UNIVERSIDAD NACIONAL DE COLOMBIA

Facultad de Ingeniería

Ingeniería de Software I

Grupo: 1

Profesor: Oscar Eduardo Álvarez Rodríguez

Integrantes:

Nicolás Rodríguez Tapia
Juan David Alarcón Sanabria
José Leonardo Pinilla Zamora
David Nicolás Urrego Botero

24 de octubre de 2025

Documentación de Patrones de Diseño en ClassMate AI

1. Patrón Singleton para la Base de Datos

1.1. ¿Qué es el Patrón Singleton?

El patrón Singleton es un patrón de diseño creacional que garantiza que una clase tenga una única instancia en toda la aplicación y proporciona un punto de acceso global a ella. Este patrón resulta especialmente útil para manejar recursos compartidos y costosos de crear, como las conexiones a bases de datos, donde múltiples instancias podrían generar ineficiencias o conflictos de concurrencia.

1.2. Implementación en ClassMate AI

En el contexto de Class AI, aunque no se implementa manualmente una clase Singleton para la base de datos, Spring Boot gestiona automáticamente este patrón a través de su contenedor de Inyección de Dependencias (IoC). Específicamente:

El DataSource es un bean Singleton: Bootura el componente DataSource (encargado de gestionar conexiones a PostgreSQL) como un bean único en el contexto de la aplicación. Esto significa que, independientemente de cuántas veces se solicite el DataSource en diferentes partes del código, siempre se devuelve la misma instancia.

Gestión de pool de conexiones: La instancia única de DataSource utiliza HikariCP (pool de conexiones por defecto en Spring Boot) para administrar eficientemente las conexiones a PostgreSQL, evitando la creación innecesaria de conexiones y optimizando el uso de recursos.

Configuración clave en application.properties:

spring.datasource.url=jdbc:postgresql://localhost:5432/classmateai

spring.datasource.username=postgres

spring.datasource.password=secret

spring.datasource.driver-class-name=org.postgresql.Driver

spring.datasource.hikari.connection-timeout=30000

spring.datasource.hikari.maximum-pool-size=10

¿Por qué es relevante?

Eficiencia en entornos concurrentes: La API REST de ClassMate AI recibe múltiples solicitudes simultáneas (por ejemplo, usuarios que acceden a transcripciones o chats RAG). El DataSource Singleton asegura que todas las operaciones de base de datos reutilicen las conexiones del pool, reduciendo la latencia y evitando cuellos de botella.

Seguridad: Al centralizar la gestión de conexiones, se evitan fugas de recursos (como conexiones no cerradas) y se garantiza un acceso controlado a la base de datos.

Integración nativa con Spring: No se requiere código adicional para implementar el patrón; este se gestiona automáticamente mediante anotaciones como @Component o @Service, asegurando que el DataSource se inicialice una sola vez al arrancar la aplicación.

2. Patrón Strategy para Exportación de Documentos

2.1. ¿Qué es el Patrón Strategy?

El patrón Strategy es un patrón de diseño conductual que define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que el algoritmo varíe independientemente de los clientes que lo utilizan, siguiendo el principio Open/Closed (abierto a la extensión, cerrado a la modificación).

2.2. Escenario de Uso en ClassMate AI

En ClassMate AI, la funcionalidad de exportación de documentos (PDF, DOCX) es un caso ideal para aplicar este patrón. Cuando un usuario solicita exportar resumen de clase o apuntes, la aplicación debe generar el documento en el formato especificado. Sin, cada formato requiere un algoritmo diferente para su generación:

PDF: Requiere bibliotecas como iText para crear estructuras PDF con texto y metadatos.

DOCX: Necesita Apache POI para manipular de Word.

Si se implementara esta lógica con condicionales (if-else), el código del servicio de exportación se volvería complejo y difícil de mantener. Por ejemplo:

```java
// ❌ Mala práctica: lógica condicional en el servicio
public byte[] export(String contenido, String formato) {
    if (formato.equals("PDF")) {
        // Lógica para PDF
    } else if (formato.equals("DOCX")) {
        // Lógica para DOCX
    }
    // ... más formatos = más condicionales
}
```

Con el patrón Strategy, cada formato se convierte en una estrategia independiente, permitiendo:

- Añadir nuevos formatos (p. ej., HTML, TXT) sin modificar el código existente.
- Desacoplar la lógica de exportación del servicio principal, mejorando la mantenibilidad y testabilidad.
- Seleccionar la estrategia adecuada en tiempo de ejecución basado en el formato solicitado.

