Antes de arrancar, un repaso.

Agencia de Aprendizaje a lo largo de la vida

Buenos Aires Ciudad
Educación

Frontend

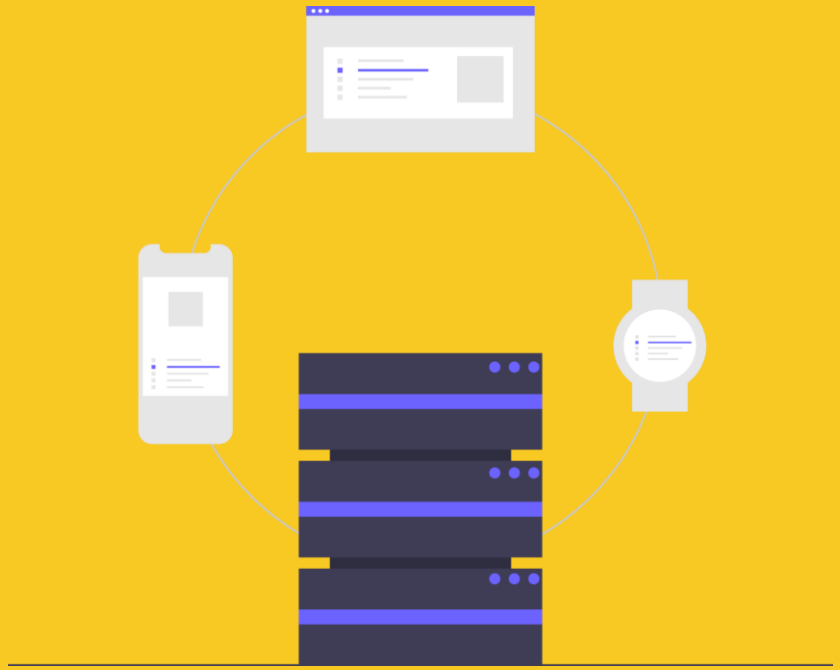
Es todo lo que pasa del lado del cliente (en el navegador).

Está compuesto por archivos estáticos en los lenguajes que el navegador puede interpretar. Estos lenguajes son **HTML** para la estructura de información, **CSS** para los estilos y **JavaScript** para la interacción dentro nuestro sitio web.

El desarrollador de **front-end** se encarga de implementar todo lo relacionado con la parte visible, lo que “toca" el usuario cuando navega por la web.

BACKEND

Introducción



Backend

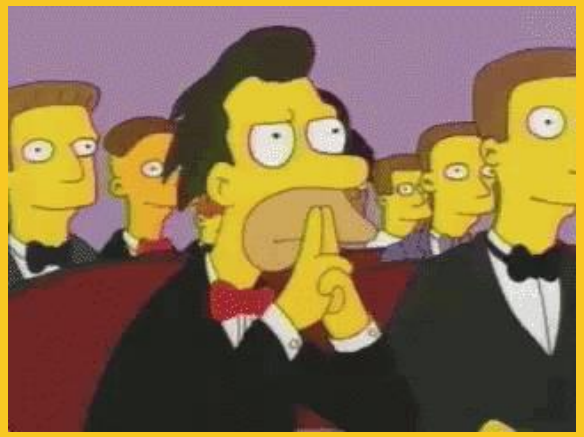
Es todo lo que pasa del lado del servidor.

El backend es la parte del desarrollo web que se encarga de que toda la lógica de una página web funcione. Se trata del conjunto de acciones que pasan en una web pero que no vemos como, por ejemplo, la consulta de datos a una BBDD y posterior envío al cliente.

Algunas de las funciones que se gestionan en la parte del back-end son:

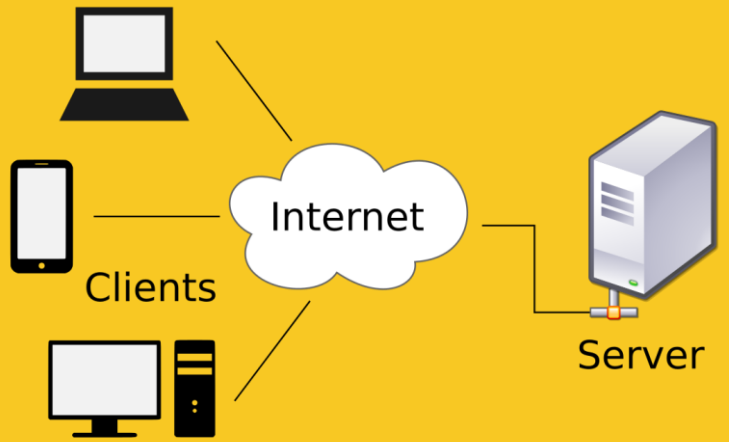
- El desarrollo de funciones que simplifiquen el proceso de desarrollo.
- Acciones de lógica.
- Conexión con bases de datos.
- Uso de librerías del servidor web (por ejemplo para implementar temas de caché o para comprimir las imágenes de la web).

Pero... ¿cómo se comunican?



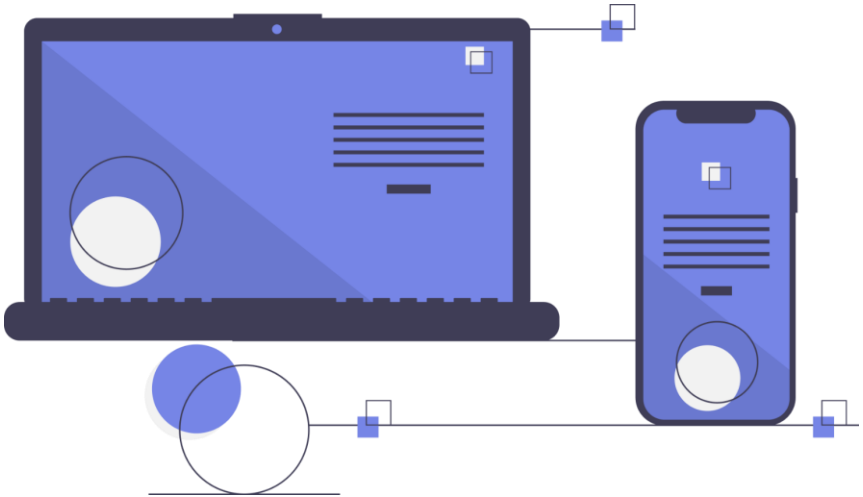
Flujo Cliente/Servidor

Es una relación en la cual un programa (el cliente) solicita un servicio o recurso de otro programa (el servidor). Este esquema refleja el flujo de información entre uno o varios dispositivos con un servidor web.



Cliente

- Envía una petición al servidor y se queda esperando por una respuesta.
- Una vez que son servidas sus solicitudes, termina el trabajo.
- Un cliente accede a un servidor y recupera servicios especiales o datos de él.



Es tarea del cliente estandarizar las solicitudes, transmitirlas al servidor y procesar los datos obtenidos para que puedan visualizarse en un dispositivo de salida como una pantalla.

Servidor

Es un programa que ofrece un servicio que se puede obtener en una red.

- Acepta la petición desde la red, realiza el servicio y devuelve el resultado al solicitante.
- El servidor comienza su ejecución antes de comenzar la interacción con el cliente.
- Su tiempo de vida o de interacción es “interminable”, una vez comienza a correr, se queda esperando las solicitudes que pudieran llegar desde los diversos clientes.

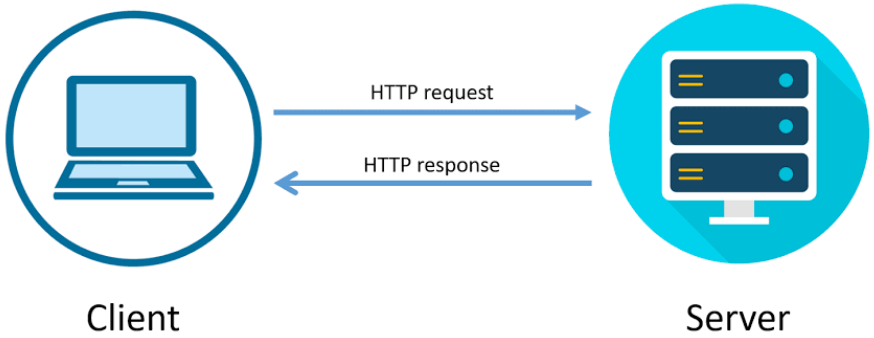
Request/Response

REQUEST

Son las solicitudes o peticiones que hacemos a través del navegador (el cliente) a un servidor.

RESPONSE

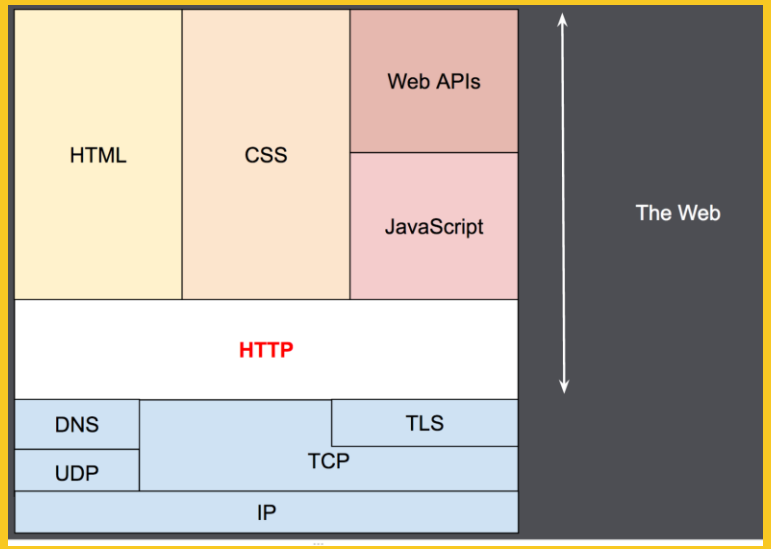
El servidor recibe nuestra solicitud, la procesa y envía como resultado una respuesta al cliente (navegador).



Protocolo HTTP

HTTP o Hyper Text Transfer Protocol es el nombre de un protocolo que nos permite realizar una petición de datos y recursos, como pueden ser documentos HTML.

Es la base de cualquier intercambio de datos en la web para la comunicación de un cliente con un servidor.



Manejo de información

HTTP es un protocolo sin estado, por lo que no guarda ninguna información sobre conexiones anteriores.

