«Talento Tech»

Back-End

Node JS

Clase 11

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 11 - REQUEST & RESPONSE

Temario:

1. Rutas
2. CORS: solicitudes entre dominios
3. Error Handle (404)
4. Rutas parametrizadas
5. POSTMAN

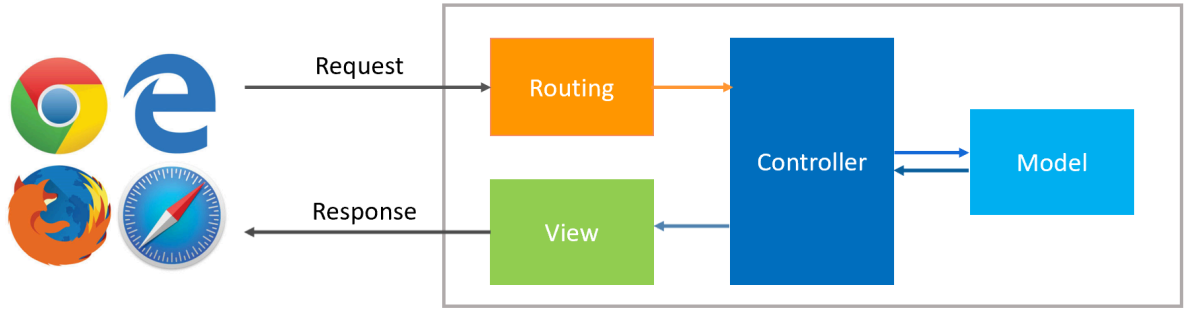
Objetivos de la Clase.

En esta clase, se busca que los estudiantes comprendan y apliquen conceptos fundamentales relacionados con el manejo de rutas en aplicaciones web, permitiéndoles estructurar de manera efectiva la navegación y la interacción entre distintas secciones de una aplicación. Además, se abordará el tema de las solicitudes entre dominios a través de CORS, explicando su importancia para la seguridad y cómo configurarlo correctamente para habilitar o restringir accesos desde diferentes orígenes. También se explorará la gestión de errores, específicamente el manejo de respuestas 404, para garantizar una experiencia de usuario más consistente y profesional en caso de recursos no encontrados. Los estudiantes aprenderán a trabajar con parámetros de ruta y de consulta (path y query params), entendiendo su uso para enviar y procesar información dinámica en las solicitudes. Finalmente, se introducirá el uso de POSTMAN como una herramienta esencial para probar y documentar las API, asegurando que los estudiantes sean capaces de verificar y depurar sus aplicaciones de manera eficiente.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Rutas

Como vimos anteriormente, en la comunicación web el cliente buscará acceder al contenido dinámico de nuestro Backend a través de peticiones HTTP a diferentes rutas o endpoints configurados en nuestra aplicación.



A través de las rutas a las que también podemos llamar ***endpoints*** vamos a definir como entregaremos la información de nuestra API Rest.

En la imagen anterior, vemos que cada request se filtra en primer lugar por la capa de ***routing*** quien decide o apunta a un controlador que implementará la lógica necesaria para devolver una respuesta. En este ejemplo, la aplicación posee una capa de vistas que devuelve HTML al cliente, sin embargo en nuestro caso, prescindimos de esta capa y la respuesta se envía en formato JSON desde cada controlador.

Recordemos que al usar el protocolo HTTP, las peticiones o requests se hacen mediante el uso de los HTTP methods.

GETPOSTPATCHPUTDELETE

Por ende, nuestro servidor no solo escuchará “paths” o rutas si no que también tendrá en cuenta el método utilizado en la request, lo cual nos permite tener rutas iguales que realicen diferentes acciones en función del método con el cual fue requerida.

Por ejemplo, no es lo mismo escuchar una ruta `/item/12345` mediante **GET**:

```
app.get('/item/12345', (req, res) => {
  // lógica para DEVOLVER el ítem solicitado
});
```

Que hacerlo mediante el método **DELETE**:

```
app.delete('/item/12345', (req, res) => {
  // lógica para ELIMINAR el ítem solicitado
});
```

