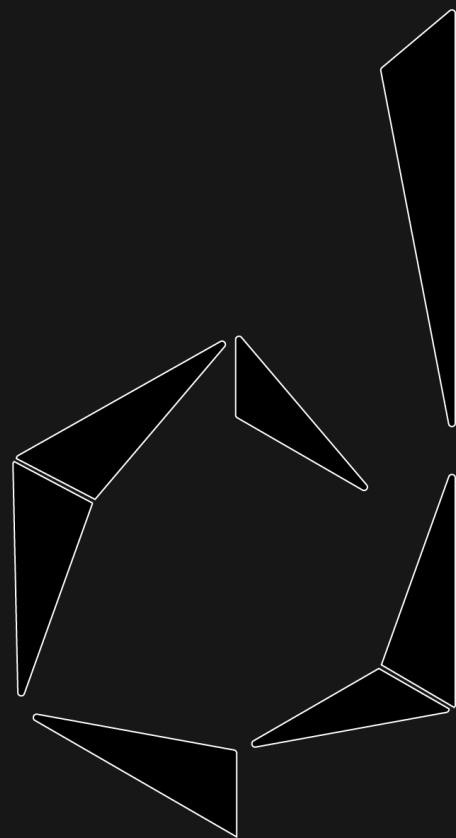


INGENIERÍA MECATRÓNICA



DI_CERO

DIEGO CERVANTES RODRÍGUEZ

PROGRAMACIÓN: DESARROLLO BACKEND

POSTMAN, INSOMNIUM, PYTHON, ETC.

Ejemplo de
Arquitectura Distribuida

Contenido

Backend Developer	3
Backend, Frontend y Fullstack Developer	3
Cómo se Construye el Backend: Concepto de API	4
HTTP: HyperText Transfer Protocol	5
Estructura REST: API con Protocolo HTTP	8
API Testing: Insomnia y Postman	9
Cloud (Nube).....	14
DevOps.....	15
Modelos de Arquitectura de Servidores: On Site, IaaS, PaaS y SaaS	16
PaaS: Heroku	17
Bases de Datos	18
Escalamiento Vertical VS. Horizontal.....	20
Escalamiento Horizontal: Heroku y Replicación de DataBases	21
Cola de Tareas: Queues	23
Server Side Rendering (SSR).....	23
Memorias.....	25
Memoria Cliente: Cookies/Sesiones	25
Memoria Servidor: Caché	26
Memoria Servidor: Proxy	27
Memoria Nube (Servidor): Bucket	28
Arquitectura y Despliegue	29
Docker: Contenedores e Imágenes.....	29
Kubernetes: Gestión de Contenedores Docker	30
Servidores MultiNube:	30
Proyecto Backend.....	40
Definición de los Requerimientos del Negocio	40
Documento de Diseño de Software: High Level System Design	41
Elaboración de la Arquitectura del Sistema:.....	49
Diseño de Bajo Nivel: Planes de Prueba TDD e Integración Continua CI/CD	52
Representación de las Bases de Datos: Definiciones y Diagrama Entidad-Relación	53
Plan de Integración Continua: CI/CD - Continuous Integration/Continuous Development	55

Seguimiento de Tareas en DevOps: Code Complete en Trello, Azure Dev Ops, etc.	57
Desarrollo del Código del Proyecto	57
Creación de las entidades de la base de datos	57
Referencias.....	59



Backend Developer

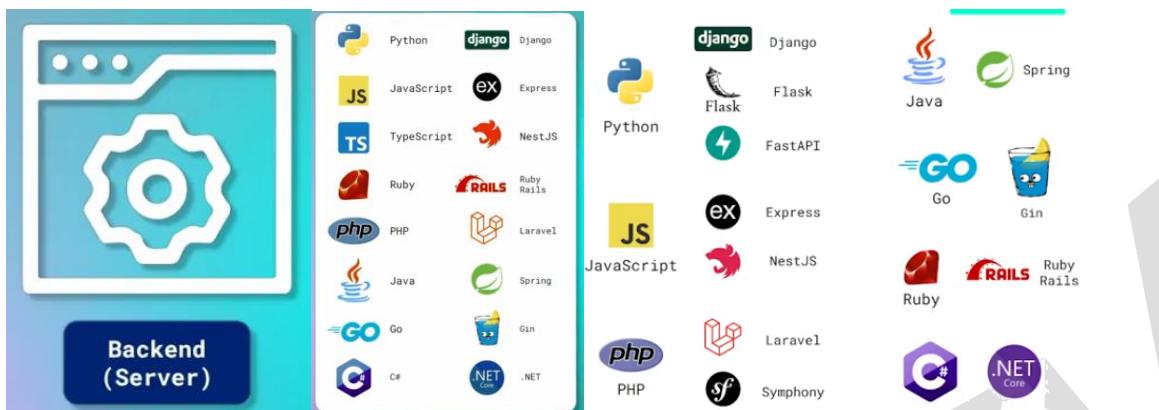
El rol principal del **backend developer** es escribir código que tenga que ver con reglas de negocio, ya sea validación, autorización de usuarios, conexión a bases de datos, manejo de datos, etc. El cual se montará sobre un servidor y será expuesto a millones de usuarios. Sin embargo, como backend también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



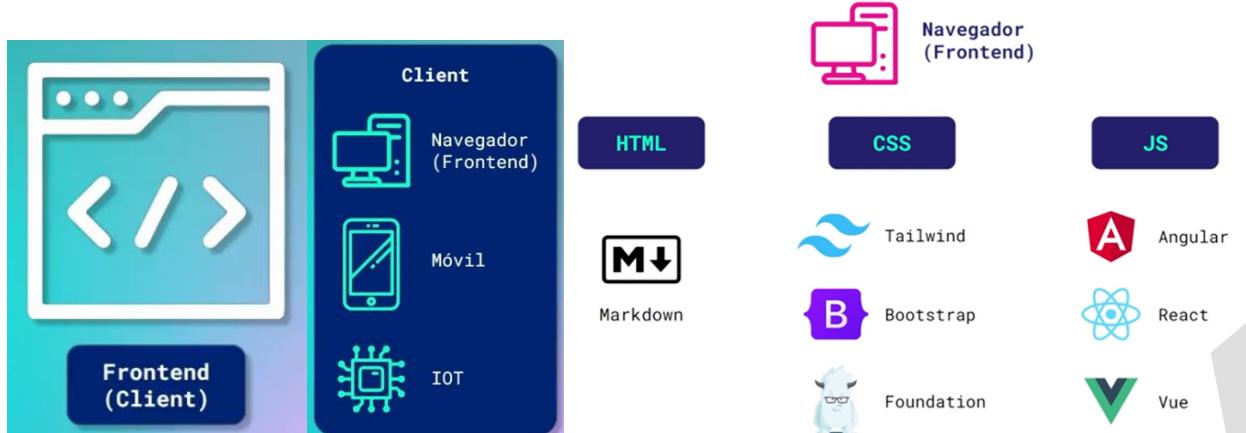
Backend, Frontend y Fullstack Developer

- **Backend Developer:** Desarrolla y pone a disposición **servicios** en los cuales los clientes (**plataformas frontend como páginas web, aplicaciones móviles o microcontroladores**) se conectan para **extraer datos** y mostrarlos en su interfaz gráfica o utilizarlos de alguna manera.
 - **Los lenguajes más populares y sus frameworks para desarrollar 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 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 provea 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 definir 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:** Estos 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 reciben código Java o Kotlin y solo pueden ser utilizados en dispositivos con sistema operativo Android.
 - **Cross Platforms:** 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.

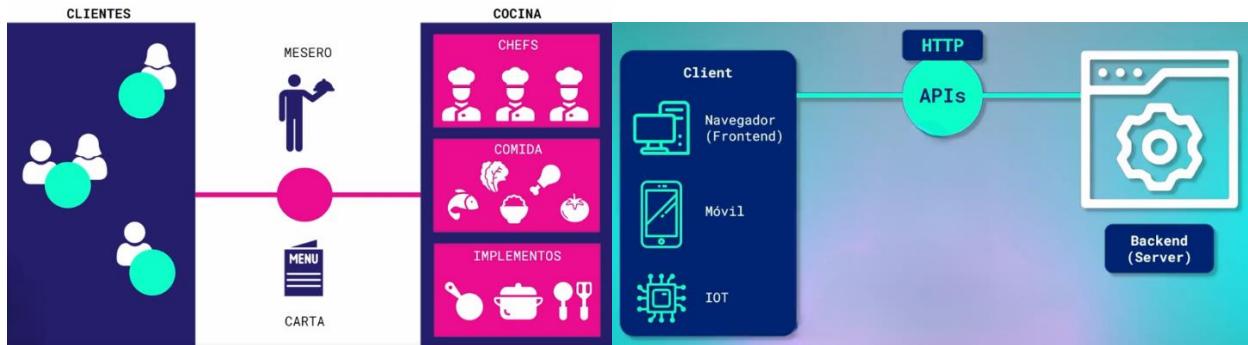


Cómo se Construye el Backend: Concepto de API

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 web, teléfono o microcontrolador** utilizado por un usuario (que *simboliza un cliente frontend*), el cual va a solicitar utilizar un servicio proporcionado por el backend (cocina).
- **Carta o Menú:** Esto representa la interfaz gráfica, ya sea una **pantalla de sitio web, aplicación móvil o interfaz IoT** con la cual el *cliente (navegador, teléfono o microcontrolador)* puede ver el listado de funciones que tiene disponibles para solicitar.
- **Mesero:** Representa la **API**, que se encarga de recoger la solicitud del comensal ya que haya visto la *carta (interfaz gráfica)* y pedir que se prepare esto en *la cocina (el backend)*.
- **Cocina:** Representa el **backend completo del servicio**, el cual utiliza las siguientes herramientas 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 *cuentan con datos que hayan sido proporcionados por el frontend*, ya sea de forma manual o que previamente hayan sido almacenados 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 *poder 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 otros *a través del protocolo HTTP* haciendo uso de **endpoints** para recibir un código o dato en formato JSON o XML que sea renderizado (interpretado) por el frontend.

HTTP: HyperText Transfer Protocol

El **protocolo HTTP** se conforma de una **URL** que nos ayude 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:** 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 (**Domain Name Server**) que nos **permite tener la propiedad de una dirección IP** numérica, la cual por motivos de facilidad veremos en forma de palabra en vez de números, a esto se le llama **dominio** y es como la gente podrá acceder a nuestro sitio web desplegado (deployado) 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 nos **permite acceder a un dato en específico, el cual 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.

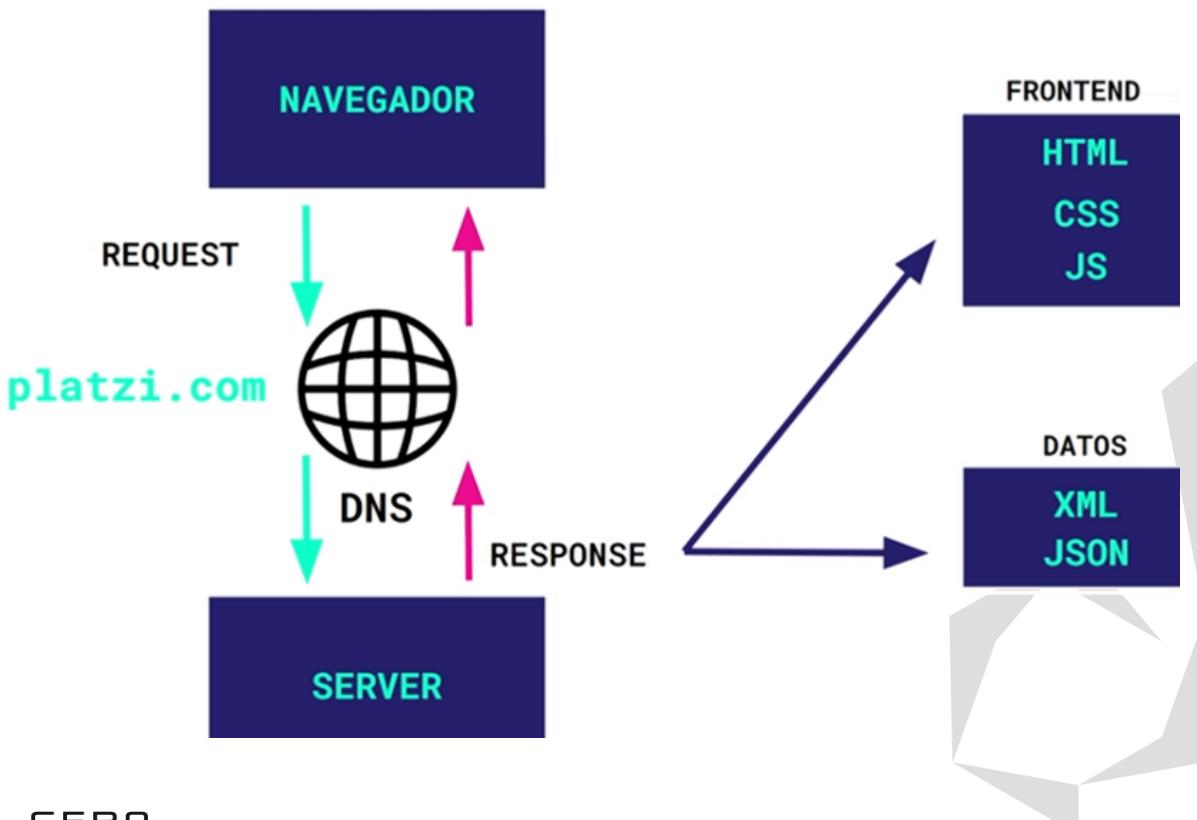


- **Status HTTP:** Algo muy interesante de los **protocolos HTTP** son sus **códigos de estatus**, estos nos **dan información para saber el estado en el que se encuentra la petición que se realizó del cliente hacia el servicio**. Su significado se encuentra categorizado por rango, el cual será explicado a continuación:

Rango de HTTP Status	Definición	Estados HTTP Más Comunes
100-199	Estos indican información de estado del servidor hacia el cliente .	100 - Continue: Solicitud recibida, continuando proceso. 101 - Switching Protocols: Inicio de cambio de protocolo. 102 - Processing: El servidor está procesando la petición.
200-299	Este es el rango más utilizado porque denota éxito en alguna acción del servidor .	200 - Ok: Petición ejecutada exitosamente. 201 - Created: Recurso creado. 202 - Accepted: Solicitud aceptada. 204 - No content: La acción fue exitosa pero no retornó ningún dato.
300-399	El rango de redirección indica que algún recurso ha sido movido de lugar .	301 - Moved permanently: El recurso fue movido de forma permanente. 302 - Found: Redirección temporal. 304 - Not modified: El recurso no ha sido modificado desde la última solicitud.

400-499	Representan errores de la parte del cliente, esto normalmente ocurre porque el cliente envió una solicitud de forma errónea.	400 - Bad Request: Solicitud hecha de forma errónea. 401 - Unauthorized: No autorizado. 403 - Forbidden: Credenciales válidas, pero acceso denegado. 404 - Not Found: Recurso no encontrado. 409 - Conflict: Conflicto con el estado actual del recurso. Por ejemplo, datos duplicados.
500-599	Representan errores que ocurren en el código que está ejecutándose del lado del servidor o en el servidor en sí.	500 - Internal Server Error: Error interno del servidor. 501 - Not Implemented: Método no soportado por el servidor. 502 - Bad Gateway: El servidor actuó como intermediario (API gateway o proxy) y recibió una respuesta inválida o ninguna respuesta del otro servidor al que intentaba conectarse (servidor upstream). 503 - Service Unavailable: El servidor no puede manejar la solicitud porque está temporalmente sobrecargado o en mantenimiento. 504 - Gateway Timeout: 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 procesados por un cliente web, smartphone o un microcontrolador.

La forma en la que funcionan las REST APIs es proporcionando 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 para *mandar data* a un servidor.
- **Read** = Método **Get** del protocolo HTTP para *recibir datos* de un server.
- **Update** = Métodos **Put/Patch** del protocolo HTTP para *editar datos* existentes.
 - **Put**: Edita **todos los datos** existentes de un recurso.
 - **Patch**: Edita **solo los datos necesarios** de un recurso.
- **Delete** = Método **Delete** del protocolo HTTP para *borrar datos* de un servidor.
 - **Options** = Método **Options** para consultar qué métodos HTTP están permitidos en un endpoint del servidor.

domain.com	CRUD
domain.com/users	CREATE = POST
domain.com/tasks	READ = GET
domain.com/companies	UPDATE = PUT / PATCH
	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	<input checked="" type="checkbox"/>	201

API Testing: Insomnia y Postman

Utilizaremos la herramienta de **Insomnia** para **testear una API** de pruebas perteneciente a 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.

The screenshot shows the 'Categories' endpoint documentation. On the left, there's a sidebar with navigation links like 'About', 'REST API', 'GraphQL', and 'Resources'. The 'Categories' link under 'REST API' is highlighted. The main content area has a title 'Categories' and a subtitle 'Retrieve a list of all available categories.' Below this is a 'Request' section showing a GET request to 'https://api.escuelajs.co/api/v1/categories'. The 'Response' section displays a JSON array of category objects, with one item shown in detail: { "id": 1, "name": "Clothes", "slug": "clothes", "image": "https://placehold.co/600x400" }. To the right, there's a sidebar titled 'On this page' with links to other API endpoints.

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

The screenshot shows the Insomnia download page. It features a large 'Download for Windows' button at the top. Below it, there's a note about agreeing to the Privacy Policy and Terms. A 'What's New? Changelog' link is also present. To the right, a portion of the Insomnia application interface is visible, showing the 'DOCUMENT' tab with an 'ENDPOINTS' list and a JSON editor pane containing a complex API response structure.

En la **interfaz de Insomnia** se tiene lo siguiente para analizar las **peticiones API REST HTTP**:

- **Environment, Collection, and Document:** Del lado superior izquierdo podemos ver:
 - **Environment:** Contiene variables que se vayan a reutilizar en las operaciones de los endpoints, como URLs base o tokens de autenticación.
 - **Collection:** Es una carpeta que agrupa endpoints con peticiones parecidas.
 - Estos se pueden importar o exportar en la página principal (home) de Insomnia, accediendo a la opción de ... → Import/Export → Import Data/Export Data → Esta acción exportará todos los endpoints con sus configuraciones en un archivo .json o .yml.
 - **Document:** Es un espacio de trabajo que contiene colecciones y environments.
 - **Barra Lateral:** Debajo de la sección anterior, igual 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**.
 - **Crear un nuevo request:** Para crear un nuevo endpoint, debemos dar clic en el botón de + que se encuentra al lado derecho del **filtro de requests** y **seleccionar el método HTTP correspondiente**, seguido del nombre del URL base y el endpoint:
 - **POST:** Método HTTP para *mandar data* a un servidor.
 - **GET:** Método HTTP para *recibir datos* de un server.
 - **PUT:** Método HTTP para editar **todos los datos** existentes de un recurso.
 - **PATCH:** Método HTTP para editar **solo los datos necesarios** de un recurso.
 - **DELETE:** Método HTTP para *borrar datos* de un servidor.
 - Las **colecciones** nos permiten categorizar los endpoints que utilicemos en folders.
- **Sección media superior:** Aquí es donde se introducen los datos del endpoint que se vaya a utilizar.
 - **Método HTTP que se utiliza + el dominio base + 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 o 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 métodos **HTTP POST, PUT, PATCH o DELETE**, el método **GET casi siempre no recibe ningún dato para realizar su función**.
 - En esta parte se puede escoger el formato de los datos del body, ya sea en **JSON, XML, etc.**
 - **Auth:** Es el método de autenticación para identificar un usuario o y asignarle ciertos permisos de seguridad o no.
 - **Headers:** Son los datos del encabezado del body perteneciente al mensaje que se manda a la API.
 - **Scripts:** Para poder escribir código backend que recabe datos.

- **Docs:** Para visualizar la documentación del endpoint.
- **Sección lateral derecha:** En esta parte 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:
 - **HTTP status code:** El código de estado que devolvió la solicitud.
 - **Response time:** El tiempo que se tardó en recuperarla.
 - **Response Size:** 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.

