Diplomatura en

Programación FullStack Developer

NodeJS
Conexión Backend-Frontend con Express

Backend vs. Frontend

1. Backend:

Parte de la aplicación que no interactúa directamente con el usuario. Incluye el servidor, la aplicación y la base de datos. Se encarga de gestionar la lógica de negocio, operaciones de datos y la ejecución de las funciones de la aplicación.

Funciones Clave:

- Procesar la lógica de negocio.
- Interactuar con bases de datos.
- Autenticación y autorización de usuarios.
- Servir datos al frontend en forma de API.

2. Frontend:

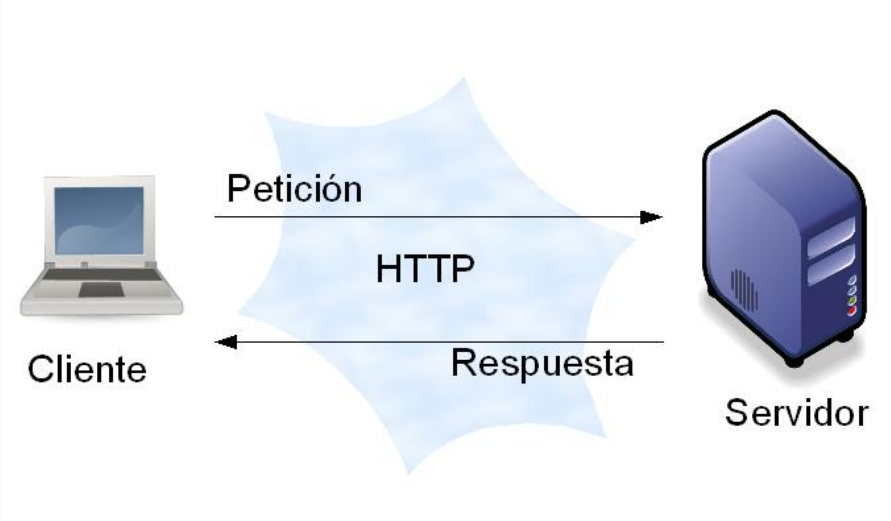
Parte de la aplicación que los usuarios interactúan directamente, usualmente a través de un navegador web. Incluye todo lo que el usuario puede ver y usar, como interfaces gráficas y diseño.

Funciones Clave:

- Presentar información y datos de forma visual.
- Permitir la interacción del usuario con la aplicación.
- Consumir datos proporcionados por el backend y mostrarlos adecuadamente.

Comunicación entre Cliente y Servidor:

Cómo Funciona: El frontend (cliente) envía una solicitud al backend (servidor), usualmente a través de HTTP/HTTPS. El servidor procesa la solicitud, interactúa con la base de datos si es necesario, y envía una respuesta de vuelta al cliente.



Conexión back-front

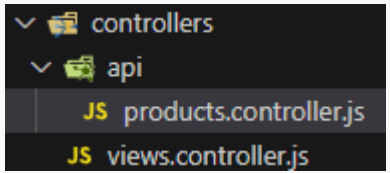
Hasta ahora, la aplicación en Express se centró en renderizar plantillas para crear interfaces de usuario dinámicas. Vamos a transformar esta aplicación en una API robusta que podrá ser consumida por clientes externos, como aplicaciones front-end o de celulares.

1. Definir Rutas para la API

Inicialmente, nuestra aplicación utilizaba métodos como res.render() para enviar HTML generado al cliente. Ahora, modificaremos el enfoque para que, en lugar de renderizar vistas, nuestra aplicación sirva datos en formato JSON.

