Curso de Back-End con Node.js (Inicial)

Clase 08

Temario

Temario

- ¿Cómo hemos organizado nuestro código hasta ahora?
- Patrones de diseño.
- MVC.
- ORM.
- Sequelize.
    - Métodos más importantes
    - Otras funcionalidades.

¿Cómo hemos organizado nuestro código hasta ahora?

¿Cómo hemos organizado nuestro código hasta ahora?

Como se ha mencionado anteriormente, Express es un framework minimalista y no-opinado.

Por lo tanto, Express no indica cómo organizar nuestro código. Por ejemplo, no indica qué archivos crear ni en qué carpetas colocarlos. El desarrollador tiene mucha libertad, lo cual conlleva a una gran responsabilidad.

Esto puede ser problemático en caso de programadores inexpertos ya que fácilmente pueden terminar con un código totalmente desorganizado, difícil de entender, difícil de mantener y tal vez hasta inseguro y poco performante.

¿Cómo hemos organizado nuestro código hasta ahora?

Algunas decisiones que hemos tomado hasta el momento son:

- Crear un archivo `routes.js` donde colocar todas las rutas de la aplicación.
- Crear un directorio `/controllers` donde colocar los archivos que contienen los *handlers* de las rutas.
- Crear un directorio `/views` donde colocar las vistas de la aplicación.
- Crear un archivo `db.js` donde colocar la información de conexión a la base de datos.

🤔 ¿Estuvieron bien estas decisiones? Si otros programadores vieran nuestro código, ¿sabrían a qué hace referencia (y para qué sirve) cada directorio y/o archivo? ¿No existirá algún *"standard"* en la industria que podamos seguir? → ¡Afortunadamente sí!

Patrones de Diseño

Patrones de Diseño

Un patrón de diseño es una solución reusable a un problema común que se da en el desarrollo de software.

El término ganó popularidad en el año 1994, cuando se publicó el libro "Design Patterns: Elements of Reusable Object-Oriented Software" por "Gang of Four" (Gamma et al.), frecuentemente abreviado como "GoF". Es de los libros más nombrados en la carrera de Ingeniería en Computación / Sistemas.

Los patrones no son soluciones "exactas", son un esqueleto.

👉 Aquí pueden ver un libro online sobre patrones, con foco en JavaScript.

MVC

MVC (1/4)

MVC es un patrón de diseño arquitectónico usado por muchos frameworks de desarrollo web (como AdonisJS en Node.js, Laravel en PHP y Rails en Ruby).

MVC propone dividir una aplicación en 3 grandes componentes con responsabilidades bien definidas con el objetivo de fomentar el reuso de código y permitir el desarrollo en paralelo (se puede trabajar en simultáneo en cada componente).

Los componentes propuestos por MVC son:

- Modelos.
- Vistas.
- Controladores.

} ¡Con estos dos ya hemos trabajado!

MVC (2/4)

Modelo:

- Es el componente central del patrón MVC.
- Expresa el comportamiento de la aplicación en términos del problema de negocio, de forma independiente a la interfaz (no importa si es una web, desktop app o mobile app).
- Gestiona los datos y reglas del negocio.
- En general, interactúa con una base de datos.
- Suele “modelar” (de ahí el nombre “modelo”) una entidad del problema.
  Ej: el modelo Usuario, el modelo Proveedor, el modelo Empleado, el modelo Producto, etc.
- En general, cada modelo se coloca en un archivo con el nombre del modelo (ej: `User.js`) dentro de un directorio llamado `/models`. Y, en general, para definir un modelo se utiliza una `class`.

MVC (3/4)

Vista:

