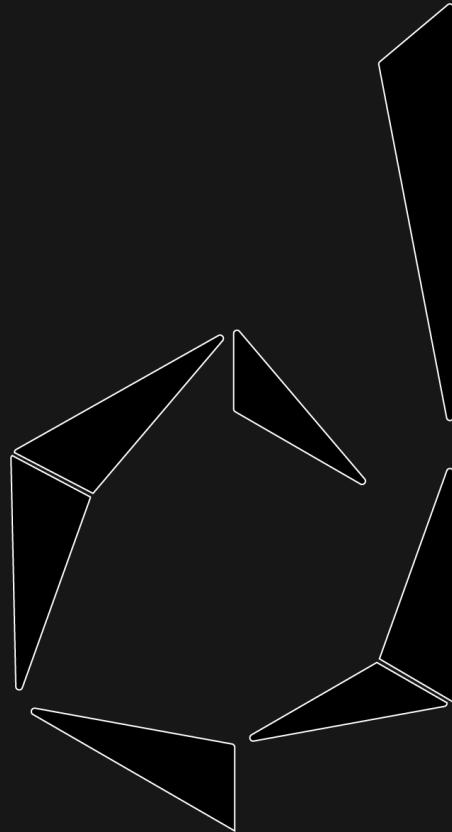


INGENIERÍA MECATRÓNICA



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:.....	17
Diseño de Bajo Nivel: Planes de Prueba TDD e Integración Continua CI/CD	20
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
Entidades, Interfaces, Clases y TDD (Test Driven Development)	27
Escalabilidad: Throttling y Retry Policy	30
Referencias.....	32



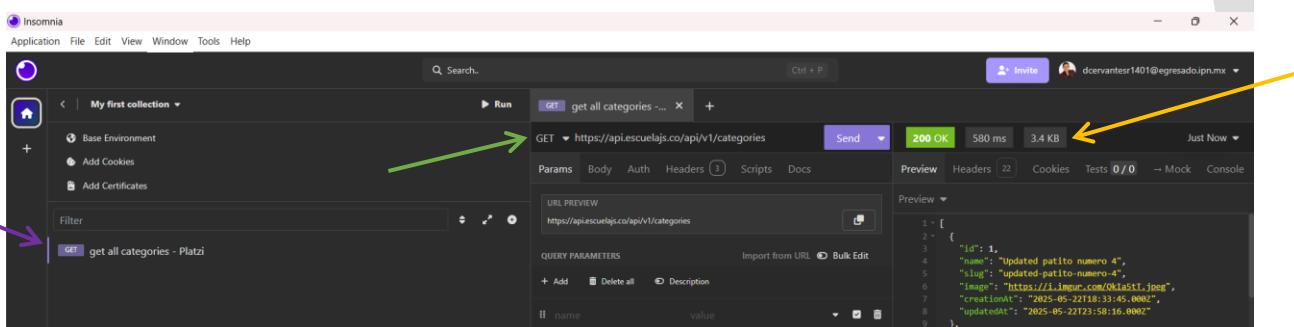
Conocimientos Previos

- **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:** Crea la **interfaz visual** que se renderiza a través de datos para que el usuario la vea e interactúe con ella, 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 incluye un método HTTP GET, POST, etc. y devuelve un código de estado (como 200, 404 o 500) que indica si la petición (request) fue exitosa, fallida o si hubo un error en el servidor.**

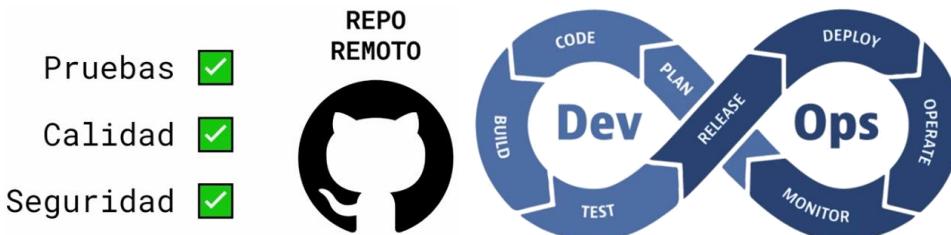
Rango de HTTP Status	Definición	Estados HTTP Más Comunes
100-199	Estos indican información de estado del servidor hacia el cliente .	100 - Continue: Solicitud recibida, continuando proceso. 101 - Switching Protocols: Inicio de cambio de protocolo. 102 - Processing: El servidor está procesando la petición.
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 en la parte del cliente , esto normalmente ocurre porque se 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	<p>Representan errores que ocurren en el código que está ejecutándose del lado del servidor o en el servidor en sí.</p>	<p>500 - Internal Server Error: Error interno del servidor.</p> <p>501 - Not Implemented: Método no soportado por el servidor.</p> <p>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).</p> <p>503 - Service Unavailable: El servidor no puede manejar la solicitud porque está temporalmente sobrecargado o en mantenimiento.</p> <p>504 - Gateway Timeout: El tiempo de espera del servidor se ha sobrepasado.</p>
----------------	---------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

- **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 sus **endpoints** funcionen correctamente. Facilitan la depuración de errores, la visualización de respuestas y la realización de pruebas básicas.



- **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, para subir nuestras aplicaciones (**deploy**) y que todos nuestros usuarios las puedan acceder, pudiendo **escalar su tamaño** según la demanda.
- **DevOps:** Es una metodología que une los equipos de desarrollo (Dev) y operaciones (Ops) para automatizar procesos de pruebas, integración continua, despliegue (**deploy**) y monitoreo. Busca reducir errores, acelerar entregas y mejorar la calidad del software.



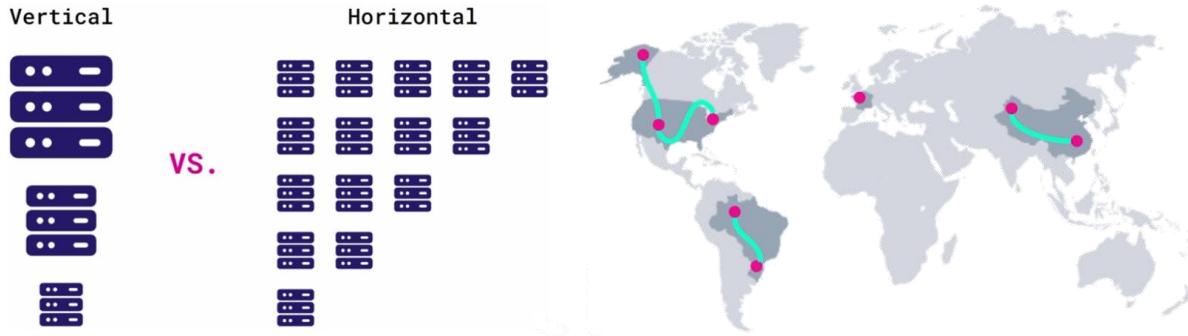
- **Modelos de Infraestructura Interna de Servidores:**

- **On Site:** La infraestructura (servidores, red, almacenamiento) se encuentra físicamente en las instalaciones del cliente y se instala todo desde cero.
- **IaaS (Infrastructure as a Service):** Es una infraestructura virtualizada, **teniendo un alto control** sobre elementos como **databases**, servidores, sistema operativo, tiempo de ejecución, etc. **Un ejemplo de este es Digital Ocean.**
- **PaaS (Platform as a Service):** Se provee una plataforma lista para desplegar código, sin preocuparse por la infraestructura, solo por la **aplicación (el código)** y **sus datos**. **Un ejemplo de este es Firebase de Google.**
- **SaaS (Software as a Service):** El usuario accede a un software ya listo para usarse, sin necesidad de desarrollarlo ni mantenerlo. **Un ejemplo es Google Drive.**

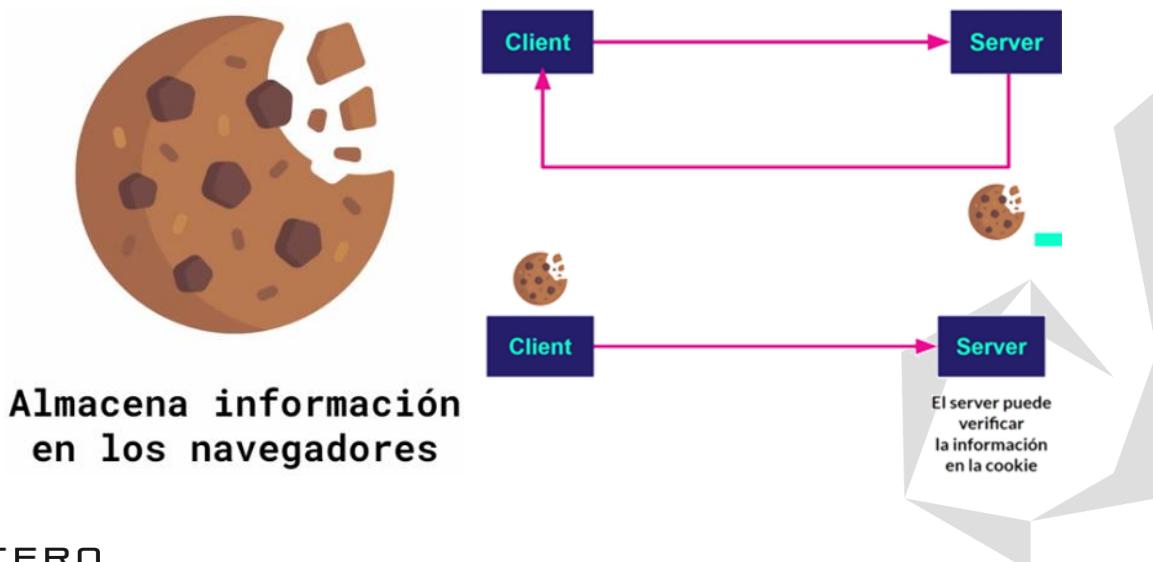


- **PaaS (Heroku):** **Plataforma como servicio** que permite **desplegar aplicaciones web de forma sencilla**. El desarrollador solo sube su código y **Heroku** se encarga del servidor, **database**, red y **escalabilidad**. Es ideal para crear proyectos de forma rápida o prototipar.

