# CREAR UN API CON NODE JS, EXPRESS Y MONGODB

GUSTAVO MORALES

# Creando APIs con Node.js, Express y MongoDB

Gustavo Morales

Este libro está a la venta en http://leanpub.com/creando-apis-con-nodejs-express-y-mongodb

Esta versión se publicó en 2020-09-01

# Índice general

# Introducción

## A quién está dirigido este libro

Este libro va dirigido a programadores que han iniciado con Node.js o tienen experiencia en otros lenguajes de *Backend* como Java, Rails o PHP, y quieren utilizar Node.js como componente de *Middleware* o *Backend* para hacer un REST API Web.

Si eres nuevo en Node.js se recomienda leer el siguiente libro de este misma serie:

- Introducción a Node.js[1]

También se asume que el lector tiene conocimientos básicos de comandos en la *Terminal* y utilización básica de *git*.

## Acerca de este libro

Este libro tiene un objetivo teórico-práctico, durante el transcurso de este construiremos una aplicación desde el inicio hasta el fin, comenzaremos por lo conceptos básicos y gradualmente en cada capítulo introduciremos nuevos conceptos, fragmentos de código y buenas prácticas, veremos como la aplicación se convierta en un proyecto robusto. Pero a su vez si usted lo desea puede saltar a un capítulo en específico, por ejemplo para ver el versionamiento de API o la persistencia de datos con MongoDB.

Utilizaremos *git* como software de control de versiones para organizar cada uno de los pasos incrementales (*features*) de la aplicación, brindando así la posibilidad de saltar a un paso específico en el tiempo de desarrollo del proyecto y si usted lo desea comenzar desde allí, *git* a la vez nos brinda la ventaja de ver los cambios en los archivos de manera visual y así poder comprobar en caso de una omisión o error.

## Acerca del autor

Soy un entusiasta de la Web desde la década de finales de los 90 cuando se utilizaba el bloc de notas y nombre de etiquetas en mayúsculas para hacer las páginas Web, soy Ingeniero de Sistemas y tengo más de diez años de experiencia como profesor de pregrado en la asignatura de programación bajo la Web y en posgrado con otras asignaturas relacionadas con el mundo de desarrollo de

[1] https://leanpub.com/introduccion-a-nodejs

software, trabajo actualmente en la industria del desarrollo de software específicamente la programación Web. Puedes encontrar más información acerca de mi hoja de vida en el siguiente enlace: http://gmoralesc.me[2].

**Gustavo Morales**

# Agradecimientos

Quiero agradecer a mi esposa Milagro y mi hijo Miguel Angel por apoyarme y entender la gran cantidad de horas invertidas frente al computador: investigando, estudiando, probando y escribiendo.

[2] http://gmoralesc.me

# ¿Que es un API?

Es un acrónimo en inglés de *Application Programming Interface*, es el contrato que una aplicación expone para que otras aplicaciones interactúen con sus métodos o servicios, a través de una **interfaz** consumiendo unos **recursos** específicos.

## ¿Que es un API Web?

En nuestro proyecto crearemos una API Web la cual utiliza como **interfaz** el protocolo Web HTTP (o HTTPS sea el caso) y los **recursos** son las **entidades**, por ejemplo nosotros vamos a crear un administrador de tareas por lo tanto nuestros **recursos** serán tareas, usuarios y grupos, entonces nuestras entidades serán: *tasks*, *users* y *groups*, normalmente las entidades se nombran el plural y las cuales se acceden a través de unas **rutas** en este caso son **urls**.

> Se recomienda utilizar el idioma inglés independientemente de la lengua nativa de la aplicación.

Existen otros tipos de API por ejemplo en vez del protocolo HTTP pueden ser *sockets* y en vez de entidades pueden ser canales, este tipo de API son muy utilizadas en juegos en línea, por lo tanto es importante definir que tipo de API estamos construyendo.

## Códigos y verbos HTTP

El protocolo HTTP utiliza diferentes **verbos** para las diferentes operaciones que se realizan y dependiendo del resultado de estas operaciones se le asignan diferentes **códigos**.

Los verbos Web más utilizados son:

- GET: Se utiliza cuando se envía una solicitud para obtener un recurso por ejemplo un documento HTML a través de una *url*.
- OPTIONS: Se utiliza para diferentes operaciones, como petición auxiliar para comprobar si el servidor soporta el método que se va a utilizar en general cualquiera diferente a GET, esta comprobación se llama *preflight*.
- POST: Se utiliza para crear un recurso, normalmente enviar información de formularios Web.

Pero no son los únicos, existen además: PUT, PATCH y DELETE para asociarlos con las diferentes operaciones. También como parte del protocolo HTTP están HEAD, CONNECT y TRACE utilizados por otras operaciones Web.

Dentro del listado de códigos tenemos:

- 2xx: Este grupo se utiliza para operaciones exitosas.
- 200: Cuando la operación es exitosa.
- 201: Lo mismo que el 200 pero en este caso cuando además se crea un recurso de manera exitosa.
- 204: Lo mismo que el 201 pero esta vez no fue necesario enviar ningún tipo de información de vuelta.
- 3xx: Este grupo se utiliza para indicar redirecciones.
- 4xx: Este grupo se utiliza para enumerar los errores del lado del cliente.
- 400: Indica que la petición no fue bien realizada normalmente cuando los datos se envían malformados.
- 401: No está autorizado para acceder al recurso, normalmente se utiliza cuando no se está autenticado.
- 403: Puede que esté "autenticado" pero no tiene permisos para acceder a ese recurso.
- 404: Recurso no encontrado.
- 422: Indica que la petición no se puede procesar debido a que faltan datos requeridos o están en un formato incorrecto.
- 5xx: Este conjunto está asociado a errores del lado del servidor.

# REST API Web

REST es un acrónimo de *Representational State Transfer*, que especifica un contrato por defecto para la relación que debe haber entre las rutas, verbos, códigos (estos dos anteriores pertenecientes al protocolo HTTP) y operaciones principales como: Crear (Create), Leer (Read), Actualizar (Update) y Borrar (Delete) conocido mucho por su acrónimo CRUD, la siguiente tabla mostrará cómo sería para nuestra entidad de tareas (*tasks*):

| | **Verbo** | **Status Code** | **Operación** | **Descripción** |
|---|---|---|---|---|
| /api/tasks/ | POST | 201 | CREATE | Crear un tarea |
| /api/tasks/ | GET | 200 | READ | Leer todos las tareas |
| /api/tasks/:id | GET | 200 | READ | Leer una tarea |
| /api/tasks/:id | PUT / PATCH | 200 | UPDATE | Actualizar la información de una tarea |
| /api/tasks/:id | DELETE | 200 (o 204) | DELETE | Eliminar una tarea |

Este contrato no es una camisa de fuerza, pero se deben respetar las definiciones básicas, no está permitido por ejemplo crear una tarea con el método DELETE en vez de POST, no tiene mucho sentido, pero se pueden añadir muchas más operaciones, recursos anidados y demás, se dice que un API es RESTful cuando cumple con este contrato mínimo.

# Inicializar proyecto

Seleccionamos un directorio de trabajo y creamos el directorio de la aplicación, ejecutamos los siguientes comandos en la Terminal:

```
mkdir checklist-api && cd checklist-api
```

Inicializamos el proyecto de Node.js:

```
npm init -y
```

El comando anterior genera el archivo `package.json` que luego se puede editar para cambiar los valores de la descripción, autor y demás.

