«Talento Tech»

Back-End

Node JS

Clase 12

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 12 - Capa lógica

Temario:

1. Express Router
2. Controladores
3. Servicios
4. Arquitectura basada en capas

Objetivos de la Clase

En esta clase, los estudiantes aprenderán a estructurar aplicaciones web robustas con Node.js y Express, enfocándose en la organización del código para un mantenimiento y escalabilidad eficientes. Se introducirá el concepto de Express Router para modularizar las rutas de la aplicación, permitiendo una gestión más clara y ordenada de las diferentes secciones del sitio web. Además, se explorará la separación de la lógica de la aplicación en controladores y servicios. Los controladores se encargarán de recibir las solicitudes del cliente, interactuar con los modelos y coordinar la respuesta, mientras que los servicios encapsularán la lógica de negocio reutilizable, promoviendo un código limpio y modular. Finalmente, los alumnos comprenderán cómo esta arquitectura facilita el manejo de la lógica de la aplicación, mejorando la legibilidad y la mantenibilidad.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Express Router

Express Router es un middleware de Express.js que permite crear manejadores de rutas modulares y montables. En esencia, permite agrupar rutas en diferentes archivos o módulos facilitando la escalabilidad al tener la capa de ruteo organizada mediante determinada lógica de negocio y limpiando el código del archivo principal o `entry point` de nuestro proyecto.

¿Por qué usar Express Router?

- **Organización:** Facilita la organización del código al separar las rutas en módulos lógicos. Esto es especialmente útil en aplicaciones grandes con muchas rutas.
- **Reutilización:** Permite reutilizar conjuntos de rutas en diferentes partes de la aplicación o incluso en otras aplicaciones.
- **Mantenibilidad:** Mejora la mantenibilidad al aislar la lógica de cada conjunto de rutas. Los cambios en un conjunto de rutas tienen menos probabilidades de afectar a otras partes de la aplicación.
- **Legibilidad:** Aumenta la legibilidad del código al dividirlo en partes más pequeñas y manejables.
- **Rutas versionadas:** Facilita la creación de APIs versionadas (v1, v2, etc.) al definir rutas separadas para cada versión.

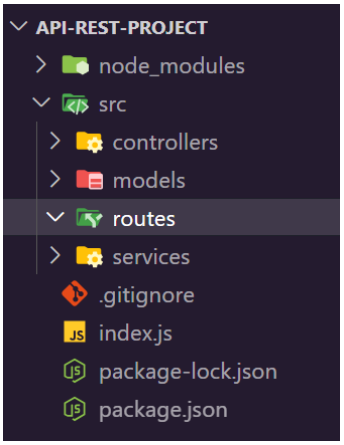
Buenos Aires aprende
Agencia de Habilidades para el Futuro

¿Cómo funciona Express Router?

Así se ve el archivo `index.js` de nuestro proyecto:

```
import express from 'express';
import cors from 'cors';

const app = express();

// Configuración básica: Permitir todos los orígenes
app.use(cors());


// Aquí irían las rutas


// Middleware para manejar errores 404
app.use((req, res, next) => {
    res.status(404).send('Recurso no encontrado');
});

const PORT = 3000;

app.listen(PORT, () => console.log(`http://localhost:${PORT}`));
```

Como podemos observar aún no tenemos rutas definidas, ya que las colocaremos en una capa diferente.

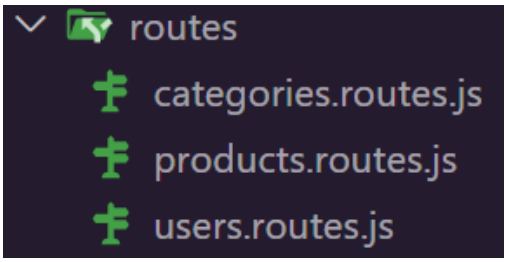


¿Te acordás de la capa **routes** que vimos cuando modelamos la API?

En esta carpeta colocaremos todos los archivos con rutas de nuestro proyecto que luego serán llamados desde `index.js`

permitiendo encapsular la lógica de ruteo y separarla de la configuración inicial del servidor.