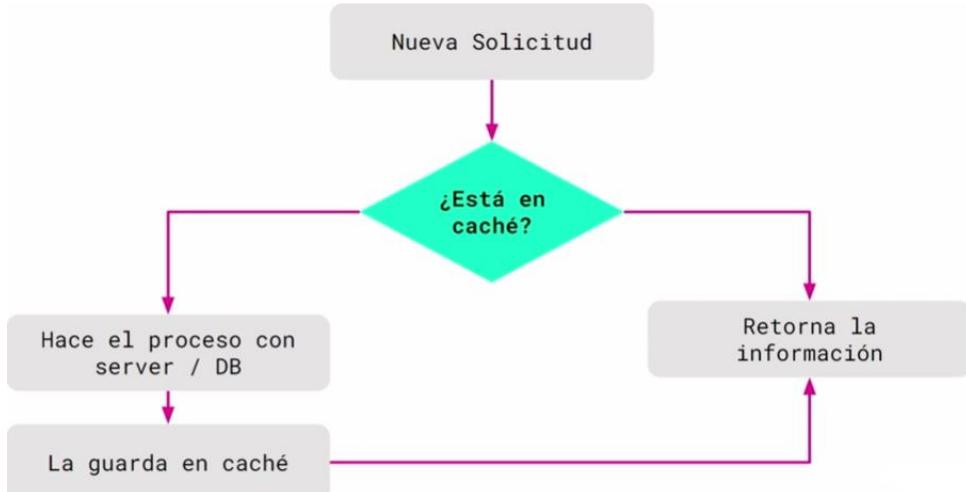
- **Escalamiento Vertical:** Consiste en **aumentar la capacidad de un solo servidor** (más RAM, CPU o **almacenamiento**) para **manejar una mayor carga de usuarios**. Es simple de implementar, pero causa problemas de disponibilidad, ya que, si ese servidor falla, todo el sistema se cae.
- **Escalamiento Horizontal:** Implica **añadir múltiples servidores o instancias, que trabajen en paralelo** (**cluster**). Esta técnica mejora la disponibilidad, demandas fluctuantes, tolerancia a fallos y capacidad de atender a muchos usuarios simultáneamente con varios servers de pocos recursos.



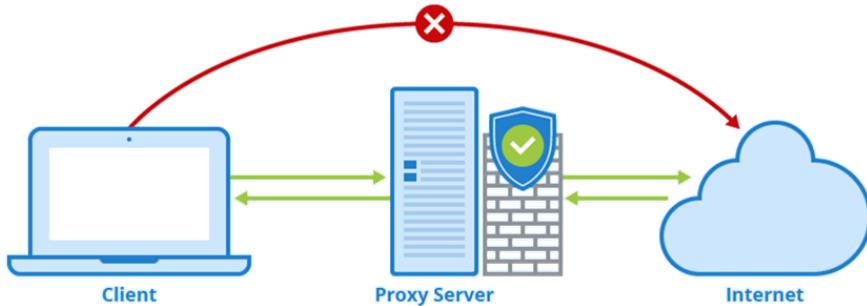
- **Bases de Datos:** Son sistemas organizados que **almacenan**, gestionan y consultan datos de forma permanente. En backend, se utilizan para guardar información de operación como usuarios, productos, reseñas, etc., y **permiten conectividad rápida mediante consultas SQL o NoSQL**.
 - **Replicación de Bases de Datos:** Técnica para **copiar y sincronizar datos entre múltiples databases**. Aumenta la disponibilidad y permite la distribución geográfica del sistema. En este tipo de arquitectura hay **DB que son de escritura** y se crean sus **réplicas de lectura**.
- **Colas de Tareas (Queues):** Son estructuras que almacenan tareas o procesos en espera, para ser ejecutados de forma asíncrona y ordenada. *Su principal diferencia con las APIs, es que estas primeras devuelven su resultado de forma rápida y por el mismo medio, mientras que las colas de tareas retornan su respuesta eventualmente y por otros medios.*
- **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 motores de búsqueda y SEO (Search Engine Optimization)*.
- **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 del lado del Servidor 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.

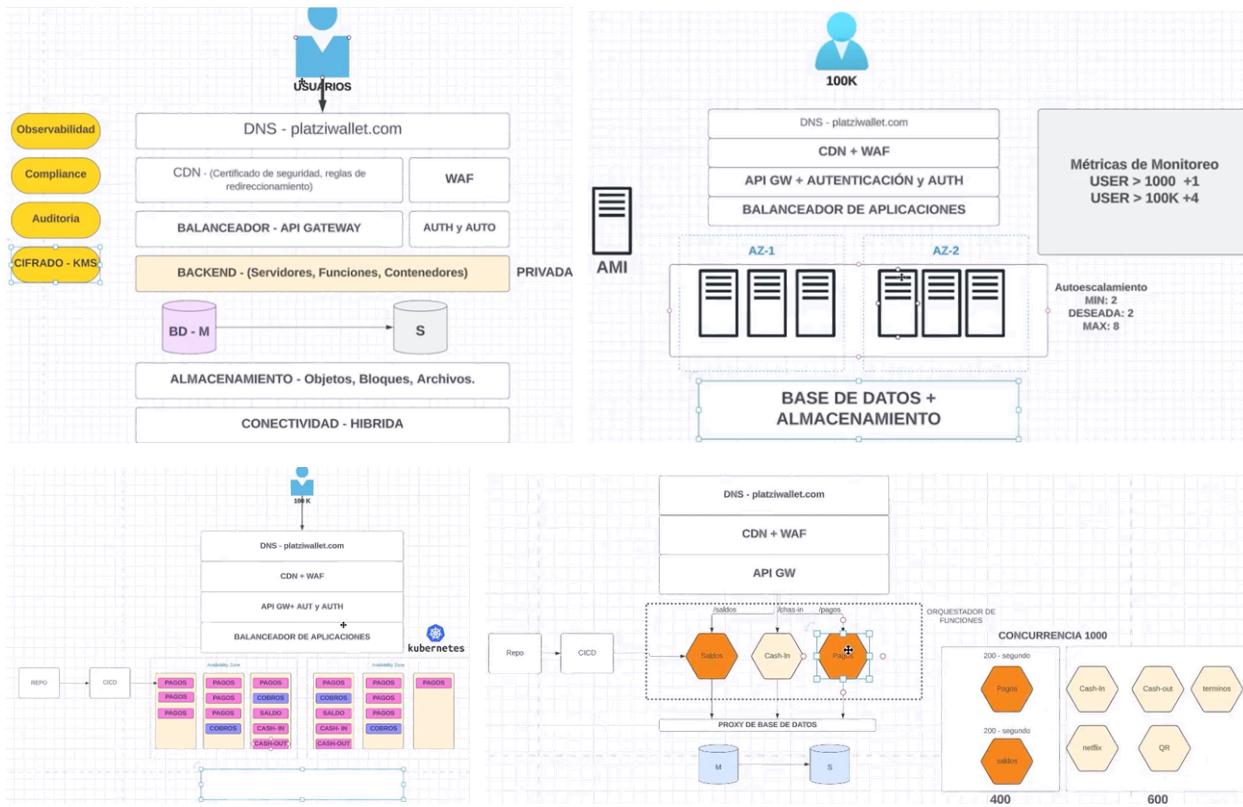


Diferencias entre las **Cookies, Memoria Caché y Proxy** son:

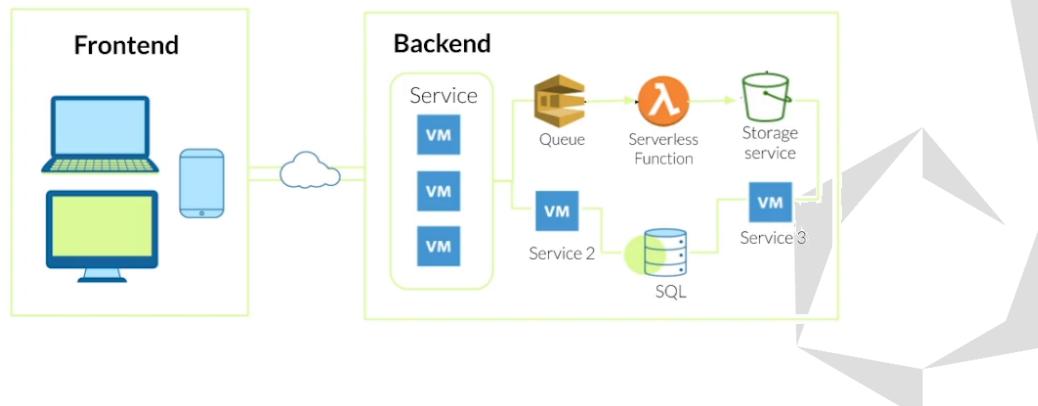
Característica	Cookie/Sesión	Memoria Caché	Memoria Proxy
¿Dónde almacena sus datos?	En el navegador.	En el servidor o navegador.	En un servidor intermediario proxy.
¿Qué almacena?	Resultados previos en el navegador.	Resultados previos en el servidor.	Copias de las respuestas HTTP/HTTPS.
¿A quién ayuda?	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 una URL temporal con autenticación.

- **Arquitectura y Despliegue:** Se refiere a cómo están organizados los componentes de un sistema (**bases de datos**, **servidores**, **servicios**, **etc.**) para estar disponibles 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 o sistemas operativos.
 - **Kubernetes:** Es una **plataforma de orquestación que gestiona múltiples contenedores Docker**, administrando su **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 (**lock-ins**) y mejora la tolerancia a fallos.

