

Red Social Distribuida - MongoDB

Jesus Eduardo Cornejo Clavel
ALUMNO

Eduardo Cornejo-Velazquez
PROFESOR

3 de mayo de 2025

Índice

1. Arquitectura del Sistema	3
1.1. Configuración de MongoDB	3
1.1.1. Configuración del Replica Set	3
1.1.2. Arquitectura de ReplicaSet y Priorización	5
1.1.3. Implementación de Seguridad	6
1.1.4. Proceso de Inicialización Automatizada	7
1.1.5. Connection Pooling y Resiliencia	7
1.2. Diagrama de Comunicación Rust	10
2. Casos de Uso	10
2.1. Autenticación y Gestión de Perfiles	10
2.2. Gestión de Publicaciones	11
2.3. Interacciones Sociales	12
2.4. Gestión de Feed	13
3. Stack Tecnológico	14
3.1. Componentes del Stack Tecnológico	15
3.1.1. Lenguajes y Frameworks	15
3.2. Configuración de MongoDB	16
3.2.1. Configuración del Replica Set	16
3.2.2. Connection Pooling y Resiliencia	16
3.3. Estructura de la Aplicación Rust	16
3.3.1. API Endpoints y Funcionalidades	16
3.3.2. Manejo de Errores y Responses	16
3.3.3. Modelos de Datos	16
4. Modelos de Datos	16
4.1. Diseño de Colecciones	16
4.1.1. Colección de Usuarios	16
4.1.2. Colección de Posts	16
4.1.3. Colección de Comentarios	17
4.1.4. Colección de Likes	17
4.1.5. Colección de Follows	18
4.2. Estrategias de Modelado de Datos	18
4.2.1. Denormalización Estratégica	18

4.2.2. Gestión de Relaciones	18
4.3. Distribución de Datos	18
4.4. Estrategias de Indexación	19
5. Endpoints de la API	19
5.1. Operaciones de Usuario	19
5.1.1. Crear Usuario	19
5.1.2. Perfiles de Usuario	20
5.2. Interacciones Sociales	20
5.2.1. Crear Posts	20
5.2.2. Funcionalidad de Comentarios	21
5.2.3. Relaciones de Follow	21
5.3. Operaciones del Sistema	21
5.3.1. Implementación del Health Check	22
5.3.2. Población de Base de Datos para Testing	22
5.3.3. Funcionalidad de Limpieza de Base de Datos	22
6. Implementación Técnica	23
6.1. Error Handling	23
6.2. State Management	24
6.3. Conexión a Base de Datos y Resiliencia	24
7. Manejo de Consistencia y Durabilidad	25
7.1. Configuración de Garantías de Consistencia	25
8. Configuración de Resiliencia	26
8.1. Gestión de Timeouts	26
8.2. Políticas de Reintento	26
8.3. Monitoreo y Health Checks	26
8.4. Balanceo de Carga	27
8.5. Error Handling	28
9. Conclusiones	28
9.1. Arquitectura Distribuida	28
9.2. Garantías de Datos	28
9.3. Resiliencia y Monitoreo	29
9.4. Escalabilidad	29

1. Arquitectura del Sistema

1.1. Configuración de MongoDB

1.1.1. Configuración del Replica Set

La aplicación implementa una arquitectura de alta disponibilidad basada en MongoDB con una configuración de Replica Set. Esta configuración consta de tres nodos:

- **Nodo Primario (Primary):** Gestiona todas las operaciones de escritura y coordina la sincronización con los nodos secundarios.
- **Nodos Secundarios (Secondary):** Dos nodos que mantienen copias sincronizadas de los datos y pueden asumir el rol primario en caso de fallos.

Esta arquitectura proporciona:

- **Alta disponibilidad:** Si el nodo primario falla, uno de los secundarios puede ser promovido automáticamente.
- **Redundancia de datos:** Los datos se replican en múltiples nodos, evitando pérdidas en caso de fallos.
- **Balanceo de lecturas:** Las operaciones de lectura pueden distribuirse entre los nodos secundarios.

La configuración se implementa mediante Podman Compose:

```
services:
  # Nodo MongoDB primario
  central-mongodb:
    image: mongo:latest
    container_name: central-mongodb
    command: mongod --replSet rs0 --keyFile /etc/mongo-keyfile --
    bind_ip_all --auth --port 27017
    ports:
      - "27017:27017"
    volumes:
      - ./mongo-keyfile:/etc/mongo-keyfile:ro
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=password
    networks:
      - ddbp_mongo-network

  # Primer nodo MongoDB secundario
  secondary-mongodb-1:
    image: mongo:latest
    container_name: secondary-mongodb-1
    command: mongod --replSet rs0 --keyFile /etc/mongo-keyfile --
    bind_ip_all --auth --port 27017
    ports:
      - "27018:27017"
    volumes:
      - ./mongo-keyfile:/etc/mongo-keyfile:ro
```

```

environment:
  - MONGO_INITDB_ROOT_USERNAME=admin
  - MONGO_INITDB_ROOT_PASSWORD=password
networks:
  - ddbp_mongo-network
depends_on:
  - central-mongodb

# Segundo nodo MongoDB secundario
secondary-mongodb-2:
  image: mongo:latest
  container_name: secondary-mongodb-2
  command: mongod --replSet rs0 --keyFile /etc/mongo-keyfile --
bind_ip_all --auth --port 27017
  ports:
    - "27019:27017"
  volumes:
    - ./mongo-keyfile:/etc/mongo-keyfile:ro
  environment:
    - MONGO_INITDB_ROOT_USERNAME=admin
    - MONGO_INITDB_ROOT_PASSWORD=password
  networks:
    - ddbp_mongo-network
  depends_on:
    - central-mongodb

```

Ruta: /docker-compose.yml - Configuración de contenedores MongoDB para ReplicaSet

El proceso de inicialización del Replica Set se maneja mediante un contenedor adicional que ejecuta un script para configurar los nodos como un conjunto replicado:

```

# Contenedor de configuración de MongoDB - inicializa el replica
set
mongo-setup:
  image: mongo:latest
  container_name: mongo-setup
  restart: "no"
  depends_on:
    - central-mongodb
    - secondary-mongodb-1
    - secondary-mongodb-2
  networks:
    - ddbp_mongo-network
  volumes:
    - ./setup-replica.sh:/setup-replica.sh:ro
  entrypoint: ["/bin/bash", "/setup-replica.sh"]

```

Ruta: /docker-compose.yml - Configuración de contenedor de inicialización para MongoDB

