**SQL - DATABASE**

Una database es un conjunto de datos pertenecientes a un mismo contexto, organizados para un propósito específico.

Funciona para:

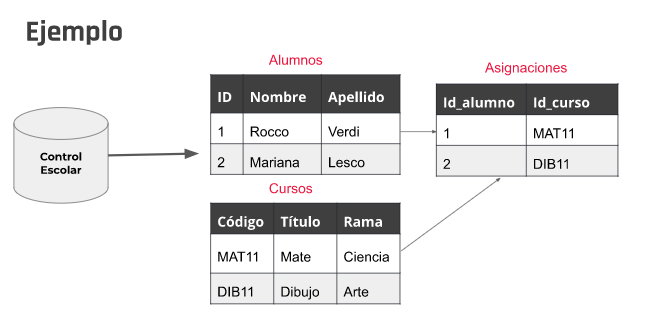
**Almacenar:** agregar, eliminar, modificar datos.

**Acceder** a datos.

**Manipular** y combinar datos.

**Analizar** datos.

Dentro de una base de datos los datos se alojan y agrupan por tablas, estas contienen filas y columnas. Las columnas definen qué clase de datos se guardará y por lo general la fila su ubicación. Dentro de esto habrán celdas, definidas en SQL como campos, que son el lugar concreto fila-columna en el que se encuentra una información.

0

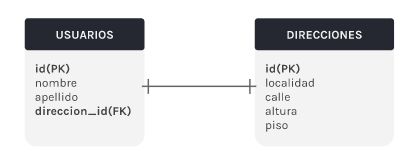
Las tablas están definidas en sql y programación por **entidades**, las cuales están compuestas por atributos. Estas entidades definen el nombre de la tabla y los atributos sus columnas.

**Primary Key:** la PK o Clave Primaria define dentro de una tabla un valor único e irrepetible que va a identificar a cada registro de dicha tabla, es decir, no pueden haber dos filas en una misma tabla con el mismo PK. Por lo general se nombra como PK el **id** de dicho registro y sirve, entre otras cosas, para relacionar los registros de una tabla con otra.

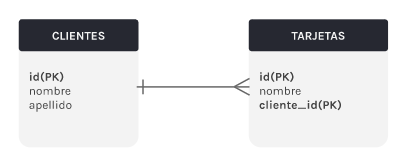
**Foreign Key:** una clave foránea o FK no es más que una referencia a la PK de otra tabla.

**RELACIONES**

Las tablas de una base de datos pueden relacionar su contenido entre ellas y hay tres tipos de relaciones: **Uno a uno (1:1):** un ejemplo que define a este tipo de relaciones es la de **usuario - dirección**. Un registro de la tabla dirección solo puede relacionarse a un usuario y a un usuario solo se le relaciona una dirección. Para identificar dichas relaciones se utiliza el sig. dibujo:



**Uno a muchos (1:M):** este se puede definir con **cliente - tarjetas.** Un cliente puede relacionarse con varias tarjetas, pero una tarjeta puede relacionarse a un solo cliente.



**Muchos a muchos (M:M):** un ejemplo claro es la relación **cliente - producto,** donde un cliente puede relacionarse con muchos productos y un producto con muchos clientes.

Para estos casos, se crean tablas intermedias o tablas pivot, en las cuales se define un PK para hacer dicho registro único e indicamos con dos FK qué cliente y qué tarjeta se relacionan. Seg. Ejm: Juan compró una pelota, Juan compró una laptop. Marta compró una laptop





**CONSTRAINS**

**UNIQUE:** sirve para identificar de forma unívoca un registro en una columna. Es prácticamente igual que el PK solo que pueden haber varios Unique en una tabla y solo un PK.

**NULL / NOT NULL:** sirve para especificar que un registro puede ser **NULL** o en el segundo caso que un registro no puede ser null, es decir, que debe tener algún tipo de dato asociado.

**DEFAULT:** sirve para indicar un valor por defecto en caso de que no se agregue ningún dato al campo.

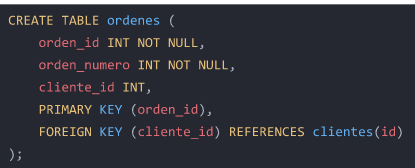
**AUTO\_INCREMENT:** sirve para que el valor de una columna se incremente automáticamente cuando ingresemos una nueva row en la table.