The screenshot shows the Insomnia REST Client interface. At the top, there's a navigation bar with Application, File, Edit, View, Window, Tools, Help, and a search bar. On the right, there are user info and a 'Run' button. Below the bar, a request card is displayed: 'GET https://api.escuelajs.co/api/v1/categories'. The status bar shows '200 OK', '580 ms', '3.4 KB', and 'Just Now'. To the right of the status bar are tabs for Preview, Headers, Cookies, Tests, Mock, and Console. The Preview tab is active, showing a JSON response with multiple categories. A yellow arrow points to the 'Preview' tab, and another yellow arrow points to the response body.

```

[{"id": 1, "name": "Updated patito numero 4", "slug": "updated-patito-numero-4", "image": "https://i.imgur.com/0Kwzti1.jpg", "createdAt": "2025-05-22T18:33:45.000Z", "updatedAt": "2025-05-22T23:58:16.000Z"}, {"id": 2, "name": "Electronics", "slug": "electronics", "image": "https://i.imgur.com/7ANVnHE.jpg", "createdAt": "2025-05-22T18:33:45.000Z", "updatedAt": "2025-05-22T18:33:45.000Z"}, {"id": 3, "name": "Furniture", "slug": "furniture", "image": "https://i.imgur.com/0phac09.json", "createdAt": "2025-05-22T18:33:45.000Z", "updatedAt": "2025-05-22T18:33:45.000Z"}, {"id": 4, "name": "Shoes", "slug": "shoes", "image": "https://i.imgur.com/m00J1e.json", "createdAt": "2025-05-22T18:33:45.000Z", "updatedAt": "2025-05-22T18:33:45.000Z"}]

```

Al utilizar herramientas de testeo de APIs se acompaña de su documentación para observar los datos que se le deben enviar a los endpoints y los que debemos recibir, como se muestra a continuación:

The screenshot shows the Platzi Fake Store API documentation for the 'Categories' endpoint. The left sidebar has sections for About, REST API (Products, Categories, Users, Auth JWT, Locations, Files, Swagger Docs), GraphQL (Products, Categories, Users, Auth JWT, Playground), and Resources (Postman). The 'Categories' link in the REST API section is highlighted. The main content area has a title 'Categories' and a sub-instruction 'Retrieve a list of all available categories.' Below it, under 'Request', is a text input field containing '[GET] https://api.escuelajs.co/api/v1/categories'. Under 'Response', there is a JSON snippet showing a list of categories with fields like id, name, slug, and image.

```

[{"id": 1, "name": "Clothes", "slug": "clothes", "image": "https://placehold.co/600x400"}, {"id": 2, "name": "Electronics", "slug": "electronics", "image": "https://placehold.co/600x400"}, {"id": 3, "name": "Furniture", "slug": "furniture", "image": "https://placehold.co/600x400"}, {"id": 4, "name": "Shoes", "slug": "shoes", "image": "https://placehold.co/600x400"}]

```

GET https://api.escuelajs.co/api/v1/categories/44

```

1 | {
2 |   "path": "/api/v1/categories/44",
3 |   "timestamp": "2025-05-23T05:35:01.694Z",
4 |   "name": "EntityNotFoundException",
5 |   "message": "Could not find any entity of type \"Category\" matching: {n \"id\": 44\\n}"
6 |

```

GET https://api.escuelajs.co/api/v1/categories/43

```

1 | {
2 |   "id": 43,
3 |   "name": "New Category",
4 |   "slug": "new-category",
5 |   "image": "https://placeimg.com/640/480/any",
6 |   "createdAt": "2025-05-23T05:32:42.000Z",
7 |   "updatedAt": "2025-05-23T05:32:42.000Z"
8 |

```

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"
}
```

POST https://api.escuelajs.co/api/v1/categories/

```

1 | {
2 |   "id": 44,
3 |   "name": "di_cero Category",
4 |   "slug": "di-ero-category",
5 |   "image": "https://placeimg.com/640/480/any",
6 |   "createdAt": "2025-05-23T05:39:36.000Z",
7 |   "updatedAt": "2025-05-23T05:39:36.000Z"
8 |

```

POST https://api.escuelajs.co/api/v1/categories/

```
{
  "name": "di_cero Category",
  "image": "https://placeimg.com/640/480/any"
}
```

201 Created

POST https://api.escuelajs.co/api/v1/categories/

```
{
  "name": "di_cero Category"
}
```

400 Bad Request

```
{
  "message": [
    {
      "name": "Image should not be empty",
      "slug": "Image must be a URL address"
    }
  ],
  "error": "Bad Request",
  "statusCode": 400
}
```

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
}
```

POST https://api.escuelajs.co/api/v1/categories/

400 Bad Request

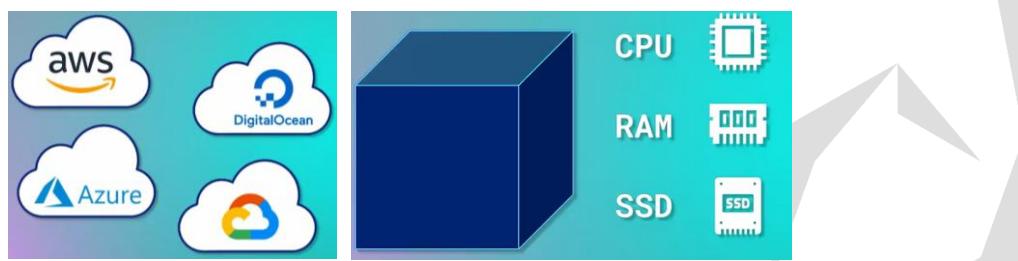
```
{
  "message": [
    {
      "name": "Image should not be empty",
      "slug": "Image must be a URL address"
    }
  ],
  "error": "Bad Request",
  "statusCode": 400
}
```

The screenshot shows two side-by-side interfaces. On the left is the **Platzi Fake Store API** documentation, which includes a sidebar with sections like About, REST API (Products, Categories, Users, Auth JWT, Locations, Files, Swagger Docs), GraphQL (Products, Categories, Users, Auth JWT, Playground), and Resources (Postman). The main content area is titled "Update a category" and shows a request example for a PUT request to `[PUT] https://api.escuelajs.co/api/v1/categories/{id}`. The request body is JSON with fields "name" and "image". On the right is the **Insomnia** API client interface, showing a successful `PUT` request to `https://api.escuelajs.co/api/v1/categories/43` with a response status of `200 OK` and a response body containing a single category object with ID 43, name "di_cero 2", slug "di-cero-2", image URL, creation date, and update date.

Nota: La gran diferencia entre los métodos HTTP **PUT** y **PATCH** es que **PATCH** permite cambiar solo uno de los datos del JSON, mientras **PUT** 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 **exponen nuestros servicios de código backend** (también llamados **granjas de servidores o data centers**), los cuales cuentan con elementos físicos como **número de CPUs, bytes de memoria RAM y capacidad memoria en Disco Duro**. En estos 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 (CSP Cloud Service Provider)** más conocidos son AWS (Amazon Web Service), Digital Ocean, Azure (Microsoft) y 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 ubicaciones estratégicas, dependiendo de dónde se encuentren físicamente mis clientes.



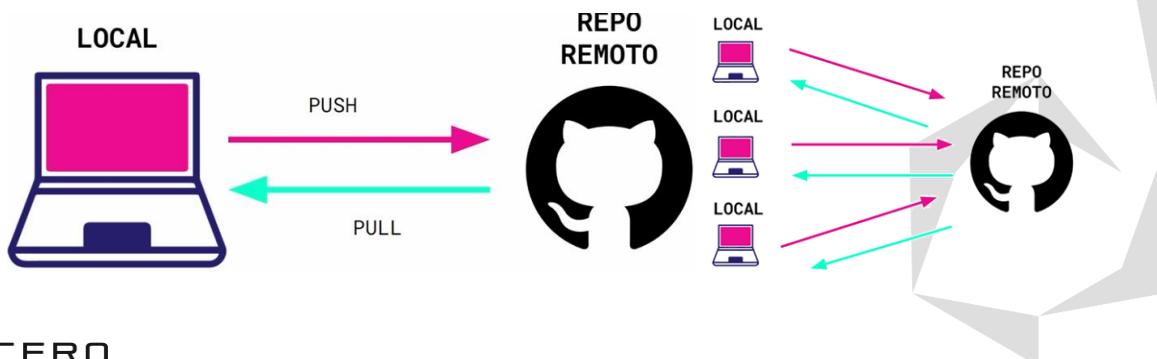
DevOps

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.

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



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

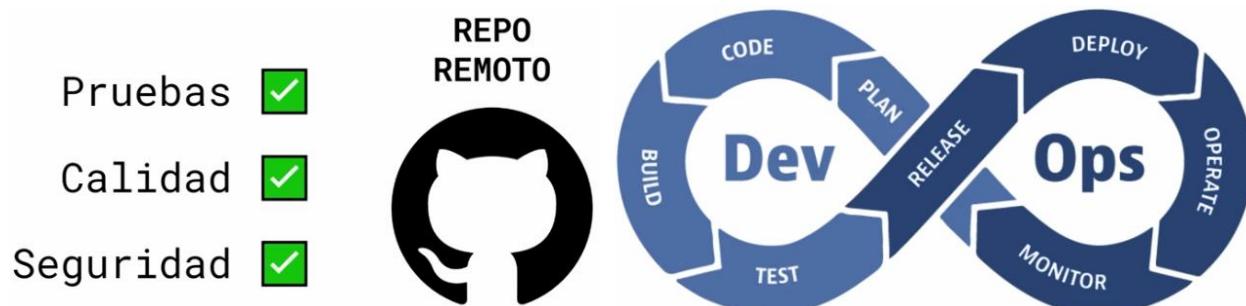


Para que se tenga un orden en la forma que van contribuyendo los miembros de un equipo en un repositorio existe el concepto de **rama**, donde se tiene la **rama principal main** y las **ramas de desarrollo fix, feature y develop** para *arreglar bugs o agregar funcionalidades al código*.

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

1. **PLAN:** Planifica o diseña el desarrollo del código.
2. **CODE:** Desarrolla el código de forma local.
3. **BUILD:** Compila, enlaza y empaqueta el código fuente en un programa ejecutable o en un artefacto listo para ser probado.
4. **TEST:** Prueba o testea el código previamente compilado.
5. **RELEASE:** Publica una versión del software que ya ha sido construida (build) y probada (test).
6. **DEPLOY:** Lo sube a la nube, exponiéndolo a todos los usuarios.
7. **OPERATE:** Opera el código en producción, gestionando temas de latencia dependiendo de la ubicación geográfica de los usuarios.
8. **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**.



Modelos de Arquitectura de Servidores: On Site, IaaS, PaaS y SaaS

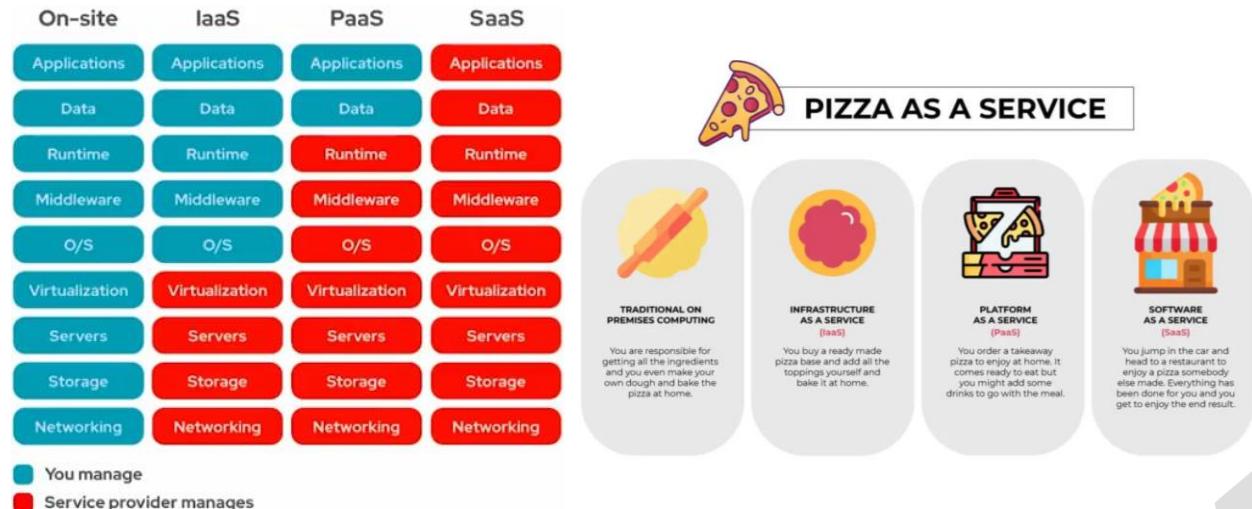
Recordemos que la **nube** se compone de varios servidores, llamados **granja de servidores** o **data centers**, en donde cada uno cuenta con un **número de CPUs**, **bytes de memoria RAM** y **capacidad de almacenamiento en disco de estado sólido SSD o mecánico**. Sin embargo, en estas granjas de servidores se puede adoptar una arquitectura específica.



- **On Site:** Esto se refiere a cuando tenemos un servidor físico propio, donde debemos manejar y montar manualmente todas sus operaciones desde cero, incluyendo instalaciones eléctricas.
- **IaaS (Infrastructure as a Service):** Esta arquitectura de servidores es la que mayor control nos proporciona de las 3 arquitecturas más utilizadas, **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 una intervención manual. Un ejemplo de este es **Firebase de Google**.
- **SaaS (Software as a Service):** En esta arquitectura se cuenta con casi nulo control en el estado de red, en los servidores, en el almacenamiento, sistema operativo, datos, etc. **es un sistema que prácticamente se maneja solo**, sin necesidad de intervenciones manuales. Un ejemplo de este son **Google Drive y Slack**.

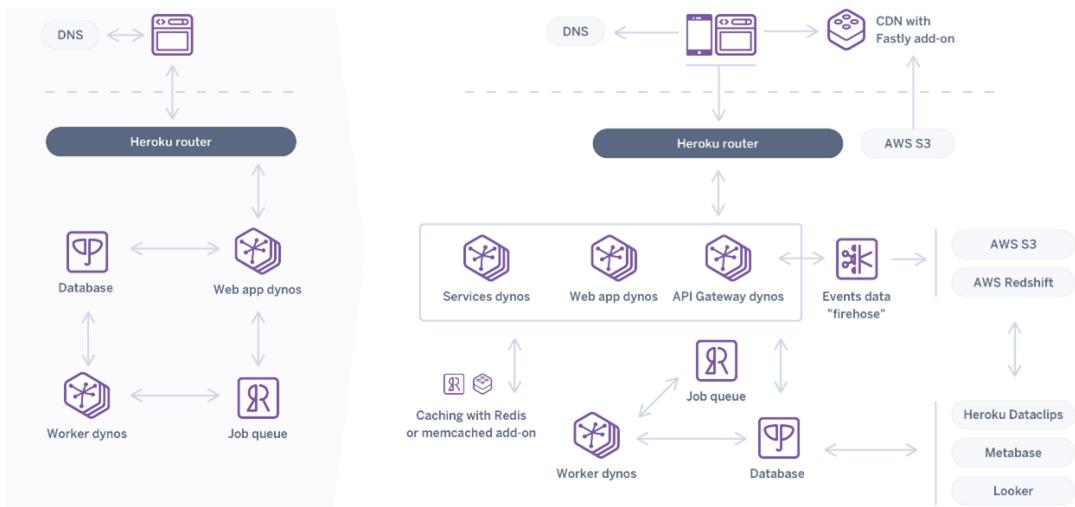
Este concepto se puede explicar con pizzas, donde en el modelo **On Site** se instalaría la cocina, se **comprarían los materiales y todo desde cero**, en el modelo **IaaS** la cocina ya está lista y solo se prepara la pizza desde cero, en el modelo **PaaS** ya se tiene la base de la pizza preparada y solo se le agregan ingredientes y en el modelo **SaaS** no se hace absolutamente nada, solo se ordena y se come la pizza.

La arquitectura de servidores más conveniente a utilizarse durante el desarrollo de un software es la **PaaS**, proporcionando un **control intermedio 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**).



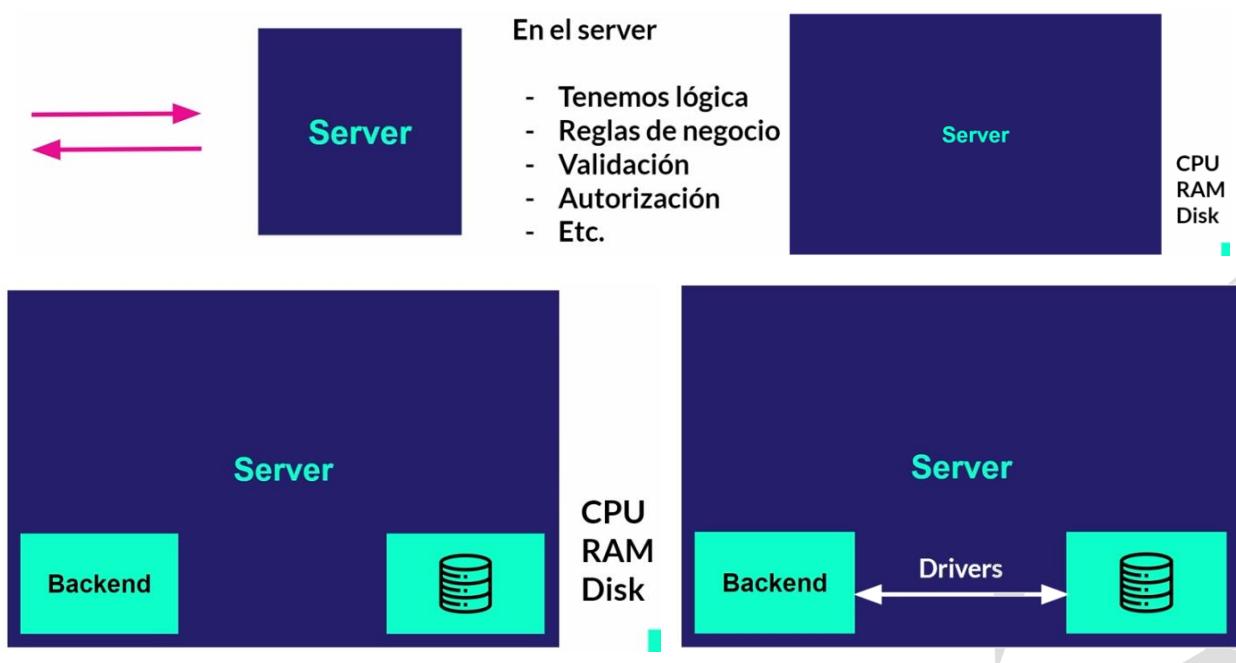
PaaS: Heroku

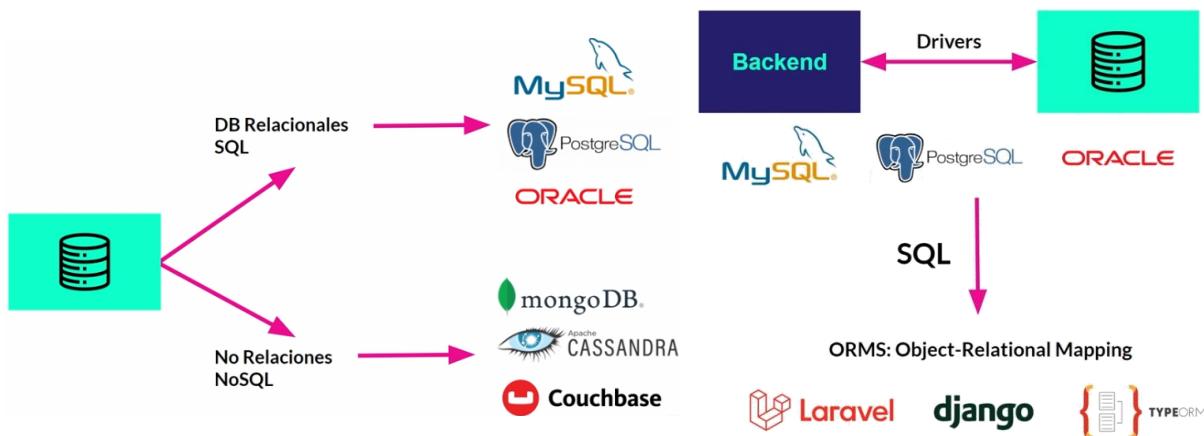
Existen herramientas con arquitecturas que ya están configuradas y manejadas como **PaaS (Plataforma como Servicio)**, una de ellas es **Heroku**, la cual es **una plataforma de nube que automatiza el despliegue (deploy) de aplicaciones y toda la configuración de servidores se proporciona como parte de su servicio**; nosotros solo nos preocupamos por desarrollar el **código (aplicación)** y supervisar la **base de datos (data)**.



