

MÓDULO 1

Ciclo de Vida del Software

INGENIERÍA DE SOFTWARE I

Albeiro Montes



UNIVERSIDAD
NACIONAL
DE COLOMBIA



Grupo de
Ambientes
Inteligentes
Adaptativos

Ciclo de Vida del Software

El ciclo de vida del software es un conjunto de fases que abarca desde el nacimiento de una idea hasta la retirada del software del mercado. Asegura que el desarrollo siga un camino lógico y estructurado.

Fases principales del ciclo de vida del software:

- Análisis
- Diseño
- Pruebas
- Despliegue
- Mantenimiento y actualización

Ciclo de Vida del Software

Fases Principales

- **Análisis y levantamiento de requerimientos:** Se identifican las necesidades del usuario y las especificaciones del software.
- **Diseño:** Creación de la arquitectura del sistema y planificación detallada de cada módulo.
- **Desarrollo e implementación:** Programación del software, pruebas unitarias, e integración de componentes.
- **Pruebas:** Verificación de que el software cumple con los requerimientos y está libre de errores críticos.
- **Despliegue:** Lanzamiento del software en el entorno de producción, accesible para los usuarios.
- **Mantenimiento y actualización:** Resolución de errores post-lanzamiento y actualización del software para cumplir con nuevas necesidades o mejoras.

Ciclo de Vida del Software

Modelos de Ciclo de Vida

- Modelo en Cascada.
- Modelo Incremental.
- Modelo Ágil.
- Modelo en Espiral.

Ciclo de Vida del Software

Modelos de Ciclo de Vida

- **Modelo en Cascada:** Proceso secuencial, donde cada fase debe completarse antes de pasar a la siguiente. Ideal para proyectos con requerimientos bien definidos desde el inicio.
- **Ejemplo:** Desarrollo de un sistema de nómina para una empresa.
 1. En este caso, el sistema de nómina tiene requerimientos bien definidos y estables: cálculo de salarios, gestión de horas extras, retenciones de impuestos, etc.
 2. Dado que las necesidades son claras desde el principio y es improbable que cambien, el modelo en cascada funciona bien aquí, ya que permite avanzar secuencialmente (recolección de requerimientos → diseño → desarrollo → pruebas → implementación).

Ciclo de Vida del Software

Modelos de Ciclo de Vida

- **Modelo Incremental:** Basado en iteraciones cortas y entregas frecuentes. Promueve la adaptación y el ajuste continuo a los cambios.
- **Ejemplo:** Desarrollo de una aplicación móvil de redes sociales.
 1. Este tipo de aplicaciones suele evolucionar rápidamente en respuesta a las tendencias del mercado y las expectativas de los usuarios.
 2. El equipo trabaja en ciclos cortos (sprints), donde en cada sprint se agregan o mejoran funciones, como chat, historias, y notificaciones.
 3. Con el modelo ágil, es posible adaptarse a cambios rápidos en los requerimientos y agregar funcionalidades con retroalimentación constante de los usuarios.

Ciclo de Vida del Software

Modelos de Ciclo de Vida

- **Modelo Ágil:** Se desarrolla el software en pequeñas partes (incrementos), permitiendo retroalimentación y mejoras continuas.
- **Ejemplo:** Creación de una plataforma de comercio electrónico.
 1. La plataforma puede lanzarse en varias fases (incrementos). En una primera fase, el sistema incluye solo el catálogo de productos y un carrito de compras.
 2. En incrementos posteriores, se añaden nuevas funciones, como sistema de pago, historial de pedidos, opiniones de usuarios y recomendaciones personalizadas.
 3. Este enfoque permite entregar rápidamente un producto mínimo viable y luego mejorar y expandir la plataforma según la retroalimentación de los usuarios.

Ciclo de Vida del Software

Modelos de Ciclo de Vida

- **Modelo en Espiral:** Enfocado en la evaluación y reducción de riesgos. Combina elementos de los modelos anteriores, con una mayor atención en la gestión de riesgos.
- **Ejemplo:** Desarrollo de un sistema de seguridad para una empresa de alta tecnología.
 1. La seguridad es una prioridad y es esencial evaluar riesgos continuamente. El modelo en espiral permite dividir el desarrollo en fases (o "espirales") donde se evalúan riesgos en cada iteración.
 2. En cada iteración, el equipo trabaja en una fase de planificación, análisis de riesgos, desarrollo y evaluación, asegurando que el sistema sea seguro y cumpla con los estándares antes de avanzar a la siguiente espiral.
 3. Este modelo es ideal en situaciones donde los riesgos son altos y es importante mitigarlos antes de continuar.

Análisis: Levantamiento de Requerimientos

¿Qué es?

Es el proceso de identificar y documentar las necesidades y expectativas de los usuarios para el software.

¿Por qué es importante?

- Reducir errores en etapas posteriores del desarrollo.
- Mejorar la comunicación con el cliente.
- Asegurar que el producto final satisfaga las expectativas del usuario.

Análisis: Levantamiento de Requerimientos

Técnicas para el Levantamiento de Requerimientos

- **Entrevistas**
- **Encuestas y cuestionarios**
- **Talleres de trabajo**
- **Prototipos**
- **Análisis de documentos**

Análisis: Levantamiento de Requerimientos

Herramientas para gestión de Requerimientos

- JIRA
- Trello
- Notion
- Visio

Análisis: Levantamiento de Requerimientos

Tipos de Requerimientos

- **Requisitos Funcionales (RF):** Los Requisitos Funcionales describen **qué hace el sistema** o las funcionalidades específicas que debe tener para cumplir con los objetivos del proyecto. Básicamente, detallan **las tareas y acciones que el sistema debe poder realizar** desde el punto de vista del usuario.
 1. **Específicos y medibles:** Cada RF debe ser claro y verificable. Si el sistema cumple o no con un requisito funcional debe poder verificarse.
 2. **Enfocados en el usuario:** Están orientados a la experiencia del usuario final o a los objetivos específicos que debe cumplir el sistema.

Análisis: Levantamiento de Requerimientos

Tipos de Requerimientos

- **Requisitos No Funcionales (RNF):** Los Requisitos No Funcionales describen cómo debe funcionar el sistema en términos de rendimiento, usabilidad, seguridad, confiabilidad y otras cualidades que no están relacionadas directamente con las funciones específicas. Estos requisitos son fundamentales para garantizar que el sistema funcione de manera eficaz y satisfactoria en un entorno real.
- **Rendimiento.**
- **Seguridad.**
- **Usabilidad.**
- **Disponibilidad.**
- **Compatibilidad.**

Análisis: Levantamiento de Requerimientos

Tipos de Requerimientos

- Los **RF** aseguran que el sistema cumpla con los objetivos del usuario y ofrezca las funcionalidades necesarias.
- Los **RNF** garantizan que el sistema ofrezca una experiencia de calidad y que cumpla con estándares importantes como la seguridad y el rendimiento.

Análisis: Documentación de Requerimientos - Formato

1. Título del Requerimiento Funcional

ID: RF-001

Nombre del requerimiento: Registro de usuario.

Descripción: El sistema debe permitir a los usuarios registrar una nueva cuenta proporcionando su nombre, correo electrónico y contraseña.

Actor(es): Usuario final (registrante).

Entrada:

- Nombre (campo de texto)
- Correo electrónico (campo de texto)
- Contraseña (campo de texto)

Salida:

- Confirmación de que el usuario ha sido registrado correctamente.
- Notificación en caso de error (ej. "Correo electrónico ya registrado").

Criterios de Aceptación:

- El sistema debe validar que el correo electrónico es único.
- La contraseña debe tener al menos 8 caracteres.
- El sistema debe enviar un correo de confirmación al usuario.

Dependencias: El sistema de gestión de correos debe estar funcionando.

Prioridad: Alta

Análisis: Documentación de Requerimientos - Formato

1. Título del Requerimiento No Funcional

ID: RNF-001

Nombre del requerimiento: Rendimiento del sistema.

Descripción: El sistema debe ser capaz de procesar hasta 500 solicitudes por segundo sin degradar su rendimiento.

Requisito: El sistema debe responder dentro de 2 segundos para el 95% de las solicitudes.

Criterios de Aceptación:

- El sistema debe ser probado con una carga de hasta 500 usuarios concurrentes.
- El tiempo de respuesta debe ser inferior a 2 segundos en el 95% de las pruebas.

Dependencias: Infraestructura de red y servidores adecuados.

Prioridad: Alta

Análisis: Documentación de Requerimientos

Priorizar los Requerimientos: Clasificar los requisitos según su importancia y urgencia, identificando cuáles son esenciales para la funcionalidad básica del sistema y cuáles pueden ser añadidos en versiones futuras.

Detallar Casos de Uso: Elaborar casos de uso para cada funcionalidad principal, especificando el flujo de eventos y cómo cada actor interactúa con el sistema.

Modelado: Utilizar diagramas UML como diagrama de casos de uso, diagrama de secuencia, y diagrama de actividades para representar visualmente la estructura y flujo de trabajo del sistema.

Historias de Usuario: Descripción detallada de una funcionalidad que tiene una estructura definida.

Análisis: Documentación de Requerimientos – Formato (HU)

ID: HU-001

Título: Registro de Usuario

Historia: Como **usuario**, quiero **registrarme en el sistema** para **poder acceder a mis datos y configuraciones personalizadas**.

Criterios de Aceptación:

1. El sistema debe solicitar el nombre completo, correo electrónico y contraseña.
2. El sistema debe validar que el correo electrónico es único y mostrar un mensaje de error si el correo ya está registrado.
3. El sistema debe asegurar que la contraseña tenga al menos 8 caracteres.
4. El sistema debe enviar un correo de confirmación al usuario.
5. El usuario debe poder ingresar con sus credenciales luego de completar el registro.

Prioridad: Alta

Estimación de esfuerzo: 5 puntos

Dependencias: Base de datos de usuarios, sistema de envío de correos electrónicos.

Notas: La interfaz debe ser fácil de usar, con validación en tiempo real de los campos obligatorios.

MÓDULO 1

Casos de Uso

INGENIERÍA DE SOFTWARE I

Albeiro Montes



UNIVERSIDAD
NACIONAL
DE COLOMBIA



Grupo de
Ambientes
Inteligentes
Adaptativos

Casos de Uso

¿Qué es un Caso de Uso?

- Un **caso de uso** es una **descripción detallada** de cómo un usuario (actor) interactúa con un sistema para lograr un objetivo específico.
- Representa un **escenario** que muestra los pasos y la secuencia de eventos necesarios para completar una tarea.
- Es una herramienta para capturar los **requisitos funcionales** del sistema.