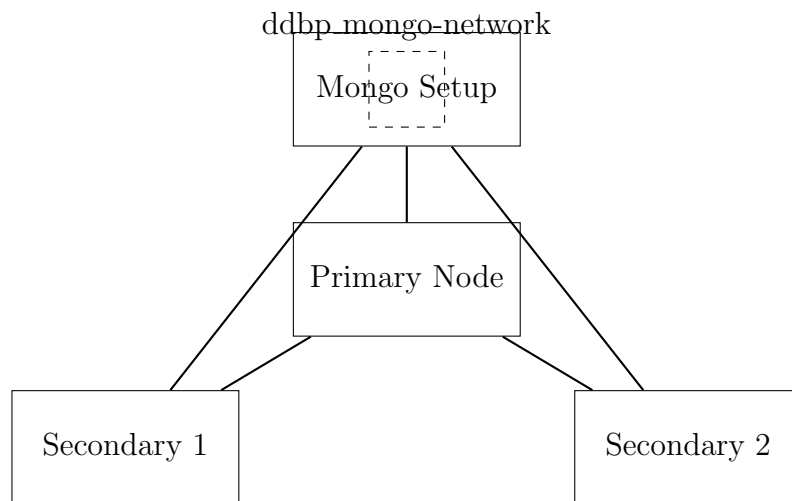


Figura 1: Arquitectura de contenedores MongoDB con Replica Set

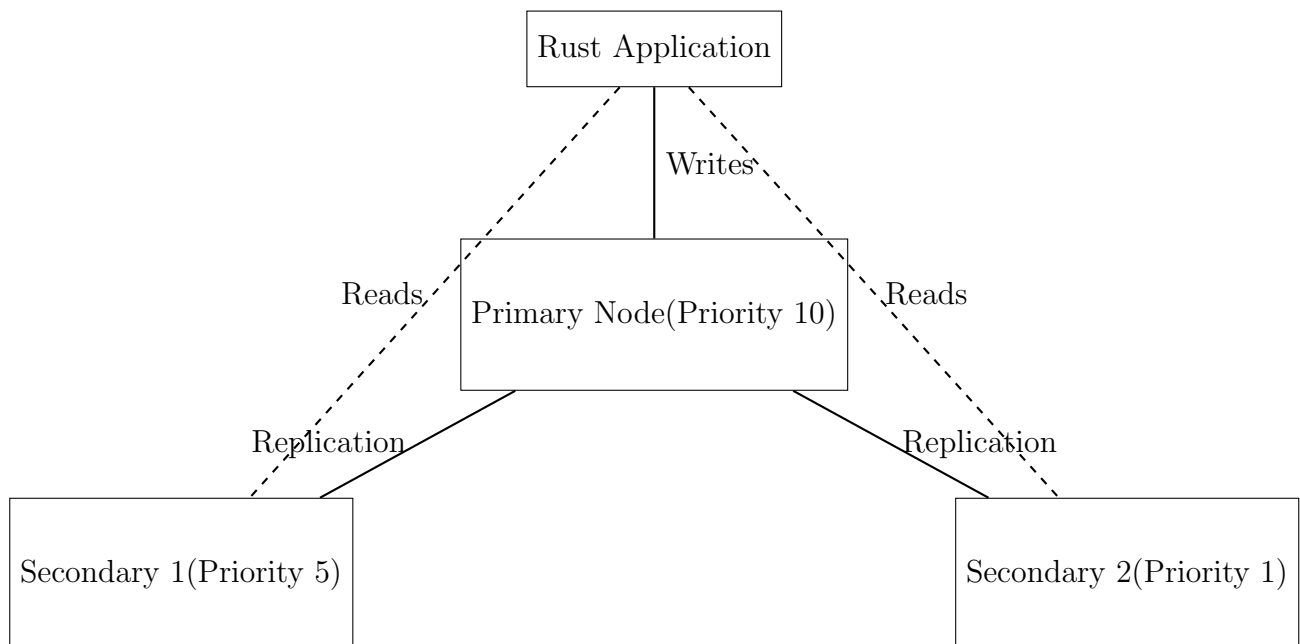


Figura 2: Arquitectura del ReplicaSet MongoDB con esquema de prioridad y flujo de datos

1.1.2. Arquitectura de ReplicaSet y Priorización

La implementación del ReplicaSet en nuestra arquitectura sigue un esquema de priorización estratégica (10-5-1) que garantiza un comportamiento predecible durante las elecciones de nodo primario y la distribución de cargas de trabajo:

- **Nodo Central (Prioridad 10):** Configurado como el servidor preferido para el rol primario, este nodo tiene la prioridad más alta para garantizar estabilidad en condiciones normales de operación.
- **Primer Nodo Secundario (Prioridad 5):** Mantiene una prioridad intermedia, sirviendo como primer candidato de respaldo para asumir el rol primario en caso de fallo del nodo central.
- **Segundo Nodo Secundario (Prioridad 1):** Con la prioridad más baja, este nodo está diseñado principalmente para mantener redundancia y servir lecturas, asumiendo el rol primario solo en situaciones extremas.

La configuración del ReplicaSet se implementa mediante un script de inicialización automatizado:

```
rs.initiate({
  _id: 'rs0',
  members: [
    { _id: 0, host: 'central-mongodb:27017', priority: 10 },
    { _id: 1, host: 'secondary-mongodb-1:27017', priority: 5 },
    { _id: 2, host: 'secondary-mongodb-2:27017', priority: 1 }
  ]
})
```

Ruta: /setup-replica.sh - Configuración de ReplicaSet con esquema de prioridad

Proceso de Elección El proceso de elección de nodo primario se rige por las siguientes reglas:

1. Los nodos participan en una elección cuando:
 - El nodo primario actual se vuelve inaccesible
 - Se produce un reinicio planificado del nodo primario
 - Se detecta una partición de red
2. El nodo con la prioridad más alta disponible gana la elección
3. Se requiere una mayoría de nodos (al menos 2 de 3) para completar una elección

1.1.3. Implementación de Seguridad

La seguridad del ReplicaSet se implementa mediante múltiples capas de protección:

Autenticación por Keyfile Los nodos del ReplicaSet utilizan un archivo de clave compartida para la autenticación interna:

```
command: mongod --replSet rs0 --keyFile /etc/mongo-keyfile --
  bind_ip_all --auth --port 27017
volumes:
  - ./mongo-keyfile:/etc/mongo-keyfile:ro
```

Ruta: /docker-compose.yml - Configuración de seguridad con keyfile

El keyfile se monta como volumen de solo lectura en cada contenedor, garantizando que solo los nodos autorizados puedan unirse al ReplicaSet.