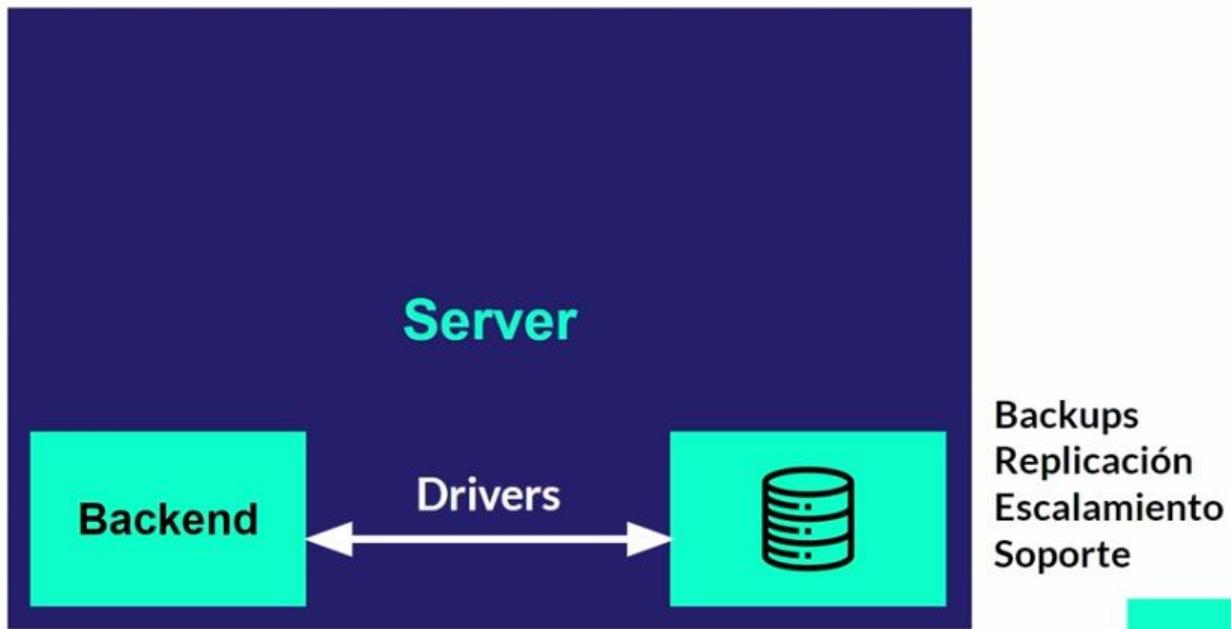
Bases de Datos

Las **bases de datos** son **plataformas intermedias entre un servidor y un cliente**, ya que cuando el cliente interactúa con el servidor, se ingresan datos importantes y relevantes con su operación. Pero como el server cuenta con recursos limitados de CPU, RAM y Disco Duro, los cuales son utilizados para realizar su operación a través de servicios backend como APIs, dichos **datos** son transferidos a elementos llamados **databases**, que de igual forma se encuentran dentro del servidor, almacenándolos de forma permanente e intermedia, esta interacción entre el backend y la DB se realiza a través de algo llamado Drivers, donde cada **base de datos**, ya sea relacional o no relacional y dependiendo también de su tipo, tiene su propio driver, el cual puede estar adaptado para conectarse a distintos lenguajes de programación en el backend. Un ejemplo de esto es como la base de datos de PostgreSQL tiene distintos drivers para conectarse a código Python, JavaScript, PHP, C# .NET, etc. Pudiendo así hasta utilizar métodos orientados a objetos para extraer o insertar datos a través de una herramienta llamada **ORM (Object Relational Mapping)**.

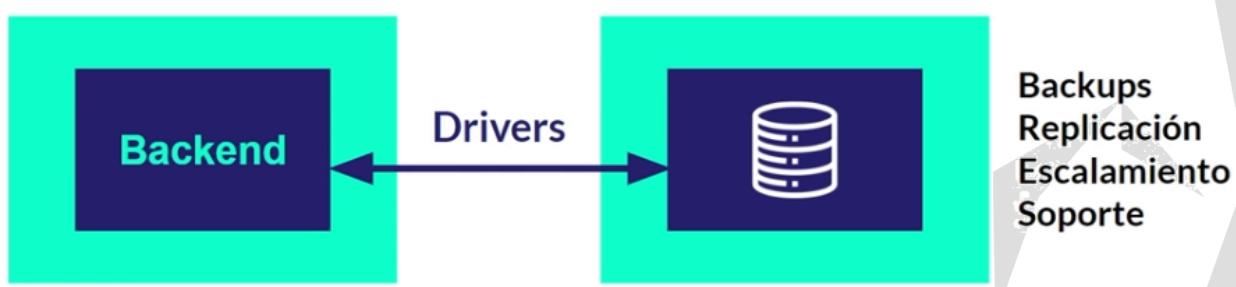




Cuando se haya recabado **un gran número de datos**, estos **pueden ser transferidos hacia un data warehouse**, para luego *analizarlos y realizar operaciones como machine learning, business intelligence, etc.* Esto último si entra dentro del **rol de backend**. Pero **los que realizan tareas de backups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



Existen servicios que pueden realizar todo este tipo de tareas de forma automática, pudiendo delegar así esta parte de la **administración de la base de datos**. Como por ejemplo **Heroku**, **Digital Ocean**, **MongoDB Altas**, **CouchBase**, etc.



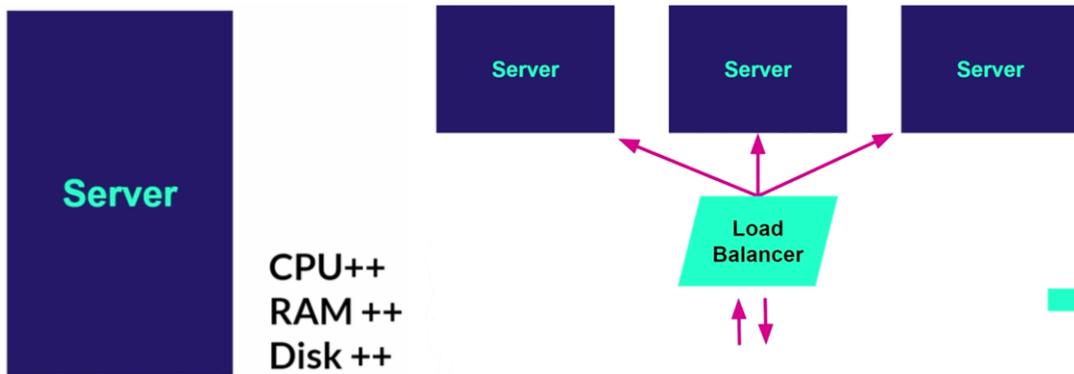


Escalamiento Vertical VS. Horizontal

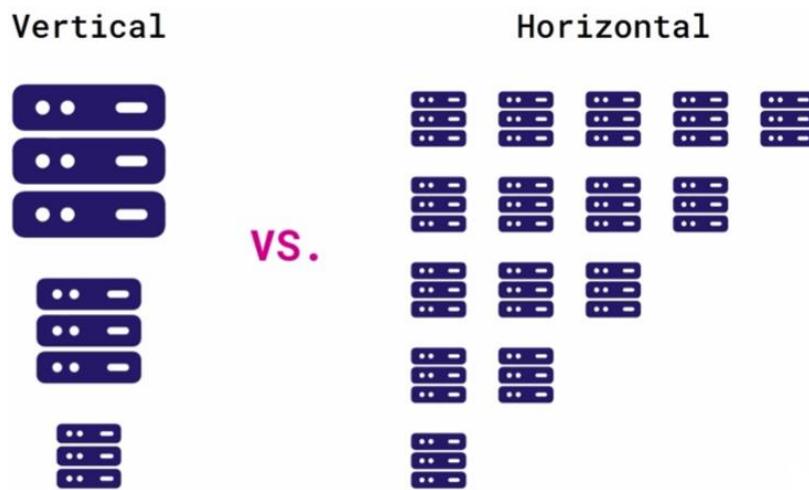
Recordemos que nuestros servidores tienen **limitaciones físicas de CPU, memoria RAM y/o Disco Duro**, pero si nuestras aplicaciones empiezan a tener muchos usuarios, esto hará que comiencen a colapsar porque sus recursos serán insuficientes.

- **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** y estas son las implicaciones de ejecutar este escalamiento:
 - **Causa problemas de disponibilidad**, ya que, si el servidor que ha sido escalado verticalmente falla, todo el sistema de producción se viene abajo.
 - 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 y luego regresar a su estado original.
- **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 **rélicas (copias)** de nuestros servicios en cada uno de ellos 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 sufren 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 especializado* llamado **load balancer** que se encarga de **gestionar las peticiones hechas a los servicios distribuidos en los diferentes servidores que soportan nuestra aplicación (cluster)**, teniendo conocimiento de cuáles son los servidores conectados a la red y para qué sirve cada uno, proporcionando redirecciónamiento.
 - Resulta **complicado de implementar**, pero su **costo es dinámico**, ya que se tienen recursos no tan elevados en cada servidor, los cuales tienen **rélicas (copias) de los servicios de la aplicación**, *pudiendo así mantener un aumento de número de usuarios sin gastar tantos recursos en un solo servidor*, sino teniendo 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.



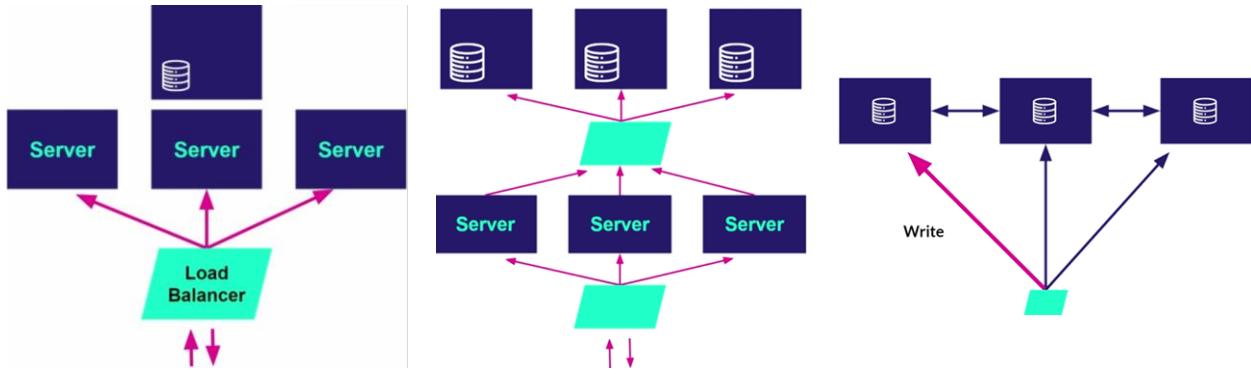
Cabe mencionar que en el **escalamiento horizontal** existe **un tipo de problema** cuando cada uno de sus servidores contenga una **base de datos propia**, ya que, **el servidor 1 no tendrá la misma base de datos que el servidor 3**, por lo que cuando el **load balancer** reciba requests, **dependiendo de a que servidor se conecte para resolver la petición, cada uno contendrá datos distintos**. Esto se puede resolver cuando la **database** esté fuera de nuestra arquitectura y sea dada por un proveedor, como MongoDB, Oracle, etc. Pero si estamos gestionando la **DB** dentro de la arquitectura de nuestros **servidores escalados horizontalmente**, este podría ser un gran problema y se necesitaría ejecutar una **replicación**.



Escalamiento Horizontal: Heroku y Replicación de DataBases

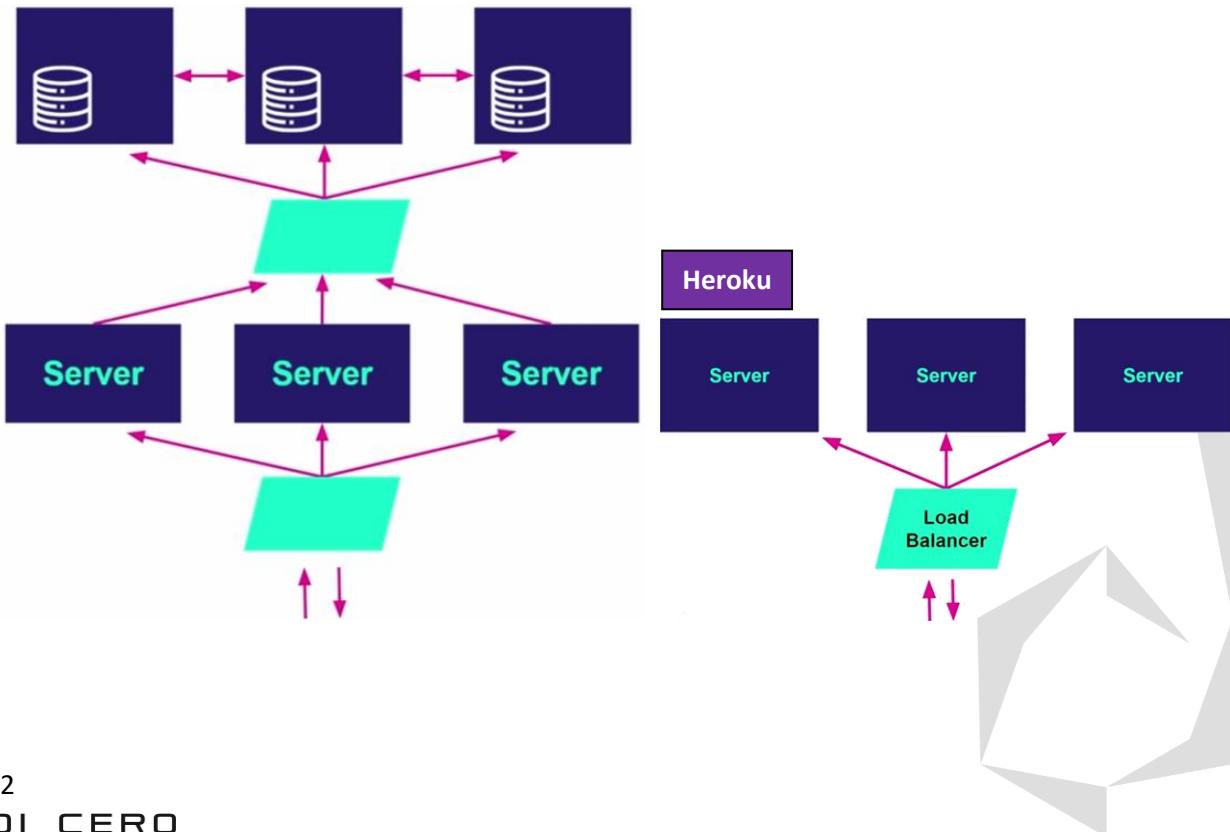
Recordemos que a un **grupo de servidores con escalamiento horizontal** se le llama **cluster**, pero, así como se describió anteriormente, *si cada uno de los servidores del cluster tiene incluida una base de datos diferente, esto causa un problema*, ya que cada uno contendrá datos distintos y el **load 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 **solo uno de los servidores esté dedicado a contener la única database de nuestro sistema**, el problema con esto es que **se crearía un cuello de botella**, ya que todos los servidores harían peticiones a él; 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**, ya que si ese

servidor cae, toda la **DB** de nuestra aplicación fallaría y *aunque aplicáramos en este 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 diferentes bases de datos estén sincronizadas**, asegurándonos que se tengan los mismos datos copiados en todas ellas dentro de nuestra red de servidores.



El objetivo de la **replicación** es que cuando se quiera **leer (GET)** los **datos** de cualquiera de los servidores, todos contengan los mismos, así, cuando se realice una **operación de escritura (POST), edición (PUT/PATCH) o eliminación (DELETE)** hacia cualquiera de ellos, las **databases** de los **distintos servidores** estén **sincronizadas** para que cada una de ellas **contengan lo mismo**.

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 se le delega a un **proveedor de despliegue de servidores (deploy) como Heroku**, creando así una arquitectura horizontal de servidores **PaaS (Plataforma como Servicio)** de forma sencilla.



Cola de Tareas: Queues

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.* Porque ningún usuario se va a quedar tanto tiempo frente a su computadora esperando a que su navegador (cliente frontend) termine de ejecutar una petición. **La diferencia principal 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.*
- **Task Queue:** Una cola de tareas **ejecuta un request eventualmente 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 cuánto 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 **task queues** 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. Un ejemplo es **SQS (Simple Queue Service)** de **AWS (Amazon Web Services)**.

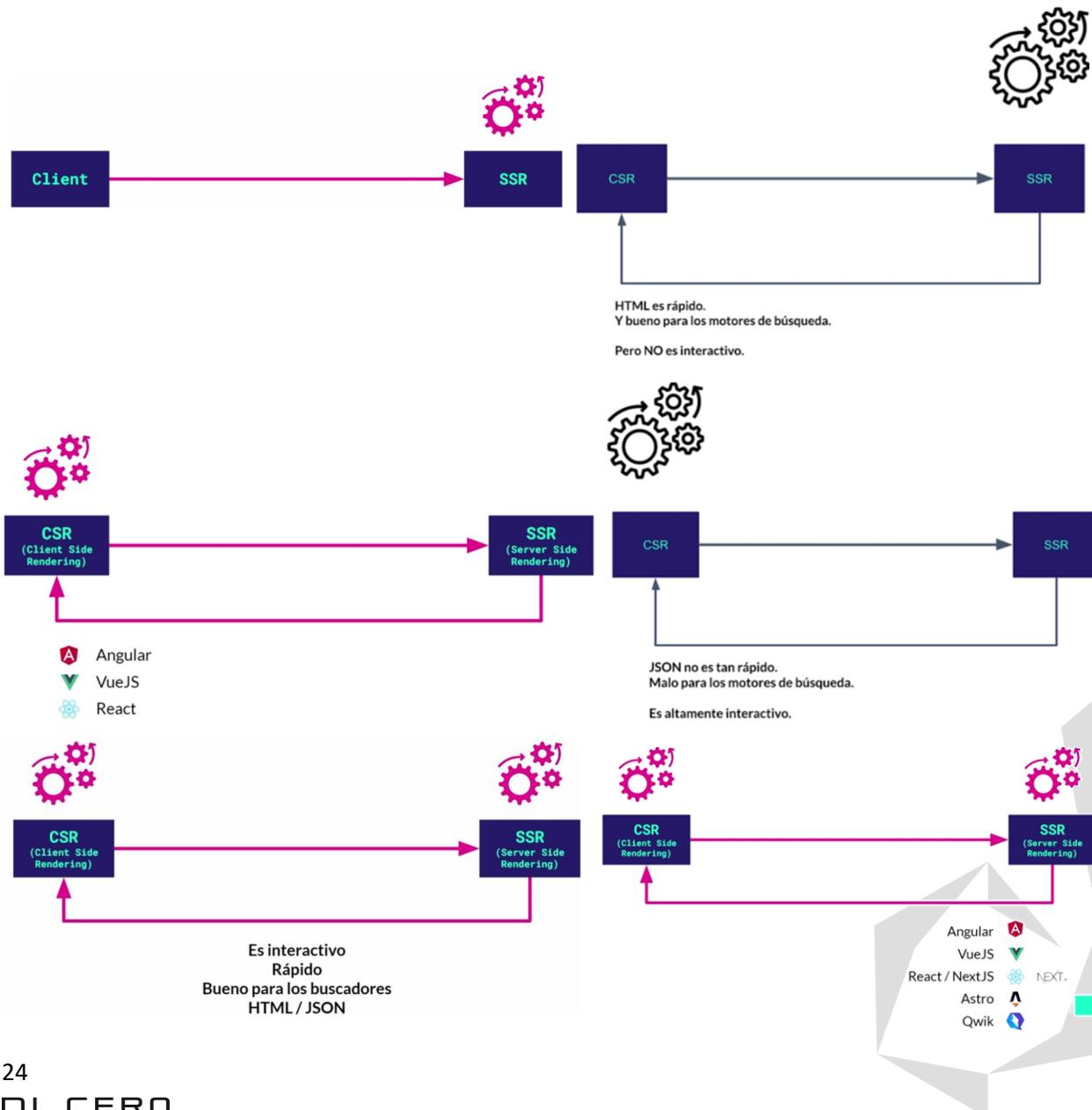


Server Side Rendering (SSR)

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 cliente**.

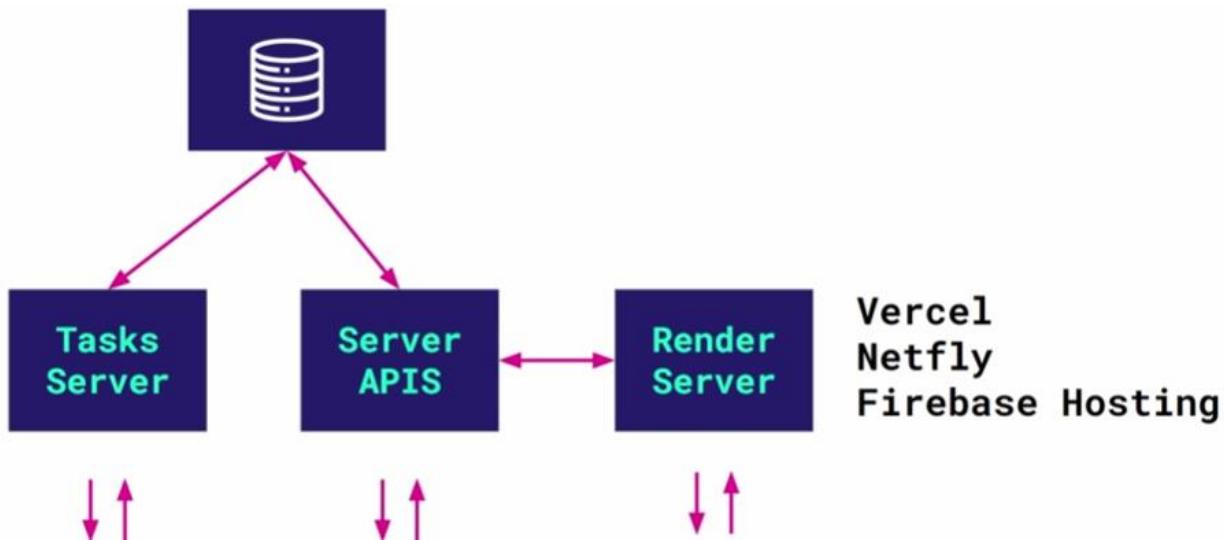
servidor, para que así la representación visual ya llegue interpretada hacia el cliente, logrando así una respuesta más rápida para el usuario, pero con la desventaja de que no es interactiva, esto implica que para que cualquier cambio de datos pueda ser visualizado, se deberá recargar el cliente, ya sea el dispositivo móvil o navegador web. Las ventajas y desventajas de ambos son:

- **CSR (Client Side Rendering)**: Es una técnica donde **el navegador (cliente)** 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 del servidor**, 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 **frameworks de 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 *tardan 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 al implementarles una **arquitectura horizontal** y todos ellos estén sincronizados para devolver **los mismos datos**, proporcionando así disponibilidad de información.
- **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 servicios como Vercel, Netfly o Firebase Hosting para su despliegue (deploy).