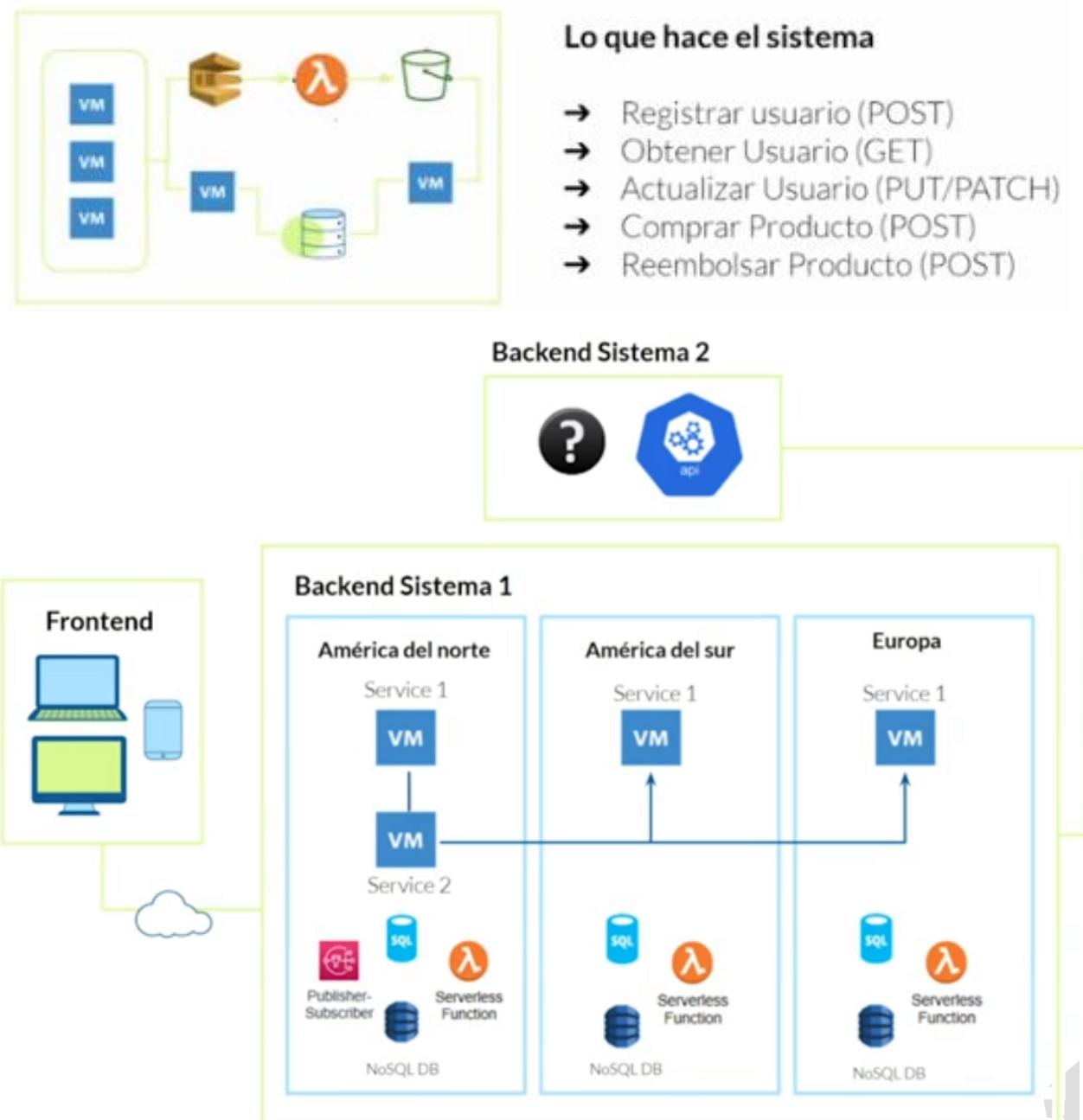


Nota: Vale la pena mencionar que todas las arquitecturas pasadas pueden ser mezcladas para obtener el mejor resultado en nuestras aplicaciones.



Ejemplo de arquitectura 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**, esto deberá realizarse tomando en cuenta la planificación de la arquitectura que se describirá en alto y bajo nivel. Para ello, los requerimientos que nos ha dado un cliente ficticio son los siguientes:

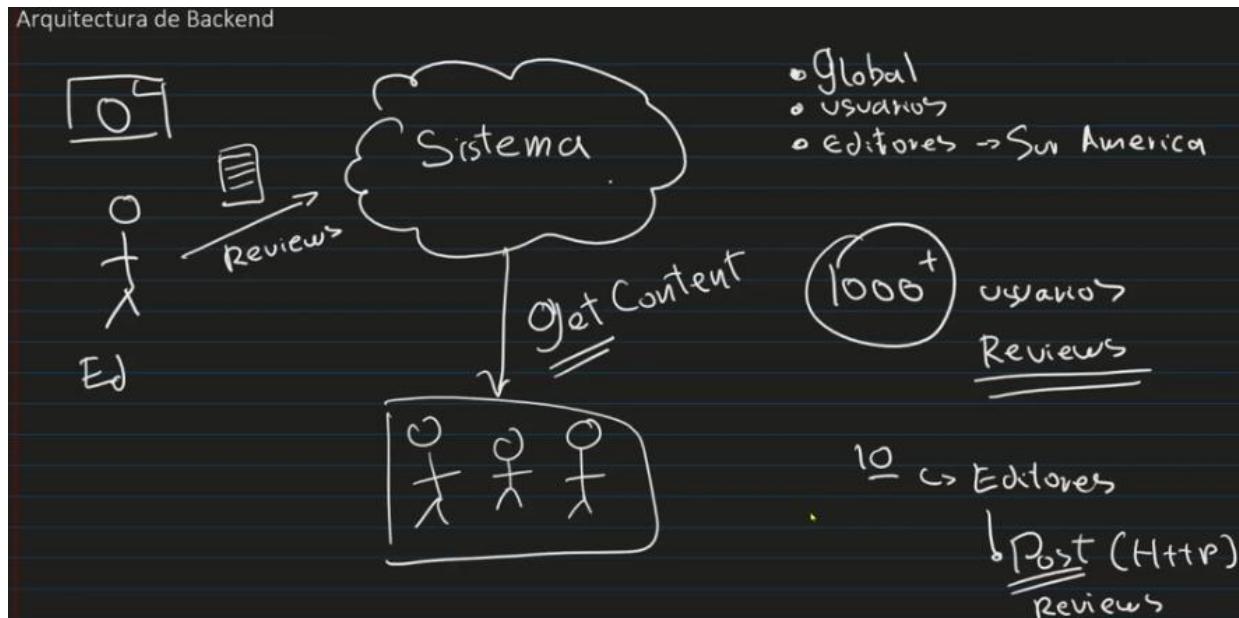
- La empresa “RandomCameraReviews” necesita **un sistema que permita que fotógrafos profesionales suban “reviews” de cámaras fotográficas**, para que **cualquier persona en todo el mundo pueda buscar los reviews y compararlas a través de su portal**. *La empresa cuenta con un equipo de developers especializados en frontend que realizará una interfaz de usuario para que los editores suban sus “reviews” y los usuarios puedan verlas*, por eso nos han solicitado que creamos 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 de su portal 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 **servicio backend** donde el **editor (Ed)** va a enviar reviews al **sistema en la nube**.
 - Método HTTP **POST** al endpoint `/reviews`.
- 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.
 - Método HTTP **GET** al endpoint `/content` sin realizar autenticación/autorización de users.
- Las características del **cluster** para ser **distribuido** geográficamente de forma correcta son:
 - **Usuarios lectores (GET)**: Con presencia global en **América del Sur, Norte América, Europa y muy poco en Asia**. Esto puede **crecer**, por lo que se necesita tener una gran **capacidad de disponibilidad** en lectura de **datos**.
 - **Disponibilidad de la base de datos de lectura (réplica)** en **América del Sur, Norte América, Europa y muy poco en Asia**, con **arquitectura horizontal**.
 - **Los editores (POST)**: Se encuentran solamente en **Sudamérica**.
 - **Disponibilidad de la base de datos de escritura** solo en **América del Sur**, con **arquitectura vertical**.

- 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 cuáles son los **requerimientos de negocio para nuestro sistema backend**, vamos a tener que plasmar eso en algo llamado **documento de diseño de alto nivel**, el cual definirá todos sus **detalles necesarios de forma agnóstica** para así poder desarrollar su arquitectura sin importar en qué proveedor de nube esté alojada o si está alojada en varias (**servidor multinube**). 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:

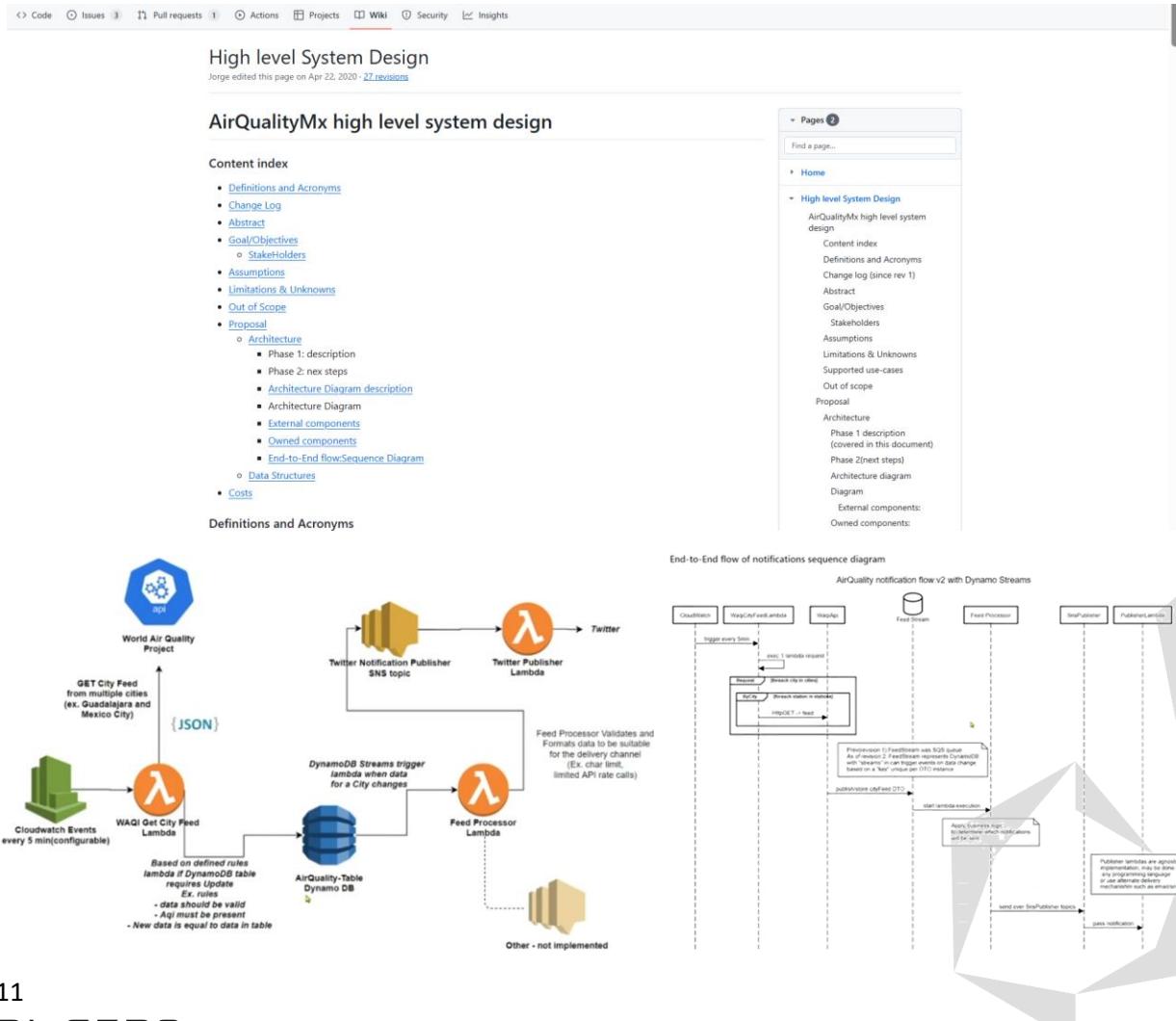
- **Definiciones y acrónimos:** Aquí se describe conceptos clave que se vayan a mencionar en el documento, para que cualquier persona de producto (no desarrollador de software) pueda entender a qué se refiere cada cosa.
- **Problema Por Resolver (Abstract/Overview):** Es una descripción general del proyecto.
- **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 programador asume con respecto al desarrollo del proyecto, que debe de ser comunicado hacia el cliente.
- **Limitaciones y Desconocimientos:** Cuestiones limitantes en infraestructura o conocimiento hacia el **escalamiento** del proyecto o preocupaciones del desarrollador que son comunicadas en esta sección al 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.
- **Proposal:** Propuesta de arquitectura, componentes internos y externos, estructura de datos y diagramas de secuencias o funcionamiento.
 - **Arquitectura General:** Descripción de elementos potenciales que conformen el frontend, backend, **bases de datos**, autenticación, despliegue en la nube y distribución global.
 - **Endpoints de la API y Componentes del Sistema:** Descripción general de los endpoints que se utilizarán y los servicios que efectuarán su uso.
- **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.



