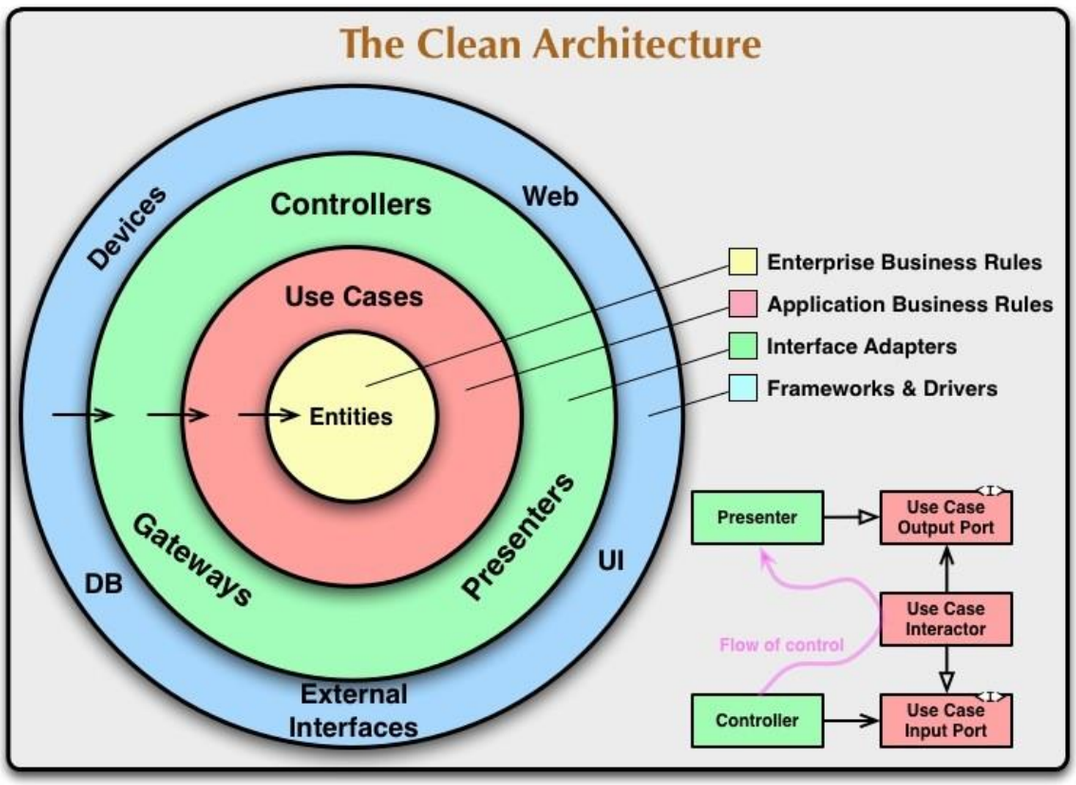
Clean Architecture en Node y Express

La arquitectura limpia o Clean Architecture, es un patrón de diseño que busca separar las preocupaciones en diferentes capas de una aplicación para lograr una estructura clara y fácilmente escalable. En este artículo, vamos a explorar cómo implementar esta arquitectura en una aplicación NodeJS, utilizando el framework Express y la ORM Sequelize.



Clean Architecture | https://blog.cleancoder.com/

Para ilustrar cómo aplicar Clean Architecture, utilizaremos un modelo de producto como ejemplo. Nuestra aplicación permitirá a los usuarios agregar, actualizar y eliminar productos, así como ver una lista de todos los productos disponibles.