Vamos a crear un endpoint que devuelva ciertos productos

Definimos el controller en controllers/api/

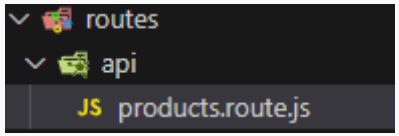


products.controller.js

```
export const getProducts = (req, res) => {
    const productos = [
        {
            id: 1,
            nombre: "Cafetera",
            descripción: "Cafetera automática con sistema de
filtro avanzado y capacidad de 1.5 litros.",
            precio: 85.50
        },
        {
            id: 2,
            nombre: "Licuadora",
            descripción: "Licuadora de 800 watts con jarra de
vidrio resistente y 5 velocidades.",
            precio: 45.75
        },
        {
            id: 3,
            nombre: "Tostadora",
            descripción: "Tostadora de acero inoxidable con
capacidad para 4 rebanadas y 7 niveles de tostado.",
            precio: 30.00
        }
    ];

    return res.json({
        success: true,
        payload: productos
    })
}
```

También definimos en routes/api/



products.route.js

```
import { Router } from "express";
import { getProducts } from "../../controllers/api/products.controller.js";

const router = new Router();

router.get('/', getProducts)

export default router
```

index.js

```
import viewRouter from './routes/views.route.js'
import productRouter from './routes/api/products.route.js'

const app = express();
```

```
// Routes
app.use('/', viewRouter)
app.use('/api/products', productRouter)
```

2. Configurar cors

Para que la API pueda ser accedida desde dominios diferentes al del servidor, es crucial implementar CORS (Cross-Origin Resource Sharing). Esto es especialmente importante en entornos de desarrollo donde el cliente y el servidor pueden no compartir el mismo origen.

Si el frontend y backend están en diferentes dominios durante el desarrollo, necesitaremos configurar CORS para permitir que el frontend acceda al backend. Instalaremos el paquete cors.

npm install cors

index.js

```
import cors from 'cors’;

const app = express();
app.use(cors());
```

3. Consumir la api

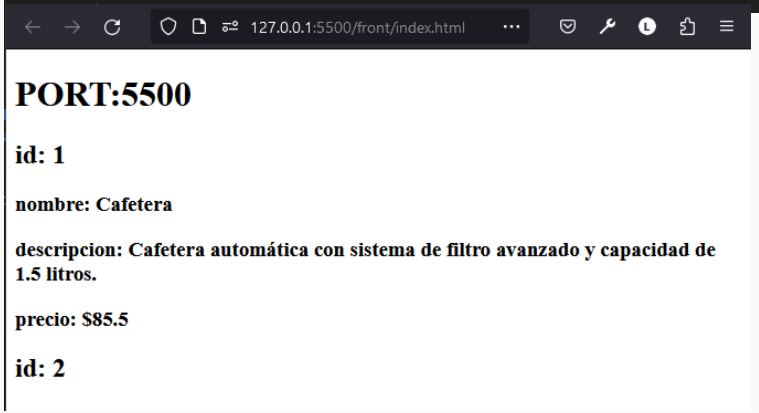
Desde vanilla JS

```html
<body>
    <h1>PORT:5500</h1>
    <div class="container"></div>
    <script src="main.js"></script>
</body>
```

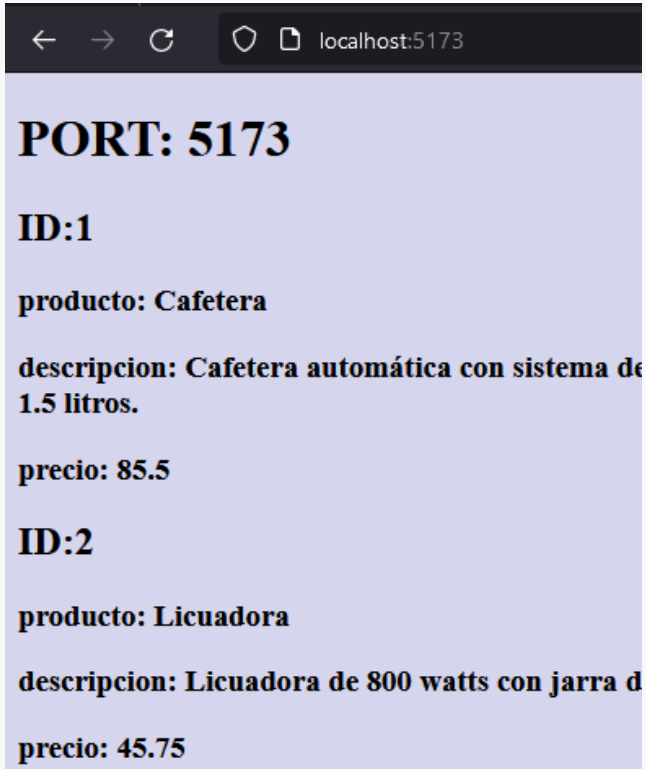
```
front
  index.html
  main.js
```

```js
const container = document.querySelector('.container')

fetch('http://localhost:3000/api/products')
    .then(response => response.json())
    .then(data => {
        data.payload.forEach(element => {
            console.log(element)
            const div = document.createElement('div');
            div.innerHTML = `<h2>id: ${element.id}</h2>
                             <h3>nombre: ${element.nombre}</h3>
                             <h3>descripcion: ${element.descripcion}</h3>
                             <h3>precio: $${element.precio}</h3>
            `
            container.append(div)
        });
    })
```



