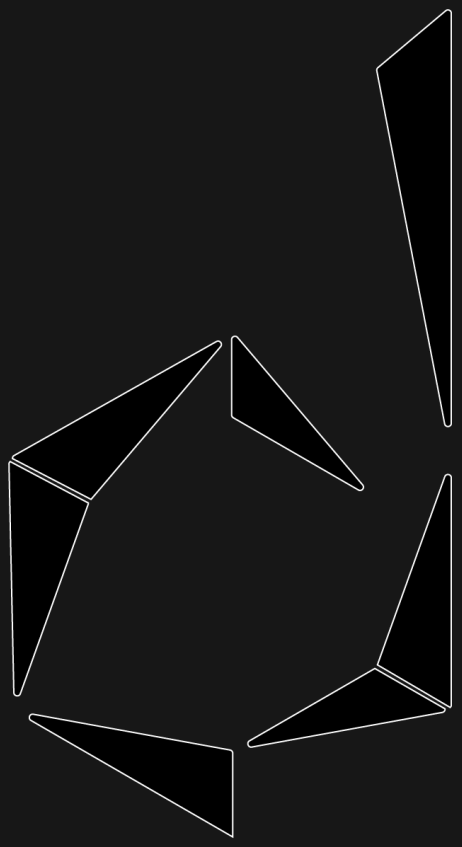
INGENIERÍA MECATRÓNICA



DI_CERO

DIEGO CERVANTES RODRÍGUEZ

PROGRAMACIÓN: DESARROLLO BACKEND

POSTMAN, INSOMNIUM, PYTHON, ETC.

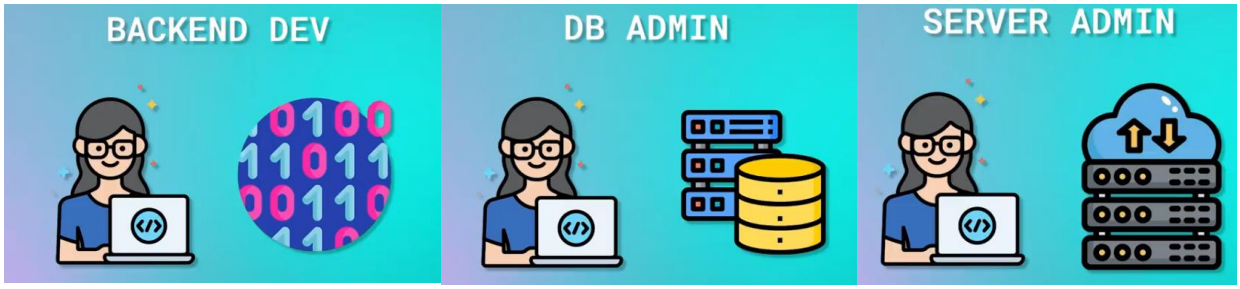
Introducción a la Programación Backend

Contenido

Backend Developer

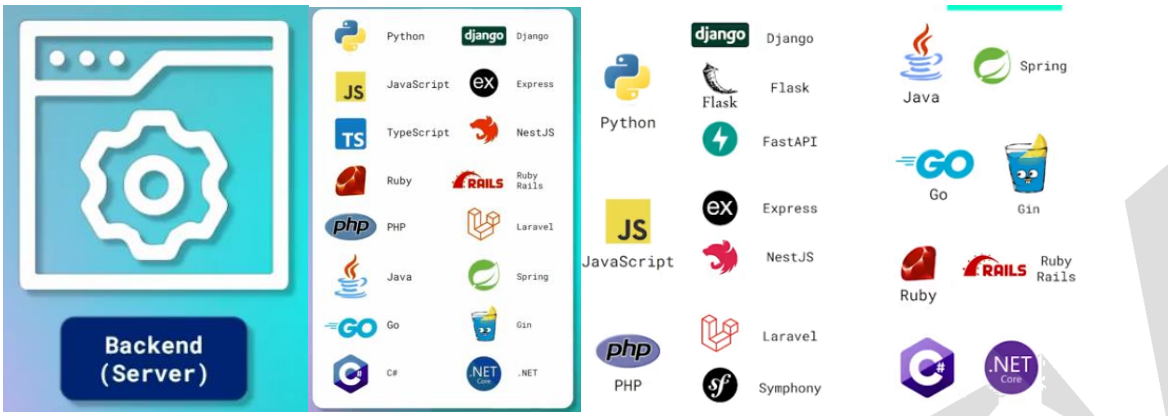
El rol principal del **backend developer** es el de **escribir código que tenga que ver con reglas de negocio, ya sea validación, autorización de usuarios, conexión a bases de datos, manejo de datos, etc**. El cual se montará sobre un servidor y será expuesto a millones de usuarios. Sin embargo, como backend también se puede adoptar los roles de:

- **DB Admin:** Este se encarga de gestionar una base de datos, tomando en cuenta sus políticas, disposición, seguridad, etc.
- **Server Admin:** Este se encarga de gestionar la seguridad de los servidores donde corre el código del backend.



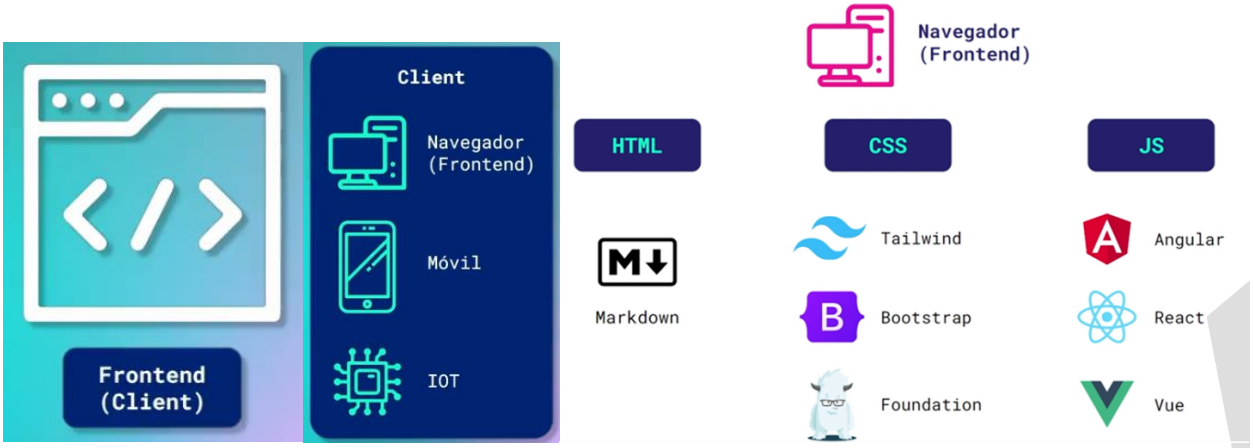
Backend, Frontend y Fullstack Developer

- **Bakend Developer:** Desarrolla y pone a disposición **servicios** en los cuales los **clientes (plataformas frontend como páginas web, aplicaciones móviles, microcontroladores, etc.)** se conectan para **extraer datos** y mostrarlos en su interfaz gráfica o utilizarlos de otra manera.
    - **Los lenguajes más populares y sus frameworks para escribir servicios de backend en un servidor son**: Python (Django o FastAPI), JavaScript (Express de Node.js), TypeScript (NestJS), PHP (Laravel), Java (Spring), C# (.NET), Go (Gin), Ruby (Ruby on Rails), etc.



- **Frontend Developer:** Se enfoca en el área del **renderizado**, esto se realiza cuando **un cliente se conecta con un servicio backend para generar una representación visual o auditiva a partir de datos**. Los diferentes clientes frontend pueden ser:

- **Navegadores:** Es el más popular, ya que se encarga de conectarse a un servicio de backend que está montado en un servidor para mostrar los datos HTML que se despliegan (renderizan) en un sitio web. Los navegadores soportan 3 tipos de datos que puede devolver el servicio de un backend, los cuales son:
  - **HTML o Markdown:** Como backend developer se podrá crear un servicio que renderice código HTML para mostrar diferentes datos en el cliente frontend o también para mostrar blogs se pueden renderizar archivos markdown, los cuales sirven para creación de textos con formato.
  - **CSS:** Es código de estilo para la estructura estética de un sitio web HTML, este cuenta con librerías como Bootstrap, Tailwind, Foundation, etc.
  - **JavaScript:** Este se utiliza para proporcionar efectos o dinamismo a un sitio web, los cuales son implementados mayormente con frameworks como Angular, React.js, Vue.js, etc.
- **Smartphones:** Los cuales extraen datos para mostrar aplicaciones móviles en los celulares. Los teléfonos soportan 3 tipos de datos que puede devolver el servicio de un código backend, los cuales son:
  - **iOS Nativo:** Estos trabajan con código Swift u Objective C y solo pueden ser utilizados en sistemas operativos de Apple.
  - **Android Nativo:** Estos pueden recibir código Java o Kotlin y solo pueden ser utilizados en dispositivos con sistema operativo Android.
  - **Cross Platforms:** Estos pueden recibir ya sea código de React Native, Flutter o .NET MAUI, lo interesante de este método es que se puede utilizar en dispositivos con sistema operativo de iOS o Android, pero tiene algunas desventajas como el no poder realizar de forma tan sencilla el uso de sensores como GPS, Cámaras, etc.
- **Microcontroladores:** Los cuales pueden crear dispositivos físicos IoT que se conecten y manden o extraigan datos de algún servidor o base de datos.

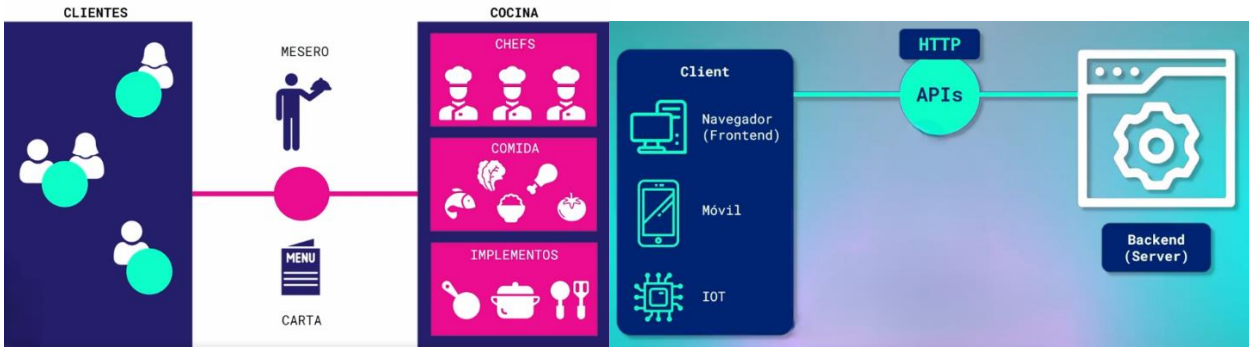


Cómo se Construye el Backend: Concepto de API

Una vez que se tiene **un cliente (frontend) y un servicio (backend)**, lo que se hace **para conectarlos es crear una API (Application Programming Interface)**, la cual **se comunica por** algo llamado **métodos HTTP**.

Para explicar el funcionamiento de las APIs, se utiliza la analogía de un mesero, donde se cuenta con uno o varios comensales (cliente frontend) y la cocina (servicio backend) de un restaurante:

- **Comensales:** Estos representan cada navegador, teléfono o microcontrolador utilizado por un usuario, que simboliza un cliente frontend, el cual va a solicitar utilizar el servicio proporcionado por el backend (cocina).
- **Carta o Menú:** Esto representa la pantalla del sitio web, aplicación móvil o interfaz IoT con la cual el usuario (navegador, teléfono o microcontrolador) puede ver el listado de funciones que puede solicitar.
- **Mesero:** Este representa la API, que se encarga de recoger la solicitud del comensal ya que haya visto la carta y pedir que se prepare esto a la cocina (el backend).
- **Cocina:** Esto representa el backend completo del servicio, el cual utiliza las siguientes partes para realizar su función.
    - **Chefs:** Endpoints de desarrollo, estos son un listado de URLs que reciben y devuelven un tipo de dato en específico cada vez que sean utilizados.
    - **Almacén de Comida:** Esto representa las bases de datos, las cuales ya cuentan con datos que hayan sido recabados del frontend, sean llenadas de forma manual o se encuentren en un data warehouse para responder a las solicitudes de los clientes.
    - **Implementos, Cubiertos o Herramientas:** Estos representan las librerías con las que cuenta cada lenguaje de programación para permitirnos conectarnos a bases de datos, realizar autentificaciones de usuarios, etc.



En resumen, **las APIs nos permiten comunicarnos con el código backend de una aplicación**, ya sea de su mismo sistema o de otro a través del protocolo HTTP haciendo uso de endpoints para recibir como valor de retorno un código que renderice el frontend o datos en formato JSON o XML.

