Trabajo Fin de Grado
en Ingeniería de las Tecnologías de Telecomunicación

Planificación de actividades asistenciales en un servicio hospitalario con OptaPlanner

Autor: José Peralta Artero

Tutor: Dra. Isabel Román Martínez

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2026

Trabajo Fin de Grado
en Ingeniería de las Tecnologías de Telecomunicación

Planificación de actividades asistenciales en un servicio hospitalario con OptaPlanner

Autor:
José Peralta Artero

Tutor:
Dra. Isabel Román Martínez
Profesora colaboradora

Dpto. de Ingeniería telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2026

Trabajo Fin de Grado: Planificación de actividades asistenciales en un servicio hospitalario con OptaPlanner

Autor: José Peralta Artero

Tutor: Dra. Isabel Román Martínez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2026

El Secretario del Tribunal

Agradecimientos

A mis padres, por la infinita paciencia que han tenido durante todo este proceso interminable. Os quiero con mi alma.

A mi hermana, por ese apoyo incondicional que siempre me ha brindado y me ha animado a seguir hacia delante sin mirar atrás, por ejercer de hermana y de mejor amiga a la vez y por enseñarme que todo lo bueno puede ser aún mejor.

A Blanca, Tinoco, Manolo, Nico y Carmen, por esas tardes de estudio y de compartir apuntes y risas acompañados de un café de máquina que sabía a gloria.

A mis amigos, por darme energía para ir a clase sonriendo cada día.

A mi pareja, porque a pesar de todo, sin ti todo esto no hubiera sido posible. Por hacer de mí una mejor persona y darme alas cuando más lo he necesitado.

Resumen

El objetivo principal de este proyecto es evolucionar una solución existente para la gestión de actividades asistenciales en el ámbito sanitario, incorporando mecanismos más eficientes y mantenibles para la generación automática de la planificación mensual. La base de este proyecto se apoya en trabajos previos, como el TFG de Miguel Ángel González-Alorda Cantero, titulado *Servicio para la gestión de actividades complementarias* [1], su posterior ampliación por Carmen Cohen Calvo en *Gestión de calendarios para un servicio hospitalario* [2] y, por último, la refactorización de José Carlos Rodríguez Morón [4] en *BPM aplicado a procesos de gestión de actividades sanitarias.*

En esta nueva iteración, se ha sustituido el planificador original —desarrollado en Python— por una solución basada en OptaPlanner, una herramienta de planificación optimizada e integrada en el ecosistema de tecnologías KIE. Esta elección permite una mayor escalabilidad, mantenibilidad y capacidad de adaptación a nuevas restricciones o escenarios clínicos.

La solución continúa desarrollándose sobre los frameworks jBPM y Spring usando las librerías de ambas tecnologías.

Abstract

The main objective of this project is to evolve an existing solution for the management of healthcare activities by incorporating more efficient and maintainable mechanisms for the automatic generation of schedules. This project builds upon previous works, including the Bachelor's Thesis by Miguel Ángel González-Alorda Cantero titled *Service for the Management of Complementary Activities* [1], its subsequent extension by Carmen Cohen Calvo in *Calendar Management for a Hospital Service* [2], and finally, the refactoring carried out by José Carlos Rodríguez Morón in *BPM Applied to Healthcare Activity Management Processes* [4].

In this new iteration, the original scheduler—developed in Python—has been replaced by a solution based on OptaPlanner, a planning tool optimized and fully integrated within the Java technology ecosystem. This change enables greater scalability, maintainability, and adaptability to new constraints or clinical scenarios.

The solution continues to be developed using the jBPM and Spring frameworks, leveraging the libraries and features provided by both technologies.

Índice

Índice de ilustraciones

Índice de tablas

1 INTRODUCCIÓN

La motivación de este trabajo de fin de grado es evolucionar un servicio preexistente destinado a la gestión de actividades del personal para un servicio hospitalario. Esta solución ha sido desarrollada y ampliada a lo largo de varios trabajos previos, proporcionando funcionalidades como la asignación automática de turnos, la notificación de planificaciones a los facultativos y la posibilidad de intercambiar actividades mediante un bot de Telegram.

1.1 Motivación

Se dispone de un conjunto de soluciones con las siguientes capacidades elementales:

- Automatizar la planificación de las actividades de los facultativos de un servicio clínico, considerando tanto las necesidades del servicio como las preferencias y particularidades de cada facultativo.
- Automatizar la notificación de las planificaciones mensuales, así como de los cambios que ocurran en la misma, a todos los facultativos implicados.
- Facilitar la gestión e intercambio de las actividades a los facultativos.

El servicio ha sido realizado en distintos TFG. Miguel Ángel González-Alorda Cantero [1] desarrolló el proyecto (*Servicio para la gestión de actividades asistenciales complementarias*), para programar y gestionar las actividades de los doctores.

Carmen Cohen Calvo [2], en su TFG (*Gestión de calendarios para un servicio hospitalario*), lo extendió para notificar las actividades asistenciales a los doctores, Luís Marín Peña [3] desarrolló el TFG (*Diseño y desarrollo de un bot Telegram para la gestión de actividades asistenciales*) un complemento que facilita el cambio de los turnos entre doctores y que se ofrece como un bot de Telegram y, por último, José Carlos Rodríguez Morón [4] refactorizó la última versión para que fuera conforme al paradigma de gestión de procesos empresariales BPM en su TFG (*BPM aplicado a procesos de gestión de actividades sanitarias*).

Este TFG plantea como objetivo reemplazar el planificador actual, implementado en Python, por un sistema más robusto y mantenible basado en OptaPlanner, una herramienta de planificación automática desarrollada en Java y optimizada para integrarse con arquitecturas empresariales modernas. El uso de OptaPlanner permitirá abordar el problema de generación de calendarios asistenciales de forma más eficiente y escalable, gracias a sus capacidades avanzadas de optimización y resolución de problemas complejos de asignación de recursos.

Esta elección no es solo técnica, sino estratégica, y da continuidad directa al trabajo de José Carlos Rodríguez Morón [3], quien refactorizó la solución al paradigma BPM. OptaPlanner no es una herramienta aislada, sino un componente central del ecosistema KIE.

Nuestro ecosistema incluye jBPM (el motor de procesos que ya se utiliza en el proyecto para la orquestación), Drools (para la gestión de reglas de negocio) y su evolución nativa de la nube, Kogito. Por lo tanto, reemplazar el módulo de Python por OptaPlanner es el paso lógico para consolidar la arquitectura BPM. Se migra a una solución donde jBPM gestiona el "qué" (el proceso) y OptaPlanner decide el "quién" y "cuándo" (la optimización) de forma coherente.

Este cambio no solo responde a una necesidad tecnológica, sino también a una mejora en la calidad del servicio, al permitir una planificación más flexible, optimizada y adaptativa. Además, OptaPlanner ofrece mejores posibilidades de integración con el ecosistema Java/Spring y jBPM(KIE) del sistema ya existente, lo que facilita su mantenimiento y evolución futura.

A nivel arquitectónico, la solución se centra en el módulo de planificación (anteriormente en Python), encargado de calcular las asignaciones mensuales.

1.2 Alcance del Proyecto

Aunque no se realiza una refactorización completa de la arquitectura del sistema, este cambio implica una modificación relevante en la lógica de generación de planificación de tareas, manteniendo el resto de los componentes del sistema sin alteraciones significativas. Con el fin de acotar el trabajo y facilitar su integración en la solución existente, se han tomado las siguientes decisiones:

- Utilizar OptaPlanner como motor de planificación, dada su capacidad para integrarse de forma nativa en aplicaciones desarrolladas con Spring Boot y KIE.
- Enfocar el alcance del proyecto exclusivamente en el proceso de generación de planificación mensual, quedando fuera de este trabajo otros procesos administrativos o funcionalidades complementarias ya contempladas en desarrollos anteriores.

1.3 Descripción de los siguientes capítulos

El resto del documento estará dividido en los siguientes capítulos:

- 2. Estado de la tecnología: Explicación de las tecnologías usadas en este proyecto.
- 3. Requisitos: Una descripción formal de los requisitos del sistema.
- 4. Trabajo realizado: Se describirá la solución desarrollada.
- 5. Conclusiones y trabajo futuro: Se resumirán los capítulos anteriores y se dará un vistazo de los futuros pasos que se tendrán que dar para continuar con el proyecto.

2 ESTADO DE LA TECNOLOGÍA

En este apartado detallaremos las tecnologías utilizadas para el diseño, desarrollo e implementación de este proyecto.

2.1 Java

Es un lenguaje de programación de alto nivel, orientado a objetos, que se utiliza ampliamente en el desarrollo de software gracias a su capacidad de ejecutarse en cualquier plataforma. [5]

En este proyecto usaremos la versión de Java 8 [6].

2.2 Spring Boot

Spring Boot[7] es un framework de código abierto orientado al desarrollo rápido y simplificado de aplicaciones Java. Su principal objetivo es reducir la complejidad en la configuración y despliegue de aplicaciones, proporcionando facilidades al desarrollador como la inyección de dependencias.

Una de sus principales fortalezas es su capacidad para integrarse fácilmente con otras tecnologías. En este proyecto, esa flexibilidad ha permitido combinar Spring Boot con OptaPlanner, el motor de resolución de restricciones seleccionado.

Esta sinergia permite una clara separación de responsabilidades: mientras que Spring Boot se encarga de la configuración de la aplicación, la gestión de dependencias y la exposición de servicios REST (para iniciar la planificación y consultar los resultados), OptaPlanner se enfoca exclusivamente en la lógica de optimización y en el cómputo de la solución óptima.

