Índice

[**¿Qué es Express.js?** 1](#_Toc101638198)

[**Configuración del entorno de desarrollo** 1](#_Toc101638199)

[**Git Ignore** 1](#_Toc101638200)

[**Linters** 1](#_Toc101638201)

[**Nodemon** 2](#_Toc101638202)

[**gitignore.io⠀servicios** 2](#_Toc101638203)

[**Routing o rutas con Express.js** 2](#_Toc101638204)

[**¿Qué es una RESTful API?** 2](#_Toc101638205)

[**GET: recibir parámetros** 4](#_Toc101638206)

[**¿Qué es un endpoint?** 4](#_Toc101638207)

[**Recibir parámetros en un endpoint** 5](#_Toc101638208)

[**GET: parámetros query** 5](#_Toc101638209)

[**Parametros Query** 5](#_Toc101638210)

[**Separación de responsabilidades con express.Router** 7](#_Toc101638211)

[**¿Qué es un middleware?** 8](#_Toc101638212)

[**App.use** 8](#_Toc101638213)

[**Instalación de Postman o Insomnia** 9](#_Toc101638214)

[**Insomnia** 9](#_Toc101638215)

[**Postman** 10](#_Toc101638216)

[**POST: método para crear** 11](#_Toc101638217)

[**Diferencias entre GET y POST** 11](#_Toc101638218)

[**Lectura del método POST desde nuestra API** 11](#_Toc101638219)

[**PUT, PATCH y DELETE** 13](#_Toc101638220)

[**PATCH** 13](#_Toc101638221)

[**DELETE** 14](#_Toc101638222)

[**Códigos de estado o HTTP response status codes** 15](#_Toc101638223)

[**Listado de códigos de estados** 15](#_Toc101638224)

[**Introducción a servicios: crea tu primer servicio** 21](#_Toc101638225)

[**Entidades** 21](#_Toc101638226)

[**Casos de uso** 21](#_Toc101638227)

[**Controladores** 21](#_Toc101638228)

[**Flujo de trabajo** 22](#_Toc101638229)

[**Async await y captura de errores** 24](#_Toc101638230)

[**¿Qué son los Middlewares?** 24](#_Toc101638231)

[**Middleware para HttpErrors** 25](#_Toc101638232)

[**Manejo de errores con Boom** 27](#_Toc101638233)

[**Validación de datos con Joi** 27](#_Toc101638234)

[**Middlewares populares en Express.js** 28](#_Toc101638235)

[**Consideraciones para producción** 28](#_Toc101638236)

# **¿Qué es Express.js?**

**Es el framework web** más popular de **Node**, **y** **es la librería subyacente** para un gran número de otros frameworks web de Node populares. Proporciona mecanismos para:

* Escritura de manejadores de peticiones con diferentes verbos HTTP en diferentes caminos URL (rutas).
* Integración con motores de renderización de “vistas” para generar respuestas mediante la introducción de datos en plantillas.
* Establecer ajustes de aplicaciones web como qué puerto usar para conectar, y la localización de las plantillas que se utilizan para renderizar la respuesta.
* Añadir procesamiento de peticiones “**middleware**” adicional en cualquier punto dentro de la tubería de manejo de la petición.

# **Configuración del entorno de desarrollo**

## **Git Ignore**

* Son archivos ignorados que suelen ser artefactos de compilación y archivos generados por el equipo que pueden derivarse de tu fuente de repositorios o que no deberían confirmarse por algún otro motivo. Ejemplo: node\_modules.

## **Linters**

* Herramientas que realizan la lectura del código fuente.
* Detectan errores/warnings de código.
* Variables sin usar o no definida, llave sin cerrar

## **Nodemon**

* Es una herramienta que ayuda a desarrollar aplicaciones en Node.js al reiniciar la aplicación automáticamente cuando hay cambios en el archivo.

## [**gitignore.io**](https://www.toptal.com/developers/gitignore)**⠀**

* Contiene la configuración del entorno que específiques.

# **Routing o rutas con Express.js**

El routing es determinar cómo una aplicación responde a una solicitud de cliente a un punto final (endpoint) en particular y un método de solicitud HTTP específico.

