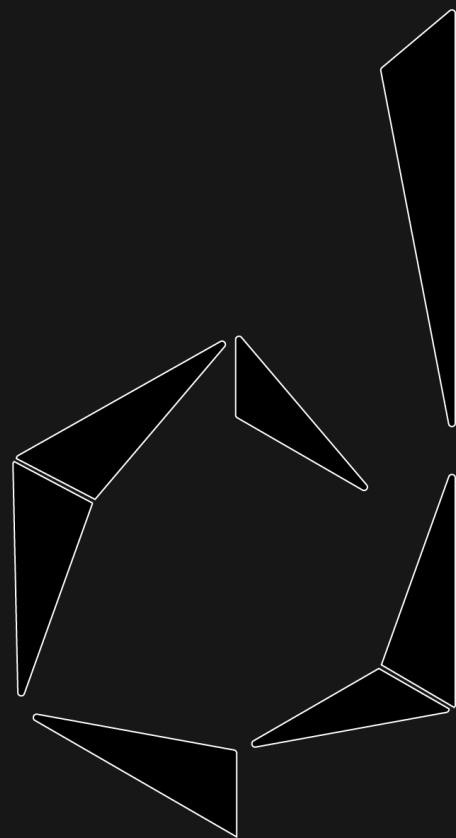


INGENIERÍA MECATRÓNICA



DI_CERO

DIEGO CERVANTES RODRÍGUEZ

PROGRAMACIÓN: DESARROLLO BACKEND

POSTMAN, INSOMNIUM, PYTHON, ETC.

Ejemplo de
Arquitectura Distribuida

Contenido

Conocimientos Previos	2
Proyecto Backend	9
Definición de los Requerimientos del Negocio	9
Documento de Diseño de Software: High Level System Design	10
Elaboración de la Arquitectura del Sistema:.....	18
Diseño de Bajo Nivel: Planes de Prueba TDD e Integración Continua CI/CD	21
Representación de las Bases de Datos: Definiciones y Diagrama Entidad-Relación	22
Plan de Integración Continua: CI/CD - Continuous Integration/Continuous Development	24
Seguimiento de Tareas en DevOps: Code Complete en Trello, Azure Dev Ops, etc.	26
Desarrollo del Código del Proyecto	26
Creación de las entidades de la base de datos	26
Referencias.....	28



Conocimientos Previos

- **Backend Developer:** Es el desarrollador encargado de construir la lógica interna de una aplicación, se ocupa del manejo de **bases de datos**, servidores, **autenticación**, **seguridad** y reglas de negocio. Su trabajo no es visible para el usuario, pero es esencial para que el sistema funcione correctamente.
- **Backend, Frontend y Fullstack Developer:**
 - **Backend Developer:** Se enfoca en el **procesamiento de datos**, **servicios**, **APIs** y lógica de negocio en el sistema.
 - **Frontend Developer:** Trabaja en la interfaz visual que el usuario ve e interactúa, usando tecnologías como HTML, CSS y JavaScript.
 - **Fullstack Developer:** Tiene habilidades en ambas áreas y puede desarrollar una aplicación completa desde la interfaz hasta el servicio y la **base de datos**.
- **HTTP (HyperText Transfer Protocol):** Es el protocolo estándar que define cómo se envían y reciben datos en la web. **Cada solicitud HTTP incluye un método (como GET o POST) y devuelve un código de estado (como 200, 404 o 500) que indica si fue exitosa, fallida o si hubo un error del servidor.**

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	400 - Bad Request: Solicitud hecha de forma errónea.

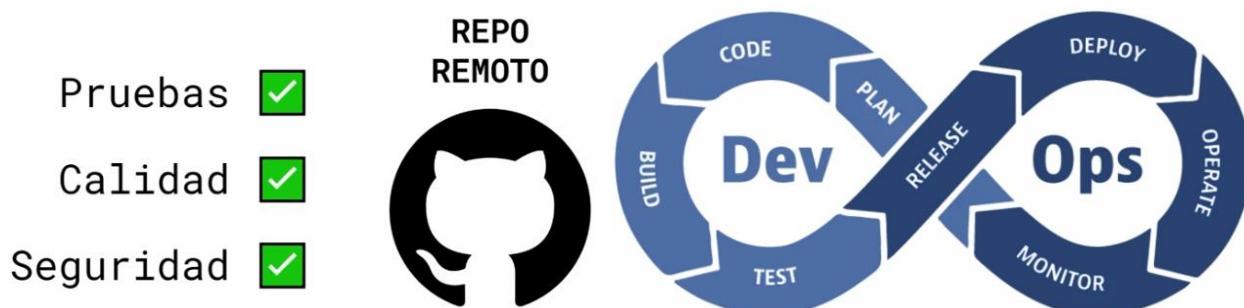
	porque el cliente envió una solicitud de forma errónea.	401 - Unauthorized: No autorizado. 403 - Forbidden: Credenciales válidas, pero acceso denegado. 404 - Not Found: Recurso no encontrado. 409 - Conflict: Conflicto con el estado actual del recurso. Por ejemplo, datos duplicados.
500-599	Representan 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.

- **API (Application Programming Interface):** Es una interfaz que permite la comunicación entre distintos sistemas. En el contexto web, conecta el frontend con el backend usando métodos HTTP, como si fuera un "mensajero" que lleva las solicitudes del usuario al servidor y devuelve los resultados.
- **Estructura REST:** Es una arquitectura para construir APIs que sean predecibles, simples y eficientes. Usa los **métodos estándar de HTTP (GET, POST, PUT, DELETE)** para crear una estructura **CRUD (Create, Read, Update & Delete)**, permitiendo así construir aplicaciones backend escalables y fáciles de mantener.
 - **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.

- **API Testing (Postman, Insomnia):** Son herramientas de escritorio que permiten enviar solicitudes a APIs de manera manual para validar que los endpoints funcionan correctamente. Facilitan la depuración de errores, la visualización de respuestas y la automatización de pruebas básicas.

The screenshot shows the Insomnia application interface. On the left, there's a sidebar with 'My first collection' and various configuration options like 'Base Environment', 'Add Cookies', and 'Add Certificates'. In the center, a request card is displayed for a 'GET' operation to 'https://api.escuelajs.co/api/v1/categories'. The 'Params' tab is selected, showing the URL and query parameters. The 'Headers' tab shows 'Content-Type: application/json'. The 'Preview' tab on the right shows the JSON response, which lists four categories: Electronics, Furniture, Shoes, and Books. The status bar at the bottom indicates 'Just Now' with a timestamp of '2025-05-22T18:33:45.000Z'.

