Adoptar un enfoque por capas

- Los frameworks como **Express,** nos permiten definir controladores de ruta como funciones de devolución de llamada, que se ejecutan cuando recibimos una solicitud de cliente. Con la cantidad de flexibilidad que brindan estos frameworks ¿Podríamos definir toda la lógica de negocio directamente dentro de esas funciones?
- Si comenzamos en este camino, el archivo de rutas de nuestro pequeño servidor puede convertirse en un código largo, desordenado, difícil de manejar, de leer, mantener y administrar, dificultando también, realizar una prueba unitaria.

Adoptar un enfoque por capas

- Por lo tanto, este sería un buen lugar para implementar el principio de programación de "separación de responsabilidades". De acuerdo con esto, deberíamos tener diferentes módulos para abordar diferentes inquietudes pertinentes a nuestra aplicación.
- Del lado del servidor, las diferentes capas (o módulos) asumen diferentes responsabilidades al procesar las solicitudes de cliente. Como ya vimos, las capas son:
  - **Controlador**: Rutas de API y endpoints.
  - **Capa de servicios**: para la lógica de negocio.
  - **Capa de acceso de datos**: para trabajar con la base de datos.

CODER HOUSE

Crear una estructura de carpetas

- Todo tiene que tener su lugar en nuestra aplicación, y una carpeta es el lugar perfecto para agrupar elementos comunes.
- Esto proporciona claridad sobre qué funcionalidad se administra y dónde, permitiéndonos organizar nuestras clases y métodos en contenedores separados que son más fáciles de administrar. A continuación se muestra una estructura de carpetas común que podemos usar como plantilla al configurar un nuevo proyecto de Node:

```
src
   ├── app.js                    app entry point
   ├── /api                   controller layer: api routes
   ├── /config                   config settings, env variables
   ├── /services              service layer: business logic
   ├── /models                   data access layer: database models
   ├── /scripts               miscellaneous NPM scripts
   ├── /subscribers           async event handlers
   └── /test                 test suites
```

CODER HOUSE

Modelos de editor/suscriptor

- Es un patrón de intercambio de datos popular en el que hay dos entidades comunicantes: editores y suscriptores.
- Los **editores** envían mensajes a través de canales específicos sin ningún conocimiento de las entidades receptoras.
- Los **suscriptores** (receptores de mensajes), por otro lado, expresan interés en uno o más de estos canales sin ningún conocimiento sobre las entidades editoriales.
- Es una buena idea incorporar un modelo de este tipo en nuestro proyecto para administrar varias operaciones secundarias correspondientes a una sola acción.

CODER HOUSE

Código limpio y fácil de leer

- También podemos consultar las **guías de estilo** de Javascript utilizados por gigantes como Google. Estas guías cubren todo, desde convenciones de nomenclatura (para archivos, variables, clases, etc.) hasta especificaciones de formato, codificaciones de archivos y mucho más.

- Al escribir código, es importante agregar **comentarios útiles** de los que otros desarrolladores de nuestro equipo puedan beneficiarse. Todo lo que se necesita es una oración de pocas palabras para ayudar a los demás en la comprensión del propósito de los fragmentos de código más complejos.

Escribir código asincrónico

- Javascript es bastante conocido por sus funciones ***callback***. También nos permiten definir el comportamiento asincrónico en Javascript. El problema con los callback es que, a medida que aumenta el número de operaciones encadenadas, el código se vuelve más difícil de manejar.

- Para resolver esto, ahora tenemos las ***Promises***, que facilitan mucho la escritura de código asincrónico. Además, tenemos ***async/await*** que la simplifica aún más, haciendo que la API sea más intuitiva y natural.

- Por lo tanto, se recomienda utilizar estas últimas en nuestras aplicaciones de Node. Ésto permite un código más limpio, mejor legibilidad, manejo de errores y pruebas más fáciles. Se mantiene un flujo de control claro y una configuración de programación funcional más coherente.

Archivos de configuración y variables de entorno

- A medida que nuestra aplicación escale, necesitaremos que ciertas opciones de configuración global sean accesibles en todos los módulos.

- Siempre es una buena práctica almacenar estas opciones juntas en un archivo separado dentro de una **carpeta de configuración** en nuestro proyecto. Esta carpeta puede contener todas sus diferentes opciones de configuración agrupadas en archivos según su uso.

```
/config
       ├── index.js
       ├── module1.js
       └── module2.js
```

- Las URLs de conexión a la base de datos se almacenan en archivos ***.env*** como variables de entorno. Así es como un archivo .env almacena datos en forma de pares clave-valor. Es un archivo secreto que no se agrega a Git.

```
DB_HOST=localhost
DB_USER=root
DB_PASS=my_password_123
```


Archivos de configuración y variables de entorno

- Una práctica de desarrollo muy común es importar todas las variables del *.env* (junto con otras opciones y configuraciones predefinidas) en los archivos de configuración y exponerlos como un objeto al resto de la aplicación.

- De esta manera, si es necesario hacer cambios, solo los realizamos en una configuración común en un lugar y eso se refleja en toda su aplicación.

```
// config/database.js

      require('dotenv').config()

      export default {
        host: process.env.DB_HOST,
        user: process.env.DB_USER,
        pass: process.env.DB_PASS,
      }
```

Testing, Logging y manejo de errores

- El ***testing*** es parte integral de cualquier aplicación de software. Nos permite probar la validez, precisión y solidez de nuestro código al sacar a la luz incluso las inexactitudes más pequeñas, no solo en el sistema en conjunto, sino también en sus componentes de forma aislada. Lo hacen de una forma automatizada.
- Las **pruebas unitarias** forman la base de la mayoría de las configuraciones de prueba. Aquí, las unidades/componentes individuales se prueban de forma aislada del resto del código para verificar su exactitud.

Testing, Logging y manejo de errores

- El ***Logging*** juega un papel importante en todo el proceso de cualquier aplicación de software.
- Desde el desarrollo hasta las pruebas y el lanzamiento para la producción, un sistema de *logging* bien implementado nos permite registrar información importante y comprender los diversos aspectos de la precisión y las métricas de rendimiento de nuestra aplicación. También facilita mucho la depuración.

CODER HOUSE

Testing, Logging y manejo de errores

- La verdad contradictoria es que los errores son buenos para los desarrolladores. Nos permiten comprender las inexactitudes y vulnerabilidades en nuestro código al alertarnos cuando el código se rompe. También brindan información relevante sobre lo que salió mal, dónde y qué se debe hacer para reparar el problema.
- Pero en lugar de permitir que Node arroje errores, interrumpa la ejecución del código e incluso falle a veces, preferimos hacernos cargo del flujo de control de nuestra aplicación manejando estas condiciones de error. Esto es lo que podemos lograr mediante el **manejo de excepciones** usando bloques try/catch.
- Al permitir a los desarrolladores administrar de manera programática tales excepciones, mantiene las cosas estables, facilita la depuración y también evita una mala experiencia para el usuario final.

CODER HOUSE