«Talento Tech»

Back-End

Node JS

Clase 10

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 10 - Modelando una API Rest

Temario:

1. API Rest
2. Estructura de archivos
3. Capas de la aplicación
4. División de responsabilidades

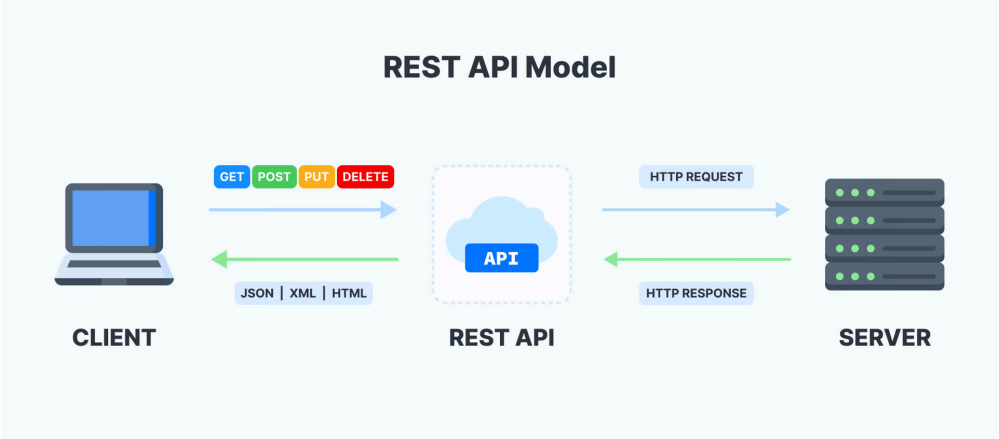
Objetivos de la Clase

En esta clase, el objetivo principal es que los estudiantes comprendan los fundamentos del modelado de una API Rest y sean capaces de aplicarlos en un proyecto práctico. Se buscará que los alumnos adquieran una visión clara sobre la estructura de archivos necesaria para organizar el código de manera eficiente, comprendiendo la importancia de definir capas dentro de la aplicación para facilitar la escalabilidad y el mantenimiento del proyecto. Además, se enfatizará la correcta división de responsabilidades dentro de cada capa, fomentando el uso de principios de diseño que aseguren un desarrollo más ordenado y modular.

API Rest

Como vimos anteriormente, una API Rest (Representational State Transfer) es un estilo de arquitectura que permite la comunicación entre sistemas a través del protocolo HTTP.

Lo más importante de una API Rest es que trabaja con *recursos*, que son básicamente las cosas o datos con los que queremos interactuar. Por ejemplo, en una tienda en línea, los recursos podrían ser los productos, las categorías o los pedidos. Cada recurso tiene una dirección única (una URL) y podemos "pedirle" al sistema que haga cosas con ellos, como obtener información, agregar algo nuevo, modificarlo o eliminarlo, usando comandos estándar como GET, POST, PUT y DELETE.



Otra característica clave de Rest es que cada interacción es *independiente*, lo que significa que cada vez que el cliente (como una app o un navegador) hace una solicitud al servidor, le envía toda la información necesaria para que esta se procese. Esto hace que el servidor no tenga que recordar lo que pasó antes, lo cual facilita que pueda manejar muchas solicitudes al mismo tiempo sin problemas.

Además, los datos en una API Rest generalmente se envían en formatos como JSON, que es fácil de leer tanto para las computadoras como para los desarrolladores. Esto ayuda a que distintos sistemas, aunque estén desarrollados con lenguajes o tecnologías diferentes, puedan entenderse perfectamente.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Estructura de archivos

En el desarrollo de software es importante tomar algunas decisiones antes de comenzar a trabajar en el código, algunas de ellas son elegir el lenguaje de programación adecuado, el tipo de arquitectura para nuestra aplicación e incluso pensar la estructura de carpetas de nuestro proyecto.