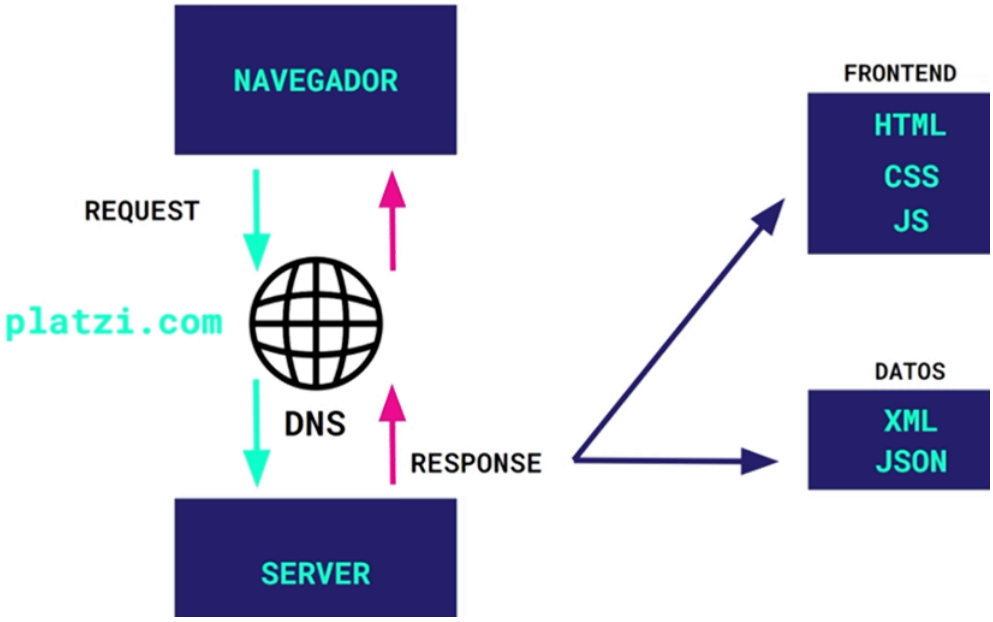
HTTP: HyperText Transfer Protocol

El **protocolo HTTP** se conforma de una **URL** que nos ayuda a **explorar y exponer servicios web**, **es la forma en la que el frontend y el backend interactúan entre sí**, el cual se conforma de las siguientes partes:

- **Protocolo:** Esta es la parte inicial del URL que indica el protocolo que se está utilizando, ya sea ftp (para transmisión de archivos), http o https (que contiene una capa de seguridad).
- **Dominio:** Esta parte **se compra a través de un servicio de host** y **nos provee con un servidor DNS**, el cual nos **permite tener la propiedad de una dirección IP**, la cual por motivos de facilidad se ve como una palabra en vez de números, llamado **dominio** y es como la gente podrá acceder a cada sitio web desplegado (deploy) en la nube.
- **Ruta o Endpoint:** Este va separado del dominio por medio de un símbolo de slash (/)
    - **Ruta:** Si nos estamos refiriendo a **una ruta** nos permitirá acceder a **cada carpeta que contenga una imagen, archivo, etc.**
    - **Endpoint:** Si nos referimos a **un endpoint**, este **permite acceder a un dato en específico que es expuesto a través de un servicio backend** en formato JSON o XML para que el frontend lo tome y este pueda ser renderizado, actualizado o interpretado por el cliente, a través de código HTML, CSS, JavaScript, Java, Kotlin, Swift, etc.

- **Status HTTP:** Algo muy interesante de los **protocolos HTTP** son sus **códigos de estado (status)**, estos nos **dan información para saber el estado en el que se encuentra la petición que se realizó del cliente hacia el servicio**. Su significado se encuentra categorizado por rango, el cual será explicado a continuación:
    - **100-199:** Estos indican **información de estado del servidor hacia el cliente**, que puede ser como la 102 = Procesando datos.
    - **200-299:** Este es el rango más utilizado porque **denota éxito en alguna acción del servidor**, por ejemplo 200 = Ok, 201 = Algo fue creado de forma exitosa, 204 = La acción fue exitosa pero no retornó ningún dato, etc.
    - **300-399:** El rango de los 300 indica que **algún recurso ha sido movido de lugar**, por ejemplo 301 = El recurso que antes estaba aquí, fue movido de lugar, 307 = El recurso fue temporalmente movido, 308 = El recurso fue permanentemente movido, etc.
    - **400-499**: **Representan errores de la parte del cliente**, esto normalmente ocurre porque el cliente envió una solicitud de forma errónea, por ejemplo 401 = No autorizado, 404 = No encontrado, 409 = Existe conflicto porque los datos están siendo mandados de forma incorrecta.
    - **500-599:** Representan errores que ocurren de parte del servidor, del código que está ejecutándose, por ejemplo 500 = Error interno del servidor, 502 = Bad Gateway, 504 = El tiempo de espera del servidor se ha sobrepasado.

Para explicar todos estos tipos de respuesta del servidor en forma divertida con memes se cuenta con el sitio: [HTTP Cats]



Estructura REST: API con Protocolo HTTP

El estándar o **estructura REST API se utiliza para desarrollar APIs que funcionen a través del protocolo HTTP**, por lo que **es la más utilizada**. La forma en la que **se piden recursos o datos a los endpoints de las REST APIs** es **a través de su dominio** y normalmente **devuelven su resultado en formato JSON**, **que luego serán procesador por un cliente web, smartphone o un microcontrolador**.

La forma en la que funcionan las REST APIs es **proporcionando como mínimo métodos CRUD** que permitan **Crear, Leer, Actualizar o Borrar datos** de una base de datos a través de sus endpoints:

- **Create =** Método **Post** del protocolo HTTP.
- **Read =** Método **Get** del protocolo HTTP.
- **Update =** Métodos **Put/Patch** del protocolo HTTP.
- **Delete =** Método **Delete** del protocolo HTTP.

domain.com	**CRUD**
domain.com/**users**	CREATE = **POST** READ = **GET**
domain.com/**tasks**	UPDATE = **PUT / PATCH**
domain.com/**companies**	DELETE = **DELETE**

Cada que se cree un endpoint se debe indicar:

1. El **nombre del endpoint**.

2. Cuál es el **método HTTP** que se ejecuta al acceder a dicho endpoint
3. **El body que contiene los datos que utilizará el método** HTTP para realizar su función.
4. **La respuesta que se obtendrá** del servidor al utilizar el endpoint.
5. **El número de status HTTP que se espera recibir.**

endpoint	method	body	response	status
/users	GET	N/A	[{}, {} …]	200
/users/:id	GET	N/A	{}	200
/users	POST	{ ... }	{ ... }	201
/users/:id	PUT / PATCH	{ ... }	{ ... }	201
/users/:id	DELETE	N/A	✅	201

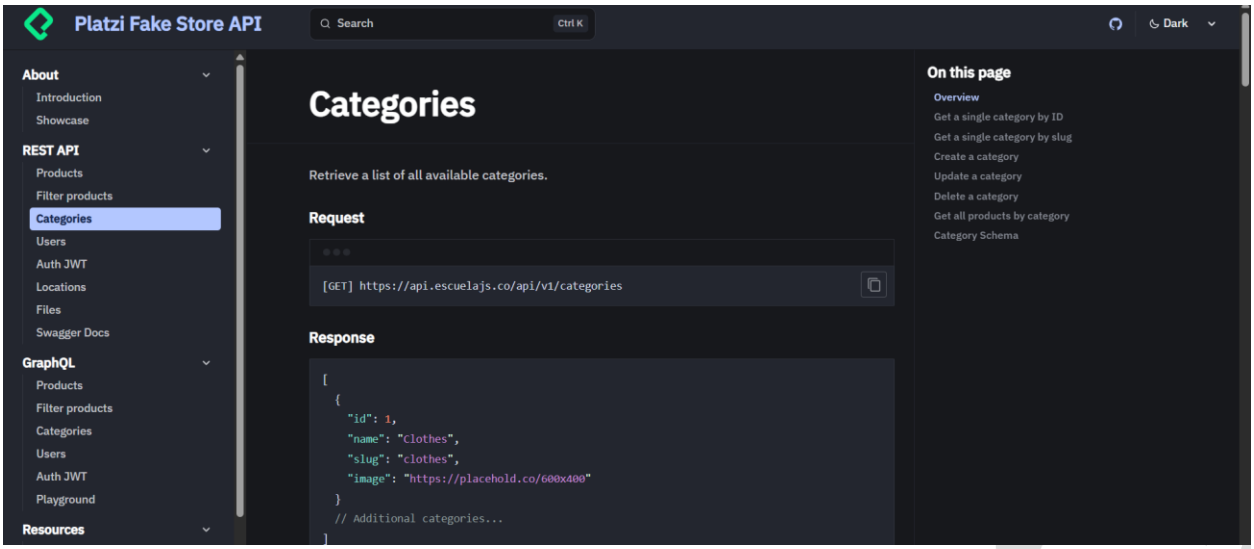
API Testing: Insomnia y Postman

Utilizaremos la herramienta de Insomnia para testear una API de prueba de Platzi que se encuentra en el siguiente dominio: https://fakeapi.platzi.com/

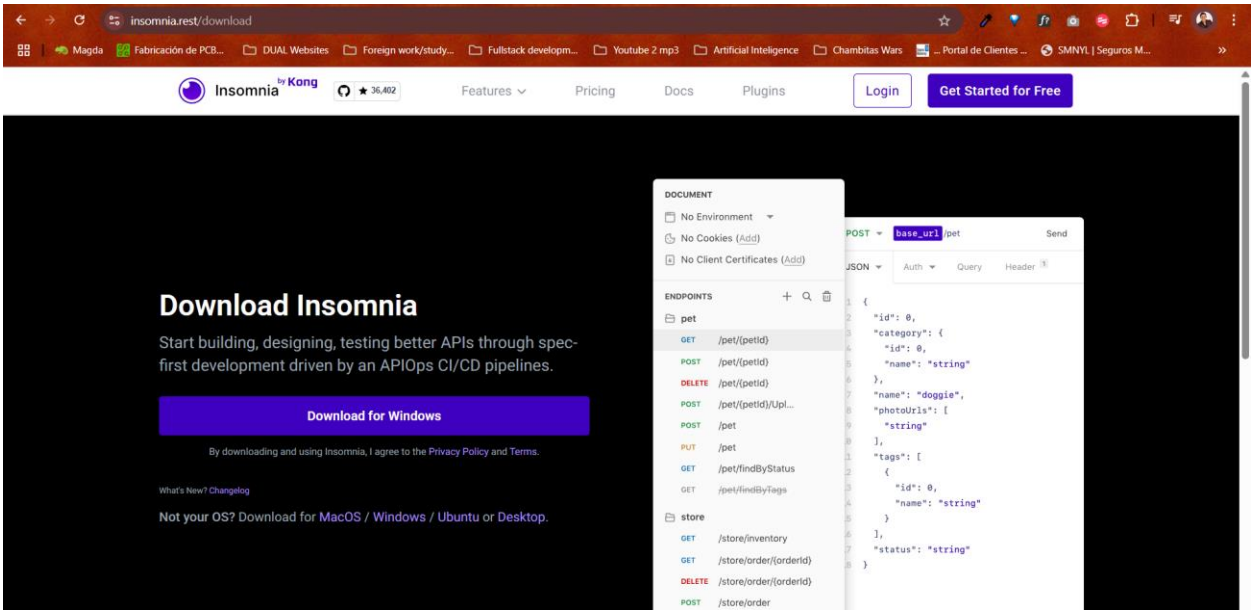
Esta API lo que muestra es una tienda en línea de mentira, la cual tiene productos, imágenes de productos, precios, categorías, etc. y utiliza una arquitectura REST, para usarla veremos su documentación y utilizaremos algunos de sus endpoints para ver cómo funciona la herramienta:

En las documentaciones de APIs lo que podremos ver será:

- **[Método HTTP que se utiliza]** + el dominio + **/el nombre del endpoint**.



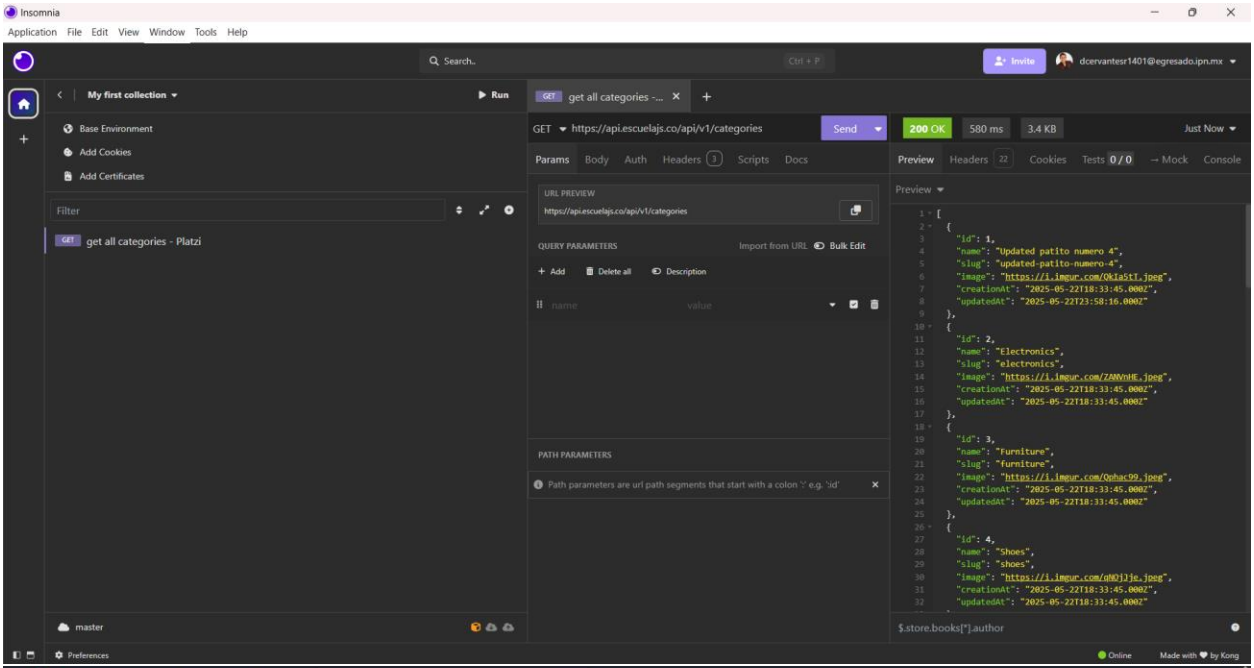
El software de Insomnia puede ser descargado del siguiente enlace: https://insomnia.rest/download