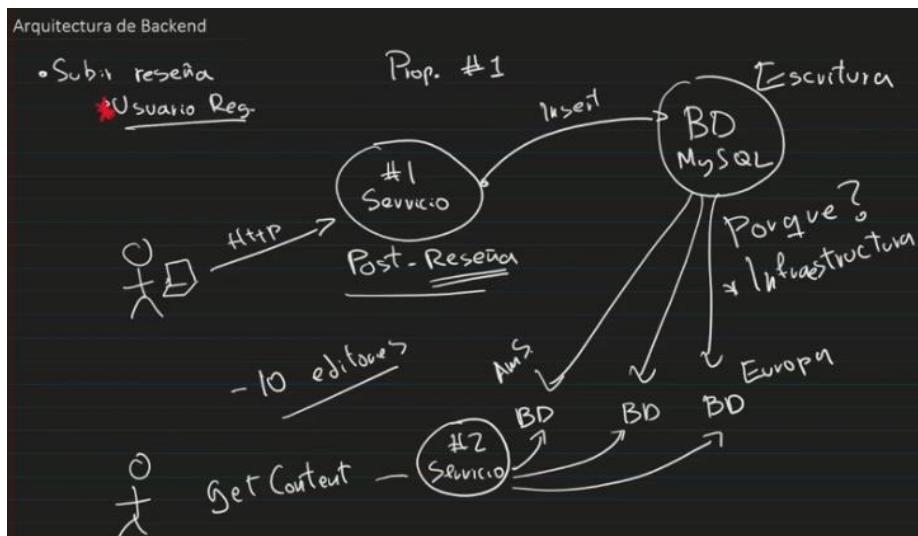
Ya sabiendo los componentes que debemos incluir, procederemos a mostrar el documento de diseño de este ejemplo de proyecto backend, cabe mencionar que **este se realiza a través de varias iteraciones entre el equipo de desarrollo y el cliente, poniendo muy claros sus alcances y objetivos**. Para ello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

- **Propuesta 1:**

- **Servicio 1 - Subir Reseñas:** Para este servicio 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 del servicio 1 o una no relacional**, sabiendo **sus beneficios y desventajas**, *el feedback que se discute en equipo debe resolver la pregunta de ¿por qué se están tomando ciertas decisiones?*
 - **Infraestructura:** *La respuesta del porque se toman decisiones debe estar basada en la infraestructura que adoptará el proyecto cuando crezca*, si se adoptará un **escalamiento vertical u horizontal** cuando esto pase y **qué proveedor cloud desplegará (deploy) al servicio, considerando costos y requests que vaya a recibir cada endpoint de nuestro servicio**.
 - 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, debido a que no se espera una pronta subida de usuarios**. Por lo tanto, podemos cumplir los requerimientos de negocio de este **servicio 1** a través de **una sola DB monolítica y relacional que sea simplemente de escritura**, para luego hacer una **réplica en otra que sí sea de lectura**.
- **Servicio 2 - Lectura de Reseñas:** Como en el caso anterior se decidió que la **database del servicio 1 fuera solo de escritura y que tenga replicación en otras de lectura**, esta parte de la **base de datos del servicio 2 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 DB debe estar **escalado horizontalmente**, proveyendo de disponibilidad en América, Europa y Asia, por lo que se necesitan 3 servidores.

- El ¿por qué? de esta decisión 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 de alto nivel (gente de negocios, no desarrolladores)** 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 (Abstract/Overview)

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 lean los reviews).

También se menciona que la empresa “RandomCameraReviews” planea distribuir mayormente en America 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

- 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 Incluido del Proyecto (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.

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
Sí	POST /reviews	Subir una nueva reseña	<input checked="" type="checkbox"/>
No	GET /content	Obtener todas las reseñas disponibles	<input type="checkbox"/>

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.
-

💰 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.



01.1_Documento de Diseño Alto Nivel.md M

Diseño de Alto Nivel

Definiciones y Acrónimos

- API:** Interfaz de Programación
- TDD:** Desarrollo Guiado por Pruebas
- HTTP:** Protocolo de Transferencia de Hypertext
- CRUD:** Crear, Leer, Actualizar, Eliminar
- POST:** Método HTTP para enviar datos
- GET:** Método HTTP para obtener datos
- Editor (Ed):** Usuario autorizado
- Usuario (Lector):** Visitante que lee
- Review (Reseña):** Contenido elaborado por un editor

Problema a Resolver (Análisis)

La meta es construir y desarrollar la aplicación en un alto nivel. Para ello, los requerimientos son:

- La empresa "RandomCameraReviews" quiere permitir a los usuarios subir y comparar reviews de cámaras fotográficas y las empresas que las crean.
- Los usuarios deben poder subir reviews de cámaras, obtener el contenido de las reviews y manejar los usuarios para su administración.

También se menciona que la empresa tiene ventas en Sudamérica, donde está su mercado más grande, pero también tienen ventas en Norte América, Europa y Asia.

En la parte superior derecha de la pantalla se muestra una barra de menú con las siguientes opciones:

- Close
- Close Others
- Close to the Right
- Close Saved
- Close All
- Copy Path
- Copy Relative Path
- Open Preview**
- Reopen Editor With...
- Share
- Reveal in File Explorer
- Reveal in Explorer View
- Keep Open
- Pin
- Split Up
- Split Down
- Split Left
- Split Right
- Split in Group
- Move into New Window
- Copy into New Window
- Add File to Chat
- Find File References

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

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

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

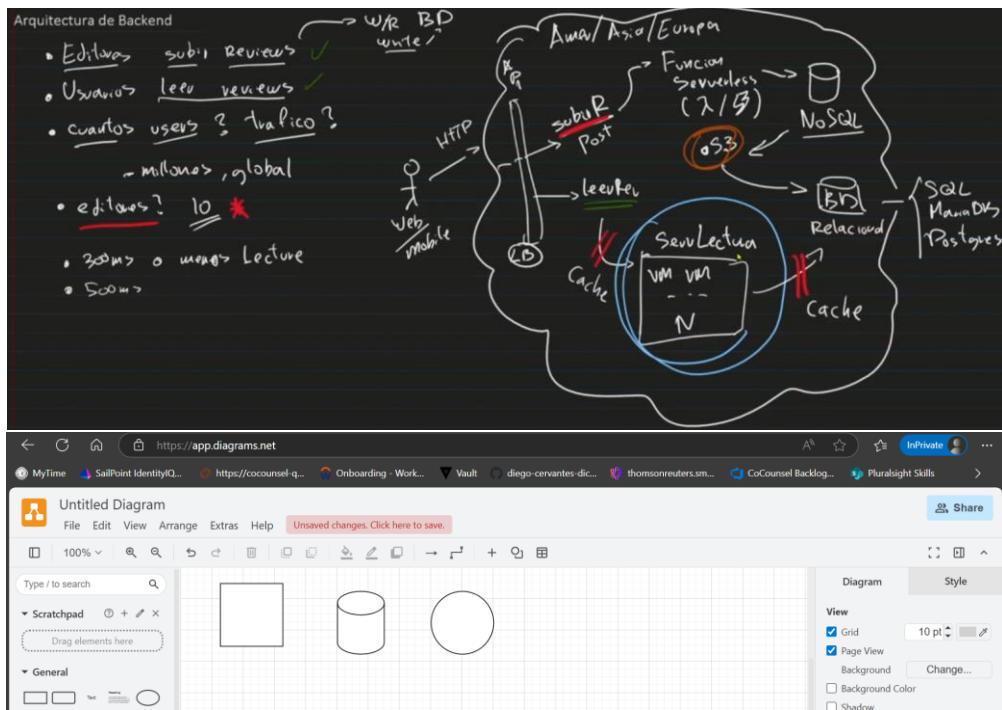
- Arquitectura Backend:** Se diseñará un programa que permita que los **editores puedan subir sus reviews**, el sistema además debe permitir tener **otro tipo de usuario que no sea editor pueda leer el contenido de dichas reviews**. Para ello contestaremos las siguientes preguntas.
 - Tráfico:** Debemos 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.
 - Escalamiento de los usuarios en el sistema:** Se sabe que en este caso no se tendrá el mismo número de **reviewers** que de **lectores**, ya que **se tendrán pocos editores** que se encontrarán principalmente en Sudamérica y **muchos lectores** en América, Europa y Asia.
 - Tiempo de respuesta:** Este es el **tiempo en milisegundos en el cual se quiere retornar la respuesta hacia el usuario**.

