Guía de estudio 2 - Introducción a Express Js

 ¡Hola! Te damos la bienvenida a esta nueva guía de estudio.

¿En qué consiste esta guía?

La siguiente guía de estudio tiene como objetivo recordar y repasar los contenidos que hemos visto en clase.

Dentro de los que se encuentran:

- Crear desde el backend una ruta GET usando Express Js.
- Crear desde el backend una ruta GET para listar un conjunto de recursos en Express Js.
- Crear una ruta POST en un backend para agregar datos en un JSON local.

¡Vamos con todo!

Tabla de contenidos

¡Comencemos!

¿Por qué aprender Express?

Express Js es un framework de JavaScript para la creación de servidores y API REST en un entorno Node Js.

Express nos hará muy sencillo crear endpoints que luego podremos consultar desde el navegador, Postman o desde una API.

Es utilizado por varias de las empresas más grandes del mundo como PayPal, Uber, Netflix, LinkedIn, IBM y muy probablemente también se esté usando en tu aplicación favorita.

Los desarrolladores que prefieren Express Js como herramienta de trabajo, típicamente lo ocupan por ser significativamente minimalista, con una sintaxis intuitiva y declarativa, además del poco tiempo de desarrollo y líneas de código que se necesita para empezar servidores y servicios web.

Características de Express

Express Js destaca entre tantas cosas por sus características sobresalientes:

- **Creación rápida de servidores**
  En muy pocas líneas de código podemos conseguir levantar un servidor o servicio web. La única dependencia o requerimiento es tener instalado Node Js.

- **Rendimiento óptimo**
  Al tratarse finalmente de JavaScript, la ejecución de los servidores tienen un rendimiento significativo en comparación con otros frameworks de backend.

- **Minimalista**
  Su capa de abstracción está representada por métodos con nombres muy declarativos que facilitan el uso del framework sin necesidad de conocer sus procesos internos.

- **Flexible**
  Express permite desarrollar aplicaciones con diferentes estructuras de proyecto como el Modelo Vista Controlador. Además, las integraciones de plugins no representan ninguna complejidad ni complicación.

- **Funcionalidades de Node**
  Ya que su creación fue hecha bajo este entorno de ejecución de Node, los servidores y servicios web pueden aprovechar todas las potencialidades y módulos para aumentar el alcance funcional.

- **Renderización de contenido dinámico**
  Express tiene una amplia compatibilidad con motores de plantillas como Pug, Ejs, Handlebars, entre otros.

- **Documentación**
  En su página oficial se puede encontrar la documentación escrita por su equipo de desarrollo, la cual está disponible en varios idiomas.

- **Comunidad**
  Para comienzos del año 2022, solo en NPM se registran más de 20 millones de descargas, sin contar sus subpaquetes creados por desarrolladores en todo el planeta.

Conceptos básicos

¿Qué son los servidores Web?

Un servidor web es una aplicación backend creada para recibir solicitudes de aplicaciones web y devolver información o inclusive sitios web.

En este módulo construiremos estas aplicaciones utilizando Node y Express y las consumiremos utilizando distintas estrategias dependiendo de lo que queramos lograr.

Arquitectura Cliente Servidor

Cuando trabajamos en FullStack usualmente tenemos al menos dos capas, una es el cliente y la otra el servidor.

El cliente le hace un pedido al servidor, y el servidor le responde.

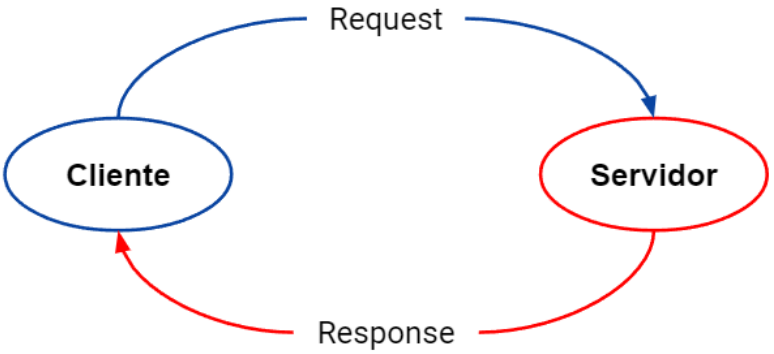


Imagen 1. Diagrama del request y el response
Fuente: Desafío Latam

El cliente puede ser un navegador, una aplicación como postman, una página web o una aplicación en React o incluso una aplicación móvil. El servidor analizará el pedido y entregará una respuesta.

¿Qué es Express?

Express es un framework que nos permite construir un servidor web de forma sencilla sobre node. Construir una APP en express será muy similar a lo que trabajamos en la unidad anterior.

Hola mundo en Express

Nuestro primer proyecto será una aplicación en express que devuelva un mensaje con el texto **Hola mundo** que podremos abrir directamente con el navegador.

Instalación de Express Js

Para empezar a utilizar Express js primero hay que instalarlo por **npm**, para esto crea una nueva carpeta que llamaremos **holamundo**, te recomendamos utilizar solo minúsculas en el nombre de la carpeta.

Dentro de la carpeta ejecuta la siguiente línea de comando para iniciar un proyecto NPM:

```
npm init -y
```

Ahora instalemos Express con la siguiente línea de comando:

```
npm install express
```

Luego de instalarse deberás ver “express” en el archivo package.json

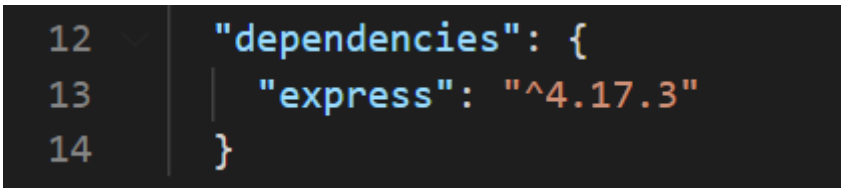


Imagen 2. Express como dependencias en el package.json
Fuente: Desafío Latam

Una vez instalado crearemos el archivo index.js en la raíz del proyecto con el siguiente código:

```
const express = require('express')
const app = express()

app.listen(3000, console.log("¡Servidor encendido!"))

app.get("/home", (req, res) => {
    res.send("Hello World Express Js")
})
```

Levantamiento del servidor

Para poder ver el proyecto con el navegador tendremos que levantar el servidor, esto lo lograremos ejecutando la siguiente línea de código en el terminal

```
node index.js
```

Al ejecutar la página verás en el terminal el mensaje ¡Servidor encendido!