¿Por qué utilizar Casos de Uso?

- Ayudan a entender las necesidades del usuario.
- Facilitan la comunicación entre los interesados, desarrolladores y analistas.
- Proporcionan una visión clara de las funcionalidades que debe ofrecer el sistema.
- Son una base para el diseño de interfaces y para definir casos de prueba.

Casos de Uso

Componentes de un Caso de Uso:

1. **Actor:** Persona, sistema o entidad externa que interactúa con el sistema.
2. **Caso de Uso:** Describe la acción o función que realiza el sistema.
3. **Escenario:** Secuencia de pasos o eventos que se siguen para completar la tarea.
4. **Precondición:** Condición que debe cumplirse antes de que el caso de uso inicie.
5. **Postcondición:** Estado del sistema después de que el caso de uso ha finalizado.
6. **Excepciones:** Errores o situaciones especiales que pueden ocurrir durante la ejecución.

Casos de Uso

Ejemplo de Caso de Uso

Caso de uso: Sistema de Reserva de Hoteles: "Hacer una Reserva"

- **Actor:** Cliente
- **Caso de Uso:** Realizar una reserva
- **Escenario:**
 - El cliente selecciona las fechas de la estancia.
 - Elige el tipo de habitación.
 - Ingresa la información de pago.
 - Confirma la reserva.
- **Precondición:** El cliente debe estar registrado.
- **Postcondición:** La reserva se confirma y se guarda en el sistema.

Casos de Uso

Ejemplo de Caso de Uso

Caso de Uso: Transferir Dinero entre Cuentas

- **Actor:** Cliente del banco
- **Precondición:** El cliente ha iniciado sesión en su cuenta.
- **Escenario:**
 - El cliente selecciona la opción "Transferencia de dinero".
 - Ingresa la cuenta de origen y la cuenta de destino.
 - Especifica la cantidad a transferir.
 - Confirma la operación.
 - El sistema verifica los fondos disponibles en la cuenta de origen.
 - Si hay fondos suficientes, se realiza la transferencia.
 - El sistema notifica al cliente el éxito de la transacción.
- **Postcondición:** El dinero se ha transferido de la cuenta de origen a la cuenta de destino.
- **Excepciones:**
 - Si no hay fondos suficientes, el sistema muestra un mensaje de error.
 - Si la cuenta de destino no existe, el sistema notifica al cliente y cancela la operación.

Casos de Uso

Ejemplo de Caso de Uso

Caso de Uso: Prestar un Libro

- **Actor:** Bibliotecario
- **Precondición:** El usuario debe estar registrado en el sistema y el libro debe estar disponible.
- **Escenario:**
 - El bibliotecario ingresa el ID del usuario.
 - El bibliotecario escanea el código del libro a prestar.
 - El sistema verifica la disponibilidad del libro.
 - El sistema actualiza el estado del libro como "Prestado".
 - El bibliotecario entrega el libro al usuario.
- **Postcondición:** El libro aparece como "Prestado" en el sistema y se registra al usuario como prestatario.
- **Excepciones:**
 - Si el libro ya está prestado, el sistema muestra un mensaje indicando que el libro no está disponible.
 - Si el usuario no está registrado, el sistema muestra un error.

Diagramas de Casos de Uso

¿Qué es un Diagrama de Casos de Uso?

- Es una representación gráfica que muestra la **relación** entre los **actores** y los **casos de uso** del sistema.
- El diagrama describe **qué** hace el sistema, pero no **cómo** lo hace.
- Ayuda a entender y comunicar los **requisitos funcionales** del sistema.

Diagramas de Casos de Uso

Componentes de un Diagrama de Casos de Uso

Los elementos principales son:

1. Actor:

1. Representa a una **persona, sistema o entidad externa** que interactúa con el sistema.
2. Se dibuja como una figura de palo (stick figure).
3. Ejemplo: Cliente, Administrador, Usuario, Sistema de Pago.

2. Caso de Uso:

1. Describe una función o una tarea específica que realiza el sistema.
2. Se representa con una **elipse**.
3. Ejemplo: "Registrar usuario", "Hacer una compra", "Enviar notificación".

3. Relaciones:

1. **Asociación**: Conexión entre un actor y un caso de uso (una línea simple).
2. **Extensión (<<extend>>)**: Indica que un caso de uso puede extender otro caso de uso bajo ciertas condiciones.
3. **Inclusión (<<include>>)**: Indica que un caso de uso siempre incluye otro caso de uso.
4. **Generalización**: Relación jerárquica entre actores o casos de uso, similar a la herencia.

4. Sistema:

1. Representa el límite del sistema y se dibuja como un **rectángulo** que contiene los casos de uso.

Diagramas de Casos de Uso

Tipos de Relaciones en los Diagramas de Casos de Uso

1. Relación de Inclusión (<<include>>):

1. Indica que un caso de uso **incluye** otro caso de uso siempre que se ejecute.
2. Ejemplo: En "Hacer una compra", siempre se incluye "Realizar pago".

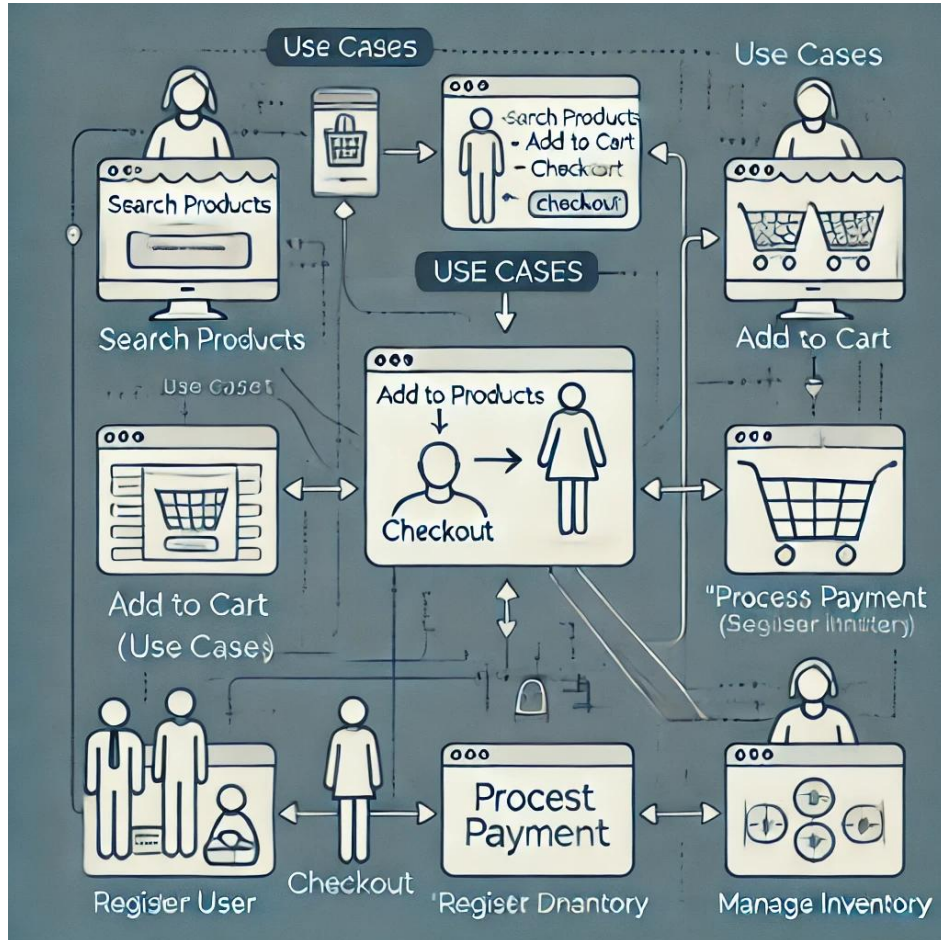
2. Relación de Extensión (<<extend>>):

1. Indica un **comportamiento opcional** o condicionado.
2. Ejemplo: "Generar recibo" extiende "Realizar pago" si el cliente solicita un recibo.

3. Generalización:

1. Se utiliza cuando un actor o caso de uso puede ser una **especialización** de otro.
2. Ejemplo: "Administrador" hereda de "Usuario", ya que tiene permisos adicionales.

Diagramas de Casos de Uso



Elaboración propia.

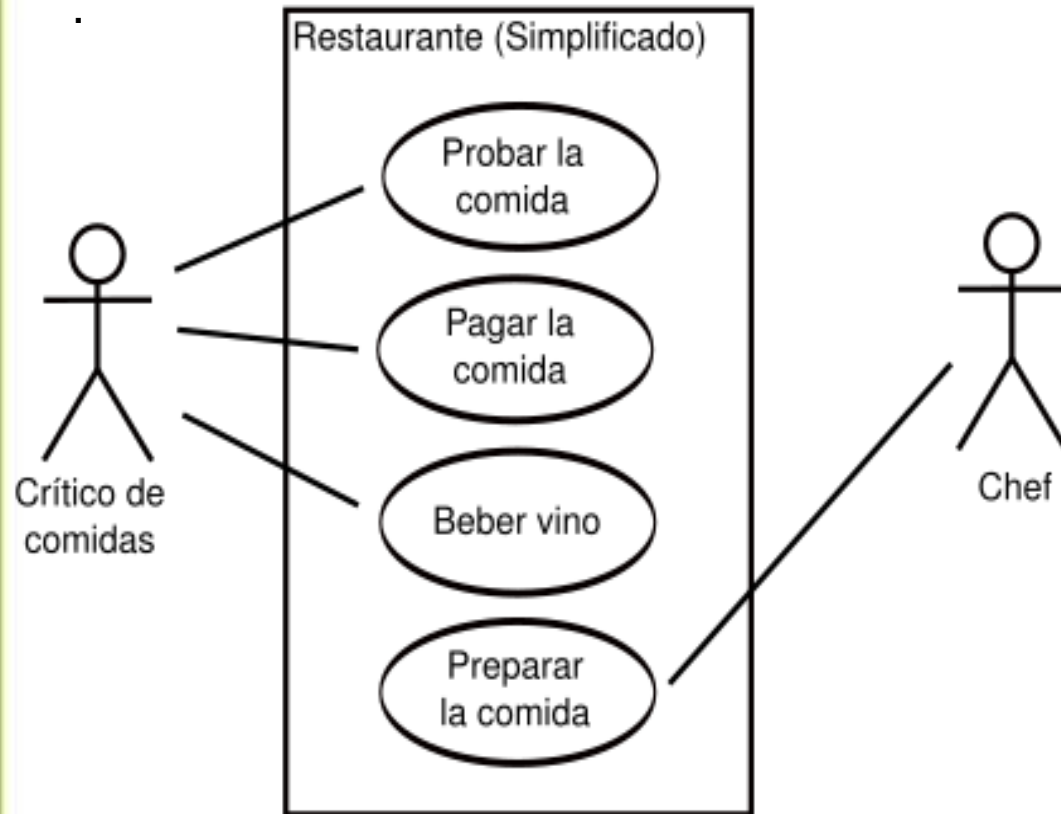
Casos de Uso

- **Buscar Productos:** Permite al cliente explorar el catálogo de productos.
- **Agregar al Carrito:** Función para seleccionar productos para la compra.
- **Realizar Pago (Checkout):** El cliente finaliza su compra a través de esta función.
- **Procesar Pago:** Caso de uso extendido por el Sistema de Pago para manejar la transacción.
- **Registrar Usuario:** Necesario para que el cliente pueda realizar compras.
- **Gestionar Inventario:** El administrador puede añadir, modificar o eliminar productos del sistema.