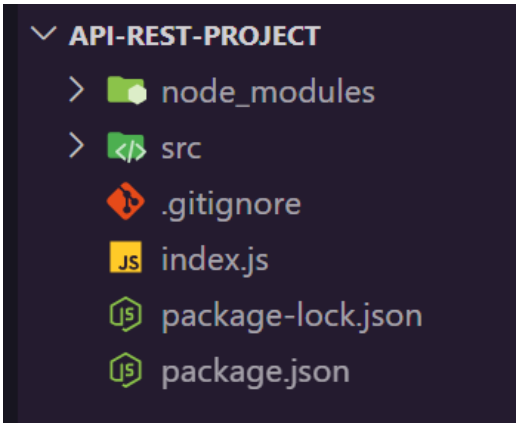
Una buena estructura también debe adaptarse al tamaño del proyecto. Si es algo pequeño, podemos empezar con algo básico, como un par de carpetas y archivos. Pero si el proyecto crece, podemos expandir la estructura para dividir aún más las responsabilidades y evitar que todo se mezcle en un solo lugar.

En esta oportunidad aprenderemos una estructura basada en capas, simple y escalable para API Rest desarrolladas en Node junto con Express.

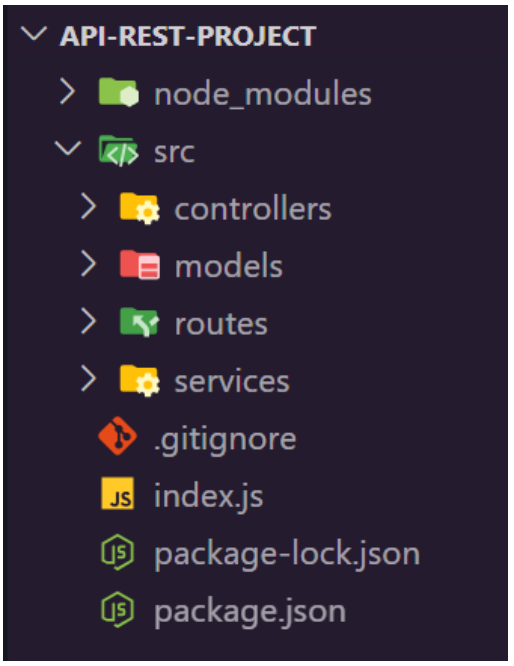
La estructura de archivos nos ayuda a que el código esté ordenado según su propósito. Por ejemplo, podríamos tener una carpeta para los archivos que manejan las solicitudes que llegan a la API, otra para los archivos que representan los datos y se conectan con la base de datos y otra para los archivos donde ocurre la "magia" de las operaciones principales. Este tipo de separación hace que sea más fácil encontrar dónde está cada cosa y trabajar en equipo sin confusiones.

Por ejemplo, una estructura típica podría verse así:

- Una carpeta llamada `src` para separar los archivos principales y de configuración del resto de la aplicación.



- Una carpeta llamada `routes`, donde se configuran las rutas o "entradas" a nuestra API.
- Otra llamada `controllers`, donde se define qué debe pasar cuando alguien usa esas rutas.
- Otra para los `models`, que conecta con la base de datos y define cómo lucen los datos.
- Y en algunos casos, puede existir una carpeta para los `services`, donde organizamos toda la lógica que no tiene que ver directamente con la base de datos o con responder solicitudes.



Es importante comprender que la estructura de archivos no es solo "dónde pongo cada archivo", sino un diseño que hace que el proyecto sea más claro, organizado y fácil de trabajar a medida que crece. Es un hábito que, si lo adoptas desde el principio, te evitará muchos dolores de cabeza más adelante.

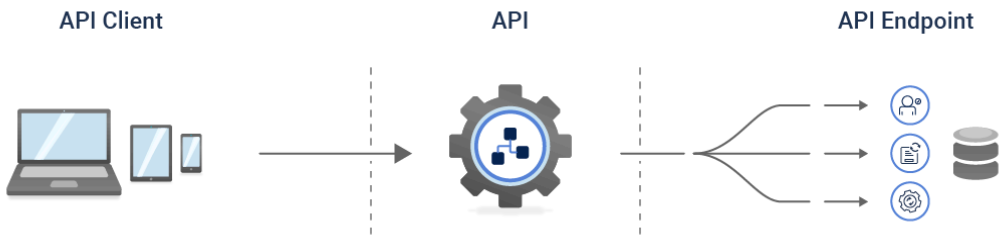
Capas de la aplicación

Cuando hablamos de las capas de una aplicación, nos referimos a cómo dividimos las responsabilidades dentro del sistema para que cada parte se encargue de algo específico. Es como construir una casa: cada piso tiene un propósito diferente, y eso hace que todo funcione de manera más ordenada y eficiente. En el caso de una API Rest, dividirla en capas nos ayuda a manejar mejor el código, hacer cambios sin romper todo y trabajar en equipo más fácilmente.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Rutas

