

**Trabajo Práctico de Investigación**

**Acceso a datos con Node.Js y ORM**

**Grupo 10**

**Alumnos**

***● Acosta, Mauro (mauroacostactes@gmail.com)***

***● D´Antiochia, Conrado Cesar Augusto (***[***conradocesaraugustodantiochia@gmail.com***](mailto:conradocesaraugustodantiochia@gmail.com)

***● Medina Villamayor, Alexis***

***● Mauriño, Emilio Raúl (***[***maurinoemilio@gmail.com***](mailto:maurinoemilio@gmail.com)***)***

***● Villa, Ricardo (***[***tec.villa.ricardo@gmail.com***](mailto:tec.villa.ricardo@gmail.com)***)***

***● Bustamante, Matias Iván (matybustamante151@gmail.com)***

**2021**

Tabla de contenido

[¿Qué es NodeJs? 3](#_Toc69832341)

[¿Para que sirve Node.Js? 3](#_Toc69832342)

[¿Qué es una ORM? 4](#_Toc69832343)

[¿Por es mejor un ORM que otro lenguaje de programación? 4](#_Toc69832344)

[ORM (Sequelize) 4](#_Toc69832345)

[Modelo 4](#_Toc69832346)

[Definición de Modelos 4](#_Toc69832347)

[Sincronización de Modelos 6](#_Toc69832348)

[Sincronizar todos los modelos a la vez 6](#_Toc69832349)

[Instancia del Modelo 6](#_Toc69832350)

[Consulta de Modelo 7](#_Toc69832351)

[Validaciones y Restricciones 9](#_Toc69832352)

[Diferencia entre validaciones y restricciones 10](#_Toc69832353)

[Biblioteca de Conectores 10](#_Toc69832354)

[Migraciones 12](#_Toc69832355)

# ¿Qué es NodeJs?

NodeJs es un entorno de tiempo de ejecución de JavaScript (de ahí su terminación en .Js).

NodeJs fue creado por los desarrolladores originales de JavaScript. Lo transformaron en algo que solo se podía ejecutarse en el navegador en algo que se podría ejecutar en los ordenadores como si de aplicaciones independiente se tratara . Tanto JavaScript como NodeJs se ejecutan en el motor de tiempo de ejecución JavaScript V8 (V8 es el nombre del motor de JavaScript que alimenta Google Chrome).

# ¿Para que sirve Node.Js?

Node.Js utiliza un modelo de entrada y salida sin bloqueo controlado por eventos que lo hace ligero y eficiente (con entrada nos referimos a solicitudes y con salidas a respuestas). Puede referirse cualquier operación desde leer o escribir archivos de cualquier tipo hasta hacer una solicitud HTTP.

La idea principal de Node.Js es usar un modelo de entrada y salida sin bloqueo y controlado por eventos para seguir siendo liviano y eficiente frente a las aplicaciones en tiempo real de uso de datos que se ejecutan en los dispositivos.

La finalidad de Node.Js no tiene su objetivo en operaciones intensivas en el uso del procesador, de hecho de usarlo para programación de más peso eliminara casi todas sus ventajas.

Donde Node.Js realmente brilla es en la creación de aplicaciones de red rápidas, ya que es capaz de manejar gran cantidad de conexiones simultáneas con un alto nivel de rendimiento, lo que equivale a una alta escalabilidad.

Ventaja

* Tiene incorporado JavaScript en la plataforma de Node.Js, siendo un lenguaje fácil de aprender.
* Se desarrolla en un entorno de tiempo de ejecución de fuentes libres que ayudara en el almacenamiento de creación de proyectos únicos.
* El modelo de entrada y salida impulsado por eventos ayuda mucho en el manejo simultaneo de peticiones

# ¿Qué es una ORM?

Una ORM procede de las siglas (Object Relational Mapping) Es un modelo de programación que transforma las tablas de las bases de datos en entidades para simplificar enormemente la tarea del programador.

El trabajo deja de ser manual ya que el ORM lo realizara de forma independiente de la base de datos. Además, gracias al mapeo automático podrás cambiar el motor de la base de datos fácilmente.

# ¿Por es mejor un ORM que otro lenguaje de programación?

Un ORM tiene algunas ventajas y desventajas también. Sus ventajas son la facilidad y velocidad de uso, la seguridad contra ataques informáticos o la forma de abstracción de la base de datos que estemos utilizando.

Por otro lado, para programar con ORM es necesario aprender su lenguaje y hay entornos con gran volumen que puede ver mermado su rendimiento.

Uno de los mapeos automáticos más utilizado es de Java y se llama Hibernate, pero también están iBatis, Ebean, para .Net nHibernate , Entity Framework, entre otros.