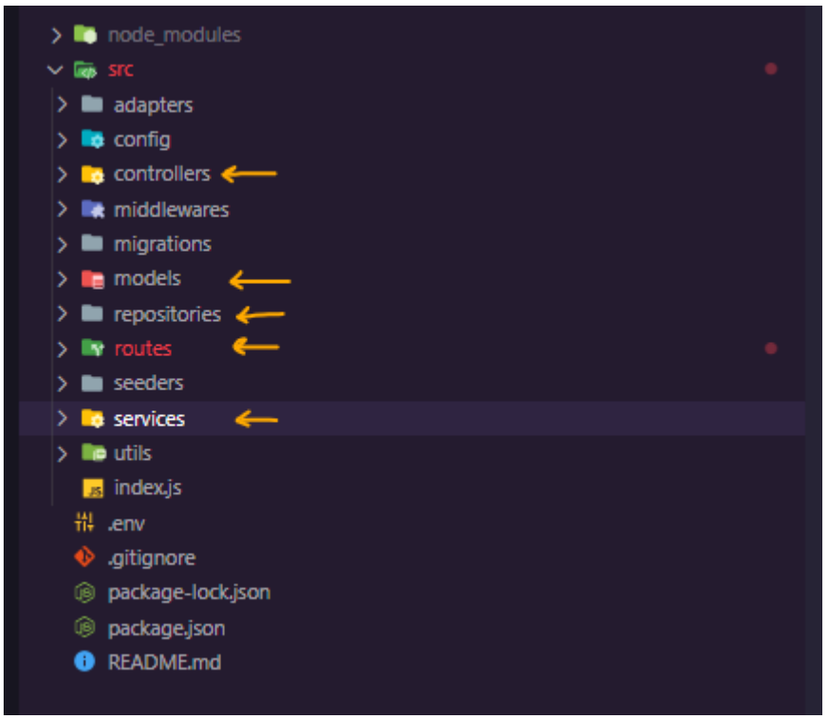
Antes de comenzar, es importante tener en cuenta que la arquitectura limpia no es una solución universal para todos los problemas de diseño de software. Cada proyecto es único y debe ser evaluado individualmente. Dicho esto, la arquitectura limpia puede ser una excelente opción para proyectos grandes y complejos que necesitan ser escalables y mantenibles a largo plazo.

Comencemos por definir las capas que utilizaremos en nuestra aplicación. Estas capas se dividirán en:

1. Routes
2. Controllers
3. Services
4. Repositories
5. Models

Cada capa tendrá una función específica y claramente definida, lo que nos permitirá escribir código modular y escalable.

Una vez creado nuestro proyecto, procedemos a crear la estructura básica de carpetas, esto debe verse mas o menos así:



Estructura básica de carpetas.

Como puedes darte cuenta no solo tenemos las carpetas señaladas, esto es porque cada proyecto puede tener más o menos carpetas según las configuraciones especificas, lo mas importante para este ejemplo son las señaladas.

Con esto claro, paso a explicar cada una de estas carpetas:

Routes

Claramente contendremos las rutas de la aplicación aquí. Las rutas son responsables de recibir las solicitudes HTTP entrantes y enviarlas al controlador correspondiente. En nuestra aplicación, utilizaremos Express para manejar las rutas, esto se ve mas o menos así:

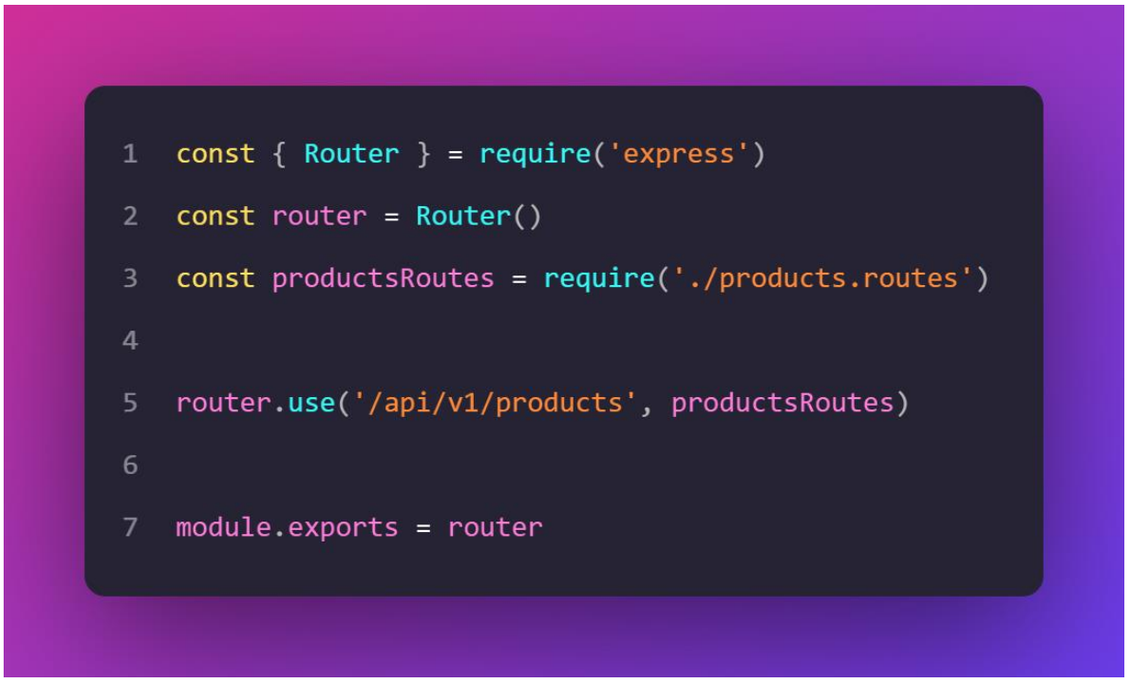
```
1  const express = require('express')
2  const productController = require('../controllers/productController')
3  const router = express.Router()
4
5  router.get('/', productController.getAllProducts)
6  router.get('/:id', productController.getProductById)
7  router.post('/', productController.createProduct)
8  router.put('/:id', productController.updateProduct)
9  router.delete('/:id', productController.deleteProduct)
10
11 module.exports = router;
```