La definición de la ruta adopta la siguiente estructura:



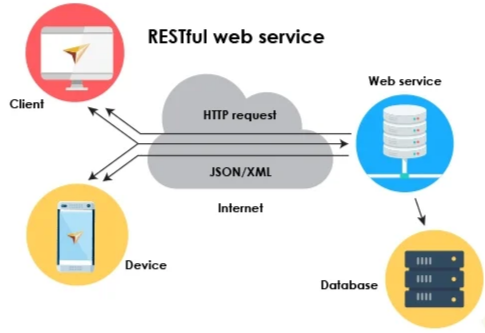
# **¿Qué es una RESTful API?**

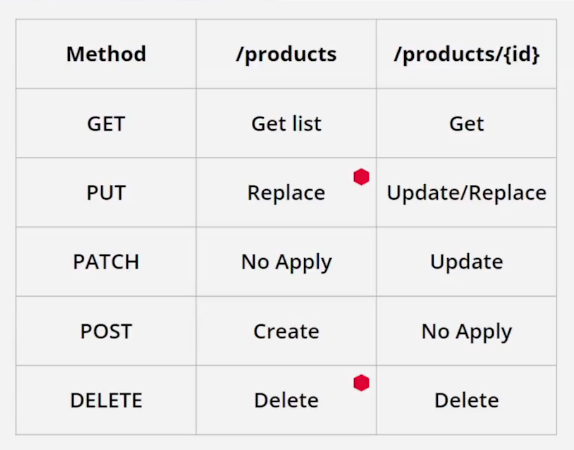
REST: Representational State Transfer.

Es una convección que se refiere a servicios web por protocolo HTTP

Métodos:

* **Get**: Obtener un recurso.
* **Put**: Modificar/Actualizar un recurso.
* **Patch**: Modificar/Actualizar parcialmente un recurso.
* **Post**: Crear un recurso.
* **Delete**: Eliminar recurso.



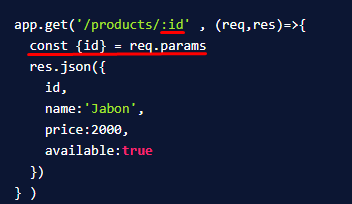


# **GET: recibir parámetros**

## **¿Qué es un endpoint?**

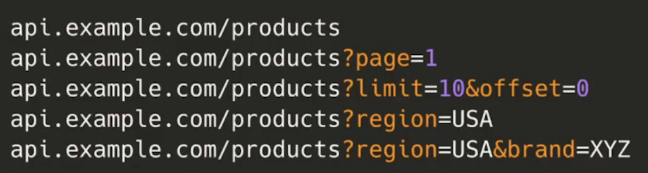
Son las URLs de una **API** o un **backend** que responden a una petición. Los mismos **entry points** tienen que calzar con un **endpoint** para existir. Algo debe responder para que se renderice un sitio con sentido para el visitante.

## **Recibir parámetros en un endpoint**

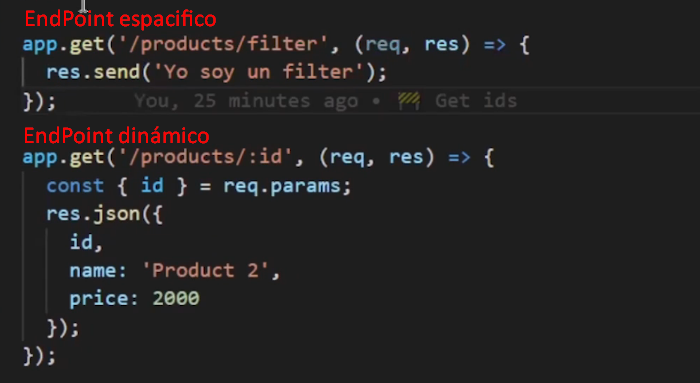


# **GET: parámetros query**

## **Parametros Query**



* Se utilizan para hacer filtros a la información que queremos de nuestra API.
* Puede ayudarnos a paginar, a establecer límites, incluso buscar palabras que coincidan con un valor.
* Los **endpoints específicos** deben declararse **antes** de los **endpoints** **dinámicos**. Esto se hace para que las rutas no se choquen en express.

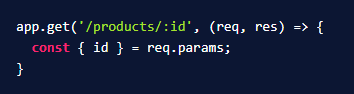


Al utilizar **app.get** ya podemos empezar a usar este verbo en nuestro endpoint.

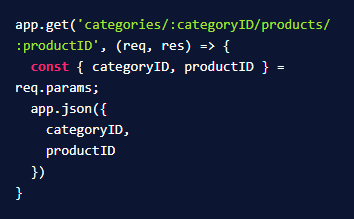
Al colocar **:** en la ruta, podemos tomar ese valor como parámetro:



Para tomar los parametros que se envían a través de la ruta, podemos usar el objeto **req.params** . Allí se almacenarán todos los parametros que enviemos en la url.



a partir de ese **id**, podemos acceder a otros endpoints a partir de allí. Por ejemplo:



# **Separación de responsabilidades con express.Router**

Crea un controlador(handler) de rutas **modulares** y **montables**. Una instancia de Router **es un sistema de enrutamiento y middleware completo,** por esa razón lo podemos tomar como si fuera una **mini app**.