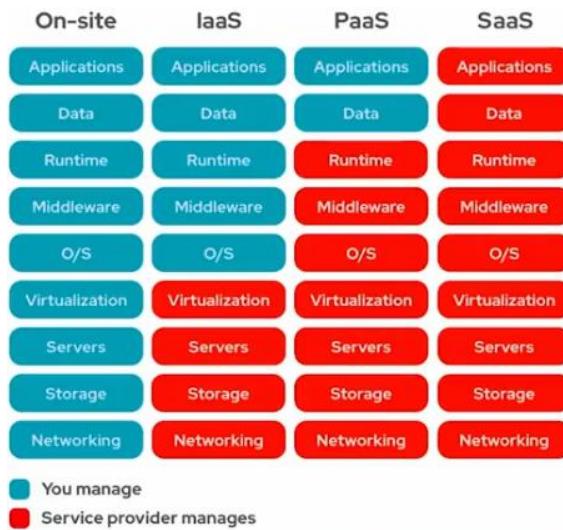
- **Cloud (Nube):** Es un modelo de computación que permite acceder a servidores, almacenamiento y servicios a través de internet. En lugar de instalar infraestructura física, se alquilan recursos en plataformas como AWS, Google Cloud o Azure, escalando según la demanda.
- **DevOps:** Es una metodología que une los equipos de desarrollo (Dev) y operaciones (Ops) para automatizar procesos como pruebas, integración continua, despliegue y monitoreo. Busca reducir errores, acelerar entregas y mejorar la calidad del software.



- **Modelos de Arquitectura: On Site, IaaS, PaaS, SaaS:**

- **On Site:** La infraestructura (servidores, red, almacenamiento) se encuentra físicamente en las instalaciones del cliente.
- **IaaS (Infrastructure as a Service):** Se alquila infraestructura virtualizada, como servidores o bases de datos, con alto control.
- **PaaS (Platform as a Service):** Se provee una plataforma lista para desplegar código, sin preocuparse por la infraestructura.

- **SaaS (Software as a Service):** El usuario accede a un software ya listo para usar, sin necesidad de desarrollarlo ni mantenerlo.

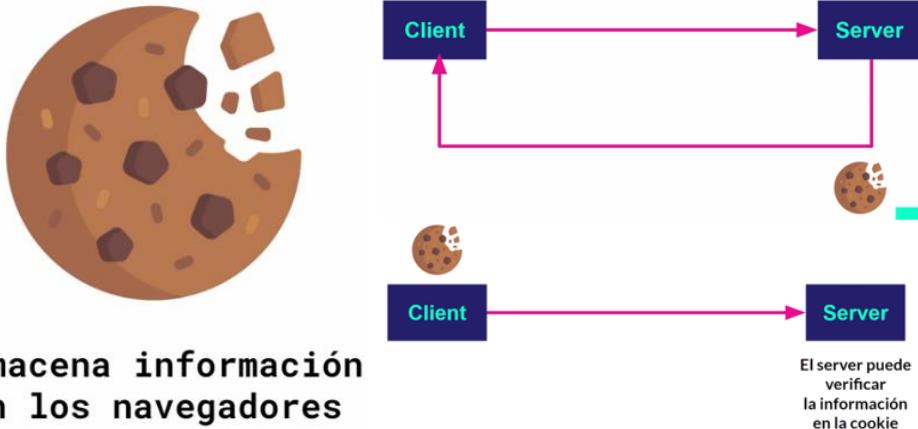


- **PaaS (Heroku):** Plataforma como servicio que permite desplegar aplicaciones web de forma sencilla. El desarrollador solo sube su código y Heroku se encarga del servidor, base de datos, red y escalabilidad, ideal para proyectos rápidos y prototipos.
- **Escalamiento Vertical:** Consiste en aumentar la capacidad de un solo servidor (más RAM, CPU o almacenamiento) para manejar mayor carga. Es simple, pero tiene un límite físico y puede generar puntos únicos de falla.
- **Escalamiento Horizontal:** Implica añadir múltiples servidores o instancias que trabajan en paralelo. Esta técnica mejora la disponibilidad, tolerancia a fallos y capacidad de atender a muchos usuarios simultáneamente.

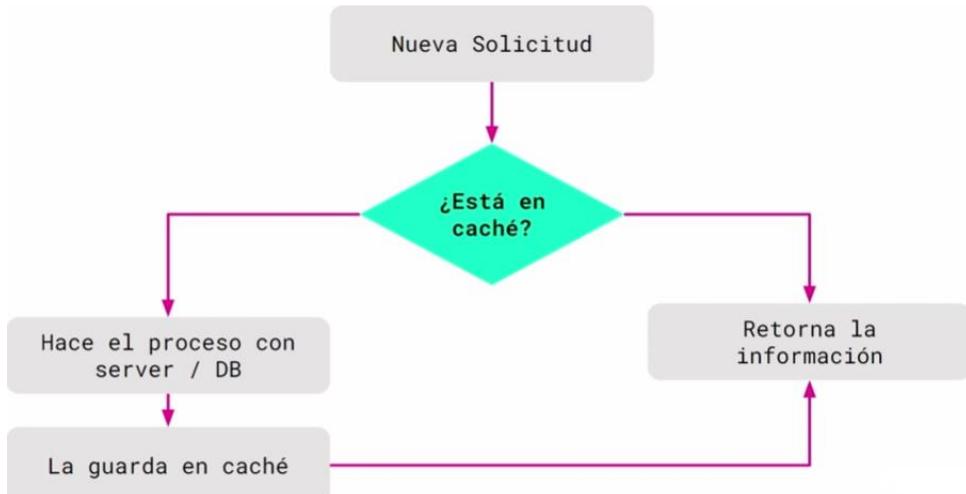


- **Bases de Datos:** Son sistemas organizados que almacenan, gestionan y consultan datos de forma persistente. En backend, se utilizan para guardar información como usuarios, productos, reseñas, etc., y permiten conectividad rápida mediante consultas SQL o NoSQL.
 - **Replicación de Bases de Datos:** Técnica para copiar y sincronizar datos entre múltiples bases de datos. Aumenta la disponibilidad, mejora tiempos de lectura y permite la distribución geográfica del sistema.
- **Colas de Tareas (Queues):** Son estructuras que almacenan tareas o procesos en espera, para ser ejecutados de forma asíncrona y ordenada. Se usan para manejar operaciones pesadas (como procesamiento de archivos o notificaciones) sin afectar la experiencia del usuario.

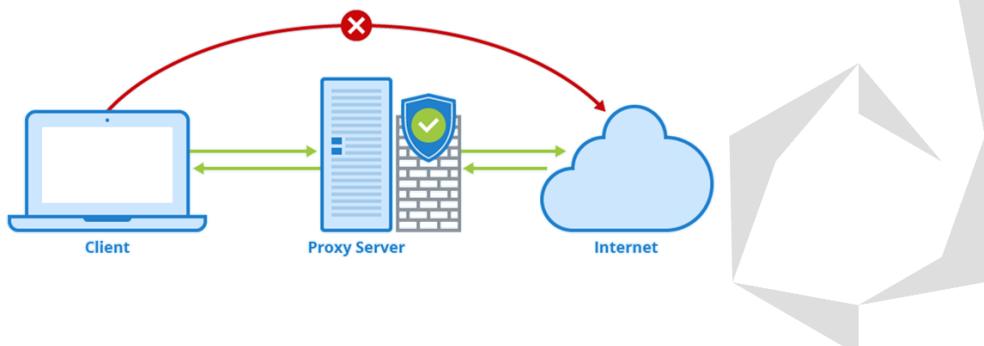
- **Server Side Rendering (SSR):** Es el proceso de generar contenido HTML desde el servidor y enviarlo completamente renderizado al navegador. Mejora el rendimiento inicial de carga y es beneficioso para el SEO.
- **Cookies/Sesiones (Memoria Cliente):** Son mecanismos para guardar información temporal en el navegador del usuario. Las cookies pueden recordar preferencias, mantener sesiones activas o almacenar identificadores de usuario entre visitas.