Creamos el archivo principal de la aplicación:

```
touch index.js
```

Agregamos el *script* inicial en el archivo `package.json` para ejecutar nuestro archivo principal:

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

Inicializamos el repositorio de git:

```
git init
```

Creamos el archivo `.gitignore` en la raíz del proyecto:

```
touch .gitignore
```

Con el siguiente contenido y lo guardamos:

```
node_modules/
.DS_Store
Thumbs.db
```

Antes de finalizar hacemos *commit* de nuestro trabajo:

```
git add --all
git commit -m "Initial commit"
```

# Crear un Web Server

Antes de comenzar el proyecto vamos a crear un servidor Web sencillo, ya que nuestra aplicación será una REST API Web que se podrá consumir mediante el protocolo HTTP, el objetivo de nuestra API es administrar un listado de tareas por usuario y que opcionalmente las tareas se pueden organizar en grupos, todo lo anterior con persistencia de datos.

Tomaremos el ejemplo del Web Server[3] que se encuentra en la documentación de Node.js y reemplazamos el contenido por el siguiente código:

```
// index.js

const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

En el ejemplo anterior Node.js utiliza el módulo de `http` para crear un servidor Web que escuchará todas las peticiones en la dirección *IP* y puerto establecidos, como es un ejemplo responderá **a cualquier solicitud** con la cadena `Hello World` en texto plano sin formato.

Iniciamos nuestra aplicación con el siguiente comando en la Terminal:

```
npm start
```

Al ejecutar el comando anterior ocupará la Terminal y no nos permitirá ejecutar nuevos comandos por lo cual debemos abrir una nueva sesión de Terminal para ejecutar los demás comandos. En resumen tendremos siempre dos sesiones de Terminal, una donde se ejecuta el servidor de nuestra aplicación y otra donde ejecutamos todos los comandos.

Accedemos al navegador en la siguiente dirección:

[3] https://nodejs.org/api/synopsis.html

```
http://localhost:3000/
```

Normalmente en un REST API Web se consume mediante el protocolo HTTP(S) y la respuesta son objetos en formato JSON (el cual introduciremos más adelante), organizado en una serie de **rutas** que adicionalmente reciben parámetros, el contrato para establecer la relación entre las acciones del REST API Web y los verbos HTTP es bastante estandarizada, pero no la estructura de la respuesta y la organización de la rutas, en esta aplicación comenzaremos con una estructura sugerida que irá cambiando para cumplir con el objetivo del proyecto.

Antes de finalizar hacemos *commit* de nuestro trabajo:

```
git add index.js
git commit -m "Create a simple Web server"
```

Más información:

- HTTP[4]

[4] https://nodejs.org/dist/latest-v6.x/docs/api/http.html

# Herramientas de desarrollo

## Nodemon

Es muy tedioso cada vez que realicemos un cambio en los archivos del proyecto tener que detener e iniciar nuevamente la aplicación, para esto vamos a utilizar una librería llamada *nodemon,* esta vez como una dependencia solo para desarrollo, ya que no la necesitamos cuando la aplicación esté corriendo en producción, esta librería nos brinda la opción de que después de guardar cualquier archivo en el directorio de trabajo, nuestra aplicación se vuelva a reiniciar automáticamente.

```
npm install -D nodemon
```

Luego modificamos la sección de `scripts` del archivo `package.json` para añadir el *script* que ejecutará nuestra aplicación en modo de desarrollo `dev` y en el *script* `start` en modo de producción, de la siguiente manera:

```
"scripts": {
  "dev": "nodemon",
  "start": "node index",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

No es necesario indicarle a `nodemon` que archivo va a ejecutar pues por convención buscará el archivo `index.js`, el cual es el punto de entrada de nuestra aplicación.

Mas adelante se configurará el *script* de pruebas: `test`.

Si el servidor se está ejecutando lo detenemos, para ello en la Terminal donde se esta ejecutando presionamos `CTRL+C`, y luego ejecutamos el comando `npm run dev` .

Antes de finalizar hacemos *commit* de nuestro trabajo:

```
git add --all
git commit -m "Add Nodemon"
```

Más información:

- Nodemon[5]

[5] https://nodemon.io/

# ESLint

Es muy recomendado añadir una herramienta que nos ayude a comprobar la sintaxis de los archivos para evitar posibles errores y porque no, normalizar la forma en que escribimos el código, para ello utilizaremos una librería llamada ESLint que junto con el editor de texto comprobará la sintaxis de los archivos en tiempo real:

```
1 npm install -D eslint
```

ESLint nos permite seleccionar una guia de estilo ya existente o también poder crear nuestra propia guia, todo depende del común acuerdo del equipo de desarrollo.

Inicializamos la configuración con el siguiente comando:

```
1 ./node_modules/.bin/eslint --init
```

Alternativamente con la utilidad `npx`, instalada globalmente por Node.js, con el siguiente comando:

```
1 npx eslint --init
```

Si ESLint está instalado de manera global, el comando anterior se podría reemplazar con: `eslint --init`

Contestamos las preguntas del asistente de configuración de la siguiente manera:

```
? How would you like to use ESLint?
To check syntax only
To check syntax and find problems
 To check syntax, find problems, and enforce code style


? What type of modules does your project use?
JavaScript modules (import/export)
 CommonJS (require/exports)
None of these
```

```
? Which framework does your project use?
React
Vue.js
❯ None of these

? Does your project use TypeScript? (y/N) N

? Where does your code run?
❯ Browser
❯❯ Node

? How would you like to define a style for your project?
❯ Use a popular style guide
Answer questions about your style
Inspect your JavaScript file(s)

? Which style guide do you want to follow?
❯ Airbnb (https://github.com/airbnb/javascript)
Standard (https://github.com/standard/standard)
Google (https://github.com/google/eslint-config-google)

? What format do you want your config file to be in?
JavaScript
YAML
❯ JSON

Checking peerDependencies of eslint-config-airbnb-base@latest
The config that you've selected requires the following dependencies:

eslint-config-airbnb-base@latest eslint@^5.16.0 || ^6.8.0 eslint-plugin-import@^2.20\
.1
? Would you like to install them now with npm? (Y/n) Y
```

Una vez finalizado el proceso creará un archivo en la raíz de nuestro proyecto llamado `.eslintrc.json`, el cual contiene toda la configuración.

Adicionalmente vamos a añadir las siguientes excepciones:

1. Vamos a utilizar el objeto global `console` para imprimir muchos mensajes en la consola y no queremos que lo marque como error.
2. A modo de ejemplo más adelante vamos añadir un parámetro que no utilizaremos inmediatamente pero nos servirá mucho para explicar un concepto el cual es *next* y tampoco queremos que no los marque como error.

```
{
  "env": {
    "commonjs": true,
    "es2020": true,
    "node": true
  },
  "extends": ["airbnb-base"],
  "parserOptions": {
    "ecmaVersion": 11
  },
  "rules": {
    "no-console": "off",
    "no-unused-vars": [
      "error",
      {
        "argsIgnorePattern": "next"
      }
    ]
  }
}
```

Antes de finalizar hacemos *commit* de nuestro trabajo

```
git add --all
git commit -m "Add ESLint and config file"
```

Más información:

- ESLint[6]

# Visual Studio Code

Esta sección es opcional, de hecho se puede utilizar cualquier editor de texto, pero este editor es muy recomendado, por lo tanto miremos todo lo que nos ofrece.