1. **Crear el archivo para un grupo de rutas:** normalmente se agrupan rutas por modelo de negocio, tendremos un archivo para las rutas que apuntan a lógica de **productos,** otras para **usuarios** o cualquier otra distinción que creamos necesaria.



El nombre de cada archivo de rutas es a criterio del desarrollador, aunque algunos ejemplos de estándares son:

- `products.routes.js`
- `productsRoutes.js`

2. **Crear una instancia de Router:** Se crea una nueva instancia de `express.Router()` en el archivo de rutas que deseamos:

```
import express from 'express';

const router = express.Router();
```

3. **Definir rutas en el Router:** Se definen las rutas utilizando los mismos métodos que se usan en la aplicación principal (`get`, `post`, `put`, `delete`, etc.):

```
import express from 'express';

const router = express.Router();

router.get('/products', (req, res) => {
```

```
    res.send('Listado de productos');
});


router.get('/products/:id', (req, res) => {
    res.send(`Producto con id ${req.params.id}`);
});

router.post('/products', (req, res) => {
    res.send('Producto creado');
});

export default router;
```

En lugar de utilizar la instancia de `app.get()` en el archivo `index.js` como vimos en nuestros primeros servidores de ejemplo, aquí utilizamos `router` y finalmente exportamos esa instancia de `router` para invocarla desde otro módulo (archivo).

4. **Invocar el Router de productos en `index.js`:** Se utiliza `app.use()` para integrar el Router en la aplicación principal, especificando un prefijo de ruta opcional:

```
import express from 'express';
import cors from 'cors';

import productsRouter from './src/routes/products.routes.js';

const app = express();

// Configuración básica: Permitir todos los orígenes
app.use(cors());

// Aquí irían las rutas
```

```
app.use('/api', productsRouter);

// Middleware para manejar errores 404
app.use((req, res, next) => {
    res.status(404).send('Recurso no encontrado');
});

const PORT = 3000;

app.listen(PORT, () => console.log(`http://localhost:${PORT}`));
```

En caso de contar con más de un módulo o archivo de rutas, el procedimiento es exáctamente igual e iremos colocando cada router uno debajo del otro:

```
// Aquí irían las rutas
app.use('/api', productsRouter);
app.use('/api', usersRouter);
```

El prefijo `/api` es una cuestión estética común en las API Rest para que las request desde los clientes a nuestro servidor sean a las rutas: `/api/products` o `/api/users`, etc. Si no quisiéramos contar con este prefijo, podríamos utilizar cualquier otro o simplemente dejar `app.use('/, productsRouter);` sin ningún prefijo.

En ocasiones el prefijo se utilizar para evitar que el servidor tenga más trabajado encontrando las rutas requeridas, por ejemplo:

```
app.use('/', productsRouter);
app.use('/', categoriesRouter);
app.use('/api', usersRouter);
```

Si el cliente solicita acceder a la ruta `/categories` dentro del router de categorias, el primer archivo en ser leído por el servidor sería el `productsRouter` ya que la "ruta base" para ambos **routers** es `/`, al no encontrar la ruta solicitada, volvería a `index.js` e ingresaría, ahora sí, en `categoriesRouter`. En cambio, si la ruta solicitada es `/api/users` al tener el prefijo `/api`, inicialmente saltea la lectura de los 2 primeros **routers** definidos, accediendo directamente al que posee la ruta solicitada.

Si bien en aplicaciones pequeñas o con pocas rutas, el impacto de esto es mínimo, en grandes aplicaciones puede ser una excelente estrategia para optimizar el rendimiento de nuestro servidor.

De esta manera, configuramos la capa de rutas de nuestra aplicación, contando con una carpeta dedicada llamada **routes** donde separamos las distintas rutas a las que los clientes nos solicitarán recursos, de manera organizada y escalable.