routes/products.routes.js

Es recomendable que exista un index.js en la carpeta routes, este será el responsable de orquestar todas las rutas de tu aplicación, de está manera podrás crear un archivo de rutas para cada recurso (Modelo) de tu app.

Como podemos ver estas rutas hacen uso de una serie de funciones desde productController, este es el controlador de nuestro modelo producto, en un rato lo veremos mejor, continuemos.

Desde routes/index.js procedemos a orquestar todas las rutas de nuestra app, esto se verá mas o menos así:



```
1  const { Router } = require('express')
2  const router = Router()
3  const productsRoutes = require('./products.routes')
4
5  router.use('/api/v1/products', productsRoutes)
6
7  module.exports = router
```

routes/index.js

Como puedes observar, desde aquí podemos configurar cosas como prefijos de api, versionado y grupos de rutas (Route Groups). De está manera podemos hacer cosas como las siguientes:

```
const { Router } = require('express')
const router = Router()
const productsRoutes = require('./products.routes')
// Otras rutas
const categoriesRoutes = require('./categories.routes')
// ...

router.use('/api/v1/products', productsRoutes)
router.use('/api/v1/categories', categoriesRoutes)

module.exports = router
```

routes/index.js

Incluso versionar estas rutas, así:

```
const { Router } = require('express')
const router = Router()
const productsRoutes = require('./products.routes')
const productsRoutesV2 = require('./products.v2.routes')

router.use('/api/v1/products', productsRoutes)
// Nueva version
router.use('/api/v2/products', productsRoutesV2)

module.exports = router
```

routes/index.js

Perfecto!, con esto claro podemos pasar a explicar la siguiente capa, los controladores.