## Instalación

El instalador oficial se puede encontrar en su (página Web oficial)[https://code.visualstudio.com/].

[6] https://eslint.org/

## Plugins

Los siguientes *plugins* son sugeridos para una mejor experiencia en el desarrollo de aplicaciones con Node.js y JavaScript.

- (Node.js Extension Pack)[https://marketplace.visualstudio.com/items?itemName=waderyan.nodejs-extension-pack]
- (ESLint)[https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint]
- (Prettier - Code formatter)[https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode]

También podemos configurar para que una vez el programador abra el proyecto en Visual Studio Code los muestre como sugerencia para instalar:

```
mkdir .vscode
touch .vscode/extensions.json
```

Con el siguiente contenido:

```
{
  "recommendations": [
    "waderyan.nodejs-extension-pack",
    "dbaeumer.vscode-eslint",
    "esbenp.prettier-vscode"
  ]
}
```

## Configuración

Para añadir nuestra configuración personalizada para el proyecto primero se debe crear el archivo de configuración:

```
touch .vscode/settings.json
```

Dentro del archivo de configuración `settings.json` colocamos la siguiente información:

```
{
  "editor.formatOnSave": true,
  "editor.renderWhitespace": "all",
  "editor.renderControlCharacters": true,
  "editor.trimAutoWhitespace": true,
  "editor.tabSize": 2,
  "files.insertFinalNewline": true,
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "prettier.singleQuote": true,
  "prettier.trailingComma": "es5"
}
```

Algunas de estas configuraciones son:

- `editor.formatOnSave`: Establecer que siempre se le dé formato al documento a la hora de guardar el archivo, que muestre todos los espacios en blanco y caracteres especiales y remueva todos los espacios en blanco que sobren.
- `files.insertFinalNewline`: Dejar una línea en blanco al final de cada archivo (requerido por defecto por ESLint) e indicar cuál será el carácter de fin de línea, esto más que todo para retrocompatibilidad con los sistemas Windows.
- Establecer el formateador de código de Visual Studio Code como Prettier.
- Finalmente si deseas añadir una coma al final de algunas estructuras de JavaScript puedes indicarle a Prettier que lo haga, esta es una buena práctica cuando se utiliza git y cada línea nueva no necesita añadir una coma en el elemento anterior.

## Formatear código fuente

Necesitamos instalar un plugin para especificar que configuración utilizará Prettier:

```
npm install -D eslint-config-prettier
```

Finalmente modificar nuevamente el archivo de `.eslint.json` de la siguiente manera:

```
  "extends": ["airbnb-base", "prettier"],
```

## Depuración con Visual Studio Code

Visual Studio Code ofrece la opción de depurar directamente en el editor donde tenemos el código fuente, lo cual es bastante útil, para llevar a cabo esto se debe crear el archivo de configuración:

```
touch .vscode/launch.json
```

Con la siguiente información:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "attach",
      "name": "Node: Attach to process",
      "processId": "${command:PickProcess}",
      "restart": true,
      "protocol": "inspector"
    }
  ]
}
```

Luego modificamos la sección de `scripts` del archivo `package.json` para añadir la opción que permitirá hacer *debug* al *nodemon*:

```
"scripts": {
  "dev": "nodemon --inspect",
  "start": "node index",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Siempre que se haga un cambio en la configuración del archivo `package.json` se debe reiniciar la aplicación una vez más para tomar los últimos cambios, lamentablemente Nodemon no observa los cambios en este archivo de configuración.

Antes de finalizar hacemos *commit* de nuestro trabajo:

```
git add --all
git commit -m "Add Visual Studio Code configuration"
```

Más información:

- VS Code - Nodemon[7]

[7] https://github.com/Microsoft/vscode-recipes/tree/master/nodemon

# Express JS

Express JS es un librería con un alto nivel de popularidad en la comunidad de Node.js e inclusive esta misma adoptó Express JS[8] dentro de su ecosistema, lo cual le brinda un soporte adicional, pero esto no quiere decir que sea la única o la mejor, aunque existen muchas otras librerías y frameworks para crear REST API, Express JS si es la más sencilla y poderosa, según la definición en la página Web de Express JS[9]: "Con miles de métodos de programa de utilidad HTTP y *middleware* a su disposición, la creación de una API sólida es rápida y sencilla."

## Añadir Express JS

Instalamos el módulo de `express` como una dependencia de nuestra aplicación:

```
1 npm install -S express
```

Esto añadirá una nueva entrada en nuestro archivo `package.json` en la sección de `dependencies` indicando la versión instalada.

Creamos un directorio llamado `server` donde se almacenarán todos los archivos relacionados con el servidor Web así como su nombre lo indica:

```
1 mkdir server
2 touch server/index.js
```

Tomaremos como ventaja la convención de Node.js en la nomenclatura de los módulos de la raíz del directorio llamándolos `index.js`

Tomaremos el ejemplo de Hello World[10] que se encuentra en la documentación de Express JS y reemplazamos el contenido del archivo por el siguiente código:

[8] https://nodejs.org/en/blog/announcements/foundation-express-news/
[9] http://expressjs.com/
[10] https://expressjs.com/en/starter/hello-world.html

```
// server/index.js

const express = require('express');

const port = 3000;

const app = express();

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(port, () => {
  console.log(`Example app listening on port ${port}!`);
});
```

Como lo vemos es muy similar al Servidor Web anterior pero con una sutiles diferencias:

- Utilizamos la librería de `express` en vez del módulo `http`.
- Creamos una aplicación de `express` (inclusive se pueden crear varias) y la almacenamos en la variable `app`.
- Esta vez solo aceptaremos peticiones en la raíz del proyecto (`/`) con el verbo GET de HTTP.

Si el servidor se está ejecutando lo detenemos con `CTRL+C`, lo ejecutamos nuevamente con el comando `node server/index` para ejecutar específicamente este último código y abrimos el navegador en la siguiente dirección `http://localhost:3000/`. Una vez lo hayamos comprobado volvemos a ejecutar la aplicación en modo de desarrollo.

Aparentemente vemos el mismo resultado, pero `express` nos ha brindado un poco más de simplicidad en el código y ha incorporado una gran cantidad de funcionalidades, las cuales veremos más adelante.

```
git add --all
git commit -m "Add Express JS"
```

# Independizar la aplicación

Antes de continuar vamos a organizar nuestra aplicación, crearemos diferentes módulos cada uno con su propósito específico, ya que esto permite separar la responsabilidad de cada uno de ellos, organizarlos en directorios para agrupar los diferentes módulos con funcionalidad común, así nuestro archivo `index.js` será más ligero, legible y estructurado.

NO es una buena práctica colocar todo el código en un solo archivo que sería difícil de mantener, esto se le conoce como aplicación monolítica.

Empezamos organizando la aplicación de Express JS:

```
// server/index.js

const express = require('express');

const app = express();

app.get('/', (req, res) => {
  res.send('Hello World');
});

module.exports = app;
```

Como podemos observar el módulo anterior tiene una sola responsabilidad, que es crear y exportar nuestra aplicación, la cual será independiente a la librería que utilizamos, pues cuando la importemos solo será `app` no `express`.