- **Caché (Memoria Servidor):** Es una copia temporal de datos almacenada en el servidor para acelerar respuestas futuras. Reduce el uso de la base de datos y mejora la eficiencia de la API.



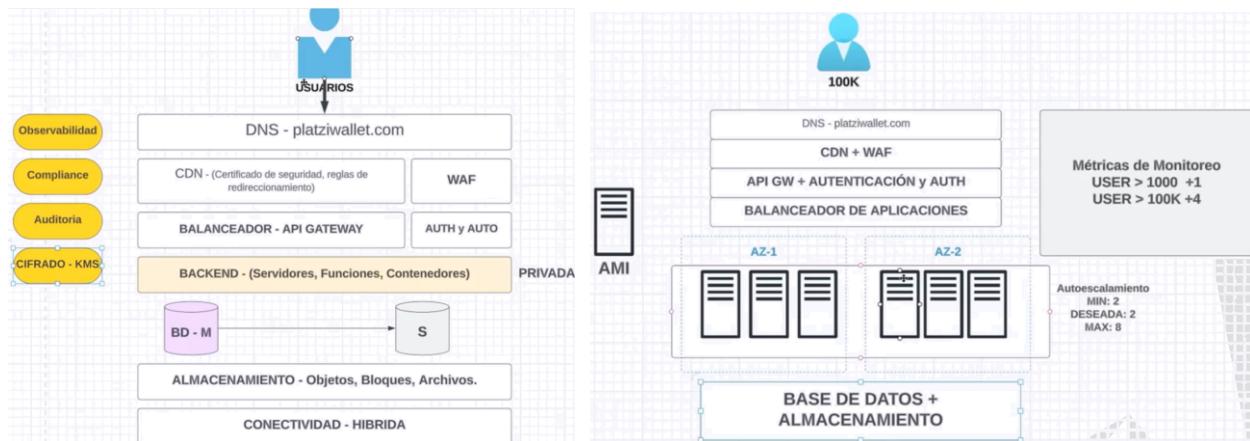
- **Proxy (Memoria Intermedia):** Es un servidor intermediario que actúa entre el cliente y el backend, almacenando respuestas comunes o redirigiendo tráfico. Aumenta la velocidad de respuesta y mejora la seguridad.

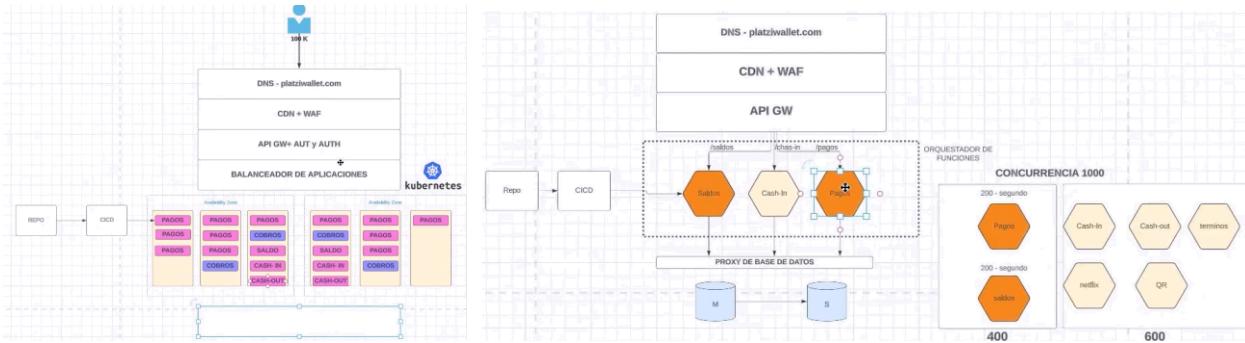


Diferencias entre las Cookies, Memoria Caché y Proxy son:

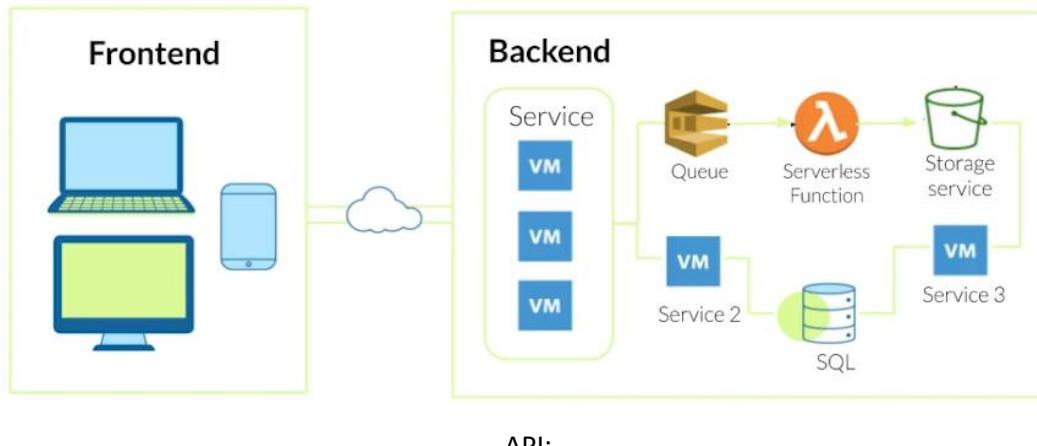
Característica	Cookie/Sesión	Memoria Caché	Memoria Proxy
¿Dónde almacena sus datos?	En el navegador.	En el servidor o navegador.	En un servidor intermedio proxy.
¿Qué almacena?	Resultados previos del navegador.	Resultados previos del servidor.	Copias de las respuestas HTTP/HTTPS.
¿A quién ayuda?	Al servidor 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.

