Universidad Tecnológica del Perú

Planit - Gestor de Tareas

Curso:

Marcos de Desarrollo web

Integrantes:

- Caso Valenzuela, Jeaneli Rosmery
- Flores Martinez, Jahmil Raul
- Bustamante Chavez, Sebastian Franco

Profesor:

Adolfo Jorge Prado Ventocilla

2025

Contenido

1. Introducción

Planit es un sistema web para la gestión básica de tareas, desarrollado utilizando el framework Spring Boot para el backend y Thymeleaf para la interfaz gráfica. Su propósito es brindar una herramienta sencilla que permita registrar, visualizar y organizar tareas de manera eficiente.

La aplicación sigue el patrón de arquitectura Modelo-Vista-Controlador (MVC), lo que permite una separación clara entre la lógica de negocio, la interfaz gráfica y la persistencia de datos. Además, durante su desarrollo se aplicaron principios de buenas prácticas de programación, orientados a la responsabilidad única y al código limpio y modular.

2. Tecnologías Usadas

- **Spring Boot**:

  Se utilizó para la parte del servidor. Con esta herramienta se desarrollaron los controladores y servicios que gestionan la lógica de negocio, además de manejar todas las conexiones con la base de datos.

- **Thymeleaf:**

  Fue el motor de plantillas que usamos para la parte visual. Nos permitió crear páginas HTML dinámicas donde se muestran los datos que vienen del backend, por ejemplo, la lista de tareas o los reportes.

- **MySQL**:

  Fue la base de datos principal donde se almacena toda la información, como las tareas, los usuarios y los equipos. Gracias a Spring Boot y JPA, la integración con MySQL fue directa y sencilla.

- **Chart.js:**

Se aplicó para mostrar gráficos en la sección de reportes, lo que permite ver de forma visual cómo se distribuyen las tareas entre completadas y pendientes.

- **FullCalendar:**

  Se usó para implementar un calendario dentro de la aplicación, donde se puede visualizar fácilmente la fecha límite de cada tarea.

- **Bootstrap:**

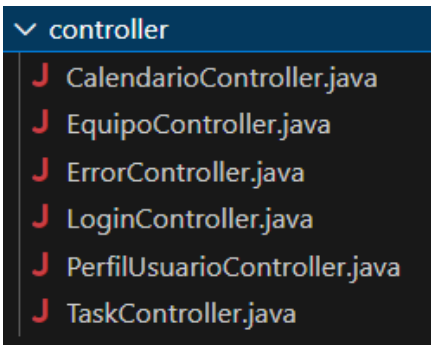
  Se empleó para darle un diseño moderno y responsivo a la interfaz. Con esta herramienta logramos que las páginas sean más ordenadas y se adapten bien a distintos tamaños de pantalla.

- **FontAwesome:**

  Fue utilizado para incluir iconos en la interfaz, lo cual hace que los botones y las opciones sean más intuitivos y visualmente agradables.

