AAA Planificación ☝️

Requisitos

Funcionales

- Los usuarios pueden publicar mensajes cortos (tweets) con un límite de 280 caracteres.
- Los usuarios pueden seguir a otros usuarios.
- Los usuarios pueden ver una línea de tiempo con los tweets de las personas a las que siguen.

No funcionales

- La aplicación debe ser altamente escalable y optimizada para lecturas.
- Baja latencia en la visualización del timeline.
- Uso de buenas prácticas en arquitectura de software.
- Posibilidad de escalar a millones de usuarios.

Problemas a Resolver

1. Gestión eficiente de la publicación de tweets.
2. Implementación del seguimiento entre usuarios.
3. Optimización de la recuperación del timeline.
4. Escalabilidad y rendimiento del sistema.
5. Estrategia de cacheo para optimizar la lectura de tweets.

Identificar Entidades Claves

1. **Usuario**:
   - Atributos:

     id,

     nombre,

     apellido,

email,

fecha_nacimiento,

password,

avatar,

banner,

biografia,

web_site,

ubicación,

created_at

2. **Tweet**:
    - Atributos:

      id,
      usuario_id,
      contenido,
      timestamp

Relación entre Entidades

- Un **Usuario** puede tener múltiples **Tweets**.
- Un **Usuario** puede seguir a otros **Usuarios**.
- La línea de tiempo de un **Usuario** está compuesta por los tweets de los usuarios a los que sigue.

Elección de Tecnologías

- **Backend**: Golang.
- **Base de Datos**: MongoDB para almacenamiento principal y Redis para cacheo del timeline.
- **Contenedores**: Docker.

- **AWS**:
  - **Lambda**: Desplegar funciones serverless que contendrán los microservicios buildeados en binarios de Go en archivos .zip
  - **Api Gateway**: Api para conectar y manejar todas las peticiones y respuestas Rest con los microservicios en Lambda
  - **Secret Manager**: Administrar credenciales de la base de datos
  - **S3**: Buckets que permiten contener y adminstrar archivos pesados como imagenes que llaman los microservicios en lambda
- **Endpoints**: Postman para documentar y automatizar
- **Admin BD**: Mongo Atlas
- **Cloud Watch**: Visualizar logs y métricas de las lambdas
- **Pruebas**: Testify para unit testing en Go.
- **Mensajería**: Kafka para procesar eventos de follow/unfollow.

Arquitectura del Sistema

- **Patrones de Diseño**:
  - Event-Driven Architecture para manejar eventos de follow/unfollow.
  - CQRS para optimizar consultas y comandos.
  - Repository Pattern para abstracción del acceso a datos.
- **Diagrama de Arquitectura**:
  - Microservicios comunicándose mediante eventos.
  - Redis como capa de cacheo.
  - MongoDB como almacenamiento persistente.
  - API Gateway manejando peticiones externas.

Estructura Backend

1. **Directorios principales**:
   - `/cmd`: Puntos de entrada de la aplicación.
   - `/internal`: Lógica de negocio y entidades (implementación DDD).
   - `/pkg`: Componentes reutilizables.
   - `/configs`: Archivos de configuración.
   - `/scripts`: Scripts para tareas como migraciones o configuración inicial.
2. **Microservicios**:
   - **User Service**: Gestión de usuarios y relaciones de seguimiento.
   - **Tweet Service**: Gestión de tweets y almacenamiento.

- **Timeline Service**: Recuperación optimizada de timelines.

Microservicios

Lista de Microservicios

1. **User Service**: Maneja el registro, autenticación y gestión de relaciones de usuarios.
2. **Tweet Service**: Se encarga de la creación, edición y eliminación de tweets.
3. **Timeline Service**: Recupera y optimiza la línea de tiempo de cada usuario.
4. **Notification Service**: Envía notificaciones a los usuarios (ej: un nuevo seguidor, un like en un tweet).

Arquitectura de Microservicios

- Cada microservicio se comunica mediante eventos a través de Kafka o RabbitMQ.
- Redis se usa para cachear timelines y reducir la carga de MongoDB.
- API Gateway gestiona todas las peticiones entrantes y distribuye a los microservicios.
- Los servicios están desplegados en contenedores con Kubernetes para escalabilidad.

Estructura de Archivos en un Microservicio

Ejemplo para `User Service`:

```
/user-service
├── cmd/                    # Puntos de entrada
│   ├── main.go             # Inicializa el servicio
├── internal/               # Lógica de negocio
│   ├── handlers/           # Controladores HTTP
│   ├── services/           # Lógica principal
│   ├── repositories/       # Acceso a la base de datos
│   ├── models/             # Definición de estructuras de datos
├── pkg/                    # Código reutilizable
├── configs/                # Configuraciones y variables de entorno
├── scripts/                # Scripts para tareas automatizadas
├── Dockerfile              # Contenedor para despliegue
├── go.mod                  # Dependencias del proyecto
└── README.md               # Documentación
```

Cada microservicio sigue una estructura similar, asegurando modularidad y mantenimiento sencillo.

```
/user-service (repo: microblogging-user)
├── /cmd
├── /internal
├── /pkg
├── /config
├── Dockerfile
├── go.mod
├── main.go

/tweet-service (repo: microblogging-tweet)
├── /cmd
├── /internal
├── /pkg
├── /config
├── Dockerfile
├── go.mod
├── main.go

/timeline-service (repo: microblogging-timeline)
├── /cmd
├── /internal
├── /pkg
├── /config
├── Dockerfile
├── go.mod
├── main.go

/notification-service (repo: microblogging-notifications)
├── /cmd
├── /internal
├── /pkg
├── /config
├── Dockerfile
├── go.mod
├── main.go

/microblogging-shared (repo de código compartido)
├── /auth
├── /config
├── /logging
├── /events
```

```
├── go.mod
├── README.md
```

Consideraciones Finales

- Se valorará la documentación clara en un README.md.
- El código debe ser dockerizable para facilitar despliegue.
- Se recomienda realizar pruebas unitarias en las funciones críticas.