Controllers

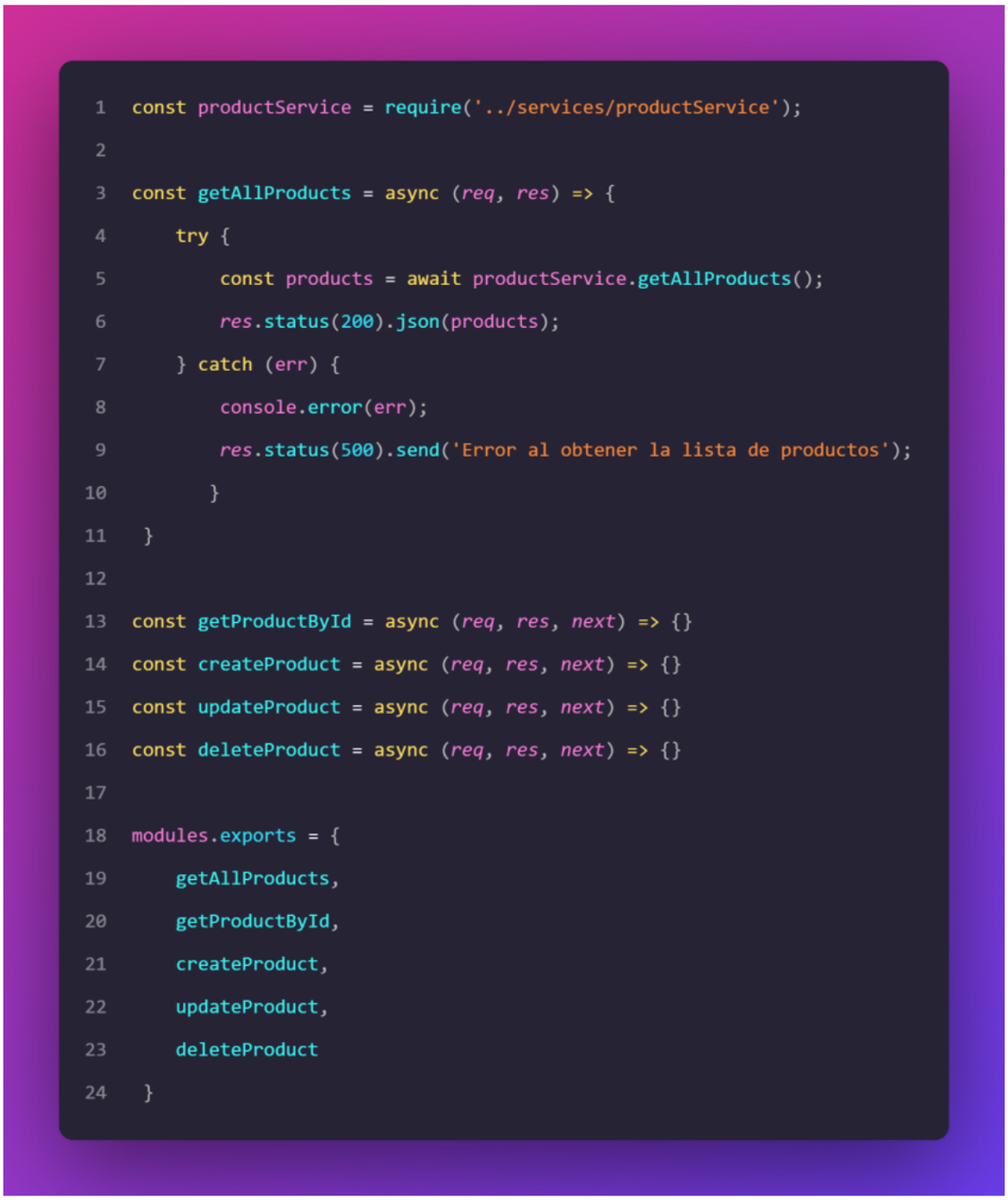
Los controladores son responsables de manejar la lógica de negocio de nuestra aplicación. Reciben solicitudes de las **rutas** y utilizan los **servicios** correspondientes para procesarlos. En nuestro ejemplo, tendremos un controlador de productos que manejará todas las solicitudes relacionadas con los productos.

Veamos como se ve el controlador de productos:

controllers/productController.js

Bastante simple!, como ya vimos los controladores son responsables de controlar el flujo datos de un lado a otro, en este caso nos permiten controlar el flujo de datos entre las rutas y los servicios que ya veremos en un momento.

Pero bueno, estos controladores hasta ahora no hacen nada, implementemos una lógica y veamos como podemos usar el servicio de producto.



```
const productService = require('../services/productService');

const getAllProducts = async (req, res) => {
    try {
        const products = await productService.getAllProducts();
        res.status(200).json(products);
    } catch (err) {
        console.error(err);
        res.status(500).send('Error al obtener la lista de productos');
       }
 }

const getProductById = async (req, res, next) => {}
const createProduct = async (req, res, next) => {}
const updateProduct = async (req, res, next) => {}
const deleteProduct = async (req, res, next) => {}

modules.exports = {
    getAllProducts,
    getProductById,
    createProduct,
    updateProduct,
    deleteProduct
 }
```

controllers/productController.js

Como ves importamos el servicio productoService, que ya veremos en un momento, la función getAllProducts solo se encarga de recibir datos de la ruta y enviarlos al servicio y luego al obtener una respuesta del servicio éste la envía a la ruta nuevamente.