Cuando hablamos de rutas en una API Rest, nos referimos a los "puntos de entrada" (también conocidos como **endpoints**) que los clientes (como aplicaciones o navegadores) usan para interactuar con nuestra aplicación. Cada ruta es como una dirección que lleva a un recurso específico dentro de la API y define qué se puede hacer con ese recurso. Por ejemplo, podrías tener una ruta para listar todos los usuarios, otra para crear uno nuevo y otra para eliminarlo.



Las rutas se construyen combinando una URL específica con un método HTTP. Los métodos más comunes son:

- **GET**, para obtener información (como listar usuarios).
- **POST**, para crear algo nuevo (como registrar un usuario).
- **PUT** o **PATCH**, para actualizar información (como cambiar el correo de un usuario).
- **DELETE**, para eliminar recursos (como borrar un usuario).

En la práctica, las rutas suelen organizarse en un archivo específico dentro de una carpeta llamada `routes` o similar. Por ejemplo, podrías tener un archivo `users.routes.js` que agrupe todas las rutas relacionadas con usuarios. Esto no solo mantiene el código más organizado, sino que también hace que sea más fácil encontrar y modificar rutas en el futuro.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Controladores

Su trabajo principal es recibir las solicitudes que llegan desde las rutas, procesarlas y decidir qué hacer con ellas. En otras palabras, son los responsables de manejar la lógica que conecta las rutas con el resto de la aplicación, actuando como un puente entre las solicitudes del cliente y los datos o servicios que necesitamos usar.

Imagina que la API es un restaurante: las rutas son los meseros que toman los pedidos, y los controladores son los chefs que preparan lo que se pide. Cada controlador se enfoca en una tarea específica, lo que ayuda a mantener el código organizado y fácil de entender.

Por ejemplo, si tienes una API que maneja usuarios, podrías tener un controlador llamado `users.controller.js` con funciones como estas:

- `getAllUsers(req, res)`: Maneja la solicitud para obtener una lista de usuarios y devuelve una respuesta con esa información.
- `getUserById(req, res)`: Busca un usuario específico basado en un parámetro como el `id` que llega en la solicitud.
- `createUser(req, res)`: Procesa los datos enviados en la solicitud para crear un nuevo usuario.
- `updateUser(req, res)`: Recibe datos actualizados y los aplica a un usuario existente.
- `deleteUser(req, res)`: Se encarga de eliminar un usuario específico.

Cada función del controlador se encarga de una acción concreta. El controlador no debe preocuparse por la lógica compleja ni por interactuar directamente con la base de datos; su trabajo es coordinar. Por ejemplo, si necesitas obtener datos, el controlador llama a un servicio o modelo y luego devuelve la respuesta al cliente.

Un detalle importante es que los controladores también se encargan de manejar los errores que puedan surgir. Por ejemplo, si alguien intenta buscar un usuario que no existe, el controlador debe devolver un mensaje claro, como "Usuario no encontrado" y un código de estado adecuado.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

El objetivo de este tema es que entiendas que los controladores son como el "centro de comando" de las solicitudes. Su responsabilidad es procesar, delegar y responder, manteniéndose simples y enfocados en su tarea específica. Así, logramos un código más organizado y fácil de mantener, donde cada parte de la aplicación tiene su rol bien definido.

Modelos

En una API, los modelos son como los planos de una máquina: describen cómo deben verse los datos y cómo deben comportarse al interactuar con la base de datos. Actúan como un “esquema” o “molde” que define la estructura de una entidad, permitiéndonos trabajar de forma ordenada y predecible con la información.

Por ejemplo, si estamos desarrollando una tienda en línea, podríamos tener un modelo llamado `Product` que represente cómo se ve un objeto producto. Este modelo incluiría propiedades como el nombre del producto, su precio, descripción, y categorías, además de especificar el tipo de datos que se espera para cada propiedad.

Los modelos no solo definen la estructura de los datos, sino que también nos facilitan el acceso y manipulación de información almacenada en bases de datos, ya sean relacionales (SQL) o no relacionales (NoSQL).