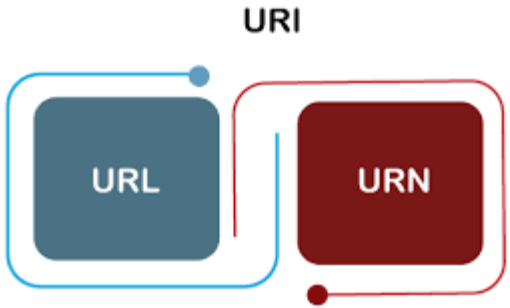
El desarrollo de aplicaciones web necesita frecuentemente mantener estado, por ejemplo, para el uso de "CARRITOS de compra" en páginas de comercio electrónico. Para esto se usan las cookies, que es información que un servidor puede almacenar en el sistema cliente.

IDENTIFICACIÓN DE RECURSOS WEB

El objetivo de una solicitud HTTP se denomina "recurso", (es decir: datos) y dicho recurso, no posee un tipo definido por defecto; puede ser un documento, o una foto, o cualquier otra posibilidad.

Cada recurso es identificado por un Identificador Uniforme de Recursos (URI) y es utilizado a través de HTTP, para la identificación del tipo de recurso.

Una URI (identificador de recursos uniformes) es un bloque de texto que se escribe en la barra de direcciones de un navegador web y está compuesto por dos partes: la URL y la URN.



URI: URL + URN

Un **URI** consta de un máximo de cinco partes, de las cuales solo dos son obligatorias:

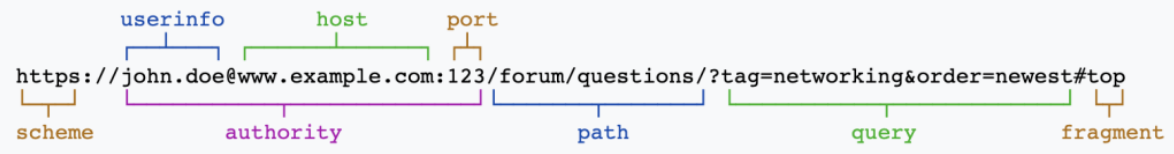
- scheme (esquema): proporciona información sobre el protocolo utilizado.
- authority (autoridad): identifica el dominio.
- path (ruta): muestra la ruta exacta al recurso.
- query (consulta): representa la acción de consulta.
- fragment (fragmento): designa una parte del recurso principal.

URL

Uniform resource locator se utiliza para indicar dónde se encuentra un recurso. Por lo tanto, también sirve para acceder a algunas páginas web por Internet.

URN

Uniform resource name es independiente de la ubicación y designa un recurso de forma permanente.



FASES HTTP

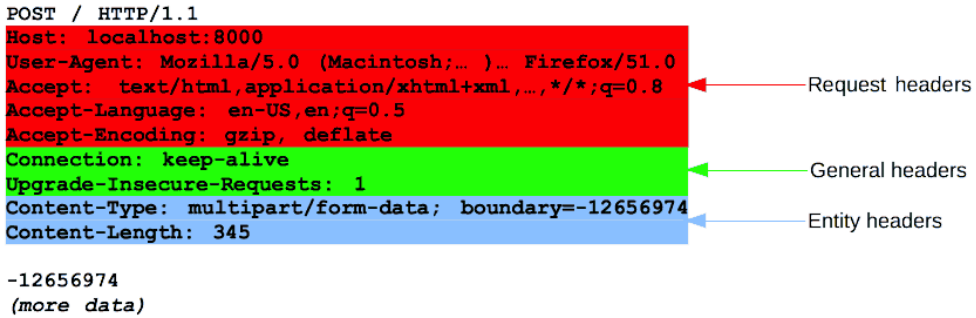
En los protocolos basados en el modelo cliente-servidor, como es el caso del HTTP, una sesión consta de tres fases:

- El cliente establece una conexión TCP.
- El cliente manda su petición, y espera por la respuesta.
- El servidor procesa la petición, y responde con un código de estado y los datos correspondientes.

HTTP HEADERS

Las cabeceras (en inglés headers) HTTP permiten al cliente y al servidor enviar información adicional junto a una petición o respuesta.

- Headers generales, (General headers), como Via (en-US), afectan al mensaje como una unidad completa.
- Headers de petición, (Request headers), como User-Agent, Accept-Type, modifican la petición especificando con mayor detalle (como: Accept-Language (en-US), o dándole un contexto, como: Referer, o restringiéndola condicionalmente, como: If-None.
- Headers de entidad, (Entity headers'), cómo Content-Length las cuales se aplican al cuerpo de la petición. Por supuesto, esta cabecera no necesita ser transmitida si el mensaje no tiene cuerpo ('body' en inglés).

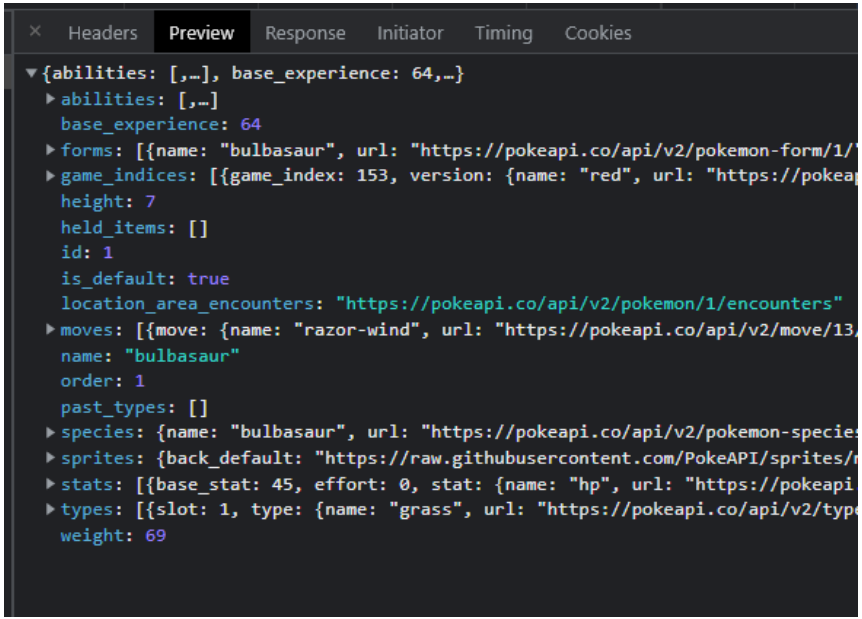


HTTP BODY

La última parte del mensaje de respuesta es el body.

No siempre se envía o se recibe uno y se utiliza para enviar información desde un cliente (navegador) o para recibirla desde el servidor.

Para ello se utiliza el METHOD HTTP conocido como POST que manipula la información en el cuerpo del mensaje a diferencia del método GET que lo hace directamente en la URL a través de los query params.



HTTP METHODS

HTTP define un conjunto de métodos de petición para indicar la acción que se desea realizar para un recurso determinado. Estos métodos son utilizados principalmente en la creación de servicios REST.

GET	Leer información
PUT	Reemplazar información
POST	Enviar/Solicitar/Crear información
PATCH	Modificar/Actualizar información
DELETE	Eliminar información

HTTP STATUS CODES

Los códigos de estado de respuesta HTTP indican si se ha completado satisfactoriamente una solicitud HTTP específica. Las respuestas se agrupan en cinco clases:

100	Respuestas informativas (100–199)
200	Respuestas satisfactorias (200–299)
300	Redirecciones (300–399)
400	Errores de los clientes (400–499)
500	Errores de los servidores (500–599)

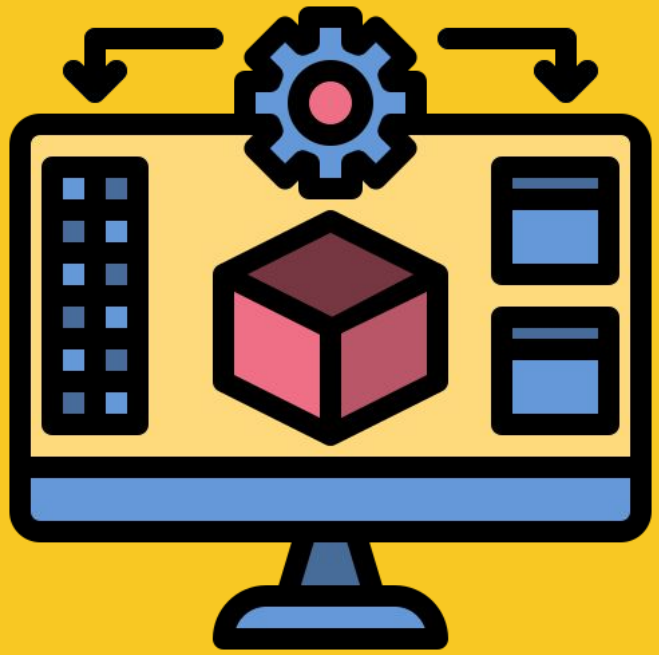
HTTP STATUS CODES

Status Code HTTP más comunes:

200	La solicitud ha tenido éxito.
302	El recurso de la URI solicitada ha sido cambiado temporalmente
400	El servidor no pudo interpretar la solicitud dada una sintaxis inválida.
401	Es necesario autenticar para obtener la respuesta solicitada.
403	El cliente no posee los permisos necesarios para cierto contenido
404	El servidor no pudo encontrar el contenido solicitado.
500	El servidor ha encontrado una situación que no sabe cómo manejarla.
503	El servidor está caído por mantenimiento o está sobrecargado

ARQUITECTURA

Patrones



¿Qué es un Patrón de Arquitectura?

Un patrón de arquitectura es una solución probada y documentada a un problema recurrente en el desarrollo de software.

Estos patrones son utilizados para resolver problemas comunes de diseño y permiten a los desarrolladores construir software escalable y robusto.

Tipos de Patrones

Algunos de ellos son:

Capas

Dividen el software en **capas**, cada una con una responsabilidad específica. Ejemplos de patrones de capas son: **MVC** (Modelo-Vista-Controlador) y **MVP** (Modelo-Vista-Presentador).

Basados en eventos

Se centran en el intercambio de mensajes o eventos entre componentes. Ejemplos de patrones de eventos son: **Publicar-Suscribir**, **Observer** y **Reactor**.