- **Latencia:** *300ms, 100ms, etc.* son **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.
- **Replicación:** 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 Sudamérica**, 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 **reproducible**, no solo **extensible**.
- 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 el 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.**
 - **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 **de forma distribuida** 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.
 - **Servicio 1 (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 o **el número de servidores escalados de forma horizontal**, 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 que se mantenga fijo y dejando los demás recursos para las funciones restantes en el mismo servidor.
 - **Servicio 2 (S2) - Backend para leer reviews 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 con el fin de 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 el auto escalamiento decide cuándo debe aumentar su número de servidores** 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 ocurre porque el **Load Balancer** primero determina que se haya montado correctamente el servidor, antes de que lo podamos utilizar.
- **DB Servicio 1 (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 **database no relacional**, la opción más conveniente sería una memoria **caché (cliente)**.
 - **DB 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 **database no relacional** para manejar los **datos del servicio 1 (S1)**.
- **DB Servicio 2 (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.
- **Servicio 3 (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.

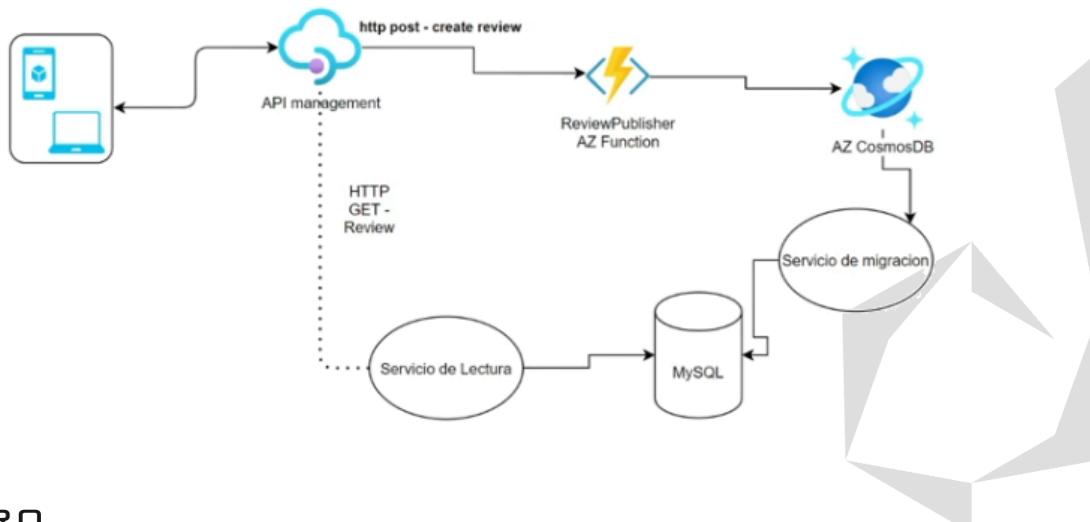


Diseño de Bajo Nivel: Planes de Prueba TDD e Integración Continua CI/CD

El diseño de bajo nivel es más específico y contempla elementos de **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 general **sistema distribuido**. *Está dirigido a gente de negocio que no tiene tanto conocimiento con temas de desarrollo de software.*

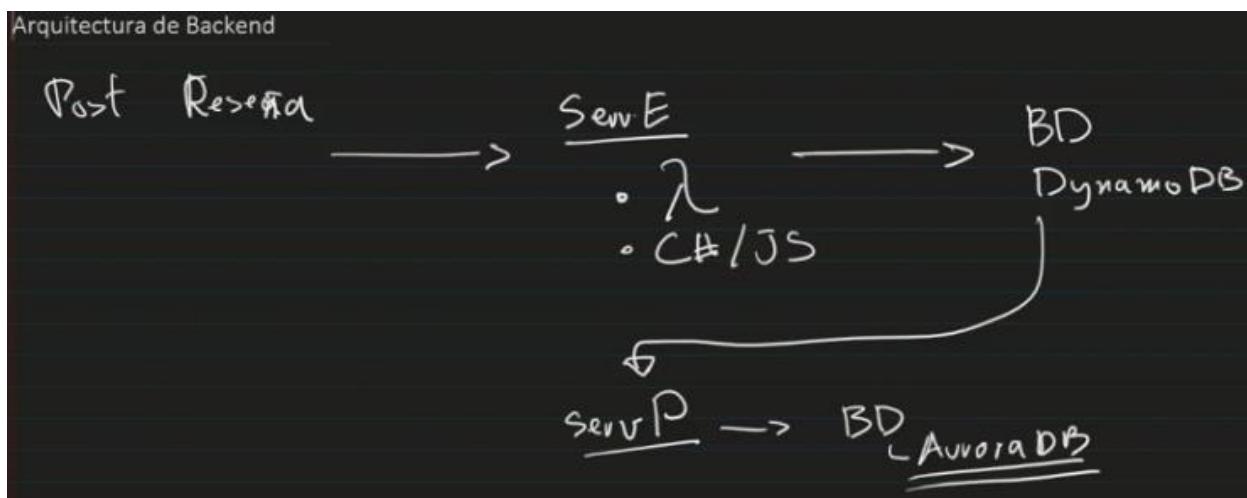


- **Diseño de Bajo Nivel:** Diagrama de la **arquitectura distribuida** que entra más en detalle, abordando temas como **lenguajes de programación elegidos**, frameworks, **motores de bases de datos**, **modelo de datos**, etc. *Está dirigido a desarrolladores de software.*
 - El **diseño de bajo nivel** 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 **DBs distintas con motores diferentes** para cada servicio o tareas de **lectura/escritura (R/W)** de datos.
- El **documento de bajo nivel** básicamente tiene la misma estructura que el de alto nivel, pero añadiendo las siguientes secciones en la parte donde se describe la arquitectura más a detalle:
 - **Casos de Uso:** Son **ejemplos de cómo se podría utilizar el sistema**, esto después se utilizará para realizar las pruebas en la fase de TDD (Test Driven Development).
 - **Diagramas y Modelos de Datos:** Son representaciones visuales y de código donde se denota el flujo de información por medio de diagramas de secuencia y UML (Unified Modeling Language) para definir y mostrar la función de cada **módulo distribuido** en el sistema y **su interconexión con los demás módulos en la red**.
 - Para la conceptualización de los diagramas anteriores es de gran utilidad crear una **tabla** donde se analicen las características de cada servicio o **DB** del sistema.

Servicio 1 (S1) - Servicio de Escritura		
Acción que realiza	Características del Servicio	Características de su database
Método HTTP POST en un endpoint llamado <code>/reviews</code>	Servicio de escritura W o E: <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. 	La base de datos es NoSQL : <ul style="list-style-type: none"> • El servicio está montado en AWS, por lo tanto, la base de datos será DynamoDB.

Servicio 3 (S3) - Servicio de Migración		
Acción que realiza	Características del Servicio	Características de su database
El servicio 3 procesa los datos recibidos de la función Lambda S1 y los pasa a una DB relacional de AWS, para que luego la función Lambda S2 proporcione esos datos al usuario.	Servicio de procesamiento de datos de una base de datos NoSQL a una relacional: <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. 	<ul style="list-style-type: none"> • Su base de datos de entrada NoSQL es DynamoDB. • Su database de salida relacional es AuroraDB.

Servicio 2 (S2) - Servicio de Lectura		
Acción que realiza	Características del Servicio	Características de su database
Método HTTP GET en un endpoint llamado /content	Servicio de lectura R o L: <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. 	La base de datos es relacional : <ul style="list-style-type: none"> El servicio está montado en AWS, por lo tanto, la base de datos será AuroraDB.



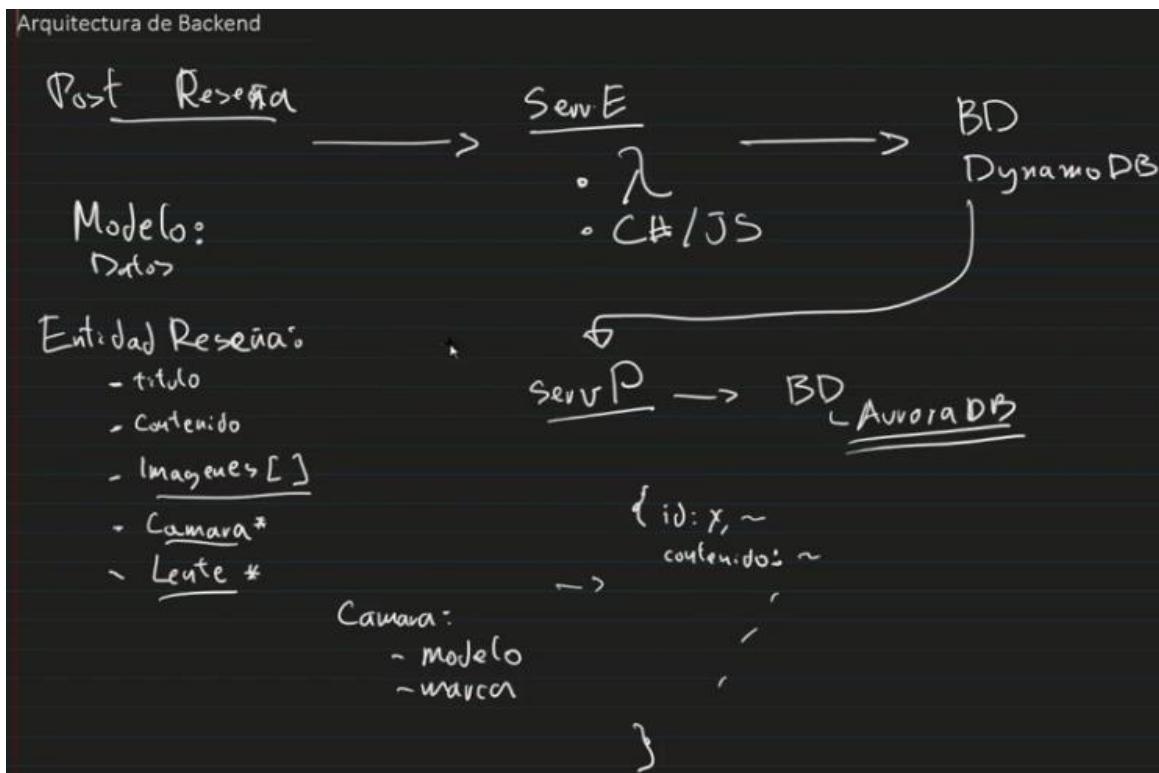
Representación de las Bases de Datos: Definiciones y Diagrama Entidad-Relación

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**.