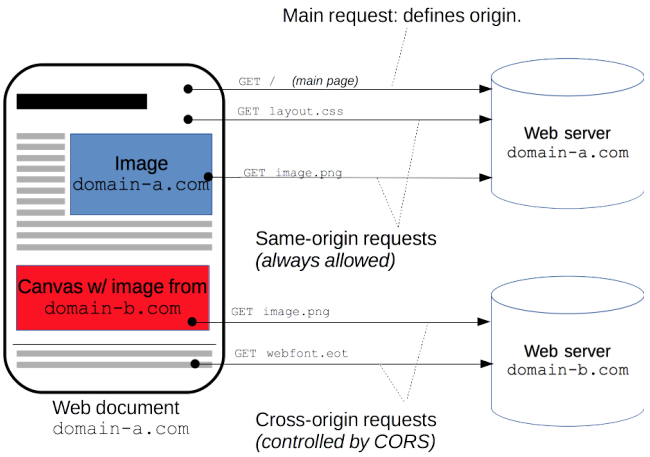
Si bien en ambos casos, la ruta es la misma, el método utilizado por el cliente al momento de realizar la petición será el que decida cual de estas 2 rutas deberá enviar la respuesta, una devolviendo la información del ítem 12345 y otra eliminando el ítem de la base de datos.

***Nota:** la variable **app** utilizada en los ejemplos anteriores es la instancia de express que invocamos al momento de crear nuestro servidor web con este framework. Si tienes dudas sobre su implementación, puedes repasar la sección "Paso a paso para crear un servidor" en la clase 9.

CORS: Solicitudes entre dominios

El Control de Acceso HTTP (CORS) es un mecanismo que utiliza headers o cabeceras HTTP adicionales para permitir que un navegador web obtenga permiso para acceder a recursos específicos en un servidor que se encuentra en un origen (dominio) diferente al del propio documento, es decir, cuando quieres acceder a los recursos de un servidor desde otro servidor diferente.

Esto ocurre cuando un user agent (como un navegador) realiza una solicitud a un recurso en un dominio, protocolo o puerto diferente al del documento actual.



Un ejemplo de una solicitud de origen cruzado es cuando el código JavaScript de una aplicación web alojada en http://dominio-a.com utiliza XMLHttpRequest para cargar el recurso http://api.dominio-b.com/data.json

Por razones de seguridad, los navegadores restringen las solicitudes HTTP de origen cruzado iniciadas por scripts. Por ejemplo, las APIs XMLHttpRequest y Fetch siguen la política de "mismo origen". Esto significa que una aplicación que utiliza estas APIs solo puede hacer solicitudes HTTP a su propio dominio, a menos que se utilicen cabeceras CORS.

¿Qué peticiones utiliza CORS?

El estándar de intercambio de origen cruzado (CORS) se utiliza para habilitar las siguientes solicitudes HTTP de sitios cruzados:

1. Invocaciones de las APIs XMLHttpRequest o Fetch en un contexto de sitio cruzado.
2. Fuentes web (utilizadas en dominios cruzados con la regla @font-face dentro de CSS), permitiendo que los servidores muestren fuentes TrueType que solo pueden ser cargadas por sitios cruzados y utilizadas por sitios web autorizados.
3. Texturas en WebGL.
4. Imágenes dibujadas en patrones utilizando drawImage.
5. Acceso a hojas de estilo (CSSOM).
6. Ejecución de scripts (para excepciones inmutables).

¿Cómo habilitar solicitudes de origen cruzado?

Hay distintas formas de hacerlo, la primera es manual y para ello vamos a utilizar el header `Access-Control-Allow-Origin` junto a otros similares.

En nuestro archivo `index.js` de nuestro servidor web, vamos a colocar el siguiente **middleware**:

```
app.use((req, res, next) => {
  // // Permitir un dominio
  res.header('Access-Control-Allow-Origin', 'https://example.com');
  // Métodos permitidos
  res.header('Access-Control-Allow-Methods', 'GET, POST, PUT,
  DELETE');
  // Encabezados permitidos
  res.header('Access-Control-Allow-Headers', 'Content-Type,
  Authorization');
  // Permitir cookies/credenciales
  res.header('Access-Control-Allow-Credentials', 'true');
  next();
});
```

El ejemplo anterior nos permite conceder determinados permisos en nuestra aplicación y al hacerlo a través de un **middleware global** nos aseguramos que las validaciones se realicen frente a cada petición.
Sin embargo, este método a veces suele resultar algo verboso y tedioso de escalar, es por ello que podemos utilizar una librería externa llamada **cors** para manejar estos permisos de forma más sencilla.