# ORM (Sequelize)

Es un ORM de Node.js basado en promesas para Postgres, MySql, MariaDB, SQLite, Microsoft SQL Server. Cuenta con un sólido soporte de transacciones, relaciones, carga ansiosa, y perezosa, replicación de lectura y mucho más.

Comando para instalar el módulo de Sequelize

$ npm install --save sequelize

## Modelo

Un modelo es una abstracción que representa una tabla en una base de datos. En Sequelize, es una clase que extiende de Model.

El modelo le dice a Sequelize varias cosas sobre la entidad que representa, como el nombre de la tabla de la bases de datos y que columna tiene y sus tipos.

Un modelo en Sequelize tiene un nombre. Este nombre no tiene que ser el mismo nombre de la tabla que representa en la base de datos. Por lo general, los modelos tienen nombres en singular (como *User)* mientras que las tablas tienen nombres en plural (como *Users*).

### Definición de Modelos

Los modelos se pueden definir de dos maneras equivalentes en Sequelize:

* Sequelize.define(modelName, attributes, options)
* Extendiendo el modelo y llamando init(attributes, options)

Utilizando sequealize.define

const { Sequelize, DataTypes } = require('sequelize');

const sequelize = new Sequelize('sqlite::memory:');

const User = sequelize.define('User', {

// Model attributes are defined here

firstName: {

type: DataTypes.STRING,

allowNull: false

},

lastName: {

type: DataTypes.STRING

// allowNull defaults to true

}

}, {

// Other model options go here

});

// `sequelize.define` also returns the model

console.log(User === sequelize.models.User); // true

Modelo Extensible

const { Sequelize, DataTypes, Model } = require('sequelize');

const sequelize = new Sequelize('sqlite::memory');

class User extends Model {}

User.init({

// Model attributes are defined here

firstName: {

type: DataTypes.STRING,

allowNull: false

},

lastName: {

type: DataTypes.STRING

// allowNull defaults to true

}

}, {

// Other model options go here

sequelize, // We need to pass the connection instance

modelName: 'User' // We need to choose the model name

});

// the defined model is the class itself

console.log(User === sequelize.models.User); // true

### Sincronización de Modelos

Cuando define un modelo, le está diciendo a Sequelize algunas cosas sobre su tabla en la base de datos. Sin embargo, ¿qué pasa si la tabla ni siquiera existe en la base de datos? ¿Qué pasa si existe, pero tiene diferentes columnas, menos columnas o cualquier otra diferencia?

Aquí es donde entra en juego la sincronización del modelo. Un modelo se puede sincronizar con la base de datos llamando a [model.sync(options)](https://sequelize.org/master/class/lib/model.js~Model.html#static-method-sync)una función asincrónica (que devuelve una Promise). Con esta llamada, Sequelize realizará automáticamente una consulta SQL a la base de datos. Tenga en cuenta que esto cambia solo la tabla en la base de datos, no el modelo en el lado de JavaScript.

* User.sync() - Esto crea la tabla si no existe (y no hace nada si ya existe)
* User.sync({ force: true }) - Esto crea la tabla, soltándola primero si ya existía
* User.sync({ alter: true }) - Esto verifica cuál es el estado actual de la tabla en la base de datos (qué columnas tiene, cuáles son sus tipos de datos, etc.), y luego realiza los cambios necesarios en la tabla para que coincida con el modelo.

Ejemplo

await User.sync({ force: true });

console.log("The table for the User model was just (re)created!");

### Sincronizar todos los modelos a la vez

Puede utilizar [sequelize.sync()](https://sequelize.org/master/class/lib/sequelize.js~Sequelize.html#instance-method-sync)para sincronizar automáticamente todos los modelos.

Ejemplo

await sequelize.sync({ force: true });

console.log("All models were synchronized successfully.");

### Instancia del Modelo

Una instancia de la clase representa un objeto de ese modelo (que se asigna a una fila de la tabla en la base de datos). De esta forma, las instancias de modelo son [DAO](https://en.wikipedia.org/wiki/Data_access_object) .

const { Sequelize, Model, DataTypes } = require("sequelize");

const sequelize = new Sequelize("sqlite::memory:");

const User = sequelize.define("user", {

name: DataTypes.TEXT,

favoriteColor: {

type: DataTypes.TEXT,

defaultValue: 'green'

},

age: DataTypes.INTEGER,

cash: DataTypes.INTEGER

});

(async () => {

await sequelize.sync({ force: true });

// Code here

})();

### Consulta de Modelo

#### Consulta INSERT

Primero, un ejemplo simple:

// Create a new user

const jane = await User.create({ firstName: "Jane", lastName: "Doe" });

console.log("Jane's auto-generated ID:", jane.id);

El [Model.create()](https://sequelize.org/master/class/lib/model.js~Model.html#static-method-create)método es una forma abreviada de crear una instancia sin [Model.build()](https://sequelize.org/master/class/lib/model.js~Model.html#static-method-build)guardar y guardar la instancia con [instance.save()](https://sequelize.org/master/class/lib/model.js~Model.html#instance-method-save).