Basados en servicios:

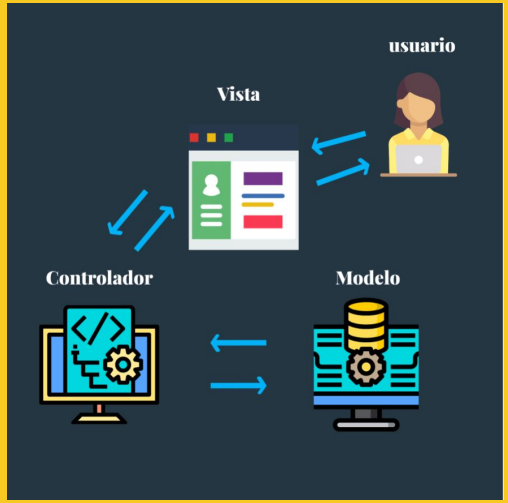
Se enfocan en la creación de servicios reutilizables. Por ejemplo: Arquitectura **SOA** (Orientada a Servicios) y **REST** (Representational State Transfer).

Basados en microservicios

Se enfocan en la creación de una arquitectura compuesta por servicios independientes que trabajan juntos para realizar una tarea. Algunos ejemplos son: **Microservicios** y **Arquitectura Hexagonal**.

¿Qué es MVC?

Es una arquitectura de software que propone la división de responsabilidades de una aplicación en 3 capas diferentes.



Principios

MVC es útil en sistemas donde se requiere el uso de interfaces de usuario, aunque en la práctica el mismo patrón de arquitectura **se puede utilizar para distintos tipos de aplicaciones**.

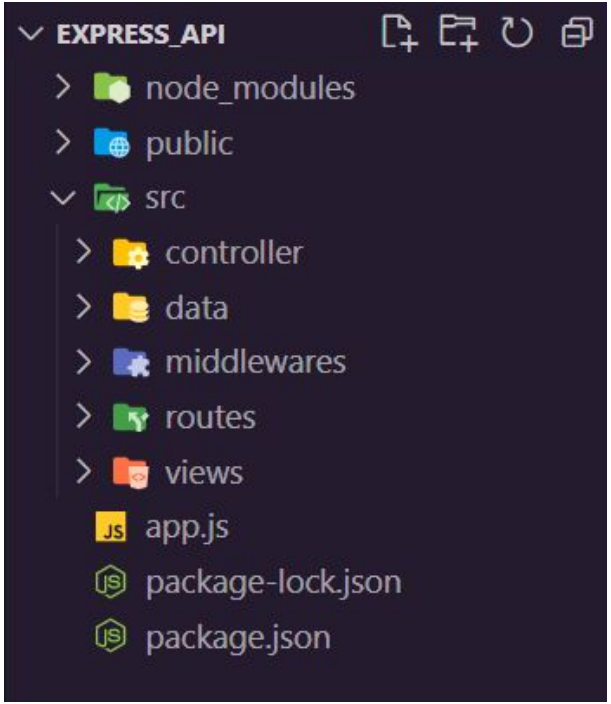
Ayuda a **crear softwares más robustos**, donde se potencie la facilidad de mantenimiento, **reutilización del código** y la separación de conceptos.

Estructura MVC

Una estructura **MVC** básica está compuesta por al menos 3 carpetas y un entry point.

En la imagen **podemos observar** el archivo **app.js** cómo entry point y las carpetas **data** (**m**odelo), **views** (**v**ista), **controller** (**c**ontrolador).

Además *se pueden tener otras carpetas* para las rutas, estilos y middlewares que **veremos más adelante**.



Modelo

Capa que trabaja con los datos.

Contiene mecanismos para acceder a la información y también para actualizar su estado.

En los modelos tendremos todas las funciones que accederán a las tablas y harán los correspondientes selects, updates, inserts, etc.

En **Node** se suele utilizar ***Sequelize***, un programa ORM que facilita el trabajo con BBDD relacionales.

Vista

Es la capa visible de nuestra aplicación.

Es el código ***HTML***, ***CSS***, ***Javascript*** necesario para renderizar y mostrar los datos e información a nuestros usuarios.

En ocasiones se suelen utilizar ***Template Engines*** o diversos frameworks como ***React*** para adoptar flexibilidad en el desarrollo.

Normalmente es la capa Front End de los proyectos y su comunicación hacia las fuentes de datos se realiza a través de los ***controladores***.

Controlador

Capa que sirve de enlace entre las vistas y los modelos

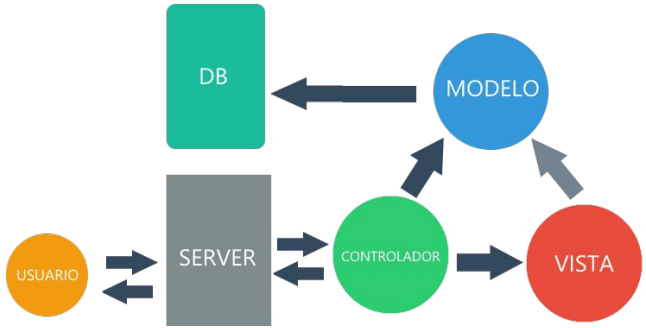
Su responsabilidad no es manipular directamente datos, ni mostrar ningún tipo de salida.

Normalmente contienen la lógica de nuestra aplicación junto con las condiciones o reglas de negocio.

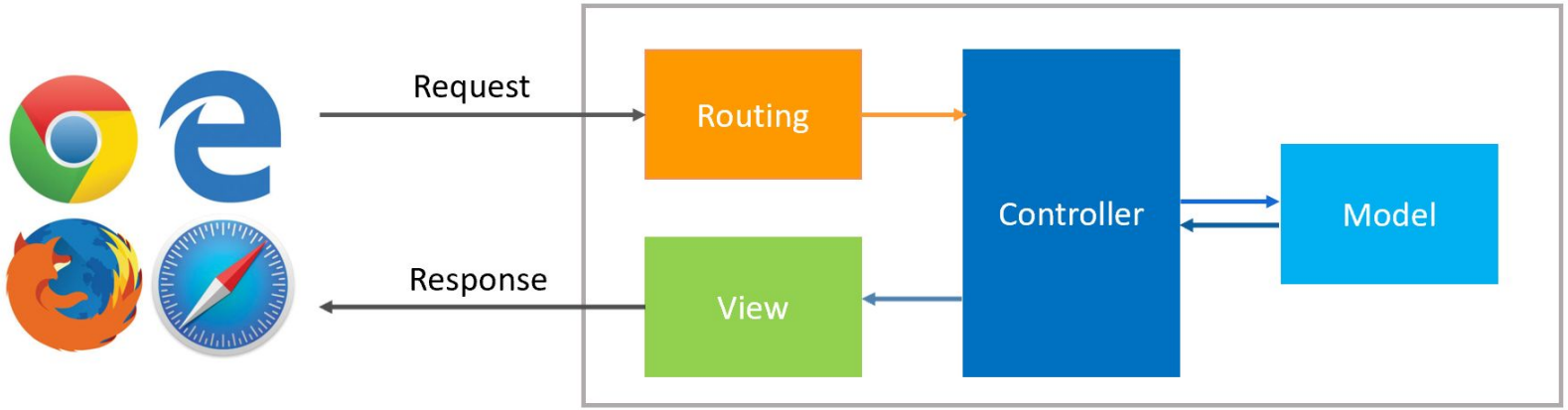
Es invocado por nuestras rutas, solicita datos al modelo y los envía a la vista para ser renderizados.

Recorrido MVC

- El usuario realiza una solicitud a nuestro servidor.
- El router invoca un controlador.
- El controlador solicita información al modelo y este a la base de datos, devuelve al controlador y retorna los datos a la vista.
- La vista crea un archivo estático y se envía al cliente.
- El cliente recibe los archivos y renderiza la aplicación.



Otro ejemplo...



¿Qué es una API Rest?

Las **API** interactúan con sistemas o PC 's de manera que el sistema **comprenda la solicitud** y la cumpla.

REST no es un protocolo ni un estándar, sino un conjunto de límites de arquitectura.

La información **se entrega por medio de HTTP** en uno de estos formatos: **JSON** (JavaScript Object Notation), ***HTML, XLT, Python, PHP o texto sin formato***.

Principios REST

REST se enfoca en exponer recursos a través de URLs y utilizar los verbos **HTTP** (**GET**, **POST**, **PUT**, **DELETE**) para manipularlos.

Se basa en la utilización de los **verbos HTTP** para realizar operaciones sobre los recursos, y en la utilización de los formatos **JSON** o **XML** para representar la información.

Además propone un conjunto de restricciones arquitectónicas, como la interfaz uniforme, el estado sin sesión, la cacheabilidad, la visibilidad y la escalabilidad, que permiten construir sistemas web flexibles y escalables.

¿Qué necesito para crear una API Rest?

Definir los recursos:

Identificar los recursos que se van a exponer en la API RESTful, como entidades de negocio o funciones específicas.

Utilizar los códigos de estado HTTP

Utilizar los códigos de estado HTTP para comunicar el resultado de la operación, como 200 para una solicitud exitosa o 404 para un recurso no encontrado.

Definir la estructura de la URL

Utilizar URLs descriptivas para cada recurso, evitando utilizar verbos o adjetivos en la URL y separando los elementos con barras diagonales.

Utilizar el formato de datos correcto

Utilizar el formato de datos adecuado para cada operación, como XML o JSON, y especificar el tipo de contenido en la cabecera HTTP.

Utilizar los verbos HTTP

GET, POST, PUT, DELETE, etc. En base a la intención para manipular los recursos. Cómo usar GET para obtener un recurso y POST para crear uno nuevo.

Utilizar la documentación

Documentar la API RESTful para que los consumidores puedan entender cómo utilizarla y qué recursos están disponibles.

Utilizar la autenticación y la autorización

Proteger la API RESTful mediante la autenticación y la autorización de los usuarios que acceden a los recursos.

MVC vs REST

Característica	MVC	REST
Enfoque	Separación de preocupaciones y organización en tres componentes claramente definidos: **Modelo**, **Vista** y **Controlador**.	Exposición de recursos y utilización de verbos **HTTP** para manipularlos.
Tipo de aplicación	Principalmente aplicaciones web, aunque también puede utilizarse en otros tipos de aplicaciones.	
Manejo de solicitudes	A través del Controlador, que actúa como intermediario entre la Vista y el Modelo.	A través de los verbos HTTP: GET, POST, PUT y DELETE.
Representación de la información	Utiliza un conjunto de estructuras de datos para representar los datos en la aplicación.	Utiliza formatos como JSON o XML para representar los datos.
Escalabilidad	Escalabilidad limitada debido a la organización de la aplicación en tres componentes.	Altamente escalable debido a la utilización de los verbos HTTP y la exposición de recursos.
Reutilización de código	Permite la reutilización de código a través de la separación clara de las responsabilidades.	Permite la reutilización de código a través de la exposición de recursos y la utilización de los verbos HTTP.