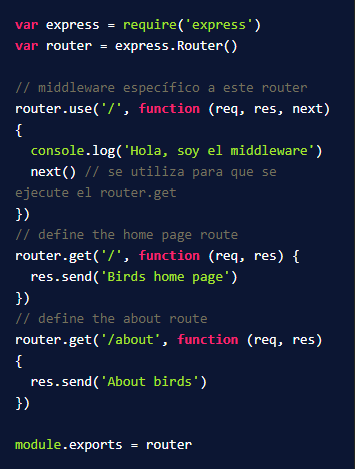
Cada módulo de nuestras rutas es una mini aplicación en la que creamos sus rutas independientes y podemos incluirle middlewares, que se ejecutarán cuando se coincida con el path.

## **¿Qué es un middleware?**

Un middleware es un bloque de código que se ejecuta entre la petición que hace el usuario (request) hasta que la petición llega al servidor.

## **App.use**

Lo que hace es montar un middleware en la ruta especificada. Por ejemplo:

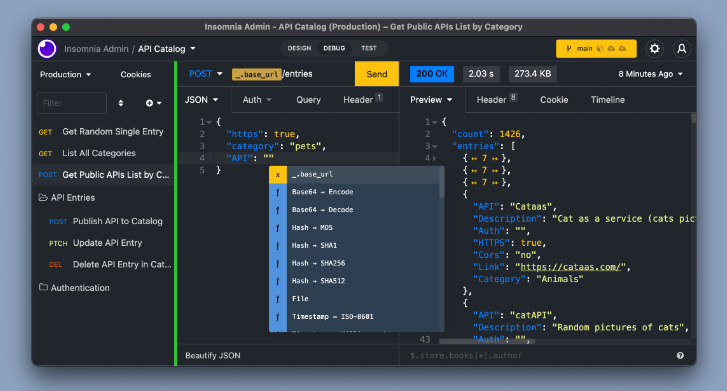


# **Instalación de Postman o Insomnia**

Para poder probar nuestra API de forma más productiva vamos a tener que usar algún **cliente de APIs** que nos permita hacer las pruebas de funcionamiento de lo que estás construyendo, los dos clientes más famosos son **Insomnia** y **Postman**.

## [**Insomnia**](https://insomnia.rest/download)

Como cliente para probar APIs tiene grandes características, destaca principalmente por una interfaz limpia y sencilla, pero a la vez potente, en donde puede configurar ambientes, exportar e importar, gran soporte con GraphQL, etc.



## [**Postman**](https://www.postman.com/downloads/)

Es uno de los más usados y legendariamente nos ha acompañado por mucho tiempo, además cuenta con características similares a Insomnia como:

* Exportar e importar,
* Ambientes.
* Entornos.
* Provee una API para hacer testing muy potente.

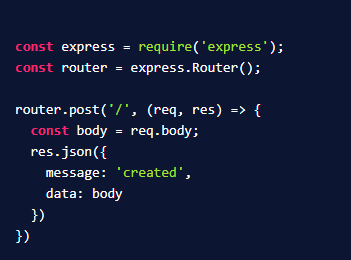
# **POST: método para crear**

## **Diferencias entre GET y POST**

* El método GET coloca los parámetros **dentro** de la URI mientras que el método POST los agrega al **cuerpo**.
* GET se utiliza esencialmente para **obtener** la información. En cambio, el objetivo del método POST es **actualizar** los datos.
* Los resultados de la consulta POST **no pueden marcarse**, mientras que los resultados de la consulta GET **pueden marcarse** porque existen en forma de URL.
* En el método GET **la información es visible** **en la URL** lo que aumenta las vulnerabilidades y el riesgo de hacking. Por el contrario, el método POST **no muestra variables** **en la URL** y también se pueden utilizar múltiples técnicas de codificación, lo que lo hace resistente.
* Cuando se utiliza el método GET en el formulario, **sólo se aceptan caracteres ASCII** en los tipos de datos. Por el contrario, el método POST **no enlaza tipos de datos de formulario** **y permite caracteres binarios y ASCII**.
* El tamaño variable en el método GET **es de aproximadamente 2000 caracteres**. A la inversa, el método POST **permite hasta 8 Mb de tamaño variable**.
* Los datos del método GET se pueden almacenar en [**caché**], mientras que los datos del método POST **no**.

## **Lectura del método POST desde nuestra API**

Lo primero que debemos hacer es crear la forma en como el servidor tratará la petición de tipo POST.

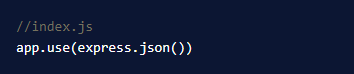


Luego usamos Insomnia y/o Postman, luego enviamos los datos con el siguiente formato json.



Veremos que nos aparece el json de respuesta, pero solamente con el valor de **message: 'created'** y no el cuerpo de la petición (la **data**).