Relaciones

- **Asociación:** Línea simple que conecta actores con los casos de uso que pueden ejecutar.
- **Extensión (<<extend>>):** Indica un comportamiento opcional, como "Procesar Pago", que extiende "Realizar Pago".
- **Inclusión (<<include>>):** Ejemplo de "Registrar Usuario", que siempre se incluye al realizar una compra.

Diagramas de Casos de Uso



Desde Wikipedia

Casos de Uso

1. **Probar la comida:** Caso de uso que representa al cliente probando los alimentos.
2. **Pagar la comida:** Caso de uso donde el cliente realiza el pago por los alimentos consumidos.
3. **Beber vino:** Caso de uso que permite al cliente consumir bebidas alcohólicas, específicamente vino.
4. **Preparar la comida:** Caso de uso correspondiente a la preparación de los platos, realizado por el personal del restaurante.

Actores

- **Cliente** (actor a la izquierda): Interactúa con los casos de uso "Probar la comida", "Pagar la comida" y "Beber vino".
- **Cocinero o Personal del Restaurante** (actor a la derecha): Asociado con el caso de uso "Preparar la comida".

Relaciones

- Cada línea conecta a un actor con los casos de uso que realiza, indicando la interacción directa entre los usuarios y las funcionalidades del sistema.

Diccionario de Datos

¿Qué es un diccionario de datos? ¿Por qué es importante?



Diccionario de Datos

¿Qué es un diccionario de datos?

Herramienta de documentación que busca dar precisión semántica sobre los que se usan en un esquema de base de datos. Ayuda a los desarrolladores a tener claridad sobre cómo se deben manipular los registros.

¿Por qué es importante?

- Claridad semántica.
- Escalabilidad.
- Auditoría.
- Registro.

Diccionario de Datos

Formato de diccionario de datos

Campo	Descripción	Tipo	Tamaño	Restricción	Ejemplo
id	Identifica de manera única un registro	Integer	N/A	Llave primaria	3
nombre	Almacena el nombre del cliente	varchar	255	Obligatorio	"Nombre"
correo_electrónico	Almacena el correo electrónico del cliente	varchar	50	Opcional, único	"correo@gmail.com"
direccion	Almacena la dirección del cliente	text	N/A	Opcional	"Calle 123"

MÓDULO 1

Diseño

INGENIERÍA DE SOFTWARE I

Albeiro Montes



UNIVERSIDAD
NACIONAL
DE COLOMBIA



Grupo de
Ambientes
Inteligentes
Adaptativos

Diseño

La etapa de diseño es una fase del ciclo de vida del software donde se crea una solución técnica detallada basada en los requisitos. Esta etapa transforma los requisitos en una guía clara para la implementación.

Objetivos

- Definir la arquitectura del sistema.
- Establecer los componentes y módulos.
- Crear prototipos de interfaces de usuario.
- Detallar la lógica interna y las interacciones entre módulos.

Diseño (etapas principales)

Pasos de la etapa de diseño:

Diseño Arquitectónico: Definir la estructura general del sistema.

Diseño de la Base de Datos: Crear el modelo de datos y definir las relaciones.

Diseño de Interfaces de Usuario: Crear prototipos y wireframes.

Diseño Detallado de Componentes: Especificar la lógica interna y diagramas UML.

Diseño de Interfaces de Módulos: Definir la comunicación entre módulos.

Revisión y Validación: Verificar y ajustar el diseño con el equipo.

Estimaciones: A partir de la metodología de trabajo, se define antes o después de las demás etapas del diseño

Diseño (estimaciones)

Estimaciones:

Son las evaluaciones que se realizan para identificar los recursos necesarios para el cumplimiento del objetivo (refactorizar o implementar un nuevo producto).

Estimación de tiempo: Determinar cuánto tiempo se necesitará para completar una tarea o proyecto.

En contrataciones modalidad **ODS**, normalmente se debe aumentar un porcentaje con respecto al costo de acuerdo con:

- Pagos por reenefuente (10%).
- Aportes a seguridad social (IBC).

Diseño (estimaciones)

Estimaciones (tiempo):

Existen varias **metodologías y técnicas** utilizadas para establecer **estimaciones de tiempo** en proyectos, especialmente en el desarrollo de software. Estas metodologías ayudan a hacer predicciones más realistas y confiables sobre el tiempo que tomará completar tareas, entregables o el proyecto completo. Algunas de las metodologías más comunes incluyen:

1. Estimación por Analogía:

Un equipo de desarrollo ha trabajado en un proyecto similar de una aplicación de comercio electrónico. En ese proyecto, se tardaron 300 horas en desarrollar la funcionalidad de pago. Ahora, en un nuevo proyecto, se tiene una funcionalidad de pago muy similar. Usando **estimación por analogía**, el equipo puede estimar que el tiempo necesario será de unas 300 horas, ya que las funcionalidades son casi idénticas.

Diseño (estimaciones)

Estimaciones (tiempo):

2. Descomposición (WBS - Work Breakdown Structure):

Este método implica descomponer el proyecto en tareas más pequeñas y manejables. Luego, se estiman los tiempos de cada tarea individualmente y se suman para obtener la estimación total del proyecto.

Un proyecto de desarrollo de software se descompone en las siguientes tareas:

Tarea 1: Diseño de interfaz de usuario (UI) – **Estimación:** 80 horas.

Tarea 2: Desarrollo de la funcionalidad de registro de usuario – **Estimación:** 120 horas.

Tarea 3: Integración de la base de datos – **Estimación:** 100 horas.

Tarea 4: Pruebas y depuración – **Estimación:** 40 horas.

Al sumar las estimaciones de todas las tareas ($80 + 120 + 100 + 40$), se obtiene una estimación total de 340 horas.

Diseño (estimaciones)

Estimaciones (tiempo):

3. Delphi:

Esta metodología se basa en un grupo de expertos que, de manera anónima, proporcionan sus estimaciones sobre el tiempo necesario para completar tareas. Las respuestas se recopilan, se discuten y se ajustan hasta que se obtiene una estimación consensuada.

Un equipo de expertos (5 personas) estima el tiempo necesario para implementar una nueva funcionalidad en una aplicación. Cada experto da una estimación de tiempo en horas. Los resultados iniciales son:

Experto 1: 40 horas

Experto 2: 50 horas

Experto 3: 60 horas

Experto 4: 45 horas

Experto 5: 55 horas

Se calcula el promedio para determinar que la duración es de 50 horas.

Diseño (estimaciones)

Estimaciones (tiempo):

4. PERT:

Este método utiliza tres estimaciones para cada tarea (RF o RNF):

Optimista (O): El tiempo más corto posible.

Pesimista (P): El tiempo más largo posible.

Realista (R): El tiempo más probable.

Luego, se calcula el resultado usando la fórmula PERT

$$t = \frac{O + 4R + P}{6}$$

t: tiempo de desarrollo de cada actividad.

Diseño (estimaciones)

Estimaciones (tiempo):

5. Monte Carlo:

Utilizar simulaciones estadísticas para predecir el tiempo de un proyecto basado en diferentes variables e incertidumbres. Esta técnica genera múltiples escenarios posibles y los analiza para estimar el rango de tiempo probable.

Ejemplo: https://colab.research.google.com/drive/1dnTNBrnrDzcxW5_tOR5dw5AaiS9eLXOq?usp=sharing

Diseño (estimaciones)

Estimaciones (costos):

Las estimaciones por costos buscan establecer en lo posible un valor concreto para el valor a pagar o a recibir por un desarrollo de software, ya sea un nuevo aplicativo o una refactorización, de acuerdo, con la información disponible.

1. Analogía:

Al igual que en la estimación por tiempos, se puede tomar como referencia un sistema anterior que cumple con características similares al proyecto en cuestión para tomar su valor como criterio. Se recomienda tener en cuenta los factores económicos como cambios de divisas, tasas de inflación, entre otros.

Ejemplo: Un equipo de desarrollo tiene experiencia en proyectos de aplicaciones móviles similares. Basándose en proyectos anteriores, los expertos estiman que el nuevo proyecto costará **\$50,000**. Utilizan sus conocimientos sobre problemas comunes y complejidad para ajustar la estimación.

Diseño (estimaciones)

Estimaciones (costos):

2. Puntos de Función: Esta metodología mide la funcionalidad del software basada en la cantidad de entradas, salidas, consultas, archivos y interfaces externas. Cada punto de función se asigna a un costo estimado.

Elemento	Datos o control introducidos desde el usuario o sistemas externos.	Baja	Media	Alta
Entradas externas (EE)	Datos introducidos al sistema desde el usuario o sistemas externos (ej., formularios).	4	5	7
Salidas Externas (SE)	Datos generados por el sistema y presentados al usuario (ej., informes o pantallas).	5	7	10
Consultas Externas (CE)	Solicitudes de datos sin modificar la base de datos (ej., búsquedas).	4	6	8
Archivos Internos (AI)	Archivos dentro del sistema que contienen datos estructurados y son manipulados por el sistema.	7	10	15
Interfaces Externas (IE)	Archivos o sistemas externos con los que el sistema intercambia información (ej., conexiones con bases de datos externas).	5	7	10

Diseño (estimaciones)

Estimaciones (costos):

2. Puntos de Función:

Ejemplo: El proyecto incluye:

- **15 entradas:** (formularios de usuario)
- **10 salidas:** (reportes)
- **5 consultas:** (búsquedas)
- **3 archivos internos:** (bases de datos)
- **2 interfaces externas:** (API)

Utilizando una tabla estándar, se asignan **4 puntos de función** para cada entrada, **5 para cada salida**, **4 para consultas**, **10 para archivos internos** y **7 para interfaces externas**. La suma total es:

$$Total\ Puntos = \sum (entradas, salidas, consultas, archivos, interfaces)$$

$$Total\ Puntos = (15 \times 4) + (10 \times 5) + (5 \times 4) + (3 \times 10) + (2 \times 7) = 60 + 50 + 20 + 30 + 14 = 174$$

Estableciendo el costo por función \$200, el total sería $174 \times \$200 = \34.800 .

