

2025

Tutorial Desarrollo de API REST con Spring Boot

HENRY ANTONIO MENDOZA PUERTA

Introducción

Este tutorial está orientado al desarrollo de una API REST utilizando Spring Boot, enfocada en la gestión de tareas para equipos de desarrollo. El objetivo es construir el backend de un sistema tipo *Task Manager*, que exponga endpoints para registrar developers, crear tareas y controlar su asignación según reglas de negocio específicas.

El enfoque se centra en la lógica de negocio y el diseño de una API RESTful, sin incluir interfaz gráfica, de forma que pueda ser consumida por cualquier cliente (app web, móvil u otro servicio).

Reglas de Negocio

1. Límite de tareas activas por developer: Un developer no puede tener más de N tareas activas (`PENDING` o `IN_PROGRESS`) asignadas simultáneamente.
2. Estados válidos de una tarea:
 - o `PENDING`: tarea creada, aún no iniciada.
 - o `IN_PROGRESS`: en desarrollo.
 - o `COMPLETED`: finalizada.
 - o `CANCELLED`: descartada.

Las transiciones deben ser coherentes (ej. no se puede pasar de `COMPLETED` a `IN_PROGRESS`).
3. Restricción en la asignación: Solo puede asignarse una tarea si está en estado `PENDING` y el developer no ha superado su límite.
4. Control en el cambio de estado: Una tarea no puede marcarse como `COMPLETED` si no estuvo antes en `IN_PROGRESS`.
5. Restricción para eliminar developers: Un developer no puede eliminarse si tiene tareas activas (no completadas o canceladas).
6. Validaciones obligatorias:
 - o El título y descripción de la tarea son requeridos.
 - o El nombre del developer debe ser único y no vacío.

¿Qué aprenderás?

- Estructurar un proyecto Spring Boot para una API REST.
- Definir modelos, servicios, controladores y repositorios.
- Aplicar validaciones y lógica de negocio.
- Implementar manejo de excepciones centralizado.
- Usar DTOs para desacoplar la lógica interna de las peticiones y respuestas.
- Exponer endpoints RESTful de manera organizada.
- Probar los endpoints con herramientas como Postman.

Este ejercicio práctico está diseñado para fortalecer tus conocimientos en el desarrollo de APIs limpias, mantenibles y listas para producción.

Tecnologías y Herramientas

Este tutorial emplea tecnologías modernas del ecosistema Java para el desarrollo de una API REST. A continuación, se detallan las herramientas y frameworks que se utilizarán:

Lenguaje y Framework

El desarrollo de esta API REST se basa en los siguientes proyectos del ecosistema Spring:

- Java 21 – Lenguaje de programación base.
- Spring Boot – Proyecto principal para la construcción y ejecución simplificada de aplicaciones backend.
- Spring Data – Proyecto que facilita el acceso a datos mediante abstracciones sobre JPA y otras tecnologías de persistencia.

Base de Datos

- PostgreSQL – Sistema de gestión de base de datos relacional, usado para almacenamiento persistente.

Validación y Manejo de Errores

- Jakarta Bean Validation – Validación declarativa de datos de entrada (@NotNull, @Size, etc.).
- Spring Exception Handling (@RestControllerAdvice) – Mecanismo centralizado para manejar excepciones y retornar mensajes de error en formato JSON.

Pruebas Funcionales

- Postman – Herramienta para ejecutar y verificar llamadas HTTP a los endpoints REST. Se usarán pruebas funcionales para validar el comportamiento completo de la API y sus reglas de negocio.

Build y Configuración

- Maven – Gestión de dependencias y construcción del proyecto.
- application.properties – Archivo de configuración para definir la conexión a PostgreSQL, puertos, y otras propiedades clave del entorno.