- **Bucket (Memoria Nube):** Es un espacio de almacenamiento en la nube (como en AWS S3 o Firebase Storage) que permite guardar archivos como imágenes, audios o documentos, accesibles vía URL o autenticación.
 - **Arquitectura y Despliegue:** Se refiere a cómo están organizados los componentes de un sistema (bases de datos, servidores, servicios, etc.) y cómo se pone en marcha para estar disponible al usuario, incluyendo herramientas de automatización y escalabilidad.
 - **Docker:** Es una herramienta que permite empaquetar una aplicación junto con todas sus dependencias en un contenedor portátil y reproducible. Facilita el desarrollo, pruebas y despliegue en distintos entornos.
 - **Kubernetes:** Es una plataforma de orquestación que gestiona múltiples contenedores Docker. Administra el escalado, despliegue, distribución y recuperación automática de aplicaciones en producción.
 - **Servidor Multinube:** Es una estrategia de despliegue que utiliza más de un proveedor de servicios en la nube simultáneamente. Aumenta la disponibilidad, evita la dependencia de un solo proveedor y mejora la tolerancia a fallos.



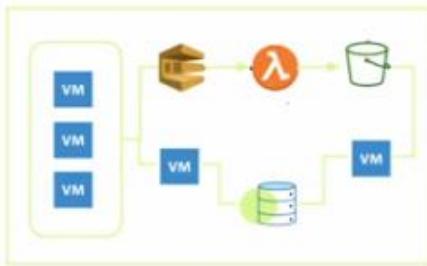


Nota: Vale la pena mencionar que todas las arquitecturas pasadas pueden ser mezcladas para obtener el mejor resultado en nuestras aplicaciones.



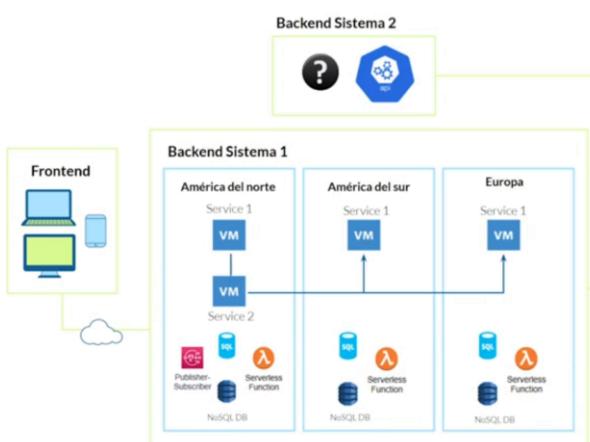
API:

Cómo está implementado el sistema



Lo que hace el sistema

- Registrar usuario (POST)
- Obtener Usuario (GET)
- Actualizar Usuario (PUT/PATCH)
- Comprar Producto (POST)
- Reembolsar Producto (POST)



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:

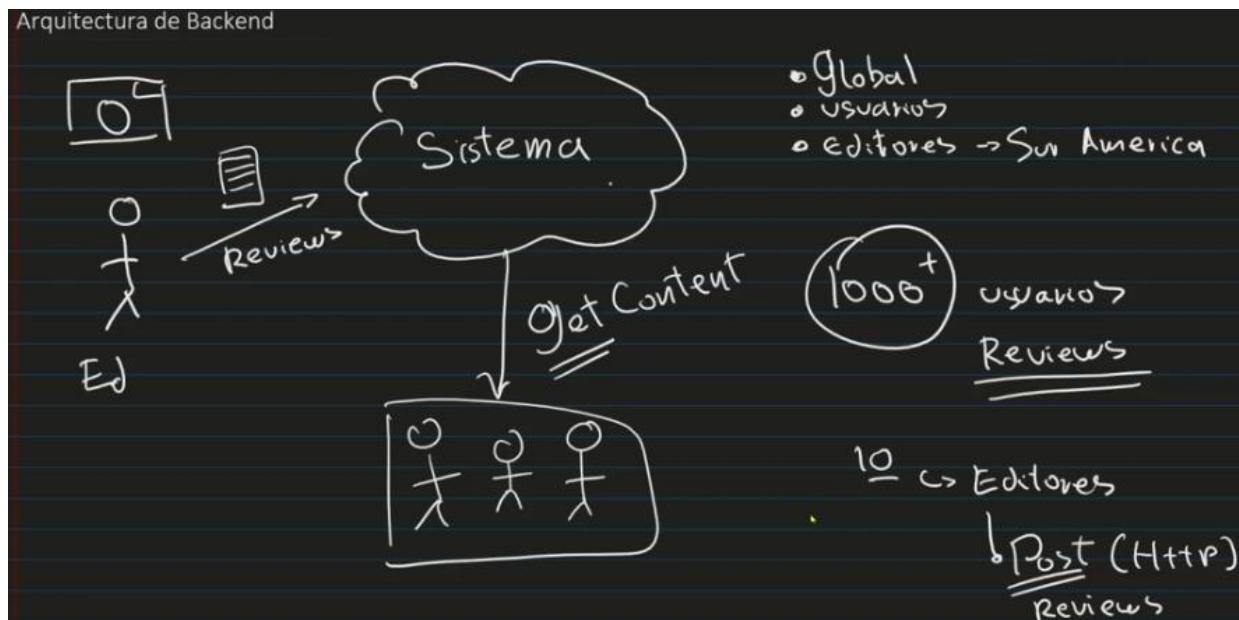
- La empresa “RandomCameraReviews” necesita un sistema que permita que fotógrafos profesionales suban “reviews” de cámaras fotográficas, para que cualquier persona en todo el mundo pueda buscar los reviews y compararlas a través de su portal. La empresa cuenta con un equipo de developers especializado 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:

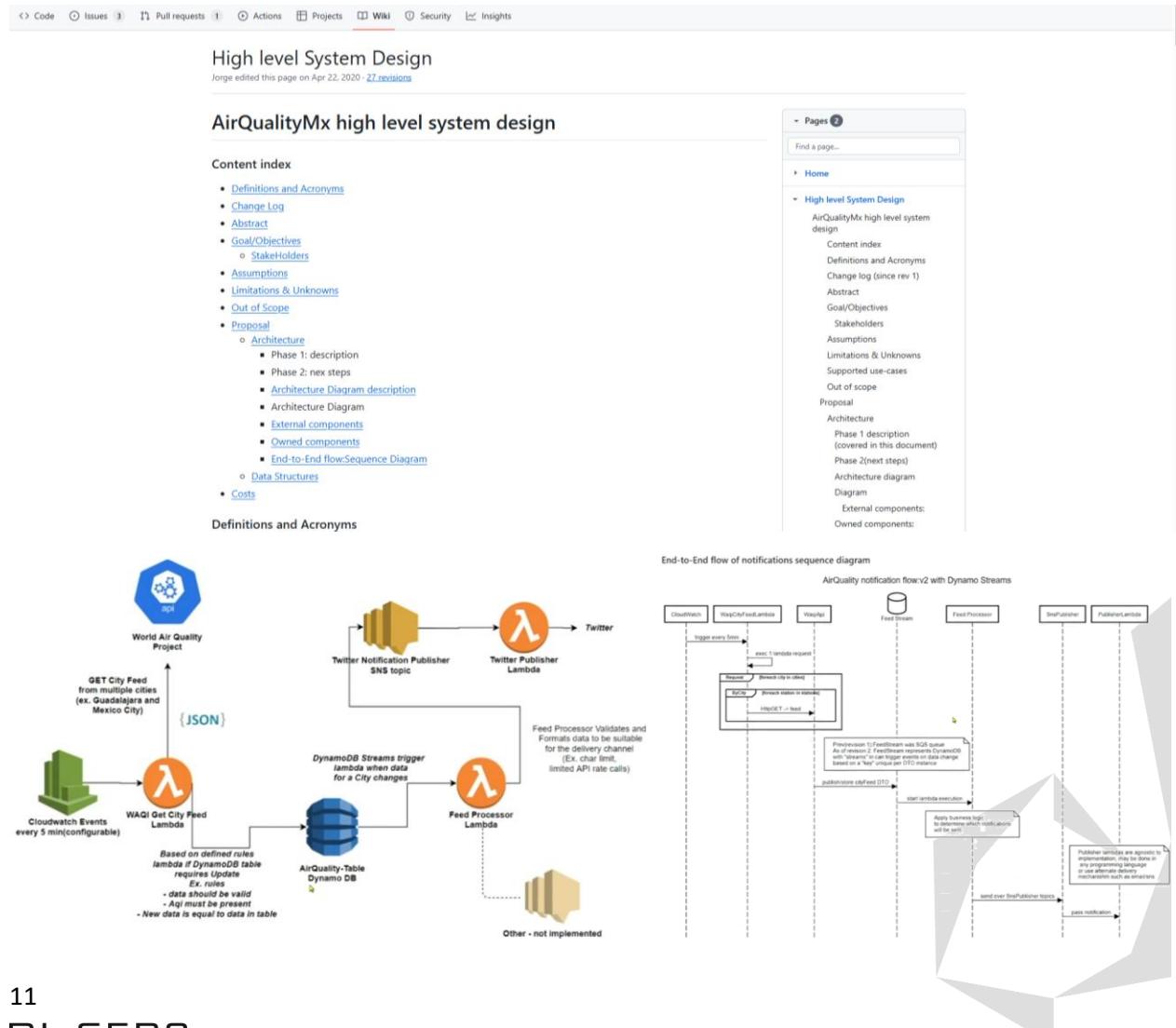
- **Definiciones y acrónimos:** Aquí se describe conceptos clave que se vayan a mencionar en el documento, para que cualquier persona de producto (no desarrollador de software) pueda entender a qué se refiere cada cosa.
- **Problema Por Resolver (Abstract/Overview):** Es una descripción general del proyecto.
- **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.

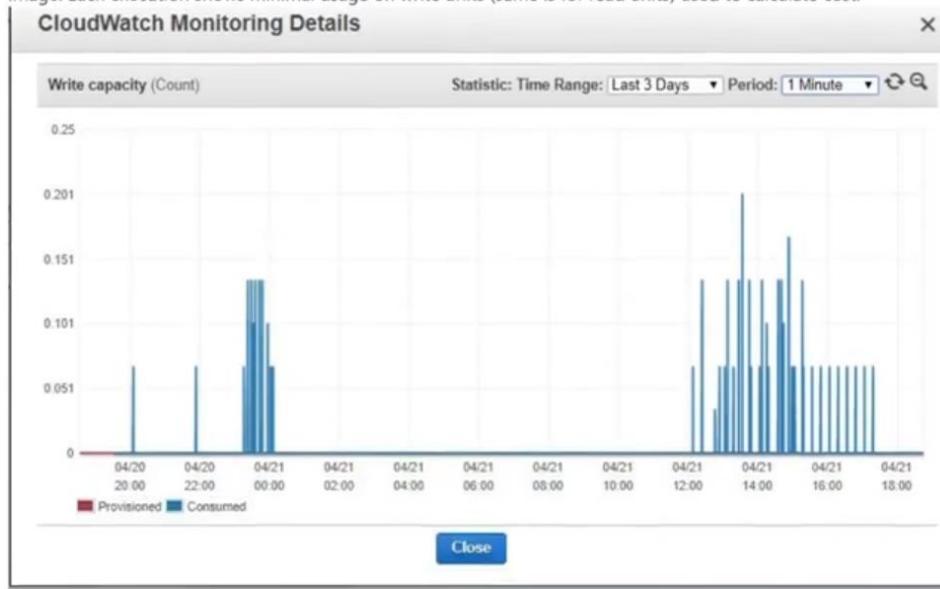
Un ejemplo de este tipo de documentos se encuentra en el siguiente repositorio de GitHub: [High level System Design · jorgevgut/airquality-mx Wiki](#)

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.



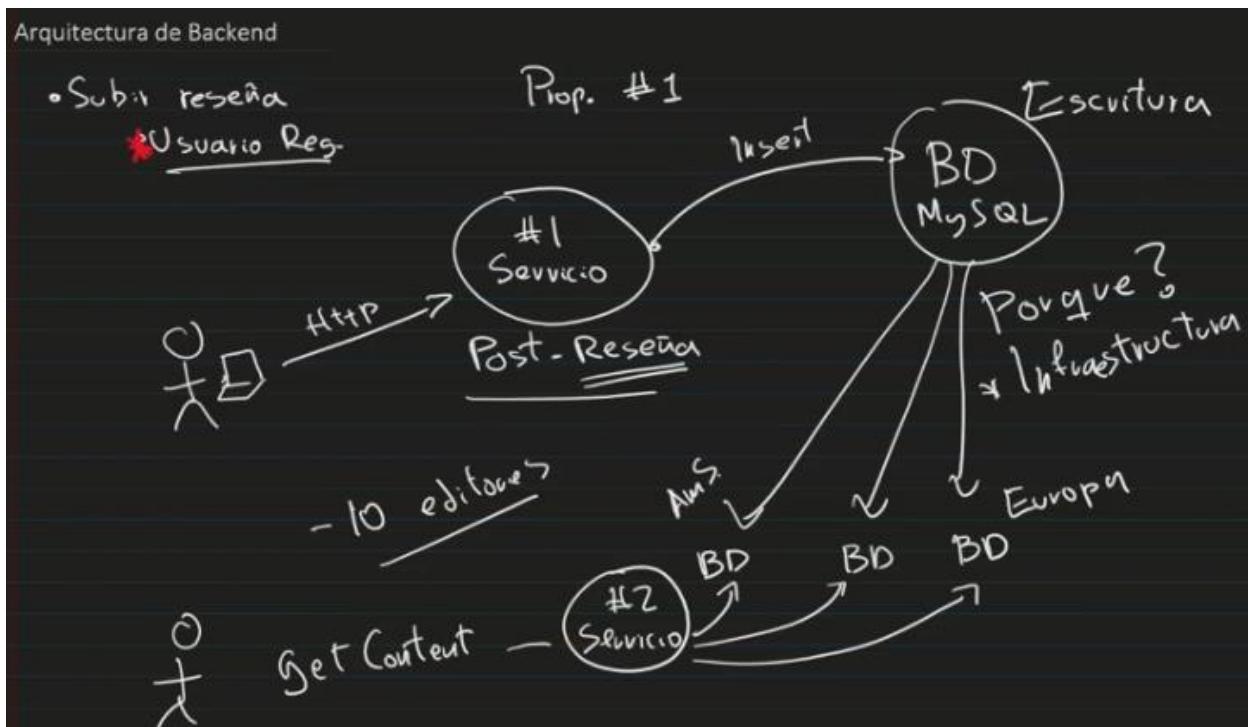
DynamoDB costs: Upon testing, costs are significantly low to perform a reliable calculation, as shown in the following image. Each execution shows minimal usage on write units (same is for read units) used to calculate cost.