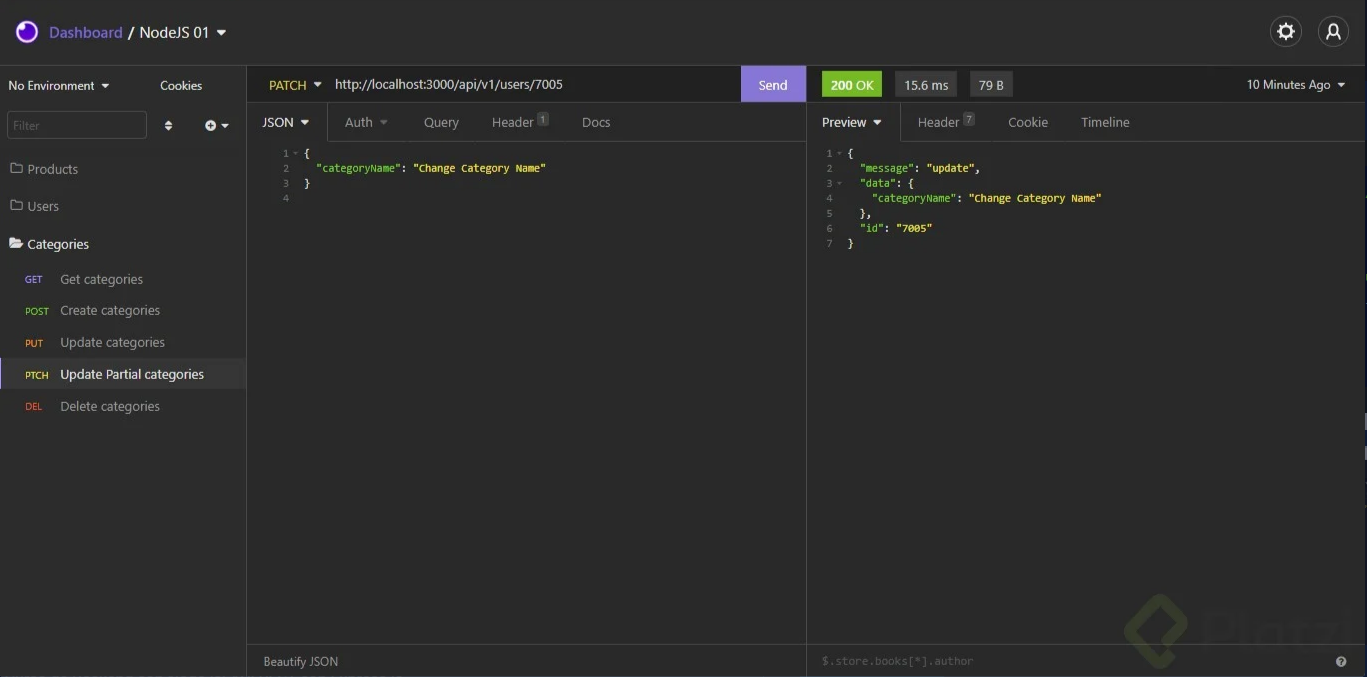
Esto es porque necesitamos aplicar un middleware nativo de express antes de aplicar nuestra llamada al método POST.