Instalamos la dependencia:

```
npm install cors
```

Configuramos un **middleware global** con la implementación de la nueva librería:

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

const app = express();

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

// Configuración avanzada: Permitir dominios específicos
const corsOptions = {
    // Dominios permitidos
    origin: ['https://example.com', 'https://anotherdomain.com'],
    // Métodos HTTP permitidos
    methods: ['GET', 'POST', 'PUT', 'DELETE'],
    // Encabezados permitidos
    allowedHeaders: ['Content-Type', 'Authorization'],
    credentials: true // Permitir cookies o credenciales
};

app.use(cors(corsOptions));
```

En el ejemplo anterior tenemos 2 implementaciones de **cors**, la primera es una implementación básica que permite peticiones de cualquier origen y la otra es una implementación avanzada que determina mayores parámetros de configuración.

Error Handle (404)

Anteriormente exploramos los middlewares y comprendimos su propósito fundamental dentro de una aplicación Express, incluso los utilizamos para configurar **CORS** en nuestro servidor. Ahora, vamos a aplicar este concepto para resolver un problema común al que nos enfrentaremos en nuestras aplicaciones: manejar las solicitudes a rutas inexistentes.

Es frecuente que un cliente envíe una solicitud a una ruta que nuestro servidor no tiene definida. Esto podría ocurrir por errores en la URL, intentos de acceso a recursos inexistentes o simples exploraciones. Como desarrolladores, debemos garantizar que el servidor responda de manera clara y consistente en estos casos, sin necesidad de crear manualmente una ruta para cada posible solicitud inválida, lo cual sería tedioso e impráctico.

Para solucionar esto, utilizamos un middleware que se ejecutará después de haber evaluado todas las rutas definidas en la aplicación. Este middleware capturará cualquier solicitud no gestionada previamente y responderá con un mensaje estandarizado, indicando que el recurso solicitado no existe. Veamos un ejemplo práctico:

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

const app = express();

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

app.get('/item/12345', (req, res) => {
```

```
  // lógica para DEVOLVER el item solicitado
});

app.delete('/item/12345', (req, res) => {
  // lógica para ELIMINAR el item solicitado
});

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

En este código, después de declarar las rutas válidas (`/item/12345` para GET y DELETE), agregamos un middleware que captura cualquier solicitud no manejada previamente. Este middleware utiliza el método app.use() y envía al cliente una respuesta con el código de estado 404, que es el estándar HTTP para indicar que un recurso no fue encontrado.

Este enfoque permite manejar automáticamente cualquier ruta no definida, como `/products` o `/users`, sin necesidad de crear manualmente respuestas individuales. Además, este procedimiento forma parte de un proceso más amplio conocido como manejo de errores o error handling, cuyo objetivo es capturar y gestionar de manera eficaz diferentes escenarios en los que la aplicación pueda fallar.

Aunque en este caso trabajamos específicamente con el código de estado **404** para rutas no encontradas, el manejo de errores puede abarcar otros escenarios comunes. Por ejemplo:

- **400 (Bad Request)** para solicitudes mal formadas o con datos inválidos.
- **401 (Unauthorized)** y **403 (Forbidden)** para problemas relacionados con autenticación o permisos.
- **500 (Internal Server Error)** para errores internos del servidor.

Implementar un buen manejo de errores es esencial para garantizar que nuestra aplicación no solo sea funcional, sino también robusta y confiable, brindando siempre respuestas claras y adecuadas a los usuarios frente a cualquier eventualidad.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Rutas Parametrizadas

Ahora que estamos más familiarizados con cómo funcionan las rutas para comunicar al cliente con el servidor, vamos a descubrir todo el potencial que tienen y cómo podemos aprovecharlas al máximo. Hasta ahora, podrías haber trabajado con rutas estáticas, como esta:

```
app.get('/item/12345', (req, res) => {
  // lógica para DEVOLVER el item solicitado
});
```

En este caso, el servidor solo responde si el cliente solicita específicamente el ítem **12345**. Pero, ¿qué pasa si nuestra tienda tiene 300 productos diferentes? ¿Vamos a crear 300 rutas para cada uno? Sería una locura, ¿no? Por suerte, aquí entra en acción un concepto súper útil: **los parámetros de ruta**, o **path params**.

¿Qué son los path params?

Los **path params** nos permiten crear rutas **dinámicas** en lugar de rutas fijas. En lugar de especificar un valor como 12345, podemos usar una variable que capture cualquier valor que el cliente envíe en esa posición. Por ejemplo:

```
app.get('/item/:id', (req, res) => {
   // Captura el valor dinámico de la ruta
   const itemId = req.params.id;
   // lógica para DEVOLVER el item solicitado
});
```

Aquí, el `:id` funciona como un marcador de posición. Lo que sea que el cliente envíe en el lugar de `:id` será capturado y estará disponible en `req.params.id`. Así, si el cliente pide `/item/abc123` o `/item/xyz456`, nuestro servidor puede manejar ambas solicitudes sin necesidad de crear rutas adicionales.

¿Cómo funciona en la práctica?

Cuando el cliente solicita una ruta como `/item/78910`, Express reemplaza el marcador `:id` con el valor real (`78910` en este caso) y lo coloca en la propiedad `req.params`. Luego puedes usar ese valor para cualquier lógica, como buscar el ítem en una base de datos:

```
app.get('/item/:id', (req, res) => {
  const itemId = req.params.id;
  // Aquí puedes hacer algo como buscar en tu base de datos
  res.send(`Devolviendo el ítem con ID: ${itemId}`);
});
```

Esto es especialmente útil cuando trabajamos con recursos únicos, como productos, usuarios o artículos, porque nos permite reutilizar una sola ruta para manejar múltiples solicitudes.

¿Qué pasa con los query params?

Además de los **path params**, tenemos otro tipo de parámetros que son igual de importantes: los **query params**. A diferencia de los path params, los query params no forman parte del "path" de la URL. En cambio, se envían como pares clave-valor después de un signo de interrogación (`?`). Por ejemplo:

```
/items?category=electronics&price=low
```

En este caso, `category` y `price` son query params. Los query params son útiles para enviar datos adicionales, como filtros o configuraciones, sin necesidad de modificar la estructura de la ruta.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

En Express, puedes acceder a ellos mediante `req.query`. Veamos un ejemplo:

```
app.get('/items', (req, res) => {
    const category = req.query.category;
    const price = req.query.price;