También es posible definir qué atributos se pueden configurar en el createmétodo. Esto puede ser especialmente útil si crea entradas de base de datos basadas en un formulario que puede ser llenado por un usuario. Usar eso, por ejemplo, le permitiría restringir el Usermodelo para establecer solo un nombre de usuario y una dirección, pero no una marca de administrador:

const user = await User.create({

username: 'alice123',

isAdmin: true

}, { fields: ['username'] });

// let's assume the default of isAdmin is false

console.log(user.username); // 'alice123'

console.log(user.isAdmin); // false

#### Consulta SELECT

Puede leer la tabla completa de la base de datos con el [findAll](https://sequelize.org/master/class/lib/model.js~Model.html#static-method-findAll) método.

// Find all users

const users = await User.findAll();

console.log(users.every(user => user instanceof User)); // true

console.log("All users:", JSON.stringify(users, null, 2));

SELECT \* FROM ...

Especificar atributos para consultas SELECT

Para seleccionar solo algunos atributos, puede usar la attributesopción:

Model.findAll({

attributes: ['foo', 'bar']

});

SELECT foo, bar FROM ...

Se puede cambiar el nombre de los atributos mediante una matriz anidada:

Model.findAll({

attributes: ['foo', ['bar', 'baz'], 'qux']

});

SELECT foo, bar AS baz, qux FROM ...

Puede usar [sequelize.fn](https://sequelize.org/master/class/lib/sequelize.js~Sequelize.html#static-method-fn)para hacer agregaciones:

Model.findAll({

attributes: [

'foo',

[sequelize.fn('COUNT', sequelize.col('hats')), 'n\_hats'],

'bar'

]

});

SELECT foo, COUNT(hats) AS n\_hats, bar FROM ...

Cuando utilice la función de agregación, debe darle un alias para poder acceder a ella desde el modelo. En el ejemplo anterior, puede obtener el número de sombreros con instance.n\_hats.

A veces puede resultar tedioso enumerar todos los atributos del modelo si solo desea agregar una agregación:

// This is a tiresome way of getting the number of hats (along with every column)

Model.findAll({

attributes: [

'id', 'foo', 'bar', 'baz', 'qux', 'hats', // We had to list all attributes...

[sequelize.fn('COUNT', sequelize.col('hats')), 'n\_hats'] // To add the aggregation...

]

});

// This is shorter, and less error prone because it still works if you add / remove attributes from your model later

Model.findAll({

attributes: {

include: [

[sequelize.fn('COUNT', sequelize.col('hats')), 'n\_hats']

]

}

});

SELECT id, foo, bar, baz, qux, hats, COUNT(hats) AS n\_hats FROM ...

#### Consulta de Actualización

Las consultas de actualización también aceptan la where opción, al igual que las consultas de lectura que se muestran arriba.

// Change everyone without a last name to "Doe"

await User.update({ lastName: "Doe" }, {

where: {

lastName: null

}

});

#### Consulta de Eliminación

Las consultas de eliminación también aceptan la where opción, al igual que las consultas de lectura que se muestran arriba.

// Delete everyone named "Jane"

await User.destroy({

where: {

firstName: "Jane"

}

});

Para destruir todo lo que TRUNCATEse puede usar el SQL:

// Truncate the table

await User.destroy({

truncate: true

});

## Validaciones y Restricciones

const { Sequelize, Op, Model, DataTypes } = require("sequelize");

const sequelize = new Sequelize("sqlite::memory:");

const User = sequelize.define("user", {

username: {

type: DataTypes.TEXT,

allowNull: false,

unique: true

},

hashedPassword: {

type: DataTypes.STRING(64),

is: /^[0-9a-f]{64}$/i

}

});

(async () => {

await sequelize.sync({ force: true });

// Code here

})();

### Diferencia entre validaciones y restricciones

Las validaciones son comprobaciones realizadas en el nivel Sequelize, en JavaScript puro. Pueden ser arbitrariamente complejos si proporciona una función de validación personalizada, o pueden ser uno de los validadores integrados que ofrece Sequelize. Si falla una validación, no se enviará ninguna consulta SQL a la base de datos.

Por otro lado, las restricciones son reglas definidas a nivel de SQL. El ejemplo más básico de restricción es una restricción única. Si falla una verificación de restricción, la base de datos arrojará un error y Sequelize enviará este error a JavaScript (en este ejemplo, arrojará un SequelizeUniqueConstraintError). Tenga en cuenta que en este caso se realizó la consulta SQL, a diferencia del caso de las validaciones.

