DOCUMENTACIÓN

Patrón de diseño: Singleton (Spring Boot + HikariCP)

Clasificación: Creacional, su propósito radica en controlar el proceso de creación de objetos.
Objetivo: Garantizar una única instancia del recurso compartido (pool de conexiones) a la BD en toda la app**.**

Contexto: Sin el uso del patrón singleton, cada componente podría abrir conexiones propias, lo que conlleva a un agotamiento de recursos, fugas y latencia. La solución propuesta es centralizar la creación y gestión de conexiones mediante un *pool* expuesto como un único DataSource gestionado por el contenedor (Spring). En Spring, los beans (objetos creados, configurados y administrados de manera automática por Spring) por defecto son *singleton scope,* por lo que DataSource ya es un singleton a nivel de contenedor.

Fuerzas/Restricciones:

- **Concurrencia alta:** múltiples request simultáneos.
- **Rendimiento:** minimizar costos de abrir/cerrar conexiones.
- **Mantenibilidad:** centralizar URL/credenciales/timeout.
- **Observabilidad:** ver estado del pool.

Participantes y responsabilidades:

- **Spring ApplicationContext:** crea y mantiene una instancia de DataSource.
- **DataSource (HikariDataSource):** administra el pool (tamaños, timeouts, métricas).
- **Repositorios/Servicios:** piden conexiones al DataSource (a través de JPA/Hibernate) sin preocuparse por su ciclo de vida.

Flujos:

- Al iniciar la app, Spring lee application.yml y crea un único HikariDataSource.
- Un controlador → servicio → repositorio invoca JPA.
- JPA/Hibernate solicita una conexión al DataSource → Hikari entrega una del pool.
- Al terminar la transacción, la conexión se devuelve al pool (no se cierra físicamente).

A continuación se presenta un diagrama realizado con ayuda de IA para mostrar de forma gráfica el flujo:

Diagrama_de_secuencia

Implementación en el codigo:

Como un ejemplo tangible de la implementación en el codigo, se tiene que cuando se invoca el método repo.save(u) de la clase UsuarioService, la llamada corresponde a la interfaz JpaRepository, cuya implementación es proporcionada por la clase SimpleJpaRepository de

Spring Data JPA. Este método delega la operación al EntityManager, que actúa como una fachada JPA sobre la Session de Hibernate. A partir de allí, Hibernate determina si debe ejecutar un persist (para entidades nuevas) o un merge (para entidades existentes), registra la entidad en el contexto de persistencia y genera la sentencia INSERT al momento del flush o commit, utilizando una conexión del pool HikariDataSource. En este flujo también intervienen componentes como JpaTransactionManager y SessionImpl, los cuales coordinan la transacción y la interacción con la base de datos. Finalmente, se evidencia el uso del patrón Singleton, ya que tanto el EntityManagerFactory como el HikariDataSource se instancian una sola vez durante el ciclo de vida de la aplicación, sirviendo como punto único de acceso a los recursos de persistencia.

Ejemplo: UsuarioService

```
public class UsuarioService {

    private final UsuarioRepository repo;  4 usages

    public UsuarioService(UsuarioRepository repo) { this.repo = repo; }

    @Transactional  1 usage  julianAlbarra547
    public Usuario crear(Usuario u) {
        if (repo.existsByEmailIgnoreCase(u.getEmail())) {
            throw new IllegalArgumentException("El email ya está registrado");
        }
        return repo.save(u);
    }

    public List<Usuario> listar() { return repo.findAll(); }
}
```

UsuarioRepository

```
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {  7
    Optional<Usuario> findByEmailIgnoreCase(String email);  no usages  julianAlb
    boolean existsByEmailIgnoreCase(String email);  1 usage  julianAlbarra547
}
```

HikariDataSource

```java
public class HikariDataSource extends HikariConfig implements DataSource, Closeable {

    public Connection getConnection() throws SQLException {
        if (this.isClosed()) {
            throw new SQLException("HikariDataSource " + this + " has been closed.");
        } else if (this.fastPathPool != null) {
            return this.fastPathPool.getConnection();
        } else {
            HikariPool result = this.pool;
            if (result == null) {
                synchronized (this) {
                    result = this.pool;
                    if (result == null) {
                        this.validate();
                        LOGGER.info("{} - Starting...", this.getPoolName());

                        try {
                            this.pool = result = new HikariPool( config: this);
                            this.seal();
                        } catch (HikariPool.PoolInitializationException pie) {
                            if (pie.getCause() instanceof SQLException) {
                                throw (SQLException) pie.getCause();
                            }

                            throw pie;
                        }
```

Patrón de diseño: Builder

El patrón Builder se emplea en el módulo de *Rutina* para construir objetos complejos de entrenamiento (que incluyen ejercicios, series, repeticiones y pesos) de forma modular, flexible y extensible, a partir de la información disponible en la base de datos del sistema.