Si bien ambos patrones tienen sus principios y diferencias, es posible utilizarlos juntos.

NODE JS

Introducción

Agencia de Aprendizaje a lo largo de la vida

Buenos Aires Ciudad
Educación

¿Qué es Node JS?

Es un entorno de ejecución para JavaScript orientado a eventos asíncronos diseñado para **crear aplicaciones web escalables**, construido con V8, el motor de JavaScript de Chrome, escrito en C, C++ y JavaScript.

Nos permite **desarrollar con el lenguaje Javascript** más allá del navegador.

¿Es NODE JS un lenguaje de programación?

En una palabra: ***NO***.

NODE es un entorno de ejecución que se utiliza para ejecutar JavaScript **fuera** del navegador.

Tampoco es un framework

El tiempo de ejecución de ***NODE*** se construye sobre un lenguaje de programación -en este caso, **JavaScript**- y ayuda a la **ejecución de los propios frameworks**.

En resumen, ***NODE*** no es un lenguaje de programación ni un marco de trabajo, es un entorno para ellos.

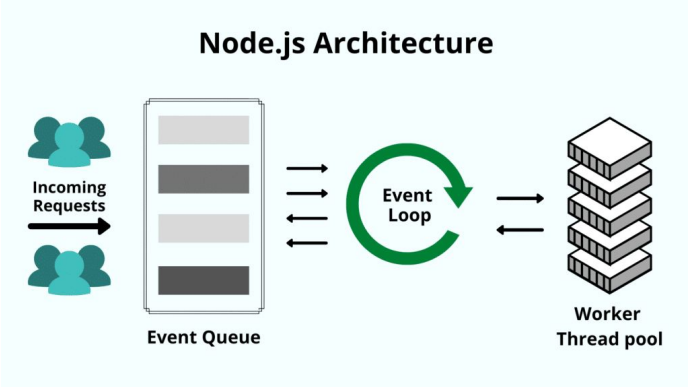
Arquitectura de Node

NODE utiliza la arquitectura *"Single Threaded Event Loop"* para manejar múltiples clientes al mismo tiempo, a diferencia de los clientes concurrentes multihilo en lenguajes como **Java**.

Single Thread VS Multi Thread

SINGLE THREAD

1. Mantiene un pool de hilos limitado para atender las peticiones.
2. Cada vez que llega una solicitud, la coloca en una cola.
3. El Event Loop espera las peticiones indefinidamente.
4. Cuando llega una solicitud, el bucle la recoge de la cola y comprueba si requiere una operación de entrada/salida (E/S) de bloqueo. Si no es así, procesa la solicitud y envía una respuesta.
5. Si la solicitud tiene una operación de bloqueo que realizar, el bucle de eventos asigna un hilo del pool de hilos internos para procesar la solicitud. Los hilos internos disponibles son limitados.
6. El Event Loop rastrea las solicitudes que se bloquean y las coloca en la cola una vez que se procesa la tarea que se bloquea. Así es como mantiene su naturaleza no bloqueante.

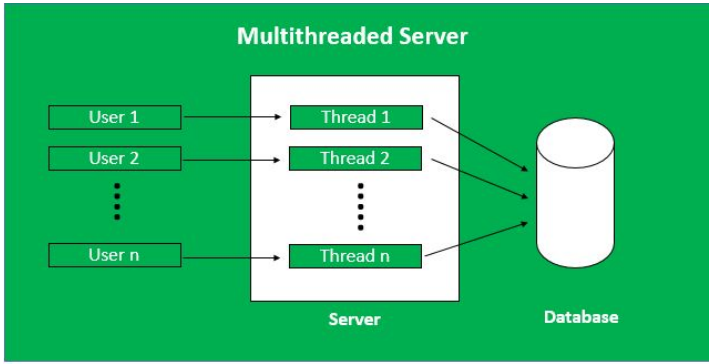


Single Thread VS Multi Thread

MULTITHREAD

En un **modelo de solicitud-respuesta multihilo**, varios clientes envían una solicitud y el servidor procesa cada una de ellas antes de devolver la respuesta.

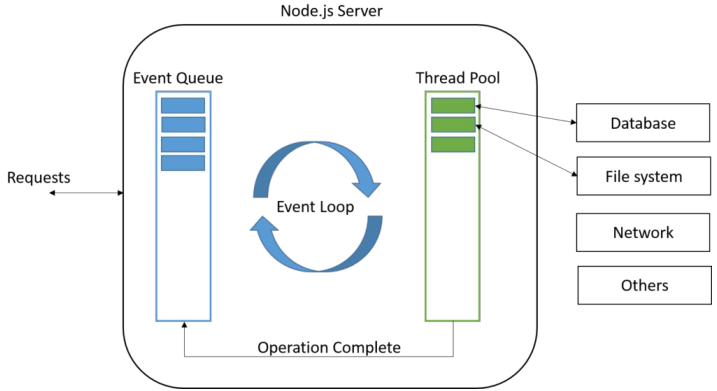
Sin embargo, se utilizan múltiples hilos para procesar las **llamadas concurrentes**. Estos hilos se definen en un pool de hilos, y cada vez que llega una petición, **se asigna un hilo individual** para manejarla.



Arquitectura Node JS

Dado que ***NODE*** utiliza menos hilos, utiliza menos recursos/memoria, lo que resulta en una ejecución más rápida de las tareas. Así que **para nuestros propósitos,** esta arquitectura de un solo hilo es equivalente a la arquitectura multihilo.

Cuando uno **necesita procesar** tareas con muchos datos, entonces tiene mucho más sentido utilizar **lenguajes multihilo como Java**. Pero para aplicaciones en tiempo real, **Node.js es la opción obvia**.



USOS COMUNES DE NODE JS

Aplicaciones de una sola página (SPA)

El Event Loop de Node.js viene al rescate aquí, ya que procesa las solicitudes de forma no bloqueante permitiendo tener peticiones para componentes específicos.

Aplicaciones basadas en REST API

JavaScript se utiliza tanto en el frontend como en el backend de los sitios. Así, un servidor puede comunicarse fácilmente con el frontend a través de APIs REST utilizando Node.js.

Chats en tiempo real

Debido a su naturaleza asíncrona de un solo hilo, es muy adecuado para procesar la comunicación en tiempo real. Se puede escalar fácilmente y se utiliza a menudo en la construcción de chatbots.

Streaming de datos

Empresas como Netflix utilizan NODE para el streaming, esto se debe principalmente a que NODE es ligero y rápido.

Ahora que conocemos cómo funciona y para qué sirve, vamos a instalarlo.

Instalación

Ingresamos a https://nodejs.org/en/ y descargamos la versión **LTS (long term support)** ya que es la más reciente y con soporte oficial recomendada para proyectos “reales” o productivos.

Primeros pasos con NodeJS

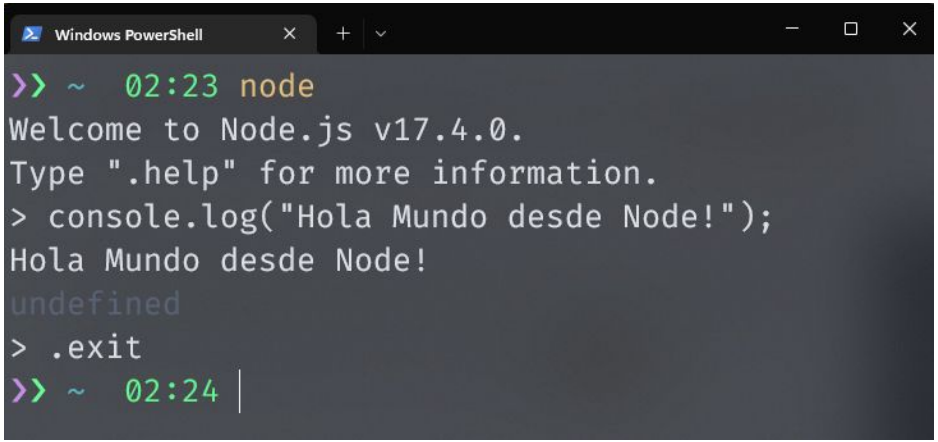
Ahora que tenemos **NODE** instalado en nuestra PC podemos trabajar con él del mismo modo que lo hacíamos con **Javascript**.

En esta ocasión para ejecutar nuestro código en lugar de usar la consola del navegador, vamos a usar la terminal de **VS CODE** o de nuestra PC.

Primeros pasos con NodeJS

También podemos escribir y ejecutar nuestro código **NODE** a través de la consola mediante el comando node.

Para salir de este modo, usamos el comando .exit



Como podemos ver cambia el entorno de ejecución pero mantenemos la misma sintaxis.

NODE JS

Web Server

En esta clase vamos a aprender a crear un servidor web con NODE

Pero antes ...

Algunos conceptos nuevos.

MÓDULOS

Uno de los problemas de Javascript desde sus inicios es organizar de una forma adecuada una aplicación grande, con muchas líneas de código.

Tener todo el código en un sólo archivo Javascript es confuso y complejo.

La solución a ese problema fueron los módulos, que permiten separar nuestro código sin tener que vincular una etiqueta script por cada archivo de Javascript en nuestro proyecto.

Historia de los módulos

El precursor para esta solución fue NodeJS quién creó el sistema de módulos conocido como ***CommonJS***.

Sin embargo, a partir de la especificación EcmaScript 2015 se introduce al lenguaje una alternativa nativa conocida como ***ES Modules***.

Ambos tienen su propia sintaxis y si bien no es común verlos en proyectos con Javascript puro, son muy usados en el Frontend en Frameworks como React o en desarrollos backend realizados con NodeJS.

Veamos un ejemplo de CommonJS

Tenemos un **módulo** con una función que permite sumar 2 números a la cual exportamos con module.exports

```
modulo.js > ...
1   function sumar(a, b) {
2       return a + b;
3   }
4
5   module.exports = sumar;
6
```

Ahora llamamos a la función sumar del archivo modulo.js y lo utilizamos en nuestro archivo index.js

```
index.js > ...
1   const sumar = require('./modulo');
2
3   const resultado = sumar(10, 17);
4
5   console.log(resultado);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

>> node  02:46  node index.js
27
>> node  02:46
```

***más adelante veremos como es la sintaxis para manejar módulos con ES Modules. Vamos despacio 😉**

Tipos de Módulos

Nativos

Son módulos que vienen incluidos dentro del código fuente de NODE.

Internos

Todos los módulos creados durante el desarrollo del proyecto y que hacen a nuestra aplicación.

Externos o de terceros

Librerías creadas por terceros y puestas a disposición a través de gestores de paquetes de NODE como NPM o YARN.