2.3 Maven

Maven[8] es una herramienta de automatización ampliamente utilizada en el ecosistema Java para la gestión y construcción de proyectos. Su propósito es facilitar el trabajo de desarrollo mediante una estructura coherente y reproducible que permite compilar, empaquetar, testear y desplegar aplicaciones de forma controlada y eficiente.

Entre sus principales objetivos se encuentran:

- Simplificar el proceso de construcción y empaquetado de proyectos.
- Estandarizar la organización del ciclo de vida del software.
- Mejorar la gestión de dependencias y la configuración del entorno.
- Promover buenas prácticas y estructuras consistentes en el desarrollo.

En el contexto del origen de este proyecto, Maven desempeña un papel central, ya que todos los componentes de la solución, desde el planificador hasta los modelos de datos o procesos, están organizados como artefactos Maven independientes. Esta modularidad permite un desarrollo desacoplado, una mejor reutilización de los módulos y una integración más fluida entre ellos.

La configuración de cada módulo, así como la declaración de librerías externas, se gestiona mediante el archivo pom.xml, en el que se definen las dependencias necesarias, los plugins requeridos y otros aspectos clave del proceso de construcción.

Gracias a Maven, es posible mantener una arquitectura limpia, escalable y fácilmente mantenible, alineada con los principios de una aplicación empresarial moderna.

2.3.1 Dependencias actualizadas y añadidas

El proyecto utiliza Apache Maven como herramienta de gestión de dependencias. Partiendo del trabajo anterior, que implementaba un entorno funcional con `Spring Boot 2.6.15` y `KIE Server 7.74.1.Final`, se han añadido nuevas dependencias para integrar el motor de optimización y mejorar la infraestructura de pruebas.

Para integrar OptaPlanner se han tenido que añadir las siguientes dependencias:

- **optaplanner-core(7.48.0.Final):** contiene los algoritmos necesarios para resolver el problema de asignación de turnos. [9]
- **optaplanner-persistence-jpa(7.48.0.Final):** facilita la integración directa entre las entidades de planificación y la base de datos, permitiendo que OptaPlanner persista las soluciones. [10]

Para solucionar problemas de incompatibilidad de versiones entre el planificador y KIE, se han realizado los siguientes cambios:

- Se ha modificado la dependencia `kie-server-spring-boot-starter` para excluir los módulos internos de Drools que entraban en conflicto con la versión del planificador. Se ha excluido `drools-canonical-model`, `drools-core-dynamic`, `drools-alphanetwork-compiler` y `drools-mvel`.
- Se han importado explícitamente las versiones necesarias de `drools-core`, `drools-compiler` y `drools-model-compiler`. Todo a la versión 7.74.1.Final.
- Se han añadido exclusiones para `xerces` y `xml-apis`. Esto fue necesario para evitar la excepción `accessExternalDTD`, un error común en entornos Java modernos cuando múltiples librerías intentan controlar el procesado de archivos XML.

Por último, se ha añadido la gestión de dependencias (dependencyManagement) para `org.testcontainers`, junto con las librerías `junit-jupiter` y `postgresql`. Esto permite levantar instancias reales de PostgreSQL en contenedores Docker efímeros durante la fase de prueba de Maven. Gracias a esto, las pruebas de integración del algoritmo de planificación se ejecutan sobre un entorno idéntico al de producción, validando correctamente las transacciones JPA y las consultas nativas. [11]

2.4 KIE

KIE (Knowledge Is Everything) [12] es una iniciativa de software de código abierto que agrupa y gobierna un conjunto de proyectos maduros enfocados en la Automatización de Negocios (Business Automation). Históricamente respaldado por JBoss y Red Hat, y actualmente en proceso de incubación bajo la Apache Software Foundation (Apache KIE), este ecosistema proporciona una plataforma unificada para modelar, automatizar y orquestar decisiones empresariales complejas.

La arquitectura de KIE se distingue por integrar de forma nativa paradigmas que tradicionalmente operaban en silos: la gestión de procesos (BPM), la lógica basada en reglas (BRE) y la optimización matemática.

2.4.1 jBPM

jBPM, acrónimo de Java Business Process Management, es una Plataforma de código abierto orientada a la automatización de procesos y decisiones de negocio. Permite modelar, ejecutar y monitorizar flujos de trabajo que reflejan la lógica de una organización. [13]

Nosotros partiremos de la solución de Jose Carlos, remplazando el proceso encargado de la planificación anterior por un nuevo planificador basado en Optaplanner.

2.4.2 Kjar

Un archivo kjar en jBPM es un paquete JAR que encapsula todos los activos de negocio para modelar procesos: modelos en BPMN, reglas definidas en DRL (Drools Rule Language), formularios de usuario, archivos de configuración y otros activos relevantes. [14]

Esta agrupación en un único artefacto facilita tanto la distribución como el mantenimiento de los procesos empresariales.

2.5 Motor de Reglas de Negocio

BRE (Business Rule Engine) o Motor de Reglas de Negocio [15], es un componente de software que permite separar la lógica de decisión empresarial del código fuente de la aplicación principal. Su función es ejecutar una o varias reglas de negocio en un entorno de producción, actuando como un sistema experto que infiere conclusiones o ejecuta acciones basándose en un conjunto de hechos conocidos.

A diferencia de la programación tradicional, donde la lógica está incrustada en el flujo de control, un BRE opera bajo un paradigma declarativo. Nosotros definimos que se debe cumplir y el motor decide cómo y cuándo evaluar esas reglas.

2.5.1 Alternativas a un BRE

Existen varias alternativas tradicionales que se evaluaron antes de seleccionar un BRE:

- **Programación imperativa o Hardcoding:** se basa en implementar la lógica mediante estructuras de control condicionales directamente en el lenguaje de programación con el que se trabaje. Una desventaja clara, sería que a medida que el problema se vuelve más complejo, el código se vuelve inmanejable y difícil de probar. Además, requeriría recompilar y desplegar la aplicación ante cualquier cambio mínimo de una regla de negocio.
- **Procedimientos almacenados en base de datos:** se basa en delegar la lógica en la base de datos. Esto genera un fuerte acoplamiento con el proveedor de base de datos, además de dificultar el control de versiones y la baja escalabilidad.
- **Motores de Scripting (Scripting Engines):** se basa en la ejecución de pequeños programas escritos en un lenguaje de programación, como podría ser Lua, Python o JavaScript, para automatizar tareas y extender funcionalidades. Sin embargo, no ofrecen las capacidades de encadenamiento de reglas y optimización de memoria que ofrece un BRE.

2.5.2 Justificación de la elección de un BRE

Se ha optado por una solución que tiene una arquitectura basada en BRE, específicamente Drools dentro de KIE, por las siguientes razones:

- **Desacoplamiento y agilidad:** permite modificar reglas de negocio sin necesidad de modificar el ciclo de vida del software principal.
- **Centralización de la lógica:** es un componente basado en la lógica, de esta manera evita que las reglas de negocio se dispersen por las capas de aplicación.
- **Eficiencia Algorítmica:** permite evaluar miles de reglas sobre un gran volumen de datos, de manera que se vuelve mucho más eficiente en comparación con una iteración imperativa.

2.6 OptaPlanner

OptaPlanner [16] es un motor de optimización de planificación de código abierto, diseñado para resolver problemas complejos de asignación de recursos en entornos empresariales. Forma parte del ecosistema KIE y está desarrollado en Java, lo que permite una integración directa con aplicaciones basadas en SpringBoot, Quarkus o plataformas Java estándar.

Esta herramienta está orientada a resolver desafíos típicos en los que se busca encontrar una solución óptima (o suficientemente buena) bajo ciertas restricciones en un tiempo razonable. Ejemplos comunes incluyen planificación de turnos, asignación de tareas, gestión de rutas de vehículos o programación de horarios. En este proyecto, se utiliza para generar la planificación mensual de actividades asistenciales que respetan tanto las restricciones de la organización como las preferencias de los facultativos.

Para lograr esto, OptaPlanner se basa en tres pilares principales: los componentes, la función de puntuación y el Solver.

2.6.1 Los componentes

El primer paso sería modelar el problema usando POJOs (Plain Old Java Objects) y etiquetas. Estas etiquetas le muestran al Solver, que es el calculador de soluciones, qué partes del problema puede modificar y cuales no:

- ***@PlanningSolution***: es la clase que encapsula la solución al problema. Contiene la lista de entidades y de *@ProblemFacts*. También contiene la puntuación de todas las soluciones que OptaPlanner va probando, así como la final.
- ***@ProblemFact***: son los datos de entrada que no cambian durante la planificación.
- ***@PlanningEntity***: son las clases que se modifican para encontrar una solución, son las entidades sobre las que decidir.
- ***@PlanningVariable***: son los valores de la entidad de planificación que el motor debe ajustar para obtener la mejor solución posible.
- ***@ValueRangeProvider***: define el espacio de búsqueda o conjunto de valores candidatos que pueden ser asignados a una variable de planificación.
- ***@PlanningId***: proporciona un identificador único y estable a un objeto. Es fundamental para que el solver pueda rastrear las entidades a lo largo de las iteraciones o compararlas.
- ***@PlanningScore***: identifica el atributo que almacenará la puntuación (HardSoftScore) de la solución en cada iteración.
- ***@PlanningPin:*** permite marcar una entidad concreta como fija. El algoritmo ignorará esta entidad y no modificará su asignación si así se desea.