Control de Acceso Basado en Roles (RBAC) Se implementa un esquema de RBAC que define roles específicos para diferentes tipos de operaciones:

```
db.createUser({
  user: 'admin',
  pwd: 'password',
  roles: [
    { role: 'readWrite', db: 'social_media_db' },
    { role: 'dbAdmin', db: 'social_media_db' },
    { role: 'userAdmin', db: 'social_media_db' }
  ]
});
```

Ruta: /setup-replica.sh - Configuración de control de acceso basado en roles

Aislamiento de Red Los contenedores operan en una red dedicada (ddbp_mongo-network) que proporciona:

- Aislamiento del tráfico de base de datos
- DNS interno para resolución de nombres entre contenedores
- Control de acceso a nivel de red

```
networks:
  ddbp_mongo-network:
    name: ddbp_mongo-network
```

Ruta: /docker-compose.yml - Definición de red dedicada para MongoDB

1.1.4. Proceso de Inicialización Automatizada

La inicialización del ReplicaSet se automatiza mediante un contenedor dedicado (mongo-setup) que ejecuta un script de configuración:

1. **Verificación de Disponibilidad:** El script verifica que todos los nodos estén operativos antes de iniciar la configuración.
2. **Inicialización del ReplicaSet:** Se configura la topología del ReplicaSet con las prioridades correspondientes.
3. **Creación de Usuarios y Colecciones:** Se establecen los usuarios administrativos y las colecciones iniciales.
4. **Verificación de Estado:** Se confirma el correcto funcionamiento del ReplicaSet mediante pruebas de estado.

El script de configuración incluye mecanismos de reintento y validación:

```
#!/bin/bash
check_mongodb_ready() {
  local host=$1
  echo "Checking if $host is ready..."
  for i in {1..30}; do
    if mongosh --host $host --port 27017 -u admin -p password \
      --authenticationDatabase admin --eval "db.adminCommand('
ping')" &>/dev/null; then
      echo "$host is ready!"
      return 0
    fi
    echo "Waiting for $host to be ready (attempt $i/30)..."
    sleep 2
  done
  return 1
}
```

Ruta: /setup-replica.sh - Script de verificación de disponibilidad de nodos MongoDB

Esta configuración automatizada garantiza una inicialización consistente y segura del clúster de MongoDB, estableciendo la base para operaciones distribuidas confiables.

1.1.5. Connection Pooling y Resiliencia

La aplicación implementa una configuración robusta para la conexión a MongoDB que garantiza alta disponibilidad, rendimiento óptimo y recuperación ante fallos:

```
// Configurar connection pooling
client_options.max_pool_size = Some(20);
client_options.min_pool_size = Some(5);
client_options.max_idle_time = Some(Duration::from_secs(60));
```

Ruta: /rust-app/src/config.rs - Configuración del pool de conexiones a MongoDB

- **Max Pool Size:** Limita el número máximo de conexiones concurrentes a 20 para prevenir saturación.
- **Min Pool Size:** Mantiene al menos 5 conexiones disponibles para minimizar latencia en solicitudes.
- **Max Idle Time:** Cierra conexiones inactivas después de 60 segundos para optimizar recursos.

```
// Configurar timeouts
client_options.connect_timeout = Some(Duration::from_secs(10));
client_options.server_selection_timeout = Some(Duration::from_secs(15));
```

Ruta: /rust-app/src/config.rs - Configuración de timeouts para conexiones a MongoDB

- **Connect Timeout:** 10 segundos para establecer la conexión inicial.
- **Server Selection Timeout:** 15 segundos para seleccionar un servidor disponible.

```
// Configurar read/write concerns para mejor fiabilidad
client_options.read_concern = Some(ReadConcern::majority());
client_options.write_concern = Some(
    WriteConcern::builder()
        .w(mongodb::options::Acknowledgment::Majority)
        .build(),
);
```

Ruta: /rust-app/src/config.rs - Configuración de consistencia y durabilidad para operaciones de MongoDB

- **Read Concern:** Configurado como `majority` para garantizar que las lecturas devuelvan datos confirmados por la mayoría de nodos.
- **Write Concern:** Configurado para requerir confirmación de la mayoría de nodos antes de considerar una escritura como exitosa.


```
// Establecer preferencia de lectura a SecondaryPreferred
let options = ReadPreferenceOptions::default();
client_options.selection_criteria = Some(
    ReadPreference::SecondaryPreferred {
        options: Some(options),
    }
    .into(),
);
```

Ruta: /rust-app/src/config.rs - Configuración de preferencia de lectura para balanceo de carga

- **Read Preference:** Configurado como **SecondaryPreferred** para dirigir operaciones de lectura a nodos secundarios cuando estén disponibles, descargando así al primario.

```
// Configurar comportamiento de reintentos
client_options.retry_reads = Some(true);
client_options.retry_writes = Some(true);

// Configurar heartbeat para detectar problemas rápidamente
client_options.heartbeat_freq = Some(Duration::from_secs(15));
```

Ruta: /rust-app/src/config.rs - Configuración de políticas de reintento y monitoreo

- **Retry Reads/Writes:** Habilitados para reintentar automáticamente operaciones fallidas.
- **Heartbeat:** Configurado para verificar el estado de los servidores cada 15 segundos.

Esta configuración integral garantiza que la aplicación pueda manejar eficientemente situaciones como:

- Fallos temporales de red
- Caída de nodos individuales
- Elecciones de nuevo primario
- Picos de carga
- Desconexiones transitorias

1.2. Diagrama de Comunicación Rust

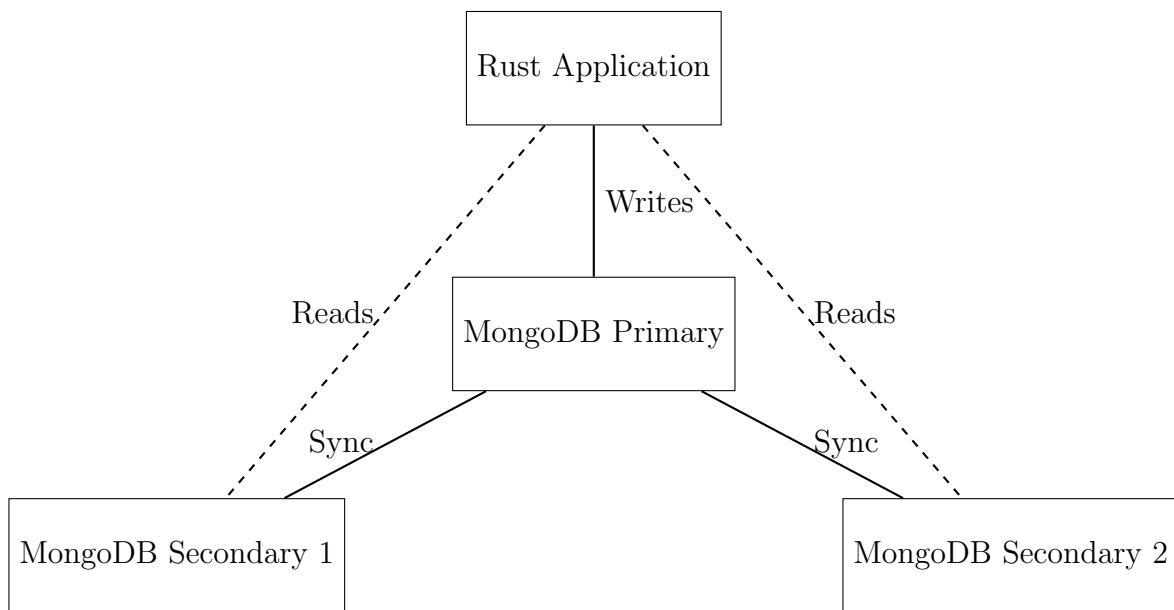


Figura 3: Diagrama de comunicación de la aplicación Rust con MongoDB ReplicaSet

Políticas de Reintento y Monitoreo

2. Casos de Uso

El sistema de Red Social Distribuida basado en MongoDB implementa diversos casos de uso fundamentales que definen la interacción entre usuarios y el sistema. A continuación, se detallan los principales casos de uso organizados por categorías funcionales.