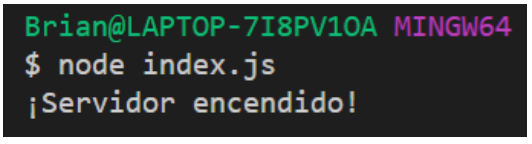


Imagen 3. Levantamiento del servidor
Fuente: Desafío Latam

Ahora que nuestro servidor está levantado, solo deberás abrir el navegador y escribir la siguiente URL:

http://localhost:3000/home

Y verás lo siguiente:

Imagen 4. Consultando la ruta **/home** con el navegador
Fuente: Desafío Latam

Advertencia: Para poder ver cambios en el sitio después de actualizar el código, se debe reiniciar el servidor para poder ver los cambios. Esto implica cerrarlo y luego volver a levantarlo. Al final de la unidad estudiaremos otra forma más cómoda de mantenerlo siempre actualizado.

Cerrar el servidor

Cuando levantamos un servidor de Express, este queda a la espera de solicitudes bloqueando la terminal:

```
Brian@LAPTOP-7I8PV1OA MINGW64 ~/Desktop/Clase de Express
$ node index.js
¡Servidor encendido!
```

Imagen 5. Terminal bloqueada por servidor encendido
Fuente: Desafío Latam

Para que los cambios que hagamos en el código del servidor sean aplicados es necesario apagar el servidor y volverlo a levantar.

Para bajar o apagar el servidor, presiona **Crtl + C** en Windows y linux, o **Cmd + C** en Mac

```
Brian@LAPTOP-7I8PV1OA MINGW64 ~/Desktop/Clase de Express
$ node index.js
¡Servidor encendido!

Brian@LAPTOP-7I8PV1OA MINGW64 ~/Desktop/Clase de Express
$
```

Imagen 6. Terminal bloqueada por servidor encendido
Fuente: Desafío Latam

Revisando el código paso a paso.

1. Importamos el paquete express y se ejecuta para obtener un enrutador (app)

```
const express = require('express')
const app = express()
```

2. Especificamos en qué puerto se levantará el servidor y se declara un mensaje por consola al levantarse

```
app.listen(3000, console.log("¡Servidor encendido!"))
```

3. Creamos un endpoint de tipo **GET** con un la dirección **/home** que al consultarse devolverá un String con el texto **"Hello World Express Js"**

```
app.get("/home", (req, res) => {
    res.send("Hello World Express Js")
})
```

¿Cómo extender nuestra primera aplicación?

Con la instancia ***app*** podremos crear todas las rutas que necesitemos. Solo debemos cumplir la siguiente sintaxis:

```
app.method( path, callback )
```

En donde:

- **app:** Es una instancia de Express que cargamos en el paso 1.
- **method:** Es un método de solicitud HTTP en minúscula. Aquí utilizaremos principalmente **get** y **post**.
- **path:** Es la ruta que será consultada por la aplicación cliente, por ejemplo /home
- **callback:** Es la función que se ejecutará cuando la ruta sea consultada.

Podemos crear tantas rutas como necesitemos, ubicándolas una debajo de otra en el código, cómo se ve en el siguiente ejemplo:

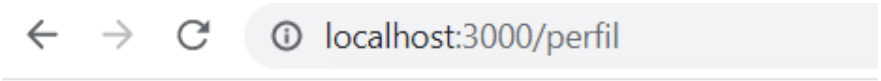
```
const express = require('express')
const app = express()

app.listen(3000, console.log("¡Servidor encendido!"))

app.get("/home", callback)

app.get("/perfil", callback)

app.get("/productos", callback)

app.post("/productos", callback)

app.get("/carrito", callback)

app.delete("/usuarios", callback)
```

Tip avanzado: Existen diferentes estructuras de proyectos que nos permiten organizar el código del servidor en diferentes archivos, separando la lógica de cada una de las rutas en módulos que puedan ser importados en el script principal. Una muy utilizada es MVC (Modelo - Vista - Controlador).

¡Manos a la obra! - Express a la obra

Crea una nueva carpeta titulada **express a la obra** y:

1. Inicia un proyecto de npm
2. Instala el paquete de express
3. Importa el paquete de express
4. Crear un servidor en el puerto 3000
5. Crea una ruta **GET /perfil** que devuelva un string con tu nombre
6. Consulta la ruta creada desde el navegador



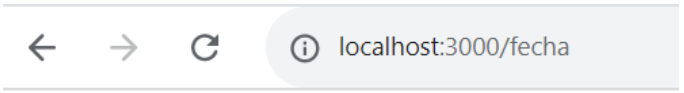
Elon Reeve Musk

Imagen 7. Resultado del ejercicio 1
Fuente: Desafío Latam

¡Manos a la obra! - Ruta GET /fecha

En la misma carpeta **express a la obra** crea otra ruta **GET /fecha** que devuelva la fecha actual usando una instancia del objeto Date:

```
const fecha = new Date()
```



"2022-04-25T19:17:56.422Z"

Imagen 8. Resultado del ejercicio 2
Fuente: Desafío Latam

Request y response

Para realizar aplicaciones web más complejas que la del ejemplo necesitaremos utilizar la información del pedido (request) y entender cómo preparar una respuesta (response).

El flujo de request y response es muy similar al de un restaurante:

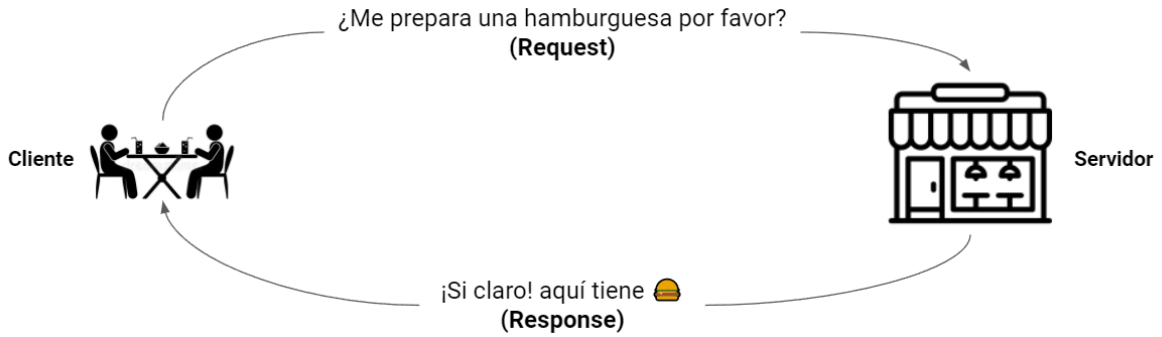


Imagen 9. Diagrama de ejemplo de la arquitectura Cliente - Servidor
Fuente: Desafío Latam

Los objetos request y response

En Express el **request** y el **response** son objetos que podemos utilizar para obtener detalles de la consulta y poder dar una respuesta.