3. Arquitectura - Modelo-Vista-Controlador (MVC):

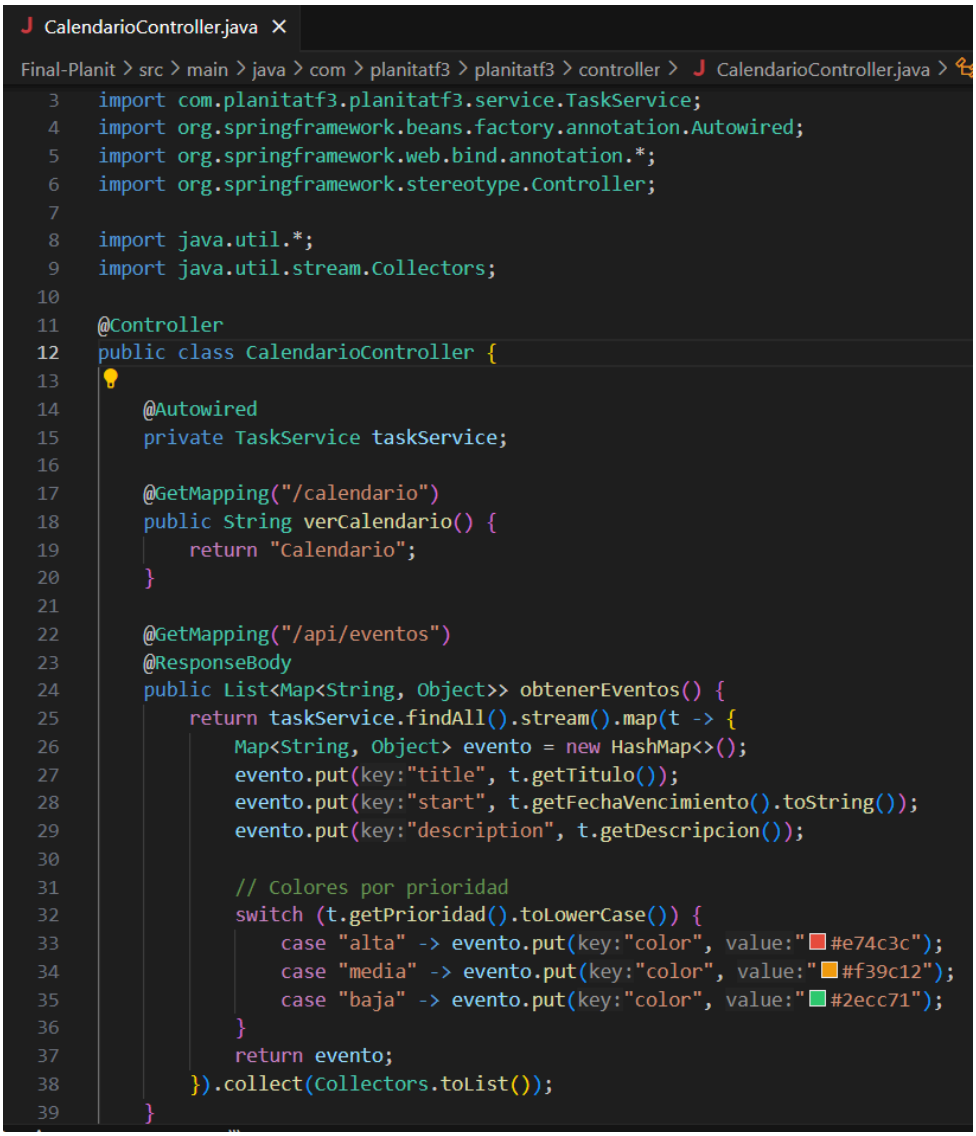
- Controllers:



- **Calendario Controller:**

  Su función principal es mostrar la vista del calendario con las tareas organizadas por fecha. Se utilizó @Controller para indicar que esta clase se encarga de atender solicitudes web relacionadas al calendario.

Se aplicó @GetMapping para definir la ruta /calendario, la cual devuelve la vista correspondiente con Thymeleaf. Además, dentro de la lógica del calendario, se podría utilizar un @ResponseBody para exponer los eventos en formato JSON, especialmente si FullCalendar consume datos directamente desde el backend. Cuando se utiliza integración con FullCalendar, normalmente se incluye un método con @GetMapping("/api/calendario") y @ResponseBody para devolver un listado de tareas o eventos con fechas. También se utiliza Model para pasar información necesaria a la vista, por ejemplo, tareas con fechas de vencimiento.



- **Equipo Controller:**

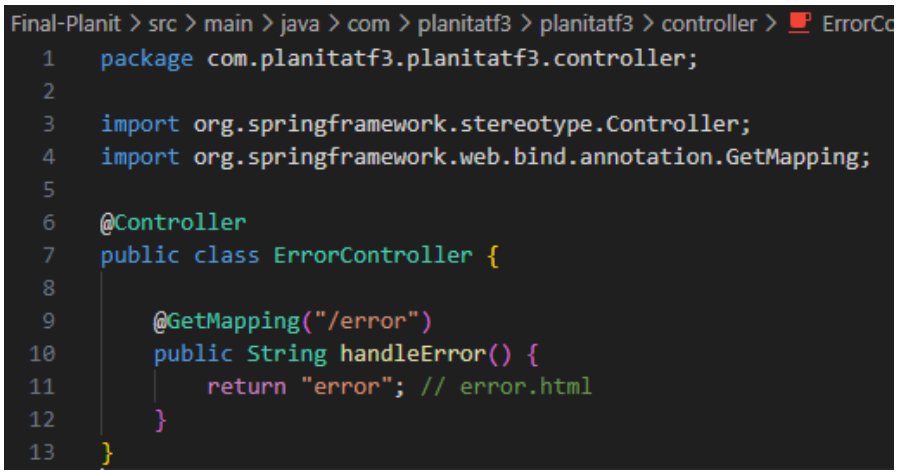
  Se usó @Controller para definir la clase como controlador, junto a @Autowired para inyectar el servicio EquipoService y UserService. Las rutas fueron definidas mediante @GetMapping para visualizar la lista de equipos y @PostMapping para registrar o actualizar información relacionada con los equipos.