Creamos el directorio y módulo básico de configuración:

```
mkdir -p server/config
touch server/config/index.js
```

En el archivo de configuración es donde centralizamos todas las configuraciones de la aplicación que hasta el momento tenemos, así cualquier cambio será en un solo archivo y no en varios, y finalmente lo exportamos como un módulo:

```
// server/config/index.js

const config = {
  server: {
    port: 3000,
  },
};

module.exports = config;
```

Para esta aplicación el `hostname` no es obligatorio, por lo cual lo omitimos en la configuración.

De vuelta en nuestro archivo principal, lo reescribimos quitándole las múltiples responsabilidades que tenía y dejándolo solo con la responsabilidad de iniciar cada uno de los componentes de la aplicación, utilizando los módulos creados posteriormente:

```
// index.js

const http = require('http');

const app = require('./server');
const config = require('./server/config');

const { port } = config.server;

const server = http.createServer(app);

server.listen(port, () => {
  console.log(`Server running at port: ${port}`);
});
```

La función `createServer` de la librería `http` es compatible con la versión del servidor de nuestra aplicación (`app`), esto es muy conveniente pues si en el futuro fuera a incorporar un servidor `https`, solo es incluir la librería y duplicar la línea de creación del servidor.

Cada vez que existe un cambio en la configuración es necesario reiniciar el servidor, presionamos `CTRL+C` en la Terminal e iniciamos nuevamente en modo desarrollo con `npm run dev`.

Si visitamos el navegador en la dirección `http://localhost:3000` todo debe estar funcionando sin ningún problema.

Antes de finalizar hacemos commit de nuestro trabajo

```
git add --all
git commit -m "Modularize Express and Add config file"
```

Más información:

- Express JS[11]

[11] http://expressjs.com/

# Configuración y variables de entorno

## Cross-env

Nuestra aplicación no siempre se ejecutará en ambiente de desarrollo, por lo cual debemos establecer cuál es el ambiente en que se está ejecutando y dependiendo de ello podemos utilizar diferentes tipos de configuración e incluso conectarnos a diferentes fuentes de datos, para ello vamos a utilizar la variable de entorno `process.NODE_ENV`, que puede ser accedida desde cualquier parte de nuestra aplicación por el objeto global `process`.

Es muy importante garantizar la compatibilidad entre los diferentes sistemas operativos ya que cada sistema establece las variables a su manera, para asegurar que funcione en todos utilizaremos una librería de *npm* llamada `cross-env`, la instalamos como una dependencia de nuestra proyecto y así aseguramos que siempre esté disponible en nuestra aplicación:

```
npm i -S cross-env
```

Finalmente para utilizarla modificaremos nuestros *npm scripts* en el archivo `package.json` de la siguiente manera:

```
"scripts": {
  "dev": "cross-env NODE_ENV=development nodemon --inspect",
  "start": "cross-env NODE_ENV=production node index",
  "test": "cross-env NODE_ENV=test echo \"Error: no test specified\"&& exit 1"
},
```

Antes de finalizar hacemos commit de nuestro trabajo:

```
git add --all
git commit -m "Add cross-env and setting the environments"
```

Más información acerca de:

- process[12]
- cross-env[13]

[12] https://nodejs.org/api/process.html
[13] https://www.npmjs.com/package/cross-env

## Dotenv

En este momento nuestras variables de configuración están almacenadas en un archivo JavaScript y sus valores están marcados en el código fuente de la aplicación, lo cual NO es una buena práctica pues se estaría exponiendo datos sensibles. La idea es extraer la información sensible y tenerla en archivos separados por entorno pues no es la misma base de datos de desarrollo que la de producción.

Ahora debemos distinguir entre la configuración de la aplicacion y la del usuario, la primera son valores que necesita la aplicación para funcionar como: numero del puerto, la dirección IP y demás, la segundas son más relacionadas con la personalización de la aplicación como: número de objetos por página, personalización de la respuesta al usuario, etc.

Utilizaremos una librería que nos permite hacer esto muy fácil llamada `dotenv`, que nos carga todas las variables especificadas en un archivo al entorno de Node.js (`process.env`), a continuación creamos el archivo de configuración en la raíz del proyecto llamado `.env` y le colocamos el siguiente contenido:

```
SERVER_PORT=3000
```

Instalamos la librería:

```
npm i -S dotenv
```

Procedemos a modificar nuestro archivo de configuración:

```
// server/config/index.js

require('dotenv').config('');

const config = {
  server: {
    port: process.env.SERVER_PORT,
  },
};

module.exports = config;
```

Como podemos ver por defecto la librería `dotenv` buscará el archivo llamado `.env`, el cual cargará las variables y su valores en el entorno del proceso donde se está ejecutando Node.js. Así nuestra configuración del sistema no estará más en el código fuente de la aplicación y adicional podemos reemplazar el archivo de configuración en cada entorno donde se publique la aplicación.

Antes de terminar un aspecto muy importante es NO incluir el archivo de configuración `.env` en el repositorio de *git*, pues esto es justo lo que queremos evitar, que nuestra configuración y valores no quede en el código fuente de la aplicación, para ello modificamos el archivo `.gitignore` de la siguiente forma:

```
node_modules/
.DS_Store
.Thumbs.db
.env
```

Como no estamos guardando este archivo en nuestro repositorio, crearemos una copia de ejemplo para que el usuario pueda renombrarlo y utilizarlo:

```
cp .env .env.example
```

Antes de finalizar hacemos commit de nuestro trabajo:

```
git add --all
git commit -m "Add .env file and update the configuration file"
```

Más información acerca de:

- dotenv[14]

[14] https://github.com/motdotla/dotenv

# Middleware para manejo de errores

Hasta el momento la única **ruta** que tenemos configurada en nuestra aplicación es: `/` que equivale a la raíz de la aplicación, pero ¿Qué sucede si tratamos de acceder otra ruta?. Pues en este momento no tenemos configurado como manejar esta excepción, en términos de *express* no existe ningún "*Middleware*" que la procese, por lo tanto terminará en un error no manejado por el usuario.

Si vamos al navegador y tratamos de acceder a una ruta como `localhost:3000/posts` obtendremos un resultado generico cómo: `Cannot GET /posts` y un código HTTP 404 que indica que el recurso no fue encontrado, pero nosotros podemos controlar esto con *express*, primero analizemos el fragmento de código que define la ruta:

```
app.get('/', (req, res) => {
  res.send('Hello World');
});
```

1. La aplicación (`app`) adjunta una función *callback* para cuando se realice una petición a la ruta `/` con el método GET.
2. Esta función *callback* recibe dos argumentos: el objeto de la petición (*request*) abreviado `req` y el objeto de la respuesta (*response*) abreviado `res`.
3. Una vez se cumplen los criterios establecidos en la definición de la ruta la función de *callback* es ejecutada.
4. Se envía la respuesta por parte del servidor (`res.send('Hello world')`) el flujo de esta petición se detiene y el servidor queda escuchando por más peticiones.

## Express middleware

"Las funciones *middleware* son funciones que tienen acceso al objeto de la solicitud (req), al objeto de respuesta (res) y a la función de que permite saltar al siguiente middleware (next).