Memorias

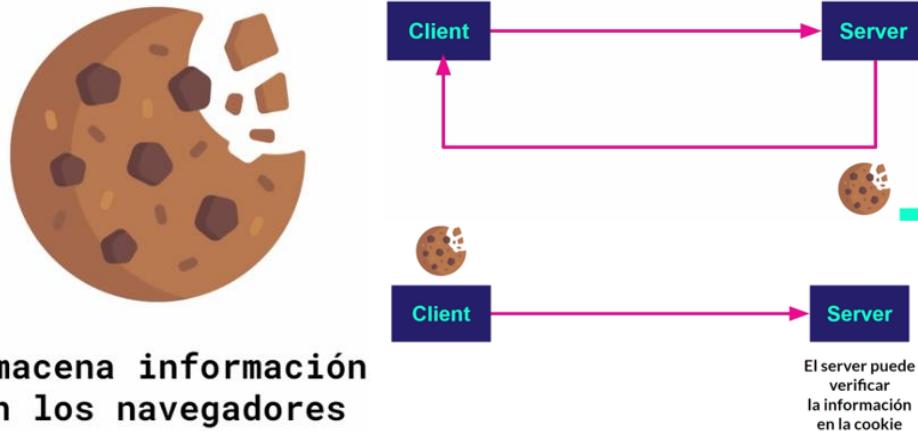
Existen distintos **tipos de memorias** del lado del cliente o del lado del servidor que sirven para facilitar las tareas de nuestras aplicaciones, las cuales serán descritas a continuación.

Memoria Cliente: Cookies/Sesiones

Las **cookies o 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 abierta (login con usuario y contraseña), etc.

Esto se realiza a través del navegador (cliente) hacia el servidor de la siguiente forma:

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 del usuario → Y finalmente esa **cookie** es retornada al servidor con los datos que extrajo del navegador para que siempre que el cliente le haga solicitudes a nuestro servicio, este recuerde los datos del usuario.



El problema con esto es que solo funciona en servicios web, si 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)**, el cual nos permite validar por medio de tokens dicha información del cliente. Aunque cabe mencionar que los JWT si se pueden utilizar también con aplicaciones web.

Memoria Servidor: Caché

Vamos a explicar el concepto de la memoria caché 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.

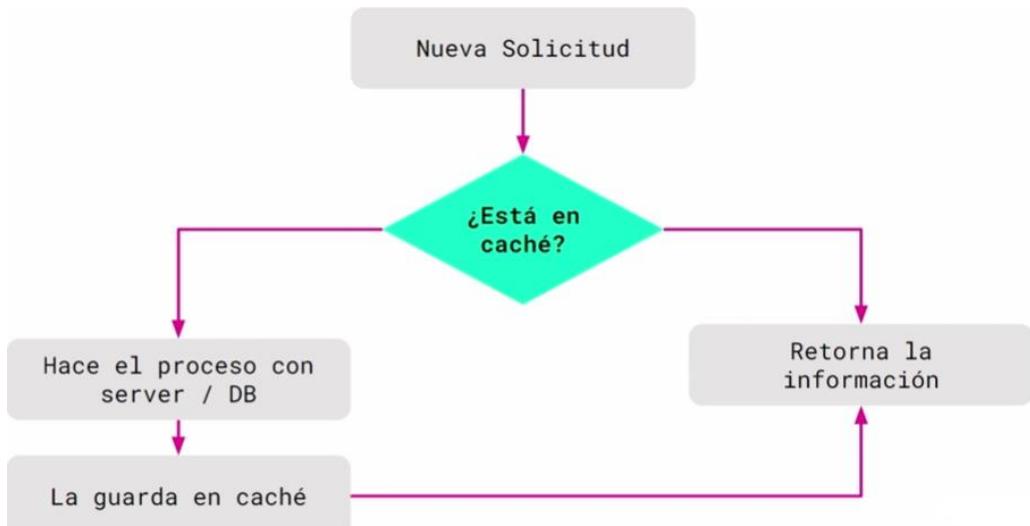
The diagram shows two side-by-side calculations of $955 + 844 = 1799$. The left calculation is labeled 'Primera vez = 30s' and the right one is labeled 'Segunda vez = 2s'. Both calculations are identical, showing the addition of 955 and 844 resulting in 1799.

La forma en la cual funciona la memoria caché, que 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 caché.
 - a. Si no se tiene previamente almacenado el resultado, se ejecuta 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 un e-commerce, 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 (Distributed Denial Of Service), esto no ocasionaría ninguna falla del sistema, ya que la memoria caché se encargaría de gestionar dicha consulta.



Memoria Servidor: Proxy

Una memoria proxy (o sistema proxy caché) es un servidor intermediario entre un cliente (navegador web, aplicación móvil o dispositivo IoT) y el servidor final, la función de este es almacenar copias

temporales de las respuestas más comunes a nuestros requests para acelerar las solicitudes. Su principal objetivo es:

- Ahorrar recursos del servidor al no repetir el mismo procesamiento muchas veces.
- Reducir el tiempo de carga para el usuario.
- Proteger al servidor original de recibir demasiadas solicitudes simultáneas.



Diferencias entre las **Cookies, Memoria Caché y Proxy** son:

Característica	Cookie/Sesión	Memoria Caché	Memoria Proxy
¿Dónde almacena sus datos?	En el navegador.	En el servidor o navegador.	En un servidor intermedio proxy.
¿Qué almacena?	Resultados previos del navegador.	Resultados previos del servidor.	Copias de las respuestas HTTP/HTTPS.
¿A quién ayuda?	Al servidor y al usuario.	Al servidor y al usuario.	A múltiples usuarios que hacen peticiones similares.
¿Es parte de la arquitectura?	No necesariamente.	No necesariamente.	Sí, es parte de la arquitectura.

Memoria Nube (Servidor): Bucket

Bucket es una unidad de almacenamiento en los servidores de la **nube, granja de servidores o data center** que funciona como una carpeta gigante para guardar y organizar nuestros archivos, imágenes, videos, documentos, backups, etc.

Imagina que estás usando una **mochila digital (bucket)** donde puedes guardar cualquier tipo de archivo, y puedes **etiquetar o configurar cada cosa que metes ahí** para que solo alguien la pueda ver, cualquiera la pueda acceder o incluso que se borre sola después de cierto tiempo.

Esto es muy útil **cuando se construye una aplicación web o móvil que necesite guardar archivos sin usar el disco duro de un servidor. Ya sea para subir una foto de perfil, guardar un PDF generado, almacenar los videos de un curso, etc.** Todos esos archivos pueden ir a un **bucket** en vez de saturar al servidor o la **base de datos**.

La forma en la que se utilizan los **buckets** en un servicio cloud es la siguiente:

- Se crea un **bucket** en una plataforma de nube como **AWS S3 (Simple Storage Service)**.
- Se le asigna un nombre único (como mi-app-archivos).
- Desde nuestra aplicación, subiremos archivos a ese **bucket** usando **código o servicios backend**.

Y en ellos se puede configurar:

- Quién puede acceder a esos archivos.
- Si deben estar encriptados.
- Cuánto tiempo se conservan.
- Y cuando alguien quiera descargar un archivo, simplemente se generará un enlace temporal o público desde ese **bucket**. **Un ejemplo de este es Mercury, que sirve para almacenar archivos.**

Los **Buckets** de igual forma cuentan con protocolos de seguridad donde se permite:

- Usar reglas para que solo usuarios autorizados puedan acceder o subir archivos.
- Evitar ataques o accesos no deseados configurando “**permisos de bucket**”.
- Bloquear accesos públicos por defecto para proteger tus datos.

Arquitectura y Despliegue

La arquitectura de software  se refiere al **diseño estructural y lógico de un sistema**, definiendo cómo **se organizan, comunican y operan los distintos componentes de una aplicación**.

Incluye decisiones sobre qué **tecnologías se usarán**, cómo se dividirá la lógica del sistema (por ejemplo, en **capas, servicios o microservicios**), cómo se manejará la **seguridad, escalabilidad, rendimiento y la tolerancia a fallos**. Una buena arquitectura asegura que un sistema sea **eficiente, mantenible, seguro, escalable y confiable**.

Docker: Contenedores e Imágenes

Docker  es una plataforma de código abierto que **permite a los desarrolladores automatizar el despliegue, escalado y administración de aplicaciones** tipo web, escritorio, móvil, **bases de datos**, machine learning, data science, IoT, sistemas embebidos, etc. **dentro de entornos virtuales ejecutables llamados contenedores**.

Docker permite que sus aplicaciones se corran en **contenedores** **aislados**, lo que **facilita su instalación, actualización y funcionamiento en cualquier entorno sin problemas de compatibilidad**, ya que estos **son entornos ligeros y portátiles** que incluyen todo lo necesario para ejecutar una aplicación, desde el sistema operativo, las bibliotecas, las dependencias y el propio código. Algunas de sus características son:

- **Archivo de configuración:** Para configurar las características de un contenedor **Docker**, se debe crear un archivo llamado **docker-compose.yml**, cuya extensión significa **YAML (YAML Ain't Markup Language)**.
- **Imagen:** Es un **paquete que contiene todo lo necesario para ejecutar una aplicación específica**, incluyendo el sistema operativo base, código, librerías y configuraciones. Es solo de lectura, lo que significa que no se puede modificar una vez creada (inmutable). Las imágenes en **Docker** se utilizan para crear **contenedores**, que son instancias ejecutables de la **imagen**.

Cada tipo de **imagen** para aplicaciones basadas en **MongoDB**, **Node.js**, **Django**, etc. se debe configurar de forma diferente, y esto se indica en la documentación de su página **Docker Hub**: <https://hub.docker.com/>

Kubernetes: Gestión de Contenedores Docker

Kubernetes 🌐 es una plataforma de código abierto que permite **gestionar, escalar y orquestar múltiples contenedores Docker** de forma automática y eficiente. Fue desarrollada por Google y ahora es mantenida por la Cloud Native Computing Foundation (CNCF).

Mientras que Docker se encarga de crear y ejecutar contenedores, Kubernetes se encarga de gestionarlos como un conjunto dinámico, realizando su balanceado de carga, recuperación automática, actualizaciones (rolling updates) y administración.

Algunas de sus funciones clave son:

- **Pods**: Agrupaciones de uno o más **contenedores** que comparten recursos y se ejecutan juntos.
- **Nodos**: Servidores (virtuales o físicos) que ejecutan los pods.
- **Cluster**: Conjunto de nodos (conjunto de servidores) que administra Kubernetes.
- **Scheduler y Controlador**: Determinan dónde y cómo se deben ejecutar los pods, manteniendo el sistema en el estado deseado.

Gracias a Kubernetes, es posible construir **arquitecturas escalables**, resilientes y de alta disponibilidad sin tener que gestionar cada **contenedor** manualmente.

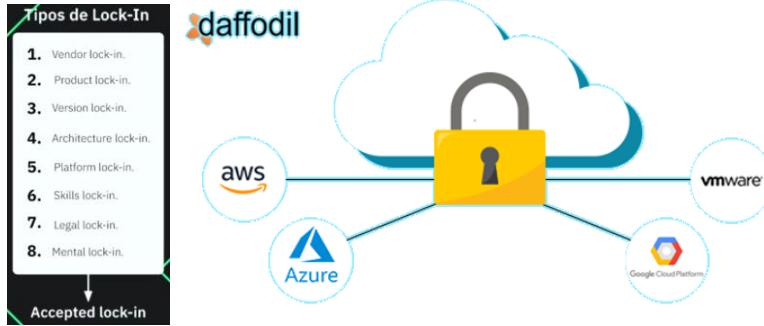
Servidores MultiNube:

Servidor Multinube es una arquitectura que permite distribuir recursos, funciones y servicios a través de diferentes proveedores de nube públicas como **AWS**, **Azure** o **Google Cloud**. Una nube pública se refiere a un proveedor **cloud (cluster)** externo que es compartido por varios usuarios. Sus diferentes tipos son:

- **Elección**: Cada parte del sistema usa la nube más adecuada para su función.
- **Agnóstico**: La arquitectura del sistema se diseña de forma independiente al proveedor, pudiendo moverse entre nubes sin dependencia de ninguna. Utiliza tecnologías portables como **contenedores**, **Kubernetes**, **APIs estándar**, etc.
- **Paralelo**: Una copia de la misma aplicación se ejecuta en varias nubes simultáneamente.
- **Segmentado**: Diferentes partes de la misma aplicación se corren en diferentes nubes.

A diferencia de utilizar **un solo servicio de nube**, a lo cual se denomina también como **arquitectura monolítica**, las **arquitecturas distribuidas multinube** están enfocadas a reducir los **lock-ins**, que son **dificultades o limitaciones para migrar de un proveedor cloud a otro**, debido a servicios exclusivos, **datos no portables** o configuraciones propias de un proveedor de nube específico, permitiendo mayor flexibilidad, resiliencia y control de costos.

- **Sistema Monolítico**: Red conformada por un solo servidor, a esta solamente se le puede aplicar **escalamiento vertical**.
- **Sistema distribuido**: Arquitectura con **escalamiento horizontal**, compuesta por múltiples servers ubicados en diferentes lugares comunicados entre sí por medio de una misma red o hasta diferentes redes, lo que mejora disponibilidad, escalabilidad y tolerancia a fallos.



En la **arquitectura multinube o distribuida**, los servicios backend suelen diseñarse de forma **cloud native**, es decir, **pensados desde el inicio para ejecutarse en entornos distribuidos y escalables**. Un componente clave es el modelo **serverless**, donde no se gestionan servidores directamente, sino que se despliegan **funciones cloud** (como AWS Lambda o Azure Functions) que se activan automáticamente por medio de eventos (como cambios en un **bucket**, **peticiones HTTP** o **entradas a una base de datos**), **escalando automáticamente** el número de usuarios de nuestras aplicaciones y cobrando solo por uso.

- **Cloud Native:** Servicios diseñados desde un inicio para montarse (deploy) en entornos **multinube**.
- **Serverless (sin servidor):** Modelo en la nube donde **no se gestionan los servidores directamente**, es decir, *no manejamos un **load balancer** para decidir cuántos servidores son necesarios ni cuando ejecutar sus ejecuciones, no se configuran máquinas virtuales, contenedores o instancias, actualizaciones del sistema operativo, parches de seguridad, etc. Todo esto lo administra automáticamente el proveedor cloud*, mientras **nosotros como backend solo escribimos funciones que se ejecuten solas cuando se necesiten, proveyendo de escalabilidad automática, sin infraestructura que administrar y con un modelo de pago basado únicamente en el uso real (pay-per-use)**.
- **Función Cloud:** Fragmento de código ejecutado en un entorno serverless. Se activa con eventos como **cambios en bases de datos**, **ejecución de colas de tareas**, **solicitudes HTTP**, etc. Los ejemplos de funciones cloud más utilizadas en el mercado son:
 - **AWS Lambda:** Ejecuta funciones en respuesta a eventos, como cambios en un **bucket AWS S3 (Simple Storage Service)** o al **recibir peticiones HTTP**.
 - **Azure Functions:** Similar a Lambda, pero de Microsoft, permite **ejecutar código bajo demanda con una facturación basada en uso**.

Los entornos serverless pueden incluir sistemas de **streams**, útiles para **procesar flujos de datos en tiempo real**. Además, para proteger el acceso a nuestros servicios, se implementan mecanismos de **autenticación y autorización**, que permiten controlar quién entra a nuestro sistema y qué puede hacer.

- **Streams:** Sistema basado en el procesamiento de flujos de datos en tiempo real.
- **Authentication:** Validación de identidad del usuario (**¿quién eres?**).
- **Authorization:** Determina qué acciones puede realizar el usuario (**¿qué puedes hacer?**).
 - Servicios como **Cognito (AWS)** o **Azure AD B2C** ofrecen autenticación, autorización, y gestión de usuarios.

Finalmente, también en las **arquitecturas multinube** se considera la planificación de recuperación ante fallos, donde los **RTO** (Recovery Time Objective) y **RPO** (Recovery Point Objective) definen tiempos máximos de inactividad y la **cantidad de datos** que se pueden perder sin afectar gravemente al negocio:

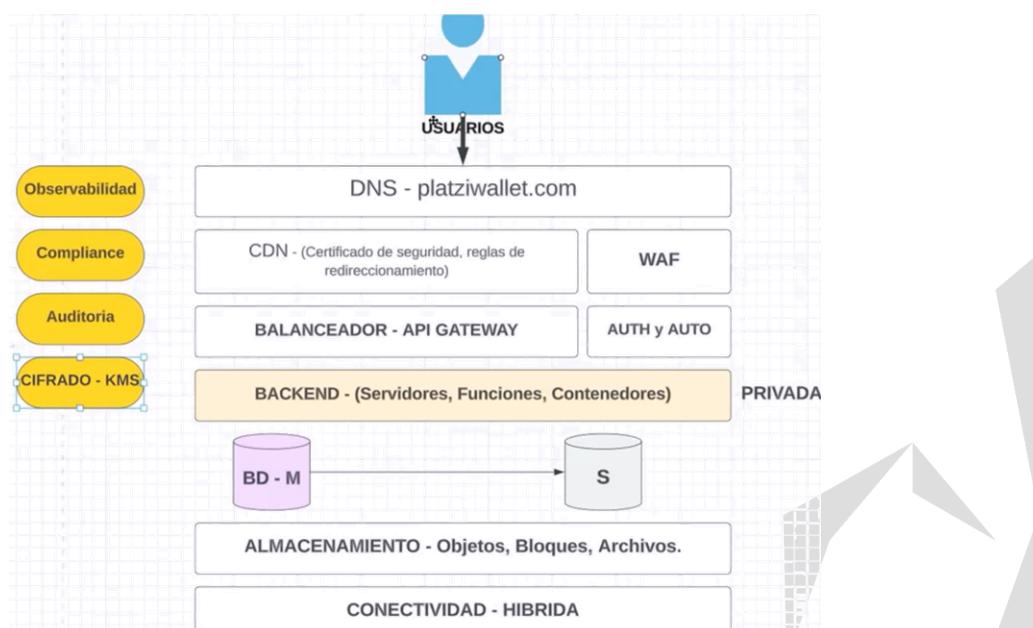
- **RTO (Recovery Time Objective):** Tiempo máximo aceptable para recuperar un servicio tras una caída.
- **RPO (Recovery Point Objective):** Cantidad máxima de **datos** que se pueden perder.

Todo esto convierte al **servidor multinube** en una solución robusta, dinámica y adaptable para los **sistemas distribuidos**. Los distintos tipos de arquitecturas en el mercado son:

1. **Arquitectura Agnóstica:** Es aquella que no depende de un solo proveedor de nube (como AWS, Azure o GCP). Está diseñada para funcionar igual sin importar en qué plataforma esté alojada.
 - a. Las aplicaciones diseñadas de esta forma pueden funcionar tanto en AWS, GCP, Azure, etc. sin cambiar su código gracias a tecnologías de contenedores como Docker, Kubernetes o Terraform.
 - i. Los **componentes de una Arquitectura Agnóstica** son:
 1. **DNS (Domain Name Server) o Servidor de Dominios:** Esta siempre es la primera capa de nuestra arquitectura, la cual nos permite traducir una dirección IP numérica en el nombre de dominio de nuestra aplicación.
 2. **CDN (Content Delivery Network):** Es una **red de servidores distribuidos (cluster)** en distintas regiones del mundo, cuyo objetivo es **entregar contenido estático** (como imágenes, videos o archivos) **desde el servidor más cercano al usuario, reduciendo la latencia y acelerando el tiempo de carga**. Aunque algunas CDNs también permiten configurar certificados SSL y reglas de seguridad o redirección, su función principal es optimizar la entrega de contenido.
 - a. **WAF (Web Application Firewall):** Capa de **protección** que **define reglas de denegación de servicio**, analizando y bloqueando así el tráfico malicioso que llegue hacia nuestra aplicación, protegiéndola contra ataques DDOS, inyecciones SQL, XSS, etc.
 3. **Load Balancer (Balanceador de Carga):** Distribuye tráfico entrante entre **múltiples servidores escalados horizontalmente** o **funciones cloud**, mejorando el rendimiento, disponibilidad y tolerancia a fallos. Este puede ser reemplazado o combinado con un:
 - a. **API Gateway:** Es un intermediario que gestiona todas las solicitudes externas que llegan a los **endpoints de una API**, sin importar si estos están organizados como microservicios o como una aplicación monolítica. Ejecutando funciones de **enrutamiento, autenticación, control de tráfico, transformación de datos, caché** y monitoreo.
 - i. **Autentificación y Autorización:** Esta capa de la arquitectura (si es que no se ha implementado ya a través de una API Gateway) **sirve para la identificación de usuarios y la autorización de sus acciones**, reconociendo así si el usuario está registrado y qué permisos tiene.
 4. **Backend:** Esta capa **se encuentra en una zona privada que está protegida por las capas anteriores en nuestra arquitectura** y se encarga

de gestionar los servicios que son proporcionados por nuestros servidores, ya sean Funciones Serverless o Contenedores (Docker o Kubernetes).

