Taller de arquitectura y patrones

Programacion II

Docente:

David Rodriguez

Integrantes:

Jhon Emerson

Pérez Ramírez

Santiago Casas

David Esteban Morales

Rebeca Pedrozo

Universidad Libre

Bogotá D.C

Introducción

En el presente informe se abordan diversas arquitecturas de software aplicadas a escenarios reales del desarrollo de sistemas modernos. Cada punto expone un enfoque arquitectónico diferente desde microservicios hasta arquitecturas orientadas a eventos o serverless con el objetivo de demostrar cómo cada modelo puede adaptarse a distintas necesidades tecnológicas y de negocio. Este trabajo busca no solo mostrar la estructura técnica de cada arquitectura, sino también sus ventajas, componentes principales, y ejemplos aplicados, permitiendo así una comprensión integral de cómo diseñar soluciones escalables, mantenibles y eficientes en distintos contextos del desarrollo de software.

1. PARTE 1

Para la parte 1 de la tarea se ha propuesto la explicación teórica y ejercicios en código que se pueden encontrar en: https://github.com/rebeca07-pedrozo/arquitectura_y_patrones

Rama: main

Arquitectura Basada en Microservicios para una Aplicación de Pedidos a Domicilio

La arquitectura de microservicios divide una aplicación en componentes pequeños, independientes y desplegables por separado. Cada servicio se enfoca en una funcionalidad específica del negocio y se comunica con otros mediante APIs REST o colas de mensajería como Kafka o RabbitMQ.

Objetivo principal: lograr escalabilidad, facilidad de mantenimiento, independencia en el desarrollo y despliegue continuo de cada módulo.

Propuesta de división en servicios:

Servicio	Funcionalidad Principal
Usuarios	Registro, login, recuperación de contraseña, actualización de perfil, validación de identidad.
Pedidos	Creación de pedidos, gestión del estado (en preparación, en camino, entregado), historial y cancelación de pedidos.
Repartidores	Registro, ubicación en tiempo real, asignación de pedidos, historial de entregas.
Productos	Administración de catálogo, precios, disponibilidad, descripción y promociones.

Cada uno tendría su base de datos independiente para asegurar el principio de autonomía, permitiendo también que se escalen horizontalmente según la demanda (por ejemplo, durante promociones).

Transformación de un Sistema Monolítico de Reservas de Hoteles a Microservicios

Un sistema monolítico de reservas de hoteles agrupa toda la lógica y datos en una sola aplicación. Esto genera cuellos de botella y dificulta escalar partes individuales del sistema.

Ventajas de migrar a microservicios:

- Escalabilidad independiente (por ejemplo, más tráfico en reservas).
- Mejora la mantenibilidad del código.
- Desarrollo distribuido por equipos.

Servicios que se pueden extraer:

Microservicio	Funciones
Servicio de Usuarios	Registro, autenticación, roles, historial.
Servicio de Hoteles	Listado de hoteles, detalles, habitaciones, ubicación.
Servicio de Reservas	Crear, modificar, cancelar y consultar reservas.
Servicio de Pagos	Procesamiento de tarjetas, verificación y reembolsos.

Servicio de Opiniones	Recolección y visualización de reseñas.

La comunicación entre servicios se realiza mediante RESTful APIs, y cada servicio tiene su propia base de datos (por ejemplo, PostgreSQL para reservas y MongoDB para opiniones).

Arquitectura MVC para Aplicación de Tareas Colaborativas (con Django)

El patrón MVC (Modelo-Vista-Controlador) separa la lógica de la aplicación en capas, promoviendo el orden y la reutilización de componentes. En Django, el patrón se adapta como MVT (Modelo-Vista-Template).

Caso de uso: Aplicación para gestión de tareas en equipos.

Componentes:

- **Modelo:** Define las entidades: Usuario, Proyecto, Tarea, Comentarios, Prioridades.
- **Vista:** Recibe solicitudes del usuario, valida y recupera datos del modelo, retornando la vista correspondiente.
- **Template:** Renderiza la información visualmente usando HTML, CSS y JS.