2.1. Autenticación y Gestión de Perfiles

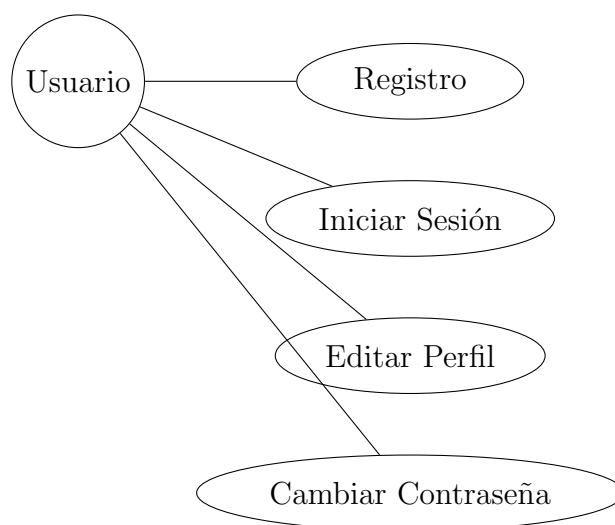


Figura 4: Casos de uso: Autenticación y Gestión de Perfiles

Descripción de Casos de Uso

■ Registro de Usuario:

- **Actor:** Usuario no registrado
- **Descripción:** El usuario proporciona nombre de usuario, correo electrónico y contraseña para crear una cuenta
- **Flujo Principal:**
 1. El usuario ingresa datos personales
 2. El sistema valida la unicidad del nombre de usuario y correo
 3. El sistema almacena credenciales con contraseña hasheada
 4. Se crea perfil de usuario vacío
- **Postcondición:** Cuenta creada y usuario autenticado

■ Inicio de Sesión:

- **Actor:** Usuario registrado
- **Descripción:** El usuario introduce credenciales para acceder
- **Flujo Principal:**
 1. Usuario ingresa nombre de usuario/email y contraseña
 2. Sistema verifica credenciales contra la base de datos
 3. Sistema genera token de autenticación
- **Postcondición:** Usuario autenticado y sesión iniciada

■ Edición de Perfil:

- **Actor:** Usuario autenticado
- **Descripción:** Modificación de información de perfil
- **Flujo Principal:**
 1. Usuario accede a sección de perfil
 2. Modifica campos (bio, avatar, nombre completo)
 3. Sistema valida y almacena cambios
- **Postcondición:** Perfil actualizado en la base de datos

2.2. Gestión de Publicaciones

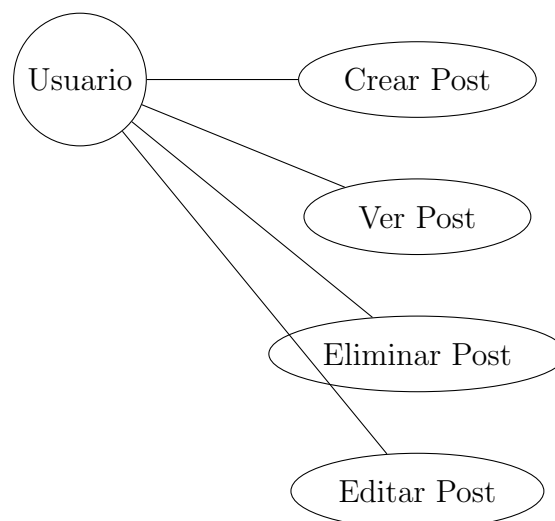


Figura 5: Casos de uso: Gestión de Publicaciones

Descripción de Casos de Uso

■ Crear Publicación:

- **Actor:** Usuario autenticado
- **Descripción:** Creación de nueva publicación con contenido
- **Flujo Principal:**
 1. Usuario redacta contenido textual
 2. Opcionalmente adjunta medios (imágenes, videos)
 3. Opcionalmente agrega etiquetas y ubicación
 4. Sistema almacena publicación y actualiza contadores
- **Postcondición:** Publicación visible para seguidores

■ Ver Publicación:

- **Actor:** Usuario (autenticado o no)
- **Descripción:** Visualización de una publicación específica
- **Flujo Principal:**
 1. Usuario accede a publicación mediante enlace o feed
 2. Sistema recupera publicación con comentarios asociados
 3. Se muestra contenido completo e interacciones
- **Postcondición:** Publicación visualizada con estadísticas actualizadas

■ Eliminar Publicación:

- **Actor:** Usuario autenticado (propietario)
- **Descripción:** Eliminación permanente de una publicación
- **Flujo Principal:**
 1. Usuario selecciona opción eliminar en su publicación
 2. Sistema solicita confirmación
 3. Sistema elimina publicación y actualiza contadores
- **Postcondición:** Publicación eliminada y referencias actualizadas

2.3. Interacciones Sociales

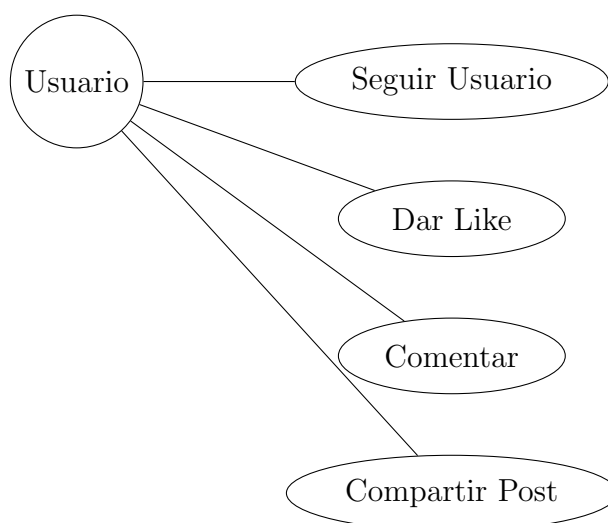


Figura 6: Casos de uso: Interacciones Sociales

Descripción de Casos de Uso

■ Seguir Usuario:

- **Actor:** Usuario autenticado
- **Descripción:** Establecer relación de seguimiento
- **Flujo Principal:**
 1. Usuario visita perfil de otro usuario
 2. Usuario selecciona opción "Seguir"
 3. Sistema registra relación en colección de follows
 4. Contadores de ambos usuarios son actualizados
- **Postcondición:** Relación establecida y contadores actualizados

■ Dar Like:

- **Actor:** Usuario autenticado
- **Descripción:** Interacción positiva con publicación
- **Flujo Principal:**
 1. Usuario visualiza publicación
 2. Usuario selecciona opción "Me gusta"
 3. Sistema registra interacción y actualiza contador
- **Postcondición:** Like registrado y contador actualizado

■ Comentar:

- **Actor:** Usuario autenticado
- **Descripción:** Añadir comentario a publicación
- **Flujo Principal:**
 1. Usuario visualiza publicación
 2. Usuario redacta comentario
 3. Sistema almacena comentario asociado a la publicación
 4. Contador de comentarios se actualiza
- **Postcondición:** Comentario visible y contador actualizado

2.4. Gestión de Feed

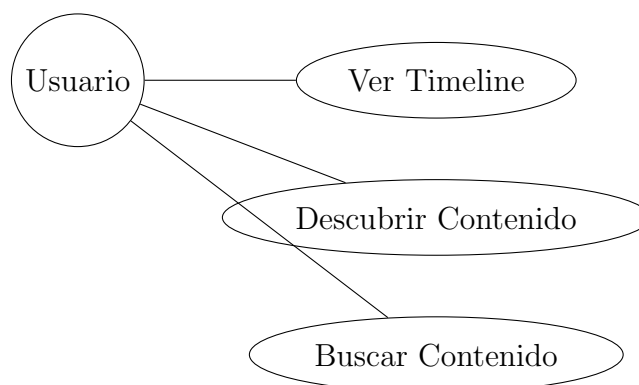


Figura 7: Casos de uso: Gestión de Feed

Descripción de Casos de Uso

- **Ver Timeline:**

- **Actor:** Usuario autenticado
- **Descripción:** Visualización de publicaciones recientes de usuarios seguidos
- **Flujo Principal:**
 1. Usuario accede a la página principal
 2. Sistema recupera publicaciones de usuarios seguidos
 3. Sistema ordena publicaciones cronológicamente
 4. Se muestra feed personalizado
- **Postcondición:** Feed actualizado visualizado

- **Descubrir Contenido:**

- **Actor:** Usuario autenticado
- **Descripción:** Exploración de contenido popular o recomendado
- **Flujo Principal:**
 1. Usuario accede a sección "Descubrir"
 2. Sistema selecciona contenido popular o relevante
 3. Se presenta contenido de usuarios no seguidos
- **Postcondición:** Usuario expuesto a nuevo contenido

- **Buscar Contenido:**

- **Actor:** Usuario
- **Descripción:** Búsqueda por términos específicos
- **Flujo Principal:**
 1. Usuario ingresa términos de búsqueda
 2. Sistema busca en publicaciones, perfiles y etiquetas
 3. Resultados mostrados por relevancia
- **Postcondición:** Resultados relevantes presentados al usuario

3. Stack Tecnológico

El sistema de Red Social Distribuida se construye sobre una arquitectura moderna compuesta por múltiples tecnologías integradas, cada una seleccionada por sus fortalezas específicas que contribuyen al rendimiento, escalabilidad y mantenibilidad del sistema.