  Cuando se necesitó capturar información desde formularios más completos, se utilizó @ModelAttribute, lo cual permitió recibir objetos completos de tipo Equipo desde los formularios Thymeleaf. También se usó @PathVariable para capturar valores dinámicos desde las rutas, como el ID del equipo cuando se consultan detalles específicos o se realizan modificaciones.

- **Error Controller:**

  En la clase ErrorController, se implementa el manejo centralizado de errores dentro de la aplicación. Se utiliza la anotación @Controller para definirlo como componente web, junto con @GetMapping("/denegado") para mostrar una vista personalizada cuando ocurre un error de acceso denegado.

  También se puede utilizar @RequestMapping("/error") para manejar cualquier error general, devolviendo una página personalizada de error en vez de la página de error por defecto. Esta clase no suele usar Model, salvo que se quiera enviar detalles específicos del error a la vista. Sirve principalmente para mejorar la experiencia del usuario cuando se encuentra con rutas no autorizadas o páginas no existentes.



- **Login Controller:**

  En el LoginController se implementaron rutas para el inicio de sesión y registro. Se utilizó @Controller para definir la clase como parte de la lógica web y @Autowired para inyectar el UserRepository o UserService. Las rutas GET (@GetMapping) se encargan de mostrar el formulario de login y el de registro, mientras que las rutas POST (@PostMapping) procesan los datos enviados desde los formularios HTML.

Aquí, @RequestParam permitió capturar los datos de usuario y contraseña directamente desde el formulario para validarlos con los datos almacenados. También se manejó HttpSession para gestionar la sesión del usuario autenticado.

Final-Planit > src > main > java > com > planitatf3 > planitatf3 > controller > LoginController.java

```java
@Controller
public class LoginController {

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private PasswordEncoder passwordEncoder;

    // Mostrar formulario de login (Spring Security lo usa automáticamente)
    @GetMapping("/login")
    public String loginForm() {
        return "Login"; // Tu vista personalizada
    }

    // Mostrar formulario de registro
    @GetMapping("/registro")
    public String mostrarFormularioRegistro() {
        return "registro";
    }

    // Procesar registro
    @PostMapping("/registro")
    public String registrarUsuario(@RequestParam String username,
            @RequestParam String password,
            @RequestParam String confirmpassword,
            Model model) {

        if (!password.equals(confirmpassword)) {
            model.addAttribute("error", "Las contraseñas no coinciden");
            return "registro";
        }

        if (userRepository.findByUsername(username).isPresent()) {
            model.addAttribute("error", "El nombre de usuario ya existe");
            return "registro";
        }

        User nuevoUsuario = new User();
        nuevoUsuario.setUsername(username);
        nuevoUsuario.setPassword(passwordEncoder.encode(password)); // encriptado
        nuevoUsuario.setRol(Rol.USER); // Asignación por defecto
        userRepository.save(nuevoUsuario);

        model.addAttribute("mensaje", "Registro exitoso. Ahora puedes iniciar sesión.");
        return "redirect:/Usuario";
    }
}
```

- **Perfil Usuario Controller:**

  En esta clase, se utilizó @Controller y @Autowired para manejar el perfil del usuario autenticado. Una de las características especiales fue el uso de @AuthenticationPrincipal, que permitió obtener directamente la información del usuario autenticado y mostrarla en la vista del perfil. Se utilizó Model para enviar los datos del perfil hacia la plantilla HTML y visualizar información personalizada.