**DDL - DEFINITION DATA LANGUAJE (Definición de estructura de una tabla)**

**CREATE, ALTER y DROP TABLE**

**Create:** sirve para crear una tabla y sintaxis es:

**CREATE** nombreTabla (**nombreColumna1** tipodeDato constrains, **nombreColumna2** tipoDato constrains)



**DROP TABLE:** sirve para eliminar una tabla en caso de que esta exista:

**DROP TABLE IF EXIST** nombreTabla.



**ALTER TABLE:** sirve para modificar la estructura de una tabla.

Funciona con tres comandos:

**ADD:** para agregar una columna.



**MODIFY:** para modificar una columna.



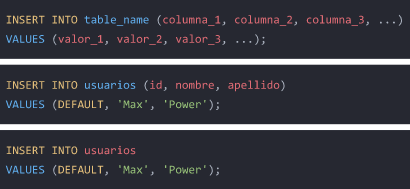
**DROP:** para eliminar una columna.



**DML (DATA MANIPULATION LANGUAGE)**

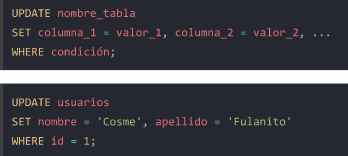
**INSERT INTO**

Se utiliza para insertar datos en una tabla. Se puede hacer especificando los campos que queremos completar, y luego colocando los VALUES en el mismo orden que especificamos los campos. Si no, podemos no especificar los campos, pero esto quiere decir que debemos respetar el orden en el que están establecidas las columnas en nuestra tabla.



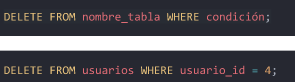
**UPDATE**

Se utiliza para actualizar algún campo o todos los campos de una row. Es importante indicar el WHERE para especificar a qué columna o columnas queremos actualizar con dichos datos.



**DELETE**

Se utiliza para eliminar uno o varios registros. En caso de que no coloquemos WHERE, borra todos los registros de la tabla.



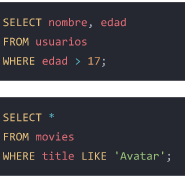
**SELECT**

Su función es traer resultados desde una tabla.



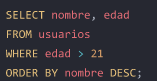
**WHERE**

El where funciona para indicar una condición en la query. Si el resultado de esta condición es true se realizará lo que la query indique.



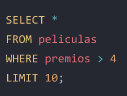
**ORDER BY**

El order by funciona para poder ordenar los resultados de una tabla en forma **ASC**ENDENTE. En caso de que queramos hacerlo de forma descendente tenemos que especificar DESC después del nombre de la columna.



**LIMIT Y OFFSET**

Se usa para limitar el número de filas que va a traer la consulta o el número máximo de filas a eliminar.



**OFFSET** sirve para saltearse las primeras n filas de la tabla. 

**ALIAS**

Sirve para darle un nombre a una columna especifica en el resultado de una consulta.



**FUNCIONES DE ALTERACIÓN**

**CONCAT**

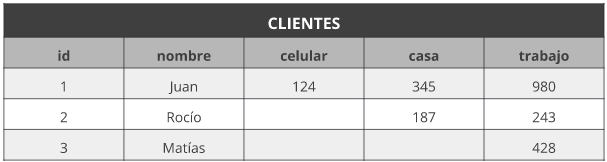
Sirve para concatenar dos o más expresiones:

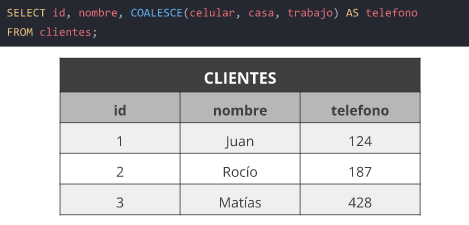
****

**COALESCE**

Coalesce se utiliza para tener un plan B. Como primera parámetro le indicaremos la columna sobre la que va a operar y como segundo parámetro le podemos pasar el valor que queramos que tenga una columna en caso de que esta sea null.

También se puede utilizar para obtener el primer dato NOT NULL de una row. En ese caso enviamos como parámetros las columnas que tenga que revisar la consulta y con el orden que queramos.





**DATEDIFF**

Esta función nos sirve para poder obtener la diferencia de días entre dos fechas.

**(Devuelve 10)**

**EXTRACT**

Sirve para extraer de una fecha la parte que le indiquemos. Estas partes se indican como second, minute, hour, day, week, month, quarter, year:



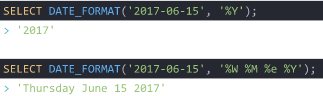
**REPLACE**

Sirve para reemplazar dentro de un string que será el primer parámetro, los caracteres que le indiquemos como segundo parámetro por los carácteres que le indiquemos como tercer parámetro.



