Reporte Técnico de Actividades Prácticas-Experimentales Nro. 004

1. Datos de Identificación del Estudiante y la Práctica

Nombre del estudiante(s)	Miguel Luna, Anthony Gutiérrez y Luis Armijos
Asignatura	Desarrollo Basado en Plataformas
Ciclo	5 A
Unidad	1
Resultado de aprendizaje de la unidad	
Práctica Nro.	004
Título de la Práctica	Implementar un servicio REST con Node.js
Nombre del Docente	Edison Leonardo Coronel Romero
Fecha	Jueves 23 de octubre
Horario	07h30 – 10h30
Lugar	Laboratorio Computación aplicada Laboratorio Desarrollo de Software Laboratorio de redes y Sistemas Operativos Laboratorio Virtual EVA Aula
Tiempo planificado en el Sílabo	3 horas

2. Objetivo(s) de la Práctica

Diseñar y desplegar un microservicio funcional que se comunique con el backend principal del proyecto.
Implementar comunicación REST entre servicios utilizando un API Gateway o endpoint compartido.
Documentar la arquitectura de microservicios en el modelo C4 y registrar evidencias del despliegue.

3. Materiales, Reactivos, Equipos y Herramientas

Computador con acceso a Internet.

- Docker / Docker Compose o entorno virtual local.
- Repositorio del proyecto (GitHub/GitLab).
- Postman / Swagger.
- Framework backend (Django REST Framework / Express / Spring Boot).
- Laboratorio de Desarrollo de Software o equipo personal.
- IDE: Visual Studio Code / IntelliJ IDEA.
- Git y Git Kraken (flujo GitFlow).
- Terminal de comandos y contenedores Docker.
- Herramientas de modelado C4 (PlantUML, Mermaid, Draw.io).

4. Procedimiento / Metodología Ejecutada

Para la implementación del microservicio se repasaron primeramente los conceptos de esta tecnología y a continuación se procedió con su implementación en el backend como se mostrará en resultados.

La arquitectura de microservicios surge como una evolución de los modelos monolíticos tradicionales, proponiendo el desarrollo de sistemas compuestos por múltiples servicios pequeños, independientes y especializados. Cada microservicio se encarga de una funcionalidad específica y se comunica con los demás mediante interfaces bien definidas, generalmente a través de APIs REST o mensajería asíncrona. Este enfoque permite una mayor escalabilidad, mantenibilidad y despliegue independiente, lo que facilita la actualización y evolución de las aplicaciones sin afectar al sistema completo. Además, promueve la adopción de tecnologías diversas en un mismo entorno, ya que cada microservicio puede desarrollarse con distintos lenguajes o frameworks según su propósito.

Por otro lado, el paradigma serverless, sin servidor, representa un modelo de ejecución en la nube donde el desarrollador no gestiona la infraestructura, sino que se enfoca únicamente en el código y la lógica de negocio. En este modelo, los recursos se asignan de forma automática bajo demanda, y el proveedor de servicios en la nube se encarga de la escalabilidad, el mantenimiento y la disponibilidad. Las funciones serverless, como AWS Lambda, Azure Functions o Google Cloud Functions, se ejecutan de manera efímera y bajo eventos específicos, lo que resulta ideal para sistemas basados en microservicios o aplicaciones que requieren alta elasticidad y reducción de costos operativos.

En cuanto a los patrones de comunicación, estos definen las formas en que los microservicios o funciones serverless interactúan entre sí. Los patrones más comunes incluyen la comunicación síncrona, donde un servicio realiza peticiones directas y espera respuestas inmediatas; por ejemplo, HTTP/REST o gRPC, y la comunicación asíncrona, que se basa en el intercambio de mensajes a través de colas o eventos, por ejemplo, RabbitMQ, Kafka o Amazon SNS/SQS. La elección del patrón adecuado depende de las necesidades del sistema: mientras la comunicación síncrona favorece la simplicidad, la asíncrona ofrece mayor resiliencia y desacoplamiento.

5. Resultados

Desarrollo

1. **Diseño del microservicio**

   Seleccionar una funcionalidad del proyecto que pueda desacoplarse (por ejemplo: módulo de notificaciones, gestión de equipos o usuarios).

   Definir su API interna y endpoints principales.

2. **Configuración del entorno**

   Crear una nueva carpeta o repositorio microservice_<buses> y configurarlo como módulo independiente.

   Implementar la estructura base con controladores, servicios y rutas.

