



Documento plantilla - 1ra Evaluación Sumativa 30% Full Stack I Proyecto: Plataforma de Reserva de Salas de Estudio

Rúbrica y detalles de la Primera evaluación evaluación

[DSY1103 Evaluación Parcial 1_EE_EP_Estudiante_2024.pdf](#)

Rúbrica y detalles de la Primera evaluación evaluación

[Caso de Estudio 1 - Perfulandia SPA.docx](#)

Contexto de la problemática

La universidad **CampusLibre** ha desarrollado una plataforma digital que permite a estudiantes, docentes y funcionarios **reservar salas de estudio** dentro del campus para actividades individuales o grupales. Actualmente, el sistema funciona bajo una arquitectura monolítica que ha comenzado a mostrar **problemas de rendimiento**, especialmente en horarios de alta demanda (como inicio de semana o temporada de exámenes). Los usuarios han reportado errores frecuentes en el proceso de reserva, falta de notificaciones y una interfaz poco intuitiva. Frente a este escenario, el equipo de desarrollo ha sido convocado para analizar el sistema actual y proponer una **nueva arquitectura basada en microservicios**, que mejore la escalabilidad, disponibilidad y experiencia de usuario de la plataforma.



Este documento tiene como propósito guiarte en la elaboración del **Documento de Requisitos del Sistema**, el cual es parte fundamental del análisis de un proyecto tecnológico. En este caso, trabajaremos con una **plataforma digital que permite reservar salas de estudio dentro de un campus universitario**,

con el fin de comprender cómo se estructura la documentación técnica inicial de un sistema informático.

Formato del documento

1 Introducción



La plataforma en cuestión ha sido implementada por la universidad **CampusLibre**, con el objetivo de permitir que estudiantes, docentes y funcionarios puedan **reservar salas de estudio** para trabajos grupales, tutorías u otras actividades académicas. Sin embargo, con el paso del tiempo y el aumento en la demanda, el sistema actual —construido bajo una **arquitectura monolítica**— ha comenzado a presentar diversos problemas: lentitud en horas punta, errores al confirmar reservas, falta de notificaciones automáticas y dificultades para mantener el sistema actualizado. Frente a este escenario, se ha tomado la decisión de rediseñar la plataforma utilizando **microservicios**, para mejorar su rendimiento, escalabilidad y mantenibilidad. A partir de este caso, aprenderás cómo levantar los requisitos de un sistema real y cómo organizar esa información de forma profesional.

2 Requisitos Funcionales

Los **requisitos funcionales** describen las funcionalidades específicas que el sistema debe ofrecer a los distintos tipos de usuarios. Se basan en lo que el sistema **debe permitir hacer**.

2.1 Gestión de Usuarios

- **RF01** – El sistema debe permitir a los usuarios registrarse mediante correo electrónico y contraseña.
- **RF02** – El administrador podrá bloquear o suspender cuentas que infrinjan las normas de uso.

2.2 Reservas de Salas

- **RF03** – El usuario podrá buscar salas por fecha, hora, número de personas y equipamiento disponible.
- **RF04** – El sistema debe enviar correos automáticos confirmando o cancelando una reserva.

2.3 Gestión de Pagos

- **RF05** – El sistema deberá integrar un módulo de pago en línea para reservar salas premium.
- **RF06** – El usuario podrá consultar su historial de reservas y pagos en un perfil personal.

 Consejo: Piensa en los botones del sistema. ¿Qué opciones debe tener cada tipo de usuario? Cada una es un posible requisito funcional.

3 Requisitos No Funcionales

Los **requisitos no funcionales** definen las **características de calidad del sistema**, como su rendimiento, seguridad o facilidad de uso.

- **RNF01** – El sistema debe estar disponible las 24 horas, todos los días, con un tiempo de actividad del 99.5%.
- **RNF02** – El sistema debe cargar las páginas críticas (como reservas) en menos de 1.5 segundos.
- **RNF03** – Toda la navegación debe estar protegida mediante HTTPS y cifrado SSL.
- **RNF04** – La aplicación debe ser responsive, adaptándose bien a dispositivos móviles y tablets.

 Tip: Imagina que ya usas el sistema. ¿Qué cosas te molestarían? ¿Qué aspectos harían que la experiencia sea fluida y confiable?

4 Entrevistas Simuladas

Las entrevistas simuladas ayudan a **entender la perspectiva de los usuarios** reales del sistema. Aquí debes imaginar una conversación con distintos perfiles y responder desde sus necesidades.



4.1 Administrador del Sistema

Pregunta: ¿Qué tareas realiza con mayor frecuencia?

Respuesta: Me encargo de revisar el uso semanal de salas, generar reportes y controlar el acceso de usuarios al sistema.

Pregunta: ¿Qué espera del nuevo sistema?

Respuesta: Me gustaría un panel más visual, con gráficos automáticos y alertas cuando una sala esté sobreutilizada.



4.2 Usuario Frecuente

Pregunta: ¿Qué experiencia ha tenido reservando salas?

Respuesta: Bastante buena, aunque a veces reservo y luego me avisan que estaba ocupada por error.

Pregunta: ¿Qué le gustaría agregar?

Respuesta: Me encantaría tener la opción de extender mi reserva sin tener que cancelar y volver a reservar otra vez.

4.3 Tipo de Usuario o Rol 3

.....

4.4 Tipo de Usuario o Rol 4

.....

5 Descripción de Fallas

En esta sección debes **identificar y describir con claridad los errores más notorios** que presenta el sistema actual de reservas. Las fallas son comportamientos **incorrectos, inesperados o deficientes** que afectan el funcionamiento normal de la plataforma y generan una **mala experiencia para los usuarios**.

Estas fallas pueden ser de tipo técnico (por ejemplo, caídas del sistema), funcional (como procesos mal implementados) o de interfaz (problemas de usabilidad). No necesitas usar lenguaje técnico avanzado, pero sí debes ser **específico** y evitar frases vagas como "el sistema anda mal".

 Piensa en esto: ¿Qué cosas fallan cuando el sistema es más necesario? ¿Qué errores hacen perder tiempo, frustran o generan desorden?