(Tomado de Utilización de middleware[15])

Esto quiere decir que nuestro *callback* en realidad es una función *middleware* de *express* y tiene la siguiente firma de parámetros:

[15] http://expressjs.com/es/guide/using-middleware.html

```
app.get('/', (req, res, next) => {
  res.send('Hello World');
});
```

Dónde `next` es otra función que al invocarse permite continuar el flujo que habíamos mencionado antes para que otras funciones *middleware* puedan continuar con el procesamiento de la petición.

Esto quiere decir que podemos agregar otro *middleware* al final del flujo para capturar si la ruta no fue procesada por ningún *middleware* definido anteriormente:

```
// server/index.js

const express = require('express');

const app = express();

app.get('/', (req, res, next) => {
  res.send('Hello World');
});

// No route found handler
app.use((req, res, next) => {
  res.status(404);
  res.json({
    message: 'Error. Route not found',
  });
});

module.exports = app;
```

Nótese varias cosas importantes en el fragmento de código anterior:

1. Se ha agregado la función `next` al middleware de la ruta raíz, pero realmente no lo estamos utilizando dentro de la función, esto es con el fin de hacer concordancia con la firma que tienen las funciones _middleware _(Más adelante la editaremos)
2. Le hemos indicado a la aplicación (`app`) utilizar una función *middleware*, nuevamente podemos saber que es una función *middleware* de *express* por su firma, revisando los parámetros.
3. Estamos estableciendo el código HTTP de la respuesta con `res.status(404)` el cual es equivalente a una página no encontrada, en este caso una ruta o recurso.
4. Cuando un servidor Web responde debe indicar el tipo de contenido que está devolviendo al cliente (por ejemplo: Navegador Web) para que esté a su vez pueda interpretarlo y mostrarlo al usuario, esto se indica en la respuesta mediante el encabezado `Content-type`, por defecto el valor es `text/plain` que se utiliza para texto plano o `text/html` para código HTML, cada uno de

estos valores se conoce como el *MIME Type* de los archivos. En este caso estamos respondiendo explícitamente un objeto tipo JSON como lo indicamos en la función `res.json()`, no tenemos necesidad de establecer el `Content-Type`, pues *express* lo hace automáticamente por nosotros a `application/json` y adicionalmente convierte el objeto literal de JavaScript a un objeto JSON como tal.

Guardamos el archivo (*nodemon* reiniciará nuestra aplicación automáticamente ) y ahora si vamos nuevamente al navegador e invocamos la ruta `localhost:3000/posts` en el cual obtendremos el mensaje que establecimos en el *middleware* que se convierte en la última función *middleware* de todos las declaradas anteriormente en esta aplicación de *express*:

```
{
  message: 'Error. Route not found'
}
```

Para hacer una analogía a la definición de los *middleware* es como una cola de funciones donde la petición se revisa comenzando desde la primera función, si el *middleware* coincide con la petición este detiene el flujo, procesa la petición y puede hacer dos cosas: dar respuesta a la petición y en este caso se detiene la revisión por completo o dejar pasar la petición al siguiente *middleware* esto se hace con la función *next*, o de lo contrario si ninguna coincide llegará al final que es el *middleware* genérico para capturar todas las peticiones que no se pudieron procesar antes.

# Manejo de errores

Ahora surge la siguiente pregunta, ¿Cómo podemos controlar los errores que sucedan en las diferentes rutas?, *express* permite definir una función middleware con una firma de parámetros un poco diferente a las anteriores, agregamos el siguiente código al final del archivo:

```
// server/index.js

...

// Error handler
app.use((err, req, res, next) => {
  const { statusCode = 500, message } = err;

  res.status(statusCode);
  res.json({
    message,
  });
});

module.exports = app;
```

Los tres puntos (...) es una convención para indicar el resto del contenido del archivo

Podemos ver varias cosas en el código anterior:

1. El *middleware* que captura errores en *express* comienza por el parámetro de error abreviado como `err`
2. Obtenemos el `statusCode` del objeto `err`, si no trae ninguno, lo establecemos en 500 por defecto que significa *Internal Server Error*, así como el `message` que este trae.
3. Finalmente establecemos el `StatusCode` en la respuesta (`res`) y enviamos de vuelta un JSON con el mensaje del error.

Ahora cada vez que invoquemos un error en cualquier *middleware* tendremos uno que lo capture y procese, asi podríamos aprovechar esto, por ejemplo, para guardar en los *logs* información relevante del error.

Para terminar cambiamos nuevamente la definición de la ruta raíz de la siguiente manera:

```
// server/index.js

const express = require('express');

const app = express();

app.get('/', (req, res, next) => {
  res.json({
    message: 'Welcome to the API',
  });
});

...
```

Nótese que ahora estamos devolviendo un archivo JSON, pues estamos creando un REST API Web.

Antes de continuar guardemos nuestro progreso:

```
git add --all
git commit -m "Add middleware for not found routes and errors"
```

# Sistema de Logs

## Utilizar Winston para los logs

Es importante para cualquier aplicación que va a salir a producción tener un sistema de *logs*, pues cuando estamos en desarrollo tenemos el control total y es muy fácil investigar para averiguar si existe algún error, lo que no sucede cuando la aplicación está ejecutándose en producción, por lo tanto el sistema de *logs* es una manera eficaz de dejar rastro de las operaciones, advertencias o errores que puedan suceder en la aplicación.

Existen muchas librerías que son de mucha utilidad así como *express*, entre ellos está Winston, el cual nos permite crear un *log* personalizado con múltiples métodos para manejar los *logs* desde imprimirlos en la consola hasta guardarlos en archivos.

Procedemos a instalar las dependencias

```
npm install -S winston
```

Debemos crear un nuevo archivo para establecer la configuración de los *logs*, estoy es muy importante si en el día de mañana se quiere cambiar de configuración o inclusive de librería el resto de la aplicación no debería verse afectada.

Creamos el archivo con su respectivo contenido:

```
// server/config/logger.js

const { createLogger, format, transports } = require('winston');

// Setup logger
const logger = createLogger({
  format: format.simple(),
  transports: [new transports.Console()],
});

module.exports = logger;
```

Como pueden observar hemos decidido para este proyecto imprimir los *logs* en la consola, pero como mencionamos antes esta librería nos permite almacenarlos en archivos, si así lo deseamos.

```
// server/index.js

const express = require('express');

const logger = require('./config/logger');

...


// No route found handler
app.use((req, res, next) => {
  const message = 'Route not found';
  const statusCode = 404;

  logger.warn(message);

  res.status(statusCode);
  res.json({
    message,
  });
});

// Error handler
app.use((err, req, res, next) => {
  const { statusCode = 500, message } = err;

  logger.error(message);

  res.status(statusCode);
  res.json({
    message,
  });
});

module.exports = app;
```

Como podemos observar en el fragmento de código anterior incluimos nuestro *logger* y lo utilizamos en los *middleware.*

Antes de continuar guardemos nuestro progreso:

```
git add --all
git commit -m "Add Winston to create a log system"
```

Más información:

- Winston[16]

## Utilizar morgan para hacer logs de las peticiones

La librería anterior nos permitió crear un *log* personalizado, pero también debemos registrar el acceso a todas las peticiones dándonos un detalle de lo que se está recibiendo en el servidor, para esto vamos a utilizar la librería de *morgan*, procedemos a instalarla de la siguiente manera:

```
npm install -S morgan
```