5. **Base de datos:** Esta capa de **database** mínimo siempre debe tener 1 **replicación**, por lo que una de ellas será la **Maestra que recibirá requests** y la otra será su **réplica**, conteniendo así una copia de sus datos que estén siempre sincronizados con la **DB** maestra. Aquí dependiendo de las funciones que se vaya a realizar con la aplicación se elige el tipo de base de datos necesaria, ya sea relacional, no relacional y que motor elegiremos, ya sea PostgreSQL, MongoDB, Oracle, etc.
 6. **Almacenamiento:** Capa de almacenamiento de nube (servidor) o del lado del cliente, donde se elegirá una unidad de memoria, ya sea **cookies**, **caché**, **proxy** o **buckets** para almacenar Objetos, Bloques y Archivos.
 7. **Conectividad Híbrida:** Esta solamente se utiliza cuando nos tenemos que conectar con un servidor en sitio (On site), donde se enlazaría este con los datos de la nube.
- ii. **Disponibilidad:** Esta arquitectura debe estar desplegada (deploy) mínimo en dos servidores distintos, para cubrir la disponibilidad del servicio. Y además debe tener dos servicios transversales en las arquitecturas:
1. **Observabilidad:** Monitoreo de métricas, umbrales, tasas, medición de tiempo y logs, ya sea de errores o de operación normal durante la operación.
 2. **Compliance:** Son reglas de calidad que se deben cumplir como por ejemplo de cifrado de datos.
 3. **Auditoría:** Esto se refiere al monitoreo que se ha efectuado en la aplicación cada desarrollador, para así tener un control de los cambios.
 4. **Cifrado:** Esta es una capa trasversal de seguridad, gestionando un Key Management System que monitoree el cifrado de la información.

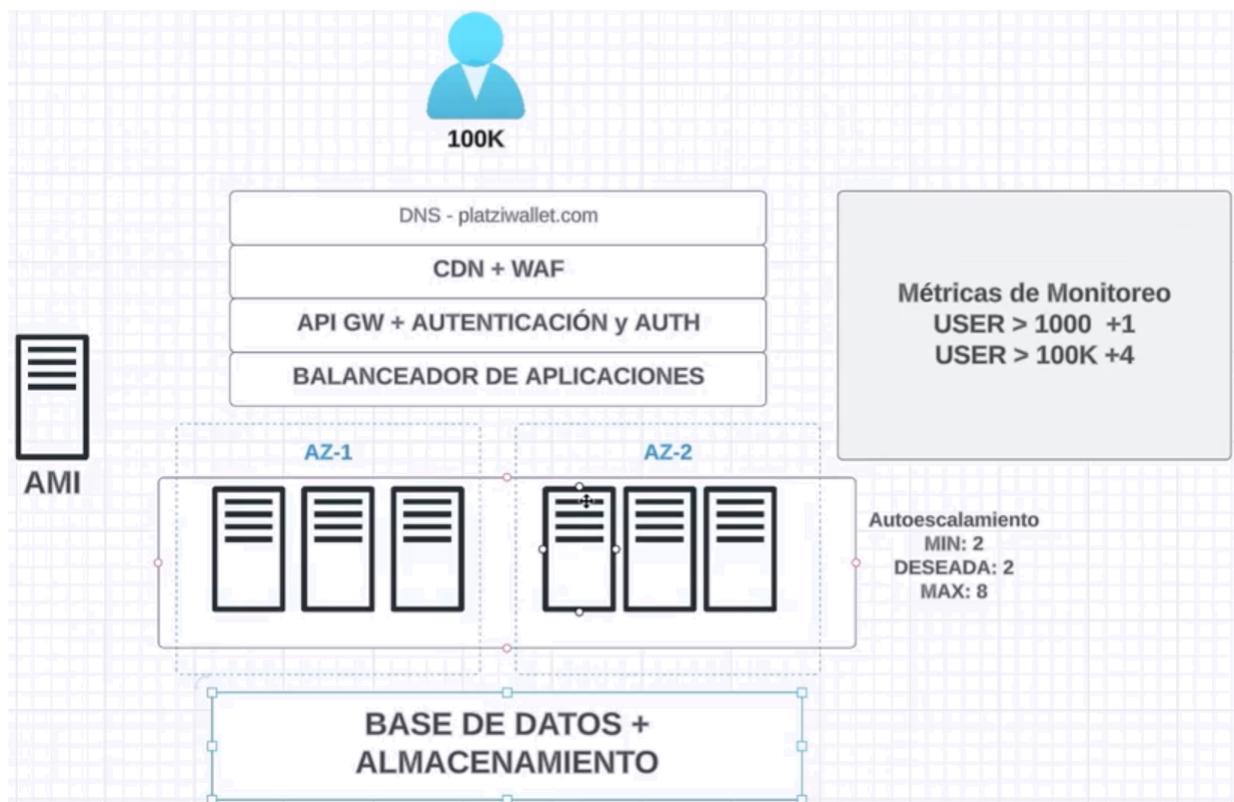


2. **Arquitectura de Servidores para Escalabilidad y Alta Disponibilidad (Usada con servicios AWS, Azure, GCP, etc.):** Es capaz de **crecer sus recursos automáticamente** cuando aumente el número de usuarios y disminuirlos cuando baje la demanda, además está diseñada para que el sistema **sigan funcionando, aunque falle una parte**. Se puede escalar:
- Horizontalmente:** Agregando más instancias.
 - Verticalmente:** Mejorando las capacidades (más RAM, CPU) de una sola instancia.
- i. Los **componentes de una Arquitectura para Escalabilidad y Disponibilidad** son:
 - DNS (Domain Name Server) o Servidor de Dominios:** Primera capa de arquitectura que traduce una dirección IP numérica en el nombre de dominio de nuestra aplicación.
 - CDN (Content Delivery Network): Red de servidores distribuidos (cluster),** cuyo objetivo es **entregar contenido estático** (como imágenes, videos o archivos) **desde el servidor más cercano al usuario, reduciendo la latencia y acelerando el tiempo de carga.**
 - WAF (Web Application Firewall):** Capa de **protección** que **define reglas de denegación de servicio**, analizando y bloqueando así el tráfico malicioso que llegue hacia nuestra aplicación.
 - API Gateway:** Intermediario que gestiona las solicitudes externas que llegan a los **endpoints de una API.**
 - Autenticación y Autorización:** Capa de la arquitectura **para la identificación de usuarios y la autorización de sus acciones.**
 - Load Balancer (Balanceador de Aplicaciones):** Distribuye tráfico entrante entre **múltiples servidores escalados horizontalmente** o **funciones cloud.**
 - Backend corriendo servidores dentro de zonas de disponibilidad (AZ):** El backend en este tipo de arquitectura cuenta con una característica de **auto escalamiento (autoscaling group)** debido a su naturaleza de **escalamiento horizontal en servidores** por lo que se pueden crear copias de nuestros servicios de forma automática en Contenedores (Docker o Kubernetes) para que se monten en servidores adicionales para cubrir la disponibilidad del servicio.
 - Para este tipo de backend con auto escalamiento** se define un número **MÍNIMO** de servidores que tengan nodos o copias de nuestro servicio, cual es la cantidad **DESEADA** promedio y cuál es la cantidad **MÁXIMA**.
 - La forma en la que se decide cuándo debe aumentar su número de servidores el auto escalamiento** es a través de **métricas de monitoreo**, donde se define un umbral en CPU, RAM y Disco Duro, aunque lo más recomendable es basarnos en el número de requests de los usuarios.
 - También, los servidores que vayan siendo creados para cubrir la demanda del incremento de usuarios contiene una AMI,** que es una imagen base con todo ya preinstalado (instancia de contenedor Docker o Kubernetes) para correr nuestros servicios, aunque esta tarda de 3 a 5 minutos en estar disponible, esto

porque el **Load Balancer** primero determina que se haya montado correctamente el servidor, antes de que lo podamos utilizar.

6. **Base de datos:** Esta capa de **database** mínimo siempre debe tener 1 **replicación**, por lo que una de ellas será la **Maestra que recibirá requests** y la otra será su **réplica**, conteniendo así una copia de sus datos que estén siempre sincronizados con la **DB maestra**.

- a. **Almacenamiento:** Capa de almacenamiento de nube, donde se elegirá una unidad de memoria, ya sea **cookies, caché, proxy o buckets** para almacenar Objetos, Bloques y Archivos.



3. **Arquitectura de Aplicaciones con Contenedores:** Divide una aplicación en **contenedores**, los cuales son **entornos ligeros y portátiles** que incluyen todo lo necesario para ejecutar una aplicación, desde el sistema operativo, las bibliotecas, las dependencias y el propio código, para ello utilizan una herramienta complementaria llamada **Kubernetes**.

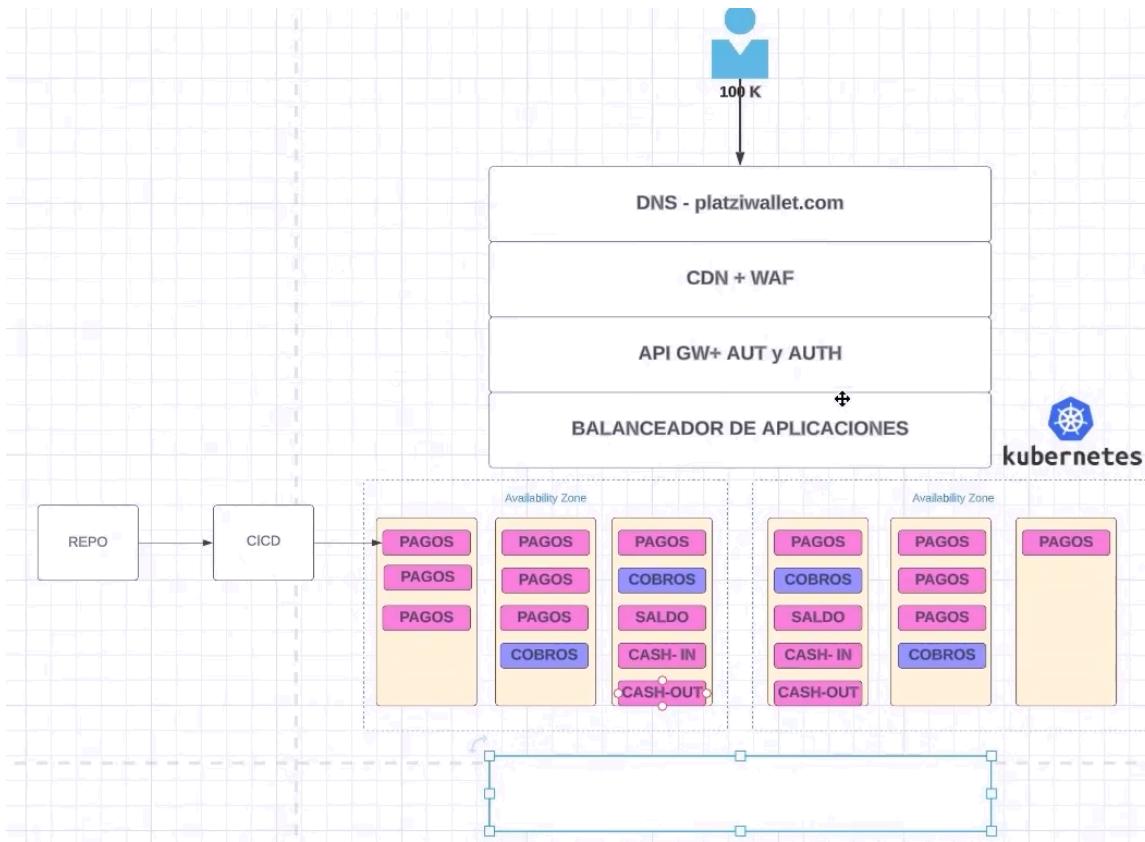
- a. La característica principal de este tipo de arquitecturas es que el backend de nuestra aplicación está empaquetada en contenedores que son gestionados por Kubernetes y cada contenedor representa un microservicio, donde se divide la aplicación en muchos servicios pequeños e independientes. Cada uno con una función específica (pagos, cobros, saldo, cash-in y cash out).

- i. Los **componentes de una Arquitectura de Aplicaciones con Contenedores** son:

1. **DNS (Domain Name Server) o Servidor de Dominios:** Primera capa de arquitectura que traduce una dirección IP numérica en el nombre de dominio de nuestra aplicación.

2. **CDN (Content Delivery Network):** Red de servidores distribuidos (**cluster**), cuyo objetivo es entregar contenido estático (como imágenes, videos o archivos) desde el servidor más cercano al usuario, reduciendo la latencia y acelerando el tiempo de carga.
 - a. **WAF (Web Application Firewall):** Capa de protección que define reglas de denegación de servicio, analizando y bloqueando así el tráfico malicioso que llegue hacia nuestra aplicación.
3. **API Gateway:** Intermediario que gestiona las solicitudes externas que llegan a los endpoints de una API.
 - a. **Autenticación y Autorización:** Capa de la arquitectura para la identificación de usuarios y la autorización de sus acciones.
4. **Load Balancer (Balanceador de Aplicaciones):** Distribuye tráfico entrante entre múltiples servidores escalados horizontalmente o funciones cloud.
5. **Backend corriendo en contenedores, gestionado por Kubernetes:** El backend en este tipo de arquitectura se encuentra empaquetado dentro de contenedores Docker, los cuales representan cada microservicio de nuestra aplicación y los servidores son gestionados por Kubernetes, pudiendo duplicar los contenedores de cada microservicio dentro de los servidores como sea necesario para cubrir las necesidades de los usuarios, logrando así tener disponibilidad, escalando tanto como en el número de servidores, como en los contenedores que cubren cada microservicio dentro de ellos.
 - a. Para asignar un microservicio (contenedor) a un servidor específico que se necesite para cubrir la disponibilidad hacia el usuario, se programan reglas de escalamiento que tengan que ver con las capacidades físicas de los servidores, ya sea número de CPUs, memoria RAM, Disco Duro o número de peticiones que reciben de los clientes.
 - b. De igual forma, se puede encontrar el caso en donde no se tienen servidores que sean manejados por Kubernetes en el backend, sino que se esté gestionando una arquitectura serverless, donde el cloud provider se encarga de la gestión de los servidores, cuando esto pase, nosotros solo nos encargaremos de las funciones cloud y la ejecución de sus microservicios.
6. **CI/CD:** En este tipo de arquitectura se considera esto y como se gestiona el desarrollo de código DevOps en el repositorio, incluyendo pruebas unitarias, etc.
7. **Base de datos:** Esta capa de database mínimo siempre debe tener 1 replicación, por lo que una de ellas será la **Maestra que recibirá requests** y la otra será su **réplica**, conteniendo así una copia de sus datos que estén siempre sincronizados con la **DB maestra**.

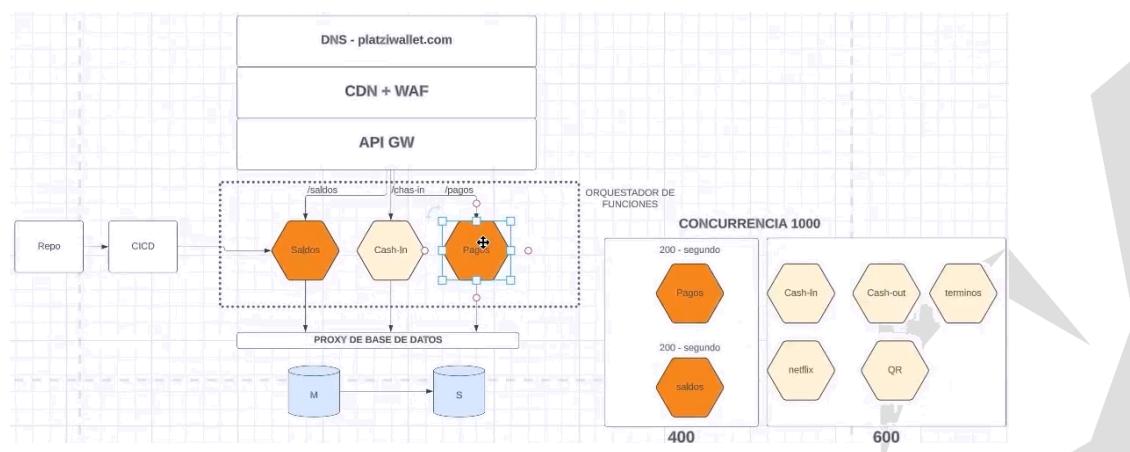
- a. **Almacenamiento:** Capa de almacenamiento de nube, donde se elegirá una unidad de memoria, ya sea **cookies**, **caché**, **proxy** o **buckets** para almacenar Objetos, Bloques y Archivos.



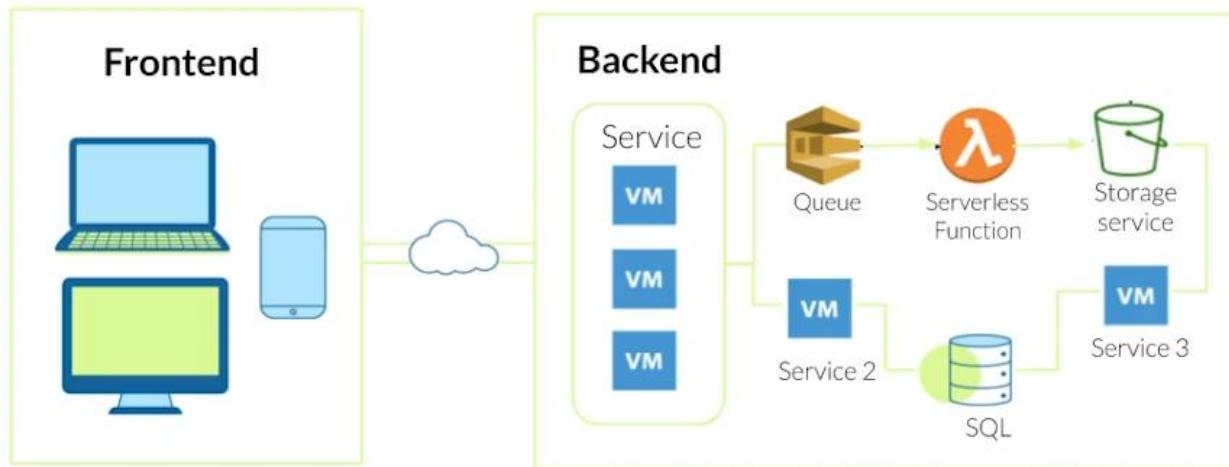
4. **Arquitectura Serverless - Funciones Cloud:** Esta arquitectura en vez de mantener la gestión de servidores, se enfoca solo en **escribir funciones que se ejecuten bajo demanda** cuando alguien las llame. Los servicios clave con los que se ejecuta esto son:

- a. AWS Lambda, Azure Functions, Google Cloud Functions, etc. Cada función representa un microservicio, donde se divide la aplicación en muchos servicios independientes. Cada uno con una acción específica (pagos, cobros, saldo, cash-in y cash out).
 - i. Los **componentes de una Arquitectura Serverless** son:
 1. **DNS (Domain Name Server)** o **Servidor de Dominios**: Primera capa de arquitectura que traduce una dirección IP numérica en el nombre de dominio de nuestra aplicación.
 2. **CDN (Content Delivery Network)**: **Red de servidores distribuidos (cluster)**, cuyo objetivo es **entregar contenido estático** (como imágenes, videos o archivos) **desde el servidor más cercano al usuario, reduciendo la latencia y acelerando el tiempo de carga**. En la arquitectura serverless, también se pueden crear cloud functions que ejecuten acciones específicas en esta capa de CDN.
 - a. **WAF (Web Application Firewall)**: Capa de **protección** que **define reglas de denegación de servicio**, analizando y bloqueando así el tráfico malicioso que llegue hacia nuestra aplicación.

3. **API Gateway:** Intermediario que gestiona las solicitudes externas que llegan a los **endpoints de una API**.
 - a. **Autentificación y Autorización:** Capa de la arquitectura para la identificación de usuarios y la autorización de sus acciones.
4. **Backend ejecutado por medio de Funciones Cloud:** El backend en este tipo de arquitectura no contiene un **Load Balancer** previo, sino más bien, que cada función serverless es ejecutada por un endpoint en específico a través de algo parecido al balanceador de carga, pero denominado como **Orquestador de Funciones**, las cuales representan cada microservicio de nuestra aplicación y los servidores son gestionados por Kubernetes, pudiendo duplicar los contenedores de cada microservicio dentro de los servidores como sea necesario para cubrir las necesidades de los usuarios, logrando así tener disponibilidad, escalando tanto como en el número de servidores, como en los contenedores que cubren cada microservicio dentro de ellos.
 - a. **Concurrencia:** A cada una de las funciones serverless de la arquitectura se le puede asignar una capacidad de procesamiento, dependiendo de su uso esperado durante la operación del sistema, indicando un número de operaciones por segunda fija y dejando los demás recursos para las funciones restantes.
5. **CI/CD:** En este tipo de arquitectura se considera esto y como se gestiona el desarrollo de código DevOps en el repositorio, incluyendo pruebas unitarias, etc.
6. **Base de datos:** Esta capa de **database** mínimo siempre debe tener 1 **replicación**, por lo que una de ellas será la **Maestra que recibirá requests** y la otra será su **réplica**, conteniendo así una copia de sus datos que estén siempre sincronizados con la **DB maestra**.
 - a. **Proxy:** Si queremos utilizar una base de datos relacional, necesitamos implementar una capa de memoria Proxy previa.
 - b. **Bases de datos no relacionales:** La más utilizada con este tipo de arquitectura es la base de datos llave-valor, pero se puede utilizar una basada en documentos, etc.