👉 ¿Cómo escribir esta sección?

- Observa cómo interactuaría un usuario típico con el sistema (por ejemplo, un estudiante que quiere reservar una sala a las 9:00).
- Imagina qué pasaría si el sistema falla en medio de esa acción.
- Describe el fallo desde la perspectiva del usuario y del sistema.
- Usa numeración clara para cada falla (F01, F02, etc.).

Ejemplos:

- **F01** - El sistema se cae los lunes a las 9:00 cuando todos los estudiantes intentan reservar al mismo tiempo.
- **F02** - Si la reserva falla, no se notifica al usuario; simplemente "no pasa nada".
- **F03** - No hay validación de horarios, por lo que dos usuarios pueden reservar la misma sala al mismo tiempo.
- **F04** - Si se actualiza el sistema, todo el sitio queda fuera de línea.
- **F05** - No hay respaldo automático de la base de datos.

 Consejo: Usa ejemplos realistas. Piensa en errores que tú mismo odiarías encontrar en una app de reservas.

6 Principales Cuellos de Botella

En esta sección debes identificar los **puntos críticos del sistema que provocan lentitud o bloqueos** en su funcionamiento general. Un **cuello de botella** es como una autopista que de repente se estrecha: aunque haya mucho potencial en otros tramos, el flujo se ve limitado por una sola parte que **no da abasto**.

 Los cuellos de botella no siempre son errores, pero sí son limitaciones técnicas o estructurales que impiden que el sistema funcione con fluidez cuando crece o cuando hay muchos usuarios conectados al mismo tiempo.

👉 ¿Cómo escribir esta sección?

1. Piensa en **momentos de alta demanda**: ¿Qué partes del sistema se "atascan" o responden lentamente?
2. Observa qué procesos **dependen demasiado unos de otros**, generando dependencia innecesaria.
3. Identifica si hay **recursos centralizados** (una única base de datos, un único servidor, una sola API, etc.).
4. Describe cómo ese cuello de botella **afecta al sistema en general**.

📌 Ejemplos:

- **CB01** - Todas las funciones dependen de un único servidor, lo que lo sobrecarga.
- **CB02** - El motor de reservas revisa todas las salas antes de confirmar, lo que retrasa el proceso.
- **CB03** - El sistema guarda todo en una sola tabla enorme de base de datos, haciendo que las consultas sean lentas.
- **CB04** - La interfaz está mezclada con la lógica de negocio, lo que dificulta cualquier cambio de diseño.

⚙️ Consejo técnico: Piensa como si fueras el sistema... ¿dónde se "atascaría" cuando hay muchos usuarios?

7 Causas Técnicas ✨

En esta sección debes analizar **por qué ocurren las fallas y los cuellos de botella**, desde una **perspectiva tecnológica**. No se trata solo de listar problemas, sino de **profundizar en las razones técnicas** que explican el mal funcionamiento o la lentitud del sistema.

 Piensa en el "detrás de escena" del sistema: ¿Cómo está construido? ¿Qué tecnologías usa? ¿Qué limitaciones tiene su arquitectura actual?

¿Cómo escribir esta sección?

- Describe el diseño actual del sistema (por ejemplo: monolítico, sin capas, sin desacoplamiento).
- Explica cómo las **tecnologías, decisiones de arquitectura o prácticas de desarrollo** están generando problemas.
- Puedes mencionar términos como base de datos centralizada, falta de pruebas, baja modularidad, etc.
- Si usas conceptos técnicos, explícalos brevemente para que cualquiera pueda entenderlos.

Ejemplos:

- **CT01** - El sistema es monolítico y no permite separar funcionalidades.
- **CT02** - No hay pruebas automatizadas, así que cada cambio es riesgoso.
- **CT03** - No se usa ningún sistema de balanceo de carga.
- **CT04** - Todo está guardado en una sola base de datos sin optimización.

 Consejo: Usa conceptos técnicos que hayas aprendido y trata de aplicarlos con lógica, sin copiar y pegar sin entender.

8 Causas Organizativas

En esta sección debes identificar los factores **humanos, administrativos o de procesos** que también están causando o reforzando los problemas del sistema. A

veces el software **no falla por problemas técnicos**, sino porque **la forma en que se organiza y gestiona su uso** no es la adecuada.

 Aunque el código esté bien hecho, un mal proceso, una mala comunicación o una cultura resistente al cambio pueden hacer que el sistema siga fallando o no evolucione.

¿Cómo escribir esta sección?

- Piensa en cómo **trabajan los equipos** involucrados en el sistema (técnicos, usuarios, administradores).
- Pregúntate si existen **buenos canales de comunicación**, procesos definidos, roles claros y apoyo de la dirección.
- Describe ejemplos de **falta de organización, liderazgo, capacitación o inversión** que impactan negativamente en el sistema.

Ejemplos:

- **C001** - El equipo de desarrollo es reducido y no hay especialistas en backend.
- **C002** - No hay procesos de despliegue continuo, todo se sube "a mano".
- **C003** - Los encargados de administración y reservas no se comunican con TI.
- **C004** - La dirección no quiere invertir en actualizar el sistema actual por miedo al cambio.

 Consejo: No todo es culpa del código. A veces los errores están en cómo se organiza el trabajo o en la cultura de la empresa.

9 Propuesta de Servicios

En esta sección debes proponer una **división del sistema en microservicios**, es decir, **pequeños módulos independientes**, cada uno con una responsabilidad específica. El objetivo es **romper el sistema monolítico** en componentes más manejables, que puedan **escalarse, actualizarse y mantenerse por separado**.

💡 Esta propuesta representa el primer paso hacia una arquitectura moderna basada en microservicios, donde **cada servicio puede ser desarrollado, desplegado y monitoreado de forma autónoma**.

👉 ¿Cómo escribir esta sección?