Diseño (estimaciones)

Estimaciones (costos):

3. Paramétrica:

Utiliza parámetros o métricas para predecir costos. Un ejemplo es calcular costos basados en líneas de código (LOC).

Ejemplo: Una compañía utiliza el parámetro "costo por línea de código (LOC)" para estimar proyectos. Si saben que el costo promedio es de \$100 por LOC y el nuevo proyecto tendrá alrededor de 10.000 LOC, entonces estiman un costo de:

$$\begin{aligned} Total &= LOC * \text{Líneas Código} \\ Total &= \$100 * 10000 = \$1'000.000 \end{aligned}$$

Diseño (estimaciones)

Estimaciones (costos):

4. Delphi:

Es un proceso iterativo en el que varios expertos realizan estimaciones de manera independiente. Los resultados se discuten y se ajustan hasta alcanzar un consenso.

Ejemplo: Tres expertos en el equipo de desarrollo realizan estimaciones independientes para un proyecto:

- **Experto A:** \$120.000
- **Experto B:** \$100.000
- **Experto C:** \$110.000

Los resultados se discuten en equipo y, tras ajustar las diferencias, se calcula el promedio y se define **\$110,000** como costo estimado.

Diseño (estimaciones)

Estimaciones (costos):

5. Bottom Up:

Se realiza una estimación detallada de cada tarea del proyecto y luego se suman los costos para obtener una estimación total.

Ejemplo: El proyecto se desglosa en tareas específicas con sus costos estimados:

- **Diseño UI:** \$10.000
- **Desarrollo Backend:** \$30.000
- **Desarrollo Frontend:** \$25.000
- **Pruebas:** \$15.000
- **Implementación:** \$5.000

El costo total se calcula sumando todas las tareas:

$$\text{Costo} = \$10.000 + \$30.000 + \$25.000 + \$15.000 + \$5000 = \$85.000$$

Diseño (estimaciones)

Estimaciones (costos):

6. Top-Down:

Se realiza una estimación global basada en una visión general del proyecto, y luego se distribuye el presupuesto entre las distintas fases o componentes.

Ejemplo: Basándose en presupuestos previos, se asigna un costo global de \$100,000 para el proyecto. Luego se distribuye este presupuesto entre diferentes fases:

Planificación: 10% → \$10.000

Desarrollo: 60% → \$60.000

Pruebas: 20% → \$20.000

Implementación: 10% → \$10.000

Diseño (estimaciones)

Estimaciones (costos):

7. COCOMO (COConstructive COst MOdel)

COCOMO es un modelo empírico que estima el esfuerzo (en persona-meses) y el costo de un proyecto de software en función de su tamaño (medido en líneas de código, LOC) y la complejidad del proyecto. Se basa en tres niveles de modelo:

1. **Modelo Básico:** Estimación inicial basada solo en el tamaño del proyecto.
2. **Modelo Intermedio:** Considera factores de costos adicionales.
3. **Modelo Detallado:** Más preciso, considerando más detalles de características del proyecto.

Categoría	Descripción	Valor de a	Valor de b
Orgánico	Proyectos pequeños, bien definidos, con pocos cambios y un equipo experimentado.	2.4	1.05
Semidetallado	Proyectos de tamaño intermedio con algunos cambios de requisitos o poco claros.	3.0	1.12
Embebido	Proyectos grandes y complejos, con requisitos poco claros, múltiples interfaces, y restricciones técnicas.	3.6	1.2

Diseño (estimaciones)

Estimaciones (costos):

7. COCOMO (COConstructive COSt MOdel)

Fórmula para estimar E , donde:

- E = (esfuerzo estimado por persona por mes).
- $KLOC$ = es el tamaño del software en **miles** de líneas de código (Kilo Lines of Code).
- a y b varían según la categoría del proyecto.

Ejemplo:

Paso 1: Establecer parámetros.

Suponga un proyecto que aproximadamente tomará 10.000 líneas de código, semidetallado.

$$a = 3.0$$

$$b = 1.12$$

$$KLOC = 10.000/1000 = 10$$

Diseño (estimaciones)

Estimaciones (costos):

7. COCOMO (COConstructive COst MOdel)

Paso 2: aplicar fórmula

$$\begin{aligned}E &= a * (KLOC)^b \\E &= 3.0 * (10)^{1.12} \\E &= 39.96\end{aligned}$$

Paso 3: interpretar

El esfuerzo estimado para completar el proyecto es 39.36 persona-meses. Esto significa que, si un equipo de trabajo estuviera compuesto por 10 personas, se necesitarían aproximadamente 4 meses para completar el proyecto (asumiendo que las 10 personas trabajen a tiempo completo).

Paso 4: estimar costos: asumiendo que cada miembro del equipo cobra \$1.500, el costo del proyecto sería de:

$$\text{Costo total} = E * \text{costo miembro equipo} = 39.96 * 1500 = \$59.940$$

Diseño (estimaciones)

Actividad

Estimar la duración y el costo de un proyecto de desarrollo de software utilizando diferentes metodologías de estimación, basadas en los Requisitos No Funcionales (RNF) y los Requisitos Funcionales (RF). Las metodologías que se deben aplicar son **Delphi**, **Monte Carlo**, **PERT** y **WBS**, para tiempo. **Puntos de Función** y **COCOMO** para costos.

RF	RNF
El sistema debe permitir a los usuarios registrarse y crear un perfil.	El sistema debe ser accesible desde dispositivos móviles y de escritorio.
El sistema debe permitir a los usuarios registrar eventos, especificando nombre, fecha, lugar y tipo de evento.	El sistema debe estar disponible 24/7 y garantizar un tiempo de actividad del 99%.
El sistema debe permitir a los usuarios inscribirse en eventos y recibir confirmaciones por correo electrónico.	El tiempo de respuesta de las solicitudes debe ser inferior a 2 segundos en el 95% de los casos.

Introducción a la Arquitectura de Software

INGENIERÍA DE SOFTWARE I

Albeiro Montes



UNIVERSIDAD
NACIONAL
DE COLOMBIA



Grupo de
Ambientes
Inteligentes
Adaptativos

Arquitectura de Software

¿Qué es la Arquitectura de Software?

Estructura general de un sistema en donde se definen los componentes, interacciones y relaciones; a partir de elementos como:

- **Escalabilidad:** Diseñar una arquitectura adecuada asegura que el sistema pueda manejar un crecimiento en usuarios o datos sin degradar el rendimiento.
- **Mantenibilidad:** Promueve un código limpio y modular, lo que reduce costos de mantenimiento a largo plazo.
- **Eficiencia:** Minimiza errores y duplicidad de esfuerzos.

Arquitectura de Software

¿Qué es una API?

API (Application Programming Interface): Es un conjunto de reglas y protocolos que permite que dos aplicaciones se comuniquen entre sí. Una API define cómo un software puede interactuar con otro.

Una API es como un menú en un restaurante:

- El cliente (la aplicación) solicita un plato del menú (la funcionalidad).
- La cocina (el sistema backend) lo prepara.
- El mesero(la API) lo entrega al cliente.

Arquitectura de Software

¿Qué hace una API?

- **Conexión entre aplicaciones:** Permiten que diferentes sistemas trabajen juntos, como una app móvil conectándose a una base de datos.
- **Abstracción de complejidad:** El usuario no necesita saber cómo funciona el backend, solo usa las funciones que la API ofrece.
- **Interoperabilidad:** Facilitan la integración de servicios de terceros, como pagos en línea (PayPal, Stripe) o mapas (Google Maps).
- **Reutilización de servicios:** Una API puede ser utilizada por múltiples plataformas: web, móvil, IoT, etc.

Arquitectura de Software

Tipos de APIs

- **APIs REST:** Basadas en HTTP. Usan métodos como GET, POST, PUT, DELETE.

Ejemplo: API de Twitter para obtener tweets.

- **APIs Públicas:** Disponibles para cualquier desarrollador.

Ejemplo: API de Google Maps.

- **APIs Privadas:** Usadas internamente por una organización.

Ejemplo: API de un banco para manejar transacciones.

Arquitectura de Software

Estructura Técnica de una API

Solicitudes (Requests): Las APIs reciben solicitudes de los clientes a través de un protocolo, generalmente HTTP o HTTPS.

Componentes clave de una solicitud:

- **Método HTTP:** Define la acción que se quiere realizar (GET, POST, PUT, DELETE, entre otros).
- **URL/Endpoint:** Dirección del recurso al que se accede (/api/users).
- **Headers:** Información adicional como autenticación, tipo de contenido, etc.
- **Body:** Datos enviados al servidor (en métodos como POST y PUT). Normalmente en formato JSON o XML.

Arquitectura de Software

Estructura Técnica de una API

Respuestas (Responses): Las APIs retornan respuestas al cliente tras procesar una solicitud. Componentes clave de una respuesta:

- **Código de estado HTTP:** Indica el resultado de la operación.
 1. **20X:** Solicitud exitosa.
 2. **40X:** Error en la solicitud.
 3. **50X:** Error en el servidor.
- 4. **Headers:** Información sobre el tipo de datos retornados.
- 5. **Body:** Contiene los datos solicitados o mensajes de error. Generalmente en formato JSON.

Arquitectura de Software

Estructura Técnica de una API

Tipos de Peticiones HTTP en APIs

GET: Se usa para obtener datos.

Ejemplo: <http://localhost:8000/api/products> (Devuelve una lista de productos).

POST: se usa para enviar datos.

Ejemplo: <http://localhost:8000/api/products> (Recibe un JSON y retorna una respuesta).

```
Body: {  
  "name": "Juan",  
  "email": "juan@example.com"  
}
```

Arquitectura de Software

Estructura Técnica de una API

Tipos de Peticiones HTTP en APIs

PUT: Se usa para actualizar datos.