2.6.2 La función de puntuación

Una vez modelado el problema, definimos las reglas de negocio o restricciones. Optaplanner utiliza un sistema de puntuación declarativo. La mejor solución será la que tenga la puntuación más alta. Hay dos tipos de reglas:

- *Hard Constraints* o restricciones duras. Estas reglas no deben romperse en ninguna circunstancia, una sola infracción puede hacer que no se pueda aceptar una solución. Por ejemplo, que un médico tenga dos turnos solapados.
- *Soft Constraints* o restricciones blandas. Son reglas que se quieren cumplir, pero se pueden romper para satisfacer una regla dura. Representan la calidad de la solución.

La puntuación se puede revisar en la salida de consola y tiene el siguiente formato:

(0 Hard/-10 Soft)

La puntuación obtenida dependerá de las restricciones que se rompan. En función de cada regla que no se cumpla, el sistema aplica una penalización. Esta penalización es el peso que se ha establecido a cada restricción. Por ejemplo, si tenemos una restricción dura que tiene un peso de 10 y la incumplimos, el sistema obtendrá el valor (-10 Hard/0 Soft). Esto puede variar según el diseño de la aplicación de las reglas, hay algunas reglas que pueden aplicar una penalización exponencial, de manera que, mientras más veces se incumpla, mayor será la penalización.

Una solución se considera viable si la parte Hard se encuentra próxima a 0. La parte Soft solo aclara la calidad de la solución.

2.6.3 El Solver

Es el motor que ejecuta los algoritmos de optimización. Su trabajo se ejecuta en dos fases:

Primero intenta generar rápidamente una primera solución factible, que no rompa reglas duras, aunque sea de baja calidad. Luego, toma la solución inicial y la mejora iterativamente. El Solver utiliza algoritmos avanzados de optimización metaheurística, como pueden ser *Simulated Annealing* [17] o búsqueda Tabú [18]. Esto lo hace para evitar atascarse en óptimos locales y extender el espacio de búsqueda.

El uso de OptaPlanner en este proyecto supone una mejora significativa respecto al planificador anterior basado en Python, ya que aporta mayor flexibilidad, escalabilidad y mantenibilidad dentro del ecosistema tecnológico KIE.

2.6.4 Relación con BRE

En este contexto, BRE se utiliza como calculadora de puntuación (Score Calculation). Optaplanner sigue un ciclo iterativo de generación y prueba para la optimización. El solver propone una posible solución y necesita saber si la solución propuesta es lo suficientemente buena. Aquí es donde entra en juego BRE, este evalúa automáticamente estos objetos contra las restricciones definidas y devuelve un valor numérico (Score) que indica la calidad de la solución

En la arquitectura diseñada, el BRE asume el rol fundamental de calculadora de puntuación (Score Calculation). Mientras el solver de OptaPlanner sigue un ciclo iterativo de generación de soluciones candidatas, este delega en el motor de reglas la tarea de evaluar su calidad frente a las restricciones definidas. Se ha seleccionado la integración con un BRE basado en Drools, frente a implementaciones imperativas tradicionales en Java, debido a su capacidad nativa para realizar un cálculo de puntuación incremental. Esta característica permite que, ante cada modificación local propuesta por el algoritmo, el sistema actualice únicamente las reglas afectadas por dicho cambio en lugar de reevaluar la solución completa desde cero. Esta eficiencia en la gestión del estado resulta determinante, ya que posibilita la evaluación de millones de movimientos por segundo y garantiza la obtención de planificaciones de alta calidad en tiempos de ejecución viables.

2.7 PostgreSQL

PostgreSQL [19] es un sistema de gestión de bases de datos relacional de código abierto, ampliamente reconocido por su robustez, rendimiento y capacidad de ampliación. Basado en el estándar SQL, PostgreSQL incorpora características avanzadas que lo convierten en una solución ideal para aplicaciones que requieren integridad, seguridad y eficiencia en la manipulación de grandes volúmenes de datos.

Entre las características más destacadas de PostgreSQL se encuentran:

- Soporte para una amplia variedad de tipos de datos (estructurados, JSON, geoespaciales, arrays, etc.).
- Consistencia e integridad de los datos, gracias al uso de transacciones ACID.
- Alto rendimiento en operaciones complejas y consultas concurrentes.
- Elevada fiabilidad, tanto en sistemas locales como distribuidos.
- Mecanismos avanzados de seguridad, como roles, autenticación y cifrado.
- Gran capacidad de extensión mediante módulos, funciones y tipos personalizados.
- Compatibilidad con búsqueda de texto avanzada, incluyendo soporte multilingüe y operadores especializados.

En este proyecto, PostgreSQL se utiliza como base de datos principal para almacenar la información relativa a usuarios, turnos, preferencias y resultados de planificación, siendo una pieza fundamental para el funcionamiento del sistema.

3 REQUISITOS

En este capítulo se describen los requisitos del proyecto.

3.1 Actores

Estos son los actores participantes en los casos de uso de este proyecto.

A-01	GenerateScheduleWorkItemHandler
Descripción	Este componente es el encargado de ejecutar la lógica de optimización de OptaPlanner.

Tabla 1: A-01 – GenerateScheduleWorkItemHandler

3.2 Caso de uso

CU-01	**Generar la planificación con OptaPlanner**
Requisitos	• Los festivos han sido establecidos con la ejecución de la tarea "Establecer festivos". • Existen doctores activos en la base de datos con sus preferencias establecidas. • Se dispone de conexión a la base de datos para leer las entidades JPA necesarias.
Descripción	El sistema calcula automáticamente la asignación de turnos y guardias para los doctores activos, maximizando el cumplimiento de las preferencias (soft constraints) y asegurando el cumplimiento de las normas (hard constraints) mediante el motor OptaPlanner.
Entradas y salidas	• Como entrada está `Id_calendario_festivos` que es el identificador del calendario del mes que contiene los días festivos ya establecidos. • Como salida está `Id_planificacion_provisional` que es el identificador del calendario del mes con la planificación generada y persisitida, lista para ser validada.

Tabla 2: CU-01 – Generar la planificación

3.3 Requisitos funcionales del Sistema

3.3.1 Requisitos de información

RI-01	Doctor			
Descripción	Representa al sanitario al que se le asignan los turnos. Contiene información personal y administrativa necesaria para la validación.			
Atributos	**Nombre**	**Tipo**	**Cardinalidad**	**Descripción**
	id	Long	1..1	Identificador del doctor.
	telegramId	String	0..1	Identificador del telegram del doctor, puede no tener ningún identificador de telegram asignado.
	firstName	String	1..1	Nombre del doctor.
	lastName	String	1..1	Apellidos del doctor.
	email	String	1..1	Correo electrónico del doctor.
	status	DoctorStatus	1..1	Disponibilidad del doctor (disponible o eliminado).
	absence	Absence	0..1	Ausencias del doctor en el mes, puede no tener ninguna ausencia registrada.
	startDate	LocalDate	1..1	Fecha de creación del doctor.
	roles	Set<Rol>	0..*	Roles del doctor, cuando se crea no tiene ningún rol asignado.
	shiftConfiguration	ShiftConfiguration	0..1	Configuración de los turnos de un doctor. Cuando se crea, puede no tener una configuración asignada.

Tabla 3: RI-01 – Doctor

RI-02	Calendar			
Descripción	Para realizar la planificación de un mes se requiere conocer los días festivos del mismo.			
Atributos	**Nombre**	**Tipo**	**Cardinalidad**	**Descripción**
	month	Integer	0..1	Mes del calendario, puede estar vacío.
	year	Integer	0..1	Año del calendario, puede estar vacío.
	dayConfigurations	SortedSet<DayConfiguration>	0..*	Lista de configuraciones de los días del mes del calendario.

Tabla 4: RI-02 – Calendar

RI-03	Schedule			
Descripción	Es la clase que envuelve todo el problema y el resultado. Se utilizará para enviar los datos al solver y para recibir la mejor solución.			
Dependencias	• RC-03: Tipos de turnos. • RN-01: Valor de la solución obtenida. • RN-14: Configuración específica de Day y Prevalencia de restricciones.			
Atributos	**Nombre**	**Tipo**	**Cardinalidad**	**Descripción**
	month	Integer	1..1	Mes de la planificación.
	year	Integer	1..1	Año de la planificación.
	calendar	Calendar	1..1	Referencia a la clase `Calendar`.
	status	ScheduleStatus	1..1	Estado de la planificación.
	days	SortedSet<ScheduleDay>	0..*	Lista de los días del calendario, puede estar vacío.
	shiftAssignments	List<ShiftAssignment>	0..*	Lista de las asignaciones de los turnos de la planificación, puede estar vacío antes de generar la planificación. Contiene la etiqueta `@PlanningEntityCollectionProperty`.

	doctorList	List<Doctor>	0..*	Lista de doctores, puede estar vacío antes de generar la planificación. Contiene la etiqueta `@ProblemFactCollectionProperty`.
	shiftList	List<Shift>	0..*	Lista de turnos de la planificación, puede estar vacío antes de generar la planificación. Contiene la etiqueta `@ProblemFactCollectionProperty`.
	dayConfiguration List	List <DayConfiguration>	0..*	Lista de configuración de los días de la planificación, puede estar vacío antes de generar la planificación. Contiene la etiqueta `@ProblemFactCollectionProperty`.
	score	HardSoftScore	0..1	Puntuación de la solución calculada por OptaPlanner, será nulo antes de resolver por primera vez. Contiene la etiqueta `@PlanningScore`.