1. **Identifica las funciones principales del sistema** (registro, reservas, pagos, notificaciones, etc.).
 2. **Agrupa cada función como un servicio separado**, dándole un nombre y una breve descripción.
 3. **Aclara qué rol cumple cada servicio**, a qué tipo de usuario sirve y cómo se relaciona con otros servicios.
 4. Si quieres, **usa nombres técnicos simples y consistentes**, como "Servicio de Reservas", "Servicio de Autenticación", etc.
-

- **S01 - Servicio de Autenticación:** Maneja el inicio de sesión, registro de usuarios y validación de tokens.
- **S02 - Servicio de Gestión de Usuarios:** Permite modificar perfiles, asignar roles (estudiante, administrador, funcionario).
- **S03 - Servicio de Gestión de Salas:** Controla el inventario de salas disponibles, sus características, horarios y ubicaciones.
- **S04 - Servicio de Reservas:** Administra la lógica de disponibilidad, creación, modificación y cancelación de reservas.
- **S05 - Servicio de Pagos:** Procesa pagos en línea para reservas de salas premium y genera comprobantes.
- **S06 - Servicio de Notificaciones:** Envía correos, alertas y recordatorios sobre reservas y vencimientos.

- **S07 - Servicio de Reportes:** Permite a los administradores generar reportes sobre uso de salas, horarios más ocupados, etc.
- **S08 - Servicio Web del Usuario:** Frontend o portal web accesible desde cualquier dispositivo, que consume los servicios anteriores.

 Tip: Cada servicio debe tener una sola responsabilidad clara. Si tu servicio empieza a “hacer demasiadas cosas”, probablemente debas separarlo.

10 Diagramas

Aquí deberás incluir **tres diagramas clave** que complementan el diseño de la nueva arquitectura. Puedes hacerlos en Lucidchart, Draw.io, Miro, Figma, Canva, etc.

10.1 Diagrama de Casos de Uso

Este diagrama debe representar **qué funcionalidades pueden usar los distintos actores del sistema**, como estudiantes, administradores, personal de salas, etc.



<<include>>

Se usa siempre, es obligatorio.

Ejemplo:

- Caso principal: **Reservar sala**
- Caso incluido: **Verificar disponibilidad**

 **Interpretación:** No puedes reservar sin primero verificar si la sala está disponible.

El sistema **siempre incluye** esa acción.

Reservar sala <<include>> Verificar disponibilidad



<<extend>>

| Se usa solo a veces, si se cumple una condición.

Ejemplo:

- Caso base: **Reservar sala**
- Caso extendido: **Aplicar descuento por fidelidad**

👉 **Interpretación:** Si el usuario es parte del programa de fidelidad, se activa el descuento.

Si no, la reserva se hace igual, pero sin extensión.

Reservar sala <<extend>> Aplicar descuento por fidelidad

Diagrama de Casos de Uso

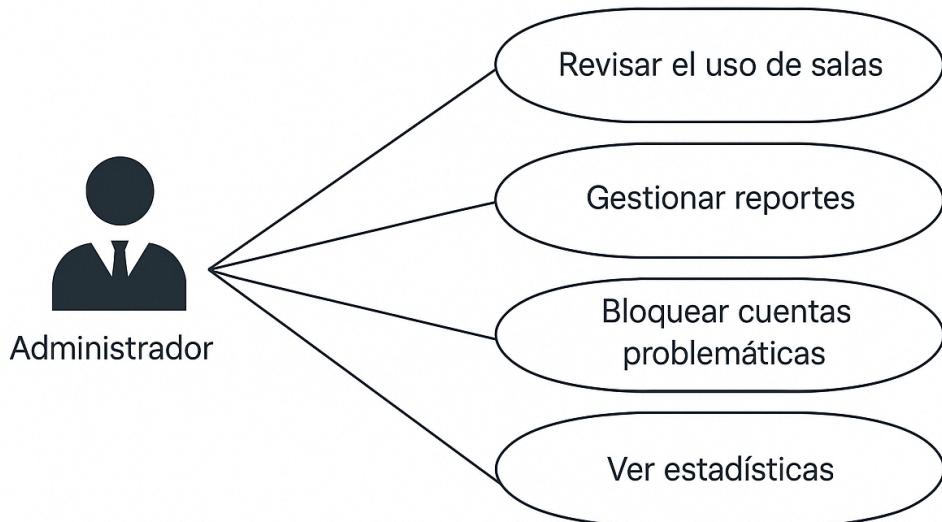
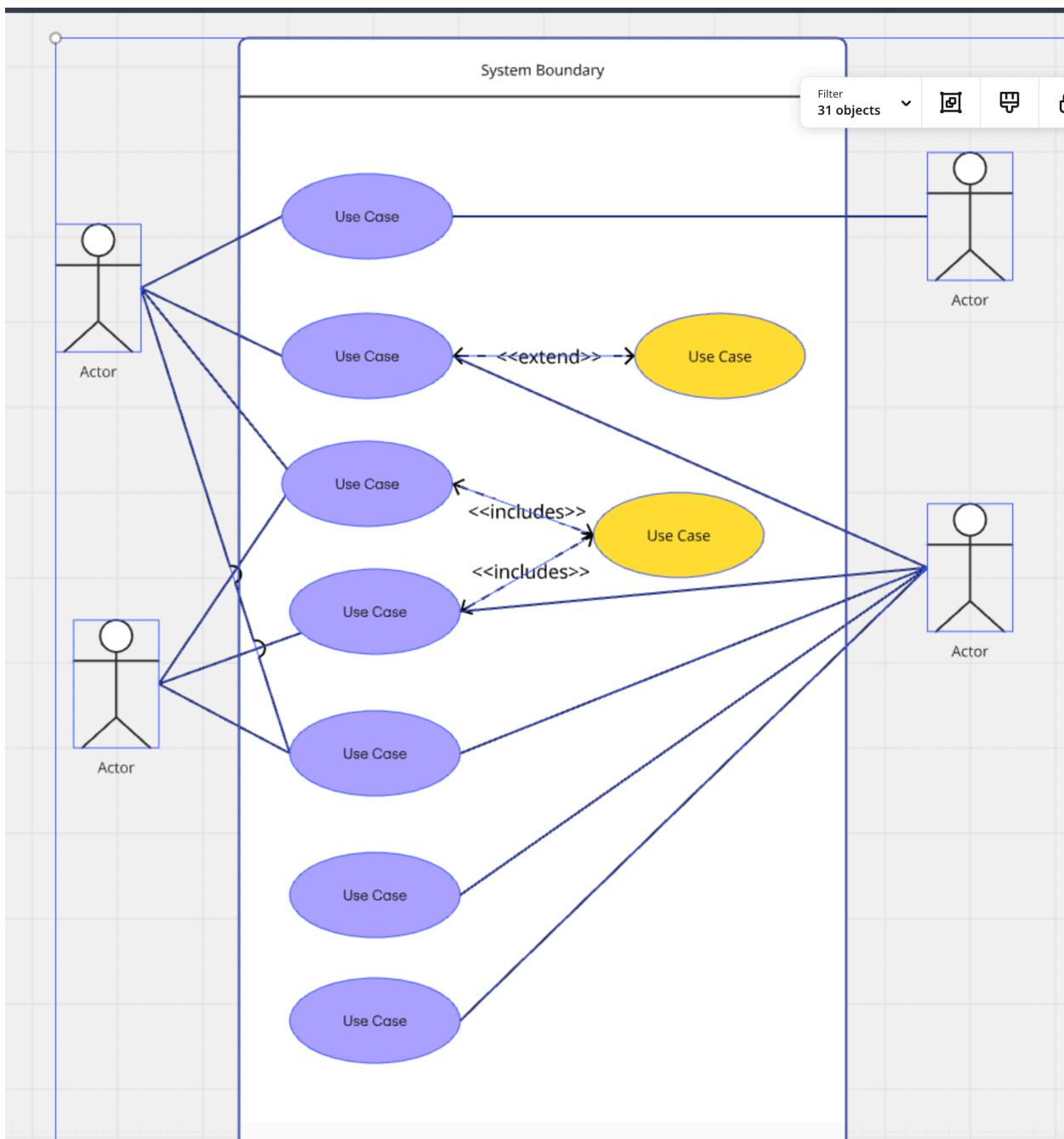