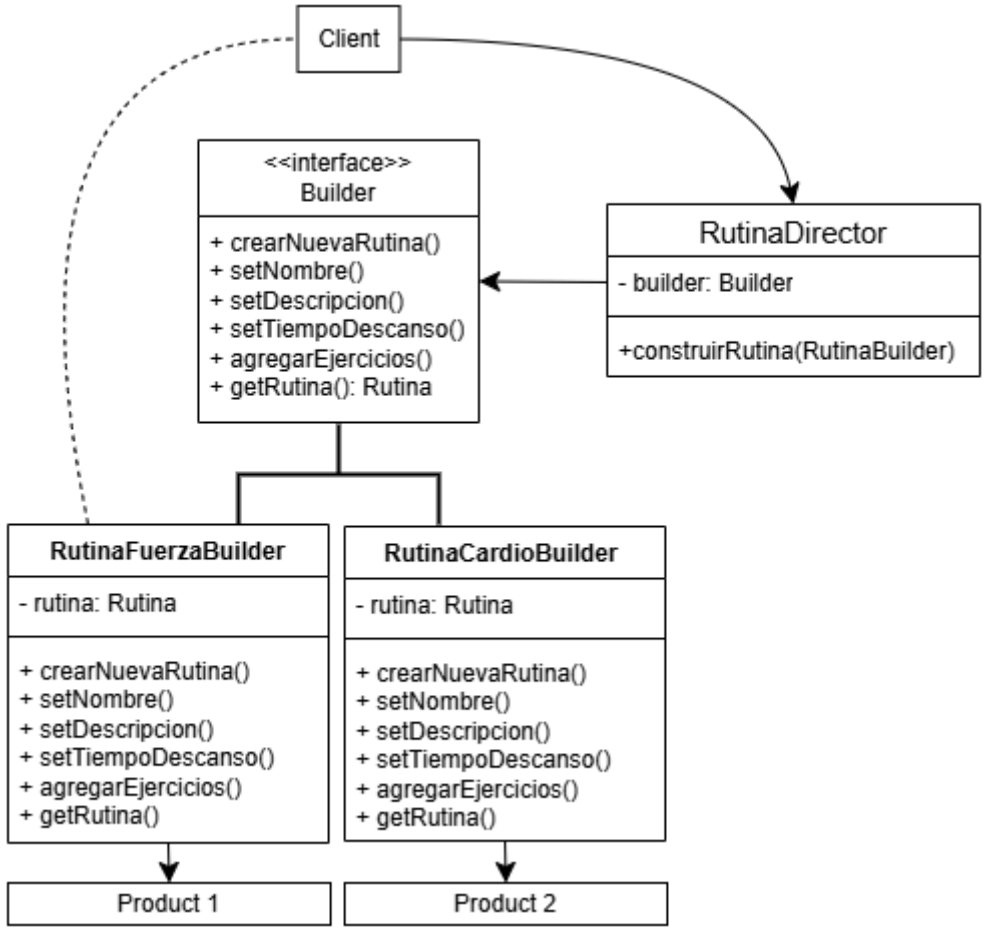
En nuestro proyecto, la entidad **Rutina** se relaciona con:

- Ejercicio
- Set (series y repeticiones de cada ejercicio)
- Perfil (usuario propietario de la rutina)

El objetivo es diseñar e implementar un mecanismo flexible que permita construir rutinas personalizadas combinando información proveniente de las entidades Rutina, Ejercicio y Set. Para lograrlo, se aplicará el patrón de diseño Builder, el cual facilita la separación entre el proceso de construcción (los pasos necesarios para formar una rutina) y la representación final, que corresponde al tipo de rutina resultante. De esta forma, se consigue una arquitectura más modular, escalable y fácil de mantener, donde la creación de rutinas se puede adaptar a diferentes perfiles o necesidades del usuario sin alterar la estructura del código existente.
Se propuso la siguiente estructura para el patrón de diseño Builder:

Rol del patrón	Clase	Descripción
Product	Rutina	Representa la rutina final con ejercicios y sets asociados.
Builder	RutinaBuilder	Define los pasos para construir una rutina.
Concrete Builders	RutinaFuerzaBuilder, RutinaCardioBuilder	Implementan los pasos específicos para construir rutinas de distintos tipos.
Director	RutinaDirector	Define el orden de construcción de la rutina.
Client	Main o RutinaService	Solicita al director que construya una rutina usando un builder específico.