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:

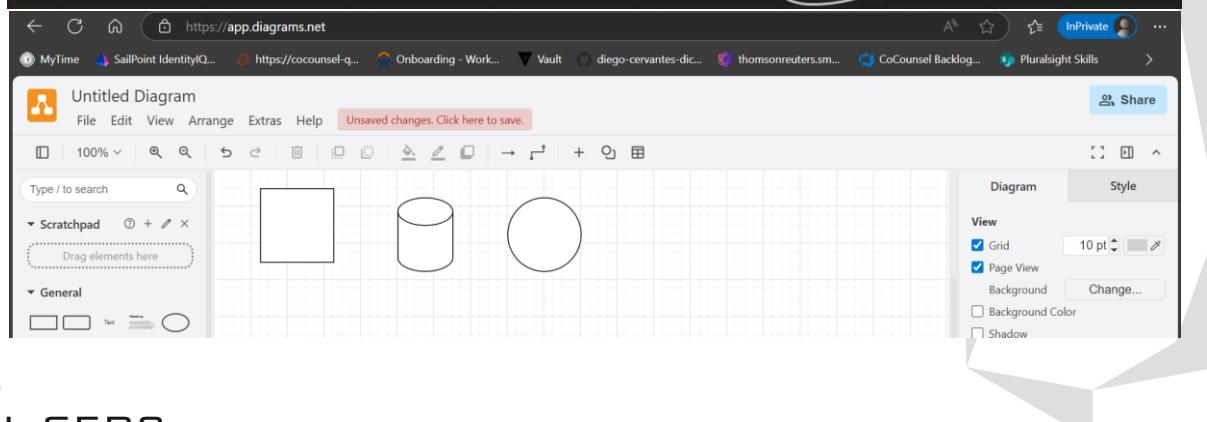
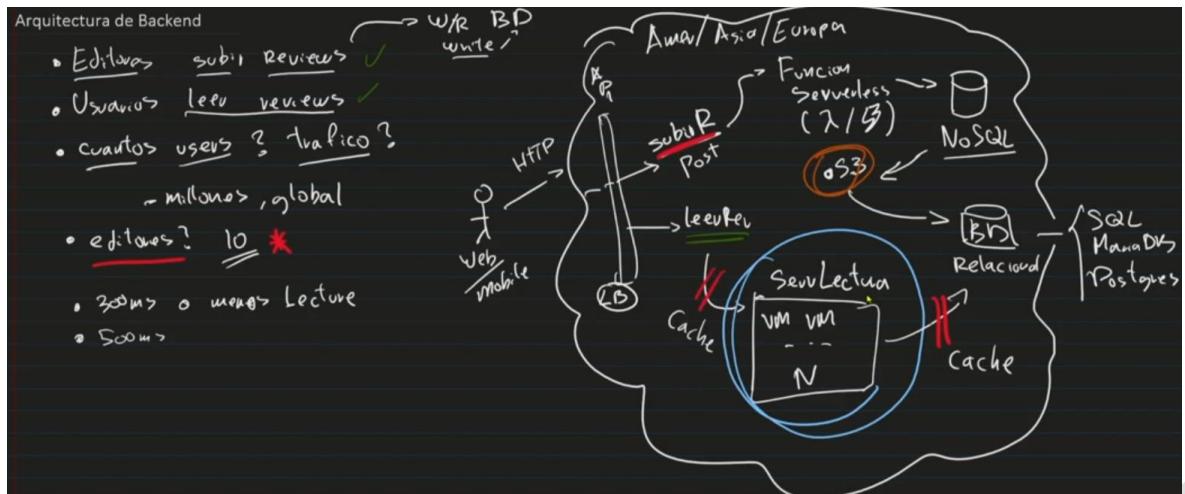
[Flowchart Maker & Online Diagram Software](#)

- 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.