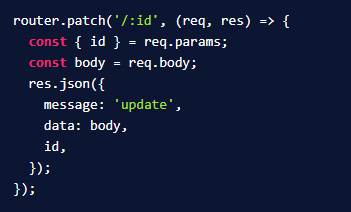


# **PUT, PATCH y DELETE**

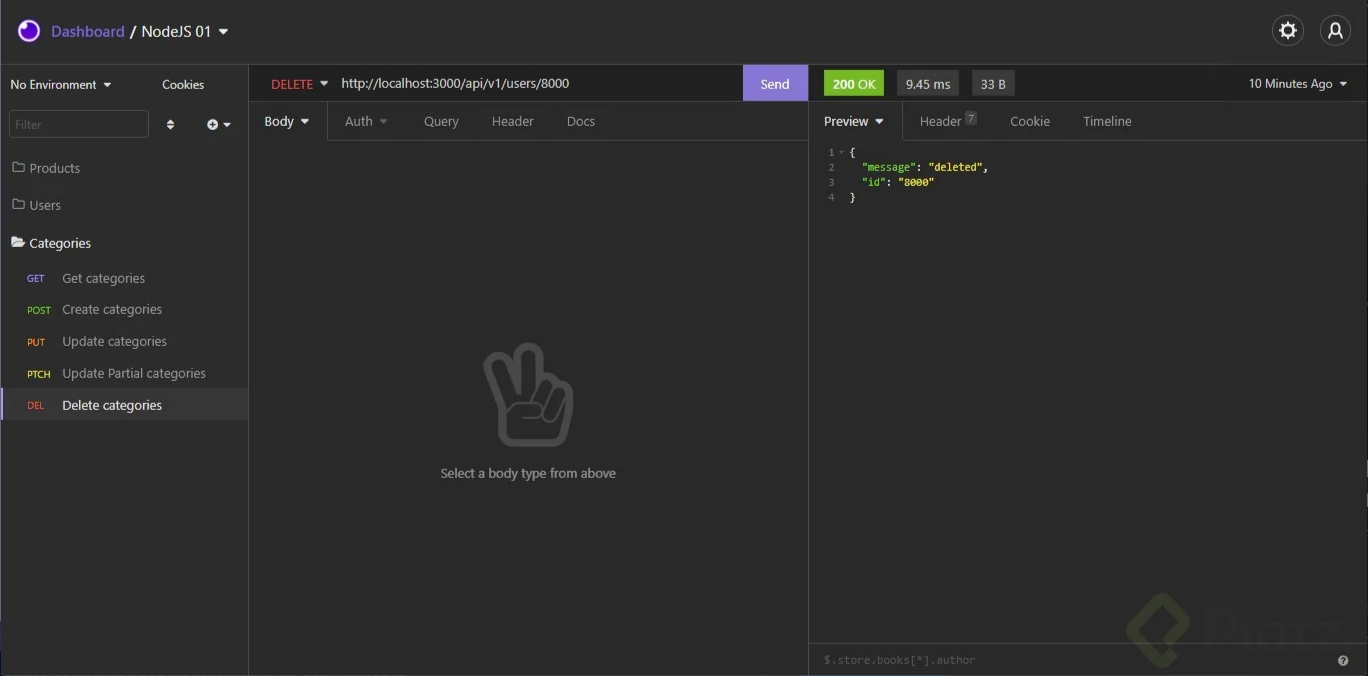
* **PUT**: Reemplaza por completo un recurso, es decir, necesita todos los parámetros por más que se deba actualizar uno sólo de ellos.
* **PATCH**: Reemplaza parcialmente un recurso, es decir, sin la necesidad de pasar todos los parámetros.
* **DELETE**: Elimina un recurso.

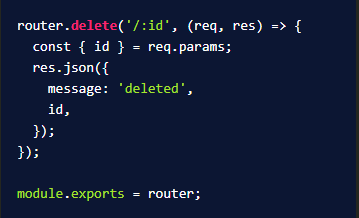
## **PATCH**





## **DELETE**





# **Códigos de estado o HTTP response status codes**

Los status code (códigos de estado) es un estándar que nos permite saber que paso con una solicitud.