Una vez finalizado, veamos como nos quedarían los archivos de nuestra aplicación si tuvieramos el `index.js` con la configuración inicial del servidor y un archivo `products.routes.js` para las rutas que brindarán acceso a la información de los productos:

products.routes.js

```
import express from 'express';
const router = express.Router();

router.get('/products', (req, res) => {
    res.send('Listado de productos');
});

router.get('/products/:id', (req, res) => {
    res.send(`Producto con id ${req.params.id}`);
});
```

```
router.post('/products', (req, res) => {
    res.send('Producto creado');
});

export default router;
```

index.js

```
import express from 'express';
import cors from 'cors';
import productsRouter from './src/routes/products.routes.js';

const app = express();

// Configuración básica: Permitir todos los orígenes
app.use(cors());

// Routers
app.use('/api', productsRouter);

// Middleware para manejar errores 404
app.use((req, res, next) => {
    res.status(404).send('Recurso no encontrado');
});

const PORT = 3000;

app.listen(PORT, () => console.log(`http://localhost:${PORT}`));
```

Ahora si el cliente realiza una **request** a nuestro servidor solicitando la ruta `/api/products/17`, obtendrá como resultado el mensaje "`Producto con id 17`" tal como lo hemos configurado en nuestro **router** de productos.

En resumen, Express Router es una herramienta esencial para construir aplicaciones Express.js bien estructuradas y mantenibles. Permite modularizar las rutas, lo que facilita la organización, la reutilización y el mantenimiento del código.

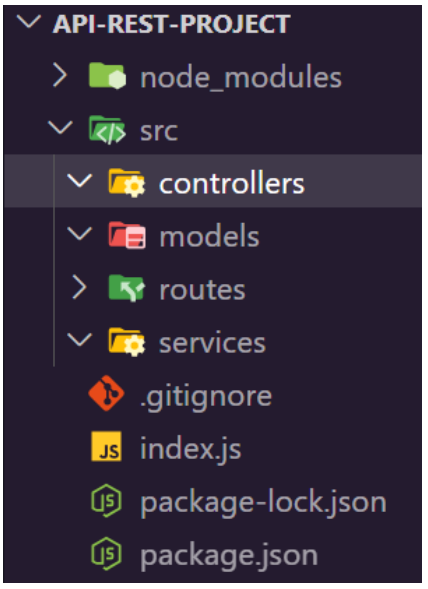
Buenos Aires aprende
Agencia de Habilidades para el Futuro

Controladores.

En una API REST, los controladores son la capa que se encargan de gestionar las solicitudes HTTP que llegan a los diferentes endpoints a través de las rutas, interactuando con la lógica de negocio (generalmente encapsulada en servicios o modelos) y generando las respuestas correspondientes.

¿Cuál es la función de un Controlador?

- **Recibir solicitudes:** El controlador recibe la solicitud HTTP del cliente desde una ruta, incluyendo la URL, los parámetros, el cuerpo de la solicitud (en caso de ser un POST, PUT, etc.), las cabeceras, etc.
- **Procesar la solicitud:** El controlador procesa la solicitud, extrayendo la información relevante y realizando las validaciones necesarias.
- **Interactuar con el modelo/servicio:** El controlador interactúa con los modelos (que representan los datos) o con los servicios (que encapsulan la lógica de negocio) para realizar las operaciones necesarias, como consultar una base de datos, realizar cálculos, etc.

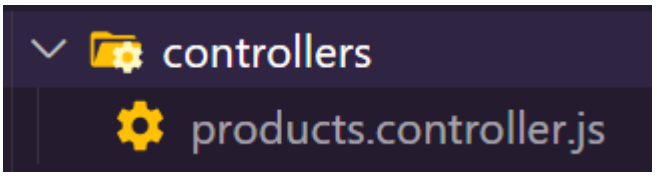


- **Generar una respuesta:** El controlador genera una respuesta HTTP para enviar al cliente, incluyendo el código de estado (200 OK, 404 Not Found, 500 Internal Server Error, etc.), las cabeceras y el cuerpo de la respuesta (que puede ser un JSON, HTML, texto plano, etc.).

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Creando un controlador

Para crear nuestro primer controlador debemos comenzar por crear un nuevo archivo dentro de la carpeta de `controllers`:



En este caso, comenzamos con el controlador para los **productos** de nuestra aplicación.

Una vez creado el archivo, debemos entender que tipos de datos estaremos devolviendo a través de nuestro controlador, por ejemplo:

`product.controller.js`: Contiene las funciones controladoras que manejan las solicitudes para el recurso `/products`. Cada función corresponde a una operación CRUD:

- `getAllProducts`: Obtiene todos los productos.
- `getProductById`: Obtiene un producto por su ID.
- `createProduct`: Crea un nuevo producto.
- `updateProduct`: Actualiza un producto existente.
- `deleteProduct`: Elimina un producto.