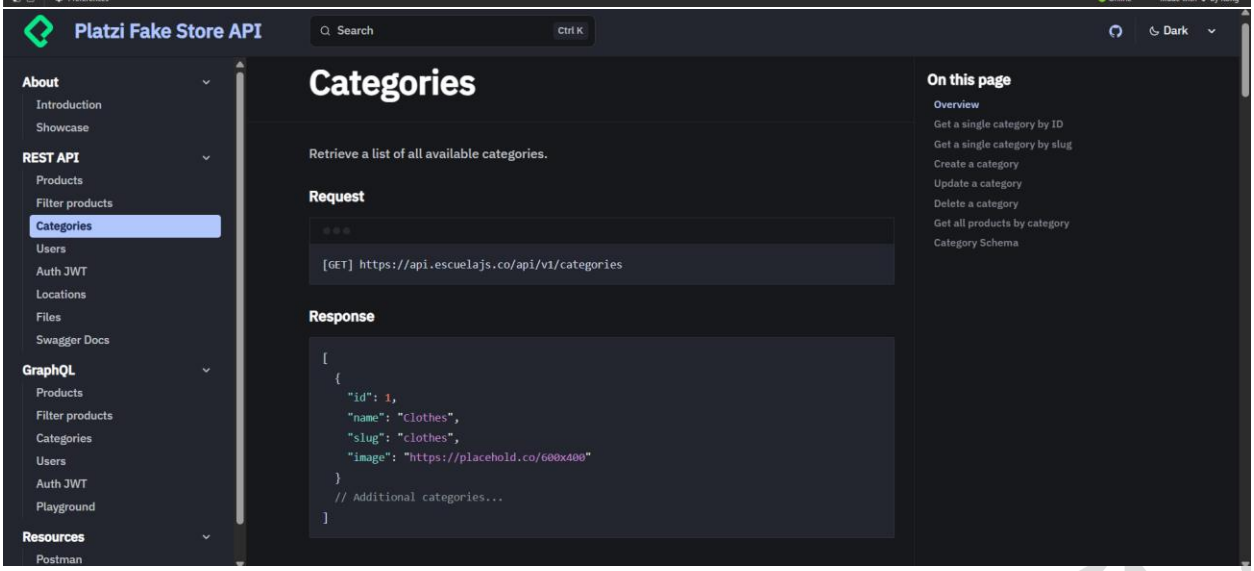


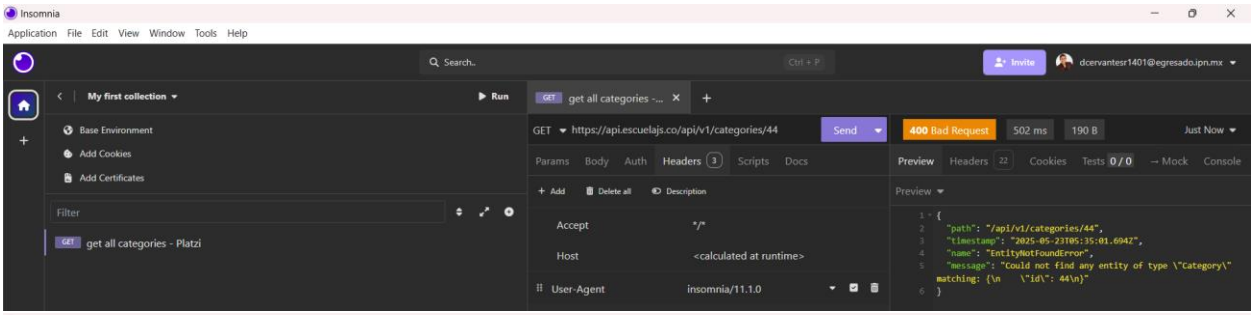
En la interfaz de Insomnia se tiene lo siguiente:

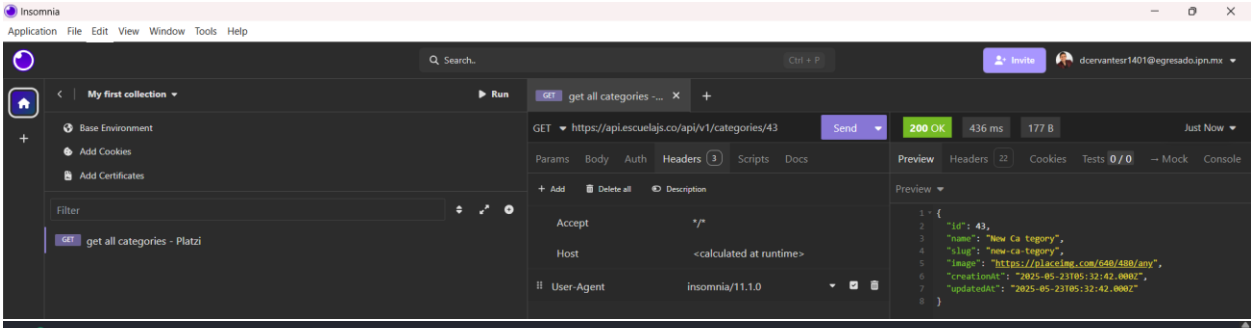
- Del lado izquierdo superior podemos ver el environment, colección o documento en el que nos encontramos, lo cual se refiere a
    - Debajo y del lado izquierdo podemos ver toda la lista de endpoints a los que podemos acceder. Cada vez que utilicemos uno nuevo, podemos asignarle un nombre y guardarlo en nuestra colección de endpoints.
        - Los endpoints guardados indicarán el método HTTP que utilizan y su nombre asignado.
            - Para crear un nuevo request y por lo tanto, guardar un nuevo endpoint, debemos dar clic en el botón de + que se encuentra al lado derecho del filtro de requests.
        - De igual forma estos pueden ser categorizados en folders.
- En la parte media superior podemos introducir la siguiente estructura del endpoint:
    - **Método HTTP que se utiliza** + el dominio + **el nombre del endpoint**.
    - Abajo del endpoint se pueden introducir sus siguientes características:
        - **Params:** Se utiliza para ver la endpoint completa y si se quieren añadir elementos como un nombre y un valor a la URL.
        - **Body:** Es el cuerpo del mensaje que se quiere mandar, osea los datos que espera, esto se utiliza solamente cuando se ejecutan comandos HTTP POST, PUT, PATCH o DELETE.
            - En esta parte se puede escoger el formato del body, ya sea en formato JSON, XML, etc.
        - **Auth:** Es el método de autenticación, para validación de usuarios o seguridad.
        - **Headers:** Es el encabezado del body del mensaje que se manda a la API.
        - **Scripts:** Para poder escribir código backend que recabe datos.
        - **Docs:** Es la documentación del endpoint.

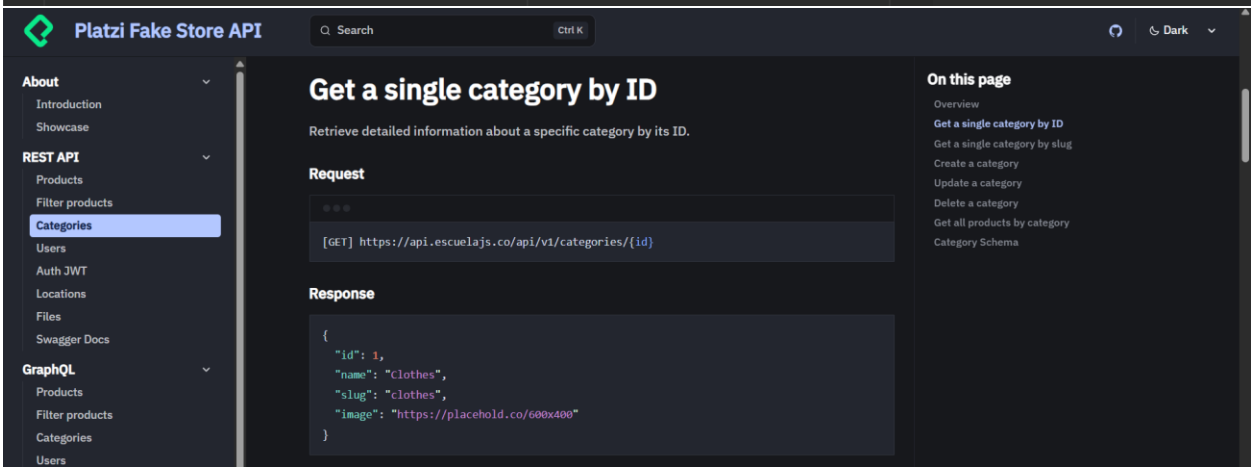
- En la parte derecha, podemos ver la respuesta de la solicitud que se realiza hacia cada endpoint de la API.
    - En la parte superior izquierda de la sección derecha podemos ver:
        - El HTTP status code que devolvió la solicitud.
        - El tiempo que se tardó en recuperarla.
        - El peso de la respuesta dada en bits de memoria.
    - Debajo de la franja de estado de la solicitud, se encuentran opciones donde se puede ver un preview de la respuesta, solo su header, las cookies que retornó y su timeline.

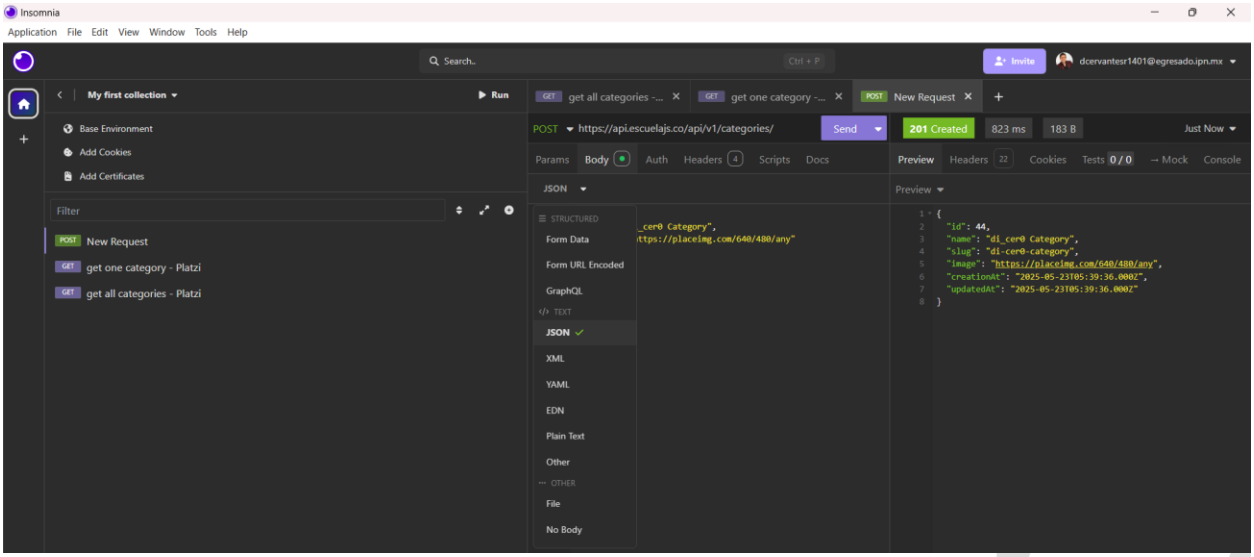


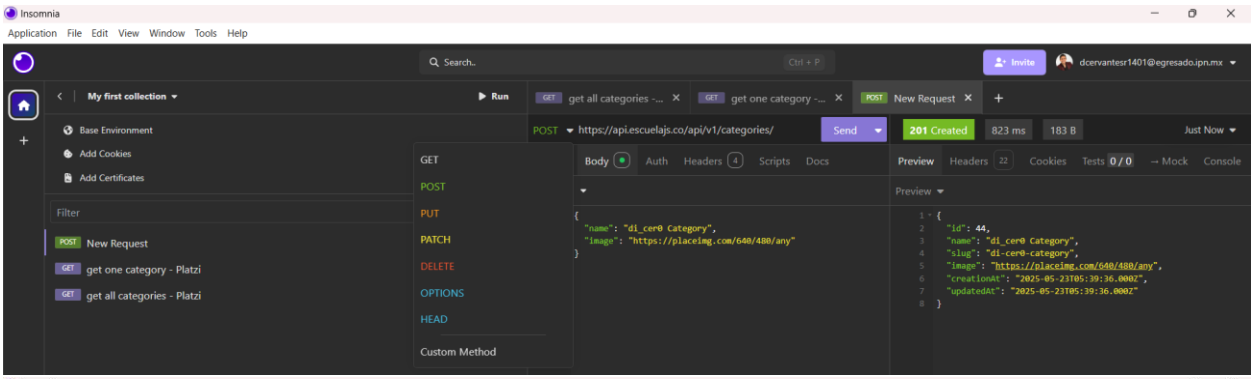


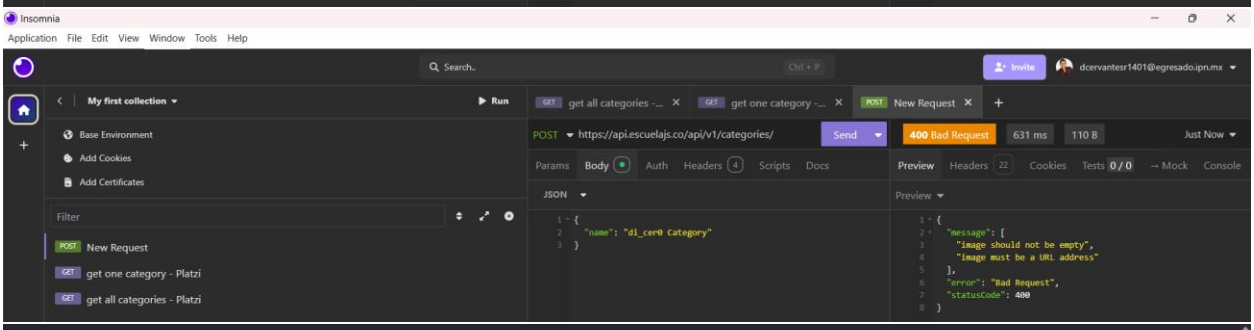

Insomnia
Application File Edit View Window Tools Help
Search..
Invite
dcervantesr1401@egresado.ipn.mx
My first collection
Run
get all categories -...
Base Environment
Add Cookies
Add Certificates
Filter
get all categories - Platzi
GET https://api.escuelajs.co/api/v1/categories/44
Send
400 Bad Request
502 ms
190 B
Just Now
Params Body Auth Headers 3 Scripts Docs
Preview Headers 22 Cookies Tests 0 / 0 Mock Console
Add Delete all Description
Accept */*
Host <calculated at runtime>
User-Agent insomnia/11.1.0
{
"path": "/api/v1/categories/44",
"timestamp": "2025-05-23T05:35:01.604Z",
"name": "EntityNotFoundError",
"message": "Could not find any entity of type \"Category\" matching: {\n \"id\": 44\n}"
}