Tabla 5: RI-03 – Schedule

RI-04	Shift			
Descripción	Representará el turno que se tiene que cubrir. Aparecerá como una lista de `@ProblemFacts` en `Schedule`.			
Dependencias	• RN-14: Configuración específica de Day y Prevalencia de restricciones.			
Atributos	**Nombre**	**Tipo**	**Cardinalidad**	**Descripción**
	id	Long	1..1	Identificador del turno.
	dayConfiguration	DayConfiguration	1..1	Configuración del día específico al que pertenece el turno.
	shiftType	String	1..1	Tipo de turno.
	requiresSkill	boolean	1..1	Indica si el turno requiere de un doctor con habilidades especiales.

	isConsultation	boolean	1..1	Indica si el turno es una consulta.

Tabla 6: RI-04 – Shift

RI-05	ShiftAssignment			
Descripción	OptaPlanner modificará esta entidad. Contiene la variable de planificación que se cambia iterativamente para encontrar la solución óptima. Vinculará `Shift` con un `Doctor`.			
Dependencias	• RC-03: Tipos de turnos.			
Atributos	**Nombre**	**Tipo**	**Cardinalidad**	**Descripción**
	id	Long	1..1	Identificador de la asignación de turnos.
	shift	Shift	1..1	Turno que se asigna a un doctor.
	doctor	Doctor	0..1	Médico asignado al turno que se intenta cubrir. Puede ser nulo si no hay médicos suficientes.
	pinned	boolean	1..1	Indica si la asignación es inamovible.
	schedule	Schedule	1..1	Referencia a la clase `Schedule` al que pertenece la asignación.

Tabla 7: RI-05 – ShiftAssignment

RI-06	ShiftConfiguration			
Descripción	Se encargará de definir el perfil de trabajo de cada médico. Contiene las restricciones que OptaPlanner debe satisfacer.			
Dependencias	• RC-03: Tipos de turnos. • RN-15: Catálogo de tipos de turnos.			
Atributos	**Nombre**	**Tipo**	**Cardinalidad**	**Descripción**
	doctorId	Long	1..1	Identificador del doctor.
	doctor	Doctor	1..1	Referencia al doctor al que se refiere esta configuración.
	maxShifts	Integer	1..1	Número máximo de turnos que realiza el médico al mes.
	minShifts	Integer	1..1	Número mínimo de turnos que realiza el médico al mes.
	numConsultation	Integer	1..1	Numero de consultas que realiza el médico al mes.

	doesCycleShifts	Boolean	1..1	Indica si el médico realiza guardias.
	hasShiftOnlyWhen CycleShifts	Boolean	1..1	Indica si el médico realiza turnos de tarde solo cuando tiene guardias.
	unwantedShifts	Set <AllowedShift>	0..*	Lista de turnos que el médico quiere evitar, puede estar vacío.
	unavailableShifts	Set <AllowedShift>	0..*	Lista de turnos que el médico no está disponible, puede estar vacío.
	wantedShifts	Set <AllowedShift>	0..*	Lista de turnos que el médico quiere realizar, puede estar vacío.
	mandatoryShifts	Set <AllowedShift>	0..*	Lista de turnos que el médico tiene que realizar, puede estar vacío.
	wantedConsultations	Set <AllowedShift>	0..*	Lista de consultas que el médico quiere realizar, puede estar vacío.

Tabla 8: RI-06 – ShiftConfiguration

3.3.2 Requisitos de conducta

RC-01	Asignación de tareas
Descripción	El sistema debe asignar automáticamente las tareas a los usuarios.

Tabla 9: RC-01 – Asignación de tareas

RC-02	Generación de planificación
Descripción	Tras elegir los festivos, el sistema deberá generar la planificación del mes siguiente al actual automáticamente.

Tabla 10: RC-02 – Generación de planificación

RC-03	Tipos de turnos
Descripción	El sistema deberá contemplar tres tipos de turnos: • Turnos de tarde o *Shifts*. Turnos normales en el hospital que se realizan por la tarde. Deberán realizar un número de turnos comprendido entre el máximo y mínimo estipulado por contrato.

	• Guardias o *CycleShifts.* Guardias de día entero que realizan los médicos. Deberán realizar guardias si así se estipuló en el contrato. • Consultas o *Consultations.* Consultas que realizan los médicos. Deberán cumplir con el número de consultas estipuladas por contrato.

Tabla 10: RC-03 – Tipos de turnos

3.3.3 Reglas de negocio

Las reglas de negocio de nuestro sistema serán las restricciones que establecemos. Declaradas en `GuardianesConstraintConfiguration.java` y definidas en `guardianesScoreRules.drl`.

Están ordenadas de mayor a menor peso. Primero reglas duras y después blandas.

RN-01	Valor de la solución obtenida
Descripción	Representa el resultado numérico de la evaluación del planificador. Se compone de dos partes: • Hard Constraint: valor que debe ser igual a cero para considerar la solución válida. Un valor negativo indica la ruptura de reglas inviolables. • Soft Constraint: valor que representa la calidad de la solución. Representa el grado de satisfacción de preferencias deseables pero no obligatorias.

Tabla 11: RN-01 – Valor de la solución obtenida

RN-02	Incompatibilidad de consultas
Descripción	**Regla dura**. El doctor que tenga asignado para un día hacer consultas, tendrá dedicación completa ese día. Es decir, no podrá hacer ni tardes ni guardias ese mismo día.

Tabla 12: RN-02 – Incompatibilidad de consultas

RN-03	Realización de guardias
Descripción	**Regla dura**. Solo los médicos que tengan establecido que realizan guardias, podrán realizar guardias a lo largo del mes.

Tabla 13: RN-03 – Realización de guardias

RN-04	Respeto de vacaciones y ausencias
Descripción	**Regla dura**. Será imposible asignar cualquier tipo de turno a un doctor en una fecha que coincida con un periodo de ausencia registrado, ya sean vacaciones, bajas o permisos.

Tabla 14: RN-04 – Respeto de vacaciones y ausencias

RN-05	Evitar duplicados
Descripción	**Regla dura**. Un doctor no podrá tener asignado más de un turno del mismo tipo en el mismo día.

Tabla 15: RN-05 – Evitar duplicados

RN-06	Vinculación condicional de turnos
Descripción	**Regla dura**. Para doctores que hayan declarado que hacen tardes y guardias el mismo día, solo se puede asignar un turno de tarde si ese mismo día tienen asignada una guardia.

Tabla 16: RN-06 – Vinculación condicional de turnos

RN-07	Cobertura total de turnos
Descripción	**Regla dura**. Todos los turnos definidos en el calendario deben tener asignados un doctor. No puede haber días sin cubrir.

Tabla 17: RN-07 – Cobertura total de turnos

RN-08	Máximo de turnos
Descripción	**Regla dura**. El número total de turnos de tarde asignados a un doctor en el mes no puede superar el limite máximo estipulado para cada médico.

Tabla 18: RN-08 – Máximo de turnos

RN-09	Mínimo de turnos
Descripción	**Regla dura**. El número total de turnos de tarde asignados a un doctor en el mes no puede ser inferior al mínimo estipulado por cada médico.

Tabla 19: RN-09 – Mínimo de turnos

RN-10	Número de consultas
Descripción	**Regla dura**. Si un doctor realiza consultas, el número total de consultas realizadas en el mes debe coincidir con la cantidad estipulada para cada médico.

Tabla 20: RN-10 – Número de consultas

RN-11	Descanso entre guardias
Descripción	**Regla blanda**. Se debe intentar que existan al menos 3 días de diferencia entre dos guardias asignadas al mismo doctor para asegurar su descanso.

Tabla 21: RN-11 – Descanso entre guardias

RN-12	Equidad de reparto de guardias
Descripción	**Regla blanda**. El sistema debe penalizar exponencialmente las soluciones con doctores que acumulen muchas más guardias que el resto, forzando un reparto equitativo de la carga de trabajo entre todos los disponibles.

Tabla 22: RN-12 – Equidad de reparto de guardias

RN-13	Evitar tardes consecutivas
Descripción	**Regla blanda**. Se debe evitar, en la medida de lo posible, asignar turnos de tarde en días consecutivos para reducir la fatiga acumulada del personal sanitario.

Tabla 23: RN-13 – Evitar tardes consecutivas

RN-14	Configuración específica de Day y Prevalencia de restricciones
Descripción	El sistema debe manejar una configuración específica para cada día del calendario. Corresponde a la entidad `DayConfiguration`. Esta entidad debe encapsular las condiciones particulares de una fecha incluyendo: • Naturaleza del día: si es laborable o festivo. • Demanda: número de guardias y consultas requeridas para ese día específico. • Disponibilidad y preferencias de los doctores para los turnos específicos de ese día.

Tabla 24: RN-14 – Configuración específica de Day y Prevalencia de restricciones

RN-15	Catálogo de tipos de turnos
Descripción	Los turnos o franjas horarias asignables en el sistema deben pertenecer a un conjunto finito y predefinido en la entidad `AllowedShift`. Cada elemento se identifica por una etiqueta de texto `shift` que no puede estar vacía y debe ser única.

Tabla 25: RN-15 – Catálogo de tipos de turnos

4 SOLUCIÓN DESARROLLADA

En este capítulo se describirá la solución desarrollada. Todo el código se encuentra disponible en:
https://github.com/tfg-projects-dit-us/GuardianesBA.git

4.1 Pilares de OptaPlanner en el sistema

4.1.1 Componentes del Sistema

Para fusionar la solución existente con la tecnología de OptaPlanner, se ha adaptado el modelo de datos utilizando las anotaciones específicas del motor. A continuación, se describe la correspondencia de los componentes de OptaPlanner y de clases Java.