Morgan finalmente es un middleware que tomará las peticiones y las imprimirá según nosotros le indiquemos:

```
// server/index.js

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

const logger = require('./config/logger');

const app = express();

// Setup middleware
app.use(morgan('combined'));

...
```

Incluimos *morgan* como un middleware, probamos accediendo a diferentes URL y vemos el resultado en la consola, pero tenemos dos salidas de logs: una producida por *winston* y otra por *morgan*, que casualmente ambas están saliendo a la consola por lo tanto añadimos la siguiente configuración adicional:

[16] https://www.npmjs.com/package/winston

```
// server/index.js

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

const logger = require('./config/logger');

const app = express();

// Setup middleware
app.use(
  morgan('combined', { stream: { write: (message) => logger.info(message) } })
);

...
```

Resulta y pasa que por defecto *morgan* coloca sus *logs* en la consola, pero nosotros le estamos indicando que utilice el *logger* que creamos con *winston*, finalmente irán a la consola, pero ahora es una sola librería que administra donde quedarán almacenados los *logs*, ya que si decidimos guardarlos en archivos, nuevamente el cambio sería solamente en una sola parte.

Si abrimos el navegador e invocamos diferentes rutas en la dirección del navegador Web podremos ver en la consola las diferentes peticiones que llegan al servidor y también los mensajes personalizados que imprimimos en los *middleware*.

Antes de continuar guardemos nuestro progreso:

```
git add --all
git commit -m "Add morgan to log the app requests"
```

Más información:

- Morgan[17]

# Hacer seguimiento a las peticiones

Nuevamente cuando nuestra aplicación está ejecutándose en producción llegan muchísimas peticiones al mismo tiempo y si deseamos identificar una en particular va a hacer muy difícil, para ello debemos identificar cada uno de ellas, utilizaremos una librería llamada *express-request-id* que añade a la petición entrante (*req*) un campo llamado *id* donde su valor es un número único y en la petición de respuesta (*res*) establece el encabezado *X-Request-Id* con el mismo valor para que el cliente pueda identificar la petición.

procedemos a instalarla de la siguiente manera:

[17] https://www.npmjs.com/package/morgan

```
npm install -S express-request-id
```

La incluimos en nuestro archivo del servidor:

```
// server/index.js

const express = require('express');
const morgan = require('morgan');
const requestId = require('express-request-id')();

const logger = require('./config/logger');

// Init app
const app = express();

// Setup middleware
app.use(requestId);
app.use(
  morgan('combined', { stream: { write: (message) => logger.info(message) } })
);

...
```

Al redirigir los mensajes de morgan a winston las peticiones siempre agregan un final de línea, por lo tanto vamos a removerlo con la siguiente librería:

```
npm i -S strip-final-newline
```

Ahora debemos incluirla en nuestro *log* de *morgan*, por lo tanto vamos a mover morgan a nuestro archivo de *logger* y lo editamos de la siguiente manera:

```
// server/config/logger.js

const { createLogger, format, transports } = require('winston');
const morgan = require('morgan');
const stripFinalNewline = require('strip-final-newline');

// Setup logger
const logger = createLogger({
  format: format.simple(),
  transports: [new transports.Console()],
});
```

```
12
13 // Setup requests logger
14 morgan.token('id', req => req.id);
15
16 const requestFormat = ':remote-addr [:date[iso]] :id ":method :url" :status';
17 const requests = morgan(requestFormat, {
18   stream: {
19     write: (message) => {
20       // Remove all line breaks
21       const log = stripFinalNewline(message);
22       return logger.info(log);
23     },
24   },
25 });
26
27 // Attach to logger object
28 logger.requests = requests;
29
30 module.exports = logger;
```

Explicamos a continuación los cambios:

- Le indicamos a *morgan* como va a calcular el token `:id`, que como vimos anteriormente nuestra nueva librería lo agrega al objeto `req`.
- Establecemos un formato personalizado basado en el `combined` que trae por defecto y añadimos el token `:id`.
- Eliminamos todos los saltos de línea y guardamos toda la configuración de morgan en la variable `requests`.
- Finalmente guardamos dentro de *logger* la configuración de *morgan*, pues recordemos que debemos añadirla como *middleware* de express en el archivo del servidor.

Actualizamos nuevamente nuestro archivo del servidor:

```
1 // server/index.js
2
3 const express = require('express');
4 const requestId = require('express-request-id')();
5
6 const logger = require('./config/logger');
7
8 // Init app
9 const app = express();
```

```
10
11 // Setup middleware
12 app.use(requestId);
13 app.use(logger.requests);
14
15 ...
```

Antes de continuar guardemos nuestro progreso:

```
1 git add --all
2 git commit -m "Add Request id to identify the request with morgan"
```

Más información:

- express-request-id[18]

## Añadir formato de la petición a los logs

En este momento los logs generados por morgan al recibir las peticiones tienen una información adicional de la petición como: Dirección IP del cliente, fecha y hora, el método HTTP y la URL, a continuación añadiremos una función que reciba la petición como parámetro y nos devuelva la misma información:

```
1 // server/config/logger.js


25 ...
26
27 // Attach to logger object
28 logger.requests = requests;
29
30 // Format as request logger and attach to logger object
31 logger.header = (req) => {
32   const date = new Date().toISOString();
33   return `${req.ip} [${date}] ${req.id} "${req.method} ${req.originalUrl}"`;
34 };
35
36 module.exports = logger;
```

A continuación vamos a realizar las siguientes modificaciones en nuestro archivo del servidor en los dos middleware asociados a errores:

[18] https://github.com/floatdrop/express-request-id

```
// server/index.js


...

// No route found handler
app.use((req, res, next) => {
  next({
    message: 'Route not found',
    statusCode: 404,
    level: 'warn',
  });
});

// Error handler
app.use((err, req, res, next) => {
  const { message, statusCode = 500, level = 'error' } = err;
  const log = `${logger.header(req)} ${statusCode} ${message}`;

  logger[level](log);

  res.status(statusCode);
  res.json({
    message,
  });
});

module.exports = app;
```

Lo primero que podemos observar es que ahora el *middleware* que nos captura cuando una ruta no es encontrada está invocando el siguiente *middleware* (`next`) con el información del error y finalmente en procesado por nuestro middleware asociado con los errores, donde creamos la cadena de log basada en la función que acabamos de crear, una nueva modificación es el `type` que por defecto es `error` si esta no es enviada en el objeto de `error`.

Si vamos al navegador e invocamos una dirección que no existe sobre nuestra API podemos ver el resultado en los *logs*.