Diagrama de Casos de Uso



<<INSERTAR DIAGRAMA DE CASOS DE USO AQUÍ>>

10.2 Diagrama de Clases

Aquí debes mostrar las **entidades principales** del sistema y cómo se relacionan entre sí. Es ideal para entender la lógica de negocio interna.

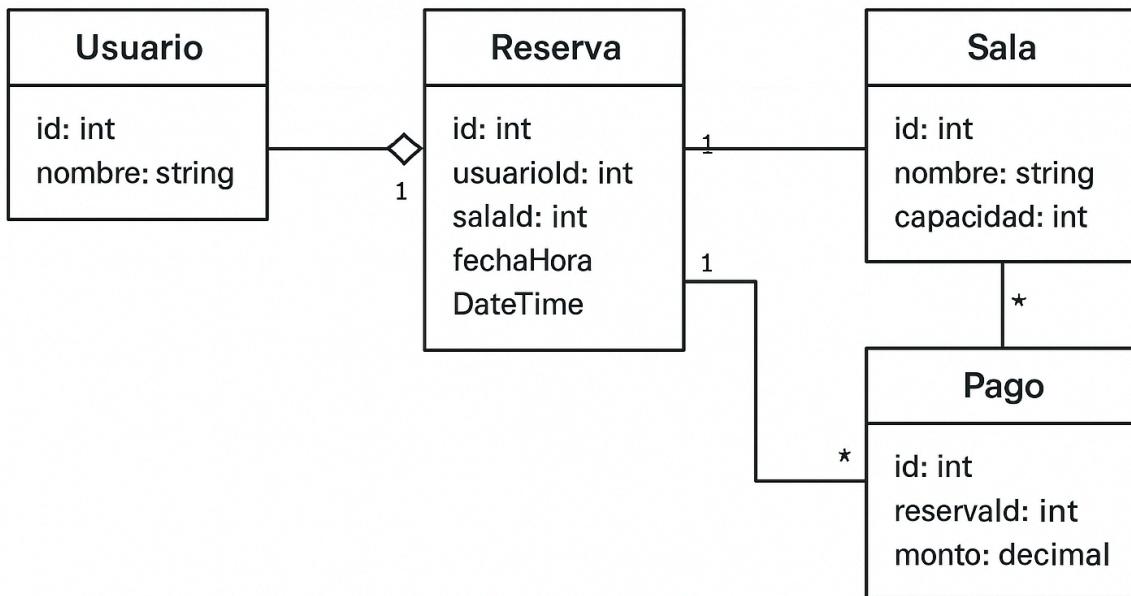
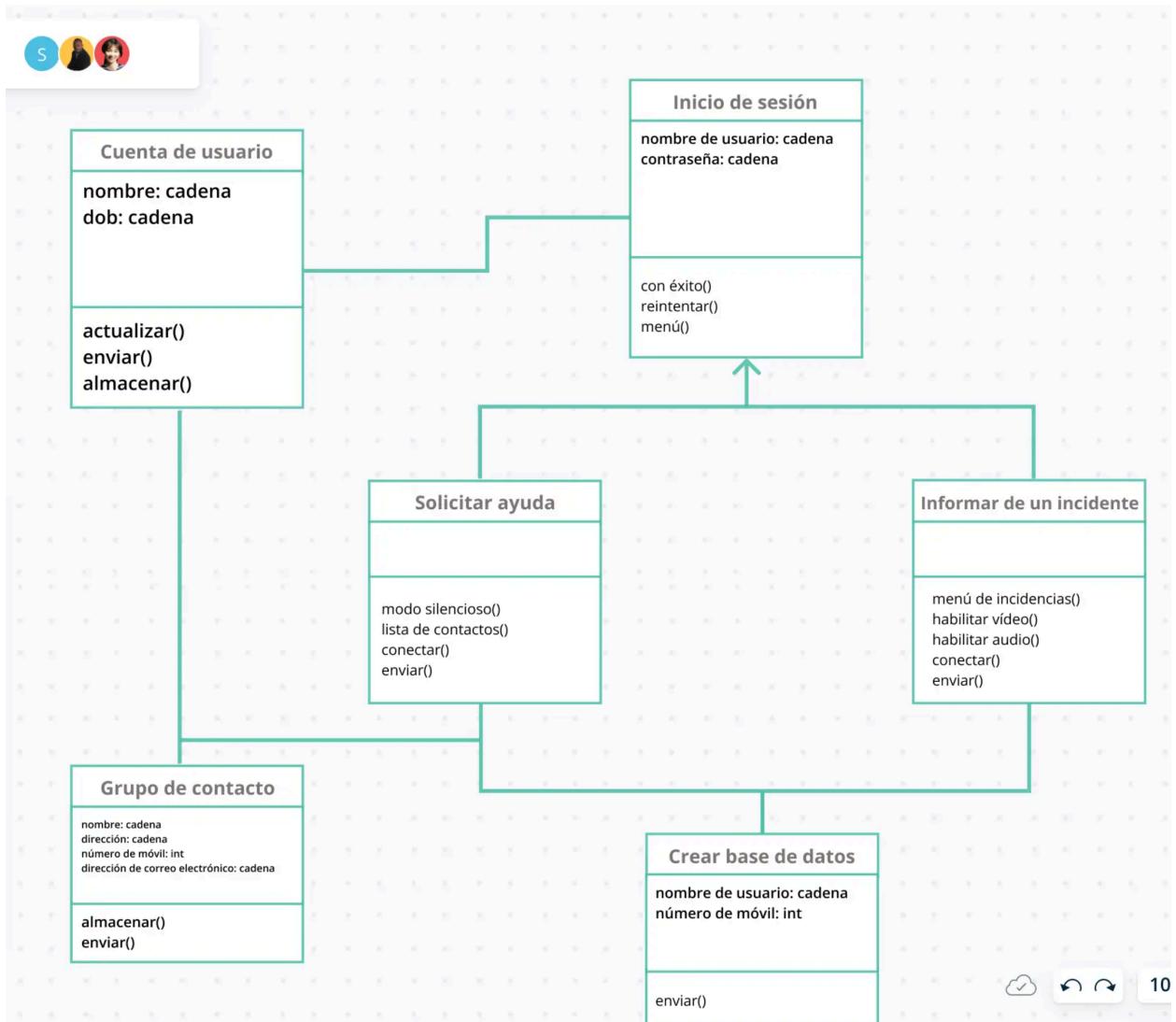