Ejemplo: <http://localhost:8000/api/products/1> (Recibe un JSON y retorna una respuesta).

Body: { "name": "Juan Actualizado" }

DELETE: se usa para eliminar datos.

Ejemplo: <http://localhost:8000/api/products/1> (Retorna una respuesta).

Arquitectura de Software

Creando una API Rest en Laravel:

En Laravel, **controladores**, **modelos** y **rutas** son los componentes principales que trabajan juntos para crear una API funcional. Estos elementos siguen el patrón **MVC (Modelo-Vista-Controlador)** y permiten una separación clara de responsabilidades en tu aplicación.

Modelos: Un modelo en Laravel representa una tabla en la base de datos y permite interactuar con ella de manera sencilla. Proporciona acceso a los datos, aplicando lógica empresarial como validaciones o relaciones entre tablas.

Características principales:

- Define la estructura y las relaciones de los datos.
- Proporciona funciones para realizar consultas (find, where, all, entre otras).
- Utiliza Eloquent, el ORM (Object-Relational Mapping) de Laravel, para mapear tablas a clases PHP.

Arquitectura de Software

Creando una API Rest en Laravel:

Modelos:

```
class User extends Model
{
    // Especificar la tabla asociada (opcional si el nombre sigue la convención)
    protected $table = 'users';

    // Permitir la asignación masiva de estos campos
    protected $fillable = ['name', 'email', 'password'];

    // Definir relaciones, por ejemplo, un usuario tiene muchos posts
    public function posts()
    {
        return $this->hasMany(Post::class);
    }
}
```

Arquitectura de Software

Creando una API Rest en Laravel:

Rutas: Las rutas son las que definen cómo las solicitudes HTTP (GET, POST, PUT, DELETE) se manejan en tu aplicación y se asocian a un controlador.

Características principales:

- Se definen en archivos como routes/api.php o routes/web.php.
- Especifican el endpoint de la API y el método HTTP asociado.
- Pueden devolver directamente una respuesta o redirigir a un controlador

Arquitectura de Software

Creando una API Rest en Laravel:

Rutas:

// Importar el controlador

use App\Http\Controllers\UserController;

// Ruta para obtener todos los usuarios, que apunta a un método llamado “index” en la clase del controlador.

Route::get('/users', [UserController::class, 'index']);

// Ruta para crear un usuario, que apunta a un método llamado “store” en la clase del controlador.

Route::post('/users', [UserController::class, 'store']);

// Ruta para obtener un usuario específico, que apunta a un método llamado “show” en la clase del controlador.

Route::get('/users/{id}', [UserController::class, 'show']);

// Ruta para actualizar un usuario, que apunta a un método llamado “update” en la clase del controlador.

Route::put('/users/{id}', [UserController::class, 'update']);

// Ruta para eliminar un usuario, que apunta a un método llamado “destroy” en la clase del controlador.

Route::delete('/users/{id}', [UserController::class, 'destroy']);

Arquitectura de Software

Creando una API Rest en Laravel:

Controladores: Los controladores son responsables de procesar la lógica de negocio. Actúan como intermediarios entre el modelo (datos) y la respuesta (JSON en una API).

Características principales:

- Contienen funciones o métodos que se ejecutan cuando se llama a una ruta.
- Generalmente interactúan con el modelo para acceder o manipular datos.
- Devuelven respuestas HTTP en formato JSON para APIs.

Arquitectura de Software

Creando una API Rest en Laravel:

Controlador:

```
class UserController extends Controller
{
    public function index()
    {
        $users = User::all();
        return response()->json($users, 200);
    }

    public function store(Request $request)
    {
        $user = User::create($request->all());
        return response()->json($post, 201);
    }
}
```

Arquitectura de Software

Creando una API Rest en Laravel: Controlador:

public function show(\$id)

```
{  
    $user = User::find($id);  
    return response()->json($user, 200);  
}
```

public function update(Request \$request, \$id)

```
{  
    $user = User::find($id);  
    $user->update($request->all());  
    return response()->json($user, 200);  
}
```

public function destroy(\$id)

```
{  
    $user = User::find($id);  
    $user->delete();  
    return response()->json(['message' => 'Usuario eliminado'], 200);  
}
```

Arquitectura de Software

Creando una API Rest en Laravel (resumen)

1. **\$fillable:** El modelo permite la asignación masiva para los campos name, email y password. Esto asegura que solo estos campos pueden ser actualizados con `$request->all()` o mediante el método `create()`.
2. **Sin validaciones:** El controlador no valida los datos de entrada, lo cual no es seguro, pero puede ser útil en pruebas o para comprender la mecánica de Laravel.
3. **Métodos CRUD:**
 - `index()`: Lista todos los registros.
 - `store()`: Crea un nuevo registro en la base de datos.
 - `show()`: Muestra un registro específico por su ID.
 - `update()`: Actualiza un registro específico por su ID.
 - `destroy()`: Elimina un registro por su ID.

Arquitectura de Software

Arquitectura Monolítica: Es una arquitectura en la que todos los componentes del sistema están integrados en una única aplicación o código base. Todo el sistema está en un solo proyecto (backend + frontend juntos).

Ventajas:

- Sencillez en el desarrollo y despliegue.
- Facilidad para pruebas iniciales.

Desventajas:

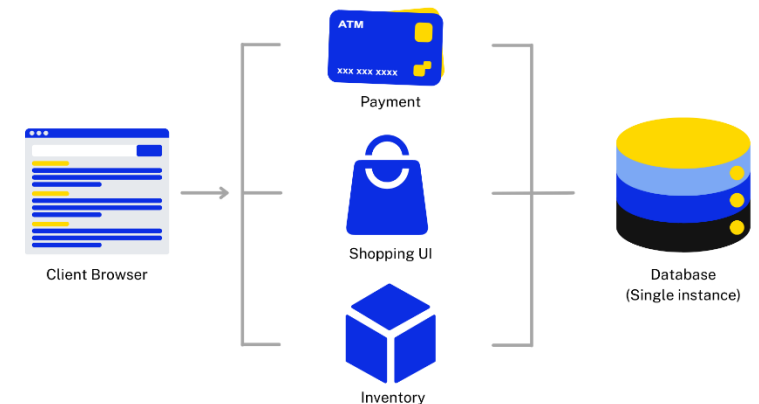
- Difícil de escalar.
- Riesgo de fallo único (si falla una parte, afecta todo).
- Complejidad a medida que crece.

Usos comunes:

- Aplicaciones simples o con un número limitado de usuarios.
- Sistemas internos o de corto plazo.



Monolithic Architecture



Arquitectura de Software

Arquitectura Backend + Frontend Separados: Separación de las responsabilidades entre la lógica de negocio (backend) y la interfaz de usuario (frontend), conectados a través de APIs.

Ventajas:

- Los desarrolladores frontend y backend trabajan de manera independiente.
- Cada parte puede escalarse de manera separada.

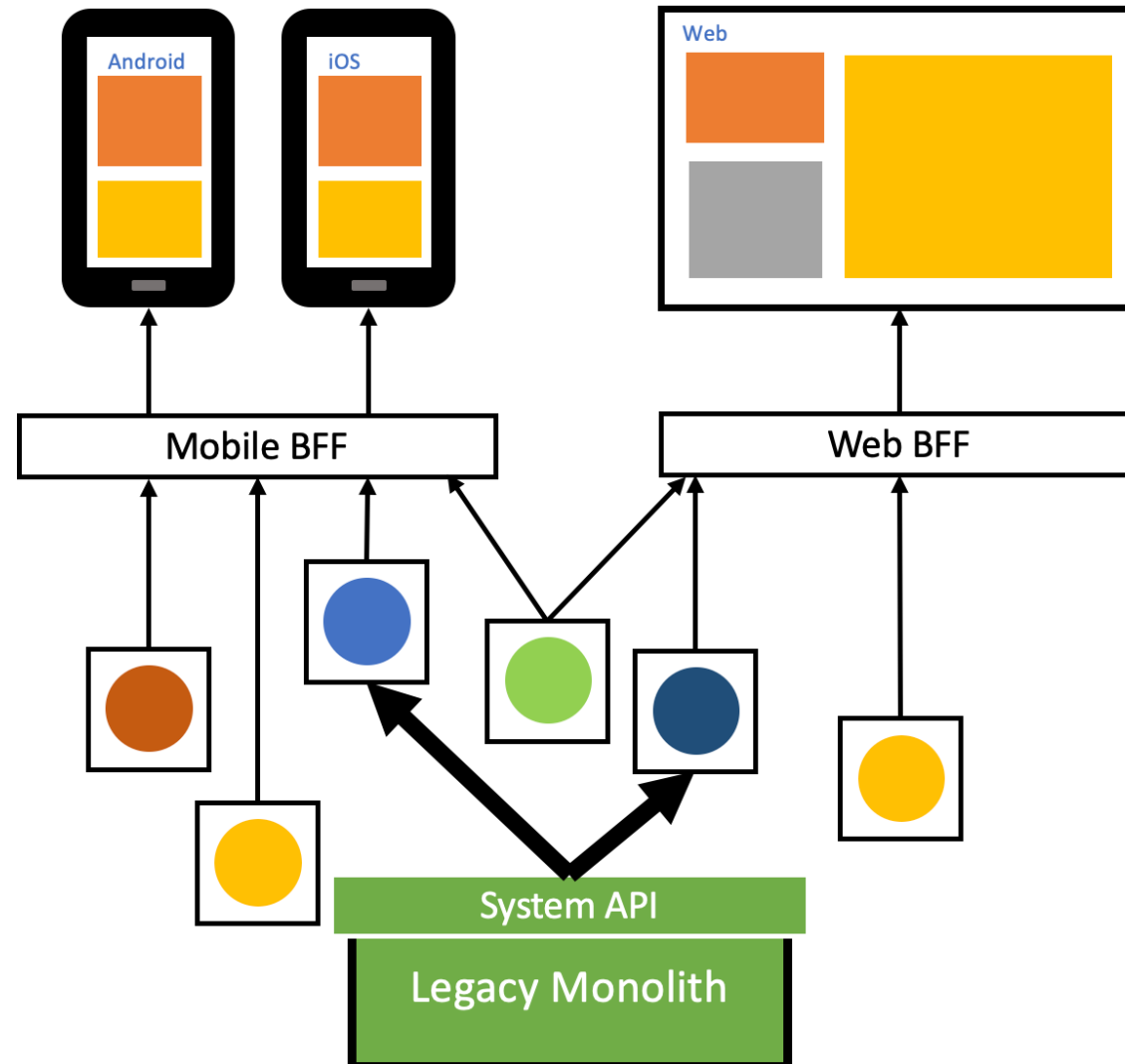
Desventajas:

- Más configuraciones y manejo de APIs.
- La comunicación entre frontend y backend puede agregar demoras.