Nota: Vale la pena mencionar que todas las arquitecturas pasadas pueden ser mezcladas para obtener el mejor resultado en nuestras aplicaciones.



API:

Cómo está implementado el sistema



Proyecto Backend

Dados ciertos requerimientos de negocio, se diseñará e implementará un sistema backend sobre el que ejecutaremos una serie de pruebas a través del desarrollo TDD (Test Driven Development), confirmando su correcto funcionamiento por medio de APIs.

Definición de los Requerimientos del Negocio

La meta es construir y desarrollar la arquitectura de un sistema backend desde cero hasta la implementación (deploy) del mismo, deberá realizarse tomando en cuenta la planificación de la arquitectura que se detalle en alto nivel. Para ello, los requerimientos que nos ha dado un cliente ficticio son los siguientes:

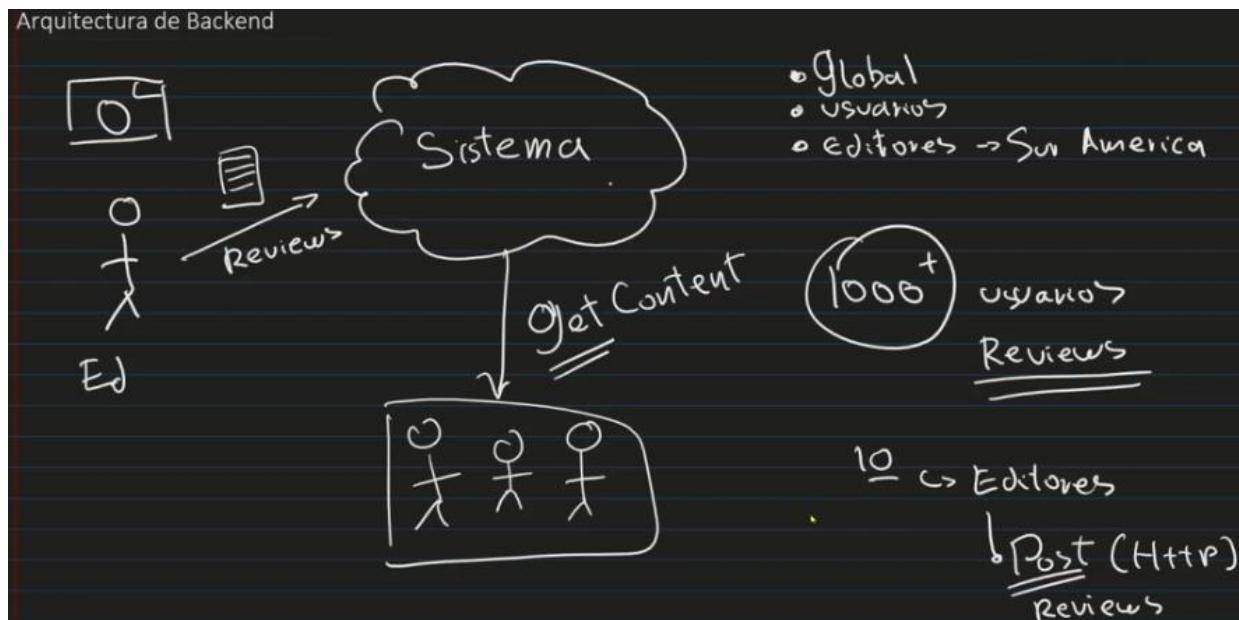
- La empresa “RandomCameraReviews” necesita un sistema que permita que fotógrafos profesionales suban “reviews” de cámaras fotográficas, para que cualquier persona en todo el mundo pueda buscar los reviews y compararlas a través de su portal. La empresa cuenta con un equipo de developers especializado en frontend que realizará una interfaz de usuario para que los editores suban sus “reviews” y los usuarios puedan verlas, y han solicitado que creemos un sistema backend, incluyendo su API, que permita realizar lo siguiente:
 - Subir reviews de cámaras fotográficas.
 - Obtener el contenido de los reviews para mostrarlo en vistas del portal en sus versiones web y mobile.
 - Manejo de usuarios para editores (no incluye visitantes que leen los reviews).

También se menciona que la empresa “RandomCameraReviews” planea distribuir mayormente en América del Sur, donde está su mercado más grande, pero también tienen ventas en Norte América, Europa y muy pocas en Asia. De igual forma, los editores se encuentran mayormente en América del Sur.

Para ello, realizamos el siguiente razonamiento a través de los datos dados por el cliente:

- El sitio es perteneciente a una empresa que vende cámaras fotográficas, y nuestro objetivo es desarrollar un **sistema backend** donde el **editor (Ed)** va a enviar **reviews** a nuestro **sistema en la nube**.
- Y también se tiene muchos usuarios que van a ser los consumidores del sistema, estos no necesitan estar registrados en nuestro sistema (no tienen login con usuario o contraseña), simplemente van a poder acceder al contenido de los reviews de las cámaras por medio de un **método HTTP GET** a un **endpoint llamado /content**.
- Las características del **cluster** para ser **distribuido** geográficamente de forma correcta son:
 - **Usuarios lectores:** Estos tienen presencia global en América del Sur, Norte América, Europa y muy poco en Asia. Esto puede crecer, por lo que necesita tener una gran **capacidad de disponibilidad en lectura de datos**.
 - **Los editores se encuentran:** Solamente se encuentran en Sudamérica.
 - Los editores son los únicos que podrán **introducir datos al sistema**, por lo que deberán poder utilizar **métodos HTTP POST con un endpoint llamado /reviews**.

- Pero cabe mencionar que **los editores serán mucho menores en número a los usuarios que visualizan las reviews**, por lo que no se necesitan muchos recursos de introducción de **datos** al sistema.



Documento de Diseño de Software: High Level System Design

Ya que se haya visto cuales son los requerimientos del cliente para nuestro sistema backend, vamos a tener que plasmar eso en algo llamado documento de diseño, el cual tendrá todos los detalles necesarios de forma agnóstica (estando diseñada para funcionar igual sin importar en qué proveedor de nube esté alojada) para poder desarrollar su arquitectura. Estos documentos normalmente se crean dentro del archivo README.md de un repositorio GitHub y se utiliza código markdown para su desarrollo, incluyendo la siguiente información:

- **Definiciones y acrónimos:** Aquí se describe conceptos clave que se vayan a mencionar en el documento, para que cualquier persona de producto (no desarrollador de software) pueda entender a qué se refiere cada cosa.
- **Problema Por Resolver (Abstract/Overview):** Es una descripción general del proyecto.
- **Objetivos:** Aquí se enlistan los objetivos que va a alcanzar el sistema backend.
 - **Stakeholders:** Es cualquier **persona, grupo u organización que tiene interés o está afectado por un proyecto**, producto o decisión.
- **Suposiciones:** Es una descripción detallada de las cosas que el desarrollador asume hacia el cliente para el desarrollo del proyecto.
- **Limitaciones y Desconocimientos:** Cuestiones limitantes en infraestructura o conocimiento hacia el escalamiento del proyecto o preocupaciones del desarrollador que son comunicadas en esta sección hacia el cliente.
- **Alcances del Proyecto (Scope):** Cosas que el desarrollador se compromete a cumplir durante el desarrollo del sistema.

- **Out Of Scope:** Cuestiones relacionadas al proyecto que salen del enfoque principal de este mismo, las cuales están fuera del alcance y no serán tratadas por el momento, sino en posibles futuras iteraciones.
- **Casos de Uso:** Ejemplos donde se podría utilizar el sistema.
- **Proposal:** Propuesta de arquitectura, componentes internos y externos, estructura de datos y diagramas de secuencias o funcionamiento.
 - **Diagramas y Modelos de Datos:** Son representaciones visuales y de código donde se denota el flujo de información y la conexión de los elementos distribuidos en la red.
- **Costos:** Es una descripción detallada de los costos de operación que generará el sistema, pudiendo añadir aquí diagramas y cotizaciones.

Un ejemplo de este tipo de documentos se encuentra en el siguiente repositorio de GitHub: [High level System Design · jorgevgut/airquality-mx Wiki](#)

También se tiene este formato de ejemplo, que se puede tomar de referencia para desarrollar el documento: [curso_backend_platzi/design_doc_template.md at main · jorgevgut/curso_backend_platzi](#)

Nota: Este tipo de documentos normalmente se redactan en inglés, pero de igual forma el uso del lenguaje se puede adaptar a las necesidades del cliente.

Screenshot of a GitHub repository showing a High-level System Design document for AirQualityMx.

The repository page shows the following details:

- Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights
- High level System Design
- Jorge edited this page on Apr 22, 2020 - 27 revisions

The main content is titled "AirQualityMx high level system design". It includes a Content index and a sidebar navigation.

Content index:

- Definitions and Acronyms
- Change Log
- Abstract
- Goal/Objectives
 - Stakeholders
- Assumptions
- Limitations & Unknowns
- Out of Scope
- Proposal
 - Architecture
 - Phase 1: description
 - Phase 2: next steps
 - Architecture Diagram description
 - Architecture Diagram
 - External components
 - Owned components
 - End-to-End flow-Sequence Diagram
 - Data Structures
 - Costs

Definitions and Acronyms:

Architecture Diagram:

```

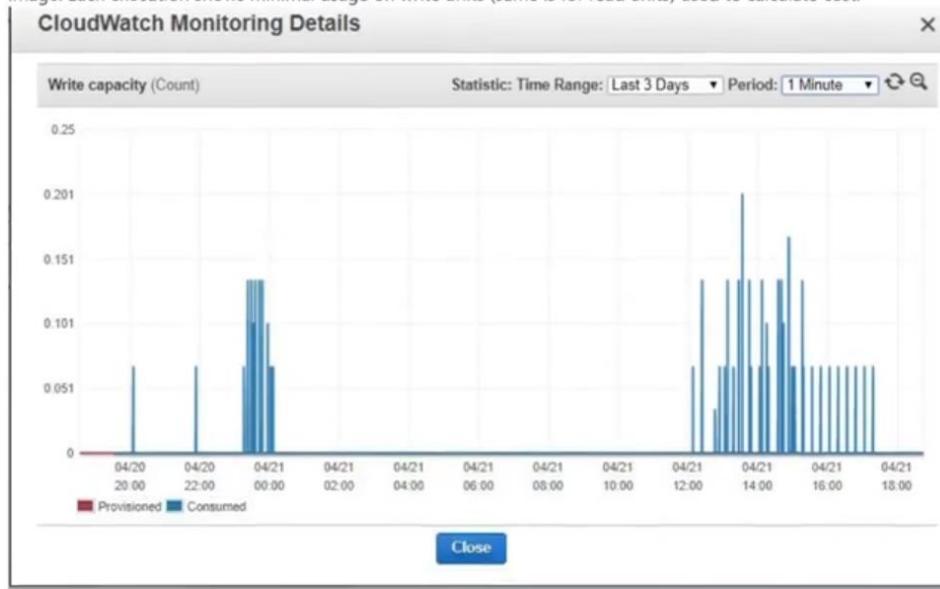
graph TD
    CloudWatchEvents[CloudWatch Events every 5 min/configurable] --> WAGI[WAGI Get City Feed Lambda]
    WAGI --> SNS[SNS topic]
    WAGI --> DDStreams[Feed Processor Lambda]
    SNS --> Twitter[Twitter Publisher Lambda]
    DDStreams --> DDStreamsLambda[Feed Processor Lambda]
    DDStreamsLambda --> AirQualityTable[AirQuality-Table Dynamo DB]
    AirQualityTable --> DDStreamsLambda
    DDStreamsLambda --> DDStreamsLambdaValidation[Feed Processor Validates and Formats data to be suitable for the delivery channel (Ex: char limit, limited API rate calls)]
    DDStreamsLambdaValidation --> Twitter
  
```

Sequence Diagram:

```

sequenceDiagram
    participant CloudWatchEvents
    participant WAGI
    participant SNS
    participant DDStreamsLambda
    participant AirQualityTable
    participant FeedProcessorLambda
    participant Twitter
    CloudWatchEvents->>WAGI: CloudWatch Events every 5 min/configurable
    activate WAGI
    WAGI->>SNS: WAGI Get City Feed Lambda
    SNS->>Twitter: Twitter Notification Publisher
    SNS->>DDStreamsLambda: DDStreams trigger lambda when data for a City changes
    activate DDStreamsLambda
    DDStreamsLambda->>AirQualityTable: Read from AirQuality-Table Dynamo DB
    AirQualityTable-->>DDStreamsLambda: Data
    DDStreamsLambda->>DDStreamsLambdaValidation: Feed Processor Validates and Formats data to be suitable for the delivery channel (Ex: char limit, limited API rate calls)
    DDStreamsLambdaValidation-->>Twitter: Twitter Publisher Lambda
    deactivate DDStreamsLambda
    deactivate WAGI
    deactivate SNS
    deactivate AirQualityTable
    deactivate FeedProcessorLambda
    deactivate Twitter
  
```

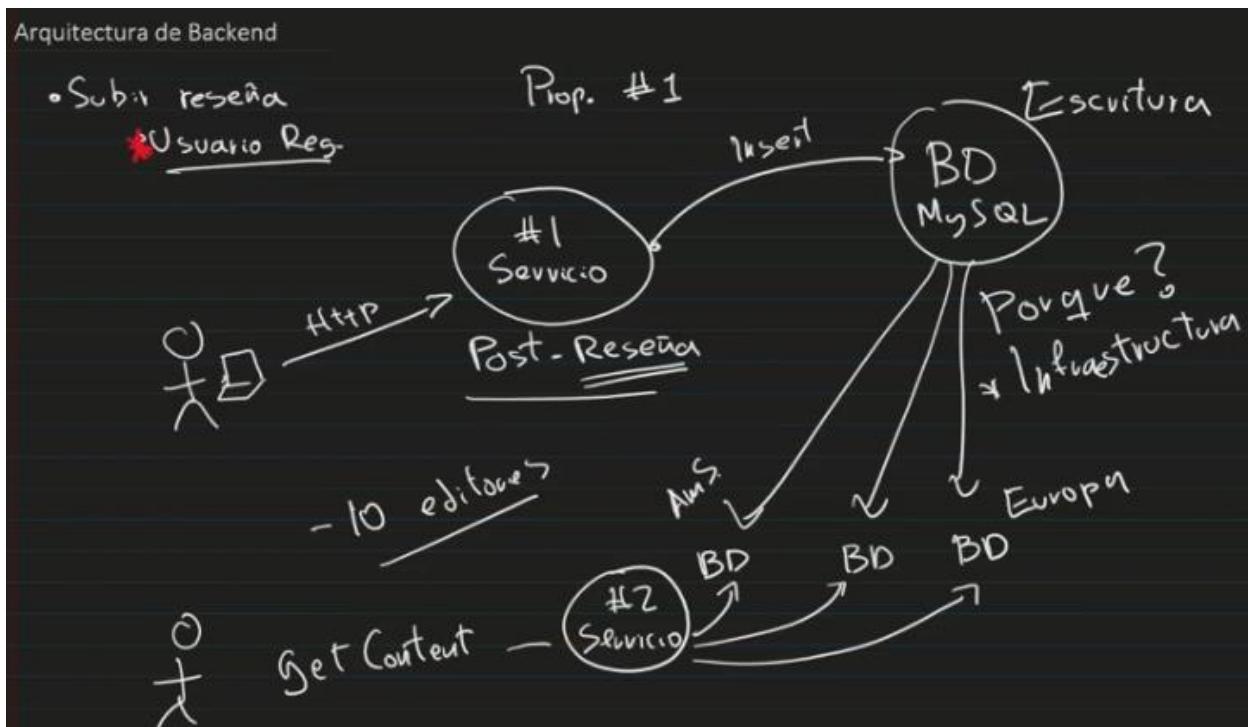
DynamoDB costs: Upon testing, costs are significantly low to perform a reliable calculation, as shown in the following image. Each execution shows minimal usage on write units (same is for read units) used to calculate cost.



Ya sabiendo los componentes que debemos incluir, procederemos a mostrar el documento de diseño de este ejemplo de proyecto backend, cabe mencionar que este documento se realiza a través de varias iteraciones entre nuestro equipo de desarrollo y el cliente, poniendo muy claros los alcances y objetivos de este, para ello consideremos la siguiente propuesta:

- Propuesta 1:
 - **Servicio 1 - Subir Reseñas:** Para esto utilizaremos el método HTTP POST con el endpoint /reviews y un método INSERT para introducir ese dato del servicio backend hacia la base de datos relacional MySQL.
 - Una toma de decisiones aquí sería la de elegir una base de datos relacional MySQL como lo expone la propuesta 1 o una no relacional, sabiendo los beneficios, el feedback que se discute en equipo debe resolver la pregunta de ¿porqué se están tomando ciertas decisiones?
 - **Infraestructura:** La respuesta del porque se toman decisiones debe estar basada en la infraestructura que adoptará el proyecto cuando escala, las medidas verticales u horizontales que se tomarán cuando esto pase y que proveedor cloud dará soporte al servicio, tomando en cuenta costos y requests que vaya a solicitar cada endpoint de nuestro servicio backend.
 - Por ejemplo, como en este caso se menciona que el número de reviewers será poco en comparación con los viewers, se podría tomar la decisión de que esta base de datos sea relacional y de bajos recursos, pudiendo ser escalada solo verticalmente por el momento, debido a que no se espera una pronta subida de usuarios en este lado, por lo que se puede decidir montar esta parte del sistema como está, de forma relacional y con una sola base de datos monolítica que sea simplemente de escritura, no de lectura, para que luego se haga una réplica en otra que sí sea de lectura.

- **Servicio 2 - Lectura de Reseñas:** Como en el caso anterior se decidió que una de las bases de datos fuera solo de escritura y que esta tenga replicación en otras de lectura, esta parte de la base de datos como si tendrá mucha gente que verá las reseñas a través del endpoint /content en zonas de Sudamérica, Norte América, Europa y muy pocas en Asia, este sistema de base de datos debe estar escalado horizontalmente, proveyendo de disponibilidad en América, Europa y Asia, por lo que se necesitan 3 servidores.
 - El porque se responde simplemente por el número de usuarios que este segundo servicio va a proveer, ya que abarca muchas zonas geográficas y una gran variedad de usuarios.
 - **Infraestructura:** Debido a la disponibilidad de datos, se deberá tener mínimo 3, máximo 4 servidores de bases de datos con réplica del servicio 1, que alimente al servicio 2, soportando así la funcionalidad del servicio en Sudamérica, América del Norte, Europa y Asia.



Por lo tanto, el documento de diseño final en formato markdown es el siguiente:

Diseño de Alto Nivel - Backend para RandomCameraReviews

🔎 Definiciones y Acrónimos

- **API**: Interfaz de Programación de Aplicaciones.
- **TDD**: Desarrollo Guiado por Pruebas (*Test Driven Development*).
- **HTTP**: Protocolo de Transferencia de Hipertexto.
- **CRUD**: Crear, Leer, Actualizar, Eliminar.
- **POST**: Método HTTP para enviar datos al servidor.
- **GET**: Método HTTP para obtener datos del servidor.

- ****Editor (Ed)**:** Usuario autorizado que redacta y sube reseñas.
 - ****Usuario (Lector)**:** Visitante que accede a las reseñas, sin autenticación.
 - ****Review (Reseña)**:** Contenido editorial escrito por fotógrafos sobre cámaras específicas.
-

Problema a Resolver

La meta es construir y desarrollar la arquitectura de un sistema backend desde cero hasta la implementación (deploy) del mismo, deberá realizarse tomando en cuenta la planificación de la arquitectura que se detalle en alto nivel. Para ello, los requerimientos que nos ha dado un cliente ficticio son los siguientes:

- La empresa “RandomCameraReviews” necesita un sistema que permita que fotógrafos profesionales suban “reviews” de cámaras fotográficas, para que cualquier persona en todo el mundo pueda buscar los reviews y compararlas a través de su portal. La empresa cuenta con un equipo de developers especializado en frontend que realizará una interfaz de usuario para que los editores suban sus “reviews” y los usuarios puedan verlas, y han solicitado que creemos un sistema backend, incluyendo su API, que permita realizar lo siguiente:
 - Subir reviews de cámaras fotográficas.
 - Obtener el contenido de los reviews para mostrarlo en vistas del portal en sus versiones web y mobile.
 - Manejo de usuarios para editores (no incluye visitantes que leen los reviews).