Gestores de Paquetes

Los package manager más conocidos para **NODE** son **NPM** y **YARN**.

Estos sirven como bibliotecas que contienen módulos de terceros con soluciones sencillas a problemas comunes y a veces no tan comunes.

Con ellos podemos instalar librerías de código que nos ayuden con tareas simples como animaciones, alertas o trabajo con fechas, sin embargo también podemos usarlo para descargar frameworks como React o Angular.

NPM

Node Package Manager es el gestor de paquetes más conocido y utilizado.

Se instala automáticamente cuando instalamos **NODE** por lo que **no** debemos instalar nada adicional.

Para poder instalar dependencias o librerías en nuestros proyectos primero hay que utilizar el comando **NPM init** o **NPM init -y** en la terminal para dar inicio a un proyecto de **NODE** gestionado por **NPM**.

```
node  03:06 npm init -y
Wrote to C:\Users\pol_m\Desktop\node\package.json:

{
  "name": "node",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

package.json

Al iniciar **NPM** en nuestro proyecto se crea un archivo llamado package.json.

Este archivo será utilizado por el gestor de paquetes para listar nuestras dependencias a medida que las vayamos instalando y para guardar información general del proyecto como:

- **Nombre**
- **Versión**
- **Descripción**
- **Autor**
- **Licencia**

La propiedad main indica el **archivo de "entrada"** de nuestro proyecto mientras que la propiedad scripts nos permite **crear** comandos para ejecutar distintas acciones de nuestro código.

¡Ahora sí!

A crear un servidor web

¿Qué es un Servidor Web?

Un servidor de **software** o ***Servidor Web HTTP*** controla cómo los usuarios de la web obtienen acceso a los archivos alojados en un servidor de hardware (una pc).

Son **capaces de comprender urls** o solicitudes a través de ellas y dar una respuesta atendiendo dicha solicitud.

Existen servidores estáticos y dinámicos.

Tipos de servidores WEB

Estático

Consiste en una computadora (hardware) con un servidor HTTP (software).

Se le dice "estático" porque envía los archivos que aloja "tal como se encuentran" (sin modificarlos) a tu navegador.

Dinámico

Consiste en un servidor web estático con software adicional, como una aplicación de servidor y una base de datos.

Se le dice "dinámico" a este servidor porque la aplicación actualiza los archivos alojados, antes de enviar el contenido a tu navegador mediante el servidor HTTP.

Para entender un poco más, creemos nuestro primer servidor estático con node

Servidor Estático Nativo

Para poder montar nuestro servidor, debemos tener un archivo de entrada o “entry point” donde realicemos la configuración inicial de nuestro Server.

Creamos nuestro entry point llamado: **app.js**

Luego importamos el módulo `http` nativo y **creamos un servidor** con el método:

`.createServer();`


```
const http = require('http');

const server = http.createServer();
```

Servidor Estático Nativo

Cada llamada a nuestro server recibe 2 parámetros super importantes: `require` y `response`, que contienen **toda la información** tanto de la solicitud como de la respuesta en ese orden.

Finalmente escribimos una cabecera mediante `.writeHead();` indicando el **tipo de contenido** que vamos a devolver y lo enviamos.

```
const http = require('http');


const server = http.createServer((req, res) => {
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    });
    res.end('Hola mundo');
});
```

Servidor Estático Nativo

Ya casi lo tenemos, ahora solo nos queda **escuchar a un puerto** para poder realizar llamadas a nuestro servidor.

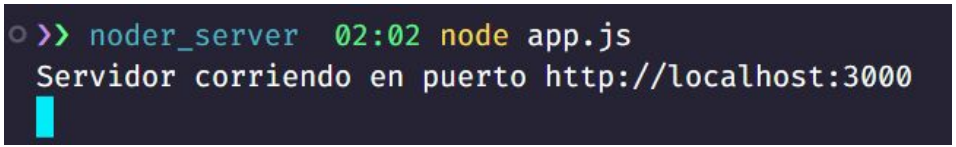
Para eso usamos el método `.listen();` el cual trabaja sobre nuestra constante `server` y recibe el puerto como primer parámetro y **un callback** en segundo.

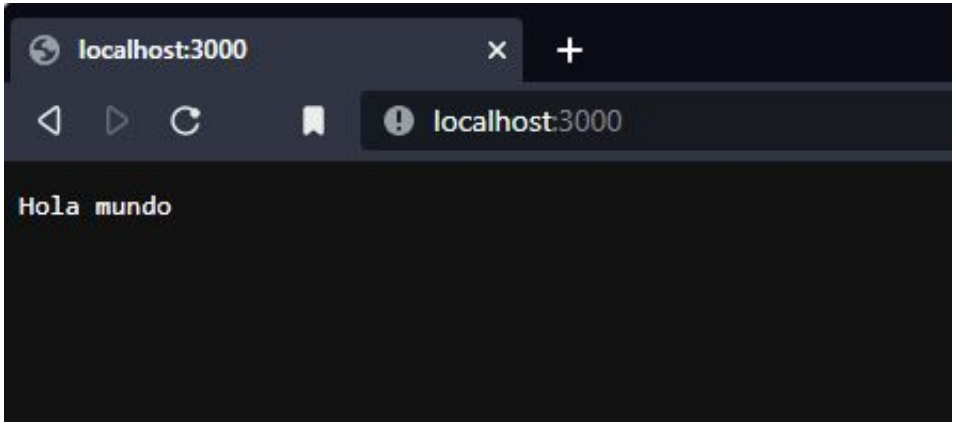
```
const http = require('http');

const server = http.createServer((req, res) => {
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    });
    res.end('Hola mundo');
});

server.listen(3000, () => console.log('Servidor
corriendo en puerto http://localhost:3000'));
```

Servidor Estático Nativo

¡Listo!

Ahora pongamos a correr nuestro servidor desde la terminal mediante el comando `node app.js` y accedemos a él desde el navegador.





En este caso devolvemos texto plano, pero intentemos con HTML.

Enviando HTML

Modificamos ligeramente la cabecera **Content-Type**.

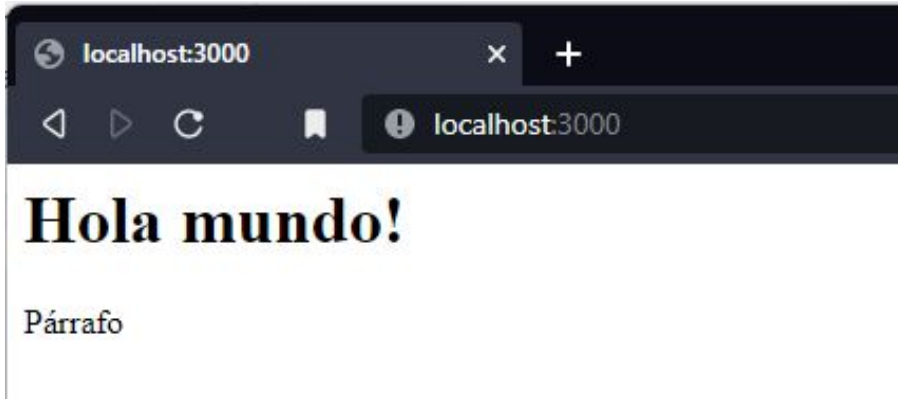
Si no indicamos el charset, el tilde de "párrafo" no se verá correctamente.

```
res.writeHead(200, {
    'Content-Type': 'text/html; charset=UTF-8'
});

res.end('<h1>Hola mundo!</h1><p>Párrafo</p>');
```

Además en lugar de texto, enviamos código HTML válido.

Reiniciamos el server y volvemos al navegador:

```
>> noder_server  02:02 node app.js
Servidor corriendo en puerto http://localhost:3000
```



Intentemos con un archivo .html

Leyendo archivos con FileSystem

FileSystem es un módulo nativo de node que **nos permite trabajar con archivos** que existan en la PC o servidor.

Veamos cómo utilizarlo para devolver un archivo HTML como respuesta.

```
const http = require('http');
const fs = require('fs');

const server = http.createServer((req, res) => {
    const file = fs.readFileSync(__dirname + '/index.html');
    res.writeHead(200, {
        'Content-Type': 'text/html; charset=UTF-8
'
    });
    res.end(file);
});
```

`.readFileSync();` lee un archivo de forma **síncrona** (bloqueante) y luego lo devolvemos como respuesta a la petición.

NODE JS

Express

Un **framework** a la medida.

Frameworks Node

Montar un server de forma nativa con Node para proyectos robustos resulta **algo tedioso y difícil de escalar**.

Por eso **existen diversos Frameworks** de Node como **HapiJS**, **Koa**, **NestJS** o **Express**, entre otros.

Este último es el más popular y actualmente se encuentra bajo el soporte de la OpenJS Foundation.

Express JS

Es un framework de aplicaciones web NODE mínimo y flexible.

Posee miles métodos y middlewares para **programas HTTP** que facilitan la creación de una API sólida de **forma rápida y sencilla**.

Una de las ventajas de Express es que nos permite levantar un servidor web muy fácilmente.

Pero antes de eso, debemos preparar nuestro proyecto para trabajar con librerías.

En la terminal corremos el comando:

```
npm init -y
```

```
>> noder_server_express  02:54 npm init -y
Wrote to C:\Users\pol_m\Desktop\noder_server_express\package.json:

{
  "name": "noder_server_express",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Express JS

Ahora nos toca **instalar Express** mediante **npm**:

```
npm install express  --save
```

El flag **--save** indica que debe registrar la **dependencia** y sus subdependencias actualizadas.

```
noder_server_express  03:00 npm install express --save

added 57 packages, and audited 58 packages in 60s

7 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

```
{
  "name": "noder_server_express",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Vemos en el package.json que contamos con una lista de dependencias con express en la versión que acaba de instalar.

Ya tenemos Express instalado.

¡Es momento de crear un nuevo server!

Servidor Estático con Express

Borramos el contenido de nuestro archivo **app.js** e importamos el módulo de express.

Luego ejecutamos la función `express();` y la guardamos en una variable llamada `app`.