**DATE\_FORMAT**

Esta función permite dar un formato específico a una columna o campo de tipo date:



**Mas info:** [**https://www.w3schools.com/sql/func\_mysql\_date\_format.asp**](https://www.w3schools.com/sql/func_mysql_date_format.asp)

**CASE**

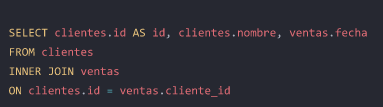
Sirve para hacer una structura case en la que primero seleccionamos que columnas queremos mostrar, con case abrimos la estructura y por cada WHEN colocamos un caso de tipo bool y un THEN en el que estableceremos su consecuencia, para finalizar establecemos un ELSE con un valor que será el default para los casos que no entren en ninguno de los establecidos anteriormente y finalizamos el ciclo con END. Luego del END le podemos dar un AS a la columna.

**JOINS**

Sirve para unir consultas a diferentes tablas en un resultado.

Con **SELECT** indicaremos qué columnas debe traer la consulta, con **FROM** indicaremos la tabla base de la consuta y con **INNER JOIN** la tabla que se combinara con la primera. Luego con **ON** indicaremos mediante qué criterio se unirán las tablas.

También podemos agregar de último un WHERE en caso de que queramos que algunas de las columnas de las tablas cumplan con un valor o rango particular.



**INNER JOIN** tiene como particularidad que no trae en el resultados los valores null, es decir, si en alguna de las columnas que trae como resultado, el valor es null, no se muestra.

En caso de que queramos que se muestren los resultados que vengan con null debemos indicar qué tabla puede traer null, si la de la derecha o la de la izquierda, siendo la de la izquierda la del from y la de la derecha la del inner join. Sería:

**LEFT JOIN:** para mostrar los resultados que vengan null de la tabla incluida en el **LEFT.**

**(mostaría todos los clientes sin ventas asociadas, es decir, mostraría los null de la tabla ventas)**

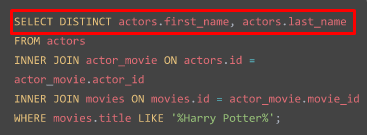
**RIGHT JOIN:** para mostrar los resultados que vengan null de la tabla incluida en el **FROM**

**(mostaría todas las ventas sin clientes asociados, es decir, mostraría los null de la tabla clientes).**

**DISTINCT**

Es un constrain que sirve para eliminar duplicados, pero solo funciona si hay filas cuyos campos son exactamente igual. Si tiene 3 campos y dos de ellos son iguales, no lo tomará como duplicado, tienen que ser **todos iguales.**

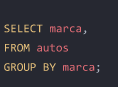
Este DISTINCT se incluye después del SELECT:

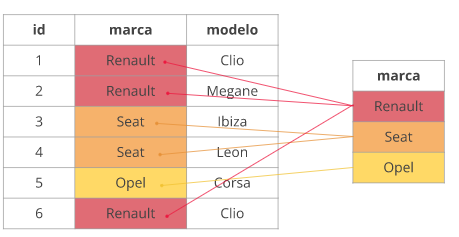


**GROUP BY**

Sirve para agrupar los registros de una tabla por los valores de una columna. Podemos agrupar así qué tipo de productos contiene la tabla o a qué marcas forman parte y a partir de esto ver el promedio de ventas, cuantos productos en total contienen, etc.

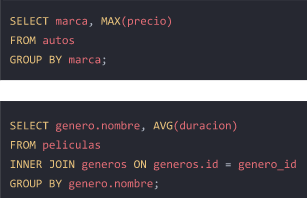
Su función es más macro, como lo dice su nombre trata de formar grupos.





En este caso en el que agrupamos los autos por marca, ya no podemos hacer selects en los que queramos mirar el ID de cada fila, ya que no están más de manera individual sino todas las filas agrupadas por marca.