   Configurar variables de entorno, dependencias y documentación básica (README).

   Configuración para el uso del docker para el levantamiento de la aplicación además de la base de datos y la autenticación de google

```yaml
services:
  backend:
    build: ./backend
    image: buses-backend-local:latest
    ports:
      - "8099:8099"
    volumes:
      - ./backend/data:/app/data
    environment:
      - JAVA_OPTS=-Xmx512m
      # En Windows, usar host.docker.internal para conectar a MongoDB local desde el contenedor
      - USE_MONGO=true
      - MONGO_URI=mongodb://host.docker.internal:27017
      - MONGO_DB=BusesApp
    # load sensitive env vars from file (backend/.env)
    env_file:
      - ./backend/.env
  frontend:
    build: ./frontend
    image: buses-frontend-local:latest
    ports:
      - "5000:5000"
    environment:
      - FLASK_ENV=production
      - API_URL=http://backend:8099
      - FLASK_SECRET_KEY=QjRReXlVb1ZYWG1RNkF4bG1xS0xVSWJfV1R3cV9rM2pZcXJqWnNlVQ==
    # Montamos el código local en el contenedor para evitar reconstrucciones durante desarrollo
    volumes:
      - ./frontend:/app
    env_file:
      - ./backend/.env
    depends_on:
      - backend
```

3. Comunicación entre servicios

Configurar API Gateway o endpoint del backend principal para enlazar el nuevo servicio.

Probar el intercambio de datos mediante peticiones HTTP.

Configuración del Api de login con java

```java
public class GoogleOAuth_api {

    @GET
    @Path("/start")
    @Produces(MediaType.APPLICATION_JSON)
    public Response start(@QueryParam("redirectTo") String redirectTo) throws Exception {
        String clientId = System.getenv(GOOGLE_CLIENT_ID);
        if (clientId == null) clientId = System.getProperty(GOOGLE_CLIENT_ID);
        if (clientId == null) clientId = System.getProperty(GOOGLE_CLIENT_ID);
        String redirectUri = System.getenv("GOOGLE_REDIRECT_URI");
        if (redirectUri == null) redirectUri = System.getProperty("GOOGLE_REDIRECT_URI");
        if (clientId == null || redirectUri == null) {
            return Response.status(Response.Status.INTERNAL_SERVER_ERROR).entity("Google OAuth not configured").buil
        }
        String state = UUID.randomUUID().toString();
        // For simplicity store state in a temporary cookie-less way: return as param (frontend should store it clie
        String scope = URLEncoder.encode("openid email profile", "UTF-8");
        String url = "https://accounts.google.com/o/oauth2/v2/auth?response_type=code&client_id=" + clientId
                + "&redirect_uri=" + URLEncoder.encode(redirectUri, StandardCharsets.UTF_8.name())
                + "&scope=" + scope + "&state=" + state + "&access_type=online&prompt=select_account";
        // return URL to frontend so it can redirect (avoids server-side sessions in this simple implementation)
        Map<String, String> resp = new HashMap<>();
        resp.put("url", url);
        resp.put("state", state);
        if (redirectTo != null) resp.put("redirect_to", redirectTo);
        return Response.ok(resp).build();
    }

    @GET
    @Path("/callback")
    @Produces(MediaType.APPLICATION_JSON)
    public Response callback(@QueryParam("code") String code, @QueryParam("state") String state, @QueryParam("error'
        // Debug incoming query params
        System.out.println("[GoogleOAuth_api] Callback received: code=" + (code == null ? "<null>" : code) + ", stat
        if (error != null) {
            System.out.println("[GoogleOAuth_api] OAuth error parameter present: " + error);
            return Response.status(Response.Status.BAD_REQUEST).entity(Collections.singletonMap("error", error)).bui
        }
```