Final-Planit > src > main > java > com > planitatf3 > planitatf3 > controller > PerfilUsuarioController.java

```java
package com.planitatf3.planitatf3.controller;

import com.planitatf3.planitatf3.model.PerfilUsuario;
import com.planitatf3.planitatf3.security.CustomUserDetails;
import com.planitatf3.planitatf3.service.PerfilUsuarioService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class PerfilUsuarioController {

    @Autowired
    private PerfilUsuarioService perfilUsuarioService;

    @GetMapping("/perfil")
    public String verPerfil(@AuthenticationPrincipal CustomUserDetails userDetails, Model model) {
        Long userId = userDetails.getUser().getId();

        // Buscar el perfil asociado al usuario autenticado
        PerfilUsuario perfil = perfilUsuarioService.findByUserId(userId).orElse(null);

        // Si existe, lo envías al modelo. Si no, muestras vista vacía o mensaje
        model.addAttribute("perfil", perfil);
        return "PerfilUsuario"; // Vista HTML que mostrarás
    }
}
```

- **Task Controller:**

  Se utilizó principalmente la anotación:

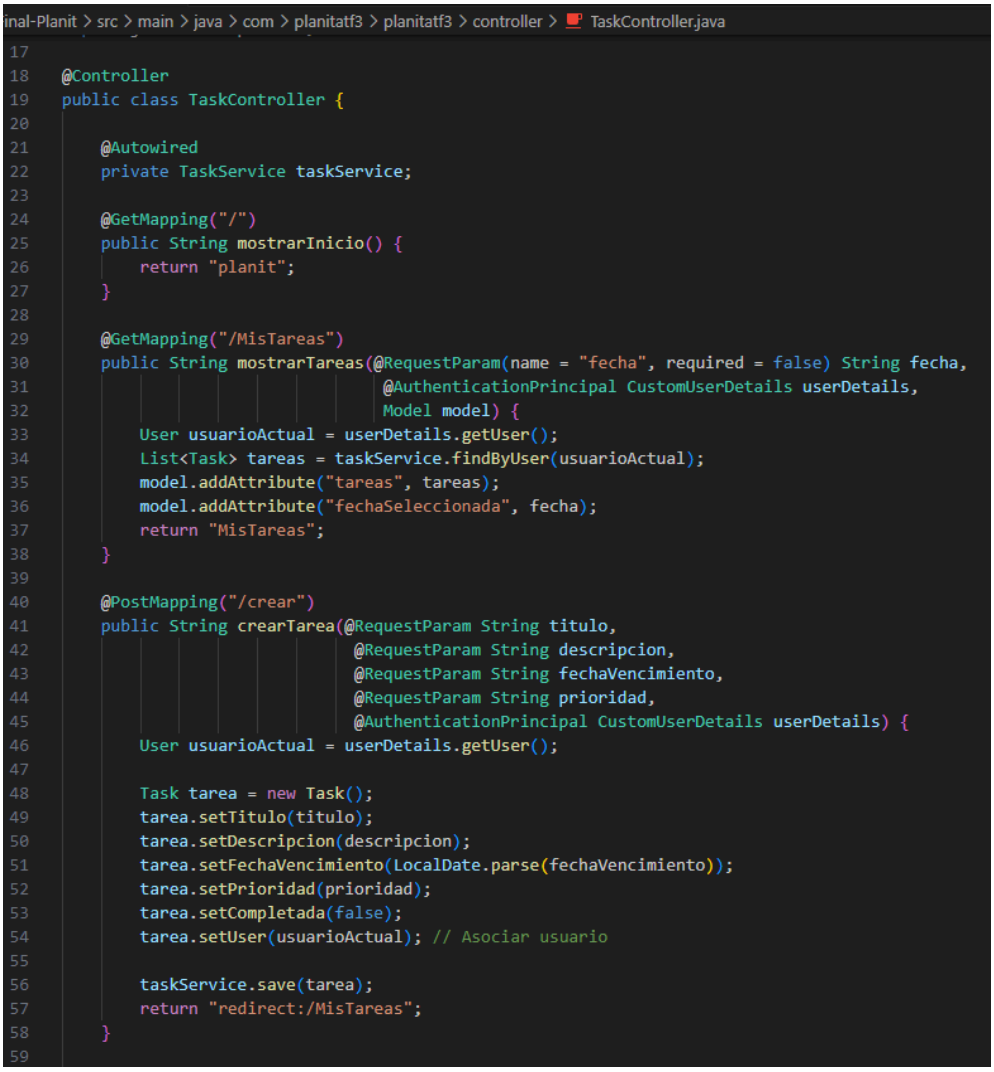
  - @Controller para indicar que se trata de un controlador encargado de manejar solicitudes web.
  - @Autowired para inyectar el servicio TaskService y así acceder a la lógica de negocio relacionada con las tareas.