**USOS MÁS ESPECÍFICOS:**

****

**FUNCIONES DE AGREGACIÓN**

**COUNT:** sirve para contar cuántos registros hay en una columna o tabla:



**MAX:** sirve para saber cuál es el valor máximo dentro de los campos de una columna.



**MIN:** sirve para saber el valor mínimo dentro de los campos de una columna.

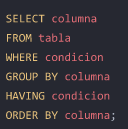
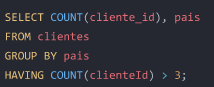


**AVG:** sirve para saber el promedio de los campos de una columna.



**HAVING**

Cumple la misma función que el where, solo que está hecho para poder ser usado en funciones de agrupación, es decir, cuando usamos un ORDER BY. HAVING proporciona una condición a la consulta cuando esta está agrupada por algún criterio.

**INTRODUCCIÓN A MODELOS Y CONSULTAS BÁSICAS**

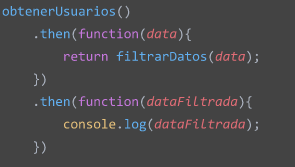
**MODELO:** es una representación de la DB en el proyecto.

**Promesas**

Se utilizan para ejecutar código asincrónico en cascada, es decir, un paso tras otro. Las promesas se establecen como una función llamada **.then()** y como parámetro recibe un callback cuyo parámetro (no importa el nombre que le demos) recibirá el return de la función que se ejecutó previamente.

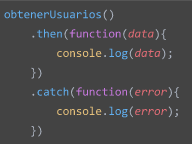
**.then():**

En el siguiente caso, la función **obtenerUsuarios()** trae info desde la DB y una vez que esta función se ejecuta por completo da paso al then en el que se pasa como parámetro de callback lo que retornó la función obtenerUsuarios y se ejecuta la función filtrarDatos con dicha data como parámetro. Luego de que terminó de filtrar salta al otro then y muestra por consola el resultado de dicha filtración.



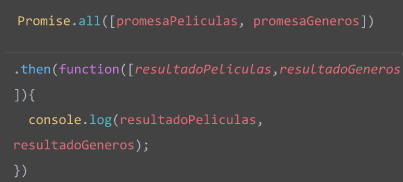
**.catch():**

En el caso de los catch, estos se colocan a lo último de la función y sirve para atrapar los errores que surjan de la ejecución de uno o varios .then. Con esto podemos redirigir la página a otra producto del error o mostrar el error por pantalla.



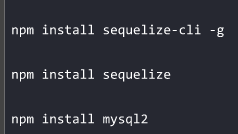
**Promise.all():**

Para los casos en los que necesitamos que dos o más promesas se resuelvan para realizar cierta acción podemos utilizar Promise.all(). Debemos primero almacenar en variables los métodos que queremos ejecutar y luego pasar como parámetro del all() un array con dichas variables. Una vez que se ejecuta, recibimos en los parámetro del primer .then los return de cada una de estas funciones ejecutadas:



**SEQUELIZE**

**Instalación del paquete:**

****

Creamos archivo **.sequelizerc** en la raíz del proyecto:

**const path = require('path')**

**module.exports = {**

**config: path.resolve('./src/database/config', 'config.js'),**

**'models-path': path.resolve('./src/database/models'),**

**'seeders-path': path.resolve('./src/database/seeders'),**

**'migrations-path': path.resolve('./src/database/migrations'),**

**}**

Desde la consola: **sequelize init** para que se generen las carpetas que establecimos en el archivo anterior.

En la carpeta config, hacemos un **module.exports** del JSON del archivo config.js luego de que lo modifiquemos con nuestras credenciales, dependiendo del ambiente en el que vayamos a trabajar (dev, test, prod):



**OBJETO DB**

Todas las configuraciones anteriores las establecemos para que sequelize conecte nuestra aplicación a la base de datos y a través del archivo **index.js** ubicado en la carpeta **models** podemos exportar el OBJ DB

**Models:** siempre creamos un modelo por cada tabla de nuestra DB. Este se guarda en **/database/models** y los nombres de estos modelos se escriben en UpperCamelCase y van en singular.

**Crear un modelo:**

**1-** Generar en el archivo un **module.exports=**

**2-** Asignamos como valor del modulo: **sequelize.DEFINE()**

**Cumple la misma función que (CREATE TABLE)**, solo que no la crea sino que le pasamos como parámetros cómo es nuestra tabla.

Para ejecutarlo vamos a definir una variable que representara la tabla que estamos creando y como valor le asignaremos el objeto sequelize con el método **.define()** al cual le daremos tres parámetros:

**alias:**  será el nombre con el que ubicamos dentro de nuestro programa a la tabla.

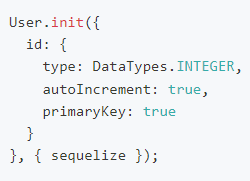
**cols:** luego un objeto literal cuyo nombre de propiedades será el nombre de las columnas y el value estará compuesto por el tipo de columna que será. En caso de que queramos agregar constrains podemos no agregar un value a la key si no pasar otro objeto literal como value específicando el tipo de dato y los constrains que tendrá.