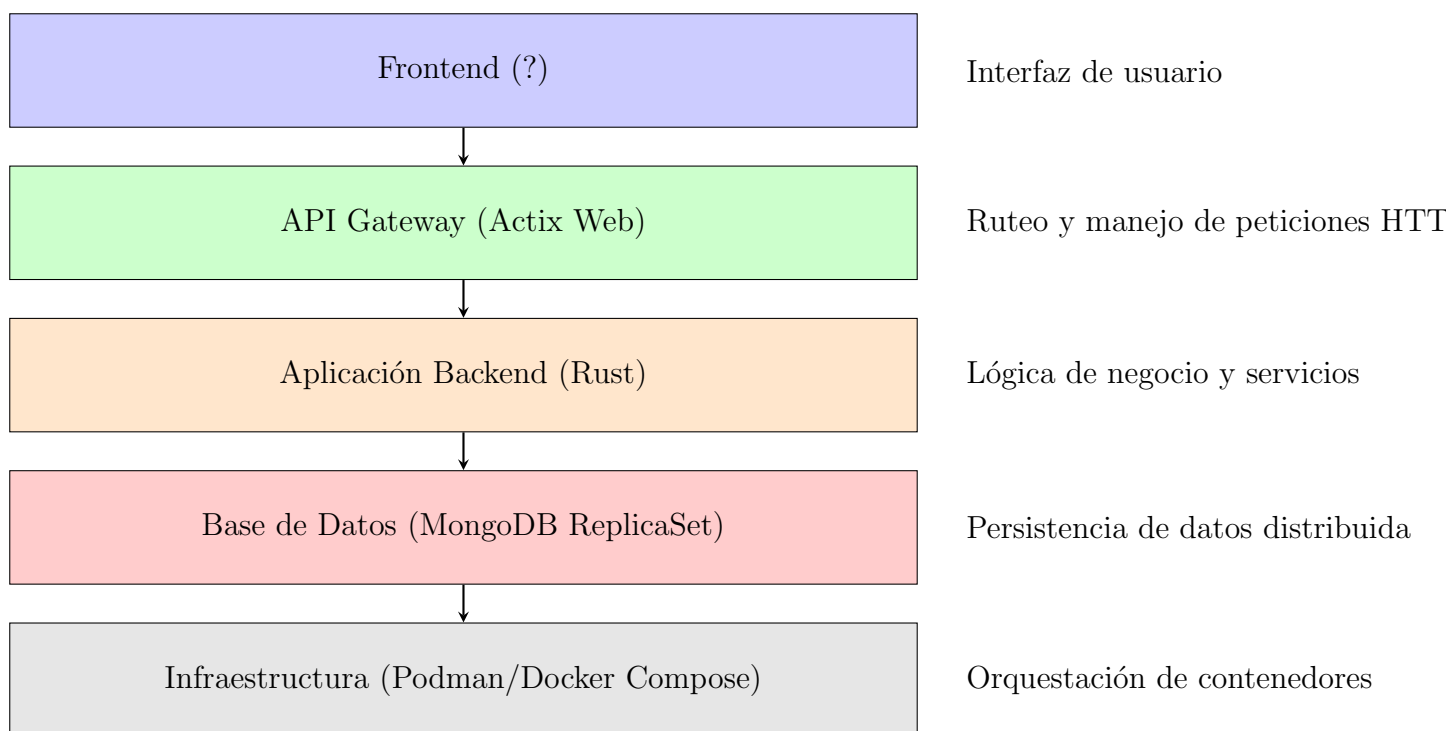


Figura 8: Arquitectura de capas del stack tecnológico

3.1. Componentes del Stack Tecnológico

3.1.1. Lenguajes y Frameworks

- **Rust**: Lenguaje de programación de alto rendimiento y seguridad de memoria utilizado para el desarrollo del backend.
- **Actix Web**: Framework asíncrono para Rust que permite construir aplicaciones web rápidas y eficientes.
- **MongoDB**: Base de datos NoSQL orientada a documentos, utilizada para almacenar datos de manera flexible y escalable.
- **Podman**: Plataforma de contenedorización que permite empaquetar aplicaciones y sus dependencias en contenedores ligeros.
- **Podman Compose**: Herramienta para definir y ejecutar aplicaciones multicontenedor, facilitando la orquestación de servicios.
- **Tokio**: Runtime asíncrono para Rust que permite la ejecución eficiente de operaciones de entrada/salida.
- **Serde**: Biblioteca de serialización/deserialización para Rust, utilizada para convertir estructuras de datos a y desde JSON.
- **Yaak**: Herramienta para testear las APIs de Actix Web, permitiendo visualizar y manejar las peticiones y respuestas de manera sencilla.

3.2. Configuración de MongoDB

3.2.1. Configuración del Replica Set

3.2.2. Connection Pooling y Resiliencia

3.3. Estructura de la Aplicación Rust

3.3.1. API Endpoints y Funcionalidades

3.3.2. Manejo de Errores y Responses

3.3.3. Modelos de Datos

4. Modelos de Datos

4.1. Diseño de Colecciones

El sistema implementa un modelo de datos distribuido optimizado para operaciones sociales y escalabilidad horizontal. Las colecciones están diseñadas para maximizar la eficiencia en un entorno distribuido mientras mantienen la consistencia de los datos.

4.1.1. Colección de Usuarios

```
{
  "_id": ObjectId(),
  "username": "string",           // nico , indexado
  "email": "string",             // nico , indexado
  "password_hash": "string",
  "profile": {
    "full_name": "string",
    "bio": "string",
    "avatar_url": "string",
    "created_at": ISODate(),
    "last_login": ISODate()
  },
  "followers_count": NumberInt, // Contador denormalizado
  "following_count": NumberInt, // Contador denormalizado
  "posts_count": NumberInt      // Contador denormalizado
}
```

Índices y Optimización

- Índice único en username: {username: 1}
- Índice único en email: {email: 1}
- Índice compuesto para búsquedas: {username: 1, email: 1}

4.1.2. Colección de Posts

```
{
  "_id": ObjectId(),
  "user_id": ObjectId(),         // Referencia al autor
```



```

    "content": "string",
    "media_urls": ["string"],    // URLs de contenido multimedia
    "created_at": ISODate(),
    "likes_count": NumberInt,    // Contador denormalizado
    "comments_count": NumberInt, // Contador denormalizado
    "tags": ["string"],
    "location": {
        "type": "Point",
        "coordinates": [NumberDouble, NumberDouble]
    }
}

```

Índices y Optimización

- Índice en user_id: {user_id: 1}
- Índice en created_at: {created_at: -1}
- Índice geoespacial: {location: "2dsphere"}
- Índice de texto en contenido: {content: "text", tags: "text"}

4.1.3. Colección de Comentarios

```

{
    "_id": ObjectId(),
    "post_id": ObjectId(),    // Referencia al post
    "user_id": ObjectId(),    // Referencia al autor
    "content": "string",
    "created_at": ISODate(),
    "likes_count": NumberInt, // Contador denormalizado
    "parent_comment_id": ObjectId() // Para comentarios anidados
}

```

Índices y Optimización

- Índice compuesto: {post_id: 1, created_at: -1}
- Índice en user_id: {user_id: 1}
- Índice en parent_comment_id: {parent_comment_id: 1}

4.1.4. Colección de Likes

```

{
    "_id": ObjectId(),
    "user_id": ObjectId(),
    "target_id": ObjectId(), // ID del post o comentario
    "target_type": "string", // "post" o "comment"
    "created_at": ISODate()
}

```

4.1.5. Colección de Follows

```
{
  "_id": ObjectId(),
  "follower_id": ObjectId(),
  "following_id": ObjectId(),
  "created_at": ISODate()
}
```

4.2. Estrategias de Modelado de Datos

4.2.1. Denormalización Estratégica

El sistema implementa denormalización selectiva para optimizar las operaciones de lectura más comunes:

- **Contadores Precalculados:** Mantenimiento de contadores en documentos de usuario para followers, following y posts.
- **Feed Desnormalizado:** Almacenamiento de información frecuentemente accedida en la colección de feed.
- **Información de Perfil Embebida:** Datos de perfil almacenados directamente en el documento de usuario.

4.2.2. Gestión de Relaciones

Las relaciones entre entidades se manejan mediante referencias, optimizando para:

- **Escalabilidad:** Las referencias permiten que los documentos crezcan independientemente.
- **Flexibilidad:** Facilita la modificación de esquemas y la evolución de la aplicación.
- **Consultas Eficientes:** Permite agregaciones y búsquedas optimizadas.

4.3. Distribución de Datos

La distribución de datos en el ReplicaSet se optimiza para garantizar:

Consistencia

- Escrituras confirmadas por mayoría de nodos
- Lecturas consistentes desde secundarios
- Manejo de conflictos mediante timestamps

Disponibilidad

- Replicación automática entre nodos
- Failover transparente para aplicaciones cliente
- Recuperación automática de nodos caídos

Particionamiento Aunque el sistema actual utiliza un ReplicaSet sin sharding, está diseñado para facilitar la futura implementación de sharding horizontal:

- Claves de documento distribuibles (`_id`)
- Índices preparados para sharding
- Modelo de datos compatible con distribución

4.4. Estrategias de Indexación

La estrategia de indexación está diseñada para optimizar los patrones de acceso más comunes:

- **Índices Únicos:** Garantizan la unicidad de usernames y emails
- **Índices Compuestos:** Optimizan consultas multi-campo frecuentes
- **Índices Geoespaciales:** Facilitan búsquedas basadas en ubicación
- **Índices de Texto:** Permiten búsquedas eficientes en contenido

Esta estructura de datos distribuida proporciona una base sólida para las operaciones de la red social, permitiendo:

- Escalabilidad horizontal futura
- Alta disponibilidad de datos
- Rendimiento optimizado para operaciones frecuentes
- Flexibilidad para evolución del sistema

5. Endpoints de la API

5.1. Operaciones de Usuario

La API expone endpoints para gestionar operaciones relacionadas con usuarios:

5.1.1. Crear Usuario

```
POST /create_user
Content-Type: application/json

{
  "username": "string",
  "email": "string",
  "password": "string",
  "profile": {
    "full_name": "string",
    "bio": "string",
    "avatar_url": "string"
  }
}
```