Enlace de la api de autenticación de google con la aplicación

```python
@router.route("/auth/google/callback", methods=["GET", "POST"])
def google_callback():
    try:
        print('[google_callback] request.url =', request.url)
        print('[google_callback] request.args =', dict(request.args))
        print('[google_callback] request.method =', request.method)
    except Exception:
        pass

    error = request.args.get("error")
    if error:
        flash(f"Error en autenticación con Google: {error}", "danger")
        return redirect(url_for("router.iniciar_sesion"))
    code = request.args.get("code")
    if not code:
        try:
            # mostrar información adicional en logs para depuración
            print('[google_callback] No code received. request.args:', dict(request.args))
            print('[google_callback] Full request headers:')
            for k, v in request.headers.items():
                print(f'  {k}: {v}')
        except Exception:
            pass
        flash("No se recibió código de Google. Revisa que la redirect URI registrada en Google Cloud Console sea e›
        return redirect(url_for("router.iniciar_sesion"))

    cfg = _load_backend_google_config()
    client_id = cfg.get("client_id")
    client_secret = cfg.get("client_secret")
    # Usar la redirect_uri configurada en backend/.env si está disponible
    redirect_uri = cfg.get("redirect_uri") or url_for("router.google_callback", _external=True)

    if not client_id or not client_secret:
        flash("Faltan credenciales de Google en el backend (.env)", "danger")
        return redirect(url_for("router.iniciar_sesion"))
```

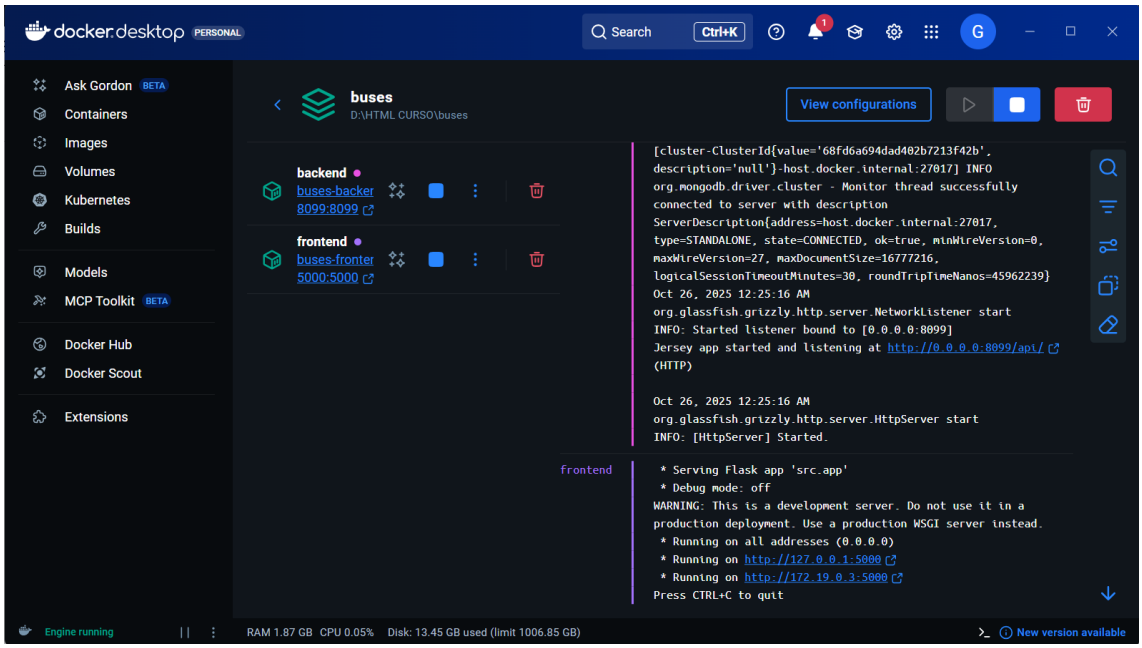
4. Despliegue y validación

Ejecutar el microservicio en contenedor Docker o entorno local.

Validar la conexión y registrar logs de comunicación.