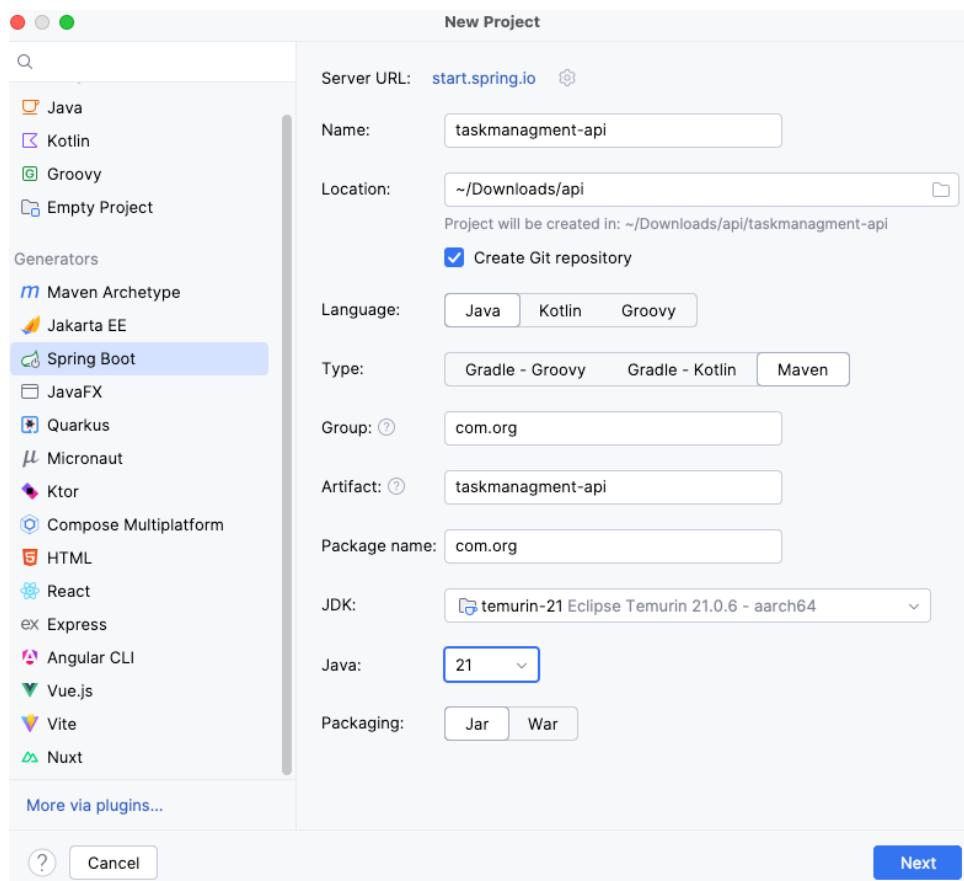
Control de Versiones

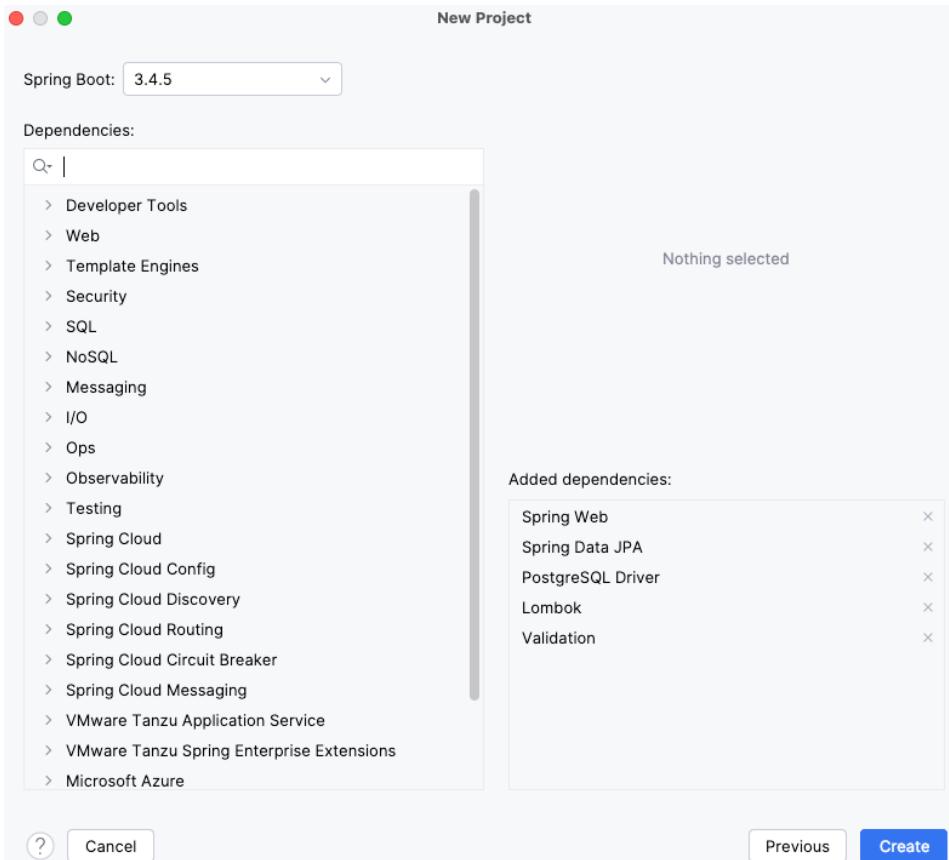
- Git – Sistema de control de versiones distribuido.
- GitHub – Plataforma de alojamiento de código fuente.
- Git Flow – Estrategia de ramificación para organizar el flujo de trabajo en el repositorio.

Configuración del Proyecto

Para implementar la solución, el proyecto incluirá las siguientes **dependencias** en su archivo `pom.xml` (Maven):

- **Spring Web** – Para construir controladores REST y manejar peticiones HTTP.
- **Spring Data JPA** – Para la integración con bases de datos relacionales mediante repositorios.
- **PostgreSQL Driver** – Para conectarse a una base de datos PostgreSQL.
- **Lombok** – Para reducir el código boilerplate mediante anotaciones como `@Getter`, `@Setter`, `@Builder`, etc.
- **Jakarta Bean Validation** – Para validar entradas de usuario usando anotaciones como `@NotNull`, `@Size`, etc.





Estructura de Carpetas del Proyecto

El proyecto `taskmanagement-api` seguirá una estructura modular basada en buenas prácticas de desarrollo con Spring Boot. A continuación, se presenta la organización propuesta.

```

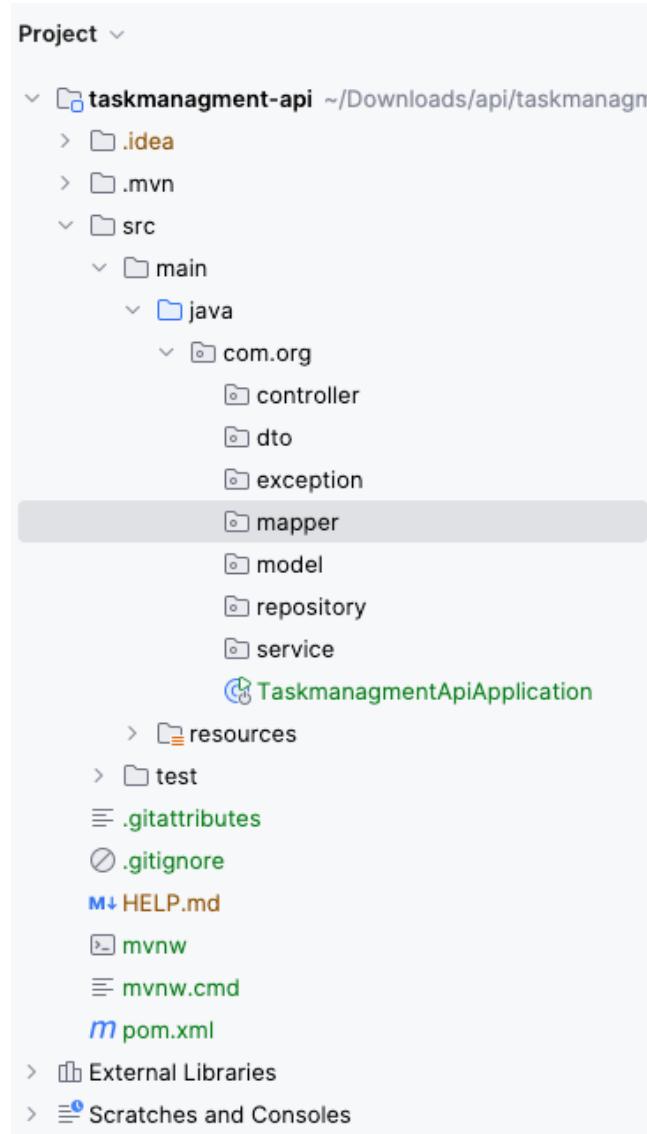
taskmanagement-api/
├── src/
│   └── main/
│       ├── java/
│       │   └── com/
│       │       └── org/
│       │           └── taskmanagement/
│       │               ├── controller/      # Controladores REST
│       │               ├── service/       # Lógica de negocio y validaciones
│       │               ├── repository/    # Interfaces JPA para acceso a datos
│       │               ├── model/        # Entidades JPA (Developer, Task, etc.)
│       │               ├── dto/         # Objetos de transferencia de datos
│       │               ├── exception/   # Manejo de excepciones personalizadas
│       │               ├── mapper/      # Conversión entre entidades y DTOs
│       │               └── TaskManagementApiApplication.java # Clase principal
│
│       └── resources/
│           └── application.properties      # Configuración del entorno (PostgreSQL, puertos, etc.)
|
└── pom.xml                                # Archivo de configuración de dependencias y build (Maven)

```

Descripción de carpetas clave

- `controller/`: Define los endpoints de la API REST.
- `service/`: Implementa la lógica de negocio y validaciones.
- `repository/`: Define interfaces JPA para operaciones sobre la base de datos.

- model/: Contiene las entidades del dominio.
- dto/: Define objetos de entrada y salida para evitar exponer directamente las entidades.
- exception/: Agrupa excepciones personalizadas y controladores de errores (@RestControllerAdvice).
- mapper/: Encapsula la lógica de conversión entre entidades y DTOs, si se utiliza mapeo manual o con librerías como MapStruct o ModelMapper
- resources/: Contiene archivos de configuración.
- pom.xml: Archivo principal de Maven para gestionar dependencias y configuración del proyecto.