Las rutas GET fueron definidas con:

- @GetMapping, por ejemplo para mostrar la lista de tareas o cargar la página principal.
- @PostMapping, capturando los datos ingresados desde el frontend mediante @RequestParam.

También se empleó Model para enviar datos dinámicos a las vistas, por ejemplo, la lista de tareas que se muestran en el HTML.

- @ResponseBody en el método /estadisticas para retornar información en formato JSON, especialmente útil para los gráficos con Chart.js.



TaskController.java

Final-Planit > src > main > java > com > planitatf3 > planitatf3 > controller > TaskController.java

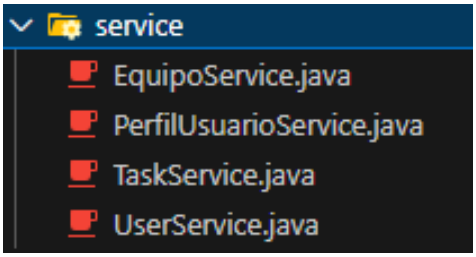
```
19  public class TaskController {
60      @PostMapping("/completar")
61      public String completarTarea(@RequestParam Long id,
62                                   @AuthenticationPrincipal CustomUserDetails userDetails) {
63          User usuarioActual = userDetails.getUser();
64
65          Optional<Task> optionalTarea = taskService.findById(id);
66          optionalTarea.ifPresent(tarea -> {
67              if (tarea.getUser().getId().equals(usuarioActual.getId())) {
68                  tarea.setCompletada(true);
69                  taskService.save(tarea);
70              }
71          });
72          return "redirect:/MisTareas";
73      }
74
75      @PostMapping("/eliminar")
76      public String eliminarTarea(@RequestParam Long id,
77                                  @AuthenticationPrincipal CustomUserDetails userDetails) {
78          User usuarioActual = userDetails.getUser();
79
80          Optional<Task> optionalTarea = taskService.findById(id);
81          optionalTarea.ifPresent(tarea -> {
82              if (tarea.getUser().getId().equals(usuarioActual.getId())) {
83                  taskService.deleteById(id);
84              }
85          });
86          return "redirect:/MisTareas";
87      }
88
89      @PostMapping("/actualizar")
90      public String actualizarTarea(@RequestParam Long id,
91                                    @RequestParam String titulo,
92                                    @RequestParam String descripcion,
93                                    @RequestParam String fechaVencimiento,
94                                    @RequestParam String prioridad,
95                                    @RequestParam(required = false) boolean completada,
96                                    @AuthenticationPrincipal CustomUserDetails userDetails) {
97          User usuarioActual = userDetails.getUser();
98
99          Optional<Task> optionalTarea = taskService.findById(id);
100         optionalTarea.ifPresent(tarea -> {
101             if (tarea.getUser().getId().equals(usuarioActual.getId())) {
102                 tarea.setTitulo(titulo);
103                 tarea.setDescripcion(descripcion);
104                 tarea.setFechaVencimiento(LocalDate.parse(fechaVencimiento));
105                 tarea.setPrioridad(prioridad);
106                 tarea.setCompletada(completada);
```

TaskController.java

Final-Planit > src > main > java > com > planitatf3 > planitatf3 > controller > TaskController.java

```
19  public class TaskController {
90      public String actualizarTarea(@RequestParam Long id,
111         return "redirect:/MisTareas";
112     }
113
114     @GetMapping("/Inicio")
115     public String inicio() {
116         return "Inicio";
117     }
118
119     @GetMapping("/reportes")
120     public String reportes() {
121         return "Reportes";
122     }
123
124     // JSON para gráficos personalizados por usuario
125     @GetMapping("/estadisticas")
126     @ResponseBody
127     public Map<String, Integer> obtenerResumen(@AuthenticationPrincipal CustomUserDetails userDetails) {
128         User usuarioActual = userDetails.getUser();
129
130         int completadas = taskService.contarPorEstadoYUsuario(true, usuarioActual);
131         int pendientes = taskService.contarPorEstadoYUsuario(false, usuarioActual);
132
133         return Map.of(
134                 "completadas", completadas,
135                 "pendientes", pendientes
136         );
137     }
138 }
139
```

- Services: Lógica de negocio



- Equipo Service:

  Gestiona toda la lógica relacionada con los equipos de trabajo.