**LOS dataTypes SE PUEDEN VER DESDE LA DOCUMENTACIÓN OFICIAL:**

[**https://sequelize.org/docs/v6/core-concepts/model-basics/#data-types**](https://sequelize.org/docs/v6/core-concepts/model-basics/#data-types)

**config:** por último podemos pasar un tercer parámetro que será la configuración de la tabla en la que especificamos el tableName y también el timestamps si es true o false. Este último se refiere a si la tabla posee dos columnas que guardan un registro de creación y modificación en sí misma.

**3- return** de la variable que contiene toda la configuración anterior.



En caso de que no queramos pasar los parámetros sequelize y dataTypes en el module.exports, pordemos hacer el require del objeto Sequelize y también de   
**sequelize = require(‘../database’)** para llamar a la estructura de database que tenemos en nuestro proyecto. Luego, los dataTypes los asignamos con Sequelize.STRING (por ejemplo) y usamos sequelize para poder invocar al método define().

**ENLAZAR EL MODELO CON EL CONTROLADOR**

**find:** cumple la misma función que **SELECT** pero tiene variantes:

**findAll():** cumple la función de **SELECT \* FROM.** Solo demos pasar el método a:

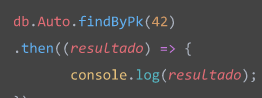
**db.AliasModelo.findAll()**

****

**findOne():** cumple la función de **SELECT \* FROM WHERE.** El where se lo vamos a pasar como parámetro, este será un objeto literal cuya key será la columna del where y el valor será la condición a cumplirse.



**findByPK():** cumple la función de **SELECT \* FROM tabla WHERE id = parámetro;**

****

**Conectar el controller con el model:**

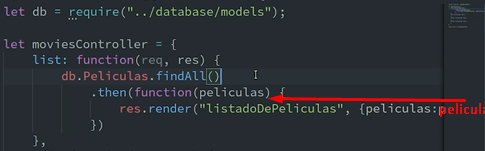
**1-** Debemos definir el modelo y exportarlo.

**2-** Debemos requerir en el controlador el objeto

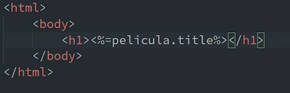
**db (const db = require(‘../database/models’)**

**3-** En la función del objeto literal del controlador, instanciamos el objeto **db** seguido del alias del modelo que queremos utilizar y a eso le colocamos la función find que queramos utilizar.

**4-** A la instanciacion del obj db le agregamos un **.then** cuyo parámetro será el resultado de la ejecución de la consulta a la base de datos y como contenido de este callback pasaremos la consecuencia a ejecutar.   
Por lo general esta consecuencia es un render cuyo primer parámetro será el nombre de la vista que queremos dirigir al usuario y como segundo parámetro pasamos un objeto literal cuyo key será el nombre que queramos utilizar en la vista para esta variable y el value será el nombre del parámetro que pasamos en el callback del then.

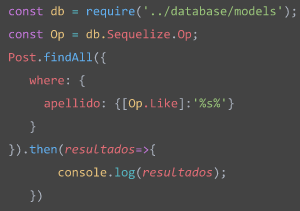


**5-** En la vista, mediante ejs, hacemos uso de la key peliculas que pasamos desde el obj. literal que establecimos como parámetro del render en el controler, para mostrar cualquier atributo de esa key peliculas (peliculas.nombre, por ejemplo).



**WHERE:** para utilizar where y poder filtrar los resultados según una condició específica, pasamos un objeto literal como parámetro del find y como key asignamos **where** y como value otro objeto literal que tendrá como key el nombre la columna que queremos filtrar y como value ese valor que queremos especificar. Esto así solo sirve para “=”, es decir, cuando buscamos coincidencias dentro del objeto que definimos en el where sobre la consulta.

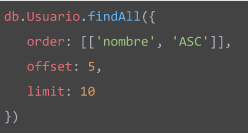
**Operadores:** en caso de que queramos hacer uso de otros operadores (mayor que, menor que, etc…) podemos hacer uso de una variable op cuyo valor sea **db.Sequelize.Op.** Al tener esta varibale instanciada solo debemos pasar en el value de la propiedad del objeto literal del where, entre corchetes, la variable Op.NOMBREOPERADOR. Este NOMBREOPERADOR hará referencia al operador en cuestión que podemos ver en (<https://sequelize.org/v5/manual/querying.html>) y luego lo único que hacemos es indicar el value con el que va a trabar dicho operador.