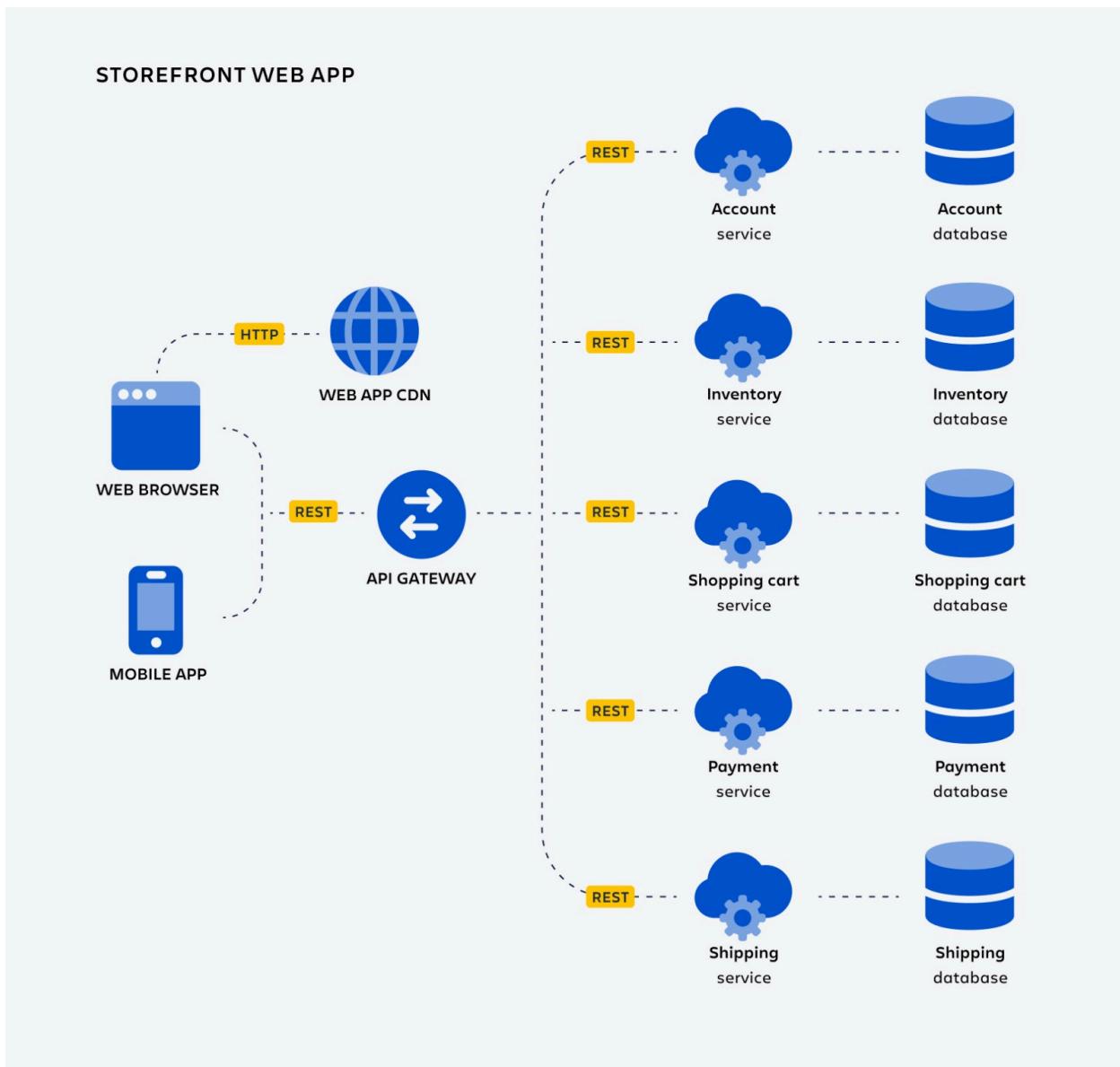
Diagrama de Clases



💡 <<INSERTAR DIAGRAMA DE CLASES AQUÍ>>

10.3 🌐 Diagrama de arquitectura de Microservicios

Este diagrama ilustra cómo se **distribuyen e interconectan los microservicios** en una infraestructura moderna, como contenedores (Docker) y orquestadores (Kubernetes)