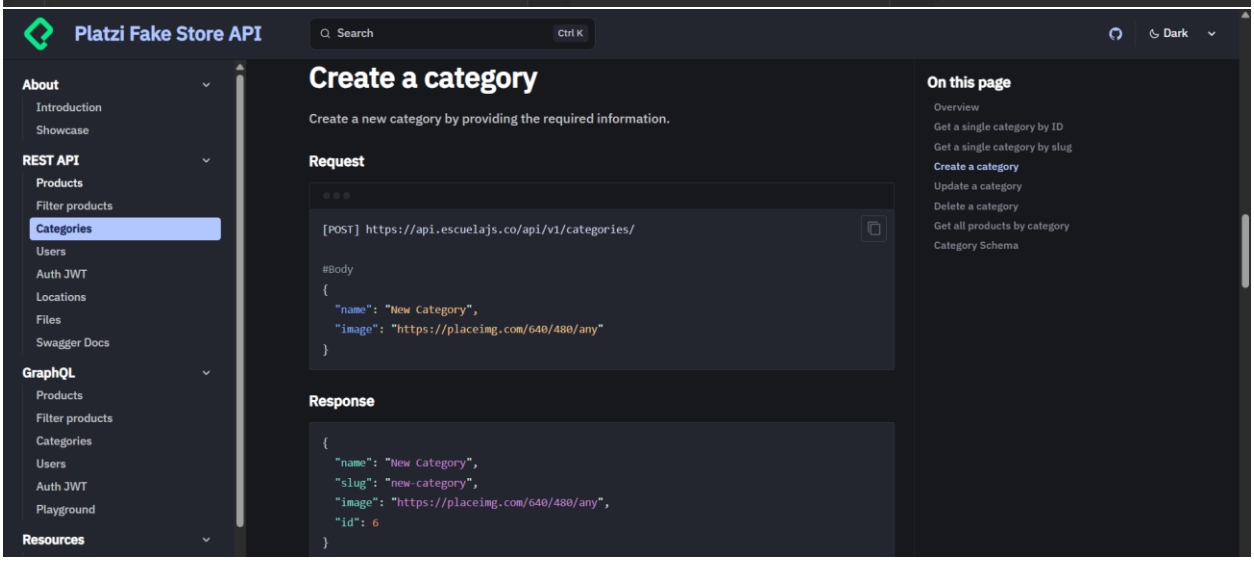

Insomnia
Application File Edit View Window Tools Help
Search..
Invite
dcervantesr1401@egresado.ipn.mx
My first collection
Run
get all categories -...
Base Environment
Add Cookies
Add Certificates
Filter
get all categories - Platzi
GET https://api.escuelajs.co/api/v1/categories/43
Send
200 OK
436 ms
177 B
Just Now
Params Body Auth Headers 3 Scripts Docs
Preview Headers 22 Cookies Tests 0 / 0 Mock Console
Add Delete all Description
Accept */*
Host <calculated at runtime>
User-Agent insomnia/11.1.0
{
"id": 43,
"name": "New Ca tegory",
"slug": "new-ca-tegory",
"image": "https://placeimg.com/640/480/any",
"creationAt": "2025-05-23T05:32:42.000Z",
"updatedAt": "2025-05-23T05:32:42.000Z"
}

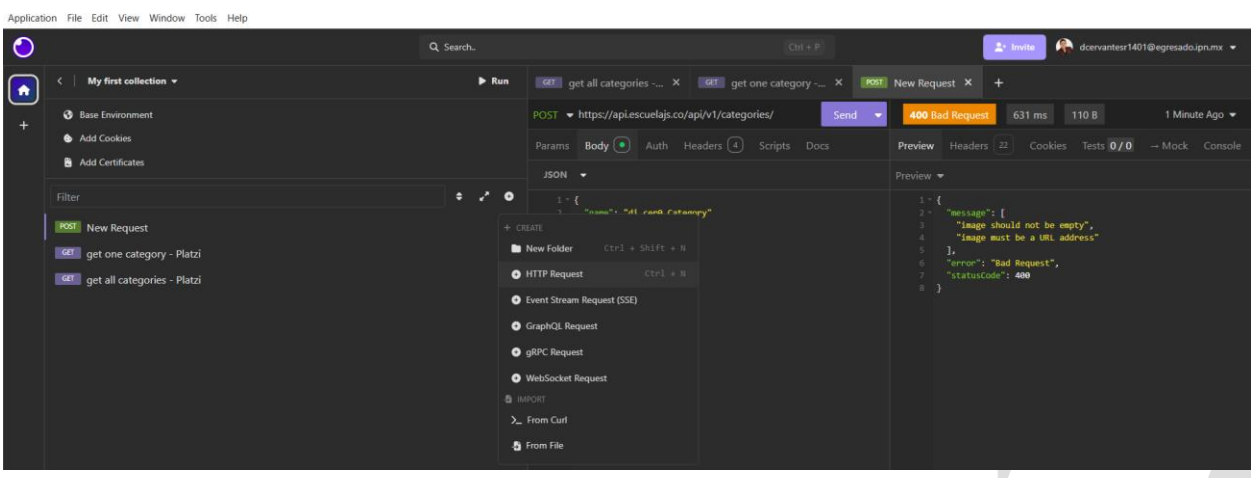

Platzi Fake Store API
Search Ctrl K
Dark
About
Introduction
Showcase
REST API
Products
Filter products
Categories
Users
Auth JWT
Locations
Files
Swagger Docs
GraphQL
Products
Filter products
Categories
Users
Get a single category by ID
Retrieve detailed information about a specific category by its ID.
Request
[GET] https://api.escuelajs.co/api/v1/categories/{id}
Response
{
"id": 1,
"name": "Clothes",
"slug": "clothes",
"image": "https://placehold.co/600x400"
}
On this page
Overview
Get a single category by ID
Get a single category by slug
Create a category
Update a category
Delete a category
Get all products by category
Category Schema

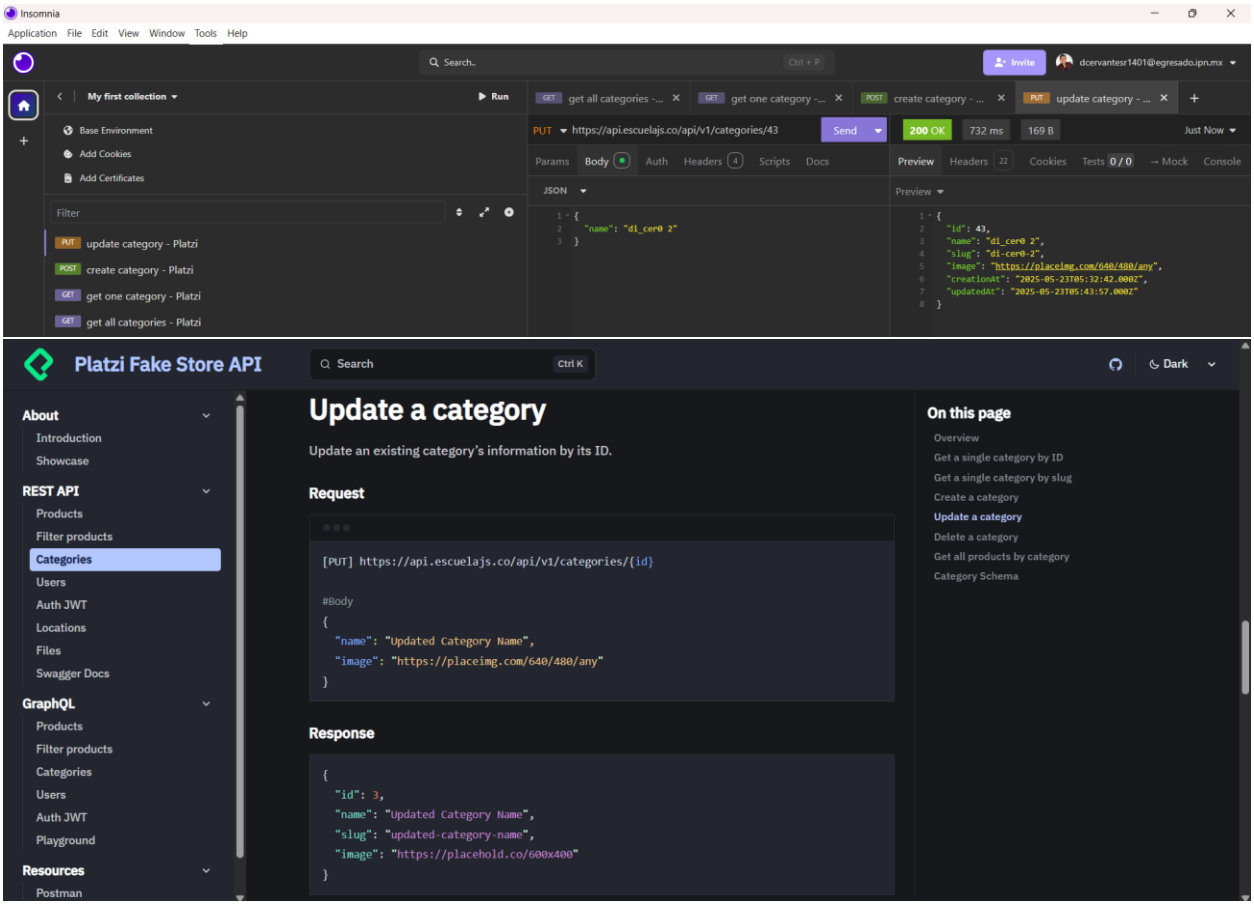

Insomnia
Application File Edit View Window Tools Help
Search..
Invite
dcervantesr1401@egresado.ipn.mx
My first collection
Run
get all categories -...
get one category -...
New Request
Base Environment
Add Cookies
Add Certificates
Filter
New Request
get one category - Platzi
get all categories - Platzi
POST https://api.escuelajs.co/api/v1/categories/
Send
201 Created
823 ms
183 B
Just Now
Params Body Auth Headers 4 Scripts Docs
Preview Headers 22 Cookies Tests 0 / 0 Mock Console
JSON
STRUCTURED
Form Data
Form URL Encoded
GraphQL
TEXT
JSON
XML
YAML
EDN
Plain Text
Other
OTHER
File
No Body
_cer0 Category",
ttps://placeimg.com/640/480/any"
{
"id": 44,
"name": "di_cer0 Category",
"slug": "di-cer0-category",
"image": "https://placeimg.com/640/480/any",
"creationAt": "2025-05-23T05:39:36.000Z",
"updatedAt": "2025-05-23T05:39:36.000Z"
}


Insomnia
Application File Edit View Window Tools Help
Search..
Invite
dcervantesr1401@egresado.ipn.mx
My first collection
Run
get all categories -...
get one category -...
New Request
Base Environment
Add Cookies
Add Certificates
POST https://api.escuelajs.co/api/v1/categories/
Send
201 Created
823 ms
183 B
Just Now
Filter
GET
POST
PUT
PATCH
DELETE
OPTIONS
HEAD
Custom Method
Body
Auth
Headers 4
Scripts
Docs
Preview
Headers 22
Cookies
Tests 0 / 0
Mock
Console
{
"name": "di_cer0 Category",
"image": "https://placeimg.com/640/480/any"
}
{
"id": 44,
"name": "di_cer0 Category",
"slug": "di-cer0-category",
"image": "https://placeimg.com/640/480/any",
"creationAt": "2025-05-23T05:39:36.000Z",
"updatedAt": "2025-05-23T05:39:36.000Z"
}


Insomnia
Application File Edit View Window Tools Help
Search..
Invite
dcervantesr1401@egresado.ipn.mx
My first collection
Run
get all categories -...
get one category -...
New Request
Base Environment
Add Cookies
Add Certificates
POST https://api.escuelajs.co/api/v1/categories/
Send
400 Bad Request
631 ms
110 B
Just Now
Filter
Params
Body
Auth
Headers 4
Scripts
Docs
JSON
Preview
Headers 22
Cookies
Tests 0 / 0
Mock
Console
{
"name": "di_cer0 Category"
}
{
"message": [
"image should not be empty",
"image must be a URL address"
],
"error": "Bad Request",
"statusCode": 400
}


Platzi Fake Store API
Search
Ctrl K
Dark
About
Introduction
Showcase
REST API
Products
Filter products
Categories
Users
Auth JWT
Locations
Files
Swagger Docs
GraphQL
Products
Filter products
Categories
Users
Auth JWT
Playground
Resources
Create a category
Create a new category by providing the required information.
Request
[POST] https://api.escuelajs.co/api/v1/categories/
#Body
{
"name": "New Category",
"image": "https://placeimg.com/640/480/any"
}
Response
{
"name": "New Category",
"slug": "new-category",
"image": "https://placeimg.com/640/480/any",
"id": 6
}
On this page
Overview
Get a single category by ID
Get a single category by slug
Create a category
Update a category
Delete a category
Get all products by category
Category Schema


Application File Edit View Window Tools Help
Search..
Invite
dcervantesr1401@egresado.ipn.mx
My first collection
Run
get all categories -...
get one category -...
New Request
Base Environment
Add Cookies
Add Certificates
POST https://api.escuelajs.co/api/v1/categories/
Send
400 Bad Request
631 ms
110 B
1 Minute Ago
Filter
Params
Body
Auth
Headers 4
Scripts
Docs
JSON
Preview
Headers 22
Cookies
Tests 0 / 0
Mock
Console
CREATE
New Folder
Ctrl + Shift + N
HTTP Request
Ctrl + N
Event Stream Request (SSE)
GraphQL Request
gRPC Request
WebSocket Request
IMPORT
From Curl
From File
{
"message": [
"image should not be empty",
"image must be a URL address"
],
"error": "Bad Request",
"statusCode": 400
}



La diferencia entre los métodos HTTP PUT y PATCH es que PATCH te permite cambiar solo uno de los datos del JSON, mientras que PUT te obliga a mencionarlos todos, aunque el valor sea el mismo en algunos y solo cambien otros.