Estos objetos están compuestos de métodos y atributos que son de gran utilidad en la comunicación de la aplicación cliente con nuestro servidor:

Request:

- **req.query:** Devuelve los parámetros escritos en la URL (query string)
- **req.params:** Devuelve los parámetros como subrecursos en la URL, por ejemplo: /productos/**:id**
- **req.body:** Devuelve el payload o cuerpo de la consulta
- **req.headers:** Devuelve un objeto con todas las cabeceras declaradas en la consulta

Response:

- **res.send():** Devuelve como respuesta de la consulta lo que sea asignado como argumento
- **res.json():** Devuelve en formato JSON lo que sea asignado como argumento
- **res.sendFile():** Permite devolver un archivo como respuesta de la consulta
- **res.status():** Define el status code que quiera devolverse al cliente de la consulta

Devolviendo un JSON en una ruta

Podemos crear rutas en nuestro servidor que tengan como objetivo específicamente devolver o recibir datos a una aplicación cliente en formato JSON, haciendo esto estaríamos desarrollando finalmente una **API REST**.

El método que usaremos para entregar información a partir de una consulta será bajo el método **GET.**

Realicemos este ejercicio creando un archivo *productos.json* en la misma carpeta donde creamos el proyecto "**Hola mundo"** con el siguiente contenido:

```
[
    {
        "id": 1,
        "nombre": "Audífonos",
        "precio": 18990
    },
    {
        "id": 2,
        "nombre": "Aro de Luz",
        "precio": 21990
    }
]
```

Ahora creemos una ruta **GET /productos** en nuestro servidor y utilicemos el objeto response de la función callback para devolver el JSON creado.

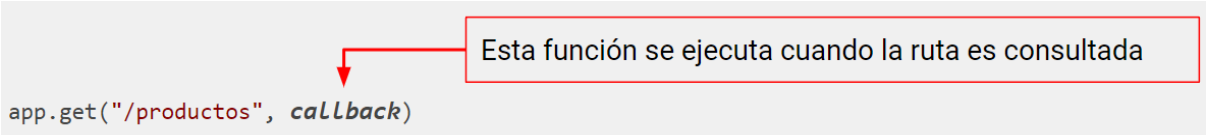


Imagen 10. Función de callback en una ruta
Fuente: Desafío Latam

Para esto será necesario importar el módulo File System para leer el archivo productos.json y posteriormente devolverlo al cliente. El código del servidor quedaría así:

```
const express = require('express')
const app = express()
const fs = require('fs')

app.listen(3000, console.log("¡Servidor encendido!"))

app.get("/productos", (req, res) => {
    const productos = JSON.parse(fs.readFileSync("productos.json"))
    res.json(productos)
})
```

Las líneas de colores representan el código nuevo en el servidor.

Ahora realicemos una consulta HTTP con la extensión Thunder Client de VSC consultando la siguiente ruta: http://localhost:3000/productos.

Si no tienes instalada la extensión Thunder Client, también puedes hacer la consulta de la ruta desde el navegador.

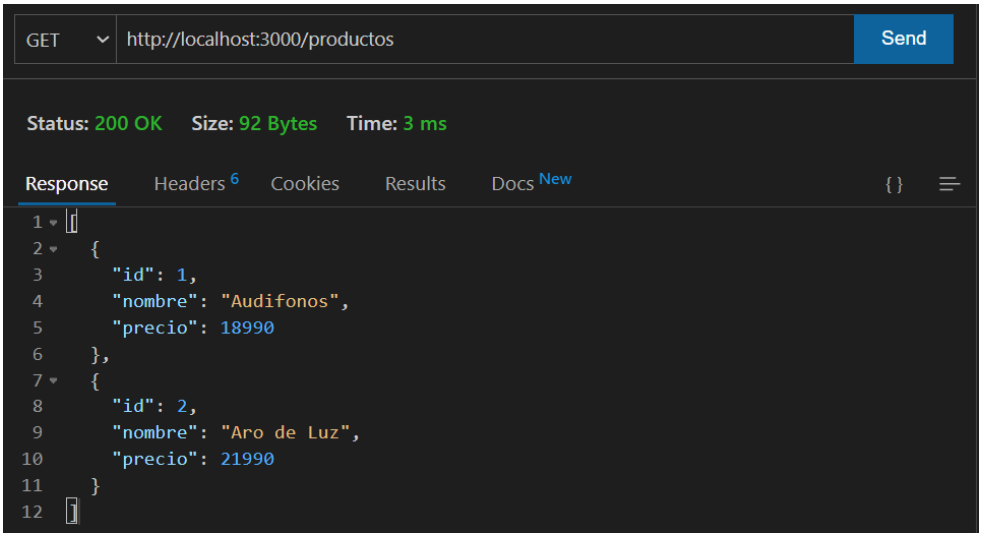


Imagen 11. Respuesta de la consulta a la ruta **/productos**
Fuente: Desafío Latam

¡Manos a la obra! - GET /usuarios

Continuando con el proyecto ***express a la obra***, crea una ruta **GET /usuarios** que devuelva un JSON con 2 usuarios.

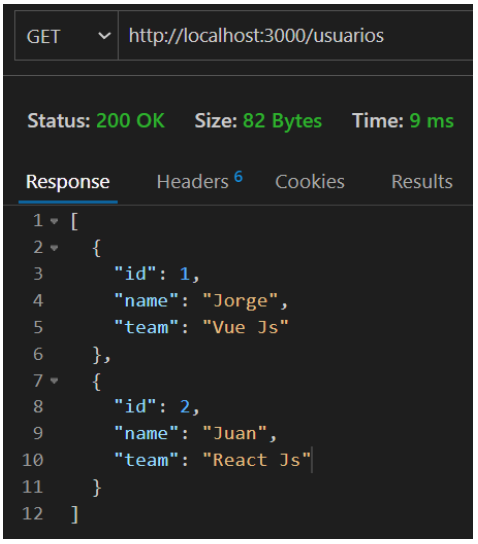


Imagen 12. Resultado del ejercicio 3
Fuente: Desafío Latam

Recibiendo un payload

Payload es un término que suena complejo, pero simplemente son los datos que el cliente envía.