 <<INSERTAR DIAGRAMA AQUÍ>>

🚀 11-Plan de Migración

🛠 11.1 - Fase 1: Análisis del Sistema Actual

Antes de hacer cualquier cambio, es fundamental **entender cómo está construido el sistema actual**. Esto incluye:

- Hacer un **mapa del sistema monolítico**, es decir, ver qué módulos existen, cómo se relacionan y qué hace cada uno.
- Identificar **qué partes se usan más**, cuáles están mal hechas, cuáles fallan más y cuáles podrían mantenerse tal como están.
- Detectar **módulos que pueden separarse fácilmente**, como por ejemplo: "envío de correos", "generación de reportes" o "autenticación".

 Ejemplo aplicado: En la plataforma de reservas, el módulo de **notificaciones por email** es un excelente candidato para convertirse en el primer microservicio porque su funcionalidad está **relativamente aislada del resto del sistema**. Es decir, su tarea principal es enviar correos automáticos cuando ocurre un evento, como la confirmación de una reserva, un recordatorio antes de la hora agendada o la notificación de una cancelación. Este tipo de acciones **no afecta directamente la lógica principal** del sistema (reservar salas), por lo que si el servicio de notificaciones fallara, el usuario aún podría reservar sin problemas. Además, puede configurarse para **recibir eventos** desde otros microservicios a través de una cola de mensajes o API, procesarlos, y actuar de forma autónoma. Esto lo convierte en una forma segura y sencilla de iniciar la migración hacia microservicios.

11.2 -Fase 2: Definición de Microservicios

Aquí debes planificar **cómo vas a dividir el sistema en servicios independientes**. Cada microservicio debe tener **una responsabilidad clara**, como si fuera una

persona de un equipo: uno se encarga de reservas, otro de pagos, otro de notificaciones, etc.

- Es recomendable empezar por **servicios “no críticos”**, que no afecten demasiado al funcionamiento general si fallan.
- Los servicios pueden estar relacionados, pero **deben poder funcionar por separado**.

 Ejemplo: El microservicio de **reportes** es ideal para comenzar, ya que su función es recopilar y mostrar información histórica del sistema, como cantidad de reservas por semana, salas más utilizadas o tiempos de uso promedio. Este tipo de datos no forma parte del proceso crítico de reservar una sala en tiempo real, por lo tanto, puede desarrollarse **de manera paralela** al sistema principal sin interrumpir su funcionamiento. Incluso si el servicio de reportes está en mantenimiento o se cae, los estudiantes y funcionarios seguirán pudiendo reservar salas sin problemas. Al separarlo como un microservicio, también se puede **optimizar su rendimiento** y generar estadísticas sin sobrecargar la base de datos principal. Además, puede crecer de forma independiente, permitiendo agregar gráficos, exportar a Excel o integrar paneles sin tocar el resto del sistema.

11.3 - Fase 3: Prueba de Concepto

Aquí se crea un **prototipo funcional de 1 o 2 microservicios**. La idea es probar que el concepto **funciona en la práctica**, sin reemplazar todavía el sistema antiguo.

- Puedes tomar el **Servicio de Gestión de Salas** o el **Servicio de Notificaciones** y desarrollarlo por separado.
- Este servicio se prueba en paralelo con el sistema actual, conectando ambos si es necesario.

 **Ejemplo:** Supongamos que se decide desarrollar una **prueba de concepto del Servicio de Gestión de Salas**. Este servicio se encargaría exclusivamente de administrar la información de cada sala: nombre, capacidad, ubicación, disponibilidad horaria, equipamiento, etc. Se puede crear una pequeña base de datos independiente solo con esos datos y construir una API sencilla para consultarlos, sin tocar el sistema actual de reservas. Luego, desde el sistema monolítico, se puede **redirigir temporalmente la consulta de salas hacia este nuevo microservicio**, validando si responde correctamente, si se comporta bien bajo carga y si se puede desplegar de forma aislada. El objetivo aquí no es reemplazar todavía el módulo original, sino **demostrar que un servicio separado puede cumplir su función y comunicarse correctamente**. Esta etapa permite detectar errores de integración tempranamente y ganar confianza para continuar la migración.



11.4 -Fase 4: Reestructuración Incremental

Aquí empieza el trabajo fuerte: **separar poco a poco los demás módulos del sistema**, como **"usuarios"**, **"reservas"**, **"pagos"** y **"autenticación"**.

Para comunicar estos microservicios entre sí, se pueden usar dos cosas:

Aquí empieza el trabajo fuerte: **separar poco a poco los demás módulos del sistema**, como "usuarios", "reservas", "pagos" y "autenticación". Cada uno de estos módulos debe transformarse en un **microservicio autónomo**, con su propia lógica, base de datos (idealmente), y forma de comunicarse con los demás. La clave en esta fase es **no romper la funcionalidad existente**, sino ir trasladando gradualmente los componentes a servicios desacoplados que se integren sin generar conflictos.

Para lograr que los microservicios **se comuniquen entre sí**, existen principalmente **dos enfoques**:

1. **APIs RESTful**: Es como si los servicios se enviaran mensajes por WhatsApp. Cada servicio tiene "endpoints" (puertas de entrada) a las que otros servicios pueden acceder usando URLs para pedir o enviar información. Por ejemplo, el servicio de reservas podría pedir al servicio de usuarios: "*Dime si este usuario está activo antes de permitir la reserva*".
2. **Colas de mensajes (RabbitMQ, Kafka)**: En lugar de pedir la información directamente, los servicios **envían mensajes a una "bandeja común"** (la cola). Otro servicio escucha esa cola y responde cuando puede. Esto permite que los servicios sean más tolerantes a fallos. Por ejemplo, cuando se crea una reserva, el sistema no le pide al instante al servicio de notificaciones que envíe un correo, sino que **envía un mensaje a la cola**, y el servicio de notificaciones lo toma y lo procesa cuando esté listo.