2.3. Implementación Propuesta

Paso 1: Definir la interfaz común

```java
// ExportStrategy.java
public interface ExportStrategy {
    byte[] export(String contenido);
}
```

Paso 2: Implementar estrategias concretas

Para PDF (usando iText):

```java
// PDFExportStrategy.java
import com.lowagie.text.Document;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

public class PDFExportStrategy implements ExportStrategy {
    @Override
    public byte[] export(String contenido) {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        try (Document document = new Document()) {
            PdfWriter.getInstance(document, outputStream);
            document.open();
            document.add(new Paragraph(contenido));
            document.close();
        } catch (IOException e) {
            throw new RuntimeException("Error generando PDF", e);
        }
        return outputStream.toByteArray();
    }
}
```

Para DOCX (usando Apache POI):

```java
// DOCXExportStrategy.java
import org.apache.poi.xwpf.usermodel.XWPFDocument;
import org.apache.poi.xwpf.usermodel.XWPFParagraph;
import org.apache.poi.xwpf.usermodel.XWPFRun;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

public class DOCXExportStrategy implements ExportStrategy {
    @Override
    public byte[] export(String contenido) {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        try (XWPFDocument document = new XWPFDocument()) {
            XWPFParagraph paragraph = document.createParagraph();
            XWPFRun run = paragraph.createRun();
            run.setText(contenido);
            document.write(outputStream);
        } catch (IOException e) {
            throw new RuntimeException("Error generando DOCX", e);
        }
        return outputStream.toByteArray();
    }
}
```

Paso 3: Servicio que utiliza las estrategias

```java
// ExportService.java
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

@Service
public class ExportService {
    private final Map<String, ExportStrategy> estrategias;

    public ExportService(List<ExportStrategy> estrategias) {
        // Mapear estrategias por su nombre simplificado (p. ej., "PDF" → PDFExportStrategy)
        this.estrategias = estrategias.stream()
            .collect(Collectors.toMap(
                estrategia -> estrategia.getClass().getSimpleName().replace("ExportStrategy", ""),
                estrategia -> estrategia
            ));
    }

    public byte[] exportar(String contenido, String formato) {
        ExportStrategy estrategia = estrategias.get(formato.toUpperCase());
        if (estrategia == null) {
            throw new IllegalArgumentException("Formato no soportado: " + formato);
        }
        return estrategia.export(contenido);
    }
}
```

Paso 4: Uso en el Controlador REST

```java
// ExportController.java
import org.springframework.web.bind.annotation.*;
import java.io.IOException;

@RestController
@RequestMapping("/export")
public class ExportController {
    private final ExportService exportService;

    public ExportController(ExportService exportService) {
        this.exportService = exportService;
    }

    @PostMapping("/{formato}")
    public ResponseEntity<byte[]> exportar(@PathVariable String formato, @RequestBody String contenido) {
        byte[] documento = exportService.exportar(contenido, formato);
        return ResponseEntity.ok()
            .contentType(MediaType.APPLICATION_OCTET_STREAM)
            .body(documento);
    }
}
```

2.4. Beneficios en ClassMate AI

Escalabilidad: Si se añade soporte para HTML o Markdown en el futuro, solo se crea una nueva clase HTMLExportStrategy y Spring Boot la inyecta automáticamente en ExportService.

Mantenibilidad: Cada estrategia es independiente y puede ser probada aisladamente (p. ej., pruebas unitarias para PDFExportStrategy).

Claridad: El servicio ExportService no contiene lógica específica de formatos, solo delega la responsabilidad a las estrategias.

Flexibilidad: El formato de exportación puede seleccionarse dinámicamente (p. ej., desde parámetros de usuario o configuración).

Conclusión

Singleton: En ClassMate AI, el patrón Singleton se aplica de forma nativa a través de Spring Boot para la gestión de conexiones a PostgreSQL, garantizando eficiencia y seguridad en entornos concurrentes.

Strategy: Este patrón resulta ideal para la funcionalidad de exportación, ya que permite un diseño flexible y mantenible que admite múltiples formatos sin complicar la lógica central.

Ambos patrones refuerzan los principios del diseño moderno: separación de responsabilidades, reutilización de código y escalabilidad, alineándose plenamente con la arquitectura por capas y el propósito educativo de ClassMate AI.