Arquitectura del proyecto

Introducción

La arquitectura de un proyecto de software ayuda a los miembros del equipo a entender cómo se organizarán y relacionarán entre sí los diferentes componentes del sistema para lograr los objetivos del proyecto. Por lo tanto, una buena arquitectura puede mejorar la eficiencia y la eficacia de la organización dentro del equipo de desarrollo.

Para ello, la arquitectura del proyecto debe ser clara, completa y coherente. Es decir, debe describir de manera detallada cómo funcionará cada componente del sistema y cómo se relacionarán entre sí, para que los miembros del equipo puedan entender su rol y contribución específica en el proyecto.

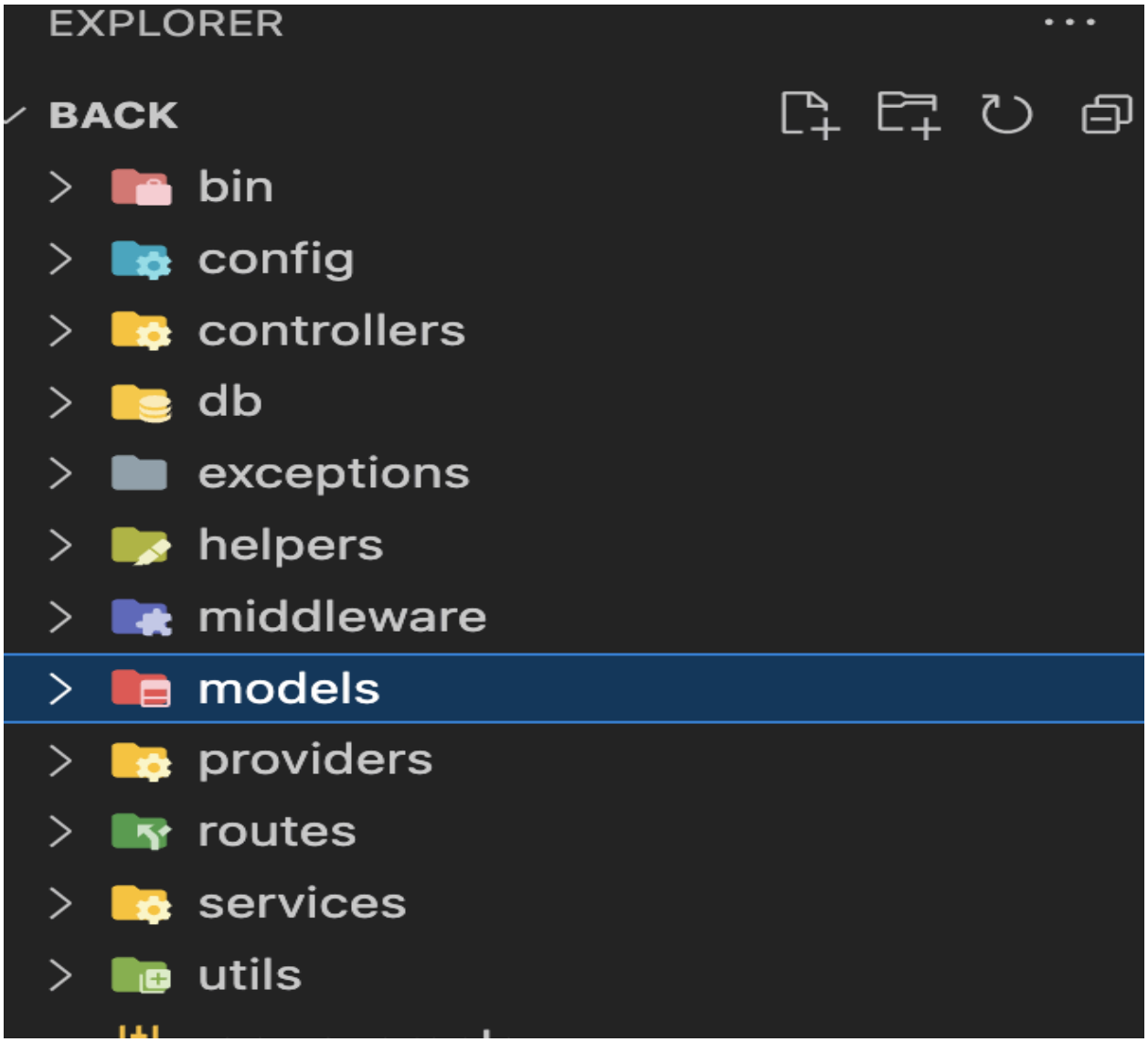
La arquitectura del proyecto también debe ser flexible y escalable para adaptarse a los cambios en los requisitos del proyecto y para permitir el crecimiento del sistema a medida que se agregan nuevos componentes. Por lo tanto, debe permitir la incorporación de nuevos miembros al equipo de desarrollo y la asignación de tareas de manera eficiente.