Ejemplo de modelo en MongoDB

Si usamos MongoDB con la librería Mongoose, un modelo de usuario podría verse así:

```
const userSchema = new mongoose.Schema({
 name: { type: String, required: true },
 email: { type: String, required: true, unique: true },
 password: { type: String, required: true },
 createdAt: { type: Date, default: Date.now }
});
```

Este esquema define cómo lucirá un objeto usuario cada vez que accedamos a los datos de uno o más usuarios en la base de datos. Aunque el contenido exacto y su funcionamiento no es relevante para para este caso ya que no trabajaremos con MongoDB, este ejemplo sirve como una representación visual del concepto de modelo.

¿Qué pasa si no usamos una base de datos como Mongo?

No siempre es necesario definir esquemas o modelos estrictos. Por ejemplo, si trabajamos con datos almacenados en archivos JSON locales, podemos acceder a ellos usando la librería `filesystem` (fs), que no requiere un esquema predefinido. En este caso, aunque no tengamos un "modelo" en el sentido clásico, la capa de modelos incluiría las funciones, métodos y configuraciones necesarias para leer, escribir y procesar esos datos de forma eficiente.

En resumen, los modelos son fundamentales para estructurar y gestionar la información en aplicaciones basadas en APIs. Su implementación depende del tipo de almacenamiento de datos que utilicemos, pero su propósito siempre será garantizar que los datos sean consistentes, accesibles y fáciles de trabajar.

Servicios

Los servicios son una parte clave de la arquitectura de una API Rest y actúan como los “motores” que realizan las operaciones necesarias para cumplir con las solicitudes que llegan a la API. Su función principal es manejar la lógica de negocio de la aplicación, es decir, todas las reglas y procesos que determinan cómo los datos deben ser tratados.

¿Cómo funcionan los servicios?

Los servicios se encuentran entre los controladores y los modelos:

- **Desde el controlador**, reciben las solicitudes para ejecutar tareas específicas.
- **Hacia los modelos**, envían solicitudes para obtener o modificar datos en la base de datos, o en otros casos, procesan datos externos.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Al estructurar la lógica de negocio en los servicios, evitamos que los controladores o los modelos se vuelvan demasiado complejos o cargados de responsabilidades. Esto facilita el mantenimiento y la escalabilidad del código.

Ejemplo de un servicio en una API Rest

Si bien profundizaremos el concepto de servicios más adelante, veamos un ejemplo práctico centrándonos en el concepto y su propósito en lugar del código:

Supongamos que estamos trabajando en una API para gestionar productos, y queremos implementar una lógica para obtener todos los productos que están en stock:

```
// product.service.js

const productModel = require('./product.model');

async function getProductsInStock() {
  const products = await productModel.getAllProducts();
  return products.filter(product => product.stock > 0);
}

async function addNewProduct(productData) {
  if (!productData.name || !productData.price) {
    throw new Error('El nombre y el precio del producto son obligatorios.');
  }
  if (productData.price <= 0) {
    throw new Error('El precio debe ser mayor que cero.');

  return await productModel.createProduct(productData);
}

module.exports = {
  getProductsInStock,
  addNewProduct,
};
```

Explicación del ejemplo

1. `getProductsInStock`: El servicio solicita todos los productos al modelo y luego filtra los que tienen stock disponible. La lógica de filtrado se queda aquí, separada del controlador y del modelo.

   Lo curioso es que podríamos contar con otro método dentro del servicio llamado `getProductsByCategory` que también solicite todos los productos al modelo pero a diferencia del anterior, se encargue de devolver al controlador solo los productos según determinada categoría.

2. `addNewProduct`: Antes de crear un nuevo producto, este servicio valida que los datos cumplan con ciertas reglas de negocio, como asegurarse de que el precio sea mayor que cero y que los campos esenciales estén presentes. Si las reglas no se cumplen, lanza un error.

La capa de servicios es esencial para manejar la lógica de negocio de manera organizada y eficiente. Su implementación no solo mejora la claridad y el mantenimiento del código, sino que también prepara a tu API para ser más flexible y escalable, permitiendo que puedas adaptarla fácilmente a nuevas necesidades o requisitos futuros.