4.1.1.1 @PlanningSolution

Se ha utilizado la clase existente `Schedule`. Esta clase representa el conjunto de datos completo del calendario, contiene tanto los datos estáticos, como los doctores disponibles o los turnos que se tienen que cubrir, como la planificación que se está generando. Al añadir esta etiqueta OptaPlanner entiende que es aquí donde se tiene que fijar para obtener los datos de interés.

También se ha añadido la puntuación de calidad de la solución generada, anotada como `@PlanningScore`.

4.1.1.2 @ProblemFact

Son listas en `Schedule`. Aquí se incluye la información estática durante la ejecución. En esta solución, las clases `Doctor`, `DayConfiguration` o `Shift` se inyectan como colección de `@ProblemFact` al ser listas, con la etiqueta `@ProblemFactCollectionProperty`. Cada una de estas listas son hechos inamovibles, son propiedades que no van a cambiar durante el desarrollo de la solución. Siendo `doctorList` el catálogo de médicos disponibles, `shiftList` los tipos de turnos definidos y `dayConfigurationList` las configuraciones específicas de los días.

4.1.1.3 @PlanningEntity

Esta etiqueta se encuentra en la clase `ShiftAsignment`. Esta clase se ha creado para que represente el hueco a rellenar, es decir, para que se encargue de la asignación de médicos a los turnos disponibles. Se ha añadido la anotación `@PlanningId` sobre un campo "id" que se ha creado para que OptaPlanner pueda identificar unívocamente cada objeto asignable. Esta anotación sirve para que el motor no pierda referencia del turno que está intentando asignar.

Dentro de la clase Schedule pasa algo parecido a lo que nos pasaba con la etiqueta anterior. La clase `ShiftAssignment` se inyecta como colección de la entidad de planificación con la etiqueta `@PlanningEntityCollectionProperty` al ser un ArrayList.

4.1.1.4 @PlanningVariable

La variable de planificación es el campo "doctor" en la clase `ShiftAssignment`. Este campo, será modificado iterativamente por OptaPlanner para encontrar una solución. La anotación también incluye `valueRangeProviderRefs = {"doctorRange"}` que conecta la variable con la lista de doctores definidas en `Schedule`.

4.1.1.5 Otras etiquetas de interés

- ***@PlanningId:*** proporciona una indentidad única y estable a cada objeto de planificación. Sirve para que el solver pueda identificar la entidad a lo largo de sus iteraciones. La encontramos en la clase `ShiftAssignment` sobre el atributo `id`.
- ***@PlanningScore:*** identifica el atributo que guardará la puntuación de la solución generada en un momento dado. Este valor puede variar en cada iteración, dependiendo de las restricciones establecidas y del tiempo empleado en resolver el problema. La encontramos en la clase `Schedule` sobre el atributo `score`.
- ***@ValueRangeProvider:*** define el espacio de búsqueda que el algoritmo usa para asignar la variable de planificación. En nuestro caso, define la lista completa de doctores disponibles que actuarán como candidatos para cubrir los turnos de la clase `ShiftAssignment`. La encontramos en la clase `Schedule` sobre la colección `doctorList`.
- ***@PlanningPin***: marca una entidad como inamovible. Cuando el valor es `true`, OptaPlanner excluye dicha entidad del proceso de optimización, respetando el valor asignado previamente. Esto es esencial para funcionalidades de planificación continua (evitar modificar turnos pasados) y para permitir asignaciones manuales forzosas por parte del usuario gestor. La encontramos en la clase `ShiftAssignment` sobre el campo `pinned`.

4.1.2 Función de puntuación

Se ha implementado la clase `GuardianesConstraintConfiguration` anotada con `@ConstraintConfiguration`. Esta clase le aporta flexibilidad al sistema debido a que almacena los pesos asociados a las reglas de negocio. Gracias a esto, el sistema permite ajustar la severidad de las restricciones en tiempo de ejecución, inyectando estos valores directamente en el motor de reglas Drools mediante la anotación `@ConstraintWeight`.

A mayor sea el peso que le otorguemos a la restricción, más predominante será con respecto a las otras reglas. La función de puntuación se ha implementado utilizando el motor de relgas JBoss Drools [20] a través del archivo `guardianesScoreRules.drl` situado en la ruta `guardianes-service/src/main/resources/us/dit/service/model/entities/score/guardianesScoreRules.drl`.

A diferencia de la programación tradicional iterativa, que usa principalmente bucles anidados, aquí se definen patrones que el motor usará para establecer una puntuación. A continuación, se describen brevemente los fundamentos de este lenguaje para facilitar la compresión de la lógica implementada.

4.1.2.1 Fundamentos del lenguaje Drools

Un archivo DRL (*Drools Rule Language*) define un conjunto de reglas que el motor evalúa. La estructura básica de una regla en Drools se divide en dos partes fundamentales:

- **LHS (*Left Hand Side*) o "When":** define las condiciones que deben cumplirse. El motor busca objetos en memoria que coincidan con los criterios nombrados. También permite el uso de lógica avanzada como `not`, para la usencia de un hecho, `exists`, para la existencia de al menos uno, o `eval`, para expresiones lógicas.
- **RHS (Right Hand Side) o "Then":** contiene las acciones que se ejecutan cuando se cumplen las condiciones de LHS. En nuestro contexto, lo usamos para modificar la puntuación de la solución mediante el objeto `scoreHolder`. Aquí es donde vamos penalizando por cada regla incumplida.

En `guardianesScoreRules.drl` se utiliza el dialecto `mvel`, que ofrece una sintaxis concisa para acceder a las propiedades de los objetos Java, a los POJOs. También se usa `accumulate` que es una función de agregación necesaria para contar, por ejemplo, cuántos turnos a realizado un doctor y verificar si cumple con su contrato.

4.1.3 Solver

Para la configuración del solver se ha utilizado un archivo de configuración llamado `guardianesSolverConfig.xml` que se encuentra en `guardianes-service/src/main/resources/solver/guardianesSolverConfig.xml`. Este fichero configura el solver cargando las reglas definidas en el archivo Drools y vinculando las clases del modelo.

A continuación, se detalla el significado de cada etiqueta empleada en la configuración:

4.1.3.1 Entorno de ejecución

Utiliza la etiqueta `<environmentMode>` que define el comportamiento determinista del generador de números aleatorios interno del solver. Se ha seleccionado el valor `REPRODUCIBLE` que fuerza al solver a utilizar una semilla aleatoria fija. Esto garantiza que, ante el mismo conjunto de datos de entrada, el solver realiza siempre exactamente la misma ejecución y produzca el mismo resultado. Es la configuración recomendada durante la fase de desarrollo y pruebas, ya que facilita la depuración de errores.

Otras opciones serían:

- `NON_REPRODUCIBLE`, que no fija la semilla y se recomienda para producción. Esto provoca cambios ligeros en cada ejecución.
- `FAST_ASSERT` o `FULL ASSERT`, que son modos de depuración intensiva que ralentizan significativamente la ejecución, pero son valiosos para detectar errores en la lógica de las reglas o en el cálculo incremental de puntuaciones.

4.1.3.2 Hilos

Utiliza la etiqueta `<moveThreadCount>` que controla el uso de hilos para el cálculo de movimientos y puntuaciones, permitiendo o no la ejecución en paralelo. Se ha seleccionado el valor `NONE` que indica al solver que se ejecutará en un único hilo dado que nuestro problema es relativamente simple y así se simplifica la ejecución.

Otras opciones serían:

- `AUTO`, para que OptaPlanner detecte automáticamente el número de núcleos disponibles en el procesador del dispositivo y utilice la mayoría de ellos para generar multiples hilos. Ideal para problemas grandes.
- Valor numérico, para fijar un número específico de hilos a usar.

En el caso de que nuestro problema fuera más completo, bastaría con utilizar una de estas dos opciones para acelerar el proceso significativamente.

4.1.3.3 Vinculación del modelo

Se utilizan dos etiquetas para conectar la configuración del solver con las clases Java específicas del sistema.

- `<solutionClass>` define la clase que representa la solución completa, en este caso `Schedule`. Esta clase contiene la lista de datos fijos, la lista de variables y la puntuación final y está anotada con `@PlanningSolution`.
- `<entityClass>` define la clase que representa la entidad de planificación, en este caso `ShiftAssignment`. Estas son las instancias que el solver modificará durante la ejecución. La clase está anotada con `@PlanningEntity`.

4.1.3.4 Puntuación

Utiliza la etiqueta `<scoreDirectorFactory>` que indica al solver cómo debe calcular la calidad de una solución. Se usa `<scoreDrl>` para apuntar al archivo de reglas de Drools. El solver carga estas reglas y las evalúa cada vez que realiza un movimiento. La ruta indicada es `us/dit/service/model/entities/score/guardianesScoreRules.drl`.

Otra opción sería utilizar la interfaz `ConstraintProvider`, pero esto implicaría no usar Drools y basarnos en un código Java puro para definir la función de puntuación.

4.1.3.5 Finalización

Utiliza la etiqueta `<termination>` para establecer cuando debe detenerse el algoritmo. Se usa `<secondsSpentLimit>` para establecer una parada forzosa tras 200 segundos. Las pruebas empíricas realizadas prueban que este tiempo es suficiente para que el algoritmo encuentre una solución óptima en este contexto.