Creación del Repositorio en GitHub y Primer Commit

A continuación, se detallan los pasos para crear un repositorio en GitHub y subir tu proyecto desde una carpeta local vacía:

Crear el Repositorio en GitHub

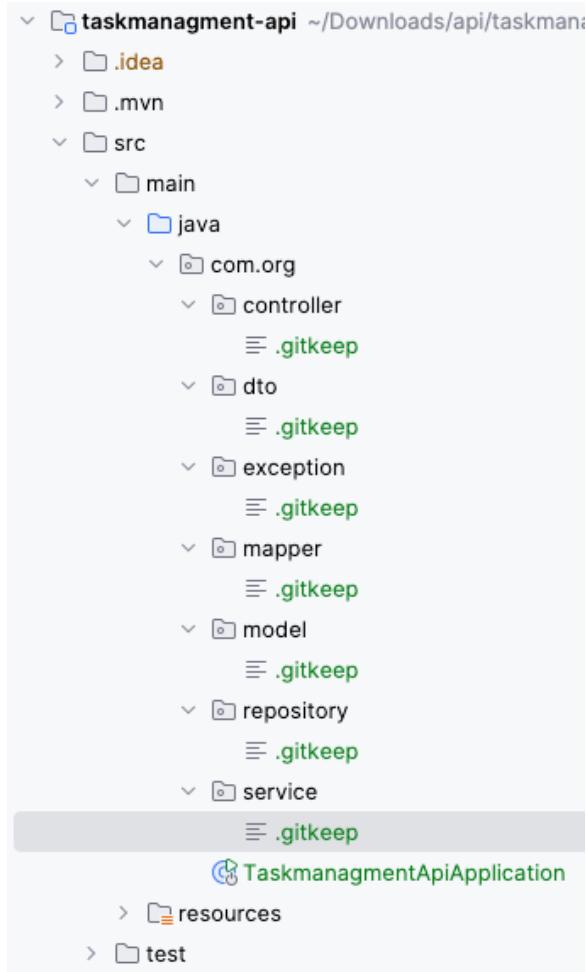
1. Ingresa a <https://github.com> y accede con tu cuenta.
2. Haz clic en el botón "New" o "Nuevo repositorio".
3. Completa los datos del repositorio:
 - o Repository name: taskmanagement-api
 - o Visibility: Pública o privada, según tu preferencia.
 - o No selecciones README, .gitignore ni licencia.
4. Haz clic en Create repository.

The screenshot shows the 'Create a new repository' page on GitHub. The form fields are as follows:

- Owner ***: hampcode
- Repository name ***: taskmanagement-api (highlighted with a blue border)
- Description (optional)**: (empty text area)
- Visibility**:
 - Public**: Anyone on the internet can see this repository. You choose who can commit.
 - Private**: You choose who can see and commit to this repository.
- Initialize this repository with:**
 - Add a README file**: This is where you can write a long description for your project. [Learn more about READMEs](#).
- Add .gitignore**
 - .gitignore template: **None** (dropdown menu)
 - Choose which files not to track from a list of templates. [Learn more about ignoring files](#).
- Choose a license**
 - License: **None** (dropdown menu)
 - A license tells others what they can and can't do with your code. [Learn more about licenses](#).
- Information**:
 - ⓘ You are creating a public repository in your personal account.
- Create repository** button (green button at the bottom right)

Subir el Proyecto desde tu Carpeta Local

1. Abre una terminal o consola en la carpeta taskmanagement-api.
2. Agregar desde IntelliJ IDEA: clic derecho en la carpeta vacía → New → File → escribe .gitkeep



Inicializa Git y realiza el primer commit:

- git init
- git add .
- git commit -m "estructura inicial del proyecto"

```
→ taskmanagement-api git:(master) ✘ git init
Reinicializado el repositorio Git existente en /Users/hampcode/Downloads/api/taskmanagement-api/.git/
→ taskmanagement-api git:(master) ✘ git add .
→ taskmanagement-api git:(master) ✘ git commit -m "primer commit"
```

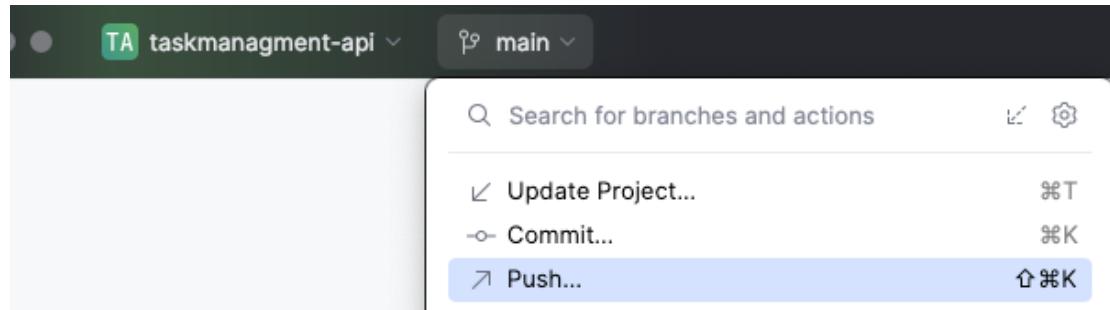
- git branch -M main
- git remote add origin https://github.com/hampcode/taskmanagement-api.git

```
→ taskmanagement-api git:(master) git branch -M main
→ taskmanagement-api git:(main) git remote add origin https://github.com/hampcode/taskmanagement-api.git
```

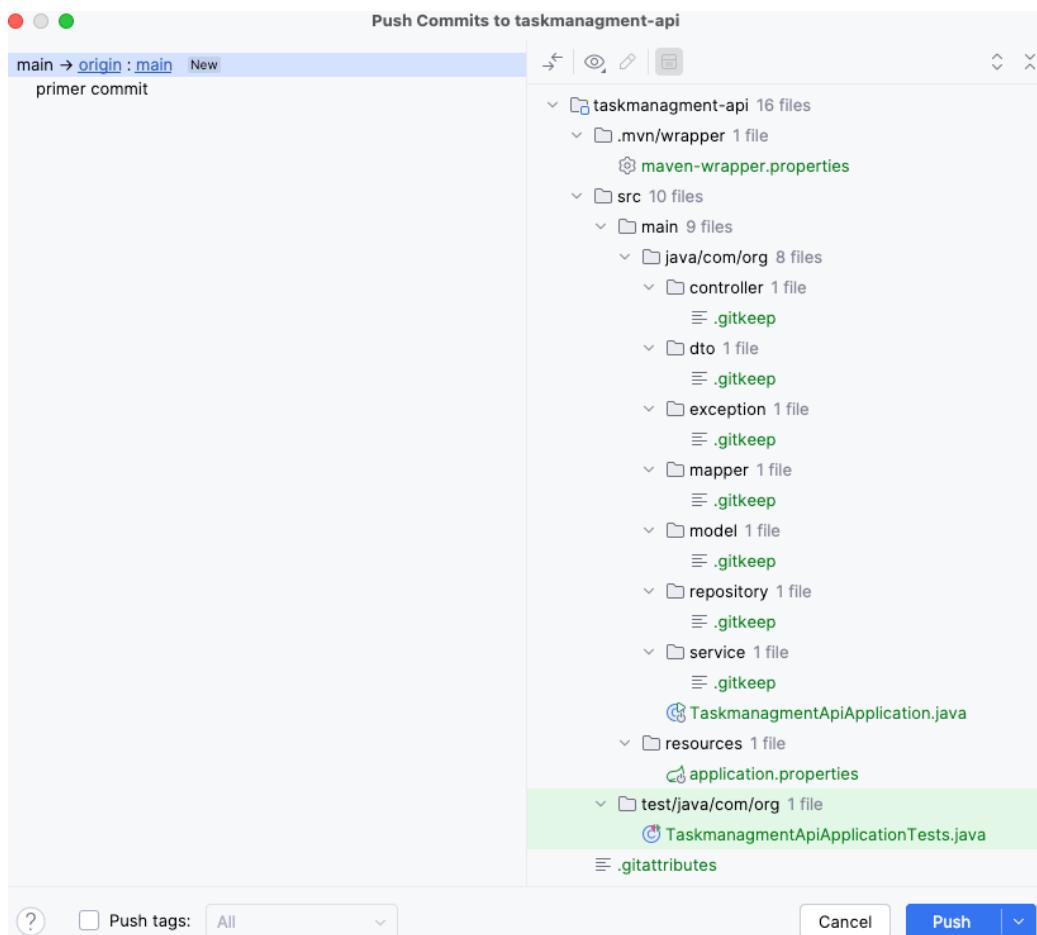
Subir Proyecto a GitHub desde IntelliJ IDEA