## **Listado de códigos de estados**

**Información**

**100**: Continue.

**101**: Switching Protocols.

**103**: Early Hints.

**Exitosos**

**200**: OK.

**201**: Created.

**202**: Accepted.

**203**: Non-Authoritative Information.

**204**: No Content.

**205**: Reset Content.

**206**: Partial Content.

**Redireccionamientos**

**300**: Multiple Choices.

**301**: Moved Permanently.

**302**: Found.

**303**: See Other.

**304**: Not Modified.

**307**: Temporary Redirect.

**308**: Permanent Redirect.

**Error del cliente**

**400**: Bad Request.

**401**: Unauthorized.

**402**: Payment Required.

**403**: Forbidden.

**404**: Not Found.

**405**: Method Not Allowed.

**406**: Not Acceptable.

**407**: Proxy Authentication Required.

**408**: Request Timeout.

**Error al cliente continuado**

**409**: Conflict.

**410**: Gone.

**411**: Length Required.

**412**: Precondition Failed.

**413**: Payload Too Large.

**414**: URI Too Long.

**415**: Unsupported Media Type.

**416**: Range Not Satisfiable.

**417**: Expectation Failed.

**418**: I'm a teapot.

**422**: Unprocessable Entity.

**425**: Too Early.

**426**: Upgrade Required.

**428**: Precondition Required.

**429**: Too Many Requests.

**431**: Request Header Fields Too Large.

**451**: Unavailable For Legal Reasons.

**Error del Servidor**

**500**: Internal Server Error.

**501**: Not Implemented.

**502**: Bad Gateway.

**503**: Service Unavailable.

**504**: Gateway Timeout.

**505**: HTTP Version Not Supported.

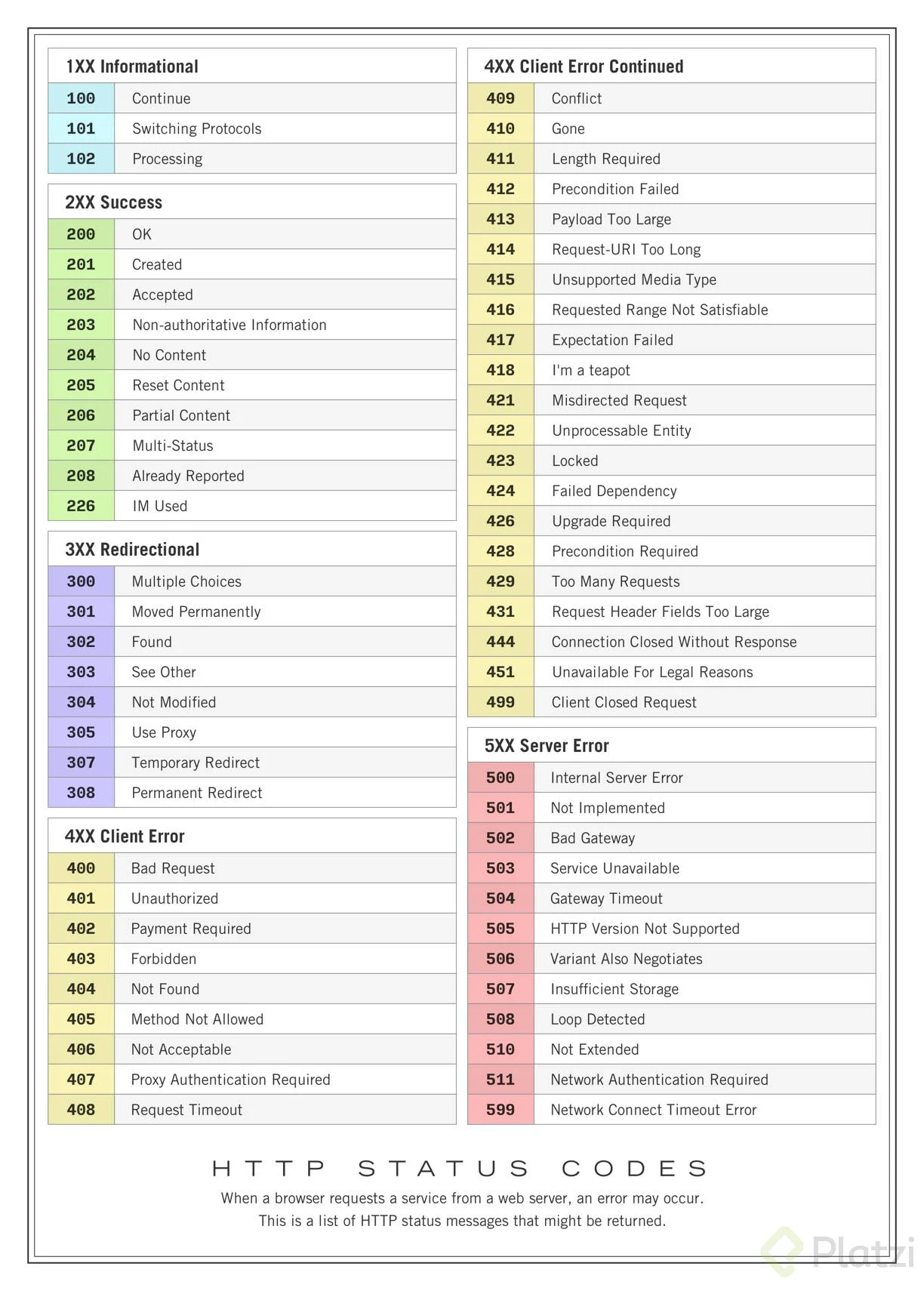
**506**: Variant Also Negotiates.

**507**: Insufficient Storage.

**508**: Loop Detected.

**510**: Not Extended.

**511**: Network Authentication Required.



Códigos de estados explicados con gatitos [aquí](https://http.cat/).

# **Introducción a servicios: crea tu primer servicio**

Los servicios son donde encapsulamos todos los casos de usos y comenzar a interactuar con la lógica de negocio.

En el caso de una tienda: hacer compras, transacciones, etc.

**Estructura**

* Esta arquitectura está definida por capas.

## **Entidades**

* En esta capa encontramos las entidades base del negocio.
* En nuestro caso: productos, categorías, órdenes de compra.

## **Casos de uso**

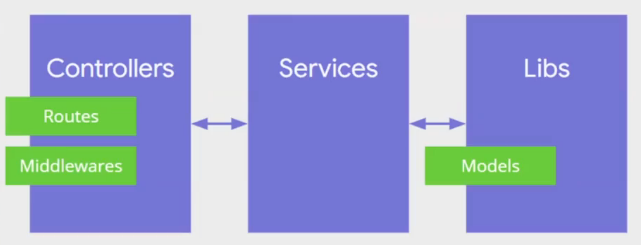
* En esta capa tenemos lo relacionado a la lógica de negocio
* En esta capa se encuentra los servicios.