Además, la arquitectura del software puede establecer pautas para la nomenclatura de archivos y carpetas, lo que ayuda a los miembros del equipo a encontrar fácilmente los archivos y carpetas relevantes. Por ejemplo, los nombres de archivos y carpetas pueden basarse en la función o responsabilidad del archivo o carpeta en el sistema.

Por otro lado, la arquitectura del software también puede definir cómo se gestionarán y mantendrán los archivos y carpetas a lo largo del ciclo de vida del proyecto. Por ejemplo, puede establecer un sistema de control de versiones para asegurarse de que los cambios en los archivos y carpetas estén bien documentados y controlados.

Una organización adecuada de las carpetas y archivos dentro del proyecto puede ser importante porque permite una mejor colaboración y eficiencia entre los miembros del equipo, al proporcionar una forma clara y estructurada de acceder y editar los archivos. Además, una buena organización de archivos y carpetas puede ayudar a reducir la cantidad de errores y conflictos que pueden surgir cuando varias personas trabajan en el mismo proyecto.

Estructura de carpetas de nuestro proyecto



Routes: contiene archivos que definen las rutas para cada endpoint de la API. Estos archivos definen las funciones que se ejecutarán cuando se solicita una ruta específica. Cada archivo de ruta puede definir una serie de rutas y los controladores asociados, que se ejecutarán cuando se accede a una ruta específica. Además, se pueden agrupar archivos de ruta en subcarpetas para mejorar la organización y la legibilidad del código.
Ejemplo:

JavaScript

```
// app.js
const hello_router = require ("./hello_router")
const express = require('express')
const app = express()
app.use ("/hello" , hello_router)
```

JavaScript

```
//hello_router.js
const router = express.Router() //<- actual express router
router.get("/wave-hand" ,(req , resp) => {
// Bussiness Logic.
return resp.json ({status : "ok" , "action" : "wave hand"})
})
//make the router visible to other js files
module.exports = router
```

```javascript
const axios= require("axios")
const resp=await
axios.get("http://localhost:8000/hello/wave-hand")
console.log(resp);
/*
will print
{
      "status" : "ok",
      "action" : "wave hand"
}
*/
```

Controllers: contiene archivos que definen las funciones que se ejecutarán cuando se solicita una ruta específica. El objetivo de la carpeta controllers es separar la lógica de la aplicación de la definición de las rutas, para facilitar la reutilización del código y mejorar la legibilidad del mismo.Cada controlador debe estar definido por un metodo HTTP (POST , PUT , GET ,DELETE)
Ejemplo:

```javascript
//Controller.js

// app.get is the controller in this case we are making a GET
controller

app.get("/wave-hand" ,(req , resp) => {

// Bussiness Logic.

return resp.json ({status : "ok" , "action" : "wave hand"})

})
```

Services: contiene archivos que encapsulan la lógica de negocio de la aplicación, proporcionando una abstracción de alto nivel de las operaciones realizadas por los controladores y otros componentes de la aplicación. El objetivo de la carpeta services es separar la lógica de negocio de la lógica de controlador, lo que ayuda a mantener el controlador simple y centrado en la gestión de solicitudes y respuestas HTTP, mientras que la lógica de negocio se separa en diferentes servicios que se pueden reutilizar en diferentes partes de la aplicación.