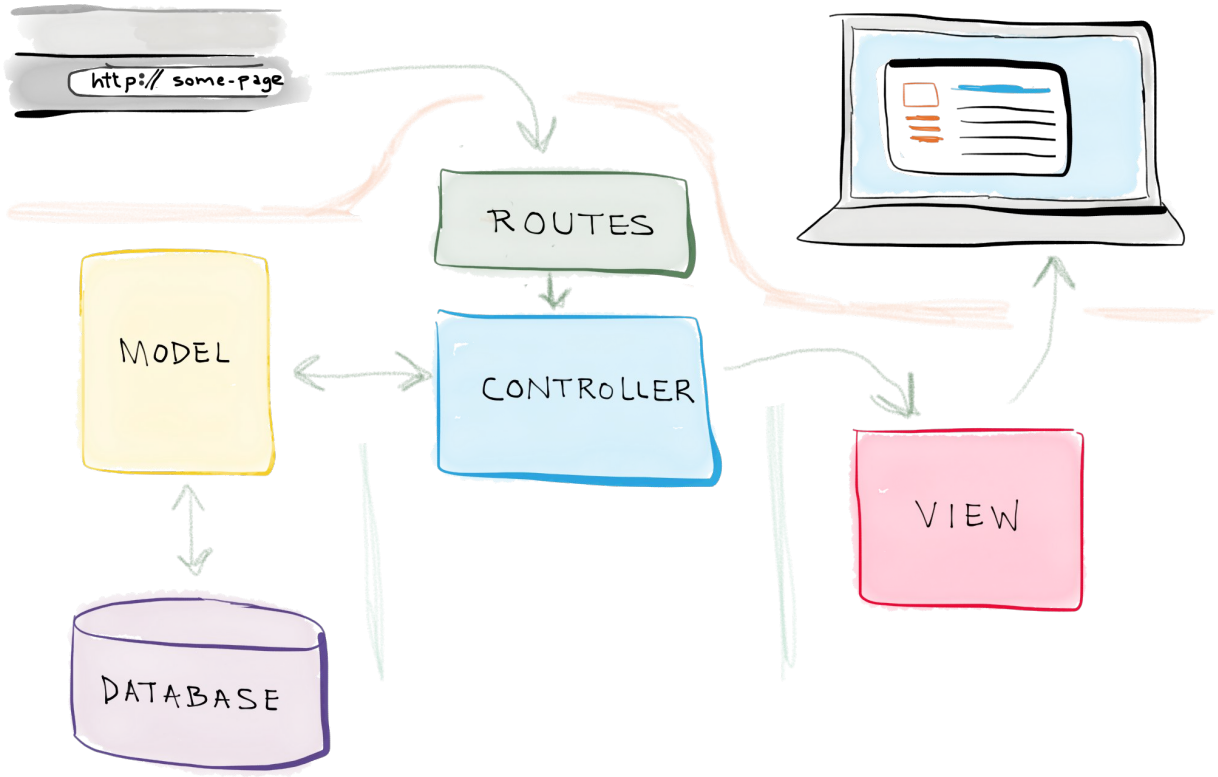
- Se encarga de presentar los datos del modelo. Es la interfaz visual (UI).
- Puede ser una web en HTML, una mobile o desktop app o incluso la pantalla de un cajero automático.
- En el caso de una web app “tradicional” (*server-side rendering*), las vistas se suelen colocar en un directorio llamado `/views`.

Controlador:

- Es un intermediario entre la vista y el modelo.
- Recibe un *request* de un navegante a través de una URL. Es el *handler* de las rutas.
- Se comunica con los modelos para obtener la información necesaria, la organiza y se la entrega a la vista.
- En general, los controladores se colocan en un directorio llamado `/controllers`.

MVC (4/4)

Diagrama:



ORM

ORM

Un ORM (Object-Relational Mapping) es una técnica de programación que permite **mapear** modelos de un sistema con tablas de una base de datos relacional.

Al usar un ORM, podemos interactuar con objetos en lugar de tener tener que escribir consultas SQL “a mano”.

Al crear, editar o eliminar un objeto, el ORM construye y ejecuta “por atrás” las consultas necesarias en la base de datos.

En general, los ORM pueden funcionar con más de un DBMS, fácil de cambiar entre uno u otro.

Sequelize

Sequelize (1/5)

Es un ORM para Node.js que funciona con los siguientes DBMS:

- Postgres.
- MySQL.
- MariaDB.
- SQLite.
- Microsoft SQL Server.

Es decir, gracias a Sequelize podremos interactuar con cualquiera de estos DBMS sin necesidad de escribir consultas SQL.

Está basado en promesas, ideal para evitar *Callback Hells*.

Documentación: https://sequelize.org.

Sequelize (2/5)

Instalación:

```
npm i sequelize
npm i mysql2
```

En caso de usar otro DBMS será necesario instalar otro módulo

Luego importarlo con:

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

Sequelize (3/5)

Ejemplo básico. Primero que nada hay que crear una instancia de Sequelize:

```
const sequelize = new Sequelize(
  process.env.DB_DATABASE, // Ej: hack_academy_db
  process.env.DB_USERNAME, // Ej: root
  process.env.DB_PASSWORD, // Ej: root
  {
    host: process.env.DB_HOST, // Ej: 127.0.0.1
    dialect: process.env.DB_CONNECTION, // Ej: mysql
  }
);
```

En caso de usar otro DBMS será necesario instalar otro módulo

Recordar: `process.env` es donde se guardan las variables de entorno, luego de usar el módulo dotenv.