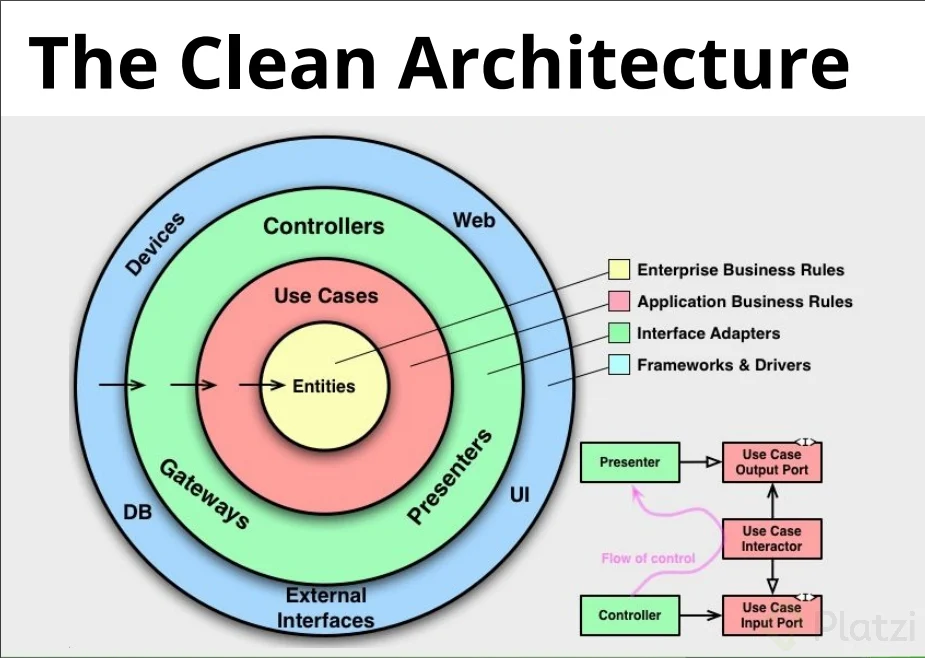
## **Controladores**

* En esta capa se brinda el acceso.
* Aquí encontramos el routing.

## **Flujo de trabajo**

* **Controladores**
  + Encontramos los routes y middlewares.
  + Acceden a la capa de servicios.
* **Servicios**:
  + Donde se encuentra la lógica de negocio.
  + Usan las librerías.
  + Las librerías se encargan de contactarse a la capa de entidades.
  + Las librerías se contactan a otras fuentes de datos: API externa o base de datos.





# **Async await y captura de errores**

* **Async** nos permite usar el keyword **await** dentro de esa funcion.
* **Await** ejecuta el contenido de una funcion de forma síncrona.

# **¿Qué son los Middlewares?**

Request → **Middleware** → Response.

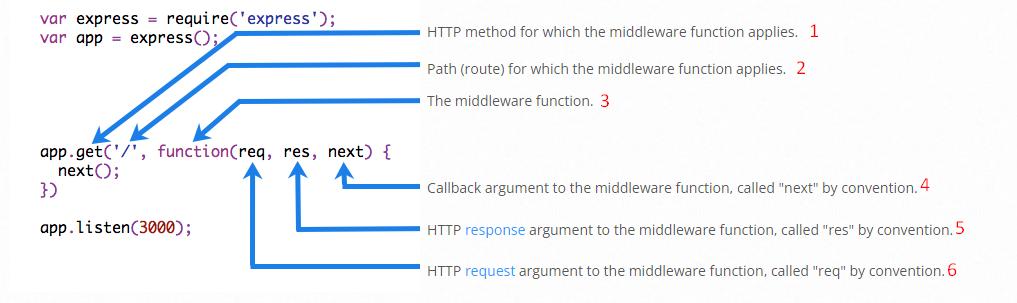
Middleware → Middleware X→ Middleware.

Las funciones de middleware **son funciones que tienen acceso al objeto de solicitud** (**req**), **al objeto de respuesta** (**res**) **y a la siguiente función en el ciclo de solicitud-respuesta de la aplicación**. La siguiente función es una función en el enrutador **Express** que, cuando se invoca, ejecuta el middleware que sucede al middleware actual.

Las funciones de middleware pueden realizar las siguientes tareas:

* Ejecutar cualquier código.
* Realice cambios en los objetos de solicitud y respuesta.
* Terminar el ciclo de solicitud-respuesta.
* Llame al siguiente middleware de la pila.

La siguiente figura muestra los elementos de una llamada de función de middleware:

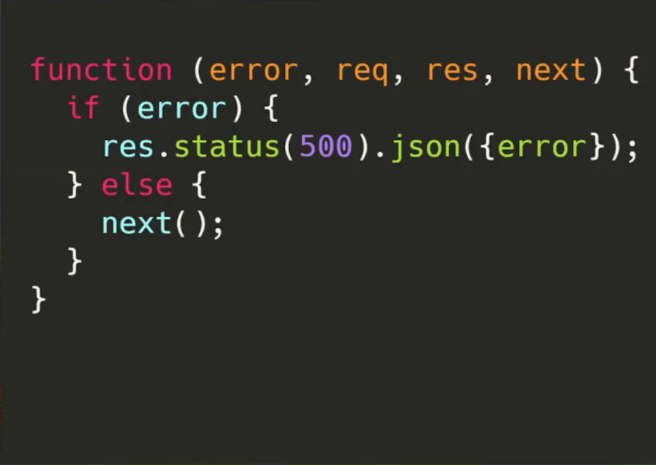


1. Método HTTP para el que se aplica la función de middleware.
2. Ruta para la que se aplica la función de middleware.
3. La función de callback.
4. Argumento de callback que llamará al siguiente middleware para ser ejecutado, llamado "next" por convención.
5. Argumento de respuesta HTTP a la función de middleware, llamado "res" por convención.
6. Argumento de solicitud HTTP para la función de middleware, llamado "req" por convención.