Antes de continuar guardemos nuestro progreso

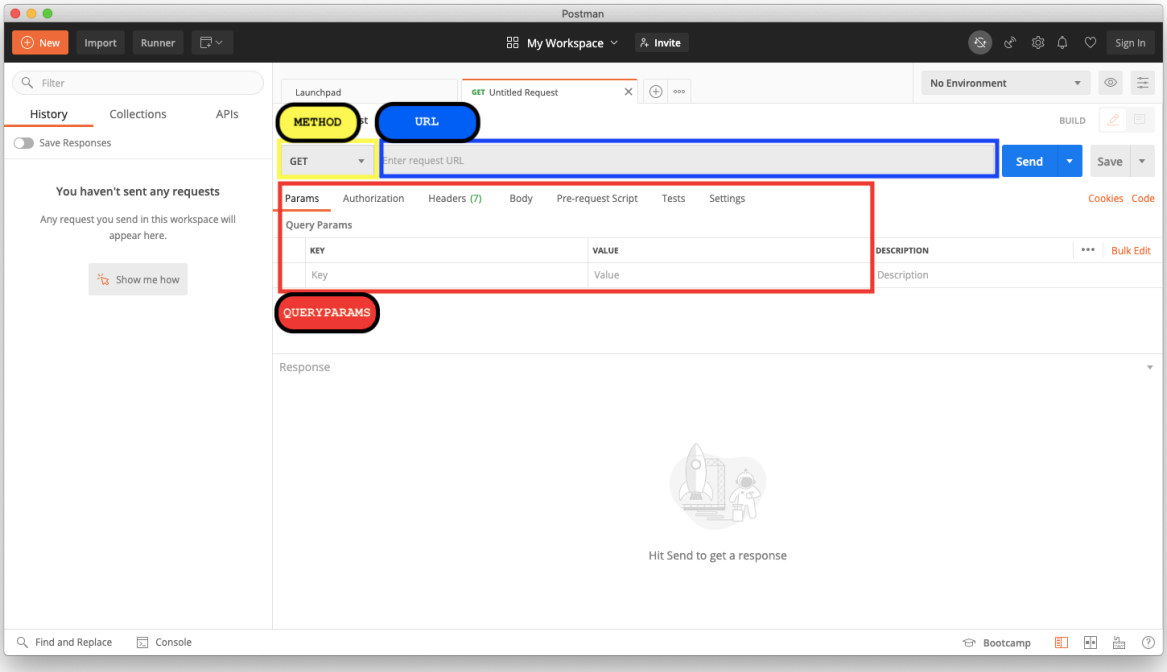
```
git add --all
git commit -m "Add log header from requests"
```

# Postman

Hasta el momento hemos podido probar nuestra REST API Web a través del navegador Web digitando la url en la barra de direcciones, cuando lo hacemos el navegador utiliza el verbo GET, pero no siempre será así, pues no vamos a poder realizar todas las peticiones a nuestra API con todos los verbos necesario del contrato REST, para poder llevar esto acabo utilizaremos una herramientas de escritorio llamada POSTMAN, la cual ofrece una interfaz gráfica para realizar todos las peticiones (*requests*) y ver las respectivas respuesta (*responses*).

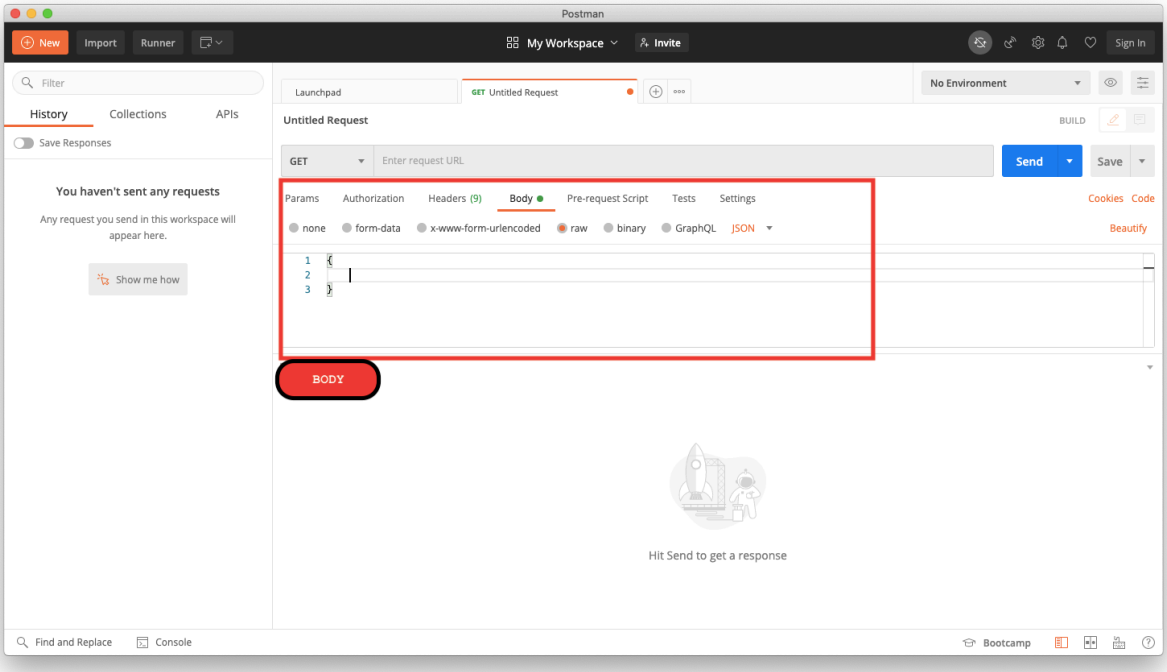
Si no es posible instalar el software en el sistema existe otra opción que lo emula en una pestaña del navegador Web llamada POSTMAN tabbed.

La siguiente imagen muestran donde seleccionar el método, la *url* y si la petición lo requiere los *queryparams*.



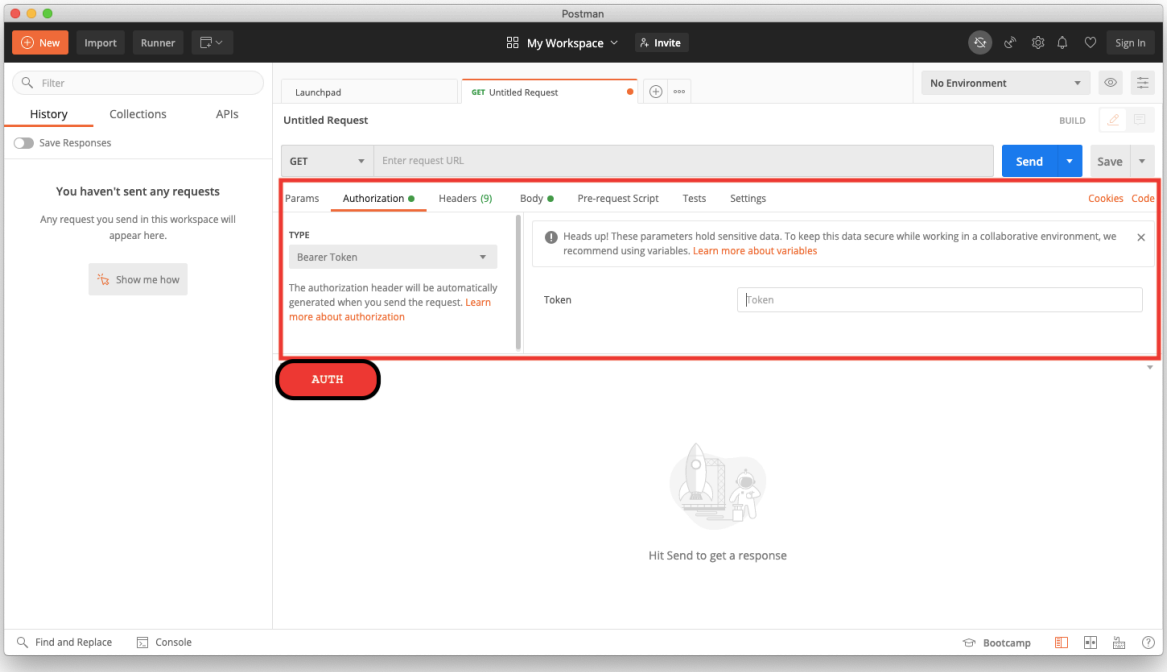

**Postman**

También si es necesario se pueden enviar datos (*body*) en la petición, utilizaremos siempre formato JSON.