Una vez hayas creado el commit inicial, puedes subirlo a GitHub directamente desde IntelliJ IDEA:

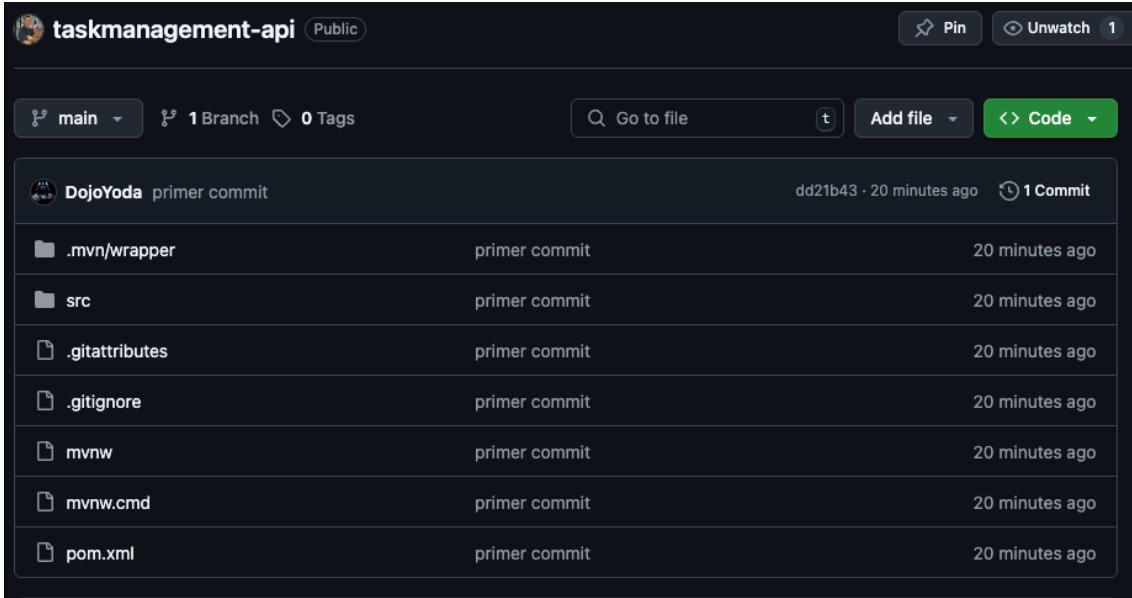
- Haz clic sobre el botón de rama (`main`, parte superior del IDE).
- Selecciona la opción **Push...** (como se muestra en la imagen):



- En la ventana emergente, asegúrate de que:
 - La rama local sea `main`
 - El destino remoto sea `origin/main`
 - Haz clic en **Push**.



Tu commit con la estructura inicial del proyecto será enviado a GitHub y aparecerá en tu repositorio remoto.



The screenshot shows a GitHub repository named "taskmanagement-api" which is public. The main branch is selected. There is 1 branch and 0 tags. A search bar and a code editor button are visible. A commit from "DojoYoda" titled "primer commit" is shown, dated "dd21b43 · 20 minutes ago". The commit includes files like ".mvn/wrapper", "src", ".gitattributes", ".gitignore", "mvnw", "mvnw.cmd", and "pom.xml", all with the same timestamp.

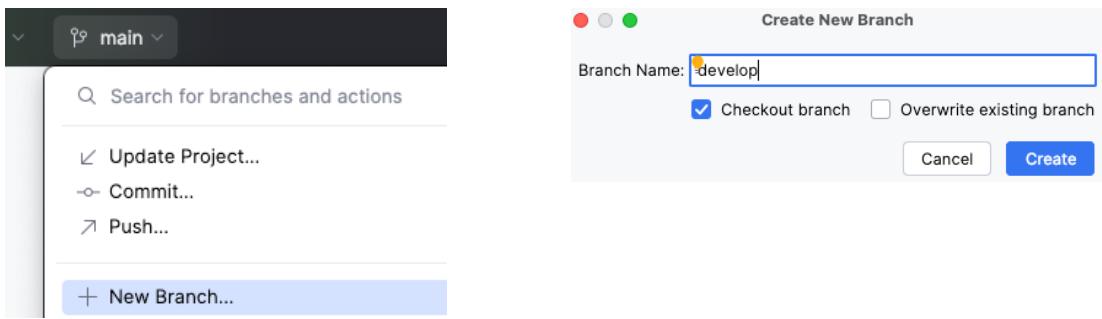
Trabajo Colaborativo con Git: Preparación del Flujo de Ramas

Antes de comenzar la implementación, es necesario establecer una estructura de ramas que facilite el desarrollo ordenado en equipo.

Crear la rama `develop`

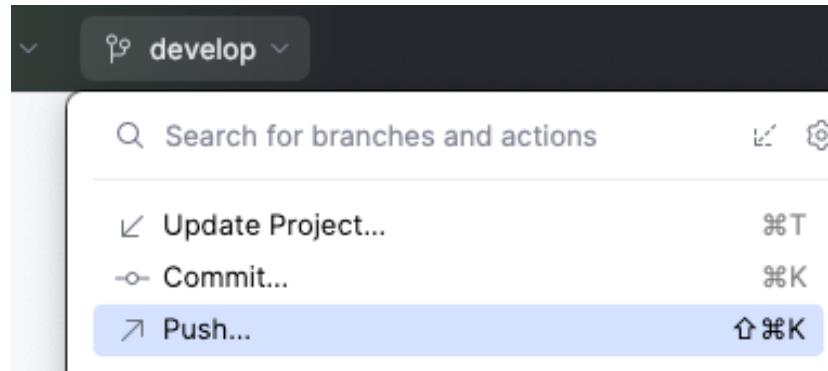
Este paso lo realiza el responsable del repositorio o el primer colaborador.