Desde React JS




```jsx
import { useEffect, useState } from "react"

function App() {
  const [products, setProducts] = useState([])

  useEffect(() => {
    fetch('http://localhost:3000/api/products')
      .then(response => response.json())
      .then(data => setProducts(data.payload))
  }, [])

  return (
    <>
      <h1>PORT: 5173</h1>
      {products.map(product =>
        <div key={product.id}>
          <h2>ID:{product.id}</h2>
          <h3>producto: {product.nombre}</h3>
          <h3>descripcion: {product.descripcion}</h3>
          <h3>precio: {product.precio}</h3>
        </div>
      )}
    </>
  )
}

export default App
```

params y query params en express

1. Params (Parámetros de Ruta):
Los **params** se refieren a los parámetros definidos en la ruta. Estos parámetros forman parte de la URL y son útiles cuando queremos identificar un recurso específico, como un id. Por ejemplo, si tenemos una URL como /users/:id, el parámetro :id es un parámetro de ruta.

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

app.get('/users/:id', (req, res) => {
    const userId = req.params.id; // Captura el valor del parámetro de la ruta
    res.send(`El ID del usuario es ${userId}`);
});

app.listen(3000, () => {
    console.log('Servidor escuchando en el puerto 3000');
});
```

En este ejemplo, si hacemos una solicitud a /users/123, el valor de req.params.id será 123.

Query Params (Parámetros de Consulta):
Los **query params** son parte de la URL después del signo de interrogación (?) y se utilizan para enviar datos opcionales en la solicitud. Podemos tener varios parámetros separados por &. Los parámetros de consulta son útiles cuando no forman parte del recurso principal, pero se usan para filtrar o modificar el resultado.

```
app.get('/search', (req, res) => {
    const searchTerm = req.query.q; // Captura el valor del parámetro de consulta 'q'
    res.send(`Buscando por: ${searchTerm}`);
});
```

En este ejemplo, si hacemos una solicitud a /search?q=javascript, el valor de req.query.q será javascript.

Diferencias Clave:
•**Params**: Forman parte de la estructura de la URL (normalmente usados para identificar recursos específicos).
•**Query Params**: Se añaden al final de la URL y son útiles para enviar datos opcionales o modificar los resultados.

Vamos a estructurar un proyecto modular en Express.js con un enfoque en JavaScript con módulos, simulando datos con un array. Tendremos una arquitectura con controladores, rutas, y un entrypoint, todo manejado con módulos ES (utilizando type: module en package.json).

```
/**
 *
my-modular-project
├── controllers
│   └── usersController.js
├── routes
│   └── usersRoutes.js
├── data
│   └── usersData.js
├── index.js
└── package.json

 */
```

1. package.json

Asegurémonos de que el proyecto está configurado para trabajar con módulos ES.
Agregamos el campo "type": "module" en el package.json:

```
{
  "name": "my-modular-project",
  "version": "1.0.0",
  "description": "Modular project with simulated data",
  "type": "module",  // Esto activa el uso de módulos ES
  "main": "index.js",
  "dependencies": {
    "express": "^4.18.1"
  },
  "scripts": {
    "start": "node index.js"
  }
}
```

Simulación de Datos: data/usersData.js

Creamos un archivo para simular los datos. En este caso, un array de usuarios:

```
export const users = [
    { id: 1, name: 'Juan Pérez', email: 'juan.perez@mail.com' },
    { id: 2, name: 'María López', email: 'maria.lopez@mail.com' },
    { id: 3, name: 'Carlos García', email: 'carlos.garcia@mail.com' }
];
```

Controlador: controllers/usersController.js

El controlador manejará la lógica de negocio, en este caso, devolverá la lista de usuarios y uno en específico:

```
import { users } from '../data/usersData.js';

export const getAllUsers = (req, res) => {
  res.json(users);
};

export const getUserById = (req, res) => {
  const userId = parseInt(req.params.id);
  const user = users.find(u => u.id === userId);
  if (user) {
    res.json(user);
  } else {
    res.status(404).json({ message: 'Usuario no encontrado' });
  }
};
```

Rutas: routes/usersRoutes.js

Este archivo define las rutas que manejarán las solicitudes y conectarán con los controladores:

```
import express from 'express';
import { getAllUsers, getUserById } from
'../controllers/usersController.js';

const router = express.Router();

router.get('/', getAllUsers);  // Obtener todos los usuarios
router.get('/:id', getUserById);  // Obtener un usuario por ID

export default router;
```

Punto de Entrada: index.js

Finalmente, el archivo principal (entrypoint) configurará Express y montará las rutas:

```
import express from 'express';
import usersRoutes from './routes/usersRoutes.js';

const app = express();
const PORT = 3000;

app.use(express.json());

// Rutas de usuarios
app.use('/users', usersRoutes);

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

Instalación y Ejecución

1.Asegurate de tener Node.js instalado.

2.Corre npm install para instalar las dependencias.

3.Ejecuta el servidor con npm start.

Esto te dará un API básico con las siguientes rutas:
•**GET** /users → Devuelve todos los usuarios.
•**GET** /users/:id → Devuelve un usuario por ID.
Todo está modularizado, trabajando con módulos ES (type: module), y simulando los datos con un array.

Para agregar **query params** a nuestro proyecto, podemos modificar el controlador para que soporte filtrado o búsqueda usando los parámetros de consulta. Vamos a hacer un pequeño ajuste para que, por ejemplo, puedas filtrar usuarios por nombre o email usando query params.

Modificando el Controlador: controllers/usersController.js

Vamos a actualizar el controlador para manejar query params en la ruta /users. El objetivo será filtrar por nombre o email si los parámetros de consulta están presentes.

```
import { users } from '../data/usersData.js';

export const getAllUsers = (req, res) => {
  const { name, email } = req.query;

  let filteredUsers = users;

  if (name) {
    filteredUsers = filteredUsers.filter(user =>
      user.name.toLowerCase().includes(name.toLowerCase())
    );
  }

  if (email) {
    filteredUsers = filteredUsers.filter(user =>
      user.email.toLowerCase().includes(email.toLowerCase())
    );
  }

  res.json(filteredUsers);
};

export const getUserById = (req, res) => {
  const userId = parseInt(req.params.id);
  const user = users.find(u => u.id === userId);
  if (user) {
    res.json(user);
  } else {
    res.status(404).send({ message: 'Usuario no encontrado' });
  }
};
```

2. Uso de Query Params

Ahora puedes hacer solicitudes GET a /users con los siguientes query params:

•**Filtrar por nombre**: /users?name=Juan

•**Filtrar por email**: /users?email=juan.perez@mail.com

•**Filtrar por ambos**: /users?name=Juan&email=juan.perez@mail.com

3. Rutas: routes/usersRoutes.js

No es necesario hacer cambios en las rutas, ya que las query params se manejan directamente en el controlador.

4. Ejemplo de Solicitudes

1.GET /users?name=Juan

- Respuesta: Filtra usuarios cuyo nombre contenga "Juan".

2.GET /users?email=maria.lopez@mail.com

- Respuesta: Filtra usuarios cuyo email sea "maria.lopez@mail.com".

3.GET /users?name=Carlos&email=carlos.garcia@mail.com

- Respuesta: Filtra usuarios cuyo nombre contenga "Carlos" y cuyo email sea "carlos.garcia@mail.com".

Flujo Completo

Esto permitirá que cualquier solicitud a la ruta /users reciba datos filtrados según los parámetros de consulta. Si no se pasa ningún query param, devolverá todos los usuarios.