Cloud (Nube)

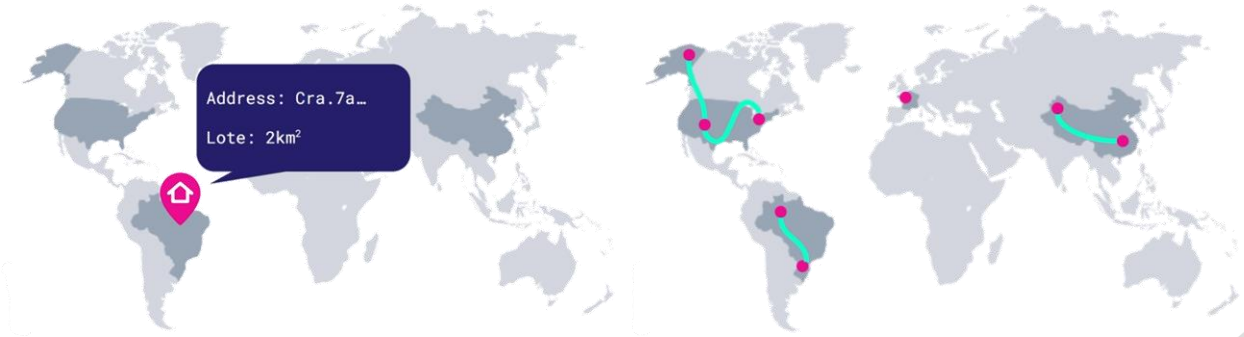
La nube se compone de uno o varios ordenadores que sirven para exponer nuestros servicios de código backend (también llamados granjas de servidores o data centers), los cuales cuentan con CPU, memoria RAM, discos duros, etc. y donde se alojan los servicios y scripts de código a donde los clientes (web, móvil o microcontroladores) acceden por medio de requests con métodos HTTP. Los proveedores de nube más conocidos son AWS (Amazon Web Service), Digital Ocean, Azure (Microsoft), GCP (Google Cloud Platform).

Los **data centers** pueden no estar solo localizados en una región geográfica, sino en varias, con el propósito de dar respuestas más rápidas a los diferentes requests que nuestros servicios puedan recibir. Logrando una menor **latencia o delay** en la recepción de los datos que estamos pidiendo. La **latencia** se refiere al tiempo que tardan los datos en pasar de un punto a otro dentro de una red. Esto implica que dentro de una **nube** puedo tener mi sistema o servicio replicado en diferentes locaciones estratégicas, dependiendo de dónde se encuentren físicamente mis clientes.

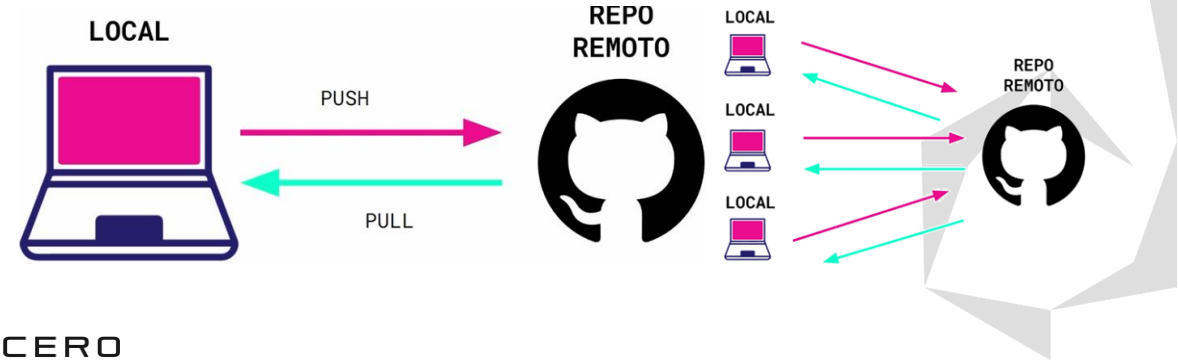
DevOps

DevOps es la parte donde el **desarrollo de código** y la **gestión de operaciones** se unifica, describiendo la forma en la que todos los miembros de un equipo seguirán un flujo de trabajo para lograr que cierto código que se está trabajando de forma local (en nuestro ordenador) llegue a la nube (granja de servidores o data center) y podamos exponerlo (hacer deploy) a miles de usuarios para que lo utilicen.

Para ello, se puede configurar la nube (data center) para que se replique este código de forma estratégica en ciertas zonas geográficas para reducir la latencia que perciban nuestros usuarios.



Para ello se utilizan herramientas de gestión de versiones como GitHub, pudiendo así trabajar varios desarrolladores de código en un mismo código dentro de un mismo repositorio.

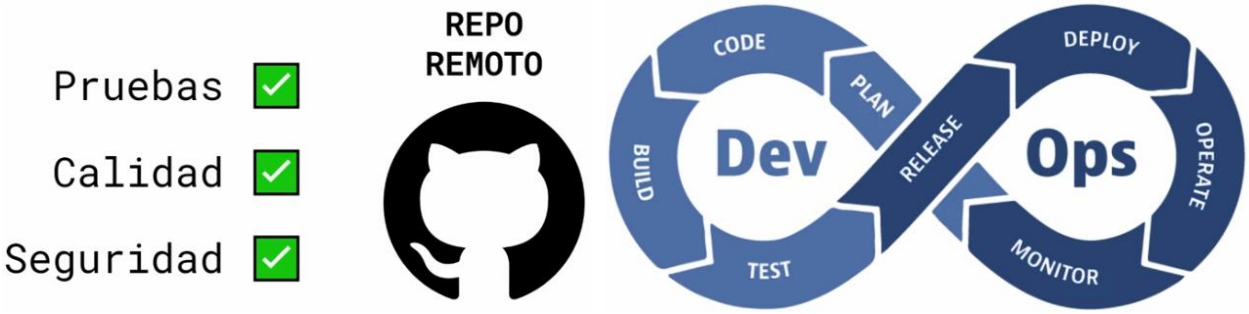


Para que se tenga un orden en la forma que van contribuyendo los miembros de un equipo en un repositorio existe el concepto de rama, donde se tiene la rama principal main, las ramas de desarrollo fix, feature y develop para arreglar bugs o agregar funcionalidades al código.

La rama principal que contiene todo el código que saldrá a producción (es decir, el que se subirá a la **nube** para ser expuesto a los usuarios) debe confirmar que el código siga cierto estándar, asegurando la calidad y seguridad del software a través de pruebas unitarias, esto normalmente se realiza por un rol del equipo de desarrollo llamado **Automation** y es muy importante, ya que si el código que ha sido lanzado a producción tiene un error o falla de seguridad, este error será expuesto a millones de usuarios, causando pérdida de datos o de usuarios y por eso es que el rol de **DevOps** es de suma importancia, ya que:

PLAN: Planifica el desarrollo del código → **CODE:** Desarrolla el código de forma local → **BUILD:** Lo construye → **TEST:** Lo prueba o testea → **RELEASE:** Realiza su lanzamiento→ **DEPLOY:** Lo sube a la nube, exponiéndolo a todos los usuarios → **OPERATE:** Opera el código en producción, gestionando temas de latencia dependiendo de la ubicación geográfica de los usuarios → **MONITOR:** Monitorea su funcionamiento a través del rol de QA (Quality Assurance).

Este flujo de trabajo no es estático, sino que se repite creando iteraciones de estos pasos y una de las herramientas más importantes para realizarlo es un servidor de repositorios como GitHub.

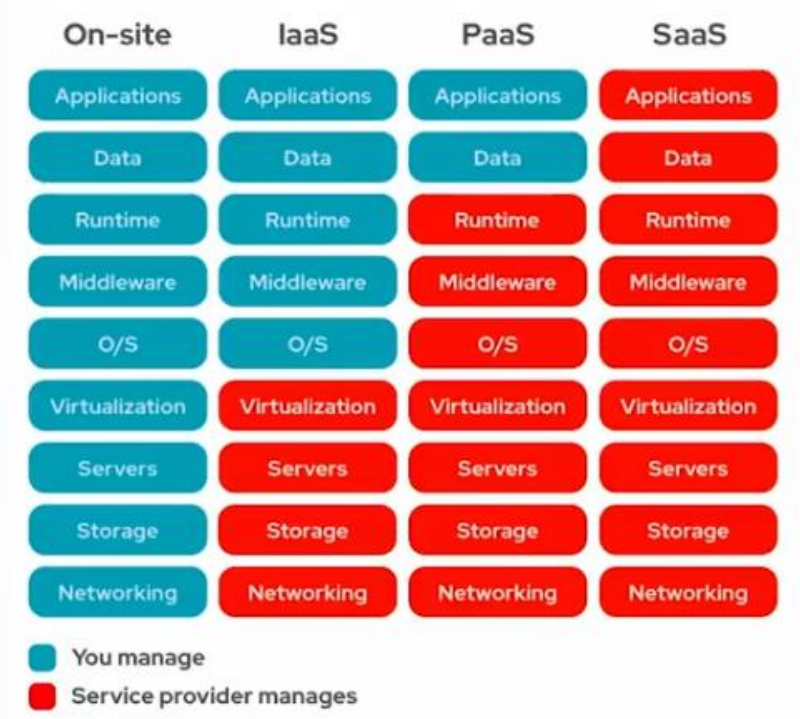


Arquitectura de Servidores

Recordemos que la nube se compone de varios servidores, llamado granja de servidores, en donde cada uno cuenta con CPU, memoria RAM, disco duro de estado sólido SSD o mecánico, etc. Sin embargo, en estas granjas de servidores se puede adoptar una arquitectura específica, dependiendo del tipo de servicio que se vaya a proporcionar a los usuarios.

- **On Site:** Esto se refiere a cuando temenos un servidor físico propio, donde debemos manejar manualmente todas sus operaciones.
- **IaaS (Infrastructure as a Service)**: Esta arquitectura de servidores es la que mayor control nos proporciona de las 3 arquitecturas, pudiendo manejar de forma manual en los servidores la aplicación, sus datos, el periodo de tiempo en el que mi programa se tarda en ejecutarse, el sistema operativo, etc. **Un ejemplo de este es Digital Ocean.**
- **PaaS (Platform as a Service)**: Aquí se cuenta con un poco más de capacidad de administración de las características del servidor como la aplicación y los datos que se manejan, necesitando en ellas intervención manual. **Un ejemplo de este es Firebase de Google.**
- **SaaS (Software as a Service)**: En esta arquitectura se cuenta con poco control en el estado de red, en los servidores, en el almacenamiento, sistema operativo, datos, ni casi nada porque es un sistema que se maneja prácticamente solo, sin necesidad de intervenciones manuales. **Un ejemplo de este son Google Drive y Slack.**

La arquitectura de servidores más conveniente a utilizarse durante el desarrollo de un software es la **PaaS**, proporcionando un control medio del servidor, ya que no siempre es necesario involucrarse en conceptos de bajo nivel como los recursos de red, storage, etc. En esta arquitectura se gestiona la **aplicación (osea el código)** y la **data (osea las bases de datos)**.