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

const app = express();
```

Servidor Estático con Express

Una vez importado el módulo y ejecutada una instancia de express tenemos que definir el puerto que va a estar escuchando nuestro servidor y configurar nuestra primera ruta con la respuesta a su petición:

```
const port = 3000;

app.get("/", (req, res) => {
  res.send('Hola Mundo!');
});

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`);
});
```

El método **get** de **app** escuchará las peticiones a la ruta **/** a través del método HTTP GET y responderá el texto citado.

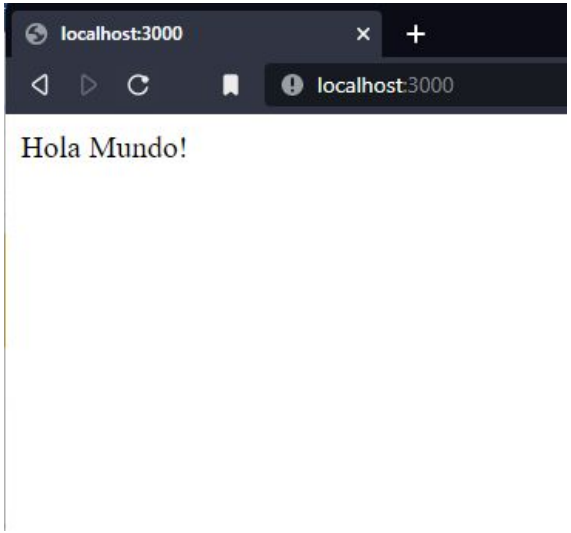
El método listen recibe un parámetro **port** con el puerto donde correrá el server y un callback que en este caso lo usamos para enviar un mensaje por consola.

Servidor Estático con **Express**

Para ejecutar nuestro servidor podemos hacerlo igual que antes mediante la terminal con `node app.js` o definiendo un script para ello en nuestro archivo **package.json**:

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node app.js"
  },
```

```
>> noder_server_express  03:18 npm start

> noder_server_express@1.0.0 start
> node app.js

Example app listening at http://localhost:3000
```



De esta manera **podemos devolver** un **texto** o **un archivo** estático que **responda a una ruta específica** mediante el método **GET**

Express Generator

Es otra forma de instalar **Express** con un ***boilerplate*** ya configurado.

Para poder usar este **comando**, hay que instalar express generator en nuestra **PC** de forma global:

```
npm install express-generator -g
```

Express Generator

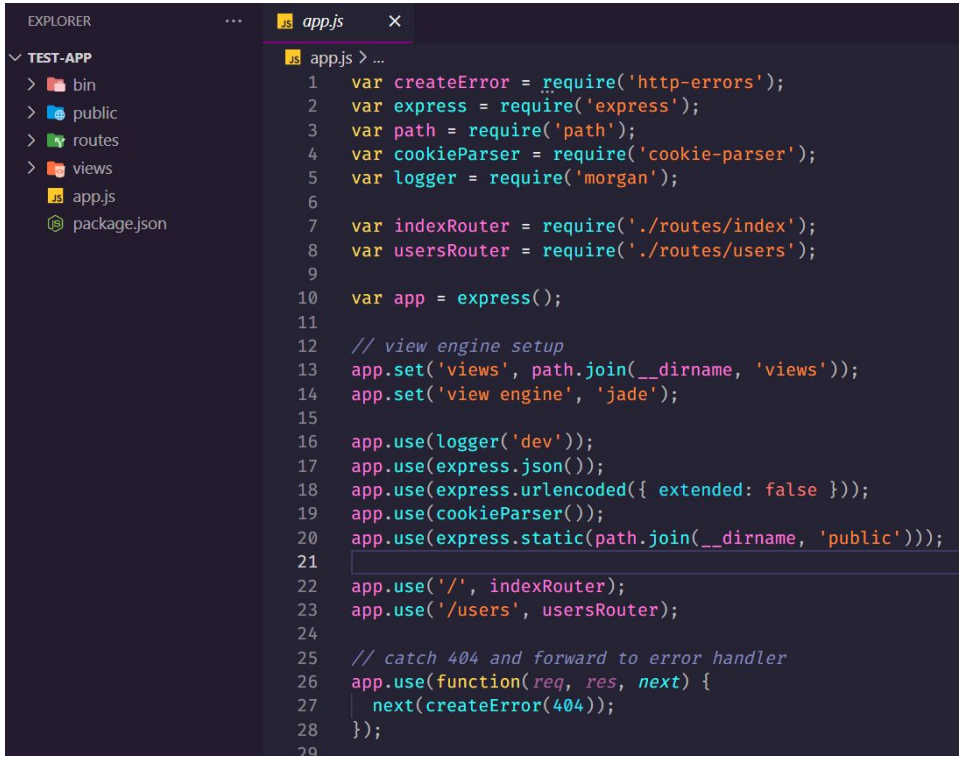
Una vez instalado podemos correr el comando:

```
express test-app
```

Esto nos creará un proyecto nuevo en una carpeta llamada ***"test-app"*** con una serie de archivos y carpetas ya preparados para comenzar a desarrollar nuestra app.

De esta manera tenemos nos ahorramos la configuración básica de express

No olvidar correr el comando npm install para instalar todas las dependencias necesarias.



Ya sabemos crear un servidor con Express, es momento de profundizar un poco más.

Archivos Estáticos

La clase pasada hablamos sobre los servidores estáticos y **dinámicos**.

Veamos cómo **podemos** con Express definir una carpeta que **sirva archivos** tal como lo haría un servidor estático.

Pero antes un paso súper necesario, **nodemon**...

Nodemon

Esta **librería** nos ayuda recargando el servidor frente a cada cambio, sin tener que hacerlo manualmente.

La instalamos como dependencia de desarrollo:

```
npm install -D nodemon
```

*** Las dependencias de desarrollo son aquellas que solo se utilizarán mientras creamos el proyecto.**

Una vez lista, modificamos ligeramente el script de nuestro **package.json** por:

```
"scripts": {
  "start": "nodemon app.js"
}
```

Node v18+

Desde la **versión 18** de **NODE** no se necesita instalar Nodemon ya que el mismo programa **cuenta** con una funcionalidad propia para recargar nuestro servidor, el flag **--watch**.

```
"scripts": {
  "start": "node --watch app.js"
}
```

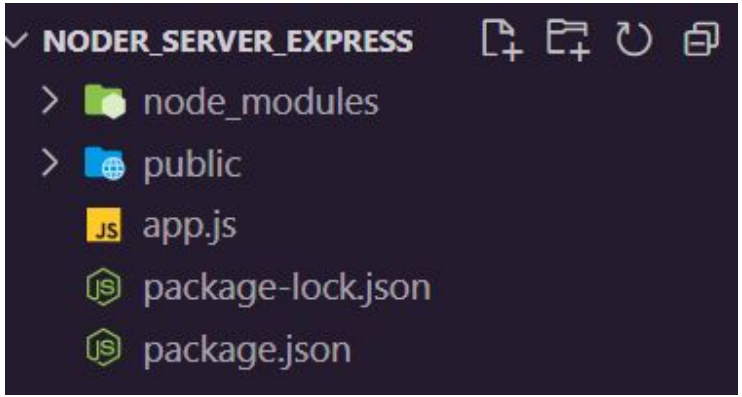
Al ser una feature tan reciente y dado que no todas las PCs son compatibles con la última versión, por el momento se puede **trabajar con Nodemon**.

Ahora sigamos con los archivos estáticos.

Carpeta public

Lo primero es **crear** una carpeta public.

Allí irán todos los archivos que deberán ser **enviados** tal como fueron alojados.



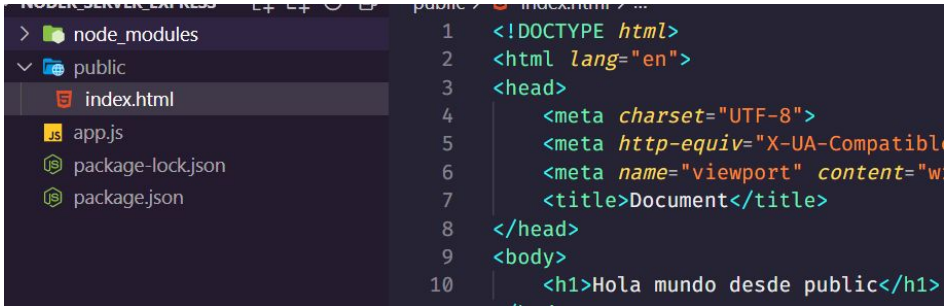
Luego en nuestro archivo **app.js** agregaremos la ruta a esta carpeta, indicando a Express de que se trata.

```
app.use(express.static('public'));
```

El método `.use()` es un **middleware**, concepto que veremos más adelante.

Por el momento debemos saber que nos permite interceptar lo que se ejecute dentro antes que la derivación de nuestras rutas.

Archivos estáticos



Hola mundo desde public

Ahora creamos un archivo **index.html** y accedemos a la ruta **http://localhost:3000**

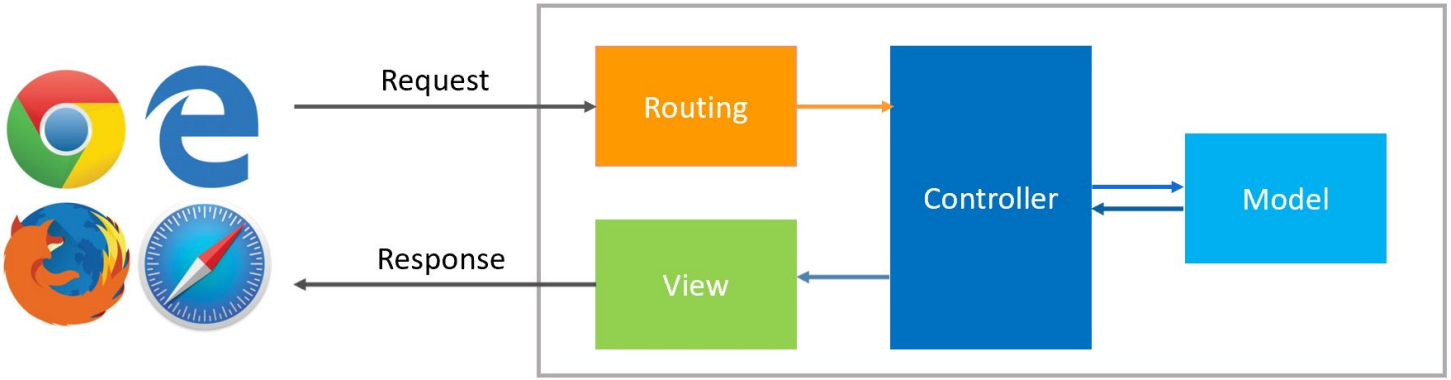
¡Magia! Nos devuelve nuestro archivo **HTML**.

Así como index.html, podemos devolver cualquier recurso a través de su ruta.

Hablando de rutas...

Rutas

El **cliente buscará** acceder al contenido **NO ESTÁTICO** de nuestro Backend a través de peticiones HTTP a diferentes **rutas** o endpoints configurados en nuestra aplicación.