Asimismo, en el archivo **`product.routes.js`** definimos las rutas para el recurso `/products` y las asociamos con las funciones controladoras correspondientes.

Veamos como sería:

```
// products.controller.js
const products = [
  {
      id: 1,
      name: 'Producto 1',
      price: 1000
  },
  {
      id: 2,
      name: 'Producto 2',
      price: 2000
  }
]

export const getAllProducts = async (req, res) => {
    res.status(200).json(products);
};

export const getProductById = async (req, res) => {
    const id = req.params.id;
    const product = products.find(product => product.id == id);
    if (product) {
        res.status(200).json(product);
    } else {
        res.status(404).json({ message: 'Producto no encontrado' });
    }
};

export const createProduct = async (req, res) => {
    const { name, price } = req.body;
    const newProduct = {
        id: products.length + 1,
        name,
        price
    };
    products.push(newProduct);
    res.status(201).json(newProduct);
};
```

```
// products.routes.js
import express from 'express';
const router = express.Router();

import {
  getAllProducts,
  getProductById,
  createProduct
} from '../controllers/products.controller';

router.get('/products', getAllProducts);

router.get('/products/:id', getProductById);

router.post('/products', createProduct);

export default router;
```

En el primer archivo, dedicado a los controladores, se definen funciones que implementan la lógica de negocio para la gestión de productos. Específicamente, `getAllProducts` se encarga de recuperar y devolver la colección completa de productos. Por su parte, `getProductById` extrae el parámetro de ruta `:id` de la solicitud y lo utiliza para buscar el producto correspondiente. Finalmente, `createProduct` procesa los datos enviados en el cuerpo de la solicitud (`req.body`) para crear un nuevo producto y agregarlo a la colección.

Un aspecto fundamental de este enfoque radica en la delegación de la lógica de respuesta a los controladores. A diferencia del manejo de rutas anterior, donde la propia ruta se encargaba de responder directamente al cliente, ahora las rutas invocan a las funciones controladoras. Este cambio se evidencia al comparar las siguientes implementaciones:

Implementación anterior (ruta manejando la respuesta):

```
router.get('/products', (req, res) => {
    res.send('Listado de productos');
});
```

Implementación actual (ruta invocando un controlador):

```
router.get('/products', getAllProducts);
```

Esta estrategia de separación de responsabilidades resulta crucial para la arquitectura de la aplicación. Al encapsular cada aspecto en su capa correspondiente, se mejora significativamente la legibilidad del código y se facilita la escalabilidad del proyecto, permitiendo la adición de nuevas funcionalidades y capas con mayor facilidad y menor riesgo de generar conflictos o dependencias innecesarias.

Servicios

Los servicios encapsulan la lógica de negocio de la aplicación. Actúan como una capa intermedia entre los controladores y los modelos (o cualquier otra fuente de datos externa). Esta separación de responsabilidades proporciona una mayor modularidad, reutilización y testabilidad del código.

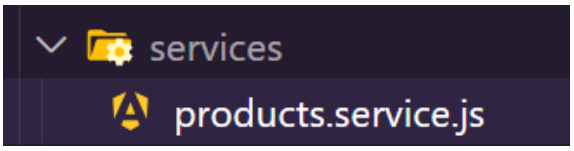
Buenos Aires aprende
Agencia de Habilidades para el Futuro

¿Cuál es la función de un Servicio?

- **Encapsular la lógica de negocio:** Contienen las reglas y procesos específicos de la aplicación. Por ejemplo, la lógica para calcular el precio final de un producto con descuentos, la lógica para procesar un pago, la lógica para enviar un correo electrónico de confirmación, etc.
- **Abstraer la lógica de acceso a datos:** A menudo, los servicios interactúan con los modelos para acceder a la base de datos. Sin embargo, los controladores no necesitan conocer los detalles de cómo se acceden a los datos. Los servicios proporcionan una interfaz limpia y abstracta para realizar estas operaciones.
- **Reutilización de lógica:** La lógica encapsulada en los servicios se puede reutilizar en diferentes partes de la aplicación o incluso en otras aplicaciones.