Otras opciones serían:

- `<unimprovedSecondsSpentLimit>` que detiene el solver si no se ha encontrado una solución mejor en unos determinados segundos. Muy útil para no perder el tiempo si el motor ya se ha estancado.
- `<bestScoreFeasible>` que detiene la ejecución en cuanto encuentra la primera solución que cumple todas las restricciones duras, es decir que la variable score tiene el valor “0hard”.
- `<minutesSpentLimit>` o `<hoursSpentLimit>` para ejecuciones de larga duración donde se establecen los minutos y las horas de ejecución respectivamente.

4.2 Integración con el modelo anterior

La integración de OptaPlanner dentro del flujo de negocio BPM requiere de una capa de servicio que actúe de intermediaria entre estos dos elementos. A su vez, gestiona la complejidad de la persistencia de datos y la coherencia transaccional.

4.2.1 SchedulerService

Esta clase funciona como el punto de entrada principal para la generación automática de calendarios. Actúa como gestor del ciclo de vida de la planificación.

El proceso de optimización es propenso a errores externos. Para mitigar esto, el servicio implementa una maquina de estados explícita sobre la entidad `Schedule`:

- Al inicio marca la planificación como "BEING_GENERATED". Esto funciona como un bloqueo lógico, impidiendo que otros procesos puedan modificar el calendario mientras el algoritmo está trabajando.
- En el caso de que la generación de la planificación se haga con éxito, transiciona a "PENDING_CONFIRMATION" habilitando la etapa de validación humana.
- El sistema captura excepciones en ejecución para transicionar a "GENERATION_ERROR" en el caso de que ocurra algún error y así, evitar que el sistema quede inutilizado.

Por otro lado, OptaPlanner requiere que todos los datos necesarios para la resolución estén cargados en memoria antes de empezar a resolver. El método `populateFactsAndEntities` extrae las entidades JPA de la base de datos, como pueden ser los doctores, la configuración de los días y los turnos vacíos, y los transforma para que el Solver pueda trabajar sobre ellos. Esto incluye la creación dinámica de los objetos `ShiftAssignment`, que actúan como contenedores vacíos que el algoritmo deberá rellenar.

Por último, a diferencia de la versión anterior en Python que se ejecutaba como un proceso externo, este servicio utiliza la interfaz `SolverManager`, para invocar al motor dentro de JVM [21]. Esta interfaz envuelve y gestiona varias instancias del solver para abstraer la complejidad de manejar hilos. La llamada es síncrona y bloqueante, lo que permite que el proceso de generación de planificación ocurra dentro de una única transacción larga. Esto garantiza la atomicidad, es decir, o se guarda una solución completa y válida o no se guarda nada. Esto evita corrupciones parciales en la base de datos.

4.2.2 OptaplannerGuardians

Esta clase es el servicio encargado de construir la solución final. Esta clase contiene la inteligencia necesaria para construir un problema viable antes de que el Solver intente resolverlo.

Uno de los problemas recurrentes encontrados durante el desarrollo de la solución fue que haya más huecos que médicos disponibles, por ejemplo, por la disponibilidad seleccionada por cada profesional. Esto es una regla blanda, se tiene que intentar respetar, pero es inviable dejar huecos vacíos. Para eso está el método `buildAndSaveInitialProblem` que soluciona este problema mediante un cálculo previo. Primero calcula la demanda total sumando el número mínimo de turnos que los doctores tienen que realizar, después calcula la capacidad base del problema, es decir, la cantidad de huecos predefinidos en el calendario. Si detecta que hay más demanda que capacidad, activa una subrutina de inyección de turnos, de manera que, el sistema cree nuevos turnos y los distribuya aleatoriamente entre los días laborables del mes. Esto asegura que el problema tenga solución, si no existiera este método, OptaPlanner se encontraría con restricciones imposibles de cumplir y nunca devolvería una solución válida.

OptaPlanner trabaja con la entidad `ShiftAssignment` vinculada a `Doctor`. Sin embargo, el sistema requiere una lógica basada en días como `ScheduleDay`. Para solucionar este problema, se crea el método `updateScheduleWithSolution` que actúa como traductor final. Una vez obtenida la planificación generada, esta función consigue transformar cada asignación de turnos en un objeto día, siguiendo la estructura que tenía el proyecto anterior.

Por último, esta clase genera un desglose textual de la puntuación obtenida. Esto aporta transparencia al proceso de decisión automática, un requisito clave.

4.2.3 SchedulerTaskService

Aunque esta clase es heredada de la arquitectura anterior, ha sufrido modificaciones críticas en su método `obtainSchedule` para soportar la complejidad del nuevo modelo de datos generado para OptaPlanner.

El modelo de datos de JPA utiliza por defecto la carga perezosa (Lazy Loading) para las colecciones. Esto es eficiente en general, pero problemático cuando una entidad compleja generada por el algoritmo debe ser enviada a la capa de visualización o validación fuera de la transacción original. Si se intenta acceder a la lista de turnos de un día fuera de la transacción, Hibernate lanzaría una excepción `LazyInitializationException`.

Para solucionar lo anterior sin comprometer el rendimiento global, se ha reescrito la lógica de recuperación de datos. En lugar de confiar en la carga automática, el servicio invoca explícitamente repositorios auxiliares. Estas llamadas fuerzan a Hibernate a ejecutar sentencias `JOIN` optimizadas que traen toda la información necesaria en memoria en el momento preciso. Esto asegura que, cuando el usuario gestor accede a la tarea de "Validar Planificación" en la interfaz web, el objeto `Schedule` está completamente listo para ser renderizado, eliminando latencias y errores de acceso a datos.

4.3 Pruebas y validación

Para garantizar la fiabilidad del algoritmo de generación de calendarios y, en concreto, la correcta implementación de la lógica de dimensionamiento dinámico de la demanda, es decir, cuando `OptaplannerGuardians` utiliza el método `buildAndSaveInitialProblem` para hacer el cálculo previo de la demanda y adapatar el problema, se ha diseñado un conjunto de pruebas unitarias. Estas pruebas permiten comprobar el correcto funcionamiento de la clase `OptaplannerGuardians` de forma aislada, sin depender del resto del sistema.

4.3.1 Entorno de pruebas y herramientas

Las pruebas se han implementado utilizando el framework JUnit 5, estándar en el ecosistema Java, combinado con Mockito para el aislamiento de dependencias.

Dado que la clase `OptaplannerGuardians` interactúa intensamente con la capa de persistencia (Repositorios JPA y `EntityManager`), se ha optado por una estrategia de Mocking. Esto consiste en simular el comportamiento de la base de datos, permitiendo:

- Controlar exactamente que datos devuelve la base de datos simulada, como pueden ser el número de médicos o festivos.
- Ejecutar las pruebas en milisegundos sin depender de entradas o salidas reales.
- Validar exclusivamente la lógica de cálculo de huecos, descartando cualquier otro tipo de errores que no sean de OptaPlanner.

4.3.2 Configuración del escenario

En nuestra clase de pruebas (`OptaplannerGuardiansTest`), el método con la etiqueta `@BeforeEach` se encarga de inicializar el entorno antes de cada ejecución. Se crean *mocks* [22] de todos los repositorios y de `EntityManager`.

Uno de los problemas afrontados durante estas pruebas fue la asignación de identificadores. En las pruebas unitarias, al no haber base de datos, los objetos carecen de identificadores, lo que provoca una excepción de tipo NullPointerException. Para solventar esto, se ha implementado un método auxiliar que utiliza la reflexión de Java. Este mecanismo permite inyectar valores en los campos privados de identificadores de las entidades, simulando el comportamiento de Hibernate tras una operación de persistencia.

4.3.3 Validación del algoritmo

El objetivo principal de estas pruebas es comprobar el mecanimo de dimensionamiento dinámico de la demanda descrito anteriormente. Este mecanismo debe ajustar la oferta de turnos (huecos disponibles) en función de los contratos de los médicos. Se han diseñado dos casos de prueba críticos:

- El primer caso es un escenario de alta demanda. Esta prueba simula una situación de sobrecarga, donde la plantilla médica requiere más horas de trabajo de las que el calendario ofrece por defecto. La prueba comprueba que el algoritmo ha detectado el déficit y ha generado dinámicamente nuevos turnos. La prueba es exitosa si el número final de turnos generados coincide exactamente con la demanda contractual, demostrando que el sistema se ha adaptado para permitir que todos los médicos cumplan su contrato.
- El segundo caso es un escenario de baja demanda. Esta prueba verifica que el sistema respeta los mínimos asistenciales del hospital incluso cuando hay pocos médicos. La prueba se asegura de que el algoritmo no reduce la oferta de turnos por debajo del mínimo necesario para el funcionamiento del hospital. La prueba valida que el número de turnos generados coincide con la capacidad base y no con la demanda de los médicos, asegurando la cobertura del servicio hospitalario.

5 CONCLUSIONES Y FUTURAS MEJORAS

5.1 Conclusiones

El objetivo de este trabajo era diseñar y desarrollar un sistema para la planificación de actividades asistenciales en un servicio hospitalario usando OptaPlanner, sustituyendo la anterior lógica que utilizaba Python. Este objetivo se ha cumplido. Tras el análisis de los resultados obtenidos se extraen las siguientes conclusiones principales:

Se ha logrado una integración nativa completa de OptaPlanner dentro del ecosistema Spring Boot y jBPM. A diferencia de la versión anterior, que dependía de la ejecución de un programa en Python, la nueva solución opera dentro de la misma JVM. Esto ha eliminado la latencia de arranque, ha suprimido errores de conversión y ha simplificado el despliegue de la aplicación.