División de responsabilidades

Como hemos visto a lo largo de esta clase, al modelar una API Rest es crucial planificar con anticipación no solo las capas que vamos a utilizar, sino también la estructura de archivos y directorios de nuestra aplicación. Esta planificación tiene como objetivo crear aplicaciones que sean fáciles de mantener y que puedan escalar o evolucionar de manera eficiente en el futuro.

Una de las ideas clave que debemos incorporar en nuestro proceso de desarrollo es la **división de responsabilidades**. Tal como hemos observado, cada capa de nuestra API Rest tiene una función específica. Esta división no solo responde a la necesidad evidente de evitar que todo el proyecto se concentre en un solo archivo, sino también a la intención consciente de organizar nuestro código de manera ordenada, asegurando que cada parte

de la aplicación tenga una responsabilidad claramente definida. De este modo, evitamos tener un código disperso y sin propósito, y en su lugar logramos una arquitectura más coherente y fácil de gestionar.

Es importante también resaltar que la división de responsabilidades y la partición del código serían mucho más complejas de implementar sin el uso de **módulos**. Como aprendimos en clases anteriores, los módulos permiten que podamos exportar e importar partes del código de forma ordenada, asegurando que solo las partes necesarias del sistema estén disponibles en los lugares correctos. Esto nos proporciona un control mucho más granular sobre cómo y dónde se utiliza cada fragmento de código, lo que facilita su mantenimiento y evolución a lo largo del tiempo.

Ejercicio Práctico

Estructura y Rutas de la API

Sabrina y Matías te observan con una mirada satisfecha, sabiendo que has hecho grandes avances. "Ahora que ya tienes una buena base trabajando con APIs, es momento de ponerte en los zapatos de un desarrollador que construye una aplicación desde cero", comenta Matías.

"Construir una API no solo se trata de hacer que los datos lleguen a tu aplicación, sino de cómo organizas todo el proyecto de manera que sea fácil de mantener y escalar", explica Sabrina mientras te señala la pantalla.

“Queremos que crees la estructura de directorios para tu API usando arquitectura REST", dice Matías con tono serio, "Esto significa que debes dividir tu código de manera ordenada en capas, tal como lo vimos en la clase. Queremos ver que uses una estructura clara para manejar rutas, controladores, modelos y servicios. Así, cuando tu proyecto crezca, será más fácil mantenerlo organizado y profesional."

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Misión:

1. Crea la estructura de directorios para tu aplicación, asegurándote de crear carpetas para **rutas**, **controladores**, **modelos** y **servicios**.
2. En tu archivo principal, crea dos rutas nuevas:
    - Una ruta que devuelva una respuesta en formato **HTML**. Podría ser algo simple, como una página de bienvenida.
    - Otra ruta que devuelva una respuesta en formato **JSON**, por ejemplo, con una lista de usuarios o productos ficticios.
    - Todavía no es necesario colocar las rutas dentro de la carpeta routes, por lo que deberás definirlas en el archivo index.js.

Materiales y Recursos Adicionales:

Guía de Arquitectura REST: Un artículo que detalla los principios básicos de la arquitectura REST, que es la base de la organización de nuestra API.

Tutorial sobre la creación de APIs RESTful con Express y Node.js: Este tutorial paso a paso te llevará desde la creación de un servidor básico hasta la implementación de rutas, controladores, y modelos.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Preguntas para Reflexionar:

- ¿Por qué es importante dividir el código en diferentes capas, como rutas, controladores, modelos y servicios, cuando se está desarrollando una API Rest?
- ¿Cómo impacta la organización adecuada de la estructura de directorios en la escalabilidad y el mantenimiento de una aplicación en el largo plazo?
- ¿Cómo podrías aplicar la arquitectura REST a un proyecto más grande y complejo, como una aplicación con múltiples recursos interrelacionados?
- En tu experiencia, ¿cómo mejora la legibilidad del código cuando se siguen buenas prácticas de organización y se usan herramientas como módulos para separar responsabilidades?

Próximos Pasos:

- **Request y Response**: llegó el momento de aplicar de forma práctica todo nuestro conocimiento sobre la comunicación web.
- **Capa lógica**: controlando la respuesta de nuestra aplicación.
- **Modelo de datos y trabajo con JSON**: consultamos datos de forma interna y los devolvemos al cliente desde nuestra API Rest.

Buenos Aires Ciudad