Los controladores no deben tener ningún tipo de lógica de negocio ya que esta estará contenida en los servicios.

Sin embargo, en la función getAllProducts no recibimos ningún tipo parámetros que pueden venir desde *body, query ó params* desde *req,* así que no se ve el verdadero propósito de los controladores, para eso veamos otro ejemplo con el controlador de udpateProduct:

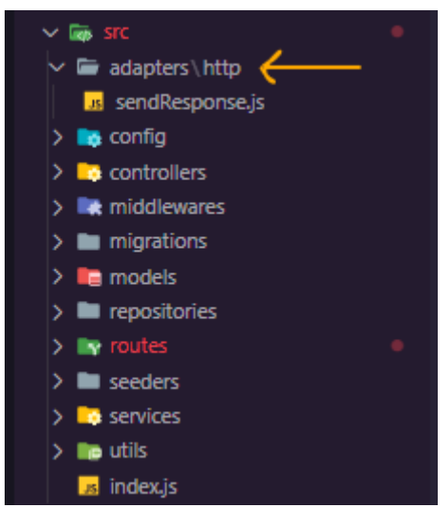
```
const updateProduct = async (req, res) => {
    try {
        const { productId } = req.params.productId
        const payload = req.body

        const product = await productService.updateProduct(productId, payload);
        sendSuccessResponse(product)
    } catch (err) {
        console.error
       }
}
```

controllers/productController.js

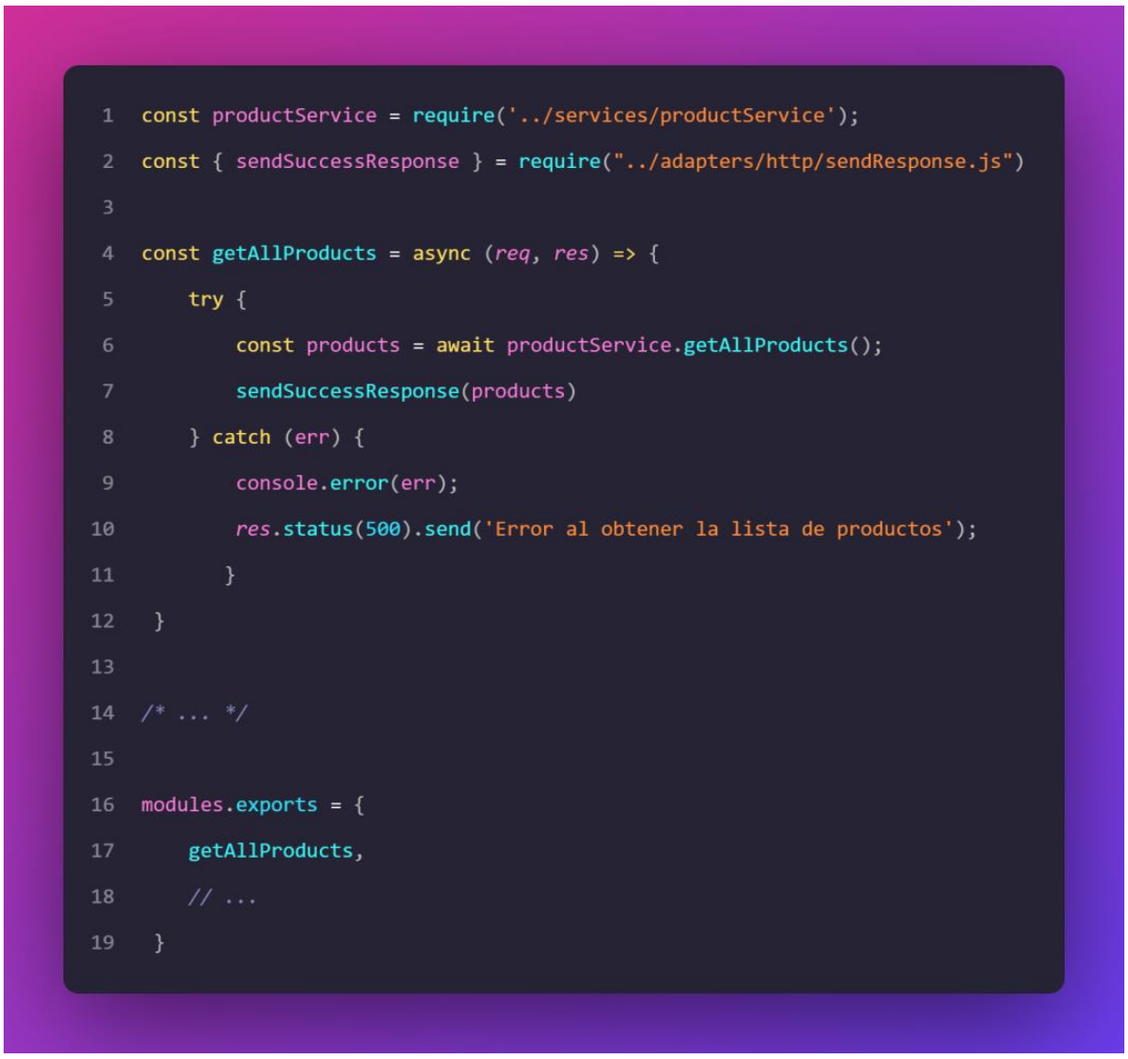
En este caso el controlador se encarga de obtener y enviar los datos requeridos por el servicio de producto para poder actualizar un producto.