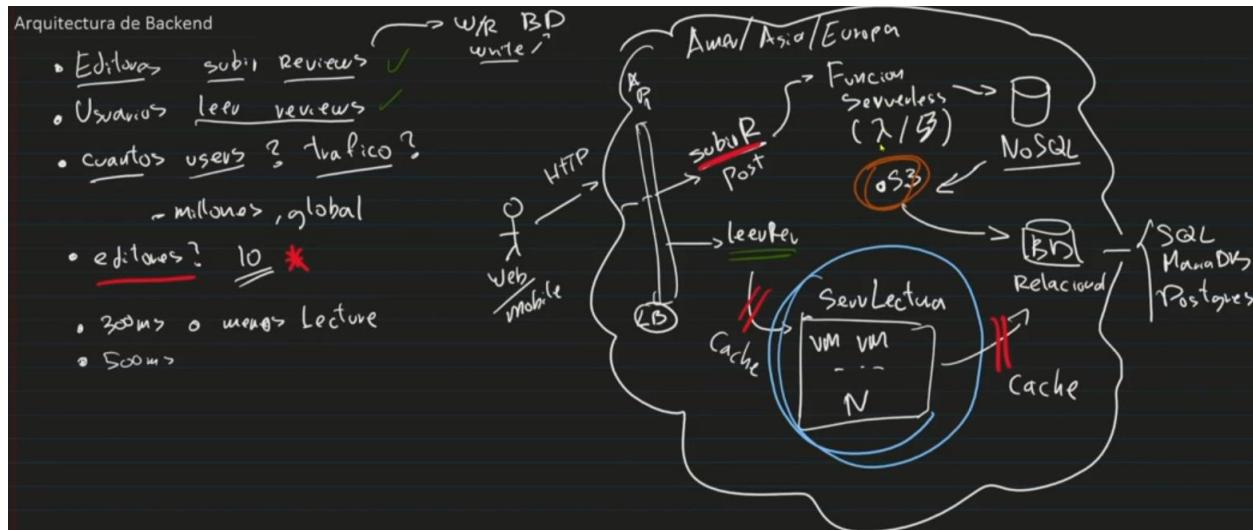


Diseño de Bajo Nivel: Planes de Prueba TDD e Integración Continua CI/CD

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.*

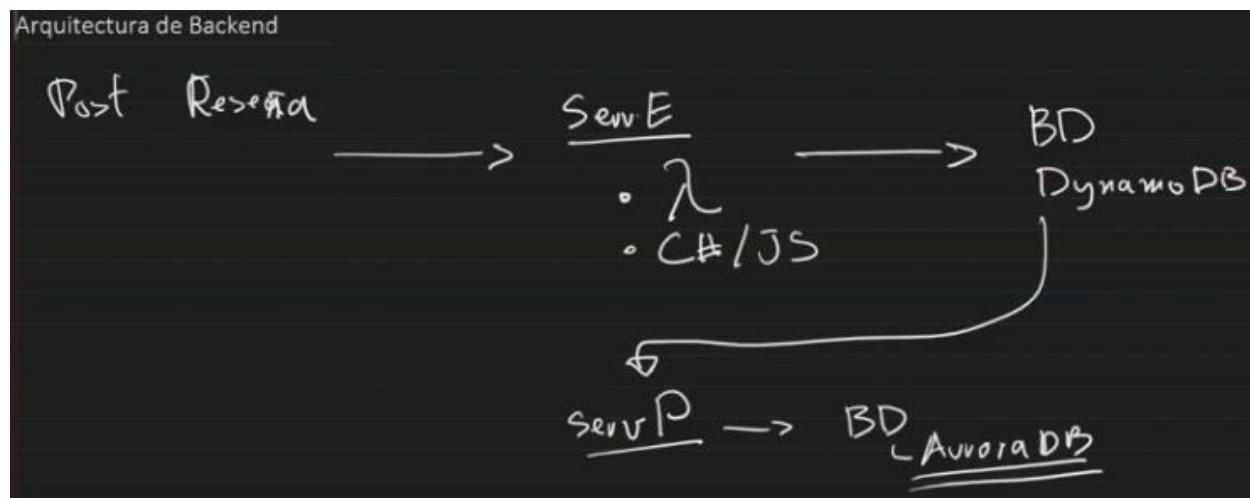


- **Diseño de Bajo Nivel:** Diagrama del sistema distribuido que entra más en detalle, abordando temas como **lenguajes de programación elegidos**, frameworks, **motores de bases de datos**, **modelo de datos**, etc. *Está dirigido a desarrolladores de software.*
 - El diseño de bajo nivel normalmente 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. |

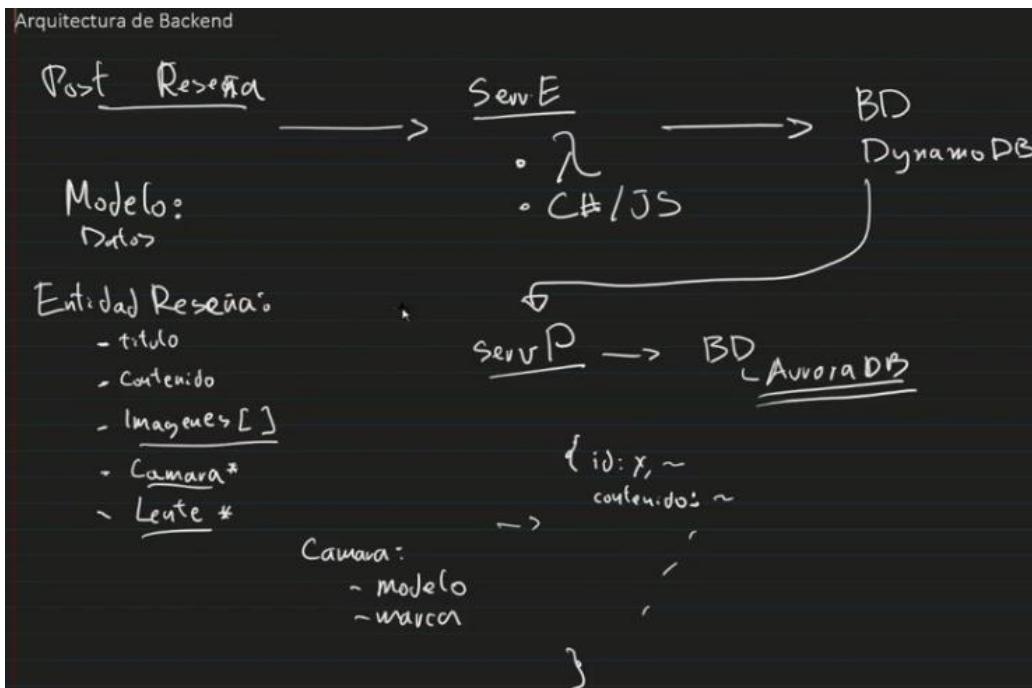
| Servicio 3 (S3) | | |
|--|---|--|
| Acción que realiza | Características del Servicio | Características de su database |
| Procesar los datos recibidos de la función Lambda S1 y pasarlo a una DB relacional de AWS. | <p>Servicio de procesamiento de datos de una base de datos NoSQL a una relacional:</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. |



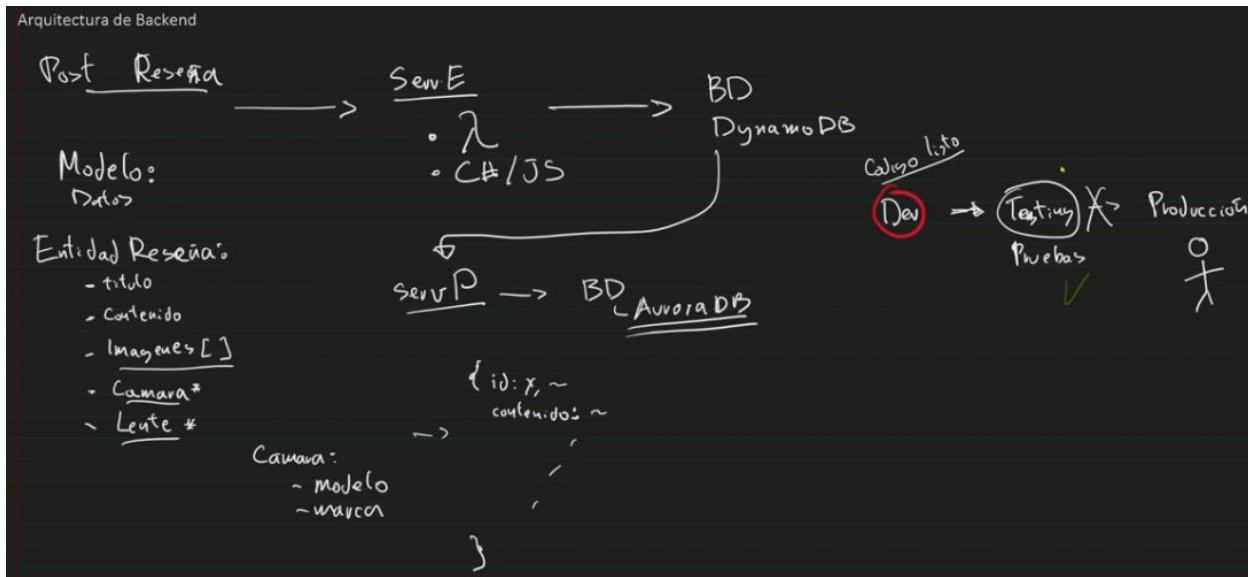
Representación de las Bases de Datos: Definiciones y Diagrama Entidad-Relación