**Postman**

Finalmente para algunas peticiones necesitaremos que tipo y donde colocar el *token* de autorización.




**Postman**

Más información:

- POSTMAN[19]
- POSTMAN Tabbed[20]
- POSTMAN - Tutoriales[21]

[19] https://www.getpostman.com/
[20] https://chrome.google.com/webstore/detail/tabbed-postman-rest-clien/coohjcphdfgbiolnekdpbcijmhambjff?hl=en
[21] https://www.getpostman.com/resources/videos-tutorials/

# Router y Routes de Express

## Definir las rutas con Route

Como ya vimos anteriormente hemos definido la ruta para la raíz de nuestra API en el archivo :

```
13 // server/index.js
14 ...
15
16 app.get('/', (req, res, next) => {
17   res.json({
18     message: 'Welcome to the API',
19   });
20 });
21
22 ...
```

De la misma manera podemos definir todas las rutas que necesitamos para nuestra API, modificamos la definición anterior de la ruta raíz:

```
13 // server/index.js
14 ...
15
16 app.get('/api/tasks', (req, res, next) => {
17   res.json({
18     message: 'GET all tasks',
19   });
20 });
21
22 ...
```

De esta manera se pueden definir todas las rutas que necesitemos y dentro de cada una de ellas hacer todo el procesamiento que necesitemos, pero imaginemos por un momento que hemos definido 10 rutas más con diferentes verbos HTTP para las diferentes operaciones relacionadas con este recurso y por algún motivo nos toca cambiar el nombre del recurso de `/api/tasks` a `/api/messages/` nos tocaría modificar todas las rutas, lo cual no es muy dinámico he inclusive se puede incurrir en errores, para esto *express* tiene un objeto llamado ***Route*** el cual permite establecer un punto de montaje para el recurso y adjuntarle todas las operaciones con los verbos que queramos, entonces tomando

nuevamente el ejemplo anterior solo nos tocaría modificar el recurso en el punto de montaje y no en todas las rutas.

Veamos cómo se utiliza realizando los cambios:

Cambiar la definición que habíamos escrito anteriormente para obtener todas los items del recurso introduciendo el `route` de *express*:

```
// server/index.js
...

app.route('/api/tasks')
  .get((req, res, next) => {
    res.json({
    message: 'GET all tasks',
  });
});

...
```

Tenemos un cambio significativo en la estructura de la definición, el punto de montaje se establece con el objeto `route` y le "anexamos" el método GET con el *middleware* correspondiente, pero también se le puede anexar diferentes métodos como: GET, POST, PUT, DELETE, etc. Lo interesante es que solo dependen del primer parámetro: el nombre de la ruta.

Creamos un directorio que contendrá todos los directorios y archivos relacionados con el API:

```
mkdir -p server/api
touch server/api/index.js
```

Editamos el último archivo creado `server/api/index.js` , cortamos la definición de la ruta del archivo `server/index.js` y lo convertimos en un módulo con el siguiente contenido:

```
// server/api/index.js

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

router.route('/tasks')
  .get((req, res, next) => {
    res.json({
      message: 'GET all tasks'
    });
  });

module.exports = router;
```

En el fragmento de código anterior estamos ejecutando e importando solamente el *Router,* no toda la librería de *express,* aunque este proceso se puede hacer en dos líneas de código también.

Modificamos el archivo del servidor importando el módulo definido anteriormente y reemplazando la definición de la ruta por el punto de montaje:

```
1  // server/index.js
2
3  const express = require('express');
4  const requestId = require('express-request-id')();
5
6  const logger = require('./config/logger');
7  const api = require('./api');
8
9  // Init app
10 const app = express();
11
12 // Setup middleware
13 app.use(requestId);
14 app.use(logger.requests);
15
16 // Setup router and routes
17 app.use('/api', api);
18
19 ...
```

Hemos realizado unos cambios muy importantes, en cuanto a organización, responsabilidad y modularidad:

1. Hemos creado un `Router` para manejar todas las operaciones relacionadas con el recurso de *tasks*, pero estas rutas a su vez son independientes del prefijo que se vaya a utilizar, es decir, ya no tiene marcado en el código antes de cada ruta el prefijo `/api`.
2. Hemos creado el punto de montaje `/api` pero esta vez le hemos adjuntado el *Router* que contiene todas las operaciones del recurso `/tasks` (por el momento), es decir que al final para acceder al recurso sigue siendo `/api/tasks`.

Si podemos visualizarlo, esto nos brinda una enorme ventaja a la hora de organizar los recursos y renombrarlos.

Ahora si vamos al navegador a la dirección `localhost:3000/api/tasks` obtenemos el siguiente resultado:

```
{
  "message": "Get all tasks"
}
```

Antes de continuar guardamos nuestro progreso:

```
git add --all
git commit -m "Setting API mount point and basic router for Tasks"
```

Más información:

- Express Routing[22]

## Manejo de versiones de API

Es normal que un REST API tenga diferentes versiones en el tiempo, pues surgen nuevas funcionalidades o cambios importantes, pero debemos asegurar cuando vayamos a realizar estos cambios no dañar la versión actual del API que ya está publicada, ya que la pueden estar utilizando muchas personas en ese momento. Esto no sucede con otros paradigmas actuales como *GraphQL* que en vez de rutas maneja un lenguaje donde el cliente es el que estructura la respuesta del servidor y realmente no existen rutas estática definidas.

Con los cambios que introducimos anteriormente será más sencillo manejar las versiones del API.

Es recomendable siempre versionar el API desde la creación de la misma, para lo cual ejecutamos los siguientes comandos:

```
mkdir -p server/api/v1
mv server/api/index.js server/api/v1/index.js
```

Finalmente cambiamos el punto de montaje de nuestra API en el archivo del servidor:

```
// server/index.js

const express = require('express');
const requestId = require('express-request-id')();

const logger = require('./config/logger');
const api = require('./api/v1');
```

[22] http://expressjs.com/es/guide/routing.html

```
16 // Setup router and routes
17 app.use('/api', api);
18 app.use('/api/v1', api);
19
20 ...
```

Este cambio nos permite utilizar dos puntos de montaje para el mismo API. Normalmente la última versión del API se utiliza bajo el prefijo `/api` solamente, y si el usuario quiere utilizar alguna versión específica le coloca el prefijo `/api/v1`. Aprovechando al máximo la independencia que provee el *Router* de *express*.

Si en el futuro lanzamos la versión 2 del API solo tendríamos que crear el directorio llamado v2 (al mismo nivel que v1) y asociarlo con el punto de montaje`/api/v2` y a su vez `/api`, garantizando así que los usuarios que desean acceder a la versión "legacy" (Versión 1) de nuestra API sería exclusivamente con el prefijo`/api/v1`.

Antes de continuar guardamos nuestro progreso:

```
1 git add --all
2 git commit -m "Versioning API"
```

Más información:

- GraphQL[23]

[23] https://graphql.org/