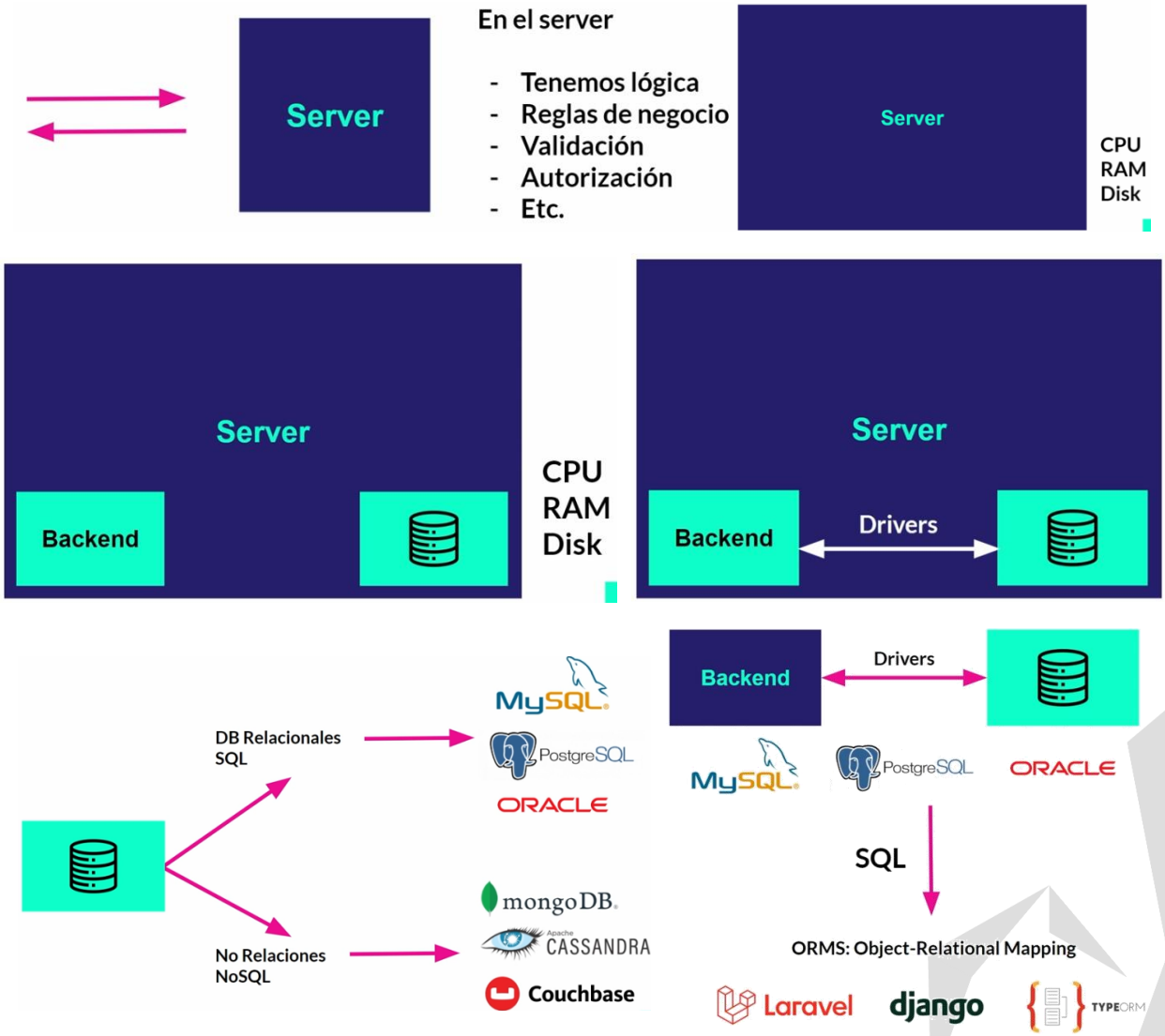


Heroku

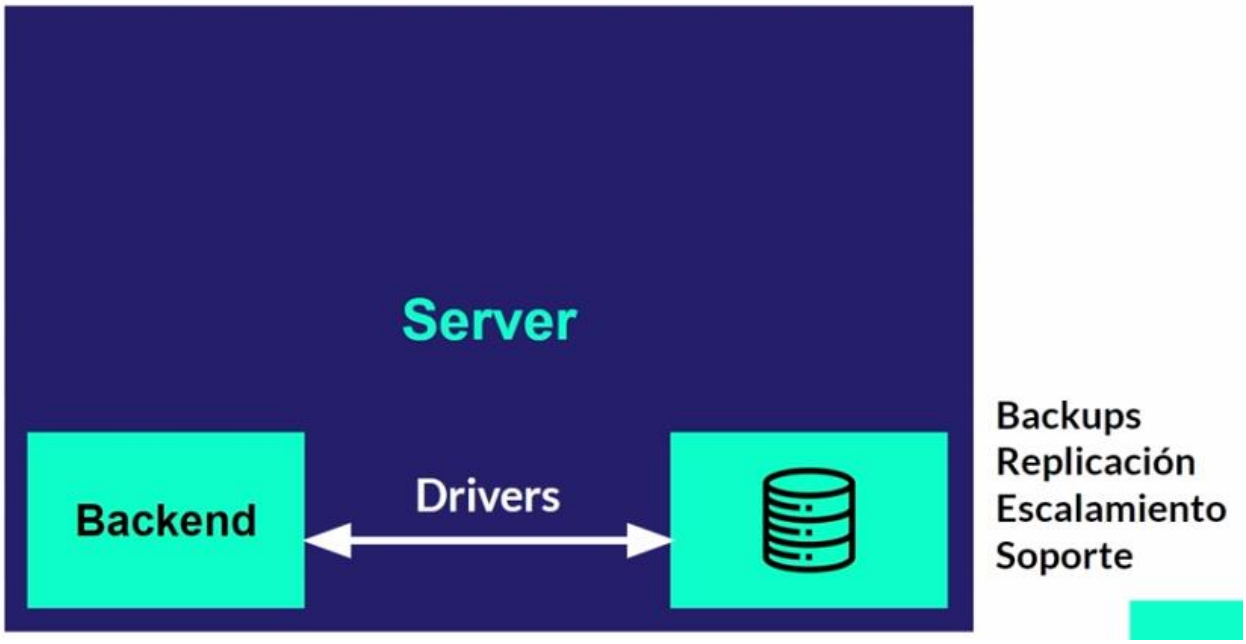
Existen herramientas con arquitecturas que ya están configuradas y manejadas como **PaaS (Plataforma como Servicio)**, una de ellas es Heroku, que es un sistema donde podemos desplegar (deploy) nuestro código y toda la parte de configuración de servidores se proporciona como parte de su servicio, dejando que nosotros solo nos preocupemos del **código (aplicación)** y la **base de datos (data)**.

Bases de Datos

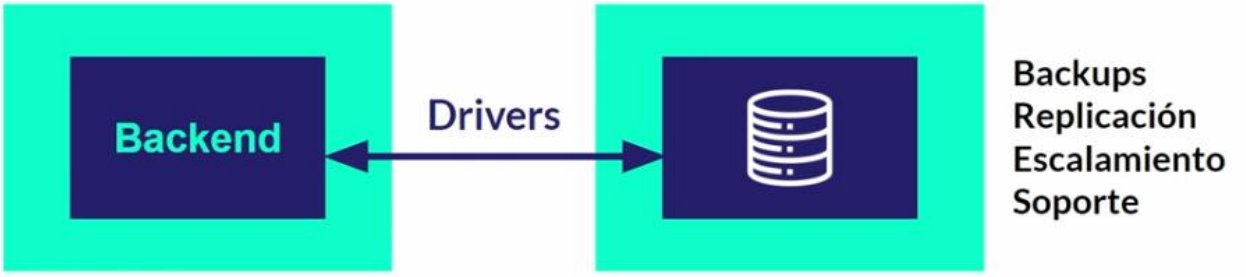
Las bases de datos son plataformas intermedias entre un servidor y un cliente, ya que cuando el cliente interactúa con el servidor, se pueden ingresar datos importantes y relevantes con su operación, pero como este cuenta con recursos limitados como CPU, RAM, Disco Duro, etc. los cuales son utilizados para realizar su operación a través de sus servicios de backend como APIs (no importando si se realiza con uno o varios frameworks o lenguajes de programación), dichos datos son pasados a elementos llamados databases a través de un servicio de base de datos que de igual forma se encuentra dentro del servidor, almacenándolos de forma intermedia, esta operación entre el backend y la base de datos se realiza a través de algo llamado Drivers, donde cada base de datos específica, ya sea relacional o no relacional y dependiendo de igual forma de su tipo, tiene su propio driver, el cual puede estar adaptado para conectarse a distintos lenguaje de programación en el backend. Un ejemplo de esto es como la base de datos de PostrgreSQL tiene distintos drivers para conectarse a código Python, JavaScript, PHP, C# .NET, etc. hasta pudiendo utilizar métodos orientados a objetos para extraer datos a través de un lenguaje orientado a objetos llamado ORM (Object Relational Mapping).

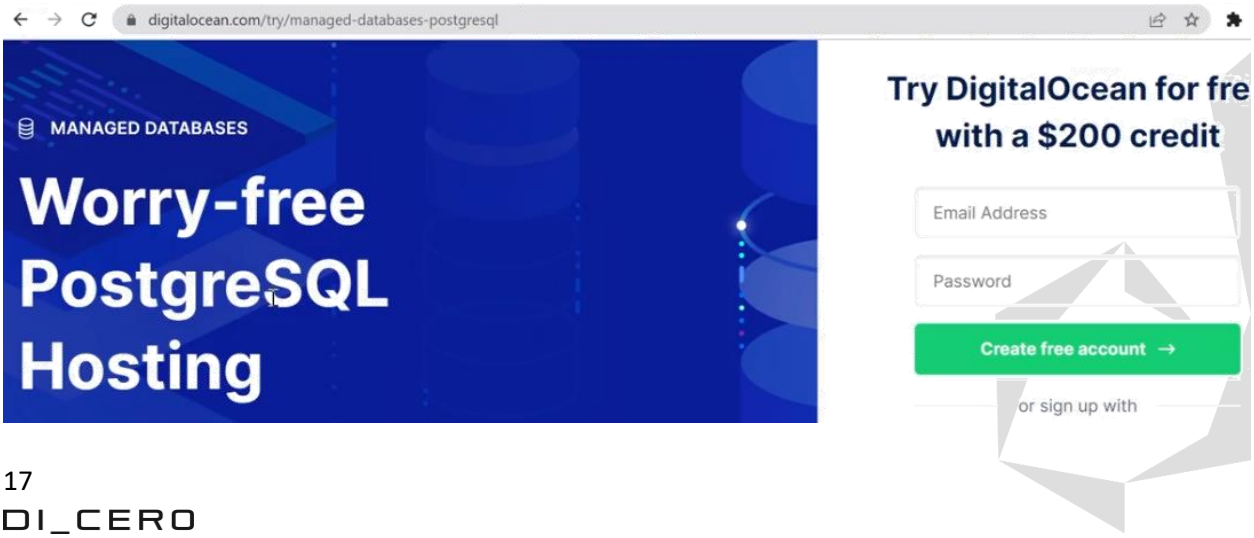


Cuando se haya recabado un gran número de datos, estos pueden ser replicados hacia un data warehouse, para luego analizarlos y realizar operaciones como machine learning, business intelligence, etc. Esto si entra dentro del rol de backend, pero los que realizan tareas de backups y soporte son correspondientes al rol de DB Admin.



Aunque existen servicios que pueden realizar todo este tipo de tareas de forma automática, delegando esta parte de administración de la base de datos. Como por ejemplo Heroku, Digital Ocean, MongoDB Altas, Coucbase, etc. que de igual forma proporciona este tipo de servicios.

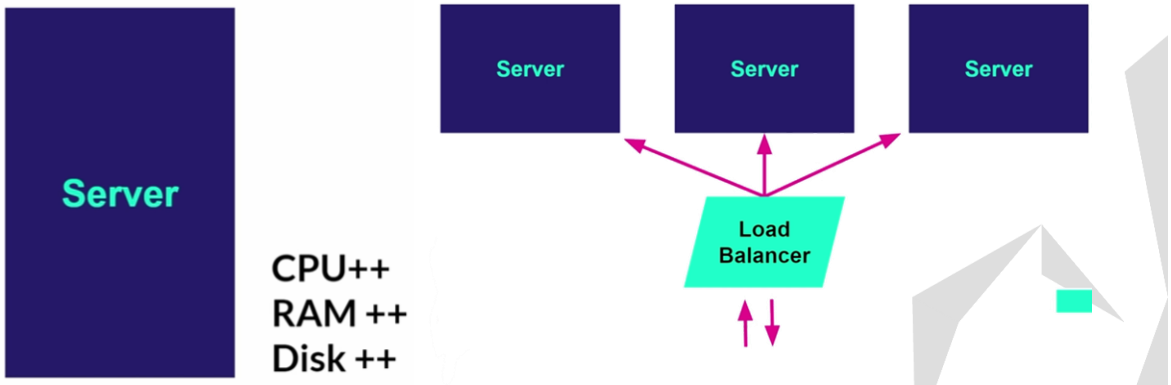




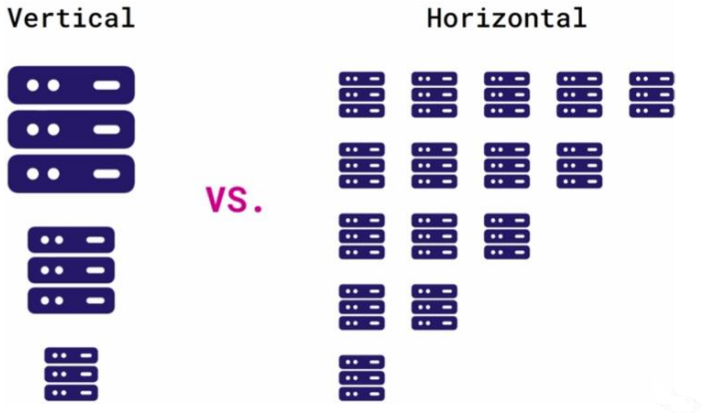
Escalamiento Vertical VS. Horizontal

Recordemos que nuestros servidores tienen limitaciones físicas como su CPU, memoria RAM, Disco Duro (o de estado sólido), etc. pero si nuestras aplicaciones empiezan a tener muchos usuarios, esto hará que comiencen a colapsar porque sus recursos serán insuficientes.

