Proyecto1 - Bases de Datos II

Portada

Proyecto 1: API de restaurantes
Curso: Bases de Datos II
Integrantes:

- Daniel Alemán
- Luis Meza

Índice

Enlace de GitHub

Repositorio del Proyecto

Enlace sobre la Arquitectura del Proyecto

Arquitectura del Proyecto

Descripción del Proyecto

Este proyecto implementa un sistema completo para gestión de restaurantes con una arquitectura de microservicios. Permite administrar restaurantes, menús, productos, reservaciones y pedidos a través de un conjunto de APIs RESTful. El sistema está diseñado con alta disponibilidad, escalabilidad y rendimiento como prioridades, empleando tecnologías modernas como balanceo de carga, sharding de bases de datos, caché distribuida y búsquedas optimizadas.

Arquitectura

Arquitectura Lógica

El proyecto sigue una arquitectura de microservicios con los siguientes componentes clave:

1. **Microservicios API**:
   - API principal para operaciones CRUD de restaurantes, menús, productos, reservas y pedidos
   - Servicio de autenticación para gestión de usuarios y tokens JWT
   - Servicio de búsqueda optimizado con Elasticsearch
2. **Balanceo de Carga**:
   - Nginx como proxy inverso y balanceador para distribuir las peticiones entre instancias
3. **Persistencia**:
   - MongoDB: Almacenamiento principal con sharding y replicación para alta disponibilidad
   - PostgreSQL: Almacenamiento alternativo configurable
   - Elasticsearch: Índice de búsqueda para consultas optimizadas
4. **Caché**:
   - Redis como almacén de caché distribuida para mejorar el rendimiento
5. **CI/CD**:
   - Pipeline automatizado para pruebas, construcción y despliegue

Estructura del Proyecto

La estructura de directorios del proyecto está organizada por funcionalidad:

```
proyecto1-bases2/
├── api/                    # API principal
├── auth_service/           # Servicio de autenticación
├── search_service/         # Servicio de búsqueda
├── pruebas/                # Scripts de prueba y generación de datos
├── docker-compose.yml      # Configuración de contenedores
```

```
├── init_cluster.sh         # Script para inicializar el cluster MongoDB
├── set_config.sh           # Script para configurar el entorno
├── nginx.conf              # Configuración del balanceador de carga
└── README.md               # Documentación
```

Cada servicio sigue una estructura MVC (Modelo-Vista-Controlador) con separación clara de responsabilidades:

```
servicio/
├── src/
│   ├── config/             # Configuración
│   ├── controllers/        # Controladores
│   ├── dao/                # Objetos de acceso a datos
│   ├── db/                 # Conexiones a bases de datos
│   ├── middlewares/        # Middlewares
│   ├── models/             # Modelos de datos
│   ├── routes/             # Definición de rutas
│   └── app.js              # Aplicación principal
├── swagger/                # Documentación de API
├── tests/                  # Pruebas
│   ├── integration/        # Pruebas de integración
│   ├── unit/               # Pruebas unitarias
│   └── utils/              # Utilidades para pruebas
└── server.js               # Punto de entrada
```

Componentes Principales

API Principal

La API principal maneja todas las operaciones CRUD relacionadas con:

- Restaurantes
- Menús
- Productos
- Reservaciones
- Pedidos

Características clave:

- Implementación RESTful con Express.js
- Documentación completa con Swagger
- Escalabilidad horizontal con múltiples instancias (api1, api2)
- Interconexión con servicios de autenticación y búsqueda

Servicio de Autenticación

Este microservicio gestiona todo lo relacionado con usuarios y seguridad:

- Registro de usuarios

- Inicio de sesión
- Verificación de tokens JWT
- Gestión de roles (administrador/cliente)

Características:

- Autenticación basada en JWT para seguridad
- Almacenamiento seguro de contraseñas con hash
- Alta disponibilidad con múltiples instancias (auth_service1, auth_service2)

Servicio de Búsqueda

Un microservicio independiente dedicado a proporcionar funcionalidad de búsqueda optimizada para el sistema:

- **Funcionalidades principales**:
    - Búsqueda de productos por texto libre
    - Búsqueda de productos por categoría
    - Indexación automática de productos nuevos
    - Reindexación manual de todo el catálogo
    - Sincronización con la base de datos principal
- **Características técnicas**:
    - Integración con Elasticsearch como motor de búsqueda de alto rendimiento
    - Índices optimizados para consultas rápidas y flexibles
    - Búsquedas tolerantes a errores tipográficos
    - Resultados relevantes con ponderación inteligente
    - Actualización en tiempo real del índice cuando cambian los productos
    - Alta disponibilidad mediante múltiples instancias (search_service1, search_service2)
    - Resiliencia ante fallos mediante manejo de errores robusto

El servicio está diseñado para funcionar de manera independiente, lo que permite que el sistema principal siga operando incluso si el servicio de búsqueda experimenta problemas temporales.