Rutas

Nuestras rutas serán definidas en Express de la siguiente manera:

```
app.get('/nosotros', (req, res) => {
    res.send(__dirname + './nosotros.html');
})
```

El método get de app escuchará las peticiones a la ruta **/nosotros** a través del método HTTP GET y responderá el archivo solicitado.

la variable app de express puede escuchar a todos los métodos HTTP, entre ellos **GET, **POST**, **PATCH**, **PUT** y **DELETE**, entre otros.*

***__dirname** nos permite tomar como referencia el lugar actual de nuestro archivo dentro del servidor y **llegar a un recurso** desde esa ruta.*

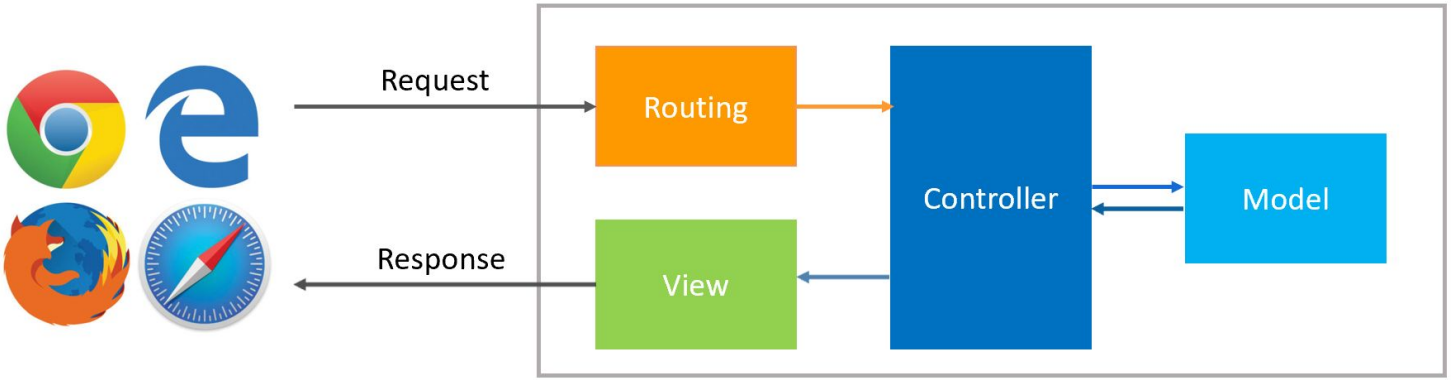
NODE JS

Rutas - Parte I

La clase anterior nos quedamos en...

Rutas

El **cliente buscará** acceder al contenido **NO ESTÁTICO** de nuestro Backend a través de peticiones HTTP a diferentes **rutas** o endpoints configurados en nuestra aplicación.



Las rutas son la manera que tiene nuestro servidor de exponer contenido *“no estático”* a través de la web.

Rutas

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

GET

POST

PATCH

PUT

DELETE

Rutas

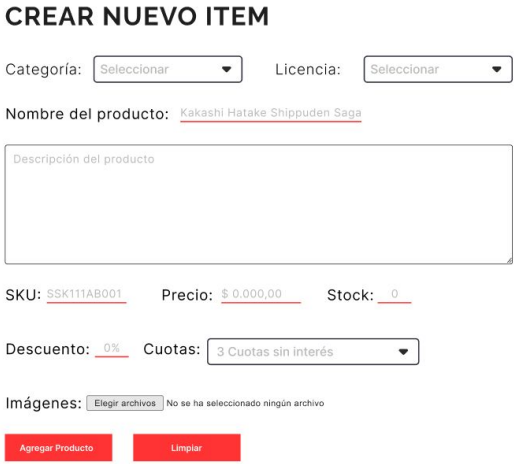
Por ende, nuestro servidor no solo escuchará **“paths”** o rutas si no que también tendrá en cuenta el *método utilizado en la request*.

Esto nos permite usar la misma ruta para diferentes cosas.

Por ejemplo...

No es lo mismo solicitar la ruta **“/admin/create”** mediante **GET** en la **URL**:

Que utilizar un formulario para enviar datos a través de **POST**:




```
<form action="/admin/create" method="POST" enctype="application/x-ww
    <div>
        <label class="form__box--label" for="collection">Categoría:</
        <select class="form__box--select" name="collection" id="colle
            <option value="" disabled selected hidden>Seleccionar</op
            <option value="Figuras Coleccionables">Figuras Coleccion
            <option value="Llaveros">Llaveros</option>
            <option value="Remeras">Remeras</option>
        </select>
    </div>
    <div>
```

Rutas

Nuestras rutas serán definidas en Express de la siguiente manera:

```
app.get('/admin/create, (req, res) => {
    res.send(__dirname + './create.html');
})
```

El método get de app escuchará las peticiones a la ruta **/admin/create** a través del método HTTP GET y responderá el archivo solicitado.

la variable **app de express puede escuchar a todos los métodos HTTP, entre ellos **GET**, **POST**, **PATCH**, **PUT** y **DELETE**, entre otros.*

***__dirname** nos permite tomar como referencia el lugar actual de nuestro archivo dentro del servidor y **llegar a un recurso** desde esa ruta.*

Respondiendo a rutas

Tenemos un archivo items.json, el cual **queremos leer** y **devolver** cuando un cliente pida la ruta `"/items"`.

```
app.get("/items", (req, res) => {
    const getItems = fs.readFileSync(__dirname + '/data/items.json');
    res.send(JSON.parse(getItems));
})
```

En este caso si **entramos** a localhost:3000 seguiremos obteniendo nuestro archivo index.html (estático) pero si solicitamos esta ruta nos devolverá un array con los **productos** en formato JSON.

***readFileSync nos devuelve un string o cadena de texto con la información y nosotros la convertimos a JSON a través del método JSON.parse().**

Rutas Parametrizadas

Rutas parametrizadas - params

Ahora supongamos que queremos **traer** solo **un item** de la lista

¿Cómo podríamos resolverlo?

Las respuestas son las ***"rutas parametrizadas"***, gracias a ellas podemos leer una parte de la URL y utilizarla para devolver una respuesta diferente según el caso.

```
app.get("/items/:id", (req, res) => {
    const id = req.params.id;
    // lógica
})
```

En este caso :id será un valor que pasaremos en la URL y será leído al momento de recibir la petición.

A través de **params**, capturamos el valor de la URL.

Rutas parametrizadas - query

Otra manera de pedir datos a través de la URL es mediante los **querys**.

`http://localhost:3000/items?licence=pokemon`

```
app.get("/items", (req, res) => {
    const licence = req.query.licence; //pokemon
      // lógica que filtra los ítems de la licencia
      pokémon
})
```

Usamos **la misma ruta que para todos los items**, solo que si recibimos un query param, capturamos el valor de la URL e incluimos una lógica para devolver solo los valores coincidentes.

NODE JS

Rutas II

Rutas

Como vimos hasta el momento, la comunicación entre clientes y servidores o entre programas que interactúan a través de la web, se hace **mediante rutas**.

A estas **rutas** se las conoce comúnmente como ENDPOINTS y necesitaremos uno por cada flujo que posea nuestro servidor.

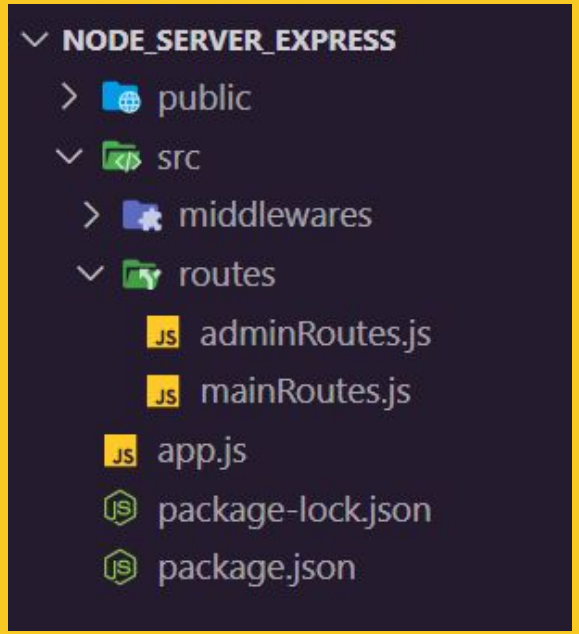
Ahora si, nuestras rutas se van complejizando.

Express Router

Permite separar rutas en diversos archivos que van a devolver a cada parte de nuestro programa.

Primero, **ordenemos nuestro código** creando una **carpeta src** para guardar toda la lógica de nuestro programa.

Dentro creamos una carpeta nueva llamada routes.



Express Router

En el archivo mainRoutes.js llamamos al método Router() de express.

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

Express Router

Luego, creamos o migramos desde el archivo app todas nuestras rutas:

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

/* MAIN ROUTES */

router.get('/home', (req, res) => res.send("Página de Home"));
router.get('/contact', (req, res) => res.send("Página de Contacto"));
router.get('/about', (req, res) => res.send("Página Sobre Nosotros"));
```

En lugar de app.get(...) utilizamos **router.get(...)**

Express Router

Una vez definidas todas las rutas o **endpoints** debemos exportar el módulo router para ser utilizado desde app.js:

```
module.exports = router;
```

Express Router

Finalmente llamamos nuestro archivo de rutas desde app.js y a través del **middleware** app.use() indicamos que peticiones deben ser respondidas con esas rutas.

```
const express = require('express');
const app = express();
const mainRoutes = require('./src/routes/mainRoutes.js')

const PORT = 3000;

app.use(express.static('public'));
app.use('/', mainRoutes);

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

Probemos nuestros endpoints.

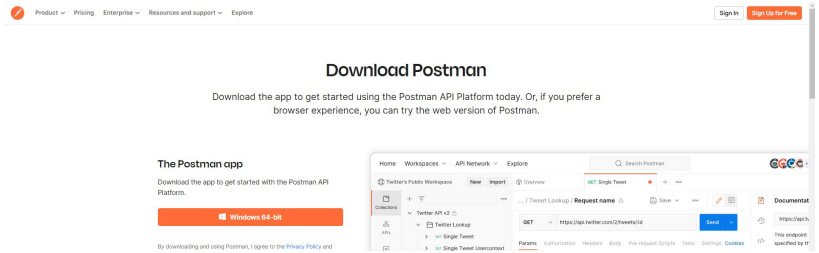
POSTMAN/INSOMNIA

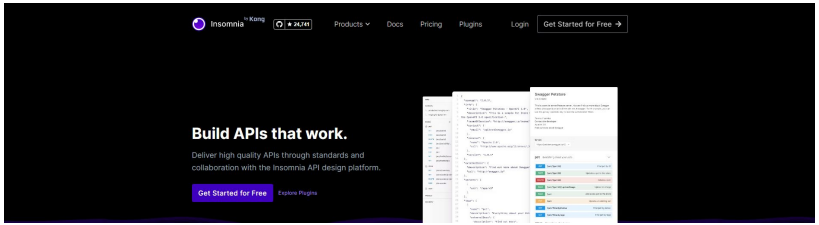
Para lograr esto podemos usar programas que nos permiten **“simular”** estas peticiones a través de los distintos métodos HTTP.