  Funciones principales:

  - findAll(): Obtiene todos los equipos registrados.
  - findById(Long id): Busca un equipo por su ID.

    - existsByNombre(String nombre): Verifica si existe un equipo con ese nombre.
    - save(Equipo equipo): Guarda o actualiza un equipo.
    - deleteById(Long id): Elimina un equipo por su ID.
    - findEquiposPorUsuario(User user): Lista los equipos en los que participa un usuario.
    - agregarMiembro(Long equipoId, Long userId): Añade un usuario a un equipo si no está ya incluido.
    - removerMiembro(Long equipoId, Long userId): Elimina un usuario del equipo si está presente.
- Perfil Usuario Service:

  Administra los datos del perfil de usuario.

  Funciones principales:

    - findByUserId(Long userId): Busca el perfil asociado a un usuario.
    - save(PerfilUsuario perfil): Guarda o actualiza el perfil del usuario.
- Task Service:

  Encargado de gestionar las tareas dentro del sistema. Funciones principales:

    - findAll(): Lista todas las tareas del sistema.
    - findByUser(user): Devuelve solo las tareas asociadas a un usuario específico.
    - findById(id): Busca una tarea por su ID.
    - save(task): Guarda o actualiza una tarea.
    - deleteById(id): Elimina una tarea por su ID.

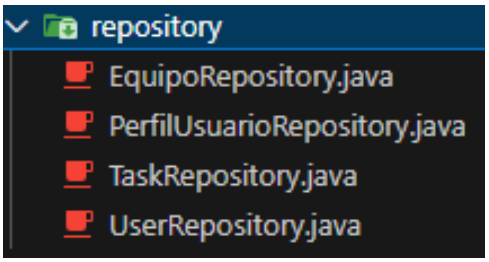
    - contarPorEstado(completada): Cuenta cuántas tareas están completadas o pendientes en total.
    - contarPorEstadoYUsuario(completada, user): Cuenta tareas completadas o pendientes para un usuario específico.
- User Service:

  Gestiona la lógica relacionada con usuarios del sistema.

  Funciones principales:

    - findByUsername(String username): Busca un usuario por su nombre de usuario.
    - save(User user): Guarda un nuevo usuario con la contraseña cifrada usando BCryptPasswordEncoder.
    - existsByUsername(String username): Verifica si el nombre de usuario ya está registrado.
    - findAll(): Lista todos los usuarios (útil para asignarlos a equipos).

- **Repositories:** Acceso a datos



- Equipo Repository:

  Este repositorio gestiona el acceso a los datos de los equipos. Funciones principales:

    - findAll(): Obtiene todos los equipos.
    - findById(id): Busca un equipo por su ID.
    - existsByNombre(nombre): Verifica si existe un equipo con ese nombre.
    - save(equipo): Guarda o actualiza un equipo.
    - deleteById(id): Elimina un equipo por su ID.
    - findByMiembrosContaining(user): Devuelve los equipos en los que participa un usuario específico.
- Perfil Usuario Repository:
- Encargado de acceder a los datos del perfil de usuario. Funciones principales:
    - findByUserId(userId): Busca el perfil asociado a un usuario por su ID.
    - save(perfil): Guarda o actualiza el perfil del usuario.
- Task Repository:

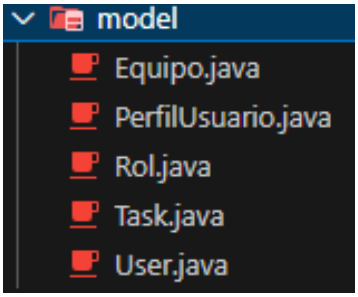
  Este repositorio maneja el acceso a las tareas almacenadas en la base de datos. Funciones principales:

    - findAll(): Lista todas las tareas.
    - findById(id): Busca una tarea por su ID.
    - findByUser(user): Devuelve las tareas asociadas a un usuario.
    - save(task): Guarda o actualiza una tarea.
    - deleteById(id): Elimina una tarea por su ID.
    - countByCompletada(completada): Cuenta tareas según su estado (completada o pendiente).
    - countByCompletadaAndUser(completada, user): Cuenta tareas por estado y por usuario.
- User Repository:

Este repositorio gestiona el acceso a los datos de los usuarios. Funciones principales:

- findAll(): Lista todos los usuarios.
- findById(id): Busca un usuario por su ID.
- findByUsername(username): Busca un usuario por su nombre de usuario.
- save(user): Guarda o actualiza un usuario.