Usos comunes:

- Aplicaciones web modernas como tiendas en línea o sistemas de gestión.

Arquitectura de Software



Arquitectura de Software

Arquitectura Microservicios: El sistema se divide en pequeños servicios independientes.

Ventajas:

- Escalabilidad horizontal.
- Despliegue independiente de cada servicio.

Desventajas:

- Mayor complejidad en la comunicación entre servicios.
- Normalmente requiere de tecnologías basada en contenedores o virtualización.

Comparación entre Monolito, Backend + Frontend y Microservicios

Característica	Monolito	Backend + Frontend	Microservicios
Simplicidad	Alta	Media	Baja
Escalabilidad	Baja	Media	Alta
Tolerancia a fallos	Baja	Media	Alta
Despliegue	Sencillo	Moderado	Complejo

Introducción a la Gestión de Microservicios

INGENIERÍA DE SOFTWARE I

Albeiro Montes



UNIVERSIDAD
NACIONAL
DE COLOMBIA



Grupo de
Ambientes
Inteligentes
Adaptativos

Gestión de Microservicios

Beneficios y Retos de la Gestión de Microservicios:

La gestión de microservicios implica administrar y coordinar múltiples servicios independientes que trabajan juntos para formar una aplicación completa.

Beneficios de una gestión centralizada:

- **Escalabilidad:** Facilita el manejo de múltiples servicios simultáneamente.
- **Seguridad:** Implementa autenticación y autorización de manera uniforme.
- **Simplificación:** Agiliza la interacción cliente-servidor al consolidar solicitudes.
- **Supervisión:** Centraliza el monitoreo y los registros de actividad.

La gestión de microservicios implica una serie de prácticas, herramientas y enfoques para:

1. **Coordinar servicios independientes.**
2. **Garantizar la comunicación eficiente** entre ellos.
3. **Proveer escalabilidad, disponibilidad y confiabilidad** en todo el sistema.

Cada microservicio es autónomo y se centra en una única funcionalidad o proceso de negocio, implementado de forma independiente.

Gestión de Microservicios

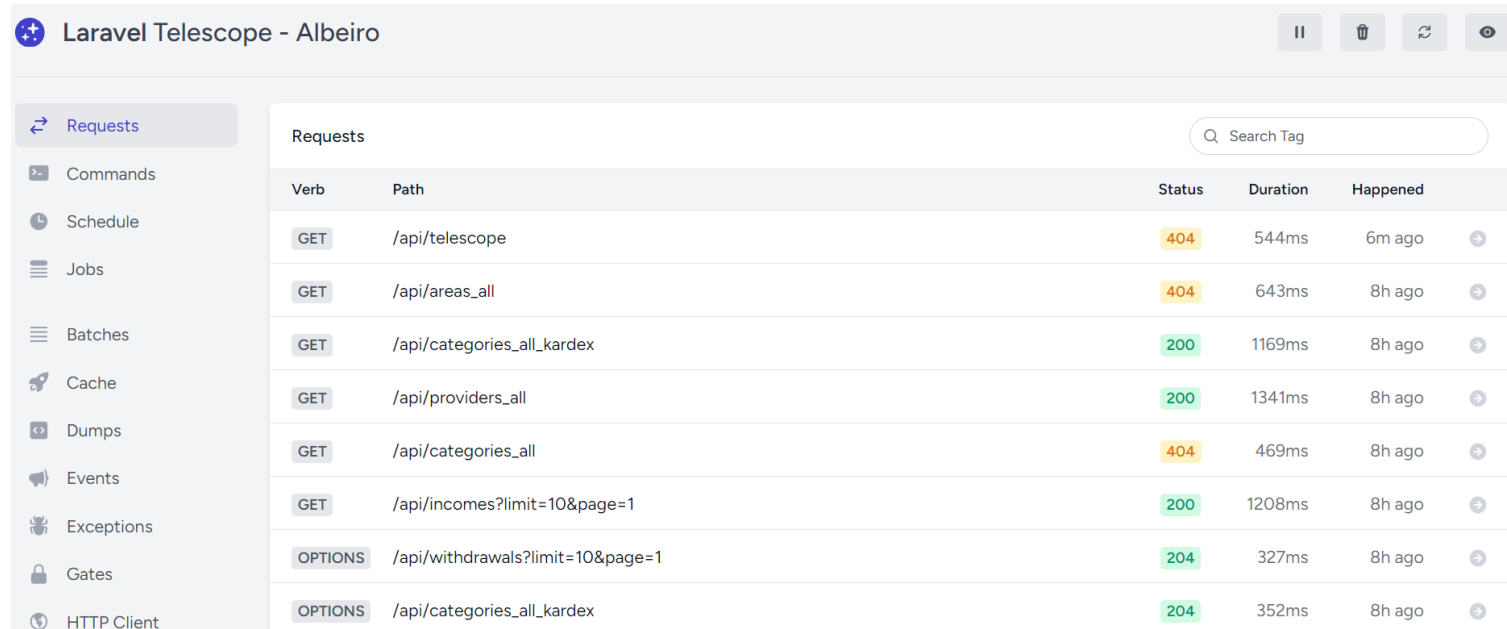
Principios de diseño y gestión

- **Independencia de servicios:**
 - Cada microservicio debe desarrollarse, desplegarse y escalarse de forma independiente.
 - Esto permite que los equipos trabajen en paralelo sin bloquearse.
- **Descentralización:**
 - Cada servicio tiene su lógica y datos independientes.
 - No existe una base de datos monolítica; cada microservicio puede usar una base de datos o tecnología adecuada a su necesidad.
- **Comunicación ligera:**
 - Usar protocolos simples (como HTTP/REST o mensajería asíncrona) para minimizar la complejidad.
- **Resiliencia y tolerancia a fallos:**
 - Los microservicios deben diseñarse para continuar funcionando incluso cuando otros servicios fallen.

Gestión de Microservicios

Herramientas para ayudar en la gestión:

- **Laravel Telescope:**
 - **Registro de solicitudes HTTP:** Muestra información sobre todas las solicitudes entrantes, como la URL, método, estado, tiempo de respuesta, encabezados y cuerpo.
 - **Consultas a la base de datos:** Rastrea todas las consultas SQL realizadas por la aplicación, incluyendo el tiempo de ejecución y el lugar donde se ejecutaron.
 - **Registro de logs:** Visualiza los mensajes de log generados por la aplicación.



The screenshot shows the Laravel Telescope interface for a user named 'Albeiro'. The left sidebar contains navigation links for Requests, Commands, Schedule, Jobs, Batches, Cache, Dumps, Events, Exceptions, Gates, and HTTP Client. The 'Requests' tab is selected, displaying a table of incoming HTTP requests. The table has columns for Verb, Path, Status, Duration, and Happened. A search bar is located at the top right of the table.

Verb	Path	Status	Duration	Happened
GET	/api/telescope	404	544ms	6m ago
GET	/api/areas_all	404	643ms	8h ago
GET	/api/categories_all_kardex	200	1169ms	8h ago
GET	/api/providers_all	200	1341ms	8h ago
GET	/api/categories_all	404	469ms	8h ago
GET	/api/incomes?limit=10&page=1	200	1208ms	8h ago
OPTIONS	/api/withdrawals?limit=10&page=1	204	327ms	8h ago
OPTIONS	/api/categories_all_kardex	204	352ms	8h ago

Gestión de Microservicios

Componentes clave en la gestión

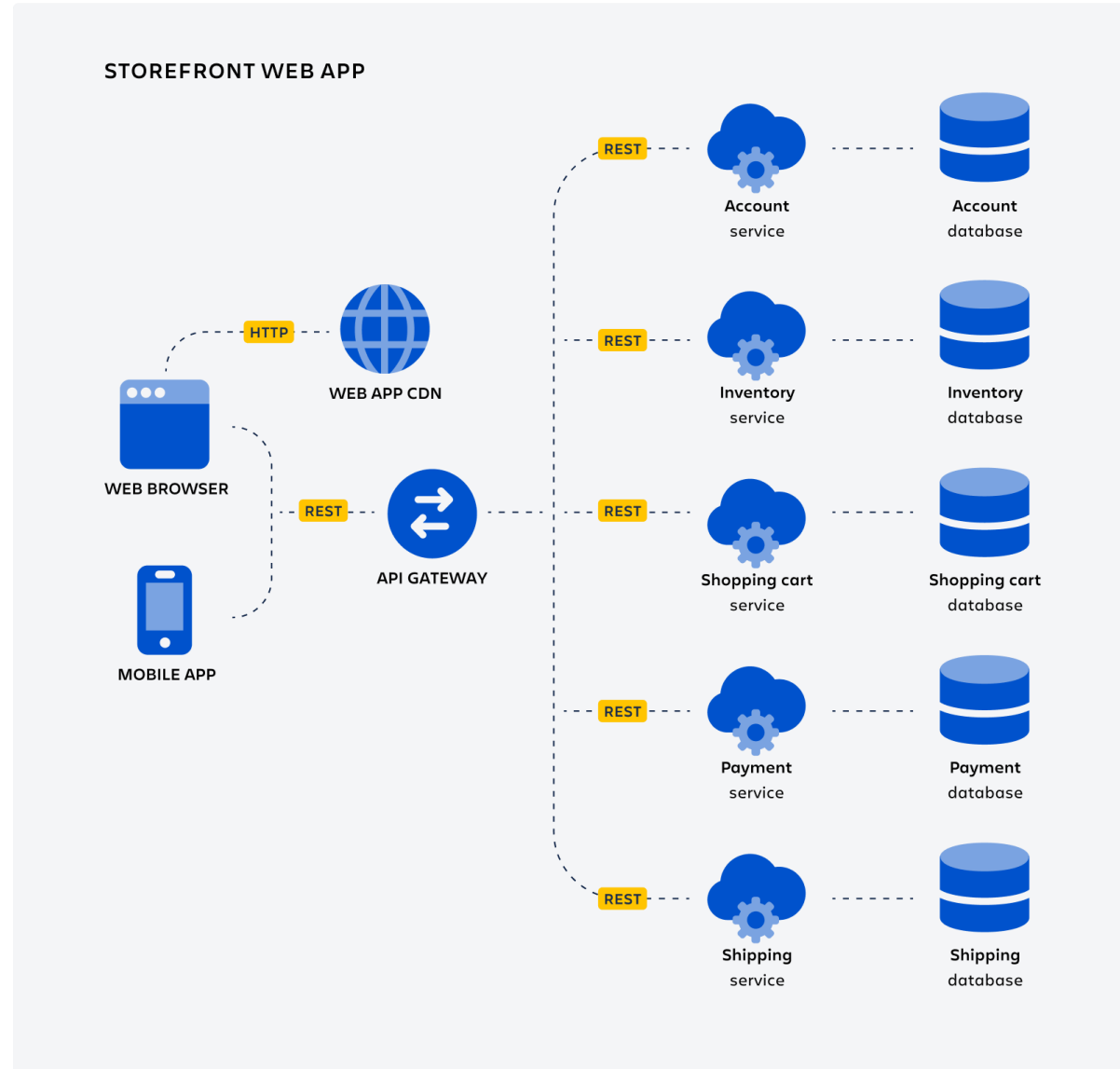
API Gateway: Actúa como un punto de entrada para las solicitudes de los clientes.