- `git checkout -b develop`
- `git push -u origin develop`



The left side of the image shows a terminal window with a dropdown menu set to "main". It has a search bar and options like "Update Project...", "Commit...", "Push...", and "+ New Branch...". The right side shows a "Create New Branch" dialog box with "Branch Name:" set to "develop", a checked "Checkout branch" option, and an unchecked "Overwrite existing branch" option. Below the dialog are "Cancel" and "Create" buttons.

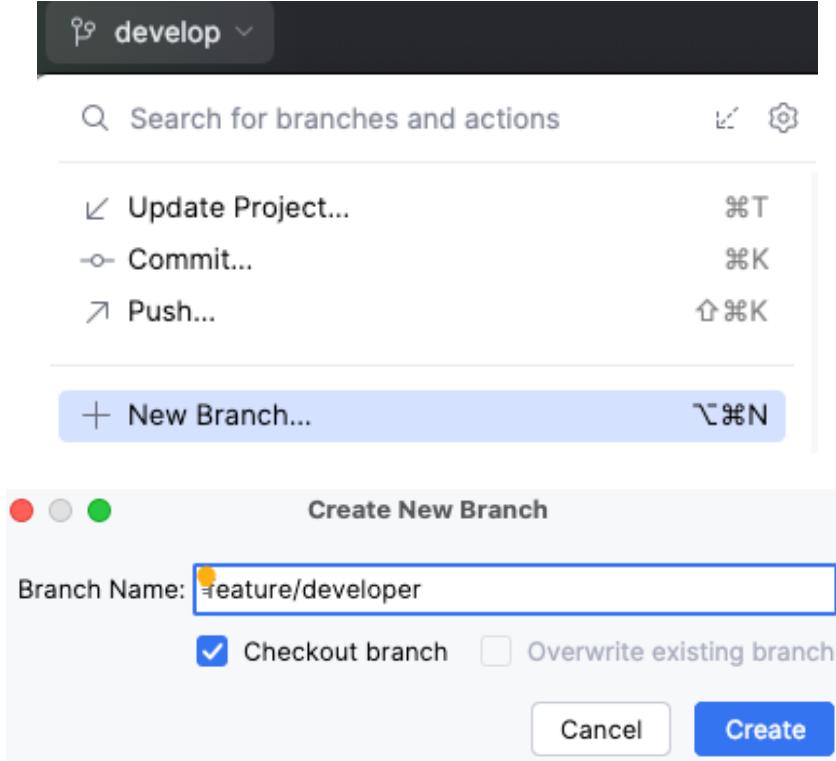
La rama `develop` será la base sobre la que se construirá todo el código del proyecto antes de integrarlo a la rama `main`. Subir la rama `develop` a GitHub



Crear una rama `feature` desde `develop`

Cada desarrollador debe crear una rama específica para la funcionalidad que va a implementar. Por ejemplo, para desarrollar el CRUD de `Developer`:

- `git checkout develop`
- `git pull origin develop`
- `git checkout -b feature/developer`



Desarrollo de la Funcionalidad Developer

Esta sección implementa el CRUD básico de developers, aplicando validaciones, separación en capas, uso de DTO y mapeo. El objetivo es permitir registrar, consultar, actualizar y eliminar developers desde la API.

Este desarrollo es realizado por **Hery**, quien trabaja en la rama `feature/developer` como parte del flujo colaborativo del proyecto.

Configuración de la Base de Datos (PostgreSQL)

Archivo: `src/main/resources/application.properties`

```
# Nombre del proyecto
spring.application.name=taskmanagement-api

# Puerto del servidor
server.port=8080

# Ruta base para los endpoints REST (por ejemplo: /api/v1/developers)
server.servlet.context-path=/api/v1

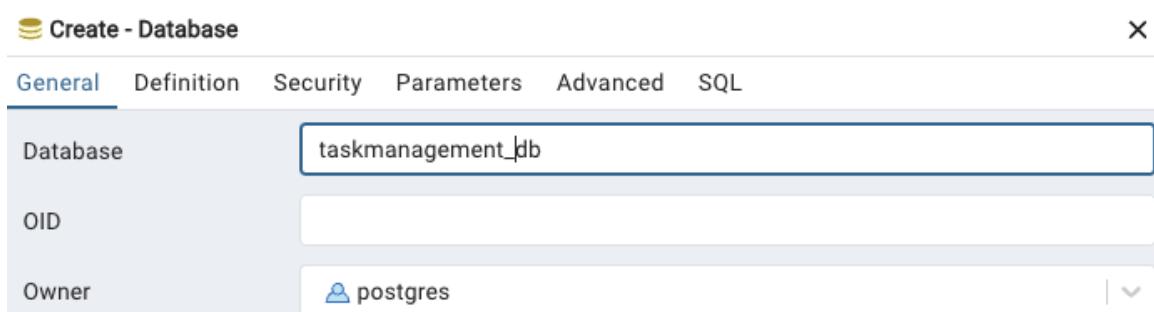
# Configuración de conexión a PostgreSQL
spring.datasource.url=jdbc:postgresql://localhost:5432/taskmanagement_db
spring.datasource.username=postgres
spring.datasource.password=adminadmin

# Dialecto y comportamiento de Hibernate (JPA)
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update          # Actualiza el esquema de la base de datos
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true # Formatea las consultas para mejor lectura

# Codificación por defecto de la conexión
spring.datasource.hikari.connection-init-sql=SET NAMES 'UTF8'
```

Crear la base de datos en PostgreSQL

Puedes crear la base de datos en tu cliente de PostgreSQL como pgAdmin



model/Developer.java

Entidad JPA que representa a un developer. Usa Lombok para reducir el código boilerplate.

```
package com.org.model;

import jakarta.persistence.*;
import lombok.*;

@Entity new *
@Table(name = "developers")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Developer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true, length = 100)
    private String name;
}
```

repository/DeveloperRepository.java

Interfaz JPA que proporciona acceso a datos para la entidad Developer.

```
package com.org.repository;

import com.org.model.Developer;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import java.util.Optional;
💡
public interface DeveloperRepository extends JpaRepository<Developer, Long> {
    Optional<Developer> findByName(String name); no usages new *
    boolean existsByName(String name); 2 usages new *
    Page<Developer> findAll(Pageable pageable); no usages new *
}
```

dto/request/DeveloperRequest.java

DTO de entrada definido como `record`. Es inmutable, conciso y apto para validaciones

```
package com.org.dto.request;

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;

public record DeveloperRequest( no usages new *
    @NotBlank(message = "El nombre es obligatorio") no usages
    @Size(max = 100, message = "El nombre no debe exceder los 100 caracteres")
    String name
) {}
```

dto/response/DeveloperResponse.java

DTO de salida definido como `record`. Expuesto como respuesta al cliente.

```
package com.org.dto.response;

public record DeveloperResponse( no usages new *
    Long id, no usages
    String name no usages
) {}
```

mapper/DeveloperMapper.java

Clase que convierte entre entidad y los DTOs `record`. El mapeo sigue siendo manual por claridad.

```
package com.org.mapper;

import com.org.dto.request.DeveloperRequest;
import com.org.dto.response.DeveloperResponse;
import com.org.model.Developer;
import org.springframework.stereotype.Component;

@Component no usages new *
public class DeveloperMapper {

    public Developer toEntity(DeveloperRequest request) { no usages new *
        Developer developer = new Developer();
        developer.setName(request.name());
        return developer;
    }

    public DeveloperResponse toResponse(Developer developer) { no usages new *
        return new DeveloperResponse(developer.getId(), developer.getName());
    }
}
```