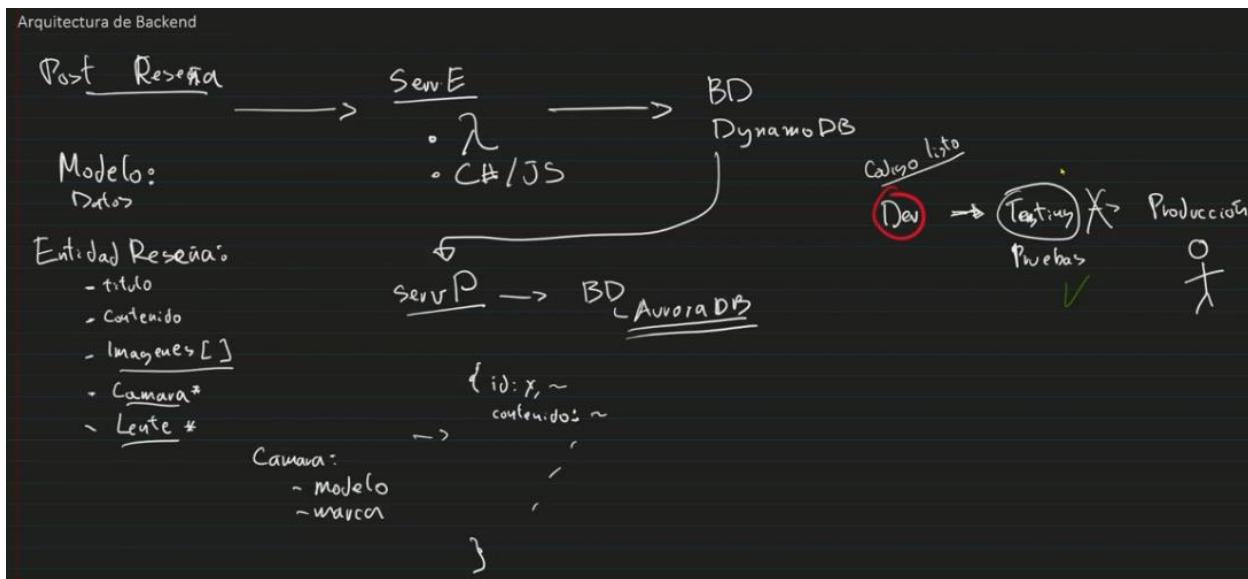
- **Modelo de datos:** Aquí se describen las características específicas de las **tablas en las DB**, diseñándolas desde cero para proporcionar el **almacenamiento de datos** del sistema en cada **servicio (S1, S2 y S3)**, pensando en el **método HTTP** que soporta cada uno, ya que, dependiendo del método, este dará, recibirá, modificará o borrará datos.
 - **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**.

En 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 **columnas** que sean otras **tablas por sí mismas**, como **cámara y lente de cámara**:

- **Cámara:** Esta es una entidad que proviene de la tabla **reseña**, la cual puede tener atributos propios 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)** se compone de un **método HTTP POST**, debemos pensar en los datos que recibe en su header para efectuar su función.



- **Plan de Pruebas (TDD - Test Driven Development):** Además de los diagramas de arquitectura y los **modelos de datos**, vale la pena pensar en un **plan de pruebas** 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 que todo funciona bien antes de hacer un deploy en la nube.
- **Integración continua (CI/CD - Continuous Integration/Continuous Development):** Esta sección contiene **diagramas de pipelines**, describiendo con ellos el proceso de **implementación de features y arreglo de bugs** cuando estos se integren al sistema en producción, realizándoles pruebas cuando se quiera subir dichos cambios a **GitHub**. Gestionando además el flujo de integración (merge) de sus ramas y asignando cada una al proceso de desarrollo:
 - **Dev branch → Testing branch; feature: Adición de nuevas funcionalidades y fix: Arreglo de bugs → 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.



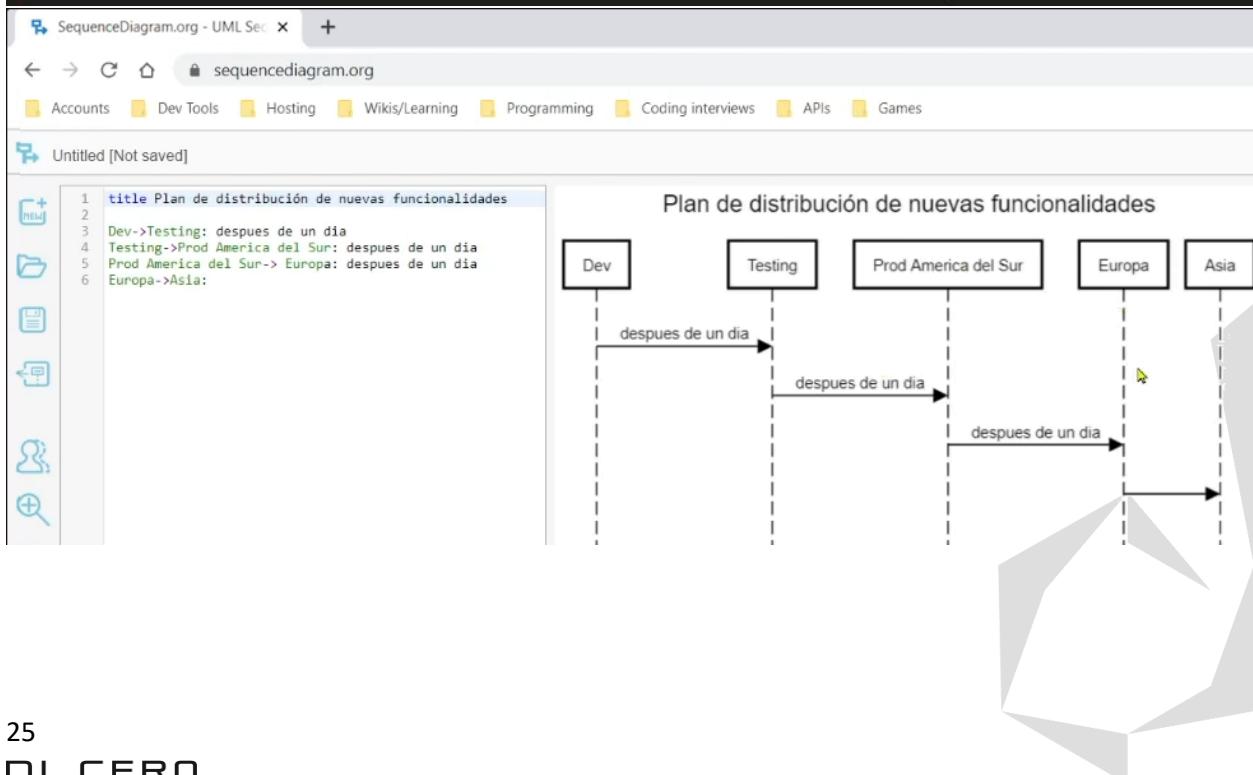
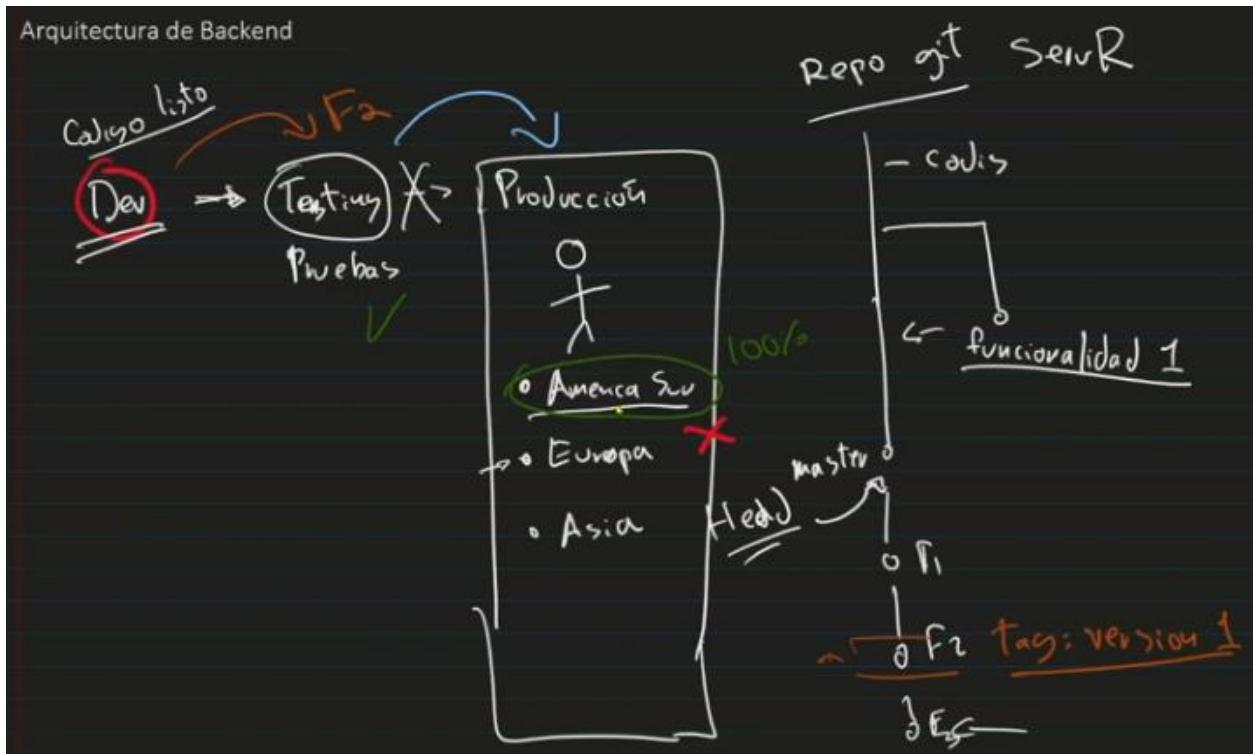
Plan de Integración Continua: CI/CD - Continous Integration/Continuous Development

Esto se refiere a **una sección en el documento de Diseño de arquitectura de software de bajo nivel** que engloba el concepto de **rollout**, describiendo así el cómo pasar una funcionalidad desarrollada en código a través de algo llamado **pipeline**, que es **una tubería de pasos a seguir para que el código que escribimos funcione al 100%, pasándolo a través de ciertas pruebas para comprobar su calidad**, con el fin de asegurarnos 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 (CI/CD)** 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**, que representan cada funcionalidad o bug que se quiere abordar.
 - Luego necesitamos levantar algo llamado **Pull Request (PR)** que *mostrará el código que fue cambiado, borrado o agregado, para luego pedir que otro desarrollador la revise y apruebe (review)*; cuando este cambio sea aprobado, se realizarán las **pruebas de nuestro pipeline**, para posteriormente **hacer un merge de dicha rama (fusión) con la main branch**, integrando así nuestro nuevo cambio al código principal del proyecto.
 - Cabe mencionar que **no forzosamente lo que esté en la rama main será lo que verá el cliente, osea la versión publicada del proyecto**, esto dependerá de cual rama sea publicada en un deploy hacia la nube de AWS, GCP, Azure, etc.
- **Pruebas (Testing):** Cuando una **rama de feature o bug quiera ser fusionada con la rama main**, debe existir una capa intermedia de **testing**, donde además del review que el programador dio a los cambios de código en la PR, se ejecuten pruebas automatizadas por scripts específicos para comprobar la funcionalidad del código, proporcionando **otra capa de monitoreo de calidad**. **Esto de igual forma se puede realizar a través de tags, que son versiones o checkpoints definidos de nuestro proyecto en GitHub**. Jenkins es de las herramientas de pipeline testing más utilizadas.