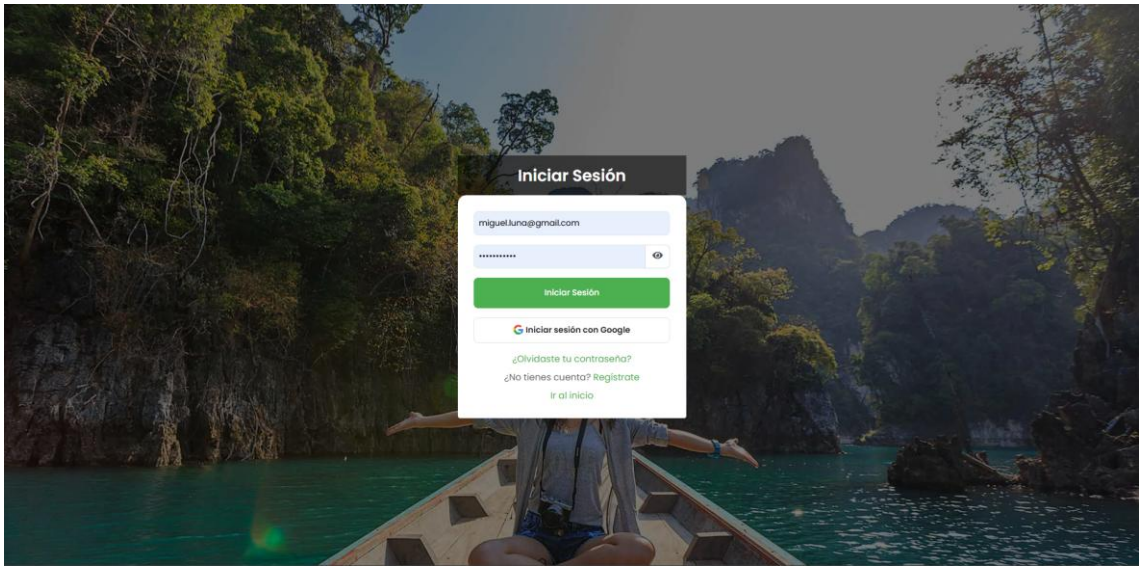
Documentar resultados en /docs/architecture/container-diagram-v2.png.