 Esta forma de comunicación hace que el sistema completo sea más flexible, rápido y resistente a errores, ya que cada microservicio puede seguir funcionando incluso si otro está temporalmente inactivo.

APIs RESTful (explicación sencilla):

Es una forma de que los servicios se "hablen" entre ellos usando direcciones URL (como en un navegador) para enviar y recibir datos.

 Por ejemplo, el servicio de reservas puede preguntarle al de usuarios: "*¿Quién está haciendo esta reserva?*" usando una

| API.

RabbitMQ o Kafka (colas de mensajes):

En vez de que los servicios se pregunten directamente entre sí, **mandan mensajes a una "cola"**, como una fila de correos, y el otro servicio los va leyendo y respondiendo cuando puede. Esto **evita que se bloqueen** si uno está lento o caído.

 RabbitMQ y Kafka son herramientas que permiten que los servicios trabajen de forma asíncrona, es decir, sin depender unos de otros para seguir funcionando.

11.5-Fase 5: Despliegue en Contenedores

Ahora que tienes varios microservicios separados, necesitas una forma **eficiente, ordenada y repetible** de **instalarlos, ejecutarlos y moverlos entre distintos entornos** (por ejemplo: del computador del desarrollador a un servidor real o a la nube). Aquí es donde entran en juego los **contenedores** y sus herramientas de orquestación.

¿Qué es Docker?

Docker es como una **caja mágica** donde guardas todo lo que necesita un microservicio para funcionar: su código, librerías, dependencias y configuraciones.

Es como preparar un tupper con tu comida lista para llevar a cualquier lugar, calentarla y que esté igual de buena en cualquier cocina .

 Ejemplo aplicado: El microservicio de pagos puede "empaquetarse" en un contenedor Docker que incluye el servidor, la base de datos local, su API REST, y se puede desplegar en cualquier servidor (incluso en una notebook de

pruebas) sin preocuparse por si tiene instalado Java, Node.js o lo que sea.

¿Qué es Kubernetes?

Kubernetes (o "K8s") es el **encargado de coordinar muchos contenedores Docker al mismo tiempo**.

Es como un **orquestador o director de orquesta** que le dice a cada músico (microservicio) cuándo entrar, cuándo parar, qué hacer si se equivoca (reiniciarlo), y en qué escenario tocar (servidor).

- Decide **cuántas copias** de cada servicio deben correr.
- Monitorea si un microservicio se cae y lo reinicia automáticamente.
- Reparte la carga entre varios servidores si es necesario.

Ejemplo aplicado: Si el microservicio de reservas se vuelve muy demandado, Kubernetes puede lanzar automáticamente 3 copias más para atender más usuarios sin colapsar el sistema.

¿Y por qué esto importa?

Porque en una arquitectura de microservicios, ya no tienes **un solo sistema grande**, sino **muchos pequeños servicios que deben convivir bien juntos**.

Docker y Kubernetes te ayudan a que cada uno tenga su espacio, no se molesten entre sí, y se puedan mover o escalar sin dramas.

 Resumen con peras y manzanas:

Docker = cada microservicio en una caja lista para usar

Kubernetes = el encargado de administrar y coordinar esas cajas dentro del sistema completo

Esta fase es clave para que el sistema sea **flexible, portable y escalable**, y te prepara para llevarlo eventualmente a **la nube (AWS, Azure, GCP)** o a servidores más grandes sin cambiar el código.

11.6 - Fase 6: Integración Continua y Despliegue Continuo (CI/CD)

Una vez que los microservicios están separados y empaquetados en contenedores, es muy importante que el proceso de **actualizar y publicar esos servicios** sea lo más automático posible. Aquí es donde entra la **Integración Continua (CI)** y el **Despliegue Continuo (CD)**, dos prácticas clave del desarrollo moderno de software.

¿Qué es CI/CD explicado con peras y manzanas?

- **Integración Continua (CI)** significa que **cada vez que alguien sube un cambio de código (por ejemplo, a GitHub)**, se ejecutan automáticamente una serie de **pruebas y validaciones** para asegurarse de que ese cambio **no rompe nada**.
- **Despliegue Continuo (CD)** significa que **si todo sale bien con las pruebas**, el sistema **publica automáticamente la nueva versión** del microservicio en el entorno de pruebas, staging o producción.

 Imagina que cada vez que actualizas un microservicio, hay un robot que revisa tu trabajo, lo aprueba y lo instala por ti, ¡sin que tengas que mover un dedo!

Ejemplo aplicado:

Supongamos que un estudiante del equipo hace una mejora en el **Servicio de Gestión de Reservas**: ahora las reservas pueden incluir una nota opcional para el administrador.

Este cambio se sube al repositorio (por ejemplo, GitHub).

Con CI/CD configurado:

1. CI (Integración Continua): Se ejecutan automáticamente pruebas unitarias para asegurarse de que:

- Las reservas aún se crean correctamente.
- Las reservas sin nota siguen funcionando.
- No se rompió la API del servicio.

2. CD (Despliegue Continuo): Si todas las pruebas pasan:

- Se genera una **nueva imagen Docker del microservicio**.
- Kubernetes la recibe y reemplaza automáticamente la versión anterior con la nueva.
- Todo esto ocurre sin que nadie tenga que entrar al servidor, copiar archivos o reiniciar manualmente.



Si algo falla, el sistema detiene el despliegue y muestra dónde ocurrió el error, para que el equipo pueda corregirlo rápido.



Herramientas comunes para CI/CD

Herramienta	¿Para qué sirve?
GitHub Actions	Automatiza pruebas y despliegues directamente desde GitHub. Muy fácil de usar para estudiantes.
GitLab CI/CD	Sistema completo de pipelines para integración y despliegue.
Jenkins	Herramienta muy usada en la industria, aunque más compleja.
CircleCI / TravisCI	Plataformas externas especializadas en CI/CD.

Resumen: CI/CD es como tener un asistente técnico invisible que prueba tu código, lo empaca y lo instala sin errores. Te permite trabajar más rápido, con más seguridad y menos estrés.