Otra buena practica que podemos integrar en la capa de los controladores es usar formateadores de respuestas http, es buena idea usar estos formateadores aquí ya que es el encargado de intercambiar datos. Para esto podemos usar una carpeta llamadas *adapters* como la del ejemplo:



Carpeta adapters/http

Un ejemplo de esto puede ser formatear la respuesta que se envía a las rutas, esto se ve mas o menos así:



```
const productService = require('../services/productService');
const { sendSuccessResponse } = require("../adapters/http/sendResponse.js")

const getAllProducts = async (req, res) => {
    try {
        const products = await productService.getAllProducts();
        sendSuccessResponse(products)
    } catch (err) {
        console.error(err);
        res.status(500).send('Error al obtener la lista de productos');
       }
 }

/* ... */

modules.exports = {
    getAllProducts,
    // ...
 }
```

adapters/http/sendResponse.js

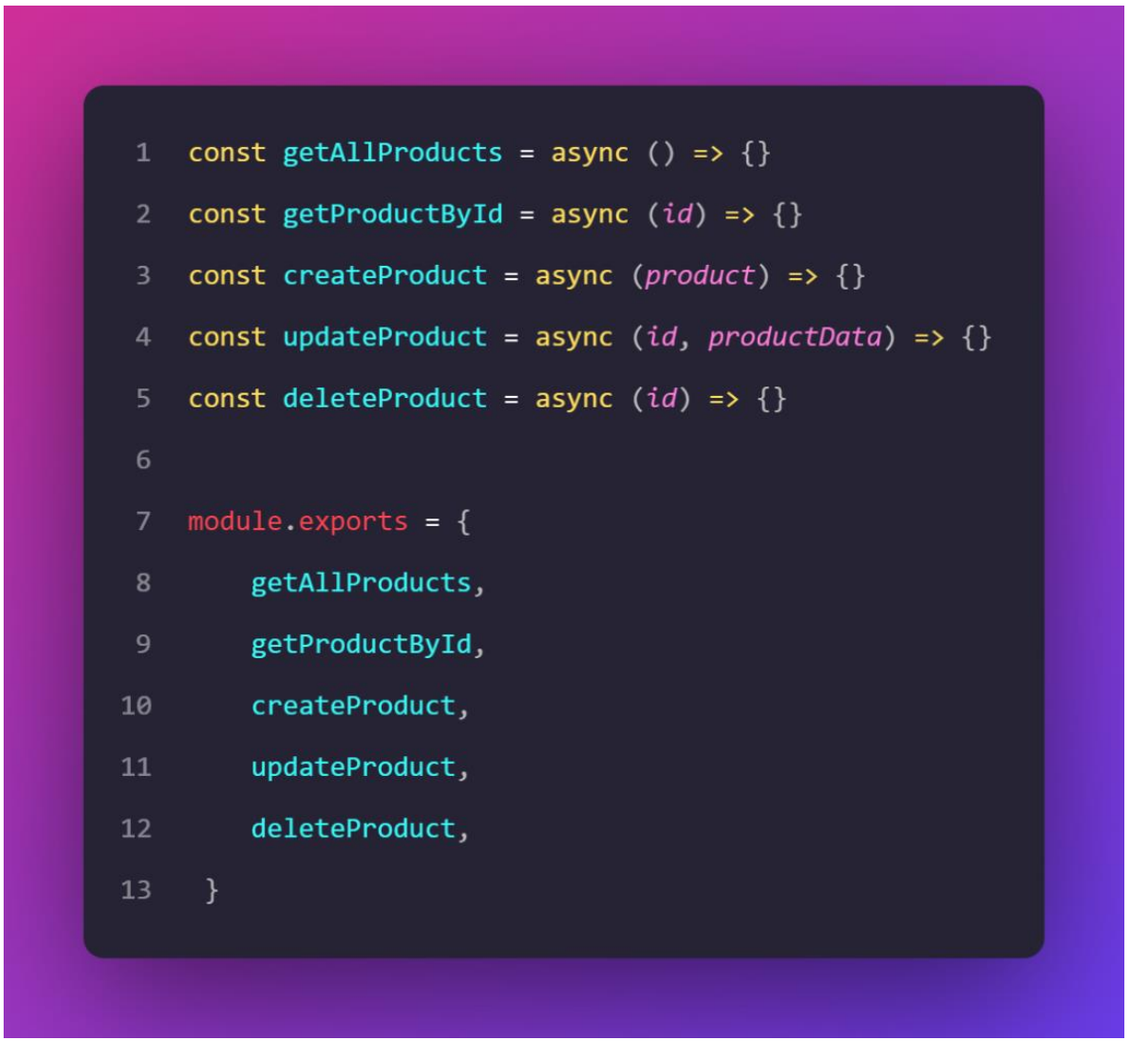
sendResponse.js expone varías funciones que se pueden usar para dar el formato deseado a las respuesta del cliente, como pueden ser success, errors, paginations, en el catch podemos usar un formateador para los errores sendErrorResponse.