- Autenticación.
- Agregado de datos desde múltiples microservicios.
- Control de tráfico.

Registro de servicios:

- Permite que los microservicios encuentren y se comuniquen entre sí dinámicamente.

Gestión de Microservicios



Gestión de Microservicios

Construyendo una API Gateway para gestionar microservicios.

Paso 1:

Generar los endpoints para hacer la petición a los microservicios en el Gateway.

Paso 2:

Generar la protección de los endpoints en los microservicios para que reciban peticiones exclusivamente del API Gateway.

Paso 3:

Configurar la petición en el Gateway para los microservicios.

Gestión de Microservicios

Construyendo una API Gateway para gestionar microservicios.

use Illuminate\Support\Facades\Http;

class ApiController extends Controller

```
{  
    //Accede a la propiedad protegida de la clase  
    ($apiUrl).  
    protected $apiUrl;  
  
    //Método constructor de la clase, el cual se ejecuta  
    cuando se genera una instancia de la clase. Se define  
    la propiedad  
    $apiUrl.  
    public function __construct()  
    {  
        $this->apiUrl = env('MICROSERVICIO_URL');  
    }  
}
```

Métodos que contienen la lógica...

```
}
```

```
public function index()  
{  
    $url = $this->apiUrl . '/products/';  
    $response = Http::get($url);  
    return $response->json();  
}  
public function show($id)  
{  
    $url = $this->apiUrl . '/products/'. $id;  
    $response = Http::get($url);  
    return $response->json();  
}  
public function store(Request $request)  
{  
    $url = $this->apiUrl . '/products/';  
    $response = Http::post($url, $request->all());  
    return $response->json();  
}  
public function update($id)  
{  
    $url = $this->apiUrl . '/products/'. $id;  
    $response = Http::put($url, $request->all());  
    return $response->json();  
}  
public function destroy($id)  
{  
    $url = $this->apiUrl . '/products/'. $id;  
    $response = Http::delete($url);  
    return $response->json();  
}
```

Gestión de Microservicios

Construyendo una API Gateway para gestionar microservicios.

Protección en el microservicio con Middleware.

```
public function handle(Request $request, Closure $next)  
{  
    $apiKey = $request->header('X-API-Key');  
    $expectedApiKey = env('API_KARDEX_KEY');  
    if ($apiKey !== $expectedApiKey) {  
        return response()->json([  
            'message' => 'Acceso denegado. Clave API incorrecta.'  
        ], 403);  
    }  
    return $next($request);  
}
```

Luego, se debe registrar el middleware en Kernel.php

```
Route::middleware(['verify_gateway_pass'])->group(function () {  
    ...Rutas  
});
```

Gestión de Microservicios

Construyendo una API Gateway para gestionar microservicios.

Modificar el método en el Gateway para enviar la API_Key al microservicio.

```
public function index()
{
    $url = $this->apiUrl . '/providers/';
    $response = Http::withHeaders(['X-API-Key' => $this->apiKey])->get($url);
    return $response->json();
}

public function show($id)
{
    $url = $this->apiUrl . '/products/'. $id;
    $response = Http::withHeaders(['X-API-Key' => $this->apiKey])->get($url);
    return $response->json();
}

public function store(Request $request)
{
    $url = $this->apiUrl . '/products/';
    $response = Http::withHeaders(['X-API-Key' => $this->apiKey])->post($url, $request->all());
    return $response->json();
}

public function update(Request $request, $id)
{
    $url = $this->apiUrl . '/products/';
    $response = Http::withHeaders(['X-API-Key' => $this->apiKey])->patch($url, $request->all());
    return $response->json();
}
```

Gestión de Microservicios

Construyendo una API Gateway para gestionar microservicios.

Modificar el método en el Gateway para enviar la API_Key al microservicio.

```
public function destroy()
{
    $url = $this->apiUrl . '/providers/';
    $response = Http::withHeaders(['X-API-Key' => $this->apiKey])->delete($url);
    return $response->json();
}
```

Introducción a Pruebas de Rendimiento

INGENIERÍA DE SOFTWARE I

Albeiro Montes



UNIVERSIDAD
NACIONAL
DE COLOMBIA



Grupo de
Ambientes
Inteligentes
Adaptativos

Introducción a Pruebas de Rendimiento

Pruebas de Rendimiento:

- Las pruebas de rendimiento son un tipo de prueba de rendimiento cuyo objetivo es determinar el comportamiento de un sistema bajo condiciones extremas.
- Se realizan con el objetivo de identificar los límites de rendimiento del sistema, detectar fallos en momentos de alta carga y mejorar la infraestructura.

Objetivos:

- Determinar el punto de quiebre del sistema.
- Ver cómo el sistema maneja la alta carga o el tráfico excesivo.
- Identificar posibles cuellos de botella y problemas de rendimiento.
- Garantizar que el sistema se recupere correctamente tras un fallo

Introducción a Pruebas de Rendimiento

1) Pruebas de Carga

Evaluar cómo el sistema se comporta bajo una carga esperada o normal, es decir, el tráfico que se espera regularmente durante el uso normal del sistema.

Descripción:

En una **prueba de carga**, el objetivo es simular el número de usuarios o solicitudes que se espera que el sistema maneje bajo condiciones normales de operación.

Ejemplo: Si se espera que una API tenga 100 usuarios concurrentes utilizando el sistema al mismo tiempo, una prueba de carga simularía esos 100 usuarios realizando tareas típicas como navegar por páginas, hacer búsquedas, etc.

Resultados esperados:

- **Tiempos de respuesta:** Se mide el tiempo promedio de respuesta de las solicitudes bajo carga normal.
- **Uso de recursos:** Se monitorea el uso de CPU, memoria y otros recursos del servidor bajo la carga esperada.
- **Comportamiento general:** Se asegura de que el sistema pueda manejar el tráfico esperado sin fallos o caídas.

Introducción a Pruebas de Rendimiento

2) Pruebas de Estrés

Determinar el comportamiento del sistema cuando se somete a condiciones extremas de carga, mucho más allá de lo esperado en un entorno de producción. La prueba de estrés va más allá de la carga normal y se utiliza para llevar al sistema hasta su límite de capacidad para observar cómo reacciona.

Resultados esperados:

- **Punto de quiebre:** Se identifica el momento en que el sistema comienza a fallar o experimentar un rendimiento deficiente.
- **Comportamiento bajo presión:** Se analiza cómo el sistema maneja la sobrecarga. Por ejemplo, ¿se detiene el sistema de forma inesperada? ¿Los usuarios reciben respuestas de error?
- **Recuperación:** Se evalúa cómo el sistema se recupera una vez que se reduce la carga. Idealmente, debería volver a la normalidad una vez que la sobrecarga termine.

Introducción a Pruebas de Rendimiento

3) Pruebas de Capacidad

Determinar la capacidad máxima que el sistema puede manejar en términos de usuarios concurrentes o solicitudes por segundo sin que el rendimiento se vea afectado. Las pruebas de capacidad están diseñadas para **medir el límite máximo** de recursos que puede soportar una aplicación sin que su rendimiento se degrade.

Resultados esperados:

- Número máximo de usuarios o solicitudes que el sistema puede manejar sin que el rendimiento se vea afectado de forma significativa.
 - Uso de recursos en el punto máximo de capacidad, para determinar qué recursos (memoria, CPU, red, etc.) son los que limitan la capacidad del sistema.
 - locust -f file.py --users 200 --spawn-rate 20
 - locust -f file.py --users 500 --spawn-rate 50
 - locust -f file.py --users 1000 --spawn-rate 100
- users 100: Inicialmente simula 100 usuarios.--spawn-rate 10: Los usuarios se agregan a un ritmo de 10 usuarios por segundo.

Introducción a Pruebas de Rendimiento

Característica	Prueba de Carga	Prueba de Estrés	Prueba de Capacidad
Objetivo principal	Evaluar el rendimiento del sistema con una carga normal y esperada.	Evaluar cómo el sistema responde a condiciones extremas (más allá de su capacidad).	Determinar la capacidad máxima del sistema antes de que su rendimiento se degrade.
Condición de prueba	Simula el número de usuarios o tráfico esperado en un escenario normal.	Simula condiciones extremas, como una gran cantidad de usuarios, solicitudes o datos.	Aumenta gradualmente la carga para identificar el número máximo de usuarios que el sistema puede manejar.
Punto de prueba	Verifica la respuesta y el comportamiento del sistema bajo una carga normal.	Lleva el sistema al límite de sus capacidades, más allá de lo esperado.	Encuentra el punto donde el rendimiento comienza a degradarse o el sistema falla debido a la saturación de recursos.
Comportamiento esperado	El sistema debería funcionar correctamente sin caídas ni errores.	El sistema probablemente fallará o mostrará errores, pero se mide cómo maneja esta caída.	El sistema debería manejar un número progresivamente mayor de usuarios hasta que se acerque a su límite de capacidad.
Enfoque de prueba	Establecer la capacidad de respuesta, tiempos de carga y rendimiento con la carga prevista.	Medir cómo se comporta el sistema cuando se somete a un uso extremo, como picos de tráfico o estrés.	Identificar el umbral máximo de usuarios o solicitudes que el sistema puede manejar antes de que se produzcan fallos.