Permitir y Rechazar valores nulos

De forma predeterminada, nulles un valor permitido para cada columna de un modelo. Esto se puede deshabilitar configurando la allowNull: false opción para una columna.

/\* ... \*/ {

username: {

type: DataTypes.TEXT,

allowNull: false,

unique: true

},

} /\* ... \*/

## Biblioteca de Conectores

MySQL

La biblioteca de conectores subyacente utilizada por Sequelize para MySQL es el paquete [mysql2](https://www.npmjs.com/package/mysql2) npm (versión 1.5.2 o superior).

Puede proporcionarle opciones personalizadas usando dialectOptionsen el

constructor Sequelize:

const sequelize = new Sequelize('database', 'username', 'password', {

dialect: 'mysql',

dialectOptions: {

// Your mysql2 options here

}

})

MariaDB

La biblioteca de conectores subyacente utilizada por Sequelize para MariaDB es el paquete [mariadb](https://www.npmjs.com/package/mariadb) npm.

Puede proporcionarle opciones personalizadas usando dialectOptionsen el constructor Sequelize:

const sequelize = new Sequelize('database', 'username', 'password', {

dialect: 'mariadb',

dialectOptions: {

// Your mariadb options here

// connectTimeout: 1000

}

});

SQLite

La biblioteca de conectores subyacente utilizada por Sequelize para SQLite es el paquete [sqlite3](https://www.npmjs.com/package/sqlite3) npm (versión 4.0.0 o superior).

El archivo de almacenamiento se especifica en el constructor Sequelize con la storageopción (utilizar :memory:para una instancia de SQLite en memoria).

Puede proporcionarle opciones personalizadas usando dialectOptionsen el constructor Sequelize:

const sequelize = new Sequelize('database', 'username', 'password', {

dialect: 'sqlite',

storage: 'path/to/database.sqlite' // or ':memory:'

dialectOptions: {

// Your sqlite3 options here

}

});

PostgreSQL

La biblioteca de conectores subyacente utilizada por Sequelize para PostgreSQL es el paquete [pg](https://www.npmjs.com/package/pg) npm (versión 7.0.0 o superior). El módulo [pg-hstore](https://www.npmjs.com/package/pg-hstore) también es necesario.

Puede proporcionarle opciones personalizadas usando dialectOptionsen el constructor Sequelize:

const sequelize = new Sequelize('database', 'username', 'password', {

dialect: 'postgres',

dialectOptions: {

// Your pg options here

}

});

Para conectarse a través de un socket de dominio Unix, especifique la ruta al directorio del socket en la hostopción. La ruta del socket debe comenzar con /.

const sequelize = new Sequelize('database', 'username', 'password', {

dialect: 'postgres',

host: '/path/to/socket\_directory'

});

MSSQL

La biblioteca de conectores subyacente utilizada por Sequelize para MSSQL es el [tedioso](https://www.npmjs.com/package/tedious) paquete npm (versión 6.0.0 o superior).

Puede proporcionarle opciones personalizadas usando dialectOptions.optionsen el constructor Sequelize:

const sequelize = new Sequelize('database', 'username', 'password', {

dialect: 'postgres',

dialectOptions: {

// Observe the need for this nested `options` field for MSSQL

options: {

// Your tedious options here

useUTC: false,

dateFirst: 1

}

}

});

## Migraciones

Al igual que usa sistemas de [control de versiones](https://en.wikipedia.org/wiki/Version_control) como [Git](https://en.wikipedia.org/wiki/Git) para administrar los cambios en su código fuente, puede usar las **migraciones** para realizar un seguimiento de los cambios en la base de datos. Con las migraciones, puede transferir su base de datos existente a otro estado y viceversa: esas transiciones de estado se guardan en archivos de migración, que describen cómo llegar al nuevo estado y cómo revertir los cambios para volver al estado anterior.

Necesitará [Sequelize Command-Line Interface (CLI)](https://github.com/sequelize/cli) . La CLI incluye soporte para migraciones y arranque de proyectos.

Una migración en Sequelize es un archivo javascript que exporta dos funciones, up y down que dictan cómo realizar la migración y deshacerla. Define esas funciones manualmente, pero no las llama manualmente; la CLI los llamará automáticamente. En estas funciones, simplemente debe realizar las consultas que necesite, con la ayuda de sequelize.querycualquier otro método que Sequelize le proporcione. No hay magia adicional más allá de eso.

Instalación del CLI

Para instalar la CLI de Sequelize:

npm install --save-dev sequelize-cli

https://fp.uoc.fje.edu/blog/que-es-un-orm-y-por-que-utilizarlo-en-el-desarrollo-de-aplicaciones/

https://openwebinars.net/blog/que-es-nodejs/