También se menciona que la empresa “RandomCameraReviews” planea distribuir mayormente en América del Sur, donde está su mercado más grande, pero también tienen ventas en Norte América, Europa y muy pocas en Asia. De igual forma, los editores se encuentran mayormente en América del Sur.

⚡ Objetivos

- Permitir a los editores subir reseñas a través de una API autenticada.
- Proporcionar una API pública para que los usuarios puedan consultar reseñas.
- Construir el sistema backend con enfoque TDD para garantizar confiabilidad.
- Facilitar el despliegue y escalabilidad geográfica para operaciones de lectura.

Stakeholders (Interesados)

- Equipo de Producto (define requerimientos del negocio).
- Equipo de Ingeniería Backend (desarrolla y mantiene la API).
- Desarrolladores Frontend (consumen la API).

- Editores (suben contenido al sistema).
- Usuarios Finales / Lectores (consumen contenido).

💭 Suposiciones

- Solo los editores requieren autenticación y acceso de escritura.
- Los usuarios no necesitan registrarse para consultar las reseñas.
- Los editores están ubicados principalmente en Sudamérica.
- La mayoría de los usuarios están en Sudamérica, Norteamérica y Europa, con menor presencia en Asia.

🚧 Limitaciones y Desconocimientos

En esta sección se describe un listado de limitaciones conocidas, ya sea de recursos o conocimientos, y se deben presentar de forma cuantificable.

- Las estimaciones de tráfico se basan en los mercados conocidos actualmente; un crecimiento rápido puede requerir un balanceo de carga.
- No se contempla la subida de archivos multimedia por el momento.
- No se incluye soporte multilenguaje.
- Las llamadas de la API que permite subir reviews (POST), no excede los límites de latencia de 500ms.
- Las llamadas a la API que permitan leer reviews (GET), deben de tener una latencia menor a 100ms.

🔍 Alcance del Proyecto

✅ Alcance Incluido (Scope)

- API REST con endpoints para la creación de reseñas (`POST /reviews`) y lectura de contenido (`GET /content`).
- Autenticación y control de acceso para editores.
- Almacenamiento y recuperación de datos de reseñas.
- Preparación para distribución geográfica en operaciones de lectura.
- Backend listo para desplegar.

❌ Fuera de Alcance (Out of Scope)

- Autenticación para los lectores.
- Implementación del frontend.

- Subida de imágenes o contenido multimedia.
- Sistema de puntuación o comentarios en las reseñas.

Casos de Uso

Esto se realiza a través de iteraciones con el cliente, donde se describe ejemplos de uso de la aplicación para que haya claridad entre ambas partes.

1. Como editor, me gustaría poder subir una review de una cámara o una review de un lente para cámara, para ello:
2. ****Ed sube una reseña****: El editor autenticado utiliza el endpoint `/reviews` para enviar una reseña.
3. ****Usuario consulta una reseña****: El visitante accede al endpoint `/content` para visualizar reseñas publicadas.
4. ****Editor actualiza una reseña****: (Posible función futura, no implementada en esta versión).
5. ****Escalabilidad para lecturas globales****: El sistema se adapta a alta demanda de lectura en distintas regiones.

Casos de Uso No Soportados

Esto se realiza a través de iteraciones con el cliente, donde se describe ejemplos de uso de la aplicación para que haya claridad entre ambas partes.

1. Como usuario me gustaría poder subir una review de cámara.

Propuesta

Arquitectura General

- ****Frontend****: Desarrollado por otro equipo, consumirá nuestra API REST.
- ****Backend****: API REST desarrollada con Python (FastAPI o Flask).
- ****Base de Datos****: PostgreSQL para almacenamiento estructurado.
- ****Autenticación****: Tokens JWT para validar editores.
- ****Despliegue****: Aplicación dockerizada, compatible con cualquier proveedor cloud.
- ****Distribución Global****: Uso de CDN o réplicas de solo lectura para escalar el endpoint `/content` .

Endpoints de la API

Método	Endpoint	Descripción	Requiere Autenticación
POST	/reviews	Subir una nueva reseña	<input checked="" type="checkbox"/>
GET	/content	Obtener todas las reseñas disponibles	<input type="checkbox"/>
No			

🏢 Componentes del Sistema

- **Servicio de Reseñas**: Maneja la creación y validación de reseñas.
- **Servicio de Contenido**: Optimizado para lectura rápida de reseñas.
- **Servicio de Autenticación**: Emite y valida tokens JWT.
- **Capa de Base de Datos**: Almacena reseñas y credenciales de editores.

📊 Modelos de Datos (SQL)

En esta sección se describen entidades, relaciones, JSONs, tablas, diagramas de entidad-relación, etc. pertenecientes a la base de datos del sistema.

```
```sql
-- Tabla de Editores
CREATE TABLE editors (
 id SERIAL PRIMARY KEY,
 name TEXT NOT NULL,
 email TEXT UNIQUE NOT NULL,
 password_hash TEXT NOT NULL
);

-- Tabla de Reseñas
CREATE TABLE reviews (
 id SERIAL PRIMARY KEY,
 title TEXT NOT NULL,
 content TEXT NOT NULL,
 camera_model TEXT NOT NULL,
 editor_id INTEGER REFERENCES editors(id),
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```

💰 Consideraciones de Costo

Contemplando 100,000 usuarios diarios, que visiten recurrentemente cada hora el sitio, se tienen los siguientes costos:

- **Hosting del Backend:** ~\$30-50 USD/mes (ej. AWS EC2, DigitalOcean).
- **Base de Datos Administrada:** ~\$15-25 USD/mes (ej. AWS RDS, Supabase).
- **CDN o Caché (para /content):** ~\$10-20 USD/mes.
- **Monitoreo y Logs (opcional):** ~\$10-15 USD/mes adicionales.

Elaboración de la Arquitectura del Sistema:

Ya que se haya elaborado un documento de diseño, sabiendo bien el objetivo, alcances y estructura del proyecto, se debe realizar un boceto de la **arquitectura del sistema distribuido** y para ello se pueden utilizar herramientas de diagramas como **Lucidchart** o **app.diagrams.net de draw.io** para describir las funciones de los **servicios backend, bases de datos, conexión y su distribución geográfica**. A continuación, se explicará el proceso de diseño paso a paso:

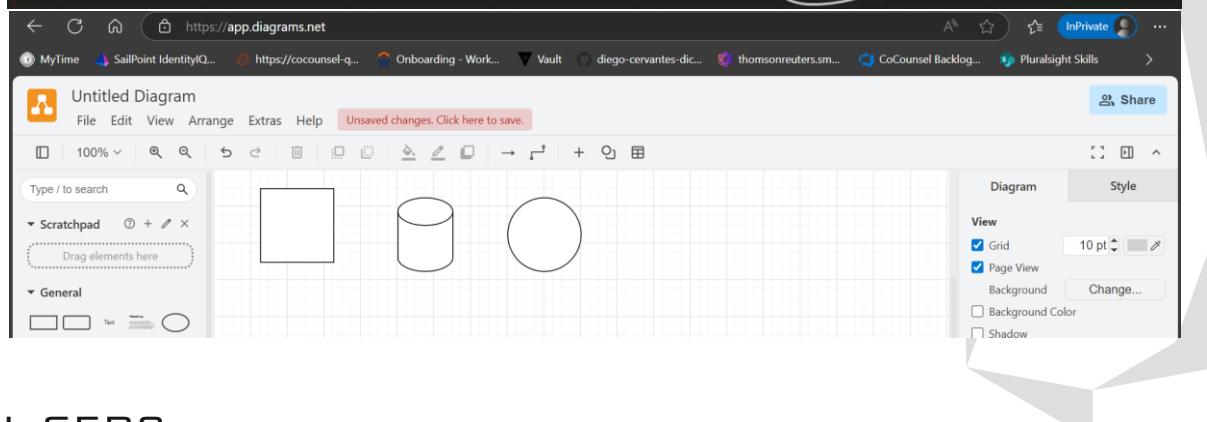
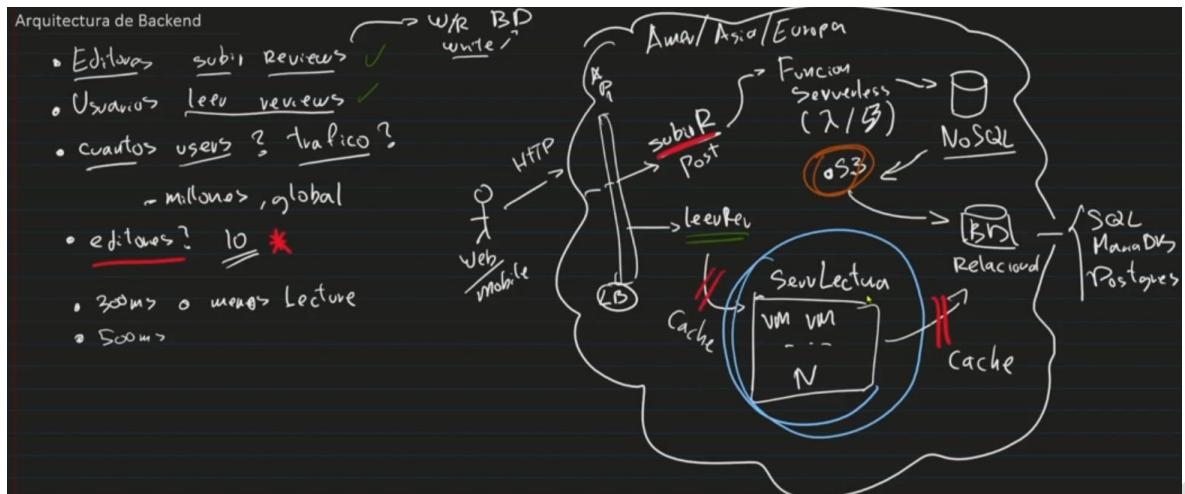
[Flowchart Maker & Online Diagram Software](#)

- Arquitectura Backend: Se diseñará un sistema que permita que los editores puedan subir sus reviews, el sistema además debe permitir tener otro tipo de usuario que no sea editor, que puedan leer dichas reviews. Para desarrollar el sistema contestaremos las siguientes preguntas.
 - **Tráfico:** Debemos saber cuántos usuarios van a estar realizando la lectura de las reviews del sistema y donde se encuentran geográficamente.
 - **Este dato es dado por el cliente: Número de usuarios y localización.**
 - **Acceso y escalamiento de los usuarios en el sistema:** Se sabe que en este caso no se tendrá el mismo número de reviewes que de lectores, ya que se tendrán pocos editores y solamente se encontrarán inicialmente en Sudamérica.
 - **Tiempo de respuesta:** Este es el tiempo en milisegundos en el cual se quiere retornar la respuesta hacia el usuario.
 - **Latencia:** 300ms, 100ms, etc. podrían ser unos tiempos de respuesta estándar, pero este dato se debe **clasificar para los tipos de servicios que dé el sistema**, en este caso **serían dos, el tiempo de respuesta en lectura y en escritura**, esto se define hacia la base de datos como **Write/Read (W/R)** y pueden ser distintos entre sí.
 - **Replicamiento:** Se puede tener una **red de servidores distribuidos** para dar **disponibilidad de datos**, abarcando en este caso **zonas de lectura en América, Europa y Asia** y de **escritura principalmente en América**, pero, aunque el sistema sea **escalable**, para que este sea **replicable**, las APIs de mis servicios deben de estar separadas por función, en este caso teniendo dos tipos de servicios, uno de **subir review (S1)** y el otro de **leer review (S2)**, donde cada uno trabajará en un servidor individual, para que así el sistema también pueda ser replicable, no solo escalable.
 - Para entender un **diagrama de arquitectura debemos tomar en cuenta las capas que lo conforman** de forma agnóstica, las más importantes son:
 - **Load Balancer (Balanceador de Carga o LB):** Distribuye tráfico entrante entre **múltiples servidores escalados horizontalmente** o **funciones cloud**, mejorando el rendimiento, disponibilidad y tolerancia a fallos. Este puede ser reemplazado o combinado con un:

- **API Gateway:** Es un intermediario que gestiona todas las solicitudes externas que llegan a los **endpoints de una API**, sin importar si estos están organizados como microservicios o como una aplicación monolítica. Ejecutando funciones de **enrutamiento, autenticación, control de tráfico, transformación de datos, caché** y monitoreo.
- **Autentificación y Autorización:** Esta capa de la arquitectura (si es que no se ha implementado ya a través de una API Gateway) **sirve para la identificación de usuarios y la autorización de sus acciones**, reconociendo así si el usuario está registrado y que permisos tiene.
- **S1: Backend para subir Reviews ejecutado por medio de Funciones Cloud:** El backend en este tipo de arquitectura tiene una **función serverless (Lambda λ de AWS o Azure function de Microsoft)** asignada a cada endpoint específico de la red, y estos son ejecutados a través del **balanceador de Carga (LB)**. Las funciones representan cada microservicio de nuestra aplicación y los servidores pueden ser gestionados por **Kubernetes**, duplicando así los **contenedores** de cada microservicio dentro de los servidores como sea necesario para cubrir las necesidades de los usuarios, logrando así tener **disponibilidad**.
 - **Concurrencia:** A cada una de las funciones serverless de la arquitectura se le puede asignar una capacidad de procesamiento, dependiendo de su uso esperado durante la operación del sistema, indicando un número de operaciones por segundo fijos para uno de ellos y dejando los demás recursos para las funciones restantes de menos importancia.
- **S2: Backend corriendo en servidores o máquinas virtuales (VM) dentro de zonas de disponibilidad (AZ):** El backend en este tipo de arquitectura cuenta con una característica de **auto escalamiento (autoscaling group)** debido a su naturaleza de **escalamiento horizontal en servidores** por lo que se pueden crear copias de nuestros servicios de forma automática en **Contenedores (Docker o Kubernetes)** para que se monten en servidores adicionales para cubrir la disponibilidad del servicio, teniendo así como resultado una **menor latencia**.
 - **Para este tipo de backend con auto escalamiento** se define un número MÍNIMO de servidores que tengan nodos o copias de nuestro servicio, cual es la cantidad DESEADA promedio y cuál es la cantidad MÁXIMA.
 - **La forma en la que se decide cuándo debe aumentar su número de servidores el auto escalamiento** es a través de **métricas de monitoreo**, donde se define un umbral en CPU, RAM y Disco Duro, aunque lo más recomendable es basarnos en el número de requests de los usuarios.
 - También, **los servidores que vayan siendo creados para cubrir la demanda del incremento de usuarios contiene una AMI**, que es una imagen base con todo ya preinstalado (instancia de contenedor Docker o Kubernetes) para correr nuestros servicios, aunque esta tarda de 3 a 5 minutos en estar disponible, esto porque el **Load Balancer** primero determina que se haya montado correctamente el servidor, antes de que lo podamos utilizar.

- **DB S1: Base de datos:** Esta capa de **database** mínimo siempre debe tener 1 **replicación**, por lo que una de ellas será la **Maestra que recibirá requests** y la otra será su **réplica**, conteniendo así una copia de sus datos que estén siempre sincronizados con la **DB maestra**.
 - **Proxy:** Si queremos utilizar una base de datos relacional, necesitamos implementar una capa de memoria Proxy previa.
 - **Almacenamiento:** Pero como estamos utilizando una base de datos no relacional, la opción de almacenamiento más conveniente sería una memoria **caché (cliente)**.
 - **Bases de datos no relacionales:** La más utilizada con este tipo de arquitectura es la base de datos llave-valor, pero se puede utilizar una basada en documentos, etc. En este caso elegiremos una base de datos no relacional para manejar los **datos del servicio 1 (S1)**.
- **DB S2: Base de datos:** Esta capa de **database** será la **réplica** de la base de datos no relacional NoSQL del **servicio 1 (S1)**, conteniendo así una copia de sus datos que estén siempre sincronizados con la **DB maestra**. Pero debido a que se está haciendo una réplica de una base de datos no relacional a una relacional, se debería tener un tercer servicio que se encargue de esta transformación de datos.
- **S3: Este servicio backend se encarga de transformar los datos No relacionales en relacionales.**

Nota: Este proceso se realiza en entrevistas para arquitectos de software o desarrolladores backend Senior llamada design interview, la cual se realiza en empresas como Google, Amazon, Facebook, etc.

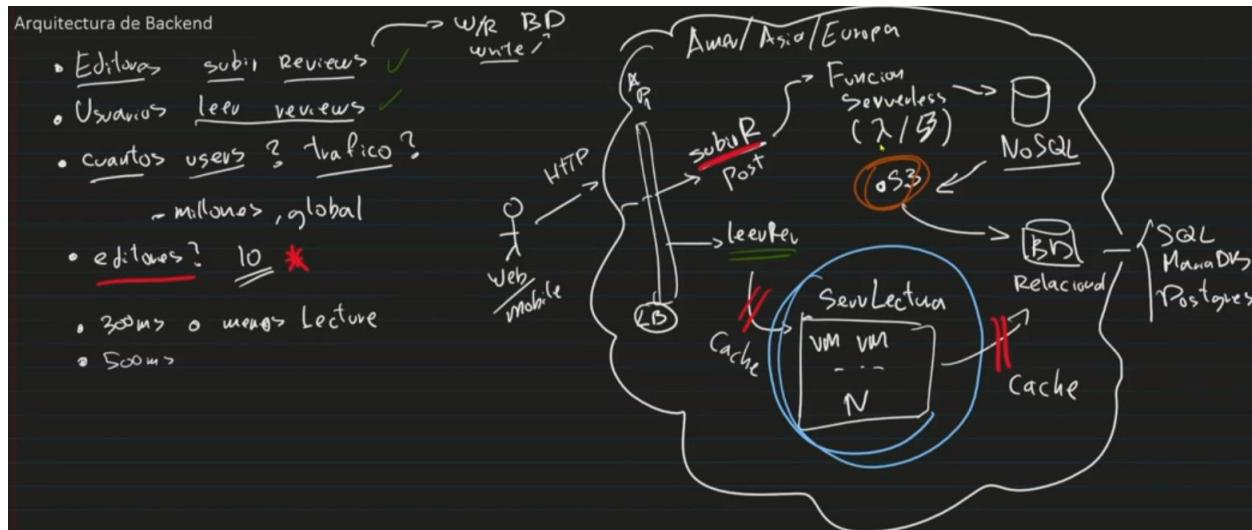


Diseño de Bajo Nivel: Planes de Prueba TDD e Integración Continua CI/CD

El diseño de bajo nivel es más específico y puede contemplar elementos como **planes de prueba (Test Driven Development o TDD)** e **integración continua (CI/CD: Continuous Integration/Continuous Development)**.

Para esta fase **TDD y CI/CD** es muy importante **seguir el diagrama de la arquitectura de alto nivel**, que es un vistazo más por encima del **sistema distribuido** para gente que no tiene que ver con desarrollo de software, además de que este debe haber estado bien hecho (**tomando decisiones siempre contestando la pregunta ¿Por qué?**), ya que, en esta fase de desarrollo, se tomarán en cuenta detalles específicos como el **lenguaje de programación de cada servicio**, *los frameworks que se utilizarán*, los **motores de bases de datos relacionales** (MySQL, PostgreSQL, etc.) **o no relacionales** (MongoDB, Firebase, etc.), el **modelo de datos** que describe las tablas y relaciones de la base de datos, etc.

- **Diseño de Alto Nivel:** Diagrama del sistema distribuido por encima. *Está dirigido a gente de negocio que no tiene tanto conocimiento con temas de desarrollo de software.*

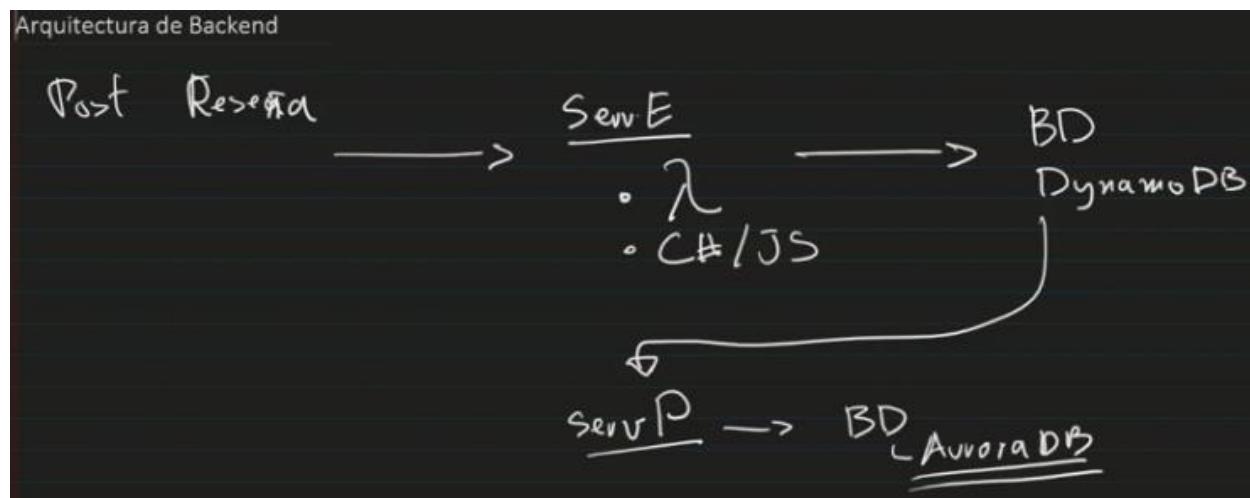


- **Diseño de Bajo Nivel:** Diagrama del sistema distribuido que entra más en detalle, abordando temas como **lenguajes de programación elegidos**, frameworks, **motores de bases de datos**, **modelo de datos**, etc. *Está dirigido a desarrolladores de software.*
 - El diseño de bajo nivel normalmente se divide en módulos, explicando y definiendo así los detalles de solamente una parte del sistema de forma individual.
 - Cabe denotar que para desarrollar cada servicio se puede adoptar un lenguaje de programación diferente dentro de la arquitectura de un sistema distribuido, de igual forma se puede hacer una combinación de bases de datos distintas con motores diferentes para cada servicio o tareas de lectura/escritura (R/W).
- La estructura del documento de bajo nivel básicamente contiene la misma estructura que el documento de alto nivel, pero añadiendo las siguientes secciones en la parte donde se describe la arquitectura, que son:
 - **Diagramas:** Describen diagramas de secuencia y UML (Unified Modeling Language) para definir y mostrar en una representación gráfica la función de cada módulo en el sistema y su interconexión con los demás módulos.