exception/DuplicateResourceException.java

Excepción para cuando ya existe un recurso con un valor único.

```
package com.org.exception;

public class DuplicateResourceException extends RuntimeException {
    public DuplicateResourceException(String message) { no usages n
        super(message);
    }
}
```

exception/ResourceNotFoundException.java

Excepción para recursos no encontrados.

```
package com.org.exception;
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) { no usages n
        super(message);
    }
}
```

exception/BusinessRuleException.java

Excepción para violaciones de reglas de negocio.

```
package com.org.exception;
public class BusinessRuleException extends RuntimeException {
    public BusinessRuleException(String message) { no usages n
        super(message);
    }
}
```

exception/GlobalExceptionHandler.java

Manejador global de errores usando ProblemDetail de Spring Boot 3.

```
package com.org.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ProblemDetail;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.context.request.WebRequest;

@RestControllerAdvice new *
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class) new *
    public ProblemDetail handleNotFound(ResourceNotFoundException ex, WebRequest request) {
        ProblemDetail pd = ProblemDetail.forStatus(HttpStatus.NOT_FOUND);
        pd.setTitle("Recurso no encontrado");
        pd.setDetail(ex.getMessage());
        pd.setProperty("path", request.getDescription(includeClientInfo: false));
        return pd;
    }

    @ExceptionHandler(DuplicateResourceException.class) new *
    public ProblemDetail handleDuplicate(DuplicateResourceException ex, WebRequest request) {
        ProblemDetail pd = ProblemDetail.forStatus(HttpStatus.CONFLICT);
        pd.setTitle("Recurso duplicado");
        pd.setDetail(ex.getMessage());
        pd.setProperty("path", request.getDescription(includeClientInfo: false));
        return pd;
    }

    @ExceptionHandler(Exception.class) new *
    public ProblemDetail handleGeneric(Exception ex, WebRequest request) {
        ProblemDetail pd = ProblemDetail.forStatus(HttpStatus.INTERNAL_SERVER_ERROR);
        pd.setTitle("Error interno");
        pd.setDetail(ex.getMessage());
        pd.setProperty("path", request.getDescription(includeClientInfo: false));
        return pd;
    }
}
```

service/DeveloperService.java

Servicio encargado de gestionar operaciones sobre developers. Aplica reglas de negocio como la validación de nombre único y el control de existencia. Utiliza `@RequiredArgsConstructor` para la inyección automática de dependencias, y `@Transactional` para garantizar la integridad de las operaciones sobre la base de datos.

```
package com.org.service;

import com.org.dto.request.DeveloperRequest;
import com.org.dto.response.DeveloperResponse;
import com.org.exception.DuplicateResourceException;
import com.org.exception.ResourceNotFoundException;
import com.org.mapper.DeveloperMapper;
import com.org.model.Developer;
import com.org.repository.DeveloperRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service 2 usages new *
@RequiredArgsConstructor
public class DeveloperService {

    private final DeveloperRepository developerRepository;
    private final DeveloperMapper developerMapper;

    @Transactional 1 usage new *
    public DeveloperResponse create(DeveloperRequest request) {
        if (developerRepository.existsByName(request.name())) {
            throw new DuplicateResourceException("Ya existe un developer con ese nombre");
        }
        Developer saved = developerRepository.save(developerMapper.toEntity(request));
        return developerMapper.toResponse(saved);
    }

    @Transactional(readOnly = true) 1 usage new *
    public List<DeveloperResponse> findAll() {
        return developerRepository.findAll() List<Developer>
            .stream() Stream<Developer>
            .map(developerMapper::toResponse) Stream<DeveloperResponse>
            .toList();
    }

    @Transactional(readOnly = true) no usages new *
    public Page<DeveloperResponse> findPaginated(int page, int size) {
        Pageable pageable = PageRequest.of(page, size);
        return developerRepository.findAll(pageable)
            .map(developerMapper::toResponse);
    }

    @Transactional(readOnly = true) 1 usage new *
    public DeveloperResponse findById(Long id) {
        Developer dev = developerRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Developer no encontrado"));
        return developerMapper.toResponse(dev);
    }
}
```

```

@Transactional 1 usage new *
public DeveloperResponse update(Long id, DeveloperRequest request) {
    Developer dev = developerRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Developer no encontrado"));

    if (!dev.getName().equals(request.name()) && developerRepository.existsByName(request.name())) {
        throw new DuplicateResourceException("Ya existe otro developer con ese nombre");
    }

    dev.setName(request.name());
    return developerMapper.toResponse(developerRepository.save(dev));
}

@Transactional 1 usage new *
public void delete(Long id) {
    Developer dev = developerRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Developer no encontrado"));

    developerRepository.delete(dev);
}
}

```

controller/DeveloperController.java

Controlador REST que define los endpoints HTTP del recurso Developer.

Gestiona operaciones CRUD sobre la ruta base /developers, mediante los verbos POST, GET, PUT y DELETE. Utiliza @RequiredArgsConstructor para la inyección automática de dependencias, ResponseEntity para mayor control en las respuestas, y validaciones con @Valid en el cuerpo de las peticiones.

@RequestBody

Indica que el parámetro del método debe ser leído desde el cuerpo de la petición HTTP y convertido automáticamente (deserializado) a un objeto Java (en este caso, un DeveloperRequest).

Es necesaria para manejar correctamente peticiones POST y PUT con contenido JSON.

@Valid

Se utiliza junto con @RequestBody para activar las validaciones declaradas en el DTO (por ejemplo: @NotBlank, @Size, etc.).

Si el cuerpo recibido no cumple con las reglas, Spring Boot lanza automáticamente un error 400 (Bad Request) con detalle del fallo.

ResponseType<T>

Clase envoltorio que representa toda la respuesta HTTP, incluyendo:

- el cuerpo (T),
- el código de estado (por ejemplo: 200, 201, 204),
- y opcionalmente, headers.

```
package com.org.controller;

import com.org.dto.request.DeveloperRequest;
import com.org.dto.response.DeveloperResponse;
import com.org.service.DeveloperService;
import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import org.springframework.data.domain.Page;
import org.springframework.web.bind.annotation.RequestParam;

import java.util.List;

@RestController new *
@RequestMapping("/developers")
@RequiredArgsConstructor
public class DeveloperController {

    private final DeveloperService developerService;

    @PostMapping new *
    public ResponseEntity<DeveloperResponse> create(@Valid @RequestBody DeveloperRequest request) {
        DeveloperResponse response = developerService.create(request);
        return ResponseEntity.status(201).body(response);
    }

    @GetMapping(new *)
    public ResponseEntity<List<DeveloperResponse>> getAll() {
        return ResponseEntity.ok(developerService.findAll());
    }

    @GetMapping("/paginated") new *
    public ResponseEntity<Page<DeveloperResponse>> getPaginated(Pageable pageable) {
        return ResponseEntity.ok(developerService.findAll(pageable));
    }

    @GetMapping("/{id}") new *
    public ResponseEntity<DeveloperResponse> getById(@PathVariable Long id) {
        return ResponseEntity.ok(developerService.findById(id));
    }

    @PutMapping("/{id}") new *
    public ResponseEntity<DeveloperResponse> update(@PathVariable Long id,
                                                    @Valid @RequestBody DeveloperRequest request) {
        return ResponseEntity.ok(developerService.update(id, request));
    }

    @DeleteMapping("/{id}") new *
    public ResponseEntity<Void> delete(@PathVariable Long id) {
        developerService.delete(id);
        return ResponseEntity.noContent().build();
    }
}
```

Pruebas Funcionales con Postman

En esta sección se realizarán pruebas sobre los endpoints del recurso Developer usando Postman. Estas pruebas permiten verificar el comportamiento de la API en operaciones básicas como crear, listar, buscar, actualizar y eliminar developers.