En algunas ocasiones no se enviarán datos adicionales y en otras sí. Los casos previos que hemos visto en la guía no requerían enviar datos adicionales, pero por ejemplo, si quisiéramos crear un artículo o un usuario, tendríamos que enviar toda la información para crearlo, estos son justamente el tipo de datos que se envían en el payload.

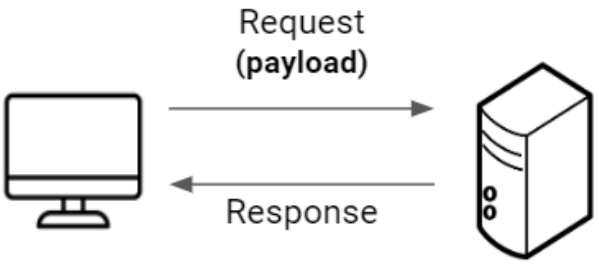


Imagen 13. Consulta al servidor con payload
Fuente: Desafío Latam

En express esta carga se encuentra en el atributo body del objeto request

```
Request: {
  body: {
    "id": 3,
    "nombre": "Monitor",
    "precio": 59990
  }
}
```

Para poder leer el payload es necesario realizar una pequeña configuración en nuestra instancia **app**. Esta configuración consiste en ejecutar una función antes de cada ruta para parsear el contenido enviado desde el cliente. Para entender bien lo que vamos a hacer tenemos que conocer el concepto de middleware

¿Qué son los middlewares?

Un middleware, particularmente un HTTP middleware, es una capa intermedia que envuelve la aplicación con la finalidad de procesar el request y el response.

Los middleware pueden ser usados para procesar fácilmente los payloads, añadir headers, comprimir los response, manejar autenticación.

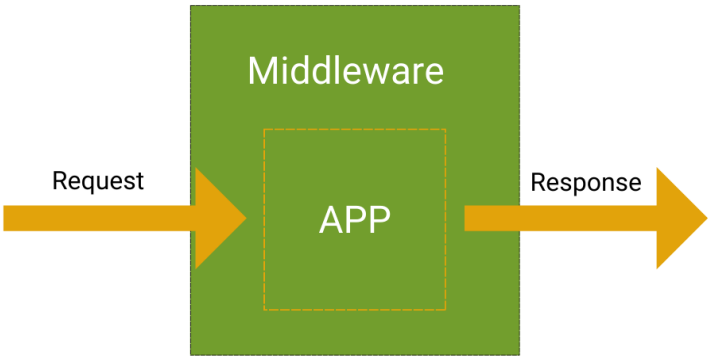


Imagen 14. El middleware envuelve la aplicación.
Fuente: Desafío Latam

En Express Js los middlewares se declaran con el método ***.use()***, pasando como argumento el middleware.

El siguiente diagrama nos ayudará a entender cómo funciona

Imagen 15. Diagrama de middlewares
Fuente: Desafío Latam

Integremos un middleware en nuestro servidor que sirva específicamente para parsear el cuerpo de una consulta y permitirle a nuestras rutas acceder los payload de cada consulta.

Para eso agregamos la siguiente línea en nuestro servidor:

```
const express = require('express')
const app = express()
const fs = require('fs')
```

```
app.listen(3000, console.log("¡Servidor encendido!"))

app.use(express.json())

app.get("/productos", (req, res) => {
    const productos = JSON.parse(fs.readFileSync("productos.json"))
    res.json(productos)
})
```

Ahora creemos una ruta **POST** que sea utilizada para agregar un nuevo producto en nuestro JSON local.

```
app.post("/productos", (req, res) => {

    // 1
    const producto = req.body

    // 2
    const productos = JSON.parse(fs.readFileSync("productos.json"))

     // 3
    productos.push(producto)

    // 4
    fs.writeFileSync("productos.json", JSON.stringify(productos))

    // 5
    res.send("Producto agregado con éxito!")
})
```

En este código estamos:

1. Almacenando en una constante llamada ***producto*** el payload de la consulta(req.body)
2. Creando una constante **productos** que almacena el contenido de *productos.json* parseado.
3. Agregando el producto recibido en la consulta al arreglo de productos.
4. Sobrescribiendo *productos.json* por el nuevo arreglo de productos que incluye el producto recibido.
5. Respondiendo al cliente un pequeño mensaje indicando que el producto fue agregado con éxito.

Ahora probemos nuestra ruta **POST /productos** realizando una consulta HTTP con la extensión Thunder Client pasando como payload lo siguiente:

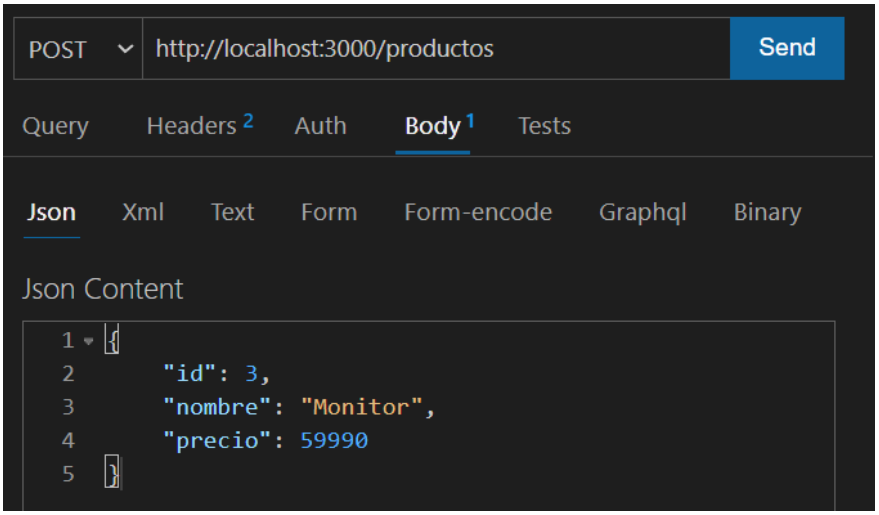


Imagen 16. Probando la ruta **POST /productos**
Fuente: Desafío Latam

Al emitir esta consulta recibimos el siguiente mensaje como respuesta:

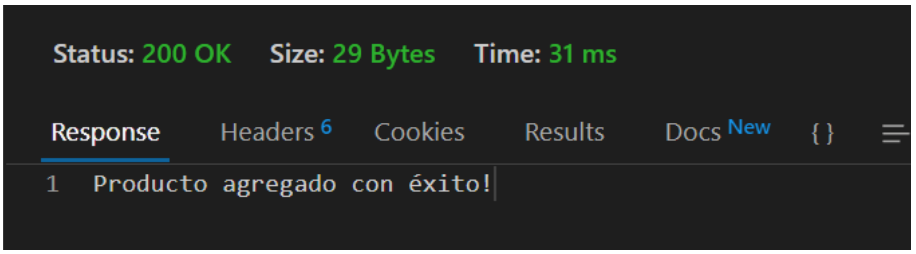


Imagen 17. Respuesta de la ruta **POST /productos**
Fuente: Desafío Latam

Ahora si revisamos nuestro archivo local productos.json notaremos que el Monitor fue agregado:

Imagen 18. Archivo *productos.json* sobrescrito con un nuevo producto
Fuente: Desafío Latam

¡Manos a la obra! - POST /usuarios

Continuando con el proyecto ***express a la obra***, crea una ruta **POST /usuarios** que reciba el registro de un nuevo usuario y lo agregue al archivo usuarios.json.

Deberás probar esta ruta con la extensión Thunder Client

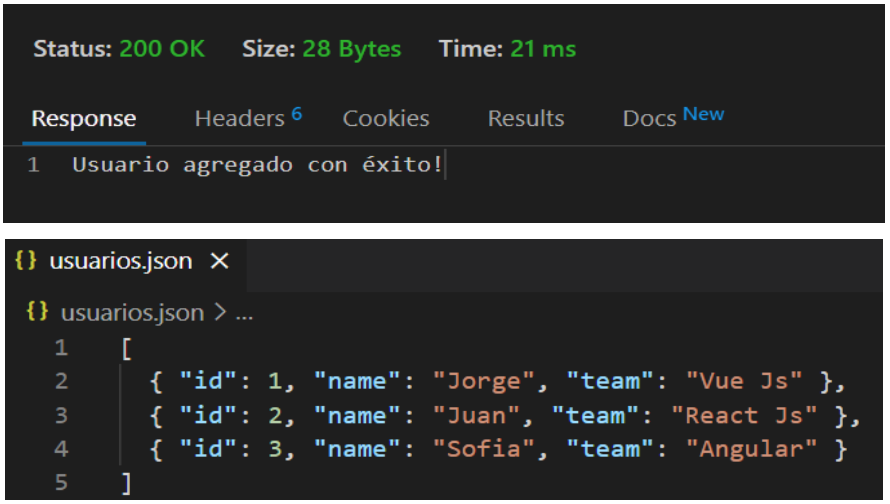


Imagen 19. Resultado del ejercicio 4
Fuente: Desafío Latam

Eliminando un recurso

Ahora que ya sabemos listar recursos y agregar nuevos, procedamos a implementar la funcionalidad de borrado utilizando una ruta **DELETE /productos/:id**

En esta ruta será necesario que el cliente nos especifique el id del producto a eliminar, para esto ocuparemos los parámetros de la URL con el ***req.params***.

En el proyecto **"Hello World Express Js"** importa el módulo *fs* y agrega la siguiente ruta:

```
app.delete("/productos/:id", (req, res) => {
    const { id } = req.params
    const productos = JSON.parse(fs.readFileSync("productos.json"))
    const index = productos.findIndex(p => p.id == id)
    productos.splice(index, 1)
    fs.writeFileSync("productos.json", JSON.stringify(productos))
    res.send("Producto eliminado con éxito")
})
```

El ***req.params*** nos retorna un objeto con todos los parámetros de la URL, cada parámetro se identificará por los 2 puntos(:) antes del nombre del parámetro.

Para eliminar el producto del arreglo ocupamos el método ***splice*** de los arreglos, posteriormente sobreescribimos el archivo *productos.json*. Cuando trabajemos con bases de datos haremos esto mismo de forma más eficiente.

Probemos esta ruta consultando con el Thunder Client y pasando un "2" como valor del parámetro id: **http://localhost:3000/productos/2**

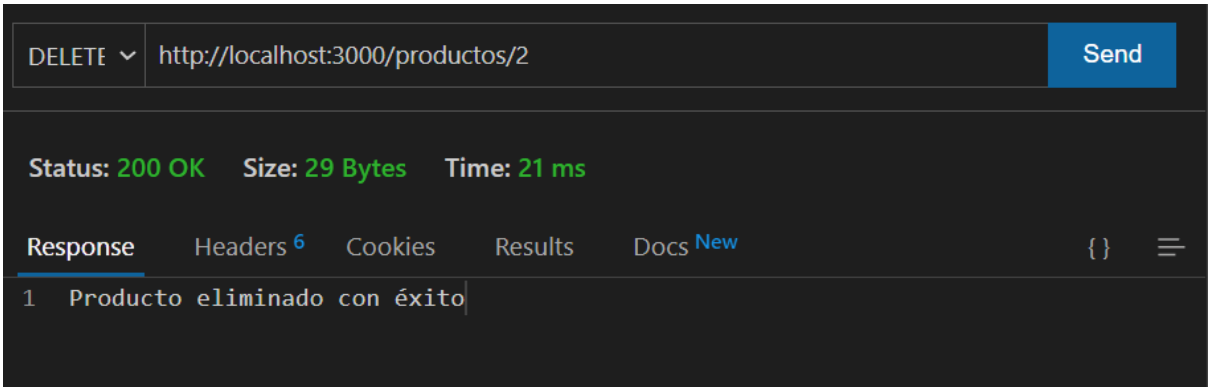


Imagen 20. Probando la ruta **POST /productos/:id**
Fuente: Desafío Latam

Ahora si vemos el archivo *productos.json* notaremos que ya no está el producto de id 2