- **RollOut:** Cuando un cambio de código haya pasado las **pruebas de calidad de código del pipeline**, se puede añadir una herramienta que realice el **deploy automático** de este cambio a cada región donde esté **distribuido de forma horizontal** el código, ya sea América, Europa, Asia, etc. haciéndolo 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 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 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 probar con test unitarias todos los **casos de uso** mencionados en el Documento de Diseño. Ya que se haya terminado de ejecutar los **casos de uso** descritos en el proyecto, se puede utilizar herramientas de **asignación de tickets DevOps** como Trello, Azure Dev Ops (ADO), etc. para agregar **nuevas features o arreglar bugs**, los tickets se apoyan de **Pull Requests de GitHub**, las cuales deben haber sido levantadas, revisadas, aprobadas y fusionadas (merge) a la rama de producción (main) para que la tarea del ticket se pueda considerar como **code complete**.

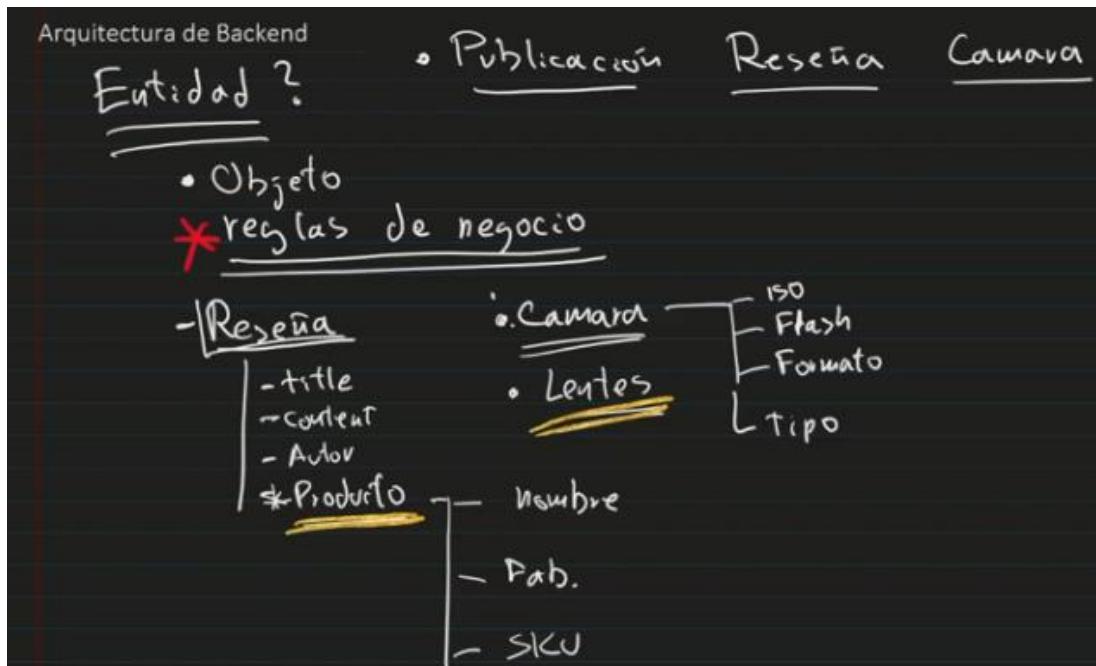
Nota: Para poder desarrollar un buen software, se debe entender el negocio.

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 DevOps 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 crear un diagrama de **entidad relación** con las **tablas de la DB** que soporten las **reglas de negocio** del **caso de uso** publicar una reseña (**HTTP POST**) en el sitio “RandomCameraReviews”.



En la siguiente parte vamos a pasar de lleno al código definiendo las respectivas clases e interfaces de las **entidades DB** para posteriormente crear una prueba y comprobar la funcionalidad del servicio.

Entidades, Interfaces, Clases y TDD (Test Driven Development)

A continuación se definirán los conceptos de interfaz, clase y los tipos de pruebas que se le pueden aplicar a estos para comprobar su funcionamiento.

- **Clase:** En programación orientada a objetos, una clase es una **plantilla o modelo** que define las **propiedades (atributos) y comportamientos (métodos)** que tendrán los **objetos** creados a partir de ella. Es la base para instanciar objetos concretos que comparten la misma estructura y funcionalidades.
 - **Objeto:** Es la instancia de una clase, osea un ejemplo específico que sigue las instrucciones dadas por el modelo que es definido dentro de una clase.
- **Interfaz:** Es una especificación que **define un conjunto de métodos y propiedades que varias clases deben implementar, pero sin contener su lógica interna**. Sirve como un contrato que asegura que las clases que la usen cumplan con un determinado comportamiento, permitiendo estandarización y flexibilidad en el diseño.

Ya habiendo definido las clases e interfaces en un código, vale la pena indicar cuales son los dos tipos de pruebas principales que existen:

- **Test unitarios:** Son tests que prueban una función de manera individual o atómica. Un ejemplo de este tipo de test es cuando creamos una función que suma únicamente números "naturales", un solo test puede ser sumar dos números "naturales", otro test diferente, puede ser que la función lance un error si se le da un número que no está dentro de la definición de numero "natural". Cada uno de estos dos sería una prueba atómica diferente, unitaria, y prueban una sola cosa. Cabe mencionar que las pruebas unitarias son fáciles de involucrar en nuestro proceso de pipeline (CI/CD) y son más rápidos de escribir, entender y también de ejecutar.
- **Tests End to End:** Al contrario de un test unitario, las pruebas "End to End", no prueban una sola cosa, sino la funcionalidad de un sistema completo de inicio a fin. Por ejemplo, si tenemos una aplicación que se asemeja a una red social popular, un único test puede involucrar pasos como los siguientes: *Desde la UI en un navegador Web.*
 1. *Abrir la página principal → Registrar una cuenta (y toda la validación que eso conlleva) → hacer login → mandar un post → añadir contactos → hacer login como los contactos → validar que se puede ver el post publicado anteriormente → cambiar la foto de perfil → validar que todos lo puedan ver → hacer un post → quitar el post → validar que en menos de X segundos los contactos que originalmente podían ver el post ya no pueden verlo, etc.*

Por lo tanto, las pruebas "End to End" son muy difíciles de escribir, ya que se debe de considerar todos los casos de uso y automatizarlos.

Las entidades, interfaces, clases y pruebas del código Python son las siguientes:

- **Entidad Producto:**
 - **Interfaces:** Atributos y métodos que debe implementar una serie de clases.
 - **IProduct:** Es la **interfaz que define dos tipos de productos, las cámaras ICamera y sus lentes ILens**. Se define como interfaz porque, aunque cámara y lente sean dos **clases** distintas, pueden tener características en común. Todos los tipos de

productos que sean creados a través de la **interfaz IProduct** comparten los siguientes atributos o métodos:

- **Atributos:** Name, Manufacturer, SKU (Stock Keeping Unit) y **Array Feature** (que es una lista de características que se pueden agregar posteriormente a un producto, las cuales serán descritas por una clase para dar flexibilidad a la interfaz).
- **Métodos:** Cada atributo debe implementar sus métodos **getter()** y **setter()**, los cuales se encargan de extraer e insertar los valores específicos a cada uno. Además, se incluye un método llamado **GetContent()**, el cual debe devolver todas las características de un producto para que estas puedan ser visualizadas por las reseñas de los productos.
 - **ICamera:** Es la interfaz que define las clases (entidades) tipo cámara, la cual debe **heredar de la interfaz IProduct**, ya que es un **producto perteneciente a los reviews de fotografías**, pero de igual forma todos los tipos de cámaras que sean creadas a través de la **interfaz ICamera** deben compartir los siguientes atributos o métodos:
 - **Atributos:** maxISO (sensibilidad de la cámara a la luz), Type (tipo de cámara) y CropFactor (campo de visión).
 - **Métodos:** Cada atributo debe implementar su **getter()** y **setter()**, los cuales se encargan de extraer e insertar los valores específicos a cada uno.
 - **ILens:** Es la interfaz que define las clases (entidades) tipo lente, la cual **hereda de la interfaz IProduct** porque es un **producto perteneciente a los reviews de fotografías**.
- **Clases (Entidades de la base de datos):** Atributos y métodos que instanciarán objetos.
 - **Feature:** Es una clase de tipo array o lista que puede agregar una serie de características o funcionalidades extra a la interfaz **IProduct**, sirve para dar flexibilidad de diseño con el fin de poder añadir atributos o métodos que no hayan sido pensados desde un inicio.
 - **Atributos:** El feature de un producto puede tener atributos como Name y Description.
 - **Métodos:** Cada atributo debe implementar su **getter()** y **setter()**, los cuales se encargan de extraer e insertar los valores específicos a cada uno.
 - **Product:** Es una clase que implementa la interfaz **IProduct** por lo que debe tener sus mismos atributos y métodos, de los cuales hereda, sirve para después poder probar a través de tests el funcionamiento de la interfaz **IProduct**.
 - **Atributos:** Son los mismos de la interfaz **IProduct**.
 - **Métodos:** Cada atributo debe implementar su **getter()** y **setter()**, los cuales se encargan de extraer e insertar los valores específicos a cada uno.
- **Pruebas TDD (Test Driven Development):**
- **Entidad Reseña:**
 - **Interfaces:** Atributos y métodos que debe implementar una serie de clases.