Creando un Servicio

El proceso de creación de un servicio sigue una estructura similar a la de los controladores. En este caso, se utiliza el directorio `services` del proyecto, donde se crea el archivo `products.service.js`. La lógica de manipulación de datos, previamente ubicada en el controlador, la migramos a este nuevo archivo.



Tras la migración, el servicio se define de la siguiente manera:

```
// products.service.js
const products = [
  {
      id: 1,
      name: 'Producto 1',
      price: 1000
  },
```

```
    {
        id: 2,
        name: 'Producto 2',
        price: 2000
    },
]

export const getAllProducts = () => {
    return products;
};

export const getProductById = async (id) => {
  return products.find(product => product.id == id);
};

export const createProduct = async (productData) => {
    const newProduct = {
      id: products.length + 1,
      name: productData.name,
      price: productData.price
    };
    products.push(newProduct);

    return newProduct;
};

// ... (funciones para updateProduct y deleteProduct de manera similar)
```

Como se observa, la lógica para buscar, filtrar y agregar productos al array se ha trasladado a métodos específicos dentro del servicio. Esta separación fortalece la arquitectura de la aplicación al distribuir las responsabilidades de manera más clara.

Veamos cómo queda ahora nuestro controlador:

```
// products.controller.js
import productsService from '../services/products.service.js';

export const getAllProducts = async (req, res) => {
    res.status(200).json(productsService.getAllProducts());
};

export const getProductById = async (req, res) => {
    const id = req.params.id;
    const product = productsService.getProductById(id);
    if (product) {
        res.status(200).json(product);
    } else {
        res.status(404).json({ message: 'Producto no encontrado' });
    }
};

export const createProduct = async (req, res) => {
    const { name, price } = req.body;
    const newProduct = productsService.createProduct({ name, price });
    res.status(201).json(newProduct);
};
```

Podemos observar que los métodos del controlador ahora se limitan a recibir la información de la solicitud (a través de parámetros de ruta, parámetros de consulta o el cuerpo de la solicitud) y a invocar el servicio correspondiente, que ejecuta la lógica necesaria. Una vez que el servicio completa su tarea, el controlador recibe la respuesta y la utiliza para responder al cliente mediante `res.json()`. Este flujo de trabajo define una clara separación de responsabilidades, mejorando la organización y mantenibilidad del código.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Arquitectura basada en capas

A lo largo de esta clase, hemos explorado diversas capas que componen una aplicación bien estructurada. Aprendimos a modularizar las rutas mediante Express Router, comprendimos el rol de los controladores en la recepción y procesamiento de solicitudes, así como en la generación de respuestas. Finalmente, introdujimos la capa de servicios y su función crucial en la encapsulación de la lógica de negocio, separándola de las demás capas y asignando a cada una una única responsabilidad.

Un punto importante a destacar, que inicialmente podría generar cierta inquietud, es que ninguna de estas capas o divisiones de archivos es estrictamente necesaria para el funcionamiento básico de una aplicación. Es posible concentrar todo el código en el archivo de entrada principal, `index.js`, sin aplicar ninguna encapsulación ni separación de responsabilidades.

Entonces, ¿por qué separar en capas? Como se mencionó anteriormente, en el ámbito de la programación existen estándares, arquitecturas y prácticas que buscan optimizar los procesos, haciéndolos más claros, simples y escalables. La separación en capas de una API REST se inscribe dentro de estas buenas prácticas, facilitando el desarrollo y mantenimiento del proyecto. Sin embargo, no se trata de un proceso obligatorio o inamovible.

Es común, durante el aprendizaje, cuestionarse si estamos realizando correctamente esta división de responsabilidades. Lo fundamental es comprender el propósito de este concepto, los roles que desempeña cada capa y confiar en el proceso de aprendizaje. Al igual que encontrar el nombre ideal para una variable, la correcta aplicación de estos conceptos se perfecciona con la práctica. Incluso, en el futuro, podrías optar por prescindir de ciertas capas o implementar otras que se ajusten mejor a las necesidades específicas de tu aplicación. La elección de la arquitectura es, en última instancia, una decisión subjetiva que se adapta al contexto del proyecto.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ejercicio Práctico

Organización de Rutas y Capas en tu API

Sabrina y Matías se acercan a revisar tu progreso. Matías sonríe con aprobación: "Has avanzado muchísimo, pero ahora es momento de subir un nivel. Necesitamos que configures la capa de rutas de manera profesional."