Perfecto!. Hasta aquí ya sabemos como organizar nuestras rutas y controladores, como usar nuestros servicios y darle un formato personalizado a las respuestas del cliente.

Ahora pasemos a la capa mas importante de todo el proyecto, los servicios.

Services

Los servicios son responsables de implementar la lógica de negocio de nuestra aplicación. Reciben solicitudes de los controladores y utilizan los repositorios correspondientes para interactuar con la base de datos. En nuestro ejemplo, tendremos un servicio de productos que manejará todas las solicitudes relacionadas con los productos.



```
const getAllProducts = async () => {}
const getProductById = async (id) => {}
const createProduct = async (product) => {}
const updateProduct = async (id, productData) => {}
const deleteProduct = async (id) => {}

module.exports = {
    getAllProducts,
    getProductById,
    createProduct,
    updateProduct,
    deleteProduct,
 }
```

Esto es similar a los controladores, de hecho las funciones se llaman exactamente igual, esto es muy común, pero no es necesario.

Como ya vimos la definición de que es un servicio, entonces ya sabemos que aquí debemos tener toda la lógica de negocio, implementemos una lógica de negocio para entenderlo mejor, y en este caso usaremos updateProduct.

Lógica de Negocio: *En informática y ciencias de la computación, en particular en análisis y diseño orientado a objetos, el término lógica de negocio es la parte de un sistema que se encarga de codificar las reglas de negocio del mundo real que determinan cómo la información puede ser creada, almacenada y cambiada.*

```
// services/productService.js
const updateProduct = async (id, productData) => {
    // Verificar si el producto existe
    const product = await productRepository.getProductById(id)
    if (!product) {
        throw new Error('Producto no encontrado')
       }

    // Verificar si el producto está bloqueado
    if ( product.isBlocked ) {
        throw new Error('Producto no puede ser modificado')
       }

    // Finalmente actualizamos
    const updatedProduct = await productRepository.updateProduct(id, productData)
    return updatedProduct;
}
```

services/productService.js

Como se puede observar la función updateProduct contiene una serie de verificaciones antes de realizar la tarea principal, actualizar el producto.