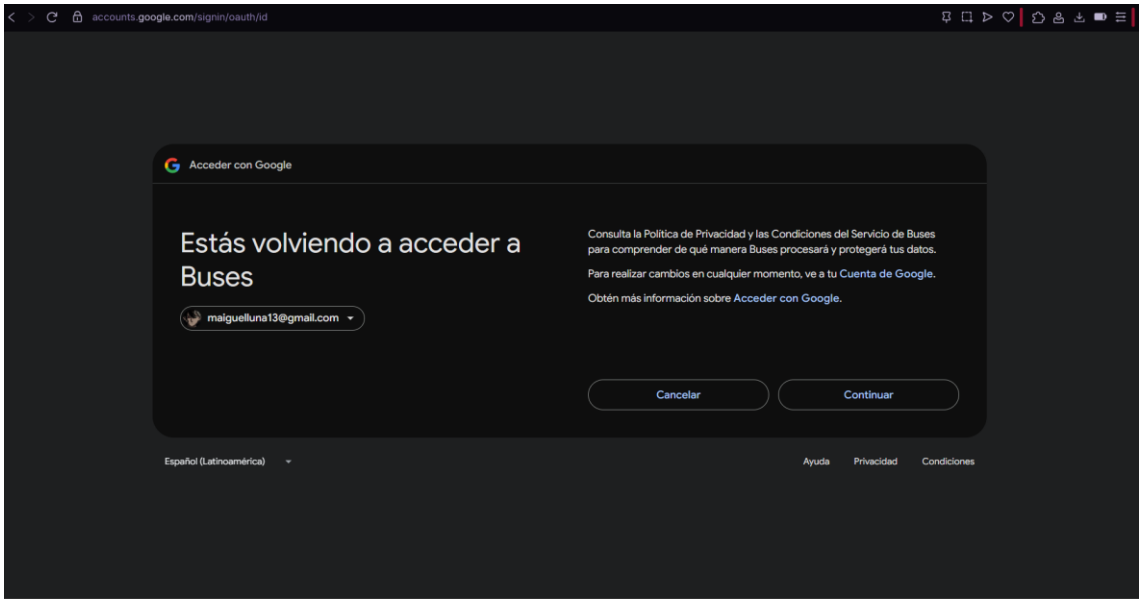
Ejecución del microservicio en Docker

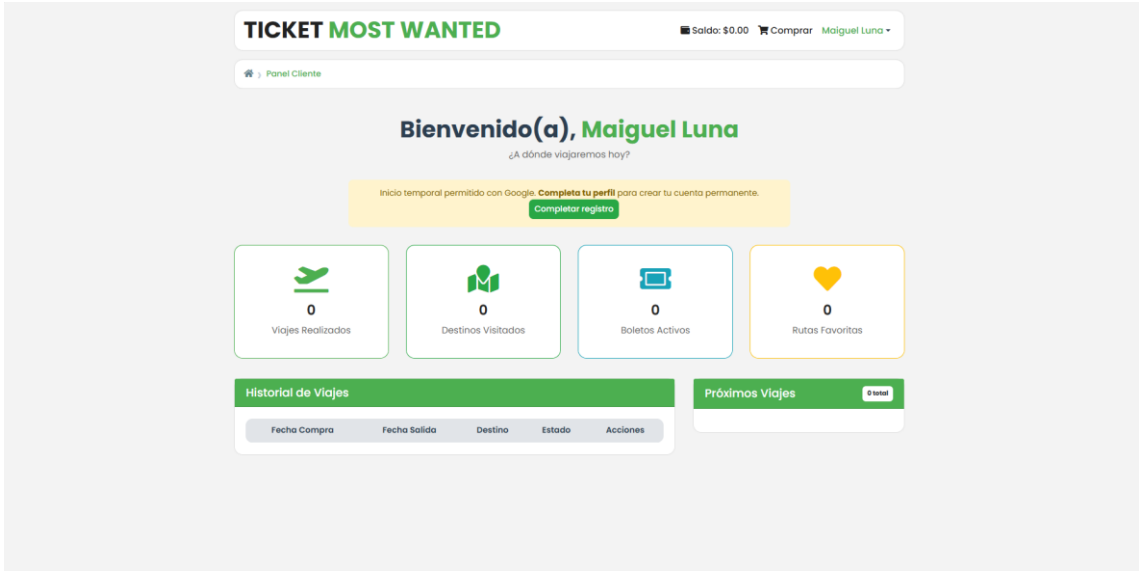


Logs de comunicación de la parte del backend

```
PS D:\HTML CURSO\buses> docker compose -f docker-compose.prod.yml logs --tail 1000 backend
ttings{maxDocumentLength=1000}, clusterSettings={hosts=[host.docker.internal:27017], srvServiceName=mongodb, mode=SINGLE, requiredClusterType=U
NKNOWN, requiredReplicaSetName='null', serverSelector='null', clusterListeners='[]', serverSelectionTimeout='30000 ms', localThreshold='30000 m
s'}, socketSettings=SocketSettings{connectTimeoutMS=10000, readTimeoutMS=0, receiveBufferSize=0, sendBufferSize=0}, heartbeatSocketSettings=Soc
ketSettings{connectTimeoutMS=10000, readTimeoutMS=10000, receiveBufferSize=0, sendBufferSize=0}, connectionPoolSettings=ConnectionPoolSettings{
maxSize=100, minSize=0, maxWaitTimeMS=120000, maxConnectionLifeTimeMS=0, maxConnectionIdleTimeMS=0, maintenanceInitialDelayMS=0, maintenanceFre
quencyMS=60000, connectionPoolListeners=[], maxConnecting=2}, serverSettings=ServerSettings{heartbeatFrequencyMS=10000, minHeartbeatFrequencyMS
=500, serverListeners='[]', serverMonitorListeners='[]'}, sslSettings=SslSettings{enabled=false, invalidHostNameAllowed=false, context=null}, a
pplicationName='null', compressorList=[], uuidRepresentation=UNSPECIFIED, serverApi=null, autoEncryptionSettings=null, dnsClient=null, inetAddr
essResolver=null, contextProvider=null}
backend-1  | [main] INFO org.mongodb.driver.cluster - Cluster description not yet available. Waiting for 30000 ms before timing out
backend-1  | [cluster-ClusterId{value='68fc3f0fcdcee8411cc9181b', description='null'}-host.docker.internal:27017] INFO org.mongodb.driver.clust
er - Monitor thread successfully connected to server with description ServerDescription{address=host.docker.internal:27017, type=STANDALONE, st
ate=CONNECTED, ok=true, minWireVersion=0, maxWireVersion=27, maxDocumentSize=16777216, logicalSessionTimeoutMinutes=30, roundTripTimeNanos=2091
6670}
backend-1  | Oct 25, 2025 3:08:01 AM org.glassfish.grizzly.http.server.NetworkListener start
backend-1  | INFO: Started listener bound to [0.0.0.0:8099]
backend-1  | Oct 25, 2025 3:08:01 AM org.glassfish.grizzly.http.server.HttpServer start
backend-1  | INFO: [HttpServer] Started.
backend-1  | Jersey app started and listening at http://0.0.0.0:8099/api/ (HTTP)
backend-1  |
backend-1  | [Auth] Login attempt for correo='stefanialuna2000@gmail.com' passwordLength=16
backend-1  | [Auth] Login attempt for correo='maiguellun12@gmail.com' passwordLength=16
backend-1  | [Auth] Login attempt for correo='maiguellun12@gmail.com' passwordLength=16
backend-1  | [main] INFO org.mongodb.driver.client - MongoClient with metadata {"driver": {"name": "mongo-java-driver|sync", "version": "4.10.2
"}, "os": {"type": "Linux", "name": "Linux", "architecture": "amd64", "version": "6.6.87.2-microsoft-standard-WSL2"}, "platform": "Java/Eclipse
 Adoptium/17.0.16+8"} created with settings MongoClientSettings{readPreference=primary, writeConcern=WriteConcern{w=null, wTimeout=null ms, jou
rnal=null}, retryWrites=true, retryReads=true, readConcern=ReadConcern{level=null}, credential=null, streamFactoryFactory=null, commandListener
s=[], codecRegistry=ProvidersCodecRegistry{codecProviders=[ValueCodecProvider{}, BsonValueCodecProvider{}, DBRefCodecProvider{}, DBObjectCodecP
rovider{}, DocumentCodecProvider{}, CollectionCodecProvider{}, IterableCodecProvider{}, MapCodecProvider{}, GeoJsonCodecProvider{}, GridFSFileC
odecProvider{}, Jsr310CodecProvider{}, JsonObjectCodecProvider{}, BsonCodecProvider{}, EnumCodecProvider{}, com.mongodb.client.model.mql.Expres
sionCodecProvider@3b2c72c2, com.mongodb.Jep395RecordCodecProvider@491666ad, com.mongodb.KotlinCodecProvider@176d53b2]}, loggerSettings=LoggerSe
ttings{maxDocumentLength=1000}, clusterSettings={hosts=[host.docker.internal:27017], srvServiceName=mongodb, mode=SINGLE, requiredClusterType=U
NKNOWN, requiredReplicaSetName='null', serverSelector='null', clusterListeners='[]', serverSelectionTimeout='30000 ms', localThreshold='30000 m
s'}, socketSettings=SocketSettings{connectTimeoutMS=10000, readTimeoutMS=0, receiveBufferSize=0, sendBufferSize=0}, heartbeatSocketSettings=Soc
ketSettings{connectTimeoutMS=10000, readTimeoutMS=10000, receiveBufferSize=0, sendBufferSize=0}, connectionPoolSettings=ConnectionPoolSettings{
maxSize=100, minSize=0, maxWaitTimeMS=120000, maxConnectionLifeTimeMS=0, maxConnectionIdleTimeMS=0, maintenanceInitialDelayMS=0, maintenanceFre
```