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.*



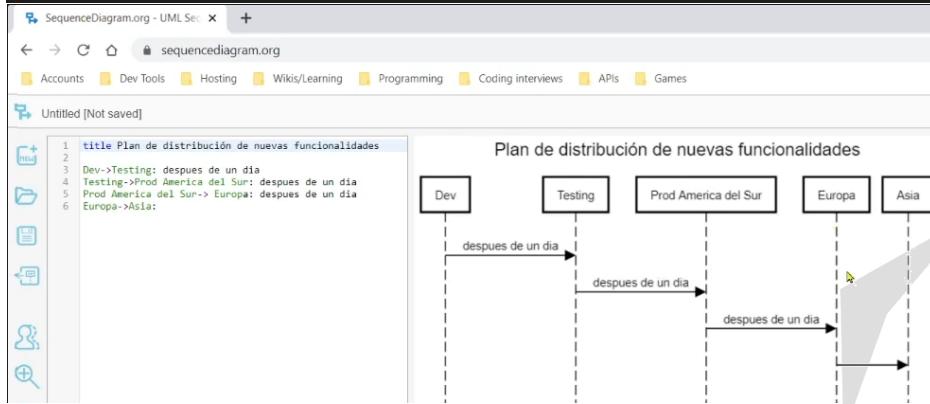
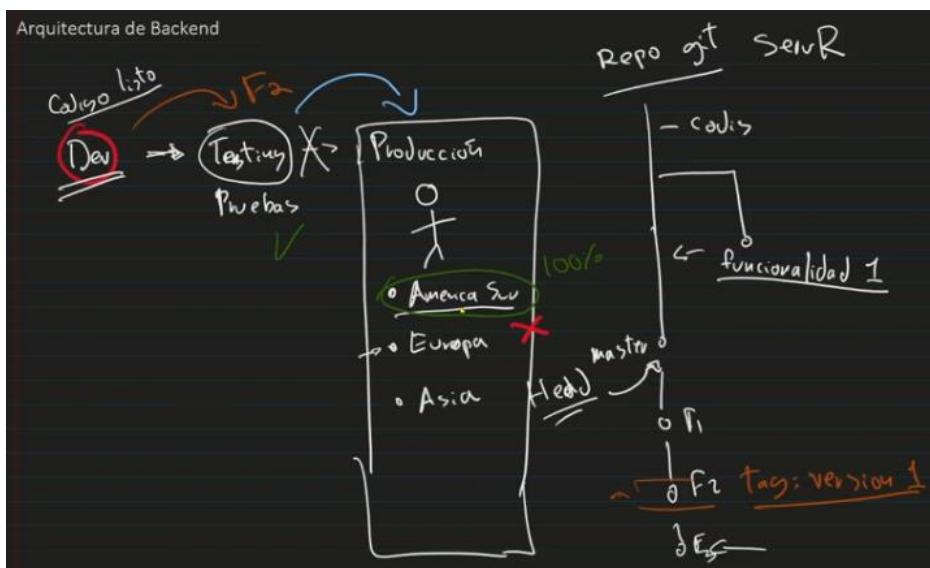
Plan de Integración Continua: CI/CD - Continuous Integration/Continuous Development

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.

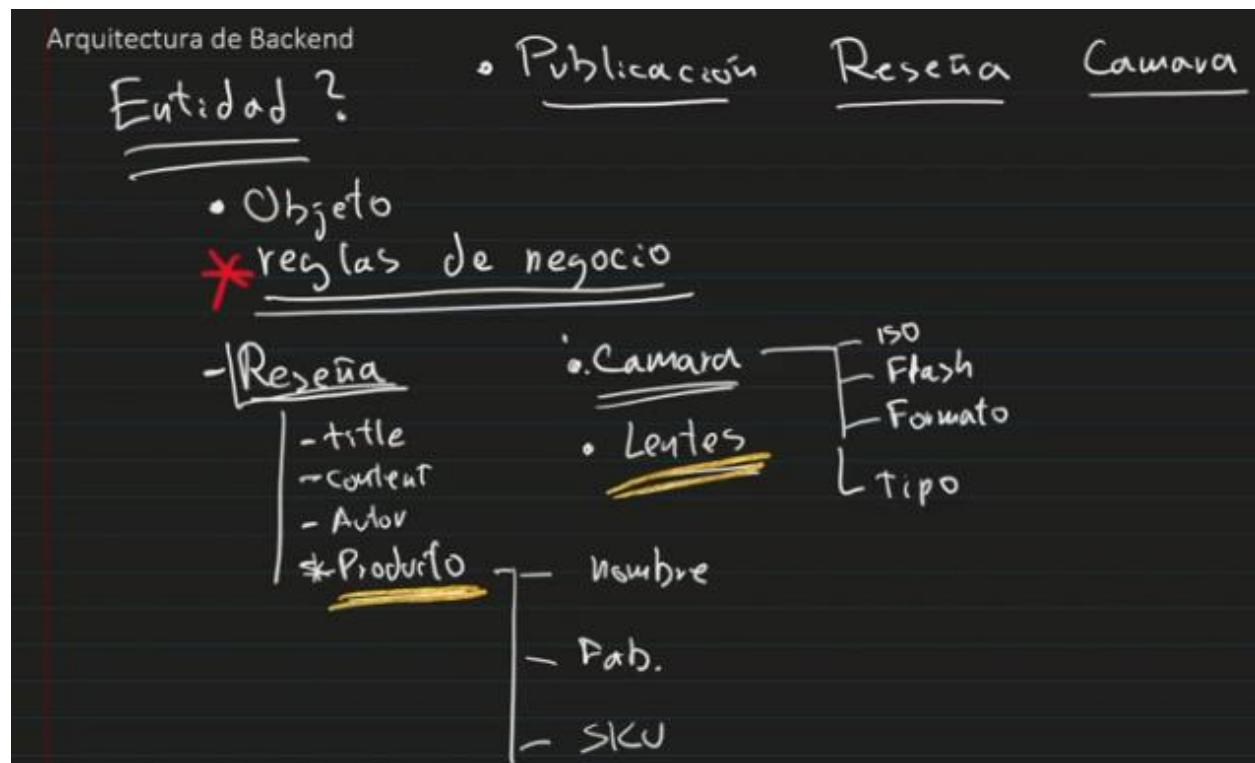
Nota: Para poder desarrollar un buen software, se debe entender el negocio.

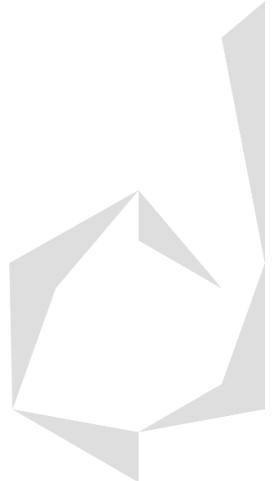
Desarrollo del Código del Proyecto

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.