Se ha aportado un algoritmo de dimensionamiento dinámico de la demanda. La solución desarrollada es capaz de detectar el déficit de huecos disponibles y adaptar la estructura del calendario, creando turnos de refuerzo, antes de iniciar la optimización. Esto garantiza que nuestro motor de optimización trabaje siempre sobre un escenario factible, evitando bloqueos por restricciones imposibles de cumplir.

5.2 Líneas futuras

Este proyecto es una mejora al proyecto de José Carlos Rodríguez Morón [4]. Se desarrolla utilizando OptaPlanner para sustituir al anterior algoritmo de generación de planificación diseñado en Python. Sin embargo, hay ciertas líneas futuras que se explican a continuación.

5.2.1 Administrar pesos y restricciones

Esta línea futura consiste en implementar una nueva tarea que permita al usuario gestor modificar los valores de `@ConstraintWeight`. Se almacenarían en la base de datos en lugar de en el código, para personalizar la importancia de las reglas sin intervención técnica.

5.2.2 Almacenar la explicación de la puntuación

Gracias a nuestra clase `OptaplannerGuardians` generamos un desglose en la salida de consola. Una futura mejora sería guardar estos datos y mostrarlos en la interfaz gráfica para el usuario administrador. De esta manera, al validar la planificación, podría volver a lanzarla o cambiar los pesos de las reglas si fuese necesario.

5.2.3 Almacenar la solución en ScheduleDay

Ahora mismo, la solución trabaja con la entidad `ShiftAssignment`. Esta línea futura consistiría en refactorizar `ScheduleDay` de manera que se pueda prescindir del método `updateScheduleWithSolution`. Esto se lograría transformando la solución de OptaPlanner en un objeto `Day`, siguiendo la estructura del proyecto anterior.

REFERENCIAS

[1] M. Á. González-Alorda Cantero, Shift scheduling and management service, Sevilla: Universidad de Sevilla, 2020.

[2] C. Cohen Calvo, Gestión de calendarios para un servicio hospitalario, Sevilla: Universidad de Sevilla, 2022.

[3] L. Marín Peña, Diseño y desarrollo de un bot Telegram para la gestión de actividades asistenciales, Sevilla: Universidad de Sevilla, 2022.

[4] J. C. Rodríguez Morón, BPM aplicado a procesos de gestión de actividades sanitarias, Sevilla: Universidad de Sevilla, 2024.

[5] Oracle Corporation, «What is Java technology and why do I need it?,» [En línea]. Available: https://www.java.com/en/download/help/whatis_java.html. [Último acceso: 08 01 2026].

[6] Oracle Corporation, «Java™ Platform, Standard Edition 8,» [En línea]. Available: https://docs.oracle.com/javase/8/docs/api/ . [Último acceso: 08 01 2026].

[7] Spring Framework, «Why Spring?,» [En línea]. Available: https://spring.io/why-spring . [Último acceso: 08 01 2026].

[8] Apache Maven Project, «What is Maven?,» [En línea]. Available: https://maven.apache.org/what-is-maven.html . [Último acceso: 08 01 2026].

[9] MVN REPOSITORY, «OptaPlanner Core >> 7.48.0.Final,» [En línea]. Available: https://mvnrepository.com/artifact/org.optaplanner/optaplanner-core/7.48.0.Final. [Último acceso: 08 01 2026].

[10] MVN REPOSITORY, «OptaPlanner Persistence Common >> 7.48.0.Final,» [En línea]. Available: https://mvnrepository.com/artifact/org.optaplanner/optaplanner-persistence-common/7.48.0.Final. [Último acceso: 08 01 2026].

[11] Testcontainers, «Testcontainers for Java,» [En línea]. Available: https://java.testcontainers.org. [Último acceso: 08 01 2026].

[12] KIE Community, «KIE Community - About,» [En línea]. Available: https://www.kie.org/about/. [Último acceso: 08 01 2026].

[13] jBPM, «What is jBPM?,» [En línea]. Available: https://www.jbpm.org/. [Último acceso: 08 01 2026].

[14] Red Hat Developer, « What is a KJAR?,» [En línea]. Available: https://developers.redhat.com/blog/2018/03/14/what-is-a-kjar. [Último acceso: 08 01 2026].

[15] SalesForce Help, «Business Rules Engine,» [En línea]. Available: https://help.salesforce.com/s/articleView?id=ind.business_rules_engine.htm&type=5. [Último acceso: 08 01 2026].

[16] Apache KIE, «OptaPlanner User Guide,» [En línea]. Available: https://kie.apache.org/docs/10.1.x/optaplanner/ . [Último acceso: 08 01 2026].

[17] GNU ORG, « Simulated Annealing,» [En línea]. Available: https://www.gnu.org/software/gsl/doc/html/siman.html. [Último acceso: 08 01 2026].

[18] GeeksforGeeks, « What is TABU Search?,» [En línea]. Available: https://www.geeksforgeeks.org/dsa/what-is-tabu-search/. [Último acceso: 08 01 2026].

[19] PostgreSQL Global Development Group, «PostgreSQL: About,» [En línea]. Available: https://www.postgresql.org/about/. [Último acceso: 08 01 2026].

[20] Drools, «Drools Documentation :: Introduction,» [En línea]. Available: https://docs.drools.org/latest/drools-docs/drools/introduction/index.html. [Último acceso: 08 01 2026].

[21] Oracle, « The Java® Virtual Machine Specification,» [En línea]. Available: https://docs.oracle.com/javase/specs/jvms/se8/html/. [Último acceso: 08 01 2026].

[22] Mockito, «Mockito framework site,» [En línea]. Available: https://site.mockito.org. [Último acceso: 08 01 2026].

GLOSARIO

ACID: Atomicity, Consistency, Isolation, Durability

API: Application Programming Interface

BPM: Business Process Management

BRE: Business Rule Engine

DRL: Drools Rule Language

IDE: Integrated Development Environment

JAR: Java Archive

JPA: Java Persistence API

JVM: Java Virtual Machine

KIE: Knowledge Is Everything

LHS: Left Hand Side

POJO: Plain Old Java Object

REST: Representational State Transfer.

RHS: Right Hand Side

SQL: Structured Query Language

TFG: Trabajo Fin de Grado

ANEXO

Despliegue del entorno de Desarrollo

En este apartado se describen los pasos a seguir para desplegar la aplicación en el entorno de desarrollo y poder seguir desarrollando sobre ella:

1. Instalación de un IDE. Cualquier IDE es válido. Por ejemplo, Eclipse IDE que se puede descargar aquí: https://www.eclipse.org/downloads/ .
2. Instalar Maven y añadir la variable PATH en Windows.

   Descarga: https://maven.apache.org/download.cgi

   Variable PATH: https://maven.apache.org/install.html
3. Instalar lombok y enlazarlo con el IDE que estemos utilizando. Se puede descargar aquí: https://projectlombok.org/download
4. Instalamos PostgreSQL. Se puede descargar aquí: https://www.postgresql.org/download/

   Una vez descargado, tendremos que configurarlo para poder crear y configurar la base de datos del proyecto:

   Accedemos con el usuario postgres:

   ```
   psql -U postgres
   ```

   Luego creamos la base de datos guardianes:

   ```
   CREATE DATABASE guardianes;
   ```

   Después creamos el usuario jbpm de esta forma y con esta contraseña:

   ```
   CREATE USER jbpm WITH PASSWORD 'jbpm.2.DDBB*';
   ```

   Accedemos a la base de datos de guardianes:

   ```
   \c guardianes
   ```

   Y le damos permiso al usuario jbpm para que pueda crear tablas y usarlas en el schema public:

   ```
   GRANT USAGE, CREATE ON SCHEMA public TO jbpm;
   ```

   Así estará lista la base de datos para usarse.
5. Instalamos Docker. Se puede descargar aquí: https://www.docker.com/products/docker-desktop/
6. Configurar el servidor DaviCAL en Docker.

   Primero deberemos ejecutar los siguientes comandos para crear un volumen para persistir la información del servidor y luego crearemos un contenedor que tendrá el servidor:

   ```
   docker volume create davical_data
   ```

   ```
   docker run -e HOST_NAME='localhost' -d --name davical -p 8080:80 -v davical_data:/var/lib/postgresql/data brompf/davical
   ```

   Nos conectamos a http://localhost:8080/

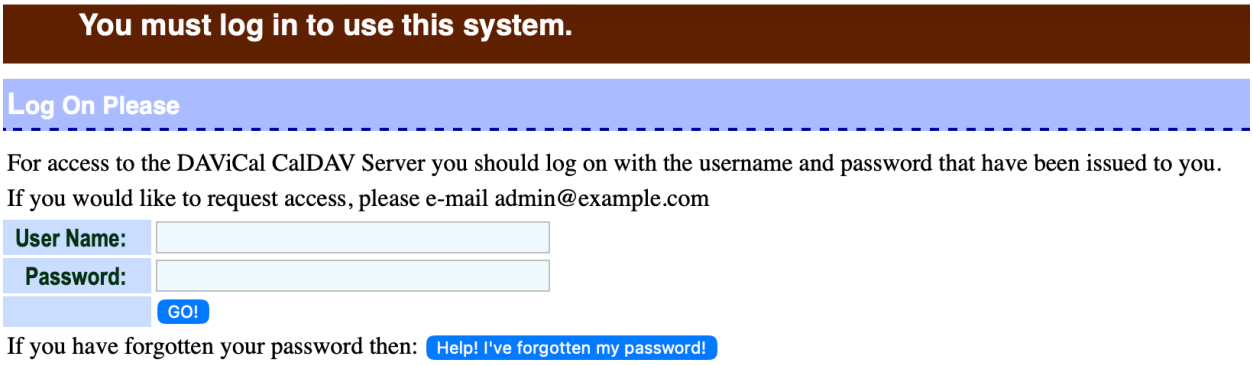

Ilustración 1. Log in del servidor DaviCal

Accedemos con el usuario admin y la contraseña 12345

Accedemos a User Functions y luego Create Principal para crear un usuario con permisos de escritura y lectura.