Este endpoint:

- Valida datos de entrada

- Verifica unicidad de username y email
- Hashea la contraseña de forma segura
- Crea el documento de usuario en MongoDB
- Retorna el ID del usuario creado

5.1.2. Perfiles de Usuario

GET /user/{user_id}

Response:

```
{
  "user_id": "ObjectId",
  "username": "string",
  "profile": {
    "full_name": "string",
    "bio": "string",
    "avatar_url": "string",
    "created_at": "ISODate",
    "last_login": "ISODate"
  },
  "stats": {
    "followers_count": number,
    "following_count": number,
    "posts_count": number
  }
}
```

Funcionalidades:

- Recuperación eficiente de perfiles
- Proyección de campos según permisos
- Caché de perfiles frecuentes
- Estadísticas en tiempo real

5.2. Interacciones Sociales

La API proporciona endpoints para gestionar las interacciones entre usuarios:

5.2.1. Crear Posts

POST /create_post

Content-Type: application/json

```
{
  "user_id": "ObjectId",
  "content": "string",
  "media_urls": ["string"],
  "tags": ["string"],
  "location": {
    "type": "Point",
```

```
    "coordinates": [double, double]
  }
}
```

Características del endpoint:

- Validación de usuario existente
- Procesamiento de contenido multimedia
- Actualización atómica de contadores
- Indexación de contenido para búsqueda

5.2.2. Funcionalidad de Comentarios

POST /create_comment

Content-Type: application/json

```
{
  "post_id": "ObjectId",
  "user_id": "ObjectId",
  "content": "string",
  "parent_comment_id": "ObjectId" // Opcional
}
```

El endpoint maneja:

- Comentarios en posts
- Respuestas a comentarios existentes
- Actualización de contadores de comentarios
- Notificaciones a usuarios relevantes

5.2.3. Relaciones de Follow

POST /follow_user

Content-Type: application/json

```
{
  "follower_id": "ObjectId",
  "following_id": "ObjectId"
}
```

Implementación:

- Verificación de usuarios existentes
- Validación de relación no duplicada
- Actualización atómica de contadores
- Generación de eventos de notificación

5.3. Operaciones del Sistema

La API incluye endpoints para monitoreo y mantenimiento del sistema:

5.3.1. Implementación del Health Check

```
GET /health
```

Response:

```
{
  "status": "Healthy",
  "mongo_status": {
    "primary": "connected",
    "secondaries": ["connected", "connected"],
    "replication_lag": [0, 1]
  }
}
```

El health check verifica:

- Conectividad con MongoDB
- Estado del ReplicaSet
- Lag de replicación
- Estado del pool de conexiones

5.3.2. Población de Base de Datos para Testing

```
POST /test/populate
```

Content-Type: application/json

```
{
  "users_count": number,
  "posts_per_user": number,
  "comments_per_post": number
}
```

Funcionalidades:

- Generación de datos de prueba realistas
- Creación de relaciones entre entidades
- Distribución estadística de interacciones
- Verificación de integridad de datos

5.3.3. Funcionalidad de Limpieza de Base de Datos

```
POST /test/clean
```

Response:

```
{
  "status": "success",
  "collections_cleaned": ["users", "posts", "comments", "likes", "follows"]
}
```

El endpoint:

- Limpia datos de prueba
- Mantiene configuración del sistema
- Verifica integridad post-limpieza
- Registra operación en logs

Todos los endpoints implementan:

- Manejo robusto de errores
- Validación de entrada
- Logging de operaciones
- Métricas de rendimiento
- Control de acceso basado en roles

6. Implementación Técnica

6.1. Error Handling

La implementación incluye un sistema robusto de manejo de errores que abarca múltiples niveles de la aplicación:

```
#[derive(Debug)]
pub enum AppError {
    // Errores de Base de Datos
    DatabaseError(mongodb::error::Error),
    ConnectionError(String),

    // Errores de Validación
    ValidationError(String),
    DuplicateKeyError(String),

    // Errores de Negocio
    NotFoundError(String),
    UnauthorizedError(String),

    // Errores del Sistema
    SystemError(String)
}
```

Ruta: /rust-app/src/error.rs - Enumeración de errores para manejo centralizado de excepciones

Estrategias de Manejo

- **Propagación Controlada:** Los errores se propagan de manera controlada mediante Result
- **Logging Estructurado:** Registro detallado de errores con contexto
- **Recuperación Automática:** Reintentos automáticos para errores transitorios
- **Respuestas HTTP:** Mapeo de errores a códigos HTTP apropiados

6.2. State Management

La gestión del estado de la aplicación se maneja mediante una estructura AppState que encapsula recursos compartidos:

```
pub struct AppState {
    pub db: Database,
    pub config: AppConfig,
    pub metrics: Arc<Metrics>,
}

impl AppState {
    pub async fn new(mongo_uri: &str, config: AppConfig) -> Result<
        Self, AppError> {
        let client_options = ClientOptions::parse(mongo_uri).await?;
        let client = Client::with_options(client_options)?;
        let db = client.database("social_media_db");

        Ok(Self {
            db,
            config,
            metrics: Arc::new(Metrics::new()),
        })
    }
}
```

Ruta: /rust-app/src/state.rs - Implementación del estado de la aplicación y gestión de recursos compartidos

Características Principales

- **Estado Compartido:** Acceso thread-safe a recursos compartidos
- **Configuración Centralizada:** Gestión unificada de configuración
- **Métricas en Tiempo Real:** Recolección de métricas de rendimiento
- **Conexión a Base de Datos:** Pool de conexiones administrado

6.3. Conexión a Base de Datos y Resiliencia

La conexión a MongoDB se implementa con énfasis en la resiliencia y rendimiento:

```
async fn setup_database(config: &Config) -> Result<Database, AppError> {
    {
        let mut client_options = ClientOptions::parse(&config.mongo_uri).
            await?;

        // Configuración de timeouts
        client_options.connect_timeout = Some(Duration::from_secs(10));
        client_options.server_selection_timeout = Some(Duration::from_secs
            (15));

        // Configuración de pool de conexiones
        client_options.max_pool_size = Some(20);
        client_options.min_pool_size = Some(5);
    }
}
```



```
// Configuración de consistencia
client_options.read_concern = Some(ReadConcern::majority());
client_options.write_concern = Some(
    WriteConcern::builder()
        .w(mongodb::options::Acknowledgment::Majority)
        .build(),
);

let client = Client::with_options(client_options)?;
Ok(client.database("social_media_db"))
}
```

Ruta: /rust-app/src/main.rs - Configuración y conexión a la base de datos MongoDB

Características del Testing

- **Base de Datos de Prueba:** Instancia dedicada para testing
- **Fixtures Automatizados:** Generación de datos de prueba
- **Limpieza Automática:** Restauración del estado inicial
- **Tests de Integración:** Pruebas end-to-end con base de datos real