Estas herramientas facilitan enviar consultas a nuestro servidor sin **depender de un Frontend armado**.

https://www.postman.com/

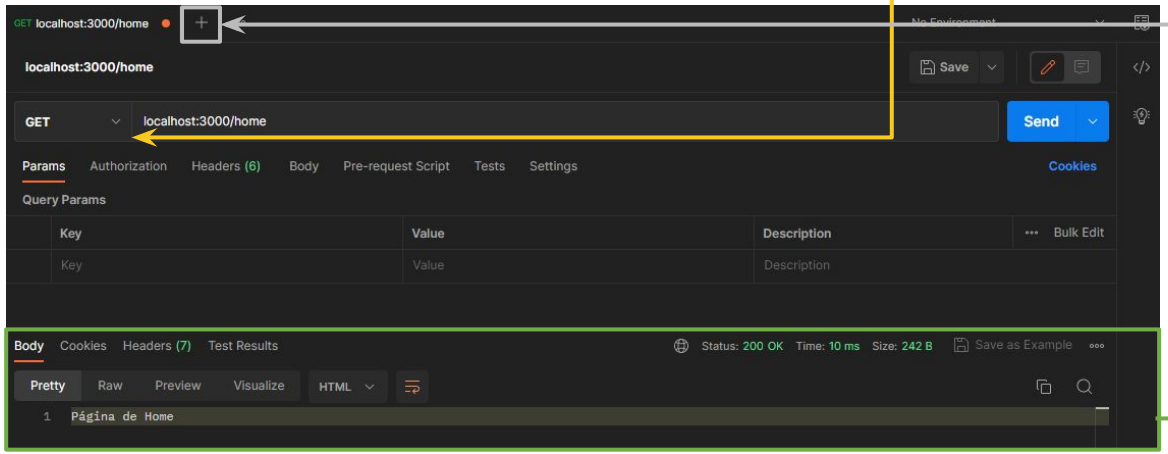
https://insomnia.rest/download





POSTMAN

Dentro de **POSTMAN** podemos crear una nueva petición con el botón de **+**, luego seleccionamos el método (en este caso **GET**) y la **url** a consultar.



Finalmente presionamos **SEND** y esperamos la respuesta del servidor.

<codoacodo/>

De esta manera probamos nuestro backend sin un frontend.

Agencia de Aprendizaje a lo largo de la vida

Buenos Aires Ciudad
Educación

Ahora con POST.

Parseando datos recibidos

Hasta el momento venimos trabajando con **GET**, pero para que nuestro servidor pueda recibir peticiones por POST necesitamos *convertir* los datos recibidos en el **BODY** a un formato que **entienda el servidor**.

app.js

```
/*
* Convertimos los datos entrantes en formato
* application/x-www-form-urlencoded y application/json
* a un formato entendible por el servidor
*/

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

Usando los **middlewares** nativos .urlencoded() y .json() podemos convertir la data de estos formatos a uno que el **servidor pueda manejar**.

***nota:** en versiones previas a **express 4.16.0** se utilizaba una librería llamada body-parser para este propósito.*

Finalmente con PUT y DELETE

Sobre escribiendo métodos

Los navegadores web, por ejemplo desde un formulario únicamente soportan los métodos **GET** y **POST**, por lo que cuando deseamos utilizar un método diferente, es preciso **sobreescribirlo**.

```
npm install method-override
```

```
const methodOverride = require('method-override');

// Override para habilitar los métodos PUT y DELETE

app.use(methodOverride('_method'));
```

Para esto utilizaremos una dependencia llamada **method-override**, otro middleware que captura la petición y si posee una mención a **algún método no soportado**, **lo sobreescribe**.

***nota:** en versiones previas a **express 4.16.0** se utilizaba una librería llamada body-parser para este propósito.*

Sobre escribiendo métodos

Ahora desde nuestros **formularios HTML**, solo debemos anteponer `?_method=PUT` o `?_method=DELETE` en la URL del atributo action para indicar el **método real** y utilizar **POST** en el atributo method del formulario.

```
<form
    action="/admin/edit/20?_method=PUT"
    method="POST"
    enctype="multipart/form-data"
>
  ...
</form>
```

Para terminar, hay una palabra que se usó mucho pero no vimos...

¿Cuál era? 🤔

Ahh, si.

Middleware

Middleware

Casi de forma instintiva, ya utilizamos varios middlewares para configurar nuestro programa.

¿Qué es un middleware entonces?

Son simplemente funciones que se ejecutan antes o después de otras y los hay de distintos tipos:

- **Middlewares de nivel de aplicación**: Son middlewares que se aplican a toda la aplicación y se configuran utilizando `app.use()`
- **Middlewares de nivel de ruta**: Son middlewares que se aplican a una ruta específica.
- **Middlewares de manejo de errores:** Son middlewares especiales que se utilizan para manejar errores en la aplicación.

A lo largo de las clases iremos viendo distintos middlewares y también aprenderemos a crear los propios.

NODE JS

Controllers

Antes de comenzar con el tema de hoy, veamos cómo crear un middleware que maneje las rutas que no existen de nuestra app.

Error 404

Para manejar este error vamos a crear un **middleware** en nuestro archivo app.js

```
(COLOCAR LUEGO DE LAS RUTAS Y ANTES DEL .listen())

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

Este código revisa el estado de la petición y si no encuentra el recurso devuelve un **código 404**. Luego en .send() podemos enviar lo que deseamos, desde un texto hasta una **vista HTML**.

Controladores

Son una de las **capas de MVC** (modelo-vista-controlador) y ayuda a separar nuestra aplicación en capas.

Hasta ahora la respuesta a un **ENDPOINT** se encontraba dentro de la ruta:

```
router.get('/home', (req, res) => res.send("Página de Home"));
```

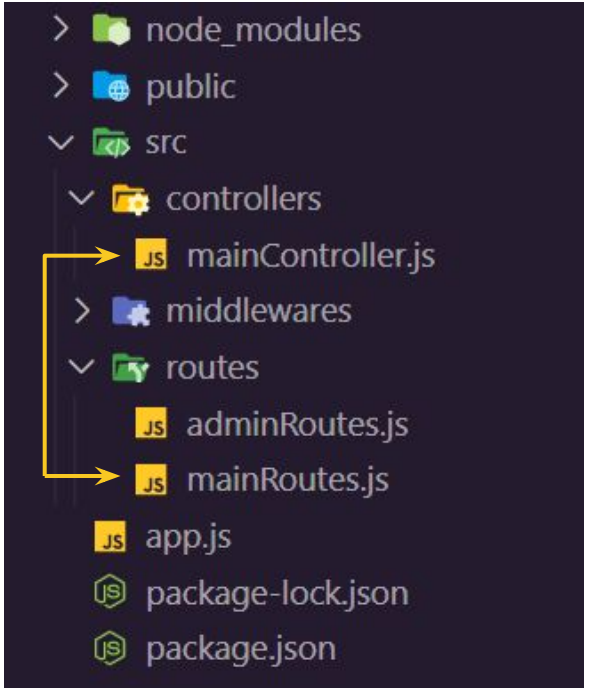
Pero, ¿qué pasa cuando tengo muchas rutas y sus respuestas son más complejas?

Controladores

Lo mejor en estos casos es separar el código para que sea más **escalable** y **legible**.

mainController.js

```
module.exports = {
  home: (req, res) => res.send("Página de Home"),
  contact: (req, res) => res.send("Página de Contacto"),
  about: (req, res) => res.send("Página Sobre Nosotros")
}
```



Controladores

Una vez que tenemos nuestros controladores los usamos en el archivo de rutas donde antes colocábamos la **lógica de respuesta.**

mainRoutes.js

```
const mainController = require('../controllers/mainController.js');
/* MAIN ROUTES */
router.get('/home', mainController.home);
router.get('/contact', mainController.contact);
router.get('/about', mainController.about);

module.exports = router;
```

De esta manera logramos dividir la lógica de nuestra aplicación.

Veámoslo en la práctica...

Datos protegidos

En una aplicación a veces necesitamos utilizar **valores** o **constantes** que **NO** se expongan en nuestro código.

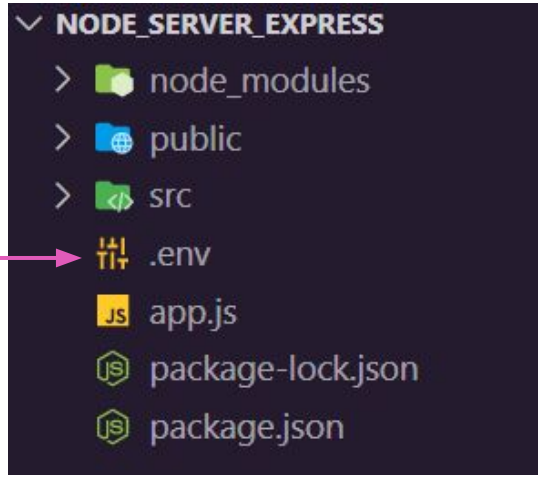
Para ello se utiliza algo conocido como .ENV.

.env

Comencemos instalando la dependencia **dotenv**:

```
npm install dotenv
```

Luego creamos un archivo en la raíz del proyecto, llamado **.env** y escribimos un valor que deseamos que sea “secreto” como el puerto a utilizar en la APP.



.env

```
PORT=3000
```

.env

Ahora podemos utilizar esa variable en nuestro archivo **app.js**

```
/* Requerimos la dependencia*/
require('dotenv').config();
/* Leemos la constante*/
const PORT = process.env.PORT;
```

De esta manera veremos que seguimos utilizando el mismo puerto pero ahora lo ***lee desde un archivo oculto***.

```
Servidor corriendo en http://localhost:3000
```

Este enfoque es muy útil por ejemplo para guardar las credenciales de acceso a una base de datos que no deben ser expuestas.