Sequelize (4/5)

Luego es necesario crear un modelo:

User.js

```
class User extends Model {}

User.init(
  {
    fullname: DataTypes.STRING,
    birthday: DataTypes.DATE,
  },
  { sequelize, modelName: "user" }
);
```

Los **modelos** son la esencia de Sequelize. Son una abstracción que representan una tabla de la Base de Datos.

Ver tipos de datos disponibles.

Sequelize (5/5)

Finalmente se utiliza el método `sync` para crear la tabla a partir del modelo.

```
sequelize.sync({ force: true }).then(() => {
  console.log(`¡Las tablas fueron creadas!`);
});
```

⚠️ También es posible usar `async/await`.

👉 En necesario tener cuidado con este método ya que, en caso de que una tabla exista previamente, la misma se borra y se crea de nuevo, perdiendo todos los datos que allí estaban. En un ambiente de desarrollo este no suele ser un problema. Para un ambiente de producción, usar migraciones en lugar de `sync`.

Métodos más importantes

Sequelize – Obtener registros

Ejemplo: obtener todos lo usuarios de la BD:

```
User.findAll().then((users) => {
  console.log(users);
});
```

⚠️ También es posible usar `async/await`.
🙌 Y lo recomendamos.

Ejemplo: obtener el usuario con id = 123.

```
User.findByPk(123).then((user) => {
  console.log(user);
});
```

⚠️ También es posible usar `async/await`.
🙌 Y lo recomendamos.

Sequelize – Crear un registro

Ejemplo: insertar un usuario en la BD:

```
User.create({
  firstname: "María",
  lastname: "Pérez",
  email: "mariaperez@gmail.com",
}).then((user) => {
  console.log(user);
});
```

⚠️ También es posible usar `async/await`.
🙌 Y lo recomendamos.

Sequelize – Eliminar un registro

Ejemplo: eliminar a todos los usuarios llamados “Pablo” de la BD:

```
User.destroy({
  where: {
    firstname: "Pablo",
  },
}).then(() => {
  console.log("¡Usuarios eliminados!");
});
```

⚠️ También es posible usar `async/await`.
🙌 Y lo recomendamos.

Sequelize – Editar varios registros

Ejemplo: modificar el apellido de todos los “Gómez” a “Pérez”:

```
User.update(
  { lastname: "Pérez" },
  {
    where: {
      lastname: "Gómez",
    },
  }
).then(() => {
  console.log("¡Usuarios actualizados!");
});
```

⚠️ También es posible usar `async/await`.
🙌 Y lo recomendamos.

Sequelize – Editar un registro

Ejemplo: modificar el email del usuario con id = 123.

```
User.findByPk(123).then((user) => {
 user
   .update({
     email: "mperez@gmail.com",
   })
   .then((user) => {
     console.log(user);
   });
});
```

⚠️ También es posible usar `async/await`.
🙌 Y lo recomendamos.

Otras funcionalidades

Otras funcionalidades

Recomendamos que investiguen sobre estas otras funcionalidades de Sequelize:

- Validaciones.
- Asociaciones / Relaciones entre entidades.
- Migraciones.
- Sequelize CLI.

👉 Además, recomendamos que miren Sequelize UI, que les podrá ser de gran ayuda a la hora de crear los modelos y las relaciones entre los mismos.