Esta implementación técnica asegura:

- Alta calidad del código
- Mantenibilidad a largo plazo
- Facilidad de debugging
- Confiabilidad del sistema

7. Manejo de Consistencia y Durabilidad

7.1. Configuración de Garantías de Consistencia

La arquitectura implementa un modelo de consistencia fuerte mediante configuraciones específicas que garantizan la durabilidad y consistencia de los datos en el entorno distribuido:

Write Concern Mayoría La configuración de write concern asegura que las operaciones de escritura sean confirmadas por la mayoría de los nodos:

```
client_options.write_concern = Some(
    WriteConcern::builder()
        .w(mongodb::options::Acknowledgment::Majority)
        .build(),
);
```

Ruta: /rust-app/src/config.rs - Configuración de garantías de escritura para durabilidad

Esta configuración proporciona:

- Garantía de durabilidad ante fallos de nodos
- Consistencia entre réplicas para operaciones posteriores
- Protección contra pérdida de datos en escenarios de failover

Read Concern Mayoría Las operaciones de lectura se configuran para garantizar consistencia:

```
client_options.read_concern = Some(ReadConcern::majority());
```

Ruta: /rust-app/src/config.rs - Configuración de garantías de lectura para consistencia

Este nivel de read concern asegura:

- Lecturas de datos confirmados por la mayoría de réplicas
- Prevención de lecturas sucias (dirty reads)
- Consistencia en operaciones distribuidas

8. Configuración de Resiliencia

8.1. Gestión de Timeouts

```
client_options.connect_timeout = Some(Duration::from_secs(10));
client_options.server_selection_timeout = Some(Duration::from_secs(15));
client_options.max_idle_time = Some(Duration::from_secs(60));
```

Ruta: /rust-app/src/config.rs - Configuración de timeouts operacionales para resiliencia

Los timeouts se configuran para:

- **Connect Timeout (10s)**: Límite para conexiones iniciales
- **Server Selection (15s)**: Tiempo máximo para selección de servidor
- **Max Idle Time (60s)**: Duración máxima de conexiones inactivas

8.2. Políticas de Reintento

```
client_options.retry_reads = Some(true);
client_options.retry_writes = Some(true);
```

Ruta: /rust-app/src/config.rs - Configuración de políticas de reintento para operaciones

Las operaciones elegibles para reintento incluyen:

- Errores de red transitorios
- Timeouts de operaciones
- Errores de escritura no fatales
- Reconexiones post-failover

8.3. Monitoreo y Health Checks

Timeouts Operacionales

```
client_options.heartbeat_freq = Some(Duration::from_secs(15));
```

Ruta: /rust-app/src/config.rs - Configuración de frecuencia de heartbeat para monitoreo

La implementación incluye:

- **Heartbeat:** Verificación periódica de nodos
- **Métricas de Replicación:** Monitoreo de lag y sincronización
- **Estadísticas Operacionales:** Seguimiento de latencia y throughput

```
async fn health_check_handler(  
    app_state: web::Data<AppState>  
) -> impl Responder {  
    match app_state.db.run_command(doc! {"ping": 1}, None).await {  
        Ok(_) => HttpResponse::Ok().json("Healthy"),  
        Err(e) => HttpResponse::ServiceUnavailable().json(e.to_string  
        ()),  
    }  
}
```

Ruta: /rust-app/src/handlers.rs - Manejador de verificación de salud del sistema

El sistema implementa verificaciones de:

- Conectividad a MongoDB
- Estado del pool de conexiones
- Métricas de rendimiento
- Uso de recursos

8.4. Balanceo de Carga

Health Checks de Aplicación La aplicación implementa estrategias de balanceo en múltiples niveles:

```
client_options.selection_criteria = Some(  
    ReadPreference::SecondaryPreferred {  
        options: Some(ReadPreferenceOptions::default()),  
    }  
    .into(),  
);
```

Ruta: /rust-app/src/config.rs - Configuración de criterios de selección para balanceo de carga

Características principales:

- Distribución de lecturas entre secundarios
- Escrituras centralizadas en primario
- Failover automático

Nivel de Aplicación

- Connection pooling optimizado (5-20 conexiones)
- Distribución de carga entre instancias
- Caché local para datos frecuentes

8.5. Error Handling

La aplicación implementa un manejo de errores multinivel:

- **Errores de Conexión:** Problemas de red o disponibilidad
- **Errores de Operación:** Fallos en operaciones específicas
- **Errores de Aplicación:** Problemas de lógica de negocio
- **Errores de Sistema:** Fallos de recursos o configuración

Estrategias de recuperación:

- **Circuit Breaker:** Prevención de cascada de fallos
- **Backoff Exponencial:** Reintento gradual
- **Fallback:** Rutas alternativas para operaciones críticas
- **Logging:** Registro detallado para análisis

9. Conclusiones

La implementación del sistema de base de datos distribuida MongoDB para nuestra aplicación de red social demuestra una arquitectura robusta y escalable que cumple con los requisitos de alta disponibilidad, consistencia y rendimiento. Los aspectos más destacados de la implementación incluyen:

9.1. Arquitectura Distribuida

- Implementación exitosa de un ReplicaSet MongoDB con esquema de prioridad 10-5-1
- Sistema de failover automático que garantiza continuidad operativa
- Balanceo de carga efectivo entre nodos primario y secundarios
- Aislamiento de red y seguridad mediante autenticación por keyfile

9.2. Garantías de Datos

- Consistencia fuerte mediante write concern y read concern mayoritarios
- Durabilidad asegurada por replicación síncrona entre nodos
- Prevención de pérdida de datos mediante confirmación distribuida
- Estrategias de normalización optimizadas para lecturas frecuentes

9.3. Resiliencia y Monitoreo

- Sistema robusto de manejo de errores y recuperación
- Monitoreo continuo mediante heartbeats y health checks
- Connection pooling optimizado para rendimiento sostenido
- Políticas de reintento configuradas para operaciones críticas

9.4. Escalabilidad

- Diseño preparado para futuro sharding horizontal
- Modelo de datos optimizado para distribución
- Índices estratégicos para consultas frecuentes
- Arquitectura modular que facilita expansión

El sistema resultante proporciona una base sólida para las operaciones de la red social, con capacidad de crecimiento y adaptación a futuros requerimientos. La combinación de MongoDB como sistema de base de datos distribuida y Rust como lenguaje de implementación ha demostrado ser efectiva para crear una aplicación robusta y de alto rendimiento.

Las decisiones arquitectónicas tomadas en términos de consistencia, disponibilidad y tolerancia a particiones (CAP) han priorizado la consistencia y disponibilidad, mientras se mantiene la capacidad de escalar horizontalmente cuando sea necesario. La implementación actual sienta las bases para futuras mejoras y optimizaciones según evolucionen los requerimientos del sistema.