Aquí se deben poner todas las reglas o validaciones que para el negocio/empresa son necesarias pasar para poder actualizar un producto. Otro ejemplo de esto sería validar si el usuario tiene los permisos necesarios.

Por otro lado, estos servicios constantemente están enviando y consultando datos de una base de datos, así que debemos proveer una forma de acceder a los datos sin que necesariamente tengan contacto directo a los modelos/tablas de una base de datos, inclusive sin que sepan a donde envían los datos ni de donde vienen.

Para esto usaremos el patrón repositorio (Repository Pattern).

Repositories

Los repositorios son responsables de interactuar con la base de datos. Son el punto de entrada para la persistencia de datos en nuestra aplicación. En nuestro ejemplo, tendremos un repositorio de productos que manejará todas las interacciones con la tabla de productos.

Pondré todos las funciones y su implementación así que la imagen será un poco alta, veamos como se ve:

```
const { Product } = require('../models')

const getAllProducts = async () => {
    const products = await Product.findAll()
    return products
 }

const getProductById = async (id) => {
    const product = await Product.findByPk(id)
    return product
 }

const createProduct = async (productData) => {
    const newProduct = await Product.create(productData)
    return newProduct
 }

const updateProduct = async (id, productData) => {
    const updatedProduct = await Product.update(productData, {
        where: { id },
    })
    return updatedProduct
 }

const deleteProduct = async (id) => {
    await Product.destroy({
        where: { id }
    })
 }

module.exports = {
    getAllProducts,
    getProductById,
    createProduct,
    updateProduct,
    deleteProduct,
 }
```

repositories/productRepository.js

Además de ser responsables de interactuar con la base de datos, los repositorios también tienen otras funciones importantes en la arquitectura de software. Algunas de las funciones adicionales de los repositorios incluyen:

1. **Abstraer la lógica de acceso a los datos:** Los repositorios ocultan la complejidad del acceso a los datos detrás de una interfaz sencilla y coherente. Esto permite que otros componentes de la aplicación, como los controladores y servicios, interactúen con los datos sin tener que preocuparse por los detalles de implementación.
2. **Facilitar las pruebas:** Al abstraer la lógica de acceso a los datos, los repositorios permiten que los componentes de la aplicación se prueben de manera aislada y sin depender de la base de datos real.
3. **Proporcionar una capa de caché:** En algunos casos, los repositorios pueden implementar una capa de caché para evitar la necesidad de hacer consultas a la base de datos cada vez que se accede a los datos.
4. **Facilitar la integración con diferentes fuentes de datos:** Al encapsular la lógica de acceso a los datos detrás de una interfaz coherente, los repositorios pueden facilitar la integración de diferentes fuentes de datos en una aplicación. Por ejemplo, se podría tener un repositorio que acceda a una base de datos relacional y otro que acceda a una API REST para obtener datos de una fuente externa.

Y para terminar tenemos la capa de modelos:

Models

Los modelos representan la estructura de nuestras tablas en la base de datos. En nuestro ejemplo, tendremos un modelo de producto que representará la estructura de la tabla de productos. No explicaré el código de estos modelos, la forma de desarrollarlos puede variar según el ORM usado.

Esta capa realmente no es obligatoria ya que lo únicos que acceden a ellos son los repositorios, así que si los datos no vienen de modelos si no de otras fuentes como Apis Rest, etc, entonces no es necesaria esta capa en la aplicación.

En próximas entregas trabajaremos en una forma de manejar los errores de toda la aplicación ya que aquí no se ha tenido en cuenta a profundidad.

En resumen, la arquitectura limpia es una metodología que permite diseñar aplicaciones escalables y mantenibles mediante la separación clara de las responsabilidades y una estructura modular. Aunque su implementación requiere un esfuerzo adicional al principio, a largo plazo, permite reducir la complejidad y facilitar la comprensión del código, lo que resulta en un ahorro de tiempo y esfuerzo. En este artículo, hemos explicado los componentes de la arquitectura limpia y proporcionado ejemplos de cómo implementarlos en NodeJs y Express.

Espero que este artículo haya sido útil para comprender cómo implementar una versión básica de arquitectura limpia en NodeJs y Express. Si tiene alguna pregunta o sugerencia, no dude en dejar un comentario a continuación.

¡Gracias por leer!