Seguridad de API's

INGENIERÍA DE SOFTWARE I

Albeiro Montes



UNIVERSIDAD
NACIONAL
DE COLOMBIA



Grupo de
Ambientes
Inteligentes
Adaptativos

Seguridad de API's

JWT (JSON Web Token) es un estándar abierto (RFC 7519) que define una forma compacta y segura de transmitir información como un objeto JSON entre partes. Es ampliamente utilizado en sistemas de autenticación y autorización para garantizar la identidad de los usuarios y permitir el acceso a recursos protegidos.

Header (Encabezado): Contiene metadatos sobre el token, como el tipo de token (JWT) y el algoritmo de firma.

Payload (Carga útil): Contiene los datos que se transmiten. Estos datos pueden incluir: Información sobre el usuario (como id, email, roles). **Claims (declaraciones):**

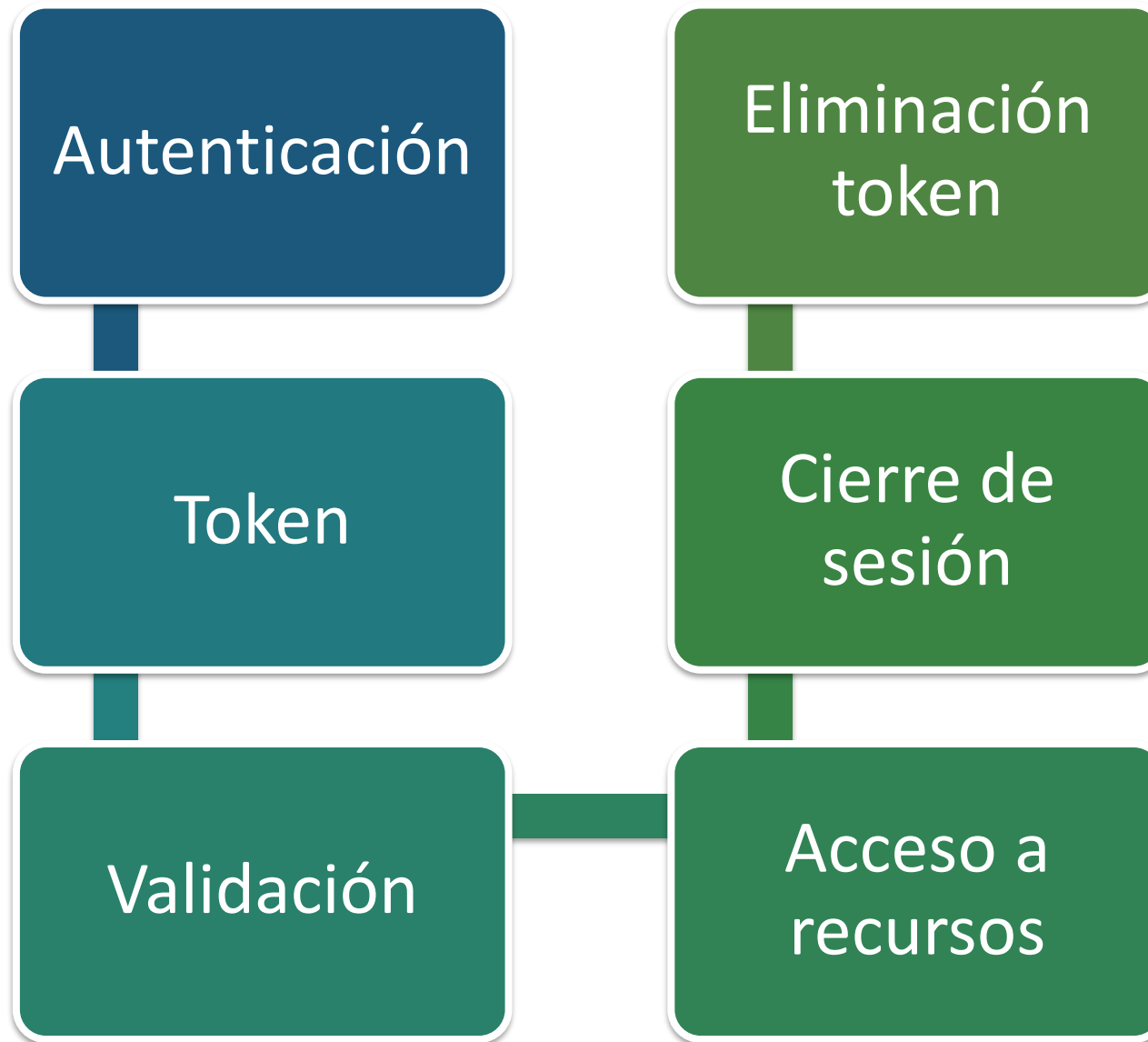
- iss (Issuer): Quién emitió el token.
- exp (Expiration): Fecha y hora de expiración.
- iat (Issued At): Cuándo fue creado el token.

Signature (Firma): Garantiza la integridad del token y verifica que no ha sido manipulado. Se genera combinando:

- El encabezado codificado en base64.
- La carga útil codificada en base64.
- Una clave secreta o privada usando un algoritmo de firma, como HMAC SHA256.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjMNTY3ODkwIiwibmFtZSI6Ikp1YW4gUC4iLCJhZG1pbil6dHJ1ZX0.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Seguridad de API's



Seguridad de API's

Configurar JWT en el Gateway:

Paso 1: *composer require tymon/jwt-auth*

Paso 2: *php artisan vendor:publish --provider="Tymon\JWTAuth\Providers\LaravelServiceProvider"*

Paso 3: *php artisan jwt:secret*

Paso 4: *Configurar el driver en config/auth.php*

```
'guards' => [  
    'web' => [  
        'driver' => 'session',  
        'provider' => 'users',  
    ],  
    'api' => [  
        'driver' => 'jwt',  
        'provider' => 'users',  
    ],  
],
```

Paso 5: *Crear controlador para manejar lógica de seguridad (php artisan make:controller AuthController)*

Seguridad de API's

Configurar JWT en el Gateway:

Paso 6: *Crear los métodos de login, logout y register.*

```
use Tymon\JWTAuth\Facades\JWTAuth;
```

```
public function login(Request $request)
{
    $credentials = $request->only('email', 'password');
    if (!$token = JWTAuth::attempt($credentials)) {
        return response()->json(['error' => 'Credenciales inválidas'], 401);
    }
    return response()->json([
        'token' => $token,
        'user' => auth()->user()
    ]);
}
```

Seguridad de API's

Configurar JWT en el Gateway:

Paso 6: *Crear los métodos de login, logout y register.*

```
public function register(Request $request)
{
    $user = User::create([
        'name' => $request->name,
        'email' => $request->email,
        'password' => bcrypt($request->password)
    ]);
    return response()->json([
        'message' => 'Usuario creado con éxito',
        'user' => $user
    ]);
}
```


Seguridad de API's

Configurar JWT en el Gateway:

Paso 6: *Crear los métodos de login, logout y register.*

```
public function logout()  
{  
  JWTAuth::invalidate(JWTAuth::getToken());  
  return response()->json(['message' => 'Sesión cerrada con éxito']);  
}
```

Seguridad de API's

Configurar JWT en el Gateway:

Paso 7: *Implementar la interfaz del JWT en el modelo de User.*

```
use Illuminate\Foundation\Auth\User as Authenticatable;
use Tymon\JWTAuth\Contracts\JWTSubject;
class User extends Authenticatable implements JWTSubject
{
    public function getJWTIdentifier()
    {
        return $this->getKey();
    }
    public function getJWTCustomClaims()
    {
        return [];
    }
}
```

Seguridad de API's

Configurar JWT en el Gateway:

Paso 8: *Generar las rutas.*

```
Route::post('login', [AuthController::class, 'login']);
Route::post('register', [AuthController::class, 'register']);

Route::middleware('auth:api')->group(function () {
    Route::post('logout', [AuthController::class, 'logout']);
    Route::get('/', function () {
        return 3;
    });
});
```

Seguridad de API's

Asignar roles a usuarios

Paso 1: *Crear la migración de roles y el modelo*
`php artisan make:model Role -m`

Paso 2: *Crear campos en la migración de roles*

```
Schema::create('roles', function (Blueprint $table)
{
    $table->id();
    $table->string('label');
    $table->string('name');
    $table->timestamps();
});
```

Paso 3: *Crear migración para editar la migración de usuarios*

```
php artisan make:migration add_role_id_to_users_table --table=users
```

Seguridad de API's

Asignar roles a usuarios

Paso 3: *Añadir los campos a la nueva migración*

```
public function up(): void
{
    Schema::table('users', function (Blueprint $table) {
        $table->unsignedBigInteger('role_id')->nullable()->after('id');
        $table->foreign('role_id')->references('id')->on('roles')->onDelete('set null');
    });
}
```

```
public function down(): void
{
    Schema::table('users', function (Blueprint $table) {
        $table->dropForeign(['role_id']);
        $table->dropColumn('role_id');
    });
}
```

Seguridad de API's

Paso 4: *Crear Middleware Role*

```
public function handle(Request $request, Closure $next, $roles)
{
    $rol = explode('|', $roles);
    $roleName = $request->user()->role->label;
    if (!in_array($roleName, $rol)) {
        return response()->json(['error' => 'No tienes permisos para acceder a esta ruta'], 401);
    }
    return $next($request);
}
```

Paso 5: *Publicar el Middleware (kernel.php)*

```
protected $middlewareAliases =
[
    'role' => \App\Http\Middleware\Role::class,
];
```

Seguridad de API's

Paso 6: *Definir relación entre usuarios y roles*

En el modelo de User

```
public function role()  
{  
    return $this->belongsTo(Role::class);  
}
```

Un usuario pertenece a un rol, porque un usuario sólo puede tener un rol.

En el modelo de Role

```
public function users()  
{  
    return $this->hasMany(User::class);  
}
```

Un rol puede estar asignado a varios usuarios.

Seguridad de API's

Paso 7: Definir rutas

Rutas públicas:

```
Route::post('login', [AuthController::class, 'login']);  
Route::post('register', [AuthController::class, 'register']);
```

Rutas privadas para todos:

```
Route::middleware(['auth:api', 'role:admin|docente'])->group(function ()  
{  
    Route::post('logout', [AuthController::class, 'logout']);  
});
```

Rutas privadas para admin:

```
Route::middleware(['auth:api', 'role:admin'])->group(function ()  
{  
    Route::get('/admin', function () {  
        return "admin";  
    });  
});
```

Rutas privadas para docente:

```
Route::middleware(['auth:api', 'role:docente'])->group(function ()  
{  
    Route::get('/docente', function () {  
        return "docente";  
    });  
});
```