- **Escalamiento Vertical:** Una forma de resolver problemas de escalabilidad al aumentar el número de usuarios en nuestra aplicación es aumentar los recursos de nuestro servidor, a esto se le llama ***escalamiento vertical***.
    - *Causa problemas de disponibilidad*, ya que, si el servidor que ha sido escalado verticalmente falla, todo el sistema de producción se viene abajo.
    - Esto es fácil de implementar, pero puede resultar muy costoso mientras vaya aumentando el número de usuarios.
    - Es difícil manejar demandas fluctuantes como por ejemplo cuando un ecommerce requiere aumentar su capacidad en fechas específicas como en un buen fin.
- **Escalamiento Horizontal:** Otra forma de resolver problemas de escalabilidad (al aumentar el número de usuarios en nuestra aplicación) es usar más de un servidor que soporte nuestra aplicación, teniendo réplicas de nuestros servicios en cada uno o teniéndolos distribuidos, a esto se le llama ***escalamiento horizontal*** y al conjunto de servidores se le llama **cluster**.
    - *No causa problemas de disponibilidad*, ya que, si uno de los servidores cae, los demás pueden soportar su función, aunque si sufrirían un estrés de memoria RAM por cubrir al servidor caído, pero todo el sistema no fallaría.
        - **Load Balancer:** Cuando se utiliza el escalamiento horizontal, existe un servidor llamado load balancer que se encarga de gestionar las peticiones hechos a los servicios distribuidos en los diferentes servidores que soportan nuestra aplicación (**cluster**), teniendo conocimiento de cuáles son los servidores conectados a dicha red y para qué sirve cada uno.
    - Resulta complicado de implementar, pero su costo es dinámico, ya que se tienen recursos no tan elevados en cada servidor, los cuales tienen réplicas de los servicios de la aplicación, pudiendo así mantener un aumento de número de usuarios sin gastar tanto en un solo servidor, sino en varios con pocos recursos.
    - Es fácil manejar demandas fluctuantes al poder variar fácilmente el número de servidores que contendrán réplicas del servicio en fechas específicas, para luego poderlos dar de baja.

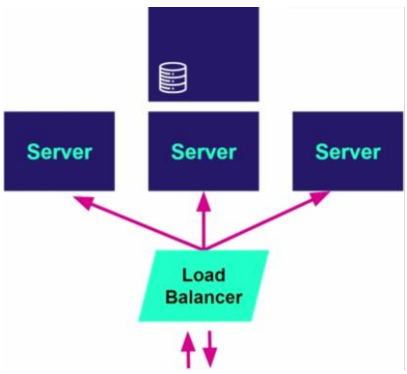


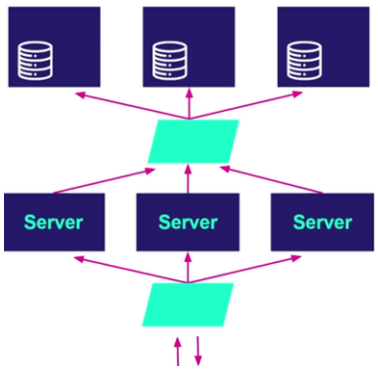
Cabe mencionar que en el escalamiento horizontal existe un tipo de problema cuando cada uno de esos servidores contenga la base de datos contenida dentro de ellos, ya que, el servidor 1 no tendrá la misma base de datos que el servidor 3, por lo que el load balancer cuando reciba requests, dependiendo de a que servidor se conecte para resolver la petición, cada uno contendrá datos distintos, pero esto se puede resolver cuando la base de datos esté fuera de nuestra arquitectura y sea dada por un proveedor, como MongoDB, Oracle, etc. Pero si estamos gestionando la base de datos dentro de la arquitectura de nuestros servidores escalados horizontalmente, si nos podemos encontrar con este problema y se necesitaría ejecutar una replicación.

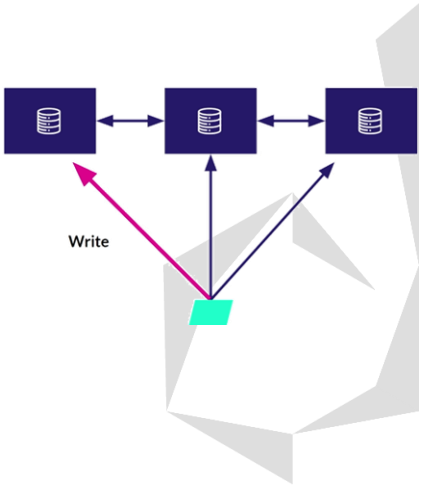


Escalamiento Horizontal: Heroku y Replicación de DataBases

Recordemos que a un **grupo de servidores con escalamiento horizontal** se le llama **cluster**, pero, así como se describió anteriormente, *si cada uno de los* ***servidores del cluster tiene incluida una base de datos diferente****, esto causa un problema*, ya que cada una contendrá datos distintos y el **Load Balancer** no podrá manejar las peticiones de los usuarios de nuestra aplicación de forma correcta, una de las soluciones que se podría implementar es que uno de los servidores esté dedicado a contener la única base de datos de nuestro sistema, el problema con esto es que se crearía un cuello de botella, ya que todos los servidores harían peticiones a este solo, aunque para mejorar esto se podría aumentar los recursos de dicho servidor, aplicando un **escalamiento vertical** solo en ese, sin embargo, esto causaría problemas de disponibilidad de la base de datos, ya que si ese servidor cae, toda la base de nuestra aplicación fallaría y aunque aplicáramos en estas un escalamiento horizontal, regresaríamos a nuestro problema inicial, donde cada base de datos tendría datos distintos. Para casos como estos es que existe la **replicación**, que es simplemente un proceso en donde las bases de datos están sincronizadas, asegurándonos que se tengan los mismos datos replicados en todas las databases de nuestra red de servidores.

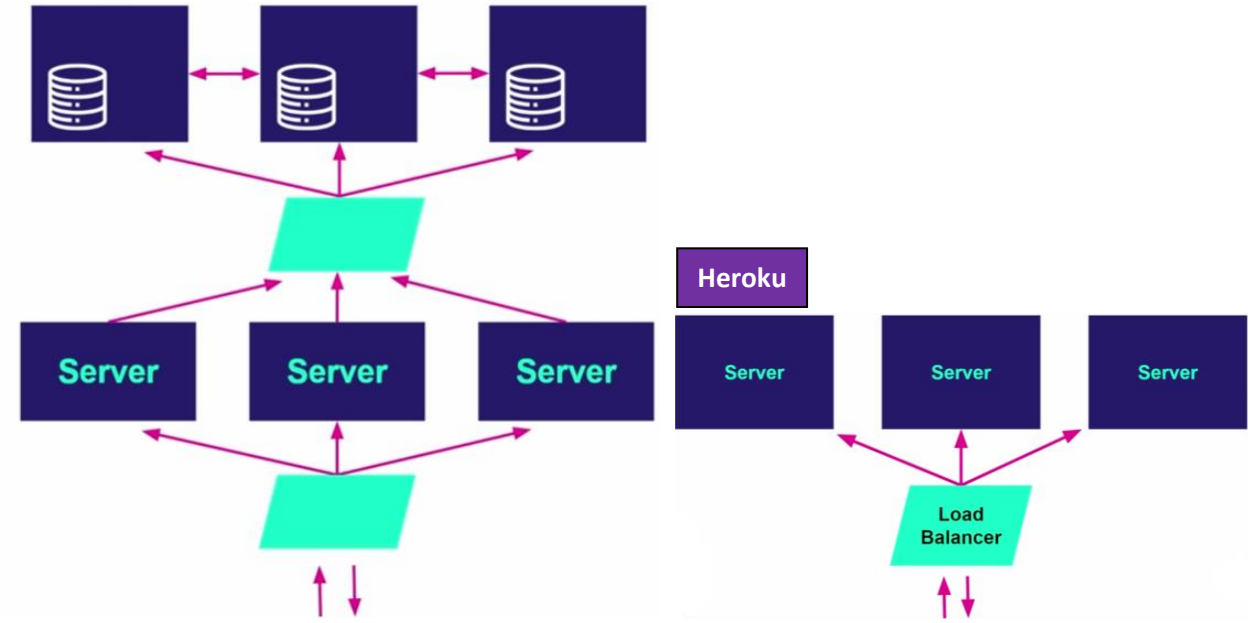






El objetivo de la **replicación** es que cuando se quiera **leer** los datos de cualquiera de los servidores de las **databases**, todos contengan los mismos datos, así, cuando se realice una operación de **escritura** o **lectura** hacia cualquiera de ellos, las **bases de datos** de los **distintos servidores** estén **sincronizados** para que cada una de ellas **contengan los mismos datos**.

Como backend developer no nos enfocaremos tanto en **la replicación**, **eso es más trabajo de los administradores de bases de datos**, pero si nos debemos encargar de la capa de servidores, donde se escala de forma horizontal los servicios de la aplicación en cada servidor individual, normalmente esto de igual forma se le delega a un proveedor de servidores como Heroku, ejecutando así una arquitectura horizontal de servidores de forma sencilla.



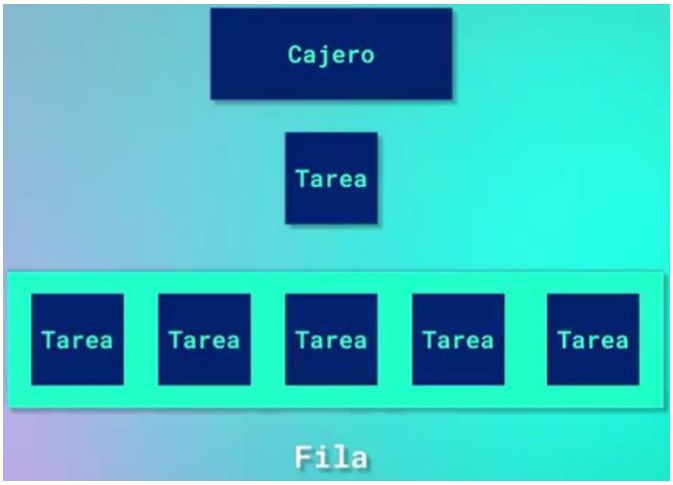
Cola de Tareas

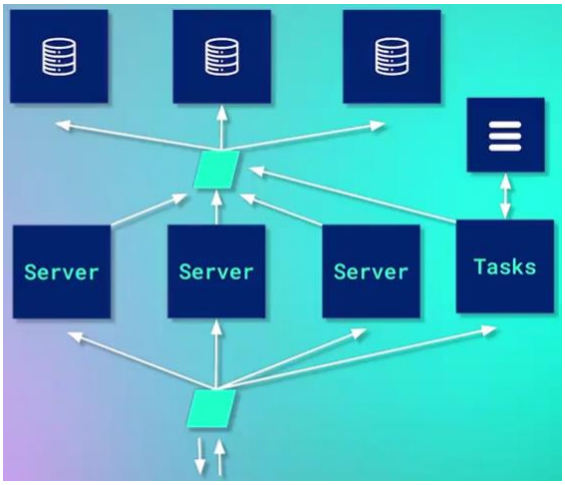
Las colas de tareas son una alternativa a sistemas que comparten datos como las APIs, ya que estas sirven para ejecutar funciones que pueden tardar tiempos largos de hasta 10 minutos, como consultas a bases de datos grandes, generación de reportes, creación de backups, generación de gráficos, archivos pdf, zip, cvs, etc. y ningún usuario se va a quedar tanto tiempo frente a su computadora esperando a que su navegador (cliente frontend) termine de ejecutar esa petición. Las diferencias principales entre una API y una cola de tareas es la siguiente:

- **APIs:** Un request se ejecuta lo más pronto posible y responde por el mismo medio.
    - Ejemplo: Cuando se hace una petición por medio de Insomnia al endpoint de una API, la respuesta de este request es inmediata y su resultado se observa en la misma interfaz de usuario de Insomnia.
- **Task Queue:** Una cola de tareas eventualmente ejecuta un proceso y puede responder por otros medios.

- Ejemplo: En Facebook se puede solicitar que se realice un respaldo de toda la información de nuestro perfil en un archivo zip, incluyendo fotos, conversaciones, posts, etc. Se me dirá cuanto tiempo se tardará este proceso y el resultado lo recibiré en un email.

El medio y el tiempo de respuesta son las principales diferencias entre una **API** y una **Cola de Tareas (Task Queue)**. La forma en la que las colas de tareas funcionan es que *las tareas se apilan una tras otra y se van ejecutando una por una en el orden que van llegando, ejecutándolas todas eventualmente*, así como se realiza en el cajero de un banco. Normalmente **en el cluster de un sistema de escalamiento horizontal**, **uno de los servidores está dedicado a ejecutar las colas de tareas de forma independiente** y el load balancer sabe cuándo ejecutar este servidor en específico.