Balanceador de Carga (Nginx)

El sistema utiliza Nginx como balanceador de carga y proxy inverso para distribuir el tráfico entre múltiples instancias de cada microservicio:

- **Configuración implementada**:
    - Balanceo de carga para API principal entre instancias api1 y api2
    - Balanceo de carga para servicio de autenticación entre auth_service1 y auth_service2
    - Balanceo de carga para servicio de búsqueda entre search_service1 y search_service2
    - Enrutamiento basado en prefijos de URL (/api/, /auth/, /search/)
    - Terminación SSL centralizada
- **Características técnicas**:

    - Algoritmo de balanceo round-robin para distribución uniforme de carga
    - Compresión de respuestas para optimizar el ancho de banda
    - Buffer y timeouts configurados para operaciones de larga duración
    - Redirección inteligente basada en path de URL
    - Health checks periódicos para detectar instancias no disponibles

- **Beneficios para el sistema**:

    - Alta disponibilidad mediante múltiples instancias de cada servicio
    - Escalabilidad horizontal sencilla (añadir más instancias sin cambios en la aplicación)
    - Resistencia ante fallos de servicios individuales
    - Punto único de entrada para los clientes con enrutamiento transparente
    - Capacidad de actualizar servicios individuales sin interrumpir el sistema completo

Nginx corre en su propio contenedor Docker, y su configuración se monta desde el archivo nginx.conf en el sistema host.

Sistema de Caché (Redis)

El proyecto implementa Redis como un sistema de caché distribuida para optimizar el rendimiento y reducir la carga en las bases de datos:

- **Casos de uso implementados**:

    - Caché de productos individuales y listados completos
    - Almacenamiento temporal de resultados de consultas frecuentes
    - Caché de datos de autenticación y sesiones
    - Invalidación automática de caché al modificar recursos

- **Estrategia de caché**:

    - Implementación de patrón Cache-Aside para recursos frecuentemente accedidos
    - TTL (Time-To-Live) configurado según el tipo de datos
    - Invalidación selectiva al modificar recursos relacionados
    - Manejo de versiones de datos en caché

- **Beneficios medibles**:

    - Reducción de tiempos de respuesta de API en hasta un 80% para recursos en caché
    - Alivio significativo de carga en MongoDB y PostgreSQL
    - Mayor resistencia del sistema ante picos de tráfico
    - Acceso ultrarrápido en memoria para datos de alta demanda

Las pruebas de integración incluyen verificaciones específicas del comportamiento del caché, asegurando que el sistema maneja correctamente la obtención, almacenamiento e invalidación de datos en caché.

Bases de Datos

MongoDB (Principal)

- Configurado con sharding y replicación para alta disponibilidad y escalabilidad

- Estructura:
  - 2 shards con 3 réplicas cada uno
  - 3 servidores de configuración
  - 3 routers (mongos)
- Colecciones fragmentadas mediante hash de identificadores para distribución uniforme

PostgreSQL

- Base de datos relacional como alternativa configurable
- Ideal para consultas complejas y relaciones estructuradas

Elasticsearch

- Motor especializado para búsquedas de texto completo
- Indexación optimizada de productos para consultas rápidas

Instalación

Requisitos Previos

Para ejecutar este proyecto necesita:

- Docker y Docker Compose
- Git
- Node.js y npm (para desarrollo local)

Instalación del Proyecto

Si desea clonar el repositorio y hacer uso de él, basta con utilizar la siguiente serie de comandos:

```
git clone https://github.com/DanielAR27/proyecto1-bases2.git
cd proyecto1-bases2
```

Instalación de Módulos

Para cada servicio que requiere instalación de módulos:

- ./api
- ./auth_service
- ./search_service

Para ubicarse dentro de ellos, debe estar en la raíz del proyecto y utilizar el siguiente comando:

```
cd <nombre_del_servicio>
```

Una vez dentro, puede instalar o actualizar los módulos necesarios:

```
npm install
```

Construcción y Levantamiento de los Servicios

Para construir los contenedores e iniciar toda la aplicación, primero debe dar permisos de ejecución a los scripts:

```
chmod +x set_config.sh
chmod +x init_cluster.sh
```

Luego ejecute el script principal:

```
./set_config.sh
```

Este script realiza las siguientes acciones:

1. Levanta todos los servicios base con `docker-compose up -d`
2. Inicializa el cluster MongoDB con sharding y replicación mediante `./init_cluster.sh`
3. Levanta los servicios backend (API, Auth, Search) con `docker-compose --profile backend up --build -d`

El script `init_cluster.sh` configura MongoDB con:

- Replica Set de configuración (3 nodos)
- Dos Shards Replica Set (3 nodos cada uno)
- Habilitación de sharding en la base de datos `apidb`
- Configuración de colecciones particionadas
- Preparación de metadatos en los routers

Pruebas

Pruebas Unitarias y de Integración

El proyecto cuenta con pruebas automatizadas para cada servicio:

Servicio de Autenticación

```
# Construir contenedor de prueba
docker-compose --profile test build auth_test

# Ejecutar pruebas con cobertura
docker-compose --profile test run --rm auth_test
```