- Para la conceptualización de los diagramas anteriores es de gran utilidad crear una tabla donde se analicen las características de cada servicio o base de datos del sistema.

| Servicio 1 (S1) | | |
|---|--|---|
| Acción que realiza | Características del Servicio | Características de su database |
| Método HTTP POST en un endpoint llamado /reseña | <p>Servicio de escritura E:</p> <ul style="list-style-type: none"> Está montado en una función serverless Lambda de AWS. Su lenguaje de programación puede ser Python, C#, Java o JavaScript. | <p>La base de datos es NoSQL:</p> <ul style="list-style-type: none"> El servicio está montado en AWS, por lo tanto, la base de datos será DynamoDB. |

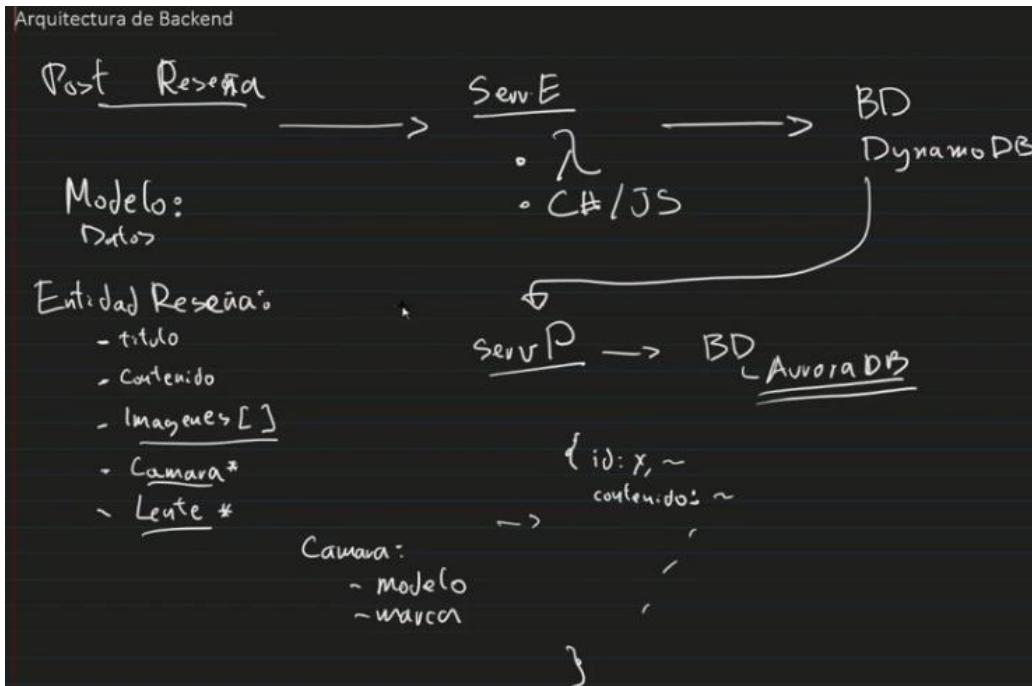
| Servicio 3 (S3) | | |
|--|---|--|
| Acción que realiza | Características del Servicio | Características de su database |
| Procesar los datos recibidos de la función Lambda S1 y pasarlo a una DB relacional de AWS. | <p>Servicio de procesamiento de datos de una base de datos NoSQL a una relacional:</p> <ul style="list-style-type: none"> Está montado en una función serverless Lambda de AWS. Su lenguaje de programación puede ser Python, C#, Java o JavaScript. | <p>La base de datos es NoSQL:</p> <ul style="list-style-type: none"> Su base de datos de entrada NoSQL es DynamoDB. Su database de salida relacional es AuroraDB. |



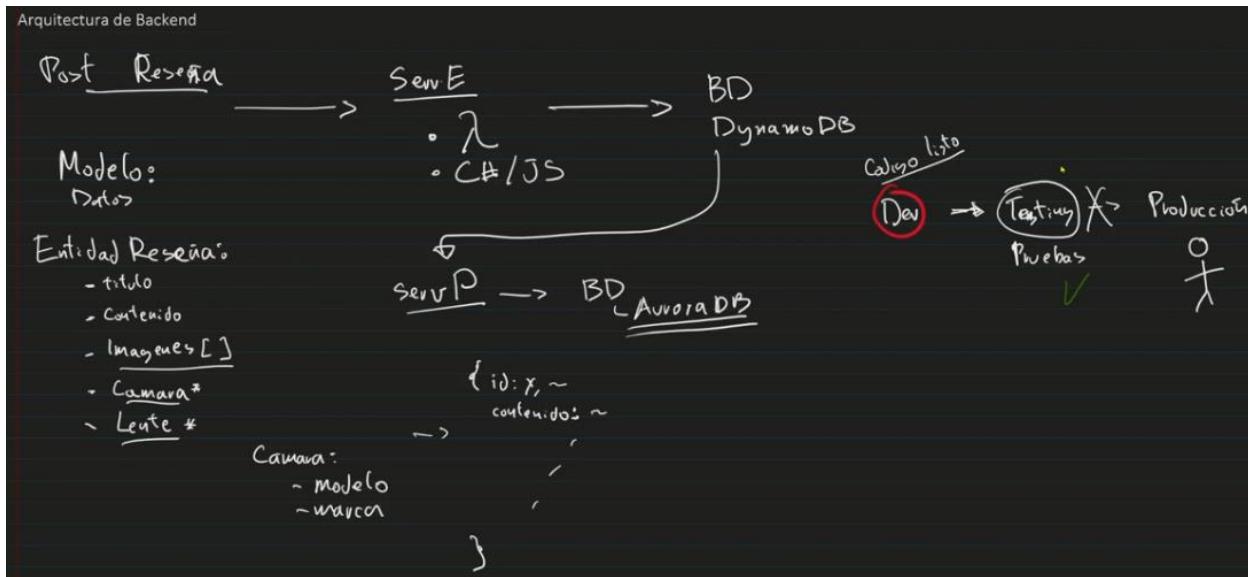
Representación de las Bases de Datos: Definiciones y Diagrama Entidad-Relación

Un **diagrama ER** (**Entidad-Relación**), es una representación gráfica que muestra cómo se relacionan las **entidades** (**objetos, conceptos, personas**) dentro de una **base de datos**.

- **Modelo de datos:** Aquí se describen las características específicas de las **tablas en las DB**, diseñándolas desde cero para soportar el almacenamiento de datos del sistema en cada servicio (S1, S2 y S3), pensando en el método HTTP que soporta cada uno, ya que, dependiendo del método, este recibirá y dará datos o solo recibirá.
 - **Entidad:** Se refiere a una **tabla** que almacena datos sobre un tipo de objeto o elemento del mundo real.
 - Cada **fila** en la **tabla** representa una **instancia individual** de esa **entidad**.
 - Cada **columna** en la **tabla** representa un **atributo o característica** de esa **entidad**.
 - **Atributo:** Son las **columnas de una tabla** que representan las **características o propiedades** de la **entidad** que está siendo modelada, todas ellas tienen un **nombre y tipo de dato** asociado.
 - **Registro:** Representa una **fila perteneciente a una tabla**. También es conocido como "**tupla**" y **contiene los valores** de los **atributos** correspondientes a una **instancia** específica de una **entidad**.
 - Viendo el ejemplo real de esta aplicación, la **entidad reseña del servicio 1 (S1)** puede tener **atributos** como **título, contenido, imágenes** en forma **de lista o array si es que son varias []**, etc. También puede tener atributos que sean en sí tablas separadas, como **cámara y lente de cámara**:
 - **Cámara:** Esta es una entidad separada de reseña, la cual puede tener atributos de **modelo, marca, etc.**
 - **Lente de cámara:** Esto se debe analizar para ver si debe ser una entidad separada o un **atributo** de la **entidad cámara**.
 - Cabe mencionar que como el **servicio 1 (S1)** es de método HTTP POST, de igual forma debemos pensar en los datos que recibe en formato JSON o XML.



- **Plan de Pruebas (TDD - Test Driven Development):** Además de los diagramas de arquitectura y los modelos de datos, vale la pena pensar de igual forma en un plan de pruebas y que valide ciertos casos de uso, simulando así un uso normal de la aplicación y previniendo los casos donde el sistema se pueda romper, comprobando así que todo funciona bien, antes de siquiera hacer un deploy en la nube.
- **Integración continua (CI/CD - Continuous Integration/Continuous Development):** Esta sección contiene diagramas de pipelines, describiendo así el proceso de implementación de features y arreglo de bugs cuando estos se implementen al sistema, realizándoles pruebas cuando se quiera subir dichos cambios a GitHub, indicando hasta el flujo de integración (merge) de ramas y asignando cada una al proceso de desarrollo: Dev branch → Testing branch → Main branch (Producción). *En conclusión, CI/CD es básicamente un flujo escrito o diagramado de cómo queremos que nuestras features o funcionalidades específicas serán aprobadas y subidas al repositorio de nuestro proyecto en GitHub.*



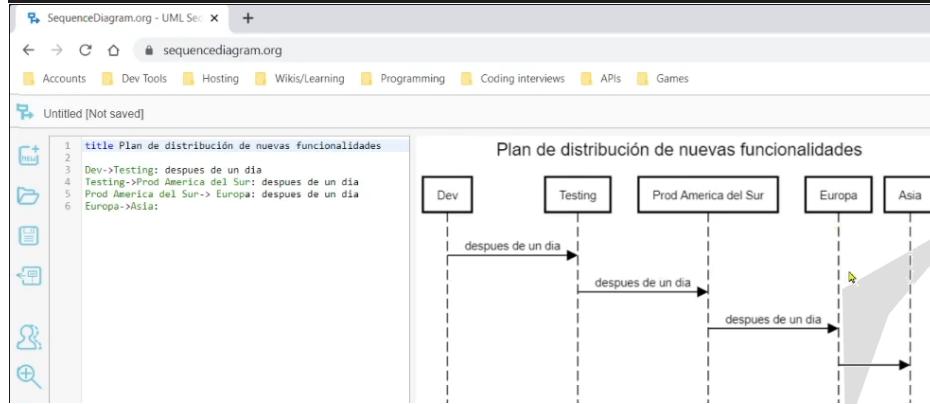
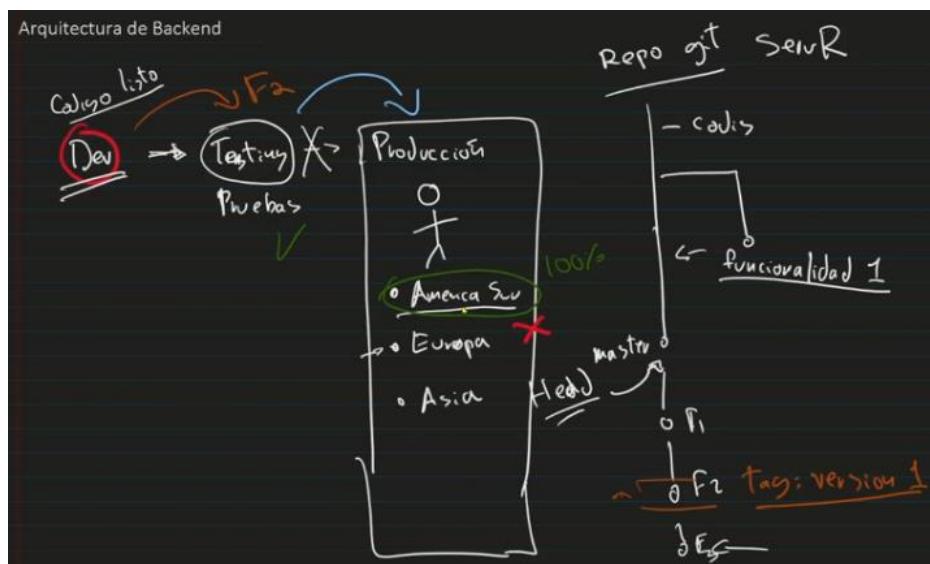
Plan de Integración Continua: CI/CD - Continuous Integration/Continuous Development

Esto se refiere a una sección en el documento de Diseño de arquitectura de software de bajo nivel, el cual engloba el concepto de **rollout**, el cual significa el proceso de pasar una funcionalidad desarrollada en código a través de algo llamado pipeline que es una tubería de pasos que hay que seguir para que el código que escribimos al 100% funcione, pasándolo por ciertas pruebas para comprobar su calidad, asegurándonos de entregar un resultado correcto de nuestro servicio a todos los usuarios. A medida que nuestro sistema se vaya escalando horizontalmente, el pipeline tiene que ser expandido para así asegurar que el código sea entregado siguiendo un estándar de calidad a todas esas regiones. Las etapas de las que se conforma la integración continua son las siguientes:

- **Development:** Es aquella fase de desarrollo donde se experimenta con el código, creándolo desde cero para implementar una nueva feature (funcionalidad) o arreglar un bug (error en el código).
 - Esto se hace a través de ramas en GitHub, la cual representa cada funcionalidad o bug que se quiere efectuar y luego necesitamos levantar algo llamado Pull Request o PR que mostrará el código que fue cambiado, borrado o agregado, para luego pedir que otro

desarrollador la revise y apruebe, osea le de un review y cuando este cambio sea aprobado, que se realicen pruebas del pipeline elegido, para posteriormente hacer un merge de dicha rama (fusión) con la main branch, integrando de esta forma nuestro nuevo cambio al código principal del proyecto.

- No forzosamente lo que esté en la rama main será lo que verá el cliente, osea la versión publicada del proyecto.
- Pruebas (Testing): Cuando una rama de feature o bug quiera ser fusionada con la rama de main, debe existir una fase intermedia donde además de que el programador que dio el review a los cambios de código, se ejecuten pruebas automatizadas para comprobar la funcionalidad del código, dando así otra capa de monitoreo de calidad del código. Esto de igual forma se puede probar a través de tags, que son versiones o checkpoints definidos de nuestro proyecto en GitHub. Una de las herramientas de testing más utilizadas es Jenkins.
 - **RollOut:** Cuando un cambio de código haya pasado las pruebas de calidad del pipeline, se puede añadir una herramienta que realice el deploy de este cambio a cada región donde esté distribuido de forma horizontal el código, ya sea América, Europa, Asia, etc. de forma programada y haciéndolo paso a paso de una región a otra siguiendo un orden específico.
 - **Diagramas de Secuencia:** Son los diagramas que describen el proceso que sigue el pipeline de nuestro código para mantener así cierto estándar de calidad. Para ello se puede utilizar esta herramienta: <https://sequencediagram.org/>



Seguimiento de Tareas en DevOps: Code Complete en Trello, Azure Dev Ops, etc.

Algo importante de mencionar igual en la planeación de un proyecto es el concepto de Code Complete, el cual se refiere a cuando una tarea está completada y no hay que escribir ni una sola línea más de código, para que esto pueda suceder, se deben haber terminado de ejecutar todos los casos de uso mencionados en el Documento de Diseño. Ya que se haya terminado de realizar los casos de uso descritos en el proyecto, para realizar tareas de mantenimiento posteriores, se puede utilizar herramientas como Trello, Azure Dev Ops, etc. para agregar nuevas features o arreglar bugs, los tickets de igual forma se apoyan de Pull Requests de GitHub, las cuales deben haber sido levantadas, revisadas, aprobadas y mergeadas a main para que la tarea del ticket se pueda considerar como code complete.

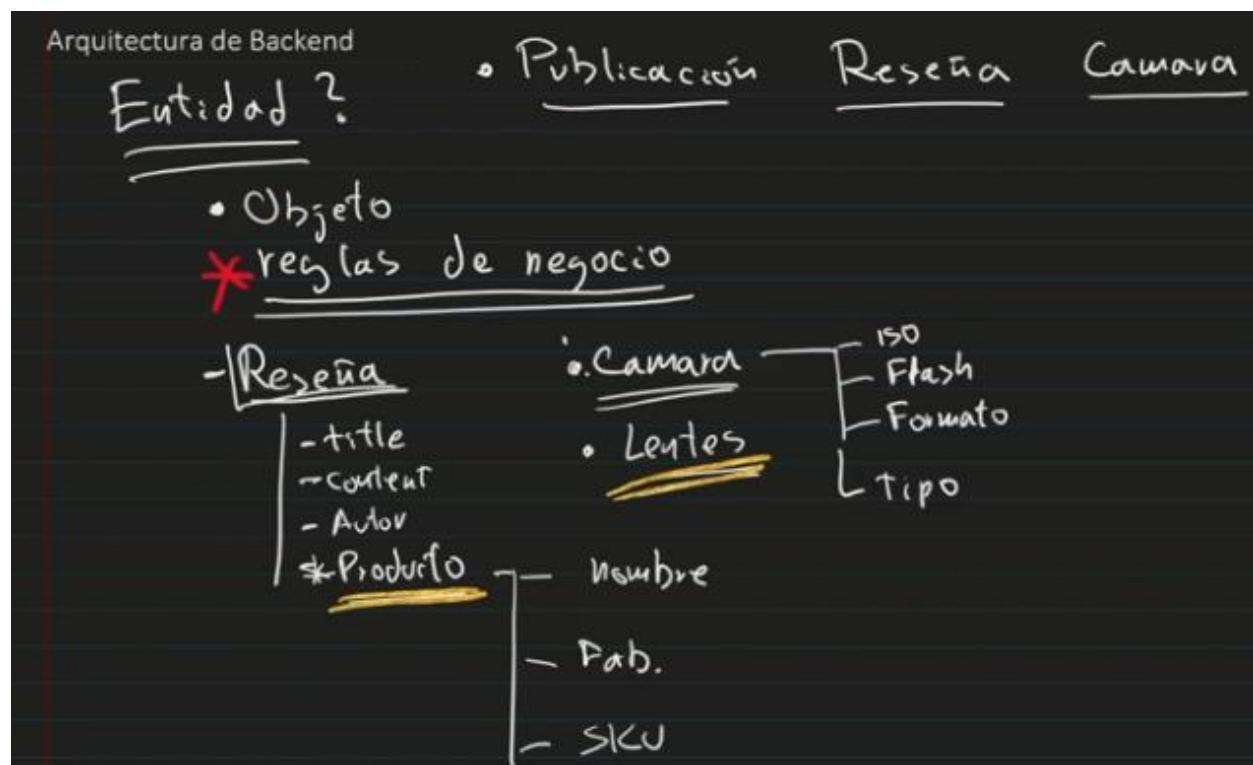
Nota: Para poder desarrollar un buen software, se debe entender el negocio.

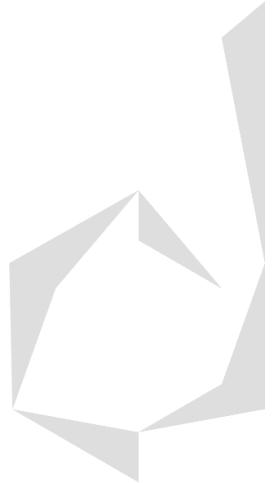
Desarrollo del Código del Proyecto

A continuación, se desarrollarán las entidades de la base de datos, el código backend, el pipeline del proyecto (CI/CD) y la plataforma de seguimiento de tickets para el desarrollo con Trello. Para realizar estas acciones nos basaremos en los requerimientos del proyecto y en los casos de uso descritos en el documento de Diseño.

Creación de las entidades de la base de datos

Ahora deberemos diagramar las entidades o tablas de la base de datos que soporten las reglas de negocio del caso de uso publicar una reseña en el sitio web de la empresa “RandomCameraReviews”.





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/>

Platzi, Carlos Zambrano, “Curso de Introducción a la Nube”, 2023 [Online], Available: <https://platzi.com/cursos/intro-nube/>

Platzi, Jorge Villalobos Gutiérrez, “Curso Práctico de Arquitectura Backend”, 2023 [Online], Available: <https://platzi.com/cursos/practico-backend/>

Lucas Da Costa, “Testing JavaScript Applications”, 2021, 1st Edition.