- **IReview:** Es la **interfaz que define las características que son descritas en las reseñas de fotografías acerca de los distintos tipos de productos** (cámaras y sus lentes). Se define como interfaz porque, aunque **producto sea una entidad distinta**, las reseñas describen características sobre ellas. Todos los tipos de productos que sean creados a través de la **interfaz IReview** comparten los siguientes atributos o métodos:
 - **Atributos:** Title, Content, Author (que puede ser un array para describir varios autores o solo uno) y **IProduct** (que es una interfaz que describe los productos que crear las fotografías analizadas en las reseñas).
 - **Métodos:** Cada atributo debe implementar sus métodos **getter()** y **setter()**, los cuales se encargan de extraer e insertar los valores específicos a cada uno.
- **Clases (Entidades de la base de datos):** Atributos y métodos que instanciarán objetos.
 - **Feature:** Es una clase de tipo array o lista que puede agregar una serie de características o funcionalidades extra a la interfaz **IProduct**, sirve para dar flexibilidad de diseño con el fin de poder añadir atributos o métodos que no hayan sido pensados desde un inicio.
 - **Atributos:** El feature de un producto puede tener atributos como Name y Description.
 - **Métodos:** Cada atributo debe implementar su **getter()** y **setter()**, los cuales se encargan de extraer e insertar los valores específicos a cada uno.
- **Pruebas TDD (Test Driven Development):** *Este tipo de pruebas se ejecutan a través de un script de código por separado que importe las diferentes interfaces o clases que se quiera poner a prueba, además tendrá en su nombre la palabra test y lo que realizará es una prueba unitaria que pondrá a prueba la funcionalidad de la entidad, interfaz o clase importada para comprobar si cumple todos los casos de uso descritos en los requerimientos de negocio.*
 - **Prueba de caso de uso:** Un Review puede tener productos dentro de ellas y el producto tiene que ser capaz de producir contenido que sea utilizado o visualizado cuando se consulte el review.
 - **Al ejecutar los tests:** Se elige un lenguaje de programación con el cual se defina algún script que utilice una librería de testing (en Python se utiliza pytest) y luego se utilizará un método (o función que es lo mismo) que indicará cuál es la lógica de negocio que se quiere ejecutar para obtener un resultado en concreto. Esto se compone de 3 etapas:
 - **Setup:** En esta sección se instancian o crean objetos de las diferentes clases o interfaces que se quiera probar, asignándole un valor a cada uno de sus atributos y utilizando sus métodos correspondientes. Para ello podemos hacer uso de alguna las siguientes herramientas.
 - **Mocks:** Es un objeto falso no implementado que se nos permite utilizar como substituto de la implementación real de una interfaz o clase sin preocuparnos por dependencias (osea otras clases o interfaces de las que dependa esta para funcionar correctamente).

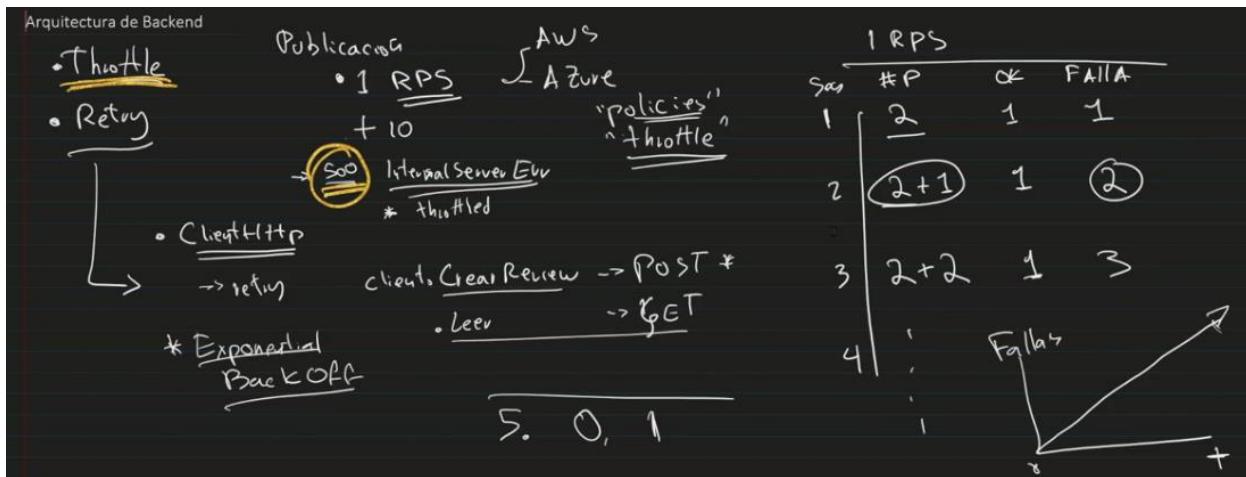
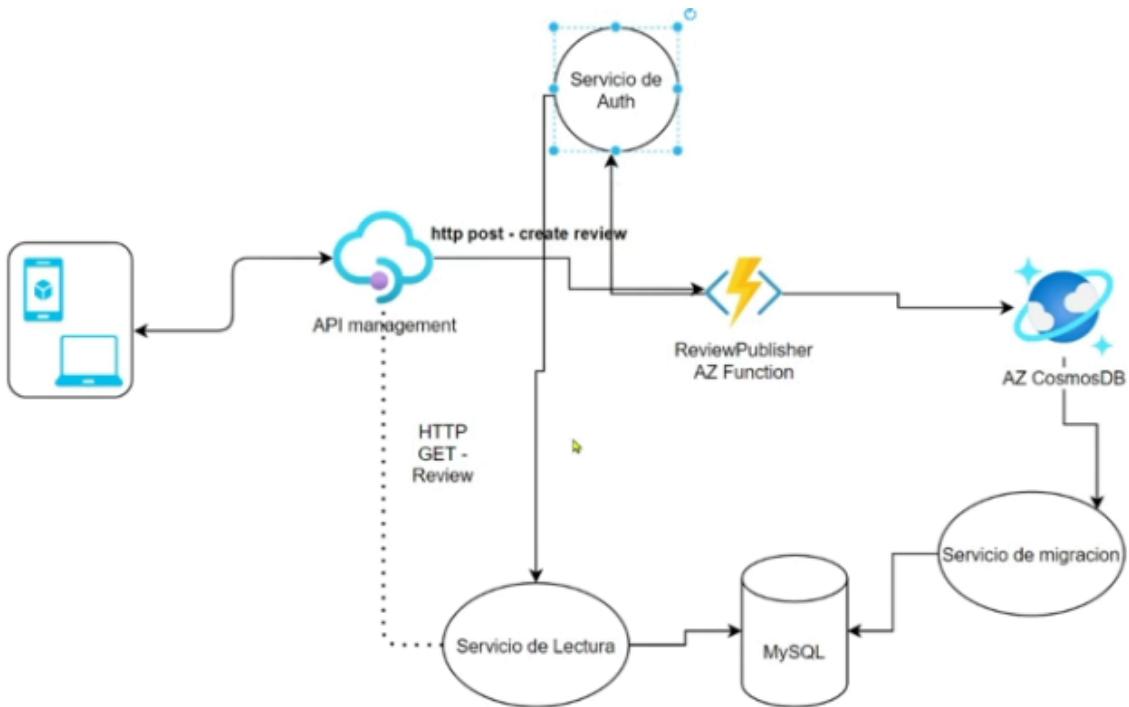
- **Interface Mock:** Para que una interfaz pueda ser probada a través de un objeto Mock, se debe crear una clase que implemente la interfaz que se quiera probar, la interfaz no puede ser probada directamente, debe tener en medio una clase de implementación siempre.
 - **ProductImpl:** Para poder probar la funcionalidad de la interfaz **IProduct** se debe crear la clase de implementación **ProductImpl** y además en esta sección se le debe asignar los valores con los que se vaya a probar, incluyendo atributos y métodos.
- **Exec:** Esta parte sirve para utilizar los objetos o instancias inicializadas en la sección de **Setup** a través de sus métodos.
 - **IProduct:** Si se está probando la interfaz de producto, se puede utilizar el método **GetContent()** que devuelve todas las características de un producto para que estas puedan ser visualizadas por las reseñas de fotografías.
- **Assert:** Esta es la etapa de validación en la prueba, donde se utilizan los métodos de la librería de testing que se haya elegido para analizar si el resultado obtenido de la prueba fue exitoso o no.
 - **IProduct:** Si se está comprobando el resultado de la interfaz de producto, se puede comprobar que lo que haya retornado el método **GetContent()** no sea null, osea que no sea un valor vacío, asegurándonos así que en verdad el método esté retornando las características del producto a analizar.
 - **Documentación de Errores:** Algo útil de la sección de assert es que se puede imprimir a través de texto en consola el resultado que se debía obtener si es que la prueba no resultó exitosa.
- Es recomendable ejecutar un script de test diferente para cada entidad distinta del código, en este caso se tendría que crear un archivo de pruebas para la entidad de **Producto** y otro para la de **Reseña**.

Escalabilidad: Throttling y Retry Policy

Retry Policy es una estrategia que se adopta cuando el tráfico de datos es demasiado alto, osea que se están haciendo muchas requests hacia un servicio, al ocurrir esta situación lo que puede pasar es que alguna de estas falle, y cuando esto suceda se puede programar estrategias para ver si ha fallado la petición y volver a intentar hacerla.

Hay otro concepto relacionado que se llama Throttling que se ocupa para evitar sobrecargar servidores, donde si se empieza a recibir un millón de requests en un segundo y este solo tiene capacidad para procesar 200 peticiones por segundo (RPS: Requests Per Second), las restantes fallarán todas, cuando esto ocurre lo que se puede hacer es que las requests que hayan fallado se reintenten a través de un Retry

Policy, pero para que estas no vuelvan a fallar como inicialmente lo hicieron, cuando se encuentre un código de error en la respuesta del servicio, este sumará un "delay" antes de volver a intentar enviar dicha petición, a este retraso se le llama Backoff.



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.