    res.send(`Categoría: ${category}, Precio: ${price}`);
  });
```

Path params vs. Query params: ¿Cuándo usar cuál?

- **Path params**: Úsalos cuando el dato sea fundamental para identificar un recurso específico, como el ID de un producto o usuario (`/item/:id`).
- **Query params**: Úsalos cuando el dato sea opcional o esté relacionado con filtros, búsquedas o configuraciones (`/items?category=electronics`).

*TIP: Los path params y query params pueden combinarse.
Por ejemplo: `/items/:id?includeDetails=true` te permite capturar tanto el ID del ítem como el valor del query param `includeDetails`.

POSTMAN

POSTMAN es una aplicación que sirve como cliente HTTP para enviar solicitudes a servidores y recibir respuestas. Su interfaz gráfica facilita realizar operaciones que normalmente requerirían comandos más complejos, como enviar solicitudes GET, POST, PUT o DELETE. Además, permite agregar encabezados personalizados, manejar autenticación y trabajar con datos en formato JSON, entre otras funcionalidades.

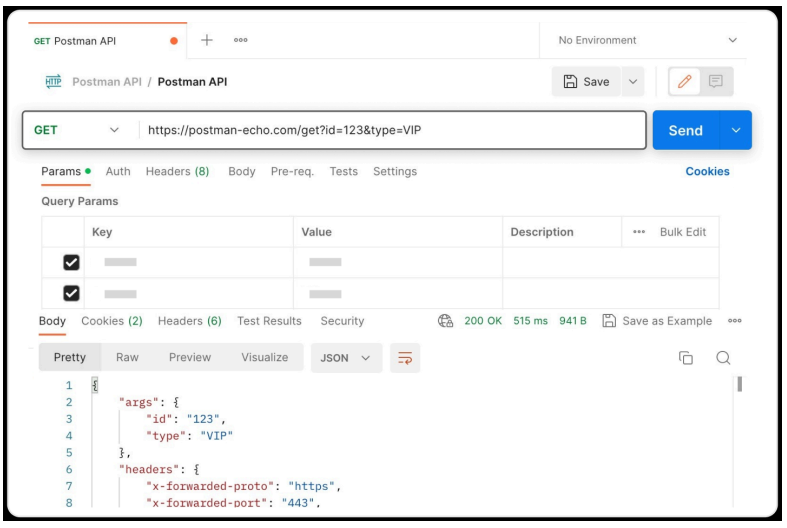
Buenos Aires aprende
Agencia de Habilidades para el Futuro

Para hacernos una idea, actualmente la única forma que tenemos para realizar una petición o request a través del método HTTP POST es mediante un formulario de HTML en una aplicación frontend. Mediante el uso de POSTMAN (o herramientas similares como insomnia) podemos realizar este tipo de peticiones y muchas más sin encontrarnos en el entorno real de implementación, lo cual resulta extremadamente útil para probar las respuestas de nuestro servidor al momento de estar desarrollando.

¿Cómo usar POSTMAN?

1. **Instalación**:
    - Descarga POSTMAN desde su sitio oficial (https://www.postman.com/) e instálalo en tu sistema operativo.
    - También puedes usar la versión web si prefieres no instalar software adicional.

2. **Primeros pasos con una solicitud**:
    - Abre POSTMAN y crea una nueva solicitud seleccionando el método HTTP (GET, POST, etc.) en el menú desplegable.
    - Ingresa la URL de tu API en el campo correspondiente, como http://localhost:3000/item/12345.
    - Haz clic en el botón "Send" para enviar la solicitud.
    - Observa la respuesta del servidor en la sección de resultados, donde se mostrarán el cuerpo, encabezados y el código de estado.



Casos prácticos con POSTMAN

- **Probar rutas dinámicas**: Si tu API tiene una ruta como /item/:id, puedes probarla ingresando una URL como http://localhost:3000/item/12345 y verificando la respuesta.
- **Simular errores**: Prueba enviar datos incompletos o en formatos incorrectos para ver cómo responde tu API a errores comunes. Esto te ayudará a mejorar el manejo de errores y validar solicitudes.
- **Automatizar pruebas**: POSTMAN tiene una funcionalidad llamada Tests, donde puedes escribir pequeños scripts en JavaScript para verificar automáticamente las respuestas, cómo asegurarte de que el código de estado sea 200 o que ciertos campos existan en el JSON de respuesta.

Ejercicio Práctico

Configuración de Rutas y Manejo de Parámetros

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."

Sabrina se cruza de brazos y asiente. "Exacto. Queremos que aprendas a organizar tus rutas, además de implementar un par de configuraciones clave como **CORS** y un manejo de errores para rutas inexistentes. Esto te ayudará a preparar tu servidor para escenarios reales."

Matías continúa: "Pero eso no es todo. También queremos que practiques cómo manejar datos dinámicos que llegan a través de las solicitudes. Deberás configurar rutas que lean tanto **path params** como **query params** y luego devolver esa información en las respuestas. Y no te olvides de usar **POSTMAN** para validar todo."

Misión:

1. Establece 4 rutas en el archivo de entrada de la aplicación. Asegúrate de:
    - Permitir solicitudes entre dominios configurando **CORS**.
    - Agregar un middleware que capture las rutas no encontradas y devuelva una respuesta 404.
2. Define rutas que utilicen:
    - **Path params**: Por ejemplo, una ruta que devuelva información basada en el ID de un producto o usuario.
    - **Query params**: Una ruta que filtre datos con base en parámetros como categoría o precio.

Usa **POSTMAN** para probar las rutas que configuraste. Asegúrate de capturar correctamente los parámetros enviados y mostrar la información de manera clara en las respuestas.

Sabrina sonríe y agrega: "Recuerda, la organización es clave. Cuando logres que todo funcione, será el momento perfecto para revisar cómo cada pieza encaja en el proyecto. ¡Buena suerte!"

Tu desafío está claro. Ahora es momento de implementar lo aprendido y demostrar tus habilidades.

Materiales y Recursos Adicionales:

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

Postman Learning Center: learning.postman.com
Guías interactivas para aprender a usar POSTMAN, desde conceptos básicos hasta funciones avanzadas como automatización y pruebas.

Artículo: “Understanding CORS”: Mozilla Developer Network (MDN)
Una guía detallada sobre qué es CORS, cómo funciona y cómo configurarlo correctamente.

Repositorio de Ejemplo: Explora ejemplos básicos de configuración de servidores Express en GitHub (Node.js Express Examples).

Preguntas para Reflexionar:

- ¿Por qué es importante organizar la estructura de un proyecto con capas como rutas, controladores y modelos? ¿Qué beneficios aporta a largo plazo?
- ¿Cómo puedes garantizar que tu API sea accesible para aplicaciones que no comparten el mismo dominio mediante el uso de CORS?
- ¿Qué diferencias encuentras entre **path params** y **query params**? ¿En qué escenarios usarías cada uno?
- ¿Cómo manejarías la validación de parámetros de ruta o consulta para asegurarte de que solo se procesen datos válidos en tu API?
- ¿Cómo contribuye el manejo de errores 404 y otras respuestas HTTP claras a mejorar la experiencia del usuario y el desarrollo de tu aplicación?

Próximos Pasos:

- **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.
- **Datos en la nube**: Configurando y accediendo a datos en un servidor externo.

Buenos Aires Ciudad