Servicio de Búsqueda

```
# Construir contenedor de prueba
docker-compose --profile test build search_test

# Ejecutar pruebas con cobertura
docker-compose --profile test run --rm search_test
```

API Principal

```
# Construir contenedor de prueba
docker-compose --profile test build api_test

# Ejecutar pruebas con cobertura
docker-compose --profile test run --rm api_test
```

Cobertura de Pruebas

El sistema utiliza Jest como framework de pruebas y se centra en dos tipos principales de pruebas:

- **Pruebas unitarias**: Verifican el comportamiento correcto de componentes individuales como DAOs, controladores y modelos.
- **Pruebas de integración**: Evalúan la interacción entre diferentes partes del sistema, incluyendo:
  - Flujos CRUD completos
  - Validación de datos de entrada
  - Manejo de permisos y autenticación
  - Integración entre servicios
  - Funcionamiento del sistema de caché

Cada prueba genera informes detallados de cobertura que muestran qué porcentaje del código está siendo evaluado.

Un ejemplo de los aspectos verificados en las pruebas de integración:

- Creación, lectura, actualización y eliminación de recursos
- Comportamiento correcto frente a entradas inválidas
- Verificación de permisos según roles de usuario
- Comportamiento del caché Redis
- Resiliencia ante fallas en servicios externos

CI/CD Pipeline

El proyecto implementa un pipeline completo de Integración Continua y Despliegue Continuo utilizando GitHub Actions, lo que garantiza la calidad del código y facilita el proceso de entrega:

Workflow de CI/CD

El pipeline se activa automáticamente en los siguientes casos:

- Con cada push a las ramas main, master o develop
- Al crear Pull Requests hacia main o master

Etapas del Pipeline

1. **Etapa de Pruebas (Test)**:
    - Clona el repositorio y configura el entorno usando template.env
    - Levanta la infraestructura completa con Docker Compose
    - Inicializa el cluster MongoDB con la configuración de sharding
    - Construye y arranca los servicios backend
    - Ejecuta las pruebas unitarias y de integración para cada servicio:
        - Pruebas del servicio de autenticación
        - Pruebas de la API principal
        - Pruebas del servicio de búsqueda
    - Genera informes de cobertura de código
2. **Etapa de Construcción y Publicación (Build-and-Push)**:
    - Se ejecuta solo después de que las pruebas sean exitosas en ramas main o master
    - Configura Docker Buildx para construcción multiplataforma
    - Autentica con GitHub Container Registry
    - Construye imágenes Docker optimizadas para cada servicio:
        - Servicio de autenticación
        - API principal
        - Servicio de búsqueda
    - Publica las imágenes en GitHub Container Registry con el tag "latest"
3. **Etapa de Despliegue (Deploy)**:
    - Crea un artefacto de despliegue que incluye:
        - Docker Compose configurado
        - Variables de entorno (.env)
        - Configuración de Nginx
        - Scripts de inicialización
    - Genera documentación detallada con instrucciones paso a paso para el despliegue
    - Sube los artefactos al sistema de almacenamiento de GitHub Actions
    - Prepara instrucciones para actualizar un despliegue existente

Este pipeline garantiza que:

- Todo el código pase las pruebas automatizadas
- Solo el código verificado se construya y publique
- Se generen artefactos consistentes para cada versión
- El proceso de despliegue sea reproducible y documentado

El workflow completo está definido en el archivo ci-cd.yml en la raíz del repositorio.

Acceso a Interfaces

Documentación de la API Rest

Para visualizar la documentación interactiva generada con Swagger:

- API Principal: `http://localhost/api/api-docs/`
- Servicio de Autenticación: `http://localhost/auth/api-docs/`
- Servicio de Búsqueda: `http://localhost/search/api-docs/`

Visualización en Tiempo Real de Postgres DB

Use PgAdmin para gestionar la base de datos PostgreSQL:

```
http://localhost:5050
```

Pasos para configurar PgAdmin:

1. Click derecho en "Servers" → Register → Server
2. En General: Nombre = "PG Docker"
3. En Connection:
    - Host: `postgres_container`
    - Port: `5432`
    - Database: `apidb`
    - Username: `postgres`
    - Password: `postgres`

Visualización en Tiempo Real de Mongo DB

Use Mongo Express para gestionar MongoDB:

```
http://localhost:8081
```

Pasos para acceder:

1. Introduzca las credenciales configuradas en el .env
2. Acceda a la base de datos "apidb" para ver las colecciones
3. Explore las colecciones: counters, menus, pedidos, productos, reservas, restaurantes, usuarios

Elasticsearch y Kibana

Para gestionar y monitorear Elasticsearch:

```
http://localhost:5601
```

Kibana ofrece una interfaz intuitiva para:

- Explorar índices
- Crear y probar consultas
- Visualizar datos
- Monitorear rendimiento

Reinicio Completo del Entorno

Si desea eliminar todos los contenedores, redes y volúmenes:

```
docker-compose down -v
```

Esto restablecerá completamente el entorno y eliminará todos los datos almacenados.

Autores: Daniel Alemán, Luis Meza
Última actualización: *11/5/2025*