Ventajas:

- Separación clara entre lógica y presentación.
- Escalabilidad para incorporar notificaciones, calendarios o integración con APIs externas (por ejemplo, Google Calendar).

Sistema Serverless para Comentarios de Clientes

Una arquitectura serverless elimina la necesidad de gestionar servidores físicos o virtuales. AWS Lambda, combinada con servicios como DynamoDB y API Gateway, permite crear soluciones completamente escalables y bajo demanda.

Arquitectura propuesta:

Componente	Descripción
API Gateway	Recibe las solicitudes HTTP.
AWS Lambda	Procesa los comentarios y los valida.

DynamoDB	Almacena los comentarios como documentos JSON.
S3 (opcional)	Almacenamiento de archivos adjuntos o imágenes.

Características:

- Pago por uso.
- Escalabilidad automática.
- Ideal para picos de carga (por ejemplo, campañas de marketing).

Arquitectura Cliente-Servidor para Gestión de Notas Universitarias

Este sistema permite a los profesores gestionar las calificaciones de los estudiantes, registrar nuevos alumnos y generar estadísticas académicas. Los estudiantes, por su parte, pueden consultar sus notas de manera segura a través de una interfaz web.
La solución se implementa bajo una arquitectura cliente-servidor. El servidor administra los datos y expone una API RESTful, mientras que el cliente interactúa directamente con el usuario final, realizando solicitudes al servidor para mostrar información y procesar acciones.

Componentes del Sistema

Cliente (Frontend)

Elemento	Detalles
Tecnologías	HTML, CSS, JavaScript, React o Angular
Funcionalidades	- Inicio de sesión (login) - Visualización de calificaciones - Formularios para agregar o editar notas (profesores) - Interfaz para visualizar estadísticas académicas (opcional)

Servidor (Backend)

Elemento	Detalles
Tecnologías	Java con Spring Boot, Node.js con Express, o Python con Flask

Funcionalidades	- Autenticación y autorización de usuarios - Gestión de estudiantes, cursos y notas - Exposición de una API RESTful para la comunicación con el cliente

Base de Datos

Elemento	Detalles
Tecnologías	MySQL o PostgreSQL
Estructura	- **Estudiantes:** datos personales - **Cursos:** información académica - **Notas:** calificaciones por curso
Relaciones	- Un estudiante puede tener muchas notas - Un curso puede tener muchas notas - Un estudiante puede estar inscrito en varios cursos

Seguridad

Elemento	Descripción
Autenticación	Uso de JWT (JSON Web Token) para proteger los endpoints
Control de acceso	Basado en roles (profesor o estudiante)

Flujo de Trabajo

Inicio de sesión

Paso	Acción
1	El cliente envía una solicitud con nombre de usuario y contraseña
2	El servidor valida las credenciales y responde con un token JWT

Gestión de Notas

Rol	Permisos

Profesor	Agregar, editar o eliminar notas Visualizar la lista de estudiantes y sus calificaciones
Estudiante	Consultar únicamente sus propias notas

Endpoints de la API RESTful

Método	Endpoint	Descripción	Acceso
POST	`/login`	Autenticar usuarios	Público
GET	`/students`	Obtener lista de estudiantes	Solo profesores
POST	`/students`	Agregar nuevos estudiantes	Solo profesores
GET	`/grades/{studentId}`	Obtener notas de un estudiante	Estudiantes y profesores
POST	`/grades/{studentId}`	Agregar o modificar notas	Solo profesores
GET	`/statistics`	Obtener estadísticas académicas	Solo profesores (opcional)

Tecnologías y Herramientas

Componente	Tecnologías / Herramientas
Frontend	HTML, CSS, JavaScript, React o Angular
Backend	Java (Spring Boot), Node.js (Express), Python (Flask)
Base de datos	MySQL o PostgreSQL
Desarrollo	IntelliJ IDEA (Java), Visual Studio Code (JavaScript/Python)
Seguridad	Autenticación con JWT (JSON Web Token)

Arquitectura Hexagonal para App E-commerce

La arquitectura hexagonal, también conocida como "puertos y adaptadores", promueve una aplicación desacoplada de sus interfaces externas. El núcleo de negocio se mantiene aislado y se conecta al resto del mundo mediante interfaces definidas (puertos).