Se utilizará una colección con una variable (`base_url`) para facilitar la gestión del entorno.

Crear la colección

1. Abrir Postman y seleccionar "Collections" > "New Collection".
2. Asignar el nombre: taskmanagement-api
3. Guardar.

The screenshot shows the 'taskmanagement-api' collection in Postman. At the top, there's a navigation bar with tabs for Overview, Authorization, Scripts, Variables, and Runs. The 'Variables' tab is currently selected. Below the tabs, there's a section for defining variables. A variable named 'base_url' is listed with its initial value as 'http://localhost:8080/api/v1' and its current value also as 'http://localhost:8080/api/v1'. There's also a note stating 'These variables are specific to this collection and its requests. Learn more about [collection variables](#)'. At the bottom, there's a search bar labeled 'Filter variables' and a button to 'Add new variable'.

Definir variable `base_url` en la colección

1. Dentro de la colección Developer API, ir a la pestaña "Variables".
2. Agregar:
 - o Variable: `base_url`
 - o Initial Value: `http://localhost:8080/api/v1`
 - o Current Value: `http://localhost:8080/api/v1`
3. Guardar cambios.

The screenshot shows the 'Variables' tab in the collection settings for 'taskmanagement-api'. A single variable, 'base_url', is listed in the table. It has an initial value of 'http://localhost:8080/api/v1' and a current value of 'http://localhost:8080/api/v1'. There's a note at the top stating 'These variables are specific to this collection and its requests. Learn more about [collection variables](#)'. Below the table, there's a search bar labeled 'Filter variables' and a button to 'Add new variable'.

Crear carpeta Developer dentro de la colección

1. Dentro de la colección Developer API, hacer clic en "Add Folder".
2. Asignar el nombre: Developer.
3. Guardar el folder (carpeta).



Agregar endpoints a la carpeta Developer

Crear Developer

- Método: POST
- URL: {{base_url}} /developers
- Body (raw / JSON):

POST Create Developer +

taskmanagement-api / developer / Create Developer

POST {{base_url}} /developers

Params Authorization Headers (8) Body Scripts Settings

Body raw binary GraphQL JSON

```
1 {
2   "name": "Carlos Romero"
3 }
4 
```

Body Cookies Headers (5) Test Results

{} JSON Preview Visualize

```
1 {
2   "id": 1,
3   "name": "Carlos Romero"
4 }
```

POST Create Developer +

taskmanagement-api / developer / Create Developer

POST {{base_url}} /developers

Params Authorization Headers (8) Body Scripts Settings

Body raw binary GraphQL JSON

```
1 {
2   "name": "Carlos Romero"
3 }
4 
```

Body Cookies Headers (5) Test Results

{} JSON Preview Visualize

```
1 {
2   "type": "about:blank",
3   "title": "Recurso duplicado",
4   "status": 409,
5   "detail": "Ya existe un developer con ese nombre",
6   "instance": "/api/v1/developers",
7   "path": "uri=/api/v1/developers"
8 }
```

Listar Developer (Todos)

- Método: GET
- URL: {{base_url}} /developers

The screenshot shows the Postman interface for a GET request to 'Listar Developers (Todos)'. The URL is {{base_url}} /developer / Listar Developers (Todos). The method is set to GET. In the Headers section, there are 6 items. The Body tab is selected, showing a JSON response with two developer objects:

```
1 [  
2 {  
3   "id": 2,  
4   "name": "Henry Mendoza Puerta"  
5 },  
6 {  
7   "id": 3,  
8   "name": "Carlos Romero"  
9 }  
10 ]
```

Listar Developers (Paginado)

- Método: GET
- URL: {{base_url}} /developers/paginated?page=0&size=5&sort=name,asc

The screenshot shows the Postman interface for a GET request to 'Listar Developers (Paginado)'. The URL is {{base_url}} /developer / Listar Developers (Paginado). The method is set to GET. In the Headers section, there are 6 items. The Query Params section has four checked parameters: page, size, sort, and content. The Body tab is selected, showing a JSON response with a single key 'content' containing two developer objects:

The screenshot shows the Postman interface for a GET request to 'Listar Developers (Paginado)'. The URL is {{base_url}} /developer / Listar Developers (Paginado). The method is set to GET. In the Headers section, there are 6 items. The Query Params section has four checked parameters: page, size, sort, and content. The Body tab is selected, showing a JSON response with a single key 'content' containing two developer objects:

```
1 {  
2   "content": [  
3     {  
4       "id": 3,  
5       "name": "Carlos Romero"  
6     },  
7     {  
8       "id": 2,  
9       "name": "Henry Mendoza Puerta"  
10    }  
11  ],
```

Obtener Developer por ID

- Método: GET
- URL: {{base_url}} /developers/1

The screenshot shows the Postman interface for a GET request. The URL is set to {{base_url}} /developers/1. The Headers tab shows 'Content-Type: application/json'. The Body tab is empty. The response body is displayed as:

```
1 {  
2   "id": 1,  
3   "name": "Carlos Romero"  
4 }
```

Actualizar Developer

- Método: PUT
- URL: {{base_url}} /developers/1
- Body (raw / JSON):

The screenshot shows the Postman interface for a PUT request. The URL is set to {{base_url}} /developers/1. The Headers tab shows 'Content-Type: application/json'. The Body tab is selected and contains the raw JSON body:

```
1 {  
2   "name": "Carlos R. Actualizado"  
3 }  
4
```

The response body is displayed as:

```
1 {  
2   "id": 1,  
3   "name": "Carlos R. Actualizado"  
4 }
```

Eliminar Developer

- Método: DELETE
- URL: {{base_url}}/developers/1

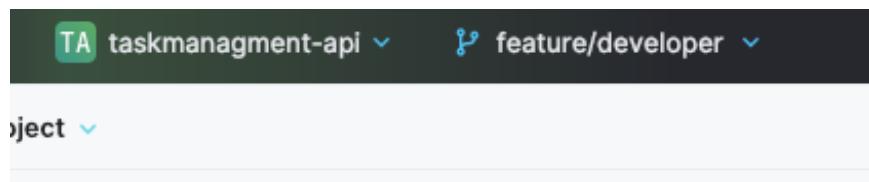
The screenshot shows the Postman application interface. At the top, there's a header bar with 'DEL Eliminar Developer' and a '+' button. Below it, a navigation bar shows 'taskmanagement-api / developer / Eliminar Developer'. The main area has a 'DELETE' button and a URL field containing '{{base_url}}/developers/1'. Below these are tabs for 'Params', 'Authorization', 'Headers (6)', 'Body', 'Scripts', and 'Settings'. Under 'Params', there's a table with one row: 'Key' and 'Value'. In the 'Body' section, there are tabs for 'Body', 'Cookies', 'Headers (3)', and 'Test Results'. The 'Body' tab is selected, showing a JSON input field with the value '1'. Below the input field are 'Preview' and 'Visualize' buttons.