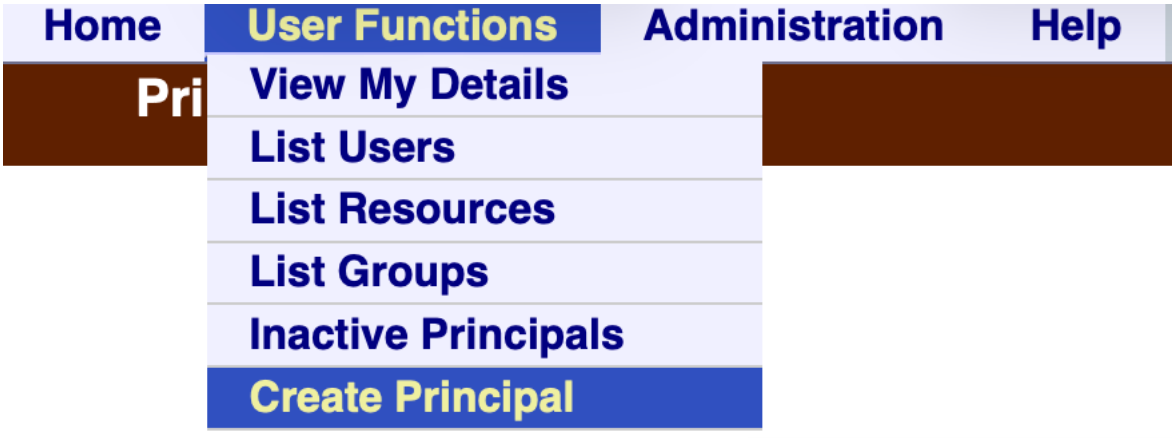

Ilustración 2. User Functions -> Create Principal

Creamos usuario y contraseña y pulsamos `All` para que nuestro nuevo usuario tenga todos los permisos.

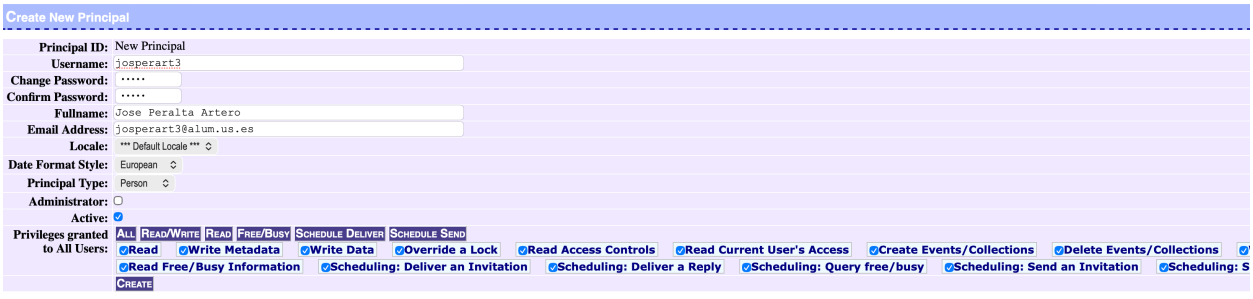

Ilustración 3. Creación de usuario

Ahora seleccionamos CREATE COLLECTION en el apartado de Principal Collections para crear el calendario.

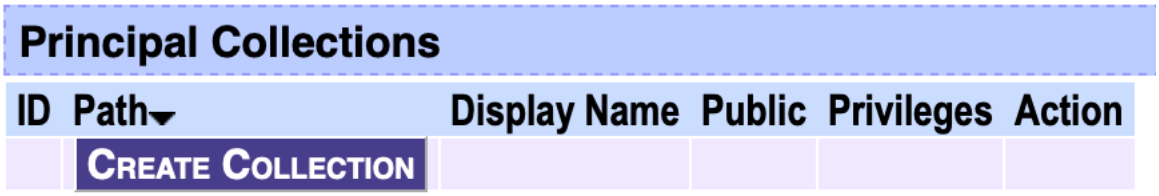

Ilustración 4. Pestaña de CREATE COLLECTION

Nos aparecerá una pestaña para crear el calendario.

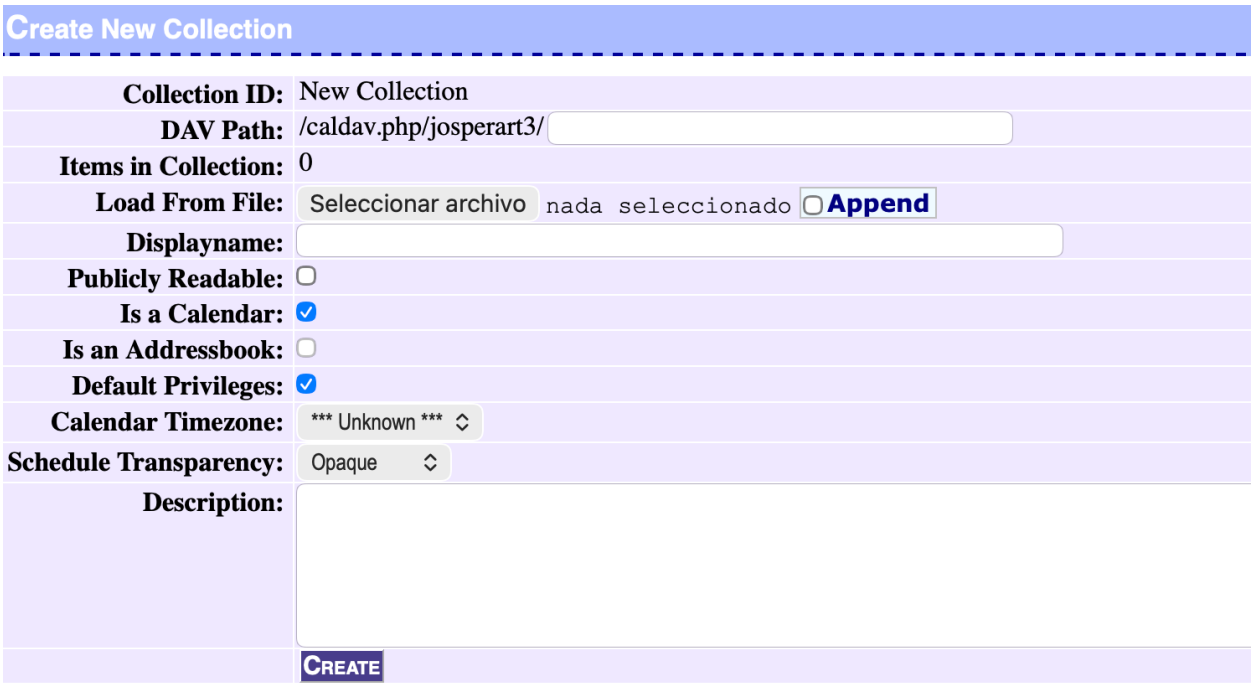

Ilustración 5. Menú de creación de calendario

Lo que tendremos que rellenar será en DAV Path y Displayname que tendremos que poner el nombre del calendario.

7. Clonar el repositorio git con el comando

   `git clone https://github.com/tfg-projects-dit-us/GuardianesBA`

8. Configurar application.properties

   Tendremos que modificar estas líneas del fichero para que el programa funcione correctamente:

```
#Calendar Properties
calendario.user = josperart3
calendario.psw = josperart3
calendario.tipo.cycle = Jornadas Complementarias
calendario.tipo.consultation = Consulta
calendario.tipo.shifts = Continuidades Asistenciales
calendario.uri = http://localhost:8080/caldav.php/josperart3/calendario1
#email Properties
```

Ilustración 6. Modificación de application.properties

Cambiamos el usuario y la contraseña por el establecido en el paso 6.

Cambiamos la uri poniendo nuestro usuario seguido del nombre del calendario establecido en el paso 6. En mi caso `/josperart3/calendario1`

9. Configuración de la aplicación de negocio.

   Tendremos que cambiar el temporizador cron situado en la línea 468 para que inicie cuando queramos nosotros.

```
<bpmn2:timerEventDefinition id="_HNOcWknQEe-E2dF2Fr56IA">
<!-- AQUI ES DONDE HAY QUE CAMBIAR LA CONFIGURACION PARA INICIAR EL PROCESO CUANDO QUERAMOS
    Si por ejemplo escribimos 0 27 13 * * ? esto quiere decir, que el proceso iniciará a las 13:27 -->
  <bpmn2:timeCycle xsi:type="bpmn2:tFormalExpression" id="_HNOcW0nQEe-E2dF2Fr56IA" language="cron">0 21 18 * * ?</bpmn2:timeCycle>
</bpmn2:timerEventDefinition>
```

Tal y como viene indicado, tiene el siguiente formato: `0 0 0 * * ?`. Normalmente estará definido como `0 0 12 15 * ?` para que inicie el día 15 de cada mes.

10. Ejecutar la aplicación.

    Por último, nos situaremos en la terminal dentro de la carpeta guardianes-service y ejecutaremos

    `.\launch.bat clean install -Ppostgres`

    Después accederemos a la aplicación desde la url http://localhost:8080/