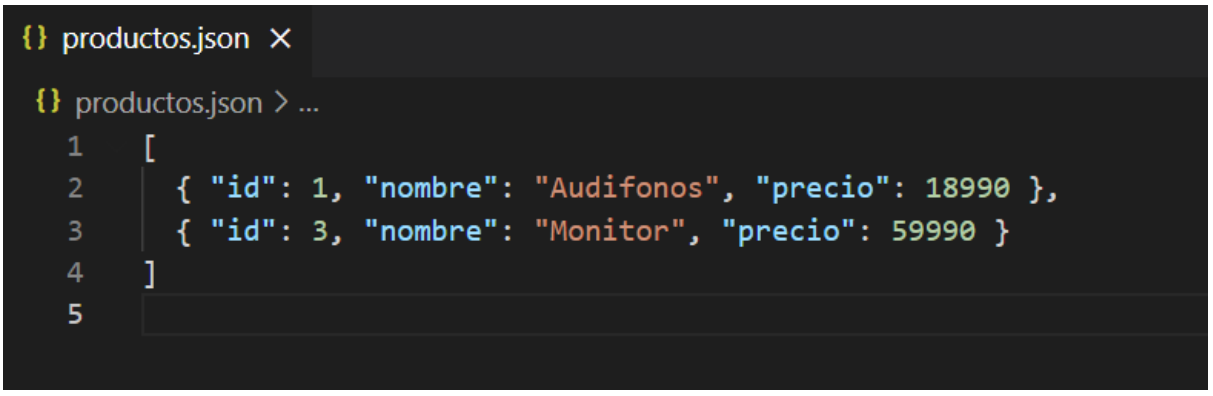


Imagen 21. JSON de productos luego de eliminarse un producto
Fuente: Desafío Latam

¡Manos a la obra! - DELETE /usuarios

Continuando con el proyecto ***express a la obra***, crea una ruta **DELETE /usuarios** que reciba como parámetro el id de un usuario y lo elimine del archivo *usuarios.json*.

Deberás probar esta ruta con la extensión Thunder Client y posteriormente revisar el JSON.

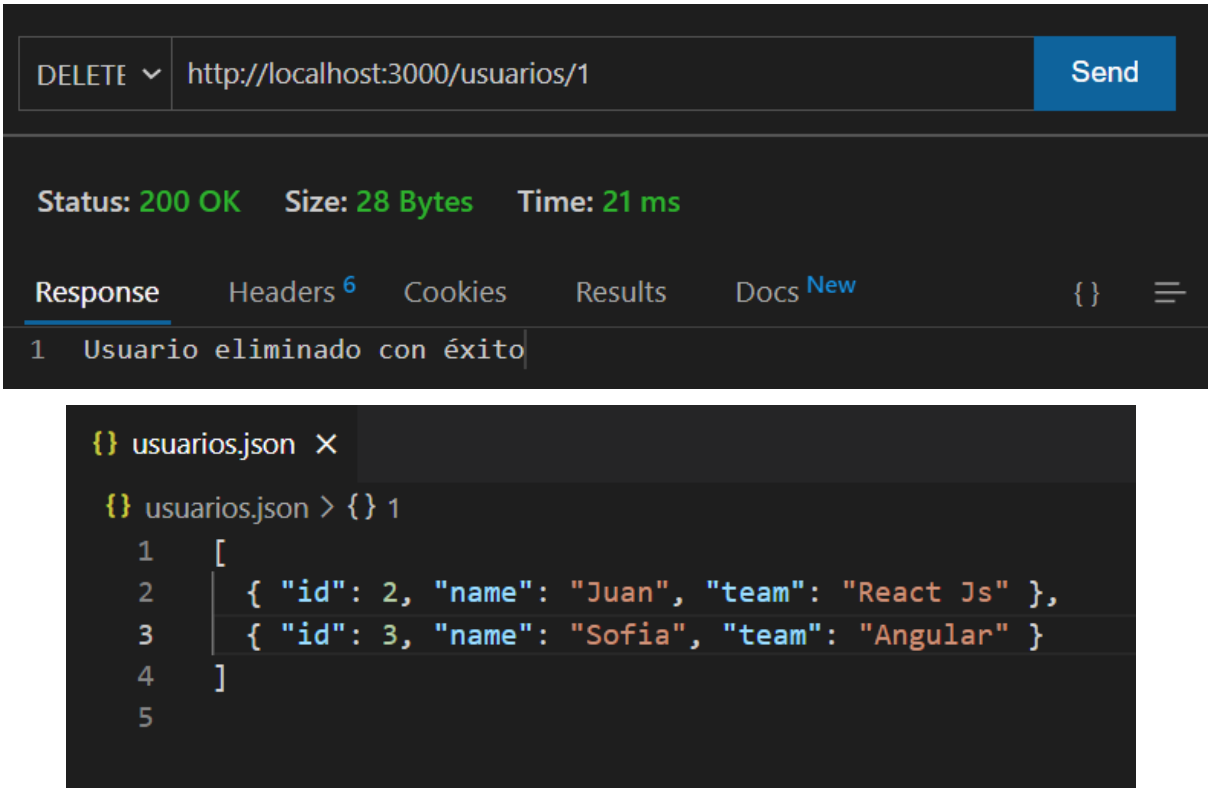


Imagen 22. Resultado del ejercicio 5
Fuente: Desafío Latam

Modificando un recurso

Para permitirle a una aplicación cliente modificar alguno de los productos que tenemos almacenados en el JSON crearemos una ruta **PUT /productos/:id**

Esta ruta tiene la particularidad de recibir el **id** del producto a modificar como parámetro en la ruta, además esperará recibir como payload el objeto del producto modificado para sustituirlo en el JSON.

En el proyecto **"Hello World Express Js"** agrega la siguiente ruta **PUT /productos/:id**

```
app.put("/productos/:id", (req, res) => {
    const { id } = req.params
    const producto = req.body
    const productos = JSON.parse(fs.readFileSync("productos.json"))
    const index = productos.findIndex(p => p.id == id)
    productos[index] = producto
    fs.writeFileSync("productos.json", JSON.stringify(productos))
    res.send("Producto modificado con éxito")
})
```

Probemos esta nueva ruta realizando una consulta con Thunder Client usando esta URL:

http://localhost:3000/productos/1

Y este body en donde estamos agregando la tilde a “Audífonos” y cambiándole el precio:

```
{
    "id": 1,
    "nombre": "Audífonos",
    "precio": 22990
}
```