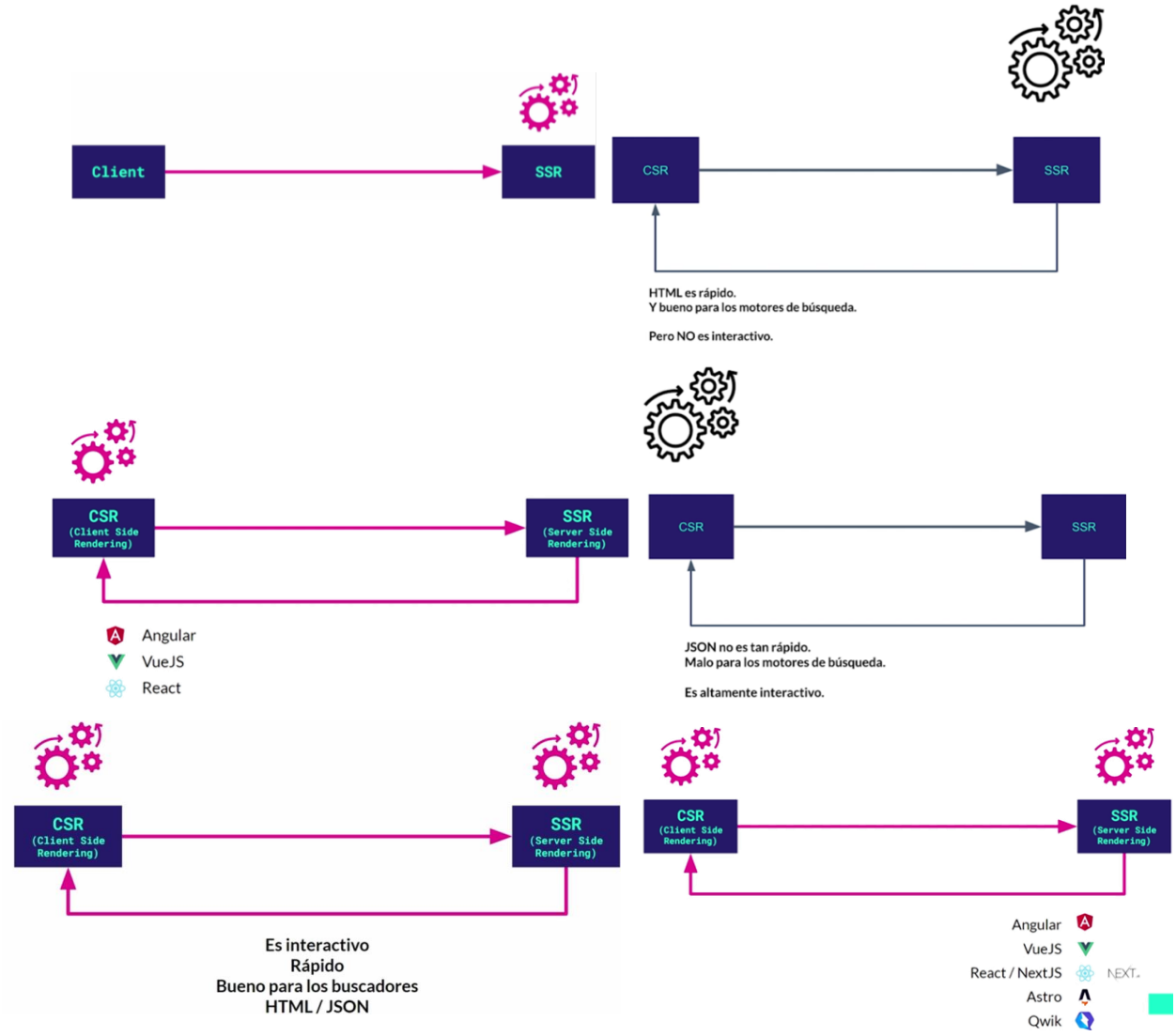




Server Side Rendering (SSR)

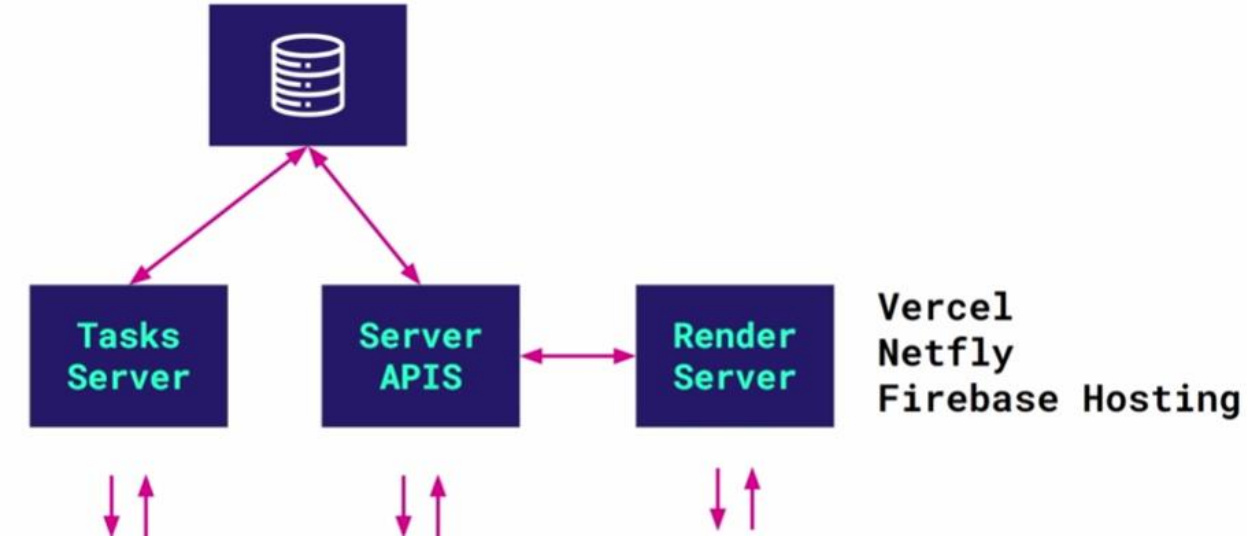
El **renderizado (rendering)** en términos generales se refiere a **generar una representación visual o auditiva a partir de datos**, los cuales se pueden encontrar en formatos **JSON, XML, HTML, etc.** ya sea **a través de dispositivos móviles o navegadores web, los cuales son clientes** que interpretan dichos datos para generar elementos estéticos a partir de ellos. Esto de igual manera puede ser realizado del lado del servidor, para que así la representación visual ya llegue interpretada hacia el cliente, logrando así una repuesta más rápida, pero con la desventaja de que esta no es interactiva, esto implica que para que cualquier cambio de datos pueda ser visualizado, se deberá recargar el cliente, ya sea el dispositivo móvil o navegador web. Las ventajas y desventajas de ambos son:

- **CSR (Client Side Rendering):** Es una técnica donde el navegador renderiza una página HTML mínima y usa JavaScript (React, VueJS, Angular, etc.) para construir la interfaz de usuario de forma dinámica con los datos extraídos usualmente en formato JSON.
- **SSR (Server Side Rendering):** Es una técnica donde el servidor genera y entrega al navegador una página HTML completa, que ya contiene los datos necesarios. Esto mejora el rendimiento de los motores de búsqueda y la indexación SEO. Luego el navegador hace la página interactiva usando JavaScript del lado del cliente. Frameworks como Next.js, React.js, Vue.js o Angular permiten usar lo mejor de ambos mundos, consumiendo APIs del lado del servidor para generar un HTML listo y manteniendo la interactividad mediante JavaScript del lado del cliente.



Debido a esta situación con los SSR, se puede asignar un servidor dentro de nuestra **arquitectura de escalamiento horizontal** dedicado al renderizado del frontend, teniendo

- **Task Server:** Un servidor dedicado a ejecutar colas de tareas, las cuales son funciones que toman más de 10 minutos en ejecutarse y devuelven su resultado en otros medios.
- **Server APIs:** Un servidor de APIs que ejecuta su función de forma inmediata y devuelve sus resultados por el mismo medio.
- **Database Server:** Un servidor que puede ser replicado en otros para que se implemente de igual forma una arquitectura horizontal y todos ellos estén sincronizados para devolver los mismos datos, proporcionando así disponibilidad de datos.
- **Render Server:** Un servidor que esté enfocado en utilizar frameworks como Angular, Vue.js, React.js, etc. para proporcionar un renderizado del lado del servidor (SSR), dando como resultado sitios web o aplicaciones móviles más veloces, utilizando servivios como Vercel, Netfly o Firebase Hosting para su implementación.



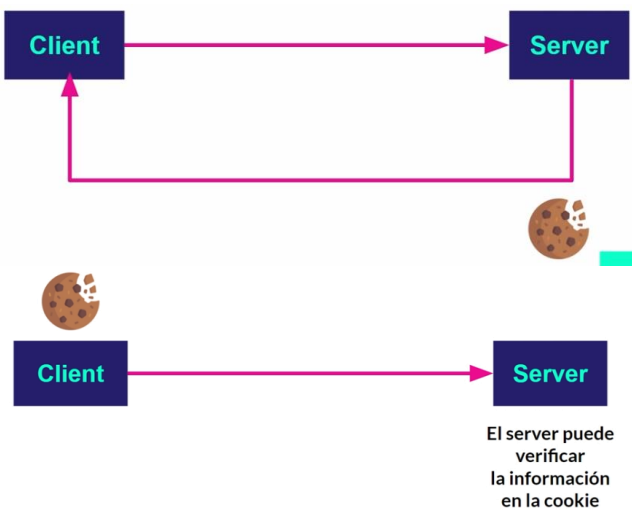
Memorias

Cookies y Sesiones

Las cookies y las sesiones son tecnologías que almacenan información de los usuarios en los navegadores para reconocerlos de forma rápida, brindando así una mejor experiencia en nuestras aplicaciones (código subido a la nube). Las cookies pueden almacenar datos como país de origen, el idioma que el usuario seleccionó previamente al desplegar la información de la aplicación, mantener una sesión (login con usuario y contraseña) abierta, etc.

La forma en la que esto se realiza es a través del navegador (cliente) hacia el servidor:

Donde primero **el cliente realiza una solicitud al servidor** → Luego **el servidor le manda una cookie con los datos que quiere recabar** sobre el cliente → **La cookie se queda almacenada en el navegador** con los datos almacenados del usuario → Y finalmente **esa cookie es retornada al servidor con los datos que extrajo del navegador** siempre que el cliente le haga solicitudes a la nube que tiene almacenado nuestro servicio.



El problema con esto es que solo funciona en servicios web, si esto se quiere realizar en aplicaciones móviles nativas Android o iOS o con microcontroladores, esta tecnología no funciona. El equivalente para este tipo de clientes es algo llamado JWT (JSON Web Token), la cual nos permite validar por medio de tokens dicha información. Aunque cabe mencionar que los JWT si se pueden utilizar también con aplicaciones web.

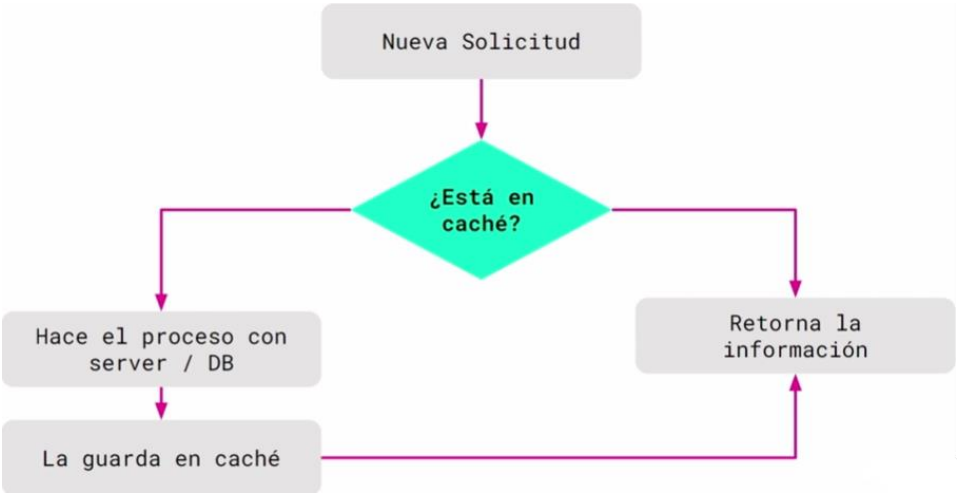
Memoria Caché

Vamos a explicar este concepto con un ejemplo cotidiano, si te pregunto por primera vez, cual es el resultado de la suma 955 + 844, tendrás que procesar la información y aproximadamente te tardarás 30 segundos en contestar la pregunta, pero si te la hago una segunda vez, el tiempo de procesamiento se reduciría a 2 segundos porque no se tiene que hacer el proceso. Así funciona la Caché, es una memoria que almacena el resultado de un proceso o procedimiento previamente ejecutado.

La forma en la que funciona la memoria caché, la cual es un sistema de almacenamiento interno de corto plazo en nuestro servidor es la siguiente:

1. Se hace una nueva solicitud de un proceso.
2. Se confirma si se tiene almacenado el resultado de este proceso en la memoria cache.
    a. Si no es así se hace el proceso en el servidor.
    b. Se guarda su resultado en la memoria caché.
    c. Se retorna la información al cliente.
3. Si se tiene almacenado el resultado, simplemente se retorna la información al cliente.

Nota: Hay que tener en cuenta que los datos de la memoria caché tienen un tiempo de expiración, ya que es una memoria volátil de corto plazo.



Los sistemas de caché es muy recomendable implementarlos en sitios donde se puede ejecutar una misma consulta varias veces, como en ecommerce, noticias, blogs, etc. pero no es recomendable implementarse en chats o sistemas que muestran datos en tiempo real, ya que podría ocasionar problemas. De igual forma se podría utilizar para evitar hackeos, ya que, si una persona maliciosa quisiera ejecutar varias solicitudes a un mismo endpoint causando así un **ataque de negación de servicio DDOS**, esto no ocasionaría ninguna falla del sistema, ya que la memoria caché se encargaría de gestionar dicha consulta.

Referencias

Platzi, Nicolás Molina, “Curso de Introducción al Desarrollo Backend”, 2018 [Online], Available: https://platzi.com/home/clases/4656-backend/56005-los-roles-del-desarrollo-backend/