Iniciar Sesión
miguel.luna@gmail.com
Iniciar Sesión
Iniciar sesión con Google
¿Olvidaste tu contraseña?
¿No tienes cuenta? Regístrate
Ir al inicio

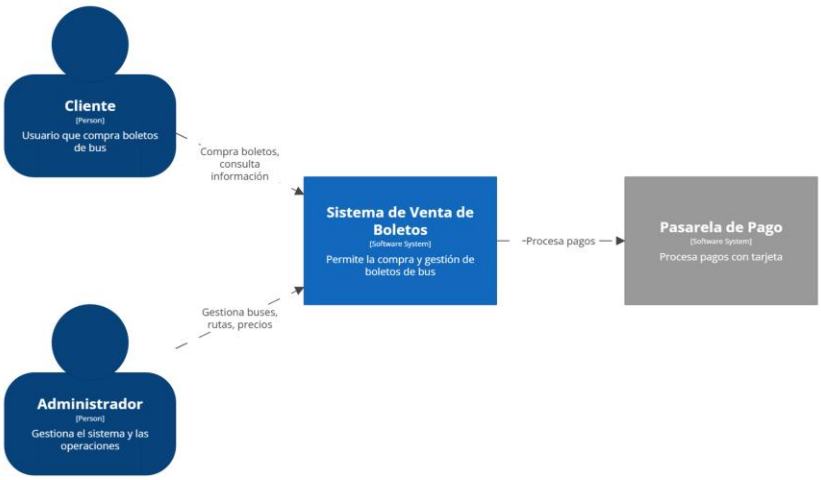

accounts.google.com/signin/oauth/id
Acceder con Google
Estás volviendo a acceder a Buses
maiguelluna13@gmail.com
Consulta la Política de Privacidad y las Condiciones del Servicio de Buses para comprender de qué manera Buses procesará y protegerá tus datos.
Para realizar cambios en cualquier momento, ve a tu Cuenta de Google.
Obtén más información sobre Acceder con Google.
Cancelar
Continuar
Español (Latinoamérica)
Ayuda
Privacidad
Condiciones