11.7 -Fase 7: Eliminación del Monolito

Esta es la **última fase del proceso de migración**, y representa el momento en que el sistema ha completado su transformación tecnológica: **ya no se depende del sistema antiguo (monolito)** y todos los componentes esenciales están **separados, aislados y funcionando como microservicios independientes**.

Para llegar a este punto, servicios críticos como:

-  Autenticación
-  Gestión de usuarios
-  Gestión de salas
-  Reservas
-  Pagos
-  Notificaciones

...ya deben estar completamente migrados, con sus propias bases de datos, APIs y entornos de despliegue, funcionando en producción de forma estable.

Ejemplo aplicado (Plataforma de Reserva de Salas de Estudio):

Imagina que el sistema completo ya está funcionando con microservicios, y el **sistema monolítico antiguo se desactiva oficialmente**.

Ahora, si un lunes a las 9:00 a.m. hay alta demanda de reservas, y por alguna razón el **servicio de reportes** se cae temporalmente, **nada grave ocurre**: los usuarios **aún pueden entrar a la plataforma, autenticarse, ver la disponibilidad de salas, hacer una reserva y recibir su confirmación por correo**. El servicio de reportes puede arreglarse de forma independiente, **sin afectar a los demás servicios**.

Antes, en el sistema monolítico, **todo estaba unido**, así que si una parte fallaba (como reportes o correos), a veces **todo el sistema colapsaba**. Ahora, cada parte tiene su espacio, su propio ritmo y su propia estabilidad.

¿Qué ocurre al eliminar el monolito?

- Se **desconecta o elimina el código antiguo**, muchas veces más difícil de mantener.
 - Se **liberan recursos del servidor**, ya que cada microservicio puede distribuirse en la nube.
 - El sistema gana en **resiliencia**, porque un error en una parte no derriba todo.
 - Los equipos pueden **trabajar de forma paralela**: mientras uno mejora el servicio de reservas, otro puede actualizar el de usuarios sin interferencias.
-

🏁 Resultado esperado

Este es el **objetivo final de toda la migración**: contar con un sistema moderno, modular, ágil, escalable y fácil de mantener. En vez de una gran bola de código, ahora tienes un conjunto de componentes especializados, que **colaboran entre sí sin estorbase**, preparados para crecer con las necesidades reales del campus universitario.

🎯 Como diríamos con peras y manzanas: pasaste de tener una máquina enorme y frágil, a tener un equipo de trabajadores especializados que se coordinan para que todo funcione bien, incluso si uno falta por un rato.

12-📌 Conclusión

El análisis realizado en este documento nos permitió comprender en profundidad las **debilidades del sistema actual** de reservas de salas de estudio en la universidad CampusLibre, así como proponer una solución estructurada y moderna basada en microservicios. A través del levantamiento de requisitos, la identificación de fallas y cuellos de botella, y el diseño de una nueva arquitectura distribuida, se establece una **hoja de ruta clara para transformar un sistema monolítico obsoleto en una plataforma ágil, escalable y resiliente**. Esta transformación no solo resolverá los problemas técnicos actuales, sino que también mejorará la experiencia de estudiantes y funcionarios al ofrecer un servicio más confiable, rápido y adaptado a los tiempos. Este caso práctico

demuestra cómo un enfoque técnico bien documentado puede impactar positivamente en el entorno académico y en la calidad de los servicios que ofrece una institución.



Instrucciones para la Elaboración del Informe de Evaluación

Para desarrollar esta evaluación con el **mayor nivel de profesionalismo posible**, cada grupo deberá elaborar un **informe completo y bien estructurado**, siguiendo las indicaciones que se detallan a continuación.



Formato y Presentación

- El informe debe elaborarse en **Google Docs**, con redacción formal y siguiendo las **Normas APA**:
 - Fuente legible (Arial o Times New Roman), tamaño 12.
 - Interlineado 1.5.
 - Citas formales si corresponde.
- Debe incluir:
 - **Portada** (con nombre del proyecto, grupo, integrantes y fecha de entrega).
 - **Índice automático**.
- Se valorará una redacción clara, coherente, sin errores ortográficos ni gramaticales.
- Pueden utilizar la plantilla proporcionada como guía para estructurar el documento y trabajar de forma **colaborativa en las 12 secciones del informe**.



Fecha límite de entrega:

Todas las secciones del informe deben estar completadas y listas para revisión **antes del domingo 13 de abril a las 23:59 horas**. Esto asegurará **igualdad de condiciones** para todos los grupos al momento de ser evaluados.

Trabajo en Equipo

- El trabajo debe dividirse de manera **equitativa entre todos los integrantes**, incluyendo a quienes no asisten de forma presencial.
- Cada miembro debe tener un **rol asignado y un aporte real** dentro del documento.
- El equipo debe mantener una **comunicación activa** y documentar los avances individuales.
- El documento final debe ser compartido con el docente otorgándole **permisos de edición**, para recibir comentarios y retroalimentación directa en el archivo.

Entrega y Evaluación

- Esta evaluación considera **únicamente el contenido del informe final**.
- El **líder del grupo** deberá entregar un documento adicional indicando el **porcentaje de aporte individual** de cada integrante.

⚠ La nota final de cada estudiante será asignada en base a su nivel de participación real, según lo declarado en este documento.

Cada estudiante deberá subir en AVA los siguientes elementos antes del plazo final:

1. **Informe en formato PDF.**
2. **Enlace del informe en Google Docs**, previamente compartido con acceso de edición al docente.

🚫 No se aceptarán entregas fuera de plazo. Cualquier entrega posterior será evaluada con nota mínima (1.0).

Recomendaciones Finales

- Usen **comentarios y sugerencias en Google Docs** para coordinar ideas, proponer mejoras y revisar el trabajo de los compañeros.

- Apóyense en la plantilla proporcionada, el material del curso y los ejemplos vistos en clases.
 - Si tienen dudas, el docente estará disponible para orientarlos en los horarios establecidos.
-

 Este informe es su carta de presentación como futuros profesionales del área tecnológica.

Tómenlo con **seriedad, compromiso y profesionalismo**.

¡Mucho éxito, equipo!