Publicación de la Rama Feature y Solicitud de Pull Request

Una vez finalizado el desarrollo de una funcionalidad en una rama `feature`, el siguiente paso es publicar esa rama en el repositorio remoto y solicitar su integración en la rama `develop`. Este flujo asegura que los cambios pasen por revisión antes de incorporarse al desarrollo principal del proyecto.

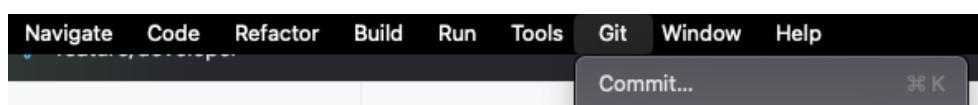
Confirmar la rama activa

Asegúrate de estar trabajando en la rama `feature/developer`, visible en la parte superior derecha de IntelliJ IDEA.

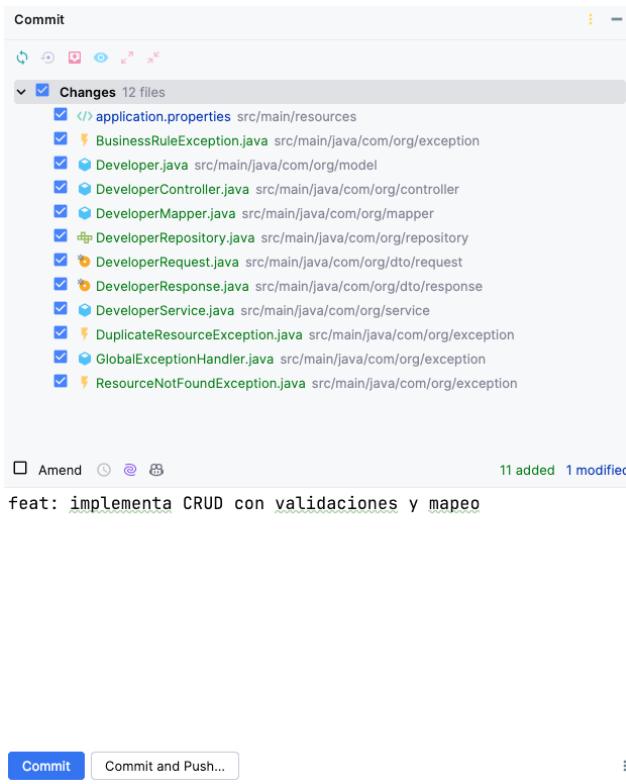


Hacer commit de los cambios

- Ve a: Git > Commit... o usa Ctrl + K / Cmd + K.
- Revisa los archivos modificados.
- Escribe un mensaje de commit siguiendo convenciones semánticas, por ejemplo:

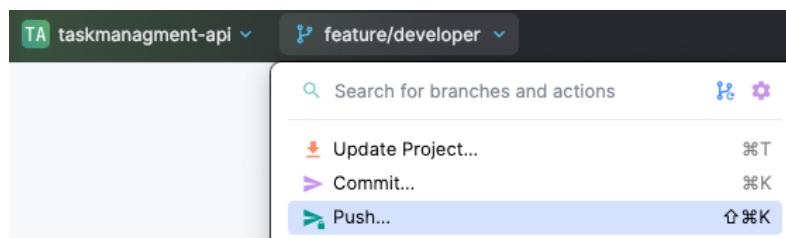


- Haz clic en Commit o Commit and Push.



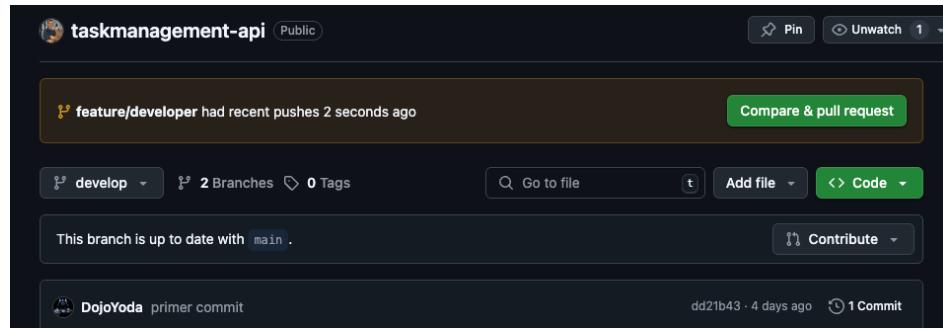
Hacer Push de la rama

- Si no lo hiciste al hacer commit, ve a Git > Push... o usa Ctrl + Shift + K / Cmd + Shift + K.
- Verifica que se está empujando la rama feature/developer al repositorio remoto.
- Haz clic en Push.



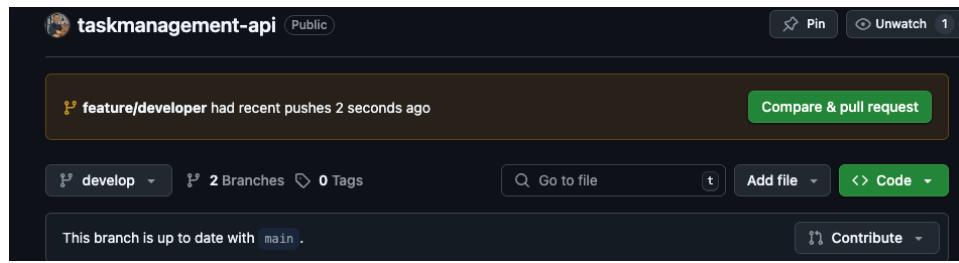
Crear el Pull Request en GitHub

- Abre tu repositorio en GitHub.
- Verás una sugerencia: Compare & pull request.
- Haz clic en ella.



Crear el Pull Request en GitHub

- Abre tu repositorio en GitHub.
- Verás una sugerencia: Compare & pull request.
- Haz clic en ella.



Completar el Pull Request

- Verifica:
 - base: develop
 - compare: feature/developer

A screenshot of the GitHub 'Comparing changes' interface. It shows a comparison between 'base: develop' and 'compare: feature/developer'. A green checkmark indicates they are 'Able to merge'. The interface includes fields for 'Add a title' (containing 'feat: implementa CRUD con validaciones y manejo de errores') and 'Add a description' (containing a detailed list of features and a merge note). A toolbar at the bottom allows for writing or previewing the description. A 'Create pull request' button is at the bottom right.

- Haz clic en Create pull request.

Revisión del Pull Request

- Un miembro del equipo (o tú mismo, si trabajas solo) debe revisar:
 - La lógica del código
 - La estructura del proyecto
 - Que se cumplan las convenciones de commits y PR
 - Que no haya conflictos con develop
- GitHub permite dejar comentarios o aprobar el PR.

Correcciones si son necesarias