TICKET MOST WANTED
Saldo: $0.00
Comprar
Maiguel Luna
Panel Cliente
Bienvenido(a), Maiguel Luna
¿A dónde viajaremos hoy?
Inicio temporal permitido con Google. Completa tu perfil para crear tu cuenta permanente.
Completar registro
0
Viajes Realizados
0
Destinos Visitados
0
Boletos Activos
0
Rutas Favoritas
Historial de Viajes
Fecha Compra
Fecha Salida
Destino
Estado
Acciones
Próximos Viajes
0 total

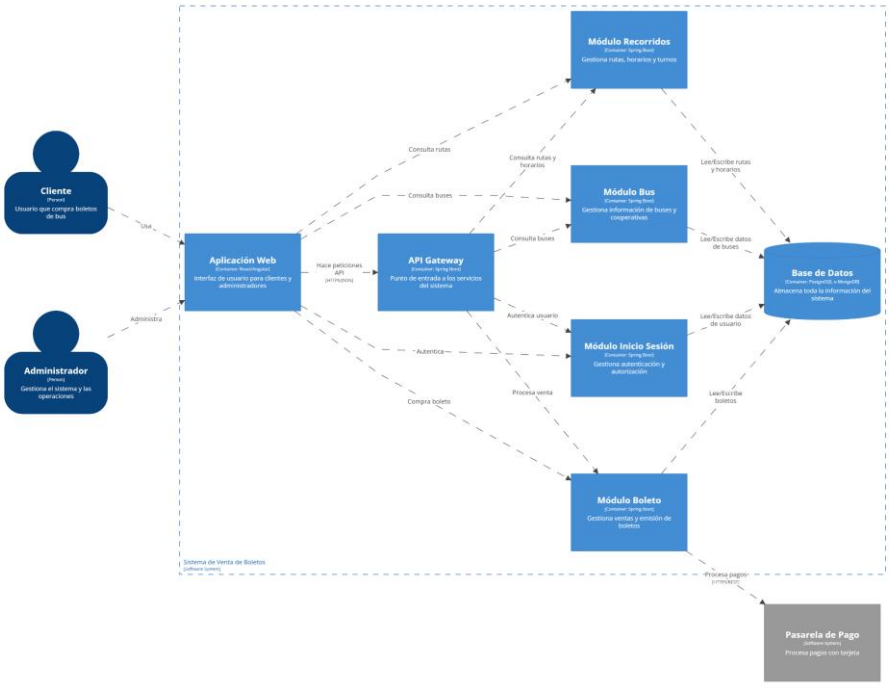
5. Registro en modelo C4

Actualizar los diagramas Container y Component incorporando el microservicio y el gateway.

Registrar capturas o exportar los diagramas actualizados.



[System Context] Sistema de Venta de Boletos
viernes, 17 de octubre de 2025, 7:49 a. m. hora de Ecuador



[Container] Sistema de Venta de Boletos
viernes, 17 de octubre de 2025, 7:49 a. m. hora de Ecuador

6. Preguntas de Control

- **¿Qué diferencia principal existe entre una arquitectura monolítica y una basada en microservicios?**

Las aplicaciones monolíticas suelen constar de una interfaz de usuario del cliente, una base de datos y una aplicación del servidor. Los desarrolladores crean todos estos módulos en un único código base. Por otro lado, en una arquitectura distribuida, cada microservicio funciona para lograr una única característica o lógica empresarial. En lugar de intercambiar datos dentro del mismo código base, los microservicios se comunican con una API.

- **¿Qué beneficios aporta el uso de un API Gateway en la integración de servicios?**

  El uso de un API Gateway en la integración de servicios, especialmente en arquitecturas basadas en microservicios o sistemas distribuidos, aporta múltiples beneficios en términos de seguridad, eficiencia, escalabilidad y mantenibilidad.

  El API Gateway actúa como una única puerta de acceso para todos los clientes; web, móvil, IoT, etc.

  Permite implementar políticas de autenticación, autorización, validación de tokens; JWT, OAuth2, y filtrado de IPs en un solo punto.

  Distribuye las solicitudes entre diferentes instancias de los servicios.

- **¿Qué riesgos se presentan al no manejar correctamente la comunicación entre microservicios?**

  No manejar correctamente la comunicación entre microservicios puede llegar a producir complejidad en la gestión y el mantenimiento además podríamos tener dificultades en las pruebas y la depuración, riesgos de seguridad y de inconsistencia de datos.