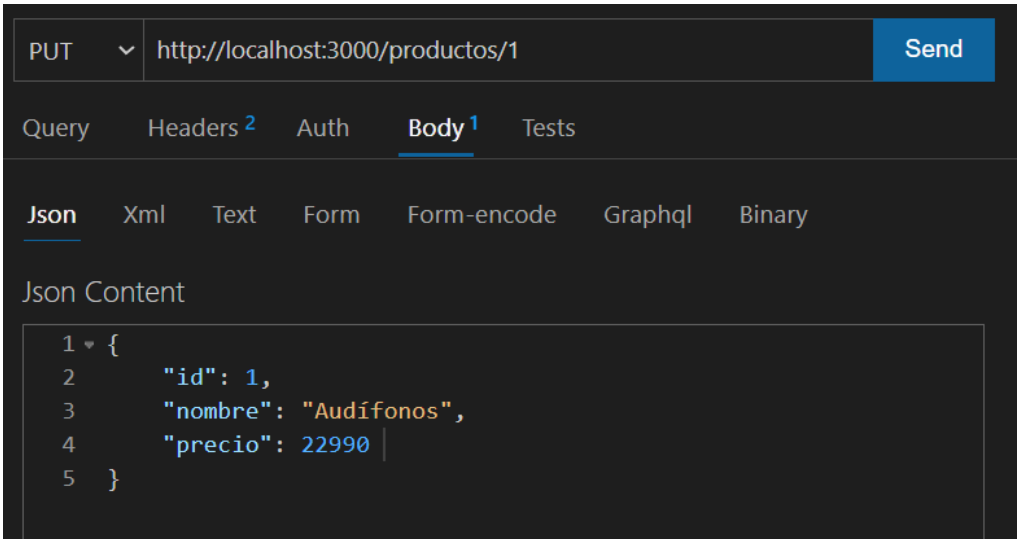


Imagen 23. Probando la ruta **PUT /productos/:id**
Fuente: Desafío Latam

Ahora si revisamos el archivo *productos.json* podremos apreciar como fue modificado específicamente el producto de id 1.

Imagen 24. Producto de id 1 modificado
Fuente: Desafío Latam

¡Manos a la obra! - PUT /usuario/:id

Continuando con el proyecto ***express a la obra:***

1. Crea una ruta **PUT /usuarios/:id**
2. Esta ruta debe recibir como parámetro el id de un usuario.
3. También debe recibir en el payload de la consulta el objeto con el usuario modificado.
4. Posteriormente, debes sobreescribir el archivo *usuarios.json* con el arreglo de usuarios modificado.
5. Prueba esta ruta realizando una consulta con Thunder Client.
6. Revisa el JSON para confirmar que el usuario fue modificado.

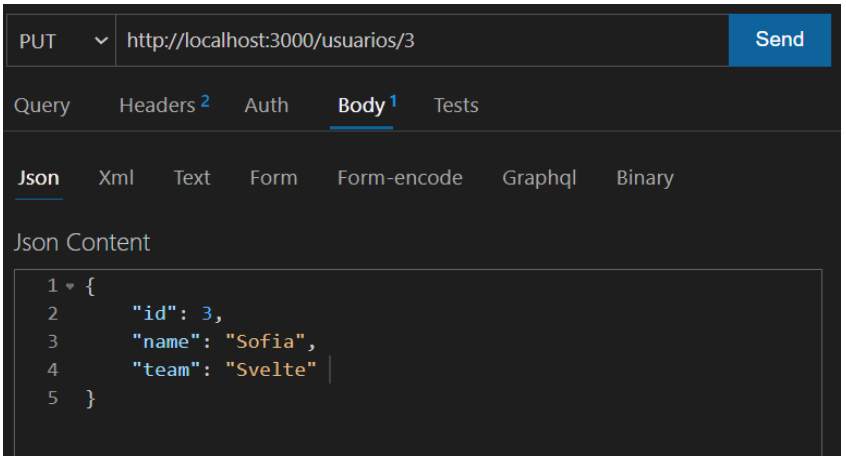


Imagen 25. Revisando el archivo JSON modificado
Fuente: Desafío Latam

Imagen 26. Resultado del ejercicio 6
Fuente: Desafío Latam

Devolviendo un HTML desde el servidor

Es posible devolver archivos desde Express a través de una ruta **GET** utilizando el método *sendFile()* del objeto *response*.

Continuando con el proyecto **"Hello World Express Js"** crea una ruta raíz **GET /** que devuelve un archivo *index.html*

```
app.get("/", (req, res) => {
    res.sendFile(__dirname + "/index.html")
})
```

El método ***sendFile()*** recibe como argumento la ruta absoluta de un archivo, por este motivo ocupamos un objeto global llamado **__dirname** que nos devuelve la dirección del script en el que se ejecuta, en este caso equivale a la dirección de la carpeta del proyecto en nuestro computador.

Ya que no tenemos aún un archivo HTML, crea manualmente un *index.html* con el siguiente código:

```
<body>
    <h1>¡Hello World Express Js!</h1>
</body>
```

Ahora probemos consultando la ruta raíz de nuestro servidor y veremos lo siguiente:

Imagen 27. Devolviendo una página web desde Express Js
Fuente: Desafío Latam

¡Muy bien! Hemos logrado devolver un sitio web desde nuestro servidor. Este HTML puede realizar ahora consultas a nuestra API REST solicitando, por ejemplo, los productos y mostrarlos en una tabla o incluso tomar datos de un usuario a través de un formulario y enviarnos la información de un nuevo usuario por medio de la ruta POST.

Entre los archivos de esta sesión encontrarás un **Apoyo Lectura - Introducción a Express** que contiene un archivo **index.html** preparado para consultar la ruta **GET /productos** que creamos al comienzo de nuestra clase y renderizar los productos en una tabla HTML.

Sustituye el *index.html* por el HTML del apoyo lectura y deberás ver lo siguiente:

Imagen 28. Productos renderizados en una tabla HTML
Fuente: Desafío Latam

Al devolver una aplicación cliente desde el mismo backend en donde tenemos el servidor y la API REST se le conoce como desarrollo de aplicaciones ***monolíticas***, sin embargo, a continuación veremos que es posible separar completamente cada uno de estos elementos y de igual manera poder comunicarse entre sí.

Live Server

Procedamos a abrir el HTML del apoyo lectura con la extensión Live Server de VSC, es decir, sin ser devuelta en una ruta de nuestro servidor.