A partir de **Express 5**, las funciones de middleware que devuelven una **Promesa** llamarán a **next(value)** cuando rechacen o arrojen un error. **next** será llamado con el valor rechazado o con el **Error** arrojado.

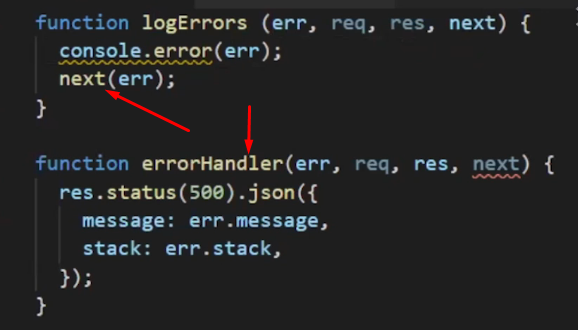
# **Middleware para HttpErrors**

Este middleware nos permite detectar errores en nuestra petición y poder así devolverlo a nuestro cliente.



* Los middlewares de tipo error se deben ser definidos **después** del routing, es decir, de las router (rutas.)
* Algo muy importante es ver el **orden en el que se ejecutan**, es decir, en el orden que se pongan en el código, será el orden que se ejecutarán uno tras otro. **Comportamiento de forma secuencial**.

**Ejemplo**



Vemos que la 1er función **logErros** ejecuta una función **next()** la cual llamará a la siguiente función **errorHandler**, si la segunda función es llamada primero no ejecutará la primera función porque no tiene la función **next()**.

# **Manejo de errores con Boom**

[Boom](https://hapi.dev/module/boom/) es una librería que nos permite manejar errores de forma fácil y dinámica con nuestros middlewares de errores.

# **Validación de datos con Joi**

[Joi](https://joi.dev/api/?v=17.6.0) es una librería que nos permite tipar y validar nuestros datos de forma fácil, sencilla y así poder notificarle al cliente si los datos enviados son los requeridos o no.

Se pueden encontrar como **Schema** o **DTO** (**D**ata **T**ransfer **O**bjects, en español, Objetos de Transferencia de datos.)

# **Middlewares populares en Express.js**

A continuación, se encuentra una lista de los middlewares más populares en Express.

* [CORS](https://expressjs.com/en/resources/middleware/cors.html): Middleware para habilitar CORS (Cross-origin Resource Sharing) en nuestras rutas o aplicación.
* [Morgan](https://expressjs.com/en/resources/middleware/morgan.html): Un logger de solicitudes HTTP para Node.js
* [Helmet](https://github.com/helmetjs/helmet): Nos ayuda a proteger nuestras aplicaciones Express configurando varios encabezados HTTP. ¡No es a prueba de balas de plata, pero puede ayudar!
* [Express Debug](https://github.com/devoidfury/express-debug): Nos permite hacer debugging de nuestras aplicaciones en Express mediante el uso de un toolbar en la página cuando las estamos desarrollando.
* [Express Slash](https://github.com/ericf/express-slash): Este middleware nos permite evitar preocuparnos por escribir las rutas con o sin slash (**/**) al final de ellas.
* [Passport](https://github.com/jaredhanson/passport): Es un middleware que nos permite establecer diferentes estrategias de autenticación a nuestras aplicaciones.

Se pueden encontrar más middlewares populares de express [aquí](https://expressjs.com/en/resources/middleware.html).

# **Consideraciones para producción**

Al subir el código a producción se deben tener en cuenta las siguientes consideraciones.

**Cors**: Que acceso y a quienes le damos acceso para hacer solicitudes.

**Https**: Nuestra API debe correr sobre servidor de HTTPS.

**Procesos de Build**: Esto se ve en procesos que compilan o transpilan código, por ej: TypeScript, Sass,

**Remover logs**: No es bueno tener logs, a veces esto tiene demoras como sucede en python, existen mejores formas para capturar logs.

**Seguridad (Helmet)**: Muy importante la seguridad y para esto se recomienda **Helmet** que es una colección de Middleware que colocan capas de seguridad a la aplicación.

**Testing**: Correr pruebas unitarias o de integración antes de salir de producción.