Ejemplo de una app e-commerce:

Puerto (interfaz)	Adaptador (implementación)
Servicio de Productos	Base de datos MySQL.
Servicio de Pago	Integración con Stripe o PayPal.
Notificaciones	Servicio de emails (SMTP, Sendgrid).
API REST	Controladores en Spring Boot o Express.js.

Beneficios:

- Aislamiento del dominio.
- Flexibilidad para pruebas unitarias.
- Cambio de proveedor externo sin modificar la lógica interna.

Arquitectura Orientada a Eventos para Notificaciones de Viajes

Este tipo de arquitectura es ideal para apps de movilidad como Uber, ya que permite reaccionar de forma asíncrona ante eventos. Cuando un conductor acepta un viaje, se lanza un evento que puede ser consumido por múltiples servicios.

Componentes clave:

Componente	Función
Productor	Emite el evento de "viaje aceptado".
Bus de eventos (Kafka)	Distribuye el evento.
Servicio de Notificación	Envía mensaje push o email.
Servicio de Monitoreo	Registra la actividad para análisis posteriores.

Ventajas:

- Escalabilidad.
- Comunicación desacoplada.
- Mejora de experiencia en tiempo real.

Componente de Autenticación con Arquitectura en Capas

La arquitectura en capas separa responsabilidades y mejora la mantenibilidad del código. Para un sistema de autenticación:

Capa	Función
Presentación	Captura de datos de login, UI responsiva.
Lógica de Negocio	Validación de credenciales, generación y validación de JWT.
Acceso a Datos	Consultas a la base de datos de usuarios, roles y permisos.

Tecnologías sugeridas:

- React (presentación)
- Express.js (lógica)
- MongoDB/PostgreSQL (datos)

Mejoras: uso de tokens JWT, validación de sesión, y OAuth para redes sociales.

Arquitectura con Colas Asincrónicas para Correos Masivos

Para evitar que el backend se bloquee al enviar miles de correos, se puede usar una arquitectura basada en colas.

Flujo general:

Componente	Función
Cliente	Solicita el envío masivo.
API	Encola los correos.

RabbitMQ o Kafka	Almacena mensajes temporalmente.
Worker	Desencola y envía los correos.
SMTP	Protocolo de envío final.

Ventajas:

- Reducción del tiempo de respuesta del servidor.
- Reintentos automáticos en caso de errores.
- Escalado horizontal del sistema de envíos.

Despliegue con Docker Compose para Tres Microservicios

Docker Compose permite definir y ejecutar múltiples contenedores como un conjunto de servicios. Es ideal para simular un entorno de producción local.

Microservicios:

- `usuario`: Node.js con Express, puerto 3000.
- `producto`: Flask en Python, puerto 5000.
- `notificacion`: Go, servidor HTTP, puerto 8080.

Archivo docker-compose.yml

```
version: '3'
services:
  usuario:
    build: ./usuario
    ports:
      - "3000:3000"
  producto:
    build: ./producto
    ports:
      - "5000:5000"
  notificacion:
    build: ./notificacion
    ports:
      - "8080:8080"
```

Cada servicio puede tener su propia base de datos y ser conectado por una red interna.

2. PARTE 2

Para la parte dos se han propuestos ejercicios con código que se pueden encontrar en https://github.com/rebeca07-pedrozo/arquitectura_y_patrones/tree/parte2

Rama: parte2

3. PARTE 3

Para la parte tres se han propuestos ejercicios con código que se pueden encontrar en https://github.com/rebeca07-pedrozo/arquitectura_y_patrones/tree/parte3

Rama: parte3

4. PARTE 4

Para la parte cuatro se han propuestos ejercicios con código que se pueden encontrar en https://github.com/rebeca07-pedrozo/arquitectura_y_patrones/tree/parte4

Rama: parte4