**ORDER:** el order se establece también dentro del objeto literal que se pasa como parámetro del método. Es una propiedad más como lo es el where y en el order se pasa un array de arrays, es decir, order tendrá como value un array, que a su vez contendrá un array en el que especificamos en el indice 0 el nombre de la columna y como indice 1 ASC o DESC.

**LIMIT y OFFSET:** esta propiedad es más sencilla, ya que se agrega también como atributo del objeto literal que pasamos como parámetro del metodo pero simplemente le damos un número como value.

**ORDER + LIMIT + OFFSET:**

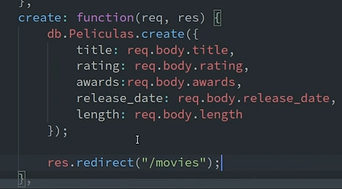


**CREATE()**

Cumple la función de hacer un INSERT INTO en la tabla, es decir, cargar nuevos registros en la tabla. Para hacerlo instanciamos el objeto DB y le indicamos el alias de la base de datos en la que queremos trabajar y a la referencia le damos el metodo create. Como parámetro del método vamos a pasar un objeto literal que tendrá que como Key el nombre de la columna de la tabla a la que le queremos agregar información y dicha información la pasamos como value.



Desde el controlador lo que hacemos es pasarle como valor a las columnas del objeto literal cada una de las key que viajaron en el body a través de post.



**UPDATE():**

Este método cumple la misma función del update de sql, es decir, actualizar la información de un registro existente en la tabla. Para actualizar, instanciamos también el objeto DB, le pasamos el alias de la tabla que queremos afectar y por último el método update, el cual tendrá **DOS parámetroS**, el primero será un objeto literal con las columnas que queramos afectar como key y nuevo valor que reemplaza como value; el segundo objeto literal será un **WHERE** en el que indicaremos con un obj. literal anidad la condición que debe cumplirse para que esta actualización se haga. **ES IMPORTANTE COLOCAR EL WHERE PORQUE SI NO PODEMOS ACTUALIZAR TODOS LOS REGISTROS DE LA TABLA CON LOS VALORES QUE PASAMOS EN EL PRIMER OBJ. LITERAL.**

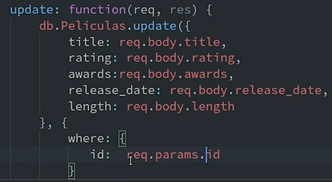
Desde el controlador lo que hacemos es:

**1- Pasar por *get* la información actual** del registro que queremos afectar. Esto lo podemos hacer con un findByPk y que busque dicha información por el id que nos llega desde la url. El resultado lo guardamos en una variable y dicha variable la compartimos como un objeto literal en el render que hacemos a la página:





**2- Recibir por post o patch** toda la información que reemplazó el cliente desde la vista y almacenarla en la base de datos. Para esta ruta creamos una segunda función desde el controlador que genere dicha actualización. Para ello utilizamos el método update y como parámetro le pasamos un objeto literal con las columnas que queremos actualizar y como value le seteamos las key de la info que viajó a través del body (title: req.body.title). Por último pasamos un segundo objeto literal como parámetro del método update que contendrá un where y su valor será un segundo objeto literal en el que indicaremos el id del registro a modificar, el value de este id lo recuperaremos de la url por lo cual será req.params.id. Por último hacemos res.redirect a la página la que queramos mandar al cliente.



**DESTROY():**

Cumple la misma función que el DELETE de sql. Pasamos también el método destroy a **db.ALIASTABLA.destroy()** y como parámetro pasamos únicamente un objeto literal que contendrá un where con otro obj.literal anidado en el que especificaremos qué registro se debe eliminar. **ES IMPORTANTE TAMBIÉN EL WHERE PORQUE SI NO PODEMOS ELIMINAR TODOS LOS REGISTROS DE LA TABLA.**

Desde el controlador lo que hacemos es generar un función delete y le pasamos también al objeto db.ALIASTABLA la función destroy a quien le pasaremos un objeto literal que contendrá un where cuyo valio será otro objeto literal anidado con la condición que se debe cumplir para que dicho registro se elimine.