Matías, con una expresión confiada, te dice: "Es hora de que trabajes como un verdadero desarrollador profesional. Sabemos que manejar rutas en el archivo principal está bien para comenzar, pero si tu aplicación crece, esa organización será insostenible."

Sabrina da un paso adelante y añade: "Por eso, queremos que trabajes con **Express Router** para separar tus rutas en módulos, que crees controladores para manejar la lógica y que comiences a trabajar con una capa de servicios para preparar datos simulados que luego enviarás a través de tus rutas."

Matías asiente y explica: "Este enfoque no solo hará que tu código sea más limpio, sino que también será más fácil de mantener y escalar. ¡Manos a la obra!"

Misión:

1. Crear rutas organizadas con Express Router
    - Migra las rutas de tu archivo principal a archivos separados en una carpeta llamada `routes` por ejemplo: `products.routes.js`
    - Usa **Express Router** para configurar el o los archivos de rutas y asegúrate de exportarlos correctamente para que pueda ser utilizado en el archivo principal.

2. Implementar controladores para manejar la lógica
    - Crea un archivo llamado `product.controller.js` dentro de la carpeta `controllers`.
    - Crea los controladores necesarios para responder a las rutas definidas en el ejercicio anterior.
    - Mueve la lógica de las rutas al controlador correspondiente y asegúrate de que las funciones sean claras y reutilizables.
3. Añadir una capa de servicios con datos simulados
    - Crea un archivo llamado `product.service.js` dentro de la carpeta `services`.
    - Simula datos en formato JSON, como una lista de productos o usuarios, y utiliza estas funciones en los controladores para devolver respuestas dinámicas.

Ejemplo práctico:
Supongamos que tienes una ruta que devuelve una lista de productos. Este es el flujo que debes implementar:

- **Archivo `productRoutes.js` (en la carpeta `routes`)**
  Define las rutas principales y vincúlalas a las funciones del controlador.
- **Archivo `productController.js` (en la carpeta `controllers`)**
  Implementa la lógica de cada ruta, como obtener productos o filtrar por categoría.
- **Archivo `productService.js` (en la carpeta `services`)**
  Crea funciones que devuelvan datos simulados para que puedan ser utilizados en el controlador.

Sabrina sonríe y dice: "Cuando termines, no olvides probar todo con **POSTMAN**. Es la herramienta ideal para asegurarte de que las rutas, controladores y servicios estén funcionando perfectamente."

Matías concluye con una mirada alentadora: "Este ejercicio es crucial para consolidar lo que has aprendido. Cuando termines, tendrás una base sólida para cualquier proyecto que emprendas. ¡Buena suerte!"

Tu desafío está claro. Ahora es momento de estructurar, modularizar y organizar como un profesional. ¡El código te espera!

Materiales y Recursos Adicionales:

Documentación Oficial de Express: expressjs.com
Explora en detalle cómo configurar rutas, middlewares y manejar errores en Express.

Documentación sobre Express Router: Sección específica de la documentación de Express.js dedicada al Router.

Preguntas para Reflexionar:

- ¿Cuáles son las ventajas de usar Express Router en comparación con definir todas las rutas directamente en la aplicación principal?
- ¿Cómo organizarías las rutas de una API REST que gestiona múltiples recursos (ej. productos, usuarios, pedidos) utilizando Express Router?
- ¿Cuál es la principal responsabilidad de un controlador en una API REST?
- ¿Por qué es importante separar la lógica de enrutamiento de la lógica de negocio en controladores separados
- ¿Qué ventajas ofrece usar servicios en cuanto a la reutilización de código?

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Próximos Pasos:

- **Modelo de datos y trabajo con JSON**: consultamos datos de forma interna y los devolvemos al cliente desde nuestra API Rest.
- **Datos en la nube**: Configurando y accediendo a datos en un servidor externo.
- **Autenticación y Autorización**: Manejando el acceso público y privado de nuestros datos.

Buenos Aires Ciudad