Ejercicio 1

Ejercicio 1

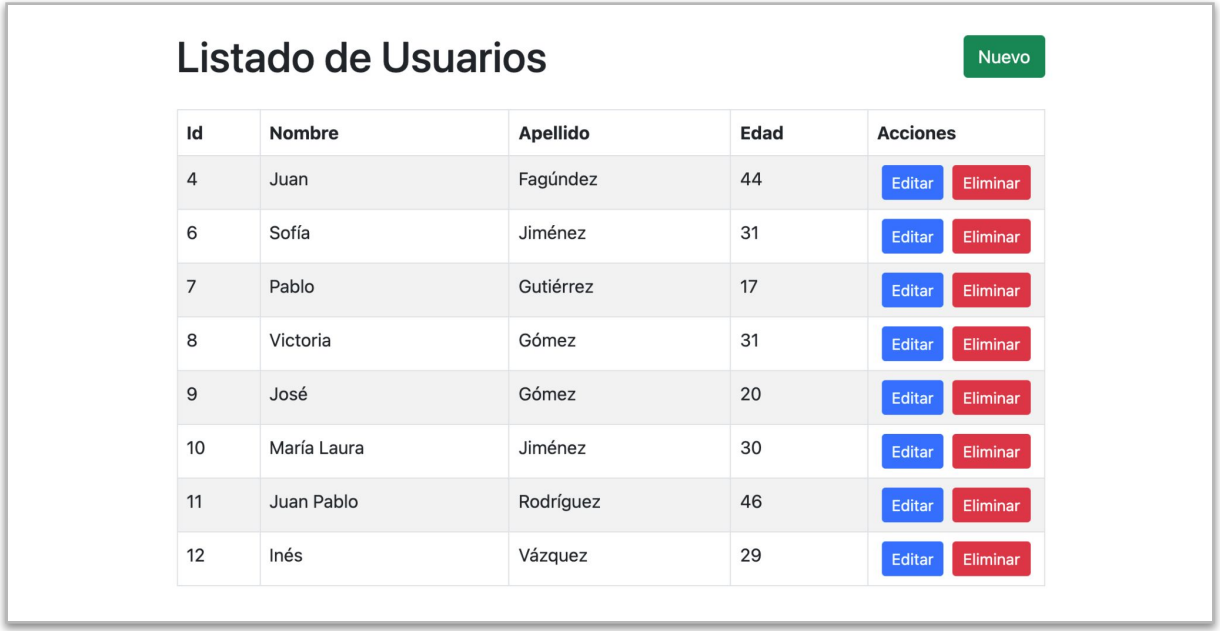
Re-hacer el ejercicio hecho con la librería `mysql2` pero ahora usando Sequelize.

👉 Ver las siguientes diapositivas de ejemplo.

Esto implica hacer un CRUD de usuarios:

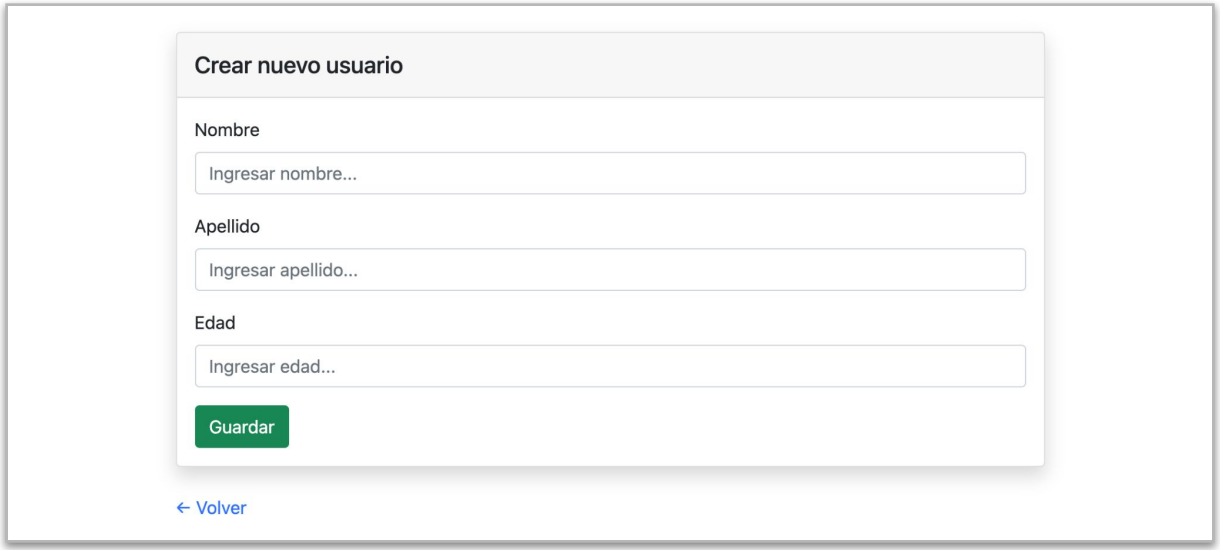
- C – Create.
- R – Read.
- U – Update.
- D – Delete.

Ejercicio 1 (cont)



URL: `/usuarios`

Ejercicio 1 (cont)



URL: `/usuarios/crear`

Ejercicio 1 (cont)

URL: `/usuarios/modificar/4`