Ejemplos:

JavaScript

```
//Controller.js

// app.get is the controller in this case we are making a GET
controller

const userService = require("./services/user")

app.get("/wave-hand" ,(req , resp) => {

// Bussiness Logic.

const response = userService.wave_hand() //cleaner code

return resp.json (response)

})
```

JavaScript

```
// services/user.js

// app.get is the controller in this case we are making a GET
controller

function wave_hand (){

      return {status : "ok" , "action" : "wave hand"}

}

module.exports = wave_hand
```

Middleware: contiene archivos que definen funciones middleware que se ejecutan antes o después de que se maneje una solicitud HTTP por el controlador correspondiente.

JavaScript

```
//middleware.js
function time_logger(req,resp,next) {
      console.log('Time:', Date.now())
      next()
}
module.exports = time_logger
```

JavaScript

```
// app.js
const middlewares= require ("./middleware.js")
const express = require('express')
const app = express()
app.use(middlewares.time_logger) //using the middleware
```

Exceptions: contiene archivos que definen las diferentes custom exceptions que se van a usar en la aplicación, esto nos permite tener una lógica de manejo de errores clara y reutilizable

JavaScript
```
//exceptions/custom_exceptions.js
const name_error = new Error("User Name error")
module.exports = name_error
```

JavaScript
```
//index.js
const custom_exceptions =
require("./exceptions/custom_exceptions")

throw name_error
```

Providers: contiene archivos que definen proveedores de servicios que pueden ser inyectados en diferentes componentes de la aplicación, se utilizan para configurar y crear instancias de otros objetos de la aplicación, como bases de datos, bibliotecas de terceros, servicios web, y otros componentes que la aplicación necesita para funcionar. El objetivo de la carpeta providers es separar la lógica de configuración y creación de objetos de la lógica de negocio y controlador de la aplicación, lo que ayuda a mantener la aplicación modular, escalable y fácilmente configurable.

JavaScript
```
//providers/storage_providers.js
const sequelize = new Sequelize('sqlite::memory:')
export function get_connection() {
     return sequelize
}
```

JavaScript
```
//app.js
```

```
const sequelize = require("providers/storage_providers")
sequelize.get_connection().query("select * from users")
...
```

Models: contiene definiciones de clases o funciones que representan entidades de la aplicación, como usuarios, publicaciones, comentarios, y otros objetos que la aplicación maneja. El objetivo de la carpeta models es separar la lógica de datos de la lógica de negocio y controlador de la aplicación, lo que ayuda a mantener la aplicación modular, escalable y fácilmente configurable.

JavaScript

```
// ./models/user.js

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
```

```
module.exports = User
```

Utils: En utils vamos a tener toda la lógica específica que puede ser usado en cualquier parte de la aplicación y no está relacionada con un área de negocio puntual, como por ejemplo lógica de encriptado o generación de un token.

JavaScript

```
// ./utils/encode64.js
function encodeB64Object(item ) {
      const stringed_item = JSON.stringify(item)
      const b64Object = base64.encode(stringed_item)
      return b64Object
}
function decodeB64string(encodedString) {
      const b64Object = base64.decode(encodedString)
      return b64Object
}
module.exports = {encodeB64Object,decodeB64string}
```

Helpers:contiene archivos que definen funciones de ayuda (helpers) que se utilizan en diferentes partes de la aplicación.
Las funciones de ayuda son funciones que realizan tareas comunes o repetitivas, como la manipulación de cadenas, la generación de identificadores únicos, la validación de datos, y otras tareas que se utilizan en diferentes partes de la aplicación.El objetivo de la carpeta helpers es separar la lógica de ayuda de la lógica de negocio y controlador de la aplicación, lo que ayuda a mantener la aplicación modular, escalable y fácilmente configurable.