El diagrama UML que ilustra este Builder es el siguiente:



En este caso product 1 y 2 son rutinas

Ventajas del Builder en Rocky

- Permite crear rutinas personalizadas con distintos tipos de ejercicios y sets.
- Facilita la extensión (nuevos tipos de rutinas sin modificar código existente).
- Mantiene una estructura limpia y desacoplada.
- Reduce errores en la construcción de objetos complejos.

El patrón Builder implementado en el módulo *Rutina* demuestra cómo una arquitectura modular permite manejar la complejidad de los objetos compuestos en aplicaciones de entrenamiento.
Además, facilita la integración con la base de datos relacional existente, manteniendo el código orientado a objetos y flexible para futuras extensiones (por ejemplo, Rutina HIIT, Rutina de Resistencia, etc.)

Implementación del patrón en nuestro proyecto

1. **Rutina:**

```java
public class Rutina {
    private String nombre;
    private String nivel;
    private List<String> ejercicios;
    private int duracion; // en minutos

    // Constructor vacío
    public Rutina() {}

    // Métodos set
    public void setNombre(String nombre) { this.nombre = nombre; }
    public void setNivel(String nivel) { this.nivel = nivel; }
    public void setEjercicios(List<String> ejercicios) { this.ejercicios = ejercicios; }
    public void setDuracion(int duracion) { this.duracion = duracion; }

    @Override
    public String toString() {
        return "Rutina: " + nombre + " (" + nivel + ")\n"
             + "Ejercicios: " + ejercicios + "\n"
             + "Duración: " + duracion + " min";
    }
}
```

2. **Builder:**

```java
public interface RutinaBuilder {
    void reset();
    void setNombre(String nombre);
    void setNivel(String nivel);
    void setEjercicios(List<String> ejercicios);
    void setDuracion(int duracion);
    Rutina getResult();
}
```

3. **Concrete Builder:**

```java
public class RutinaPersonalizadaBuilder implements RutinaBuilder {
    private Rutina rutina;

    @Override
    public void reset() {
        rutina = new Rutina();
    }

    @Override
    public void setNombre(String nombre) {
        rutina.setNombre(nombre);
    }

    @Override
    public void setNivel(String nivel) {
        rutina.setNivel(nivel);
    }

    @Override
    public void setEjercicios(List<String> ejercicios) {
        rutina.setEjercicios(ejercicios);
    }

    @Override
    public void setDuracion(int duracion) {
        rutina.setDuracion(duracion);
    }

    @Override
    public Rutina getResult() {
        return rutina;
    }
}
```

4. **Director:**

```java
import java.util.Arrays;

public class Director {
    private RutinaBuilder builder;

    public Director(RutinaBuilder builder) {
        this.builder = builder;
    }

    public void changeBuilder(RutinaBuilder builder) {
        this.builder = builder;
    }

    public void makeRutinaBasica() {
        builder.reset();
        builder.setNombre("Rutina Principiantes");
        builder.setNivel("Básico");
        builder.setEjercicios(Arrays.asList("Flexiones", "Sentadillas", "Abdominales"));
        builder.setDuracion(20);
    }

    public void makeRutinaAvanzada() {
        builder.reset();
        builder.setNombre("Rutina Avanzada");
        builder.setNivel("Avanzado");
        builder.setEjercicios(Arrays.asList("Burpees", "Dominadas", "Planchas", "Sprints"));
        builder.setDuracion(45);
    }
}
```

Ejemplo de uso:

```java
public class Main {
    public static void main(String[] args) {
        RutinaBuilder builder = new RutinaPersonalizadaBuilder();
        Director director = new Director(builder);

        director.makeRutinaBasica();
        Rutina rutinaBasica = builder.getResult();

        System.out.println(rutinaBasica);
    }
}
```

Salida esperada:

```
Rutina: Rutina Principiantes (Básico)
Ejercicios: [Flexiones, Sentadillas, Abdominales]
Duración: 20 min
```