  Ejemplo:

  Supongamos que contamos con un sistema en varios microservicios.

  Servicio A: maneja autenticación de usuarios.
  Servicio B: gestiona los pagos.
  Servicio C: envía correos de confirmación.

  Si la comunicación entre estos servicios no está bien manejada ni protegida, pueden surgir varios riesgos como la fuga de información sensible, suplantación entre servicio entre otros.

- **¿Cómo se refleja la modularidad en los niveles Container y Component del modelo C4?**

  La modularidad se refleja en el nivel de Contenedor mostrando la división del sistema en unidades de ejecución como aplicaciones web o bases de datos, y en el nivel de Componente desglosando cada contenedor en módulos internos más pequeños como controladores o servicios que realizan tareas específicas. El nivel de Contenedor organiza la arquitectura en bloques ejecutables independientes, mientras que el nivel de Componente detalla la estructura interna de estos bloques y sus relaciones lógicas.

- **¿Qué consideraciones de seguridad se deben mantener al exponer endpoints entre servicios?**

  Al exponer endpoints, se deben implementar medidas de seguridad que incluyan autenticación y autorización sólidas, cifrado de datos en tránsito y en reposo, y una defensa en profundidad con firewalls y puertas de enlace API. Consideradas a tomar en cuenta:

  - Autenticación fuerte: Utilizar protocolos modernos como OAuth o JWT para verificar las identidades de los servicios que se conectan.
  - Autorización: implementar permisos detallados para asegurar que cada servicio solo tenga acceso a los datos y funciones que necesita.
  - Cifrado en tránsito: Usar protocolos como TLS para cifrar la comunicación entre los servicios y proteger los datos mientras viajan por la red.
  - Validación de datos: Implementar la validación y desinfección de todos los datos entrantes para prevenir ataques de inyección.
  - Monitorización continua: Supervisar activamente el tráfico y la actividad de los endpoints en busca de comportamientos sospechosos.

7. Conclusiones

- La arquitectura de microservicios representa un cambio fundamental en el desarrollo de software, al dividir los sistemas en componentes pequeños, independientes y fácilmente escalables, lo que mejora la mantenibilidad, la flexibilidad y la capacidad de evolución continua frente a las arquitecturas monolíticas tradicionales.

- El enfoque serverless complementa la filosofía de los microservicios al eliminar la necesidad de gestionar servidores e infraestructura, permitiendo a los desarrolladores centrarse exclusivamente en la lógica de negocio, optimizando los recursos y reduciendo los costos operativos mediante una ejecución bajo demanda.
- Los patrones de comunicación son esenciales para garantizar la eficiencia y confiabilidad en los sistemas distribuidos, ya que determinan cómo los microservicios o funciones serverless interactúan entre sí. La elección adecuada entre comunicación síncrona o asíncrona influye directamente en la escalabilidad, el rendimiento y la resiliencia global de la arquitectura.

8. Recomendaciones

- Fomentar el diseño modular y la documentación clara de los microservicios, asegurando que cada componente tenga responsabilidades bien definidas y que las interfaces de comunicación estén correctamente descritas. Esto facilita el mantenimiento, la colaboración entre equipos y la integración de nuevos servicios en el futuro.
- Implementar entornos de prueba y monitoreo continuo en soluciones serverless, con el fin de identificar cuellos de botella, optimizar el tiempo de ejecución y controlar el consumo de recursos. El uso de herramientas de observabilidad, como logs centralizados y métricas de rendimiento, es clave para garantizar la estabilidad del sistema en producción.
- Seleccionar cuidadosamente los patrones de comunicación según las necesidades del sistema, combinando estrategias síncronas y asíncronas cuando sea necesario. En escenarios reales, un diseño híbrido puede ofrecer un equilibrio entre rapidez de respuesta y tolerancia a fallos, mejorando la eficiencia general del sistema distribuido.

9. Bibliografía / Referencias

[1] Newman, S. (2021). Building Microservices. O'Reilly Media.

[2] OWASP Foundation (2023). OWASP Secure Microservices Design.

[3] Docker Inc. Docker Documentation.

[4] Richardson, C. (2022). Microservices Patterns: With examples in Java. Manning Publications.

[5] Red Canary,(2024). "What is endpoint security? Endpoint security guide," RedCanary.com.

10. Anexos