- **Models:** Entidades



- Equipo:

  Representa un grupo de trabajo compuesto por varios usuarios.

  Atributos principales:

  - id: Identificador único.
  - nombre: Nombre del equipo (único).
  - miembros: Conjunto de usuarios que pertenecen al equipo.

  Relaciones:

  - @ManyToMany con User.

- Perfil Usuario:

Contiene información personal adicional del usuario.

Atributos principales:

- id: Identificador del perfil.
- nombre, apellido: Datos personales.
- telefono: Número de contacto.
- fotoUrl: URL de imagen de perfil.
- user: Usuario asociado.

Relaciones:

- @OneToOne con User.

- Rol:

  Enum que define los roles disponibles en el sistema.

  Valores posibles:

  - USER: Usuario estándar.
  - ADMIN: Usuario con privilegios administrativos.

  Uso: Asignado directamente en la entidad User mediante @Enumerated(EnumType.STRING).

- Task:

  Representa una tarea asignada a un usuario.

  Atributos principales:

  - id: Identificador único.
  - titulo: Título de la tarea.

    - descripcion: Detalles de la tarea.
    - fechaVencimiento: Fecha límite.
    - prioridad: Nivel de prioridad.
    - completada: Estado booleano (true/false).
    - user: Usuario responsable de la tarea.

  Relaciones:

    - @ManyToOne con User.

- User:

  Representa a un usuario registrado en el sistema.

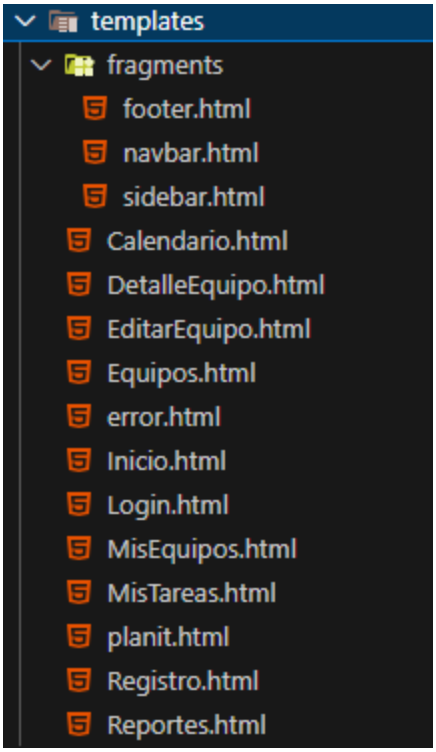
  Atributos principales:

    - id: Identificador único.
    - username: Nombre de usuario (único).
    - password: Contraseña cifrada.
    - rol: Enum (USER, ADMIN) que define el rol del usuario.
    - perfil: Relación uno a uno con PerfilUsuario.

  Relaciones:

    - @OneToOne con PerfilUsuario (bidireccional).
    - @ManyToMany con Equipo (usuarios pueden pertenecer a varios equipos).
    - @OneToMany con Task (un usuario puede tener varias tareas).

- **Templates:** Interfaz gráfica



4. Funcionalidades

Módulo	Descripción
Inicio	Dashboard con gráfico resumen de tareas
Mis Tareas	CRUD tareas con completar y eliminar
Equipos	Gestión de equipos y miembros
Calendario	Calendario FullCalendar de tareas
Reportes	Gráfico estadístico con Chart.js

Inicio:

Login:

Calendario:

Tareas:

Reportes:

5. Conclusión

El proyecto demuestra el uso de Spring Boot y patrones de diseño para crear un gestor de tareas funcional y limpio.

7. Recomendaciones

Tras revisar los criterios del proyecto se recomienda lo siguiente para futuras mejoras o ampliaciones del sistema Planit:

- Incluir una funcionalidad de recuperación de contraseña para mejorar la gestión de usuarios.
- Documentar una posible estrategia de despliegue en nube (Heroku, Railway, Fly.io) para ejecución remota.
- Implementar un módulo opcional de reportes avanzados (por prioridad, fechas) usando Chart.js.