos Rodríguez	3003
4	Proveedor D	Carrera Cuarta	Ana Gómez	3004
5	Proveedor E	Avenida Quinto	Pedro López	3005
6	Proveedor D	Avenida Sexta	Jack Sparow	3006

RF5

Crea:

insert into categoria values (7, 'Casa', 'Para la casa', 'NA');

CODIGO	NOMBRE	DESCRIPCION	CARACTERISTICASALMACENAJE
1	Aseo	Productos de limpieza y aseo	Almacenar lejos de alimentos
2	Congelados	Productos congelados	Mantener a -18°C
3	Prendas de vestir	Ropa y accesorios	Almacenar en lugar seco
4	Muebles	Muebles para hogar y oficina	Proteger de humedad y golpes
5	Herramientas	Herramientas manuales y eléctricas	Almacenar en lugar seco
6	Electrodomésticos	Electrodomésticos de cocina y hogar	Almacenar en lugar fresco y seco
7	Casa	Para la casa	NA

Lectura:

select * from categoria where categoria.codigo = 7;

CO...	NOMBRE	DESCRIPCION	CARACTERISTICASALMACENAJE
7	Casa	Para la casa	NA

• RF6

Crear:

insert into producto values (19 , 'Chicle', 500 , 700, 'Envase', 500, 'ml', '30/06/25', 2 , 1);

CODIGOBARRAS	NOMBRE	COSTOBODEGA	COSTOUNIDAD	PRESENTACION	CANTIDADPRESENTACION	UNIDADMEDIDA	FECHAEXPIRACION	CATEGORIA_CODIGO	EMPAQUE_ID
7	Camisa	15000	18000	Unidad		Unidad	31/12/25	3	1
8	Pantalón	25000	28000	Unidad		Unidad	31/12/25	3	3
9	Chaqueta	30000	35000	Unidad		Unidad	31/12/25	3	2
10	Mesa	50000	55000	Unidad		Unidad	15/01/26	4	2
11	Silla	10000	15000	Unidad		Unidad	10/02/26	4	1
12	Estante	30000	35000	Unidad		Unidad	20/02/26	4	3
13	Martillo	2000	3000	Unidad		Unidad	31/12/27	5	1
14	Destornillador	1000	1500	Unidad		Unidad	31/12/27	5	3
15	Taladro	10000	12000	Unidad		Unidad	31/12/27	5	2
16	Licudora	80000	85000	Unidad		Unidad	15/08/26	6	2
17	Microondas	120000	125000	Unidad		Unidad	20/09/26	6	2
18	Refrigerador	300000	320000	Unidad		Unidad	10/01/27	6	2
19	Chicle	500	700	Envase		500 ml	30/06/25	2	1

Leer:

select * from producto where producto.codigobarras = 4;

CODIGOBA...	NOMBRE	COSTOBODEGA	COSTOUNIDAD	PRESENTACION	CANTIDADPRESENTACION	UNIDADMEDIDA	FECHAEXPIRACION	CATEGORIA_CODIGO	EMPAQUE_ID
4	Helado de Vainilla	500	700	Envase		500 ml	30/06/25	2	1

Actualizar:

CODIGOBA...	NOMBRE	COSTOBODEGA	COSTOUNIDAD	PRESENTACION	CANTIDADPRESENTACION	UNIDADMEDIDA	FECHAEXPIRACION	CATEGORIA_CODIGO	EMPAQUE_ID
19	Papas	500	700	Envase		500 ml	30/06/25	2	1

• RF7

Crear:

insert into ordencompra values (6, '01/11/24', '30/09/24', 'Vigente', 1, 1);

ID	FECHAESPERADA	FECHACREACION	ESTADO	SUCURSAL_ID	PROVEEDOR_NIT
1	01/11/24	30/09/24	Vigente	1	1
2	10/11/24	30/09/24	Vigente	2	2
3	15/11/24	30/09/24	Vigente	3	3
4	20/11/24	30/09/24	Vigente	4	4
5	25/11/24	30/09/24	Vigente	1	5
6	01/11/24	30/09/24	Vigente	1	1

• RF8

Actualizar:

ID	FECHAESPERADA	FECHACREACION	ESTADO	SUCURSAL_ID	PROVEEDOR_NIT
1	01/11/24	30/09/24	Vigente	1	1
2	10/11/24	30/09/24	Vigente	2	2
3	15/11/24	30/09/24	Vigente	3	3
4	20/11/24	30/09/24	Vigente	4	4
5	25/11/24	30/09/24	Vigente	1	5

update ordencompra set estado = 'Anulada' where (id = 3);

ID	FECHAESPERADA	FECHACREACION	ESTADO	SUCURSAL_ID	PROVEEDOR_NIT
1	01/11/24	30/09/24	Vigente	1	1
2	10/11/24	30/09/24	Vigente	2	2
3	15/11/24	30/09/24	Anulada	3	3
4	20/11/24	30/09/24	Vigente	4	4
5	25/11/24	30/09/24	Vigente	1	5

• RF9

Mostrar:

select * from ordencompra;

ID	FECHAESPERADA	FECHACREACION	ESTADO	SUCURSAL_ID	PROVEEDOR_NIT
1	01/11/24	30/09/24	Vigente	1	1
2	10/11/24	30/09/24	Vigente	2	2
3	15/11/24	30/09/24	Vigente	3	3
4	20/11/24	30/09/24	Vigente	4	4
5	25/11/24	30/09/24	Vigente	1	5

Usuario Oracle

Usuario: ISIS2304A2824020

En este usuario se encuentra la base de datos de nuestra aplicación.