Si no conoces esta extensión te recomendamos instalarla en la tienda de extensiones de Visual Studio Code:

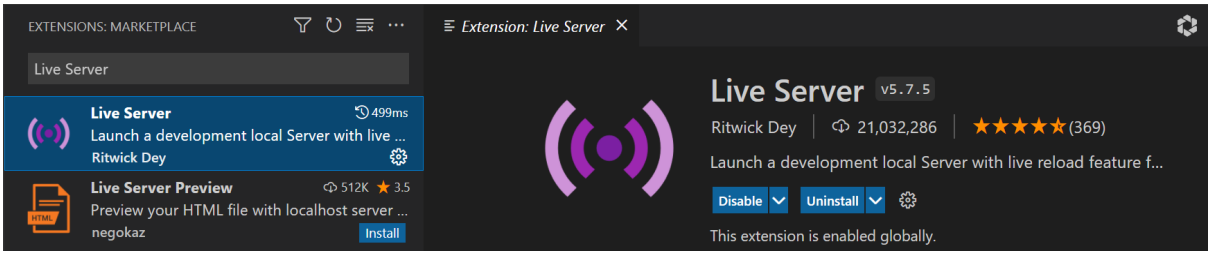


Imagen 29. Extensión Live Server en VSC
Fuente: Desafío Latam

Luego de instalarla, intenta abrir el *index.html* desde la extensión presionando clic derecho sobre el archivo y presionando "Open with Live Server" o "Abrir con Live Server"

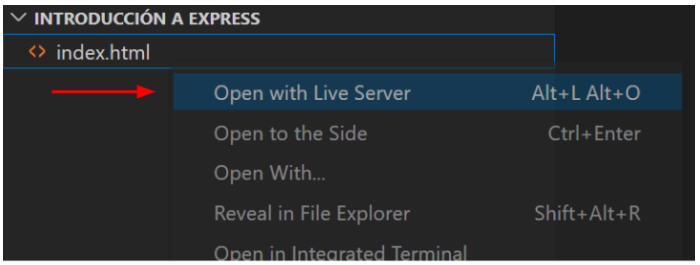


Imagen 30. Abrir un HTML con Live Server
Fuente: Desafío Latam

Esto levantará otro servidor local que nos permitirá consultar los archivos de la carpeta en el puerto 5000 a través de la URL:

http://127.0.0.1:5500 o http://localhost:5000

Sin embargo…

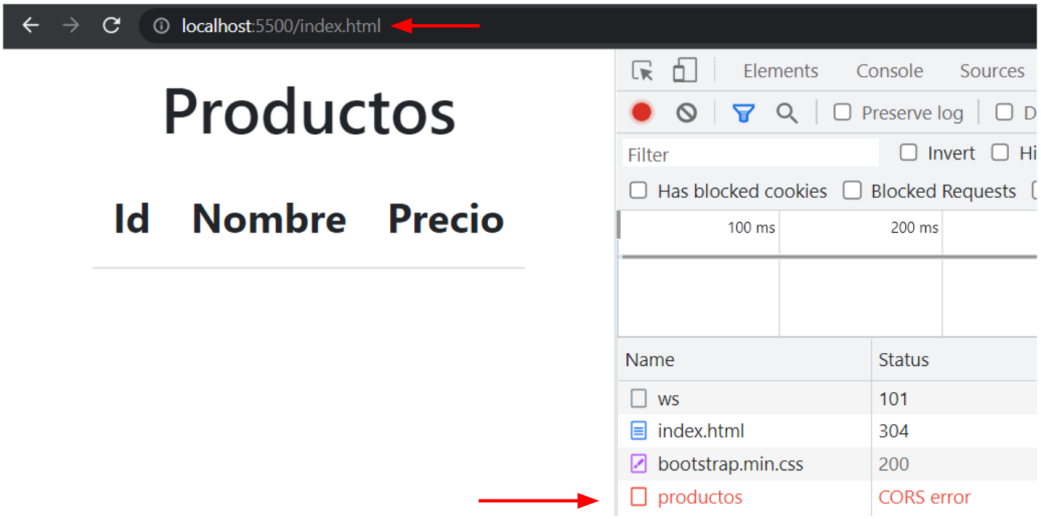


Imagen 31. Consultando ruta **GET /productos** desde otro servidor
Fuente: Desafío Latam

Si abrimos la pestaña *network* de nuestro navegador y recargamos la página, veremos que al intentar consultar la ruta **GET** http://localhost:3000/productos recibimos un error de CORS.

CORS

Los **CORS**, por sus siglas en inglés Cross Origin Request Sharing o "Intercambio de Recursos de Origen Cruzado" son una política que permite la comunicación de 2 aplicaciones de orígenes o dominios diferentes.

Los servidores que creamos con Express Js bloquean las consultas de aplicaciones externas gracias a que los CORS están deshabilitados por defecto, no obstante podemos habilitarlos instalando un paquete de NPM llamado *cors*.

```
npm install cors
```

Y posteriormente escribiendo estas 2 líneas en nuestro servidor:

```
const express = require('express')
const app = express()
const fs = require('fs')
const cors = require('cors')

app.listen(3000, console.log("¡Servidor encendido!"))

app.use(cors())

app.use(express.json())
```

```
// Rutas...
```

Ahora, si levantamos nuevamente nuestro servidor y actualizamos el navegador en la pestaña del Live Server, podremos apreciar cómo ahora sí se están consumiendo los productos sin problemas, a pesar de estar en un dominio diferente:

Imagen 32. Productos obtenidos en un dominio
Fuente: Desafío Latam

Actualizando los cambios sin reiniciar el servidor

Existe una herramienta llamada nodemon que es capaz de actualizar el servidor cada vez que hacemos un cambio, de una forma automática.

La instalación y el uso la puedes encontrar en la guía oficial https://www.npmjs.com/package/nodemon

En resumen, los pasos son instalar nodemon utilizando npm y luego ejecutar el server con nodemon index.js o el nombre del script del server.

Preguntas para una entrevista laboral.

- ¿Es Express un framework front o de back, justifique?
- ¿Qué es un middleware en Express?
- ¿Qué le falta al siguiente código para funcionar?
    - Pista: Al hacer un post a ese endpoint devuelve "Cannot read properties of undefined (reading 'id') "

```
const express = require('express')
const app = express()

app.listen(3000, console.log("¡Servidor encendido!"))

app.post("/productos", (req, res) => {
    const producto = req.body
    res.send(`id producto ${producto.id}`)
})
```

- ¿Qué significa CORS?
- ¿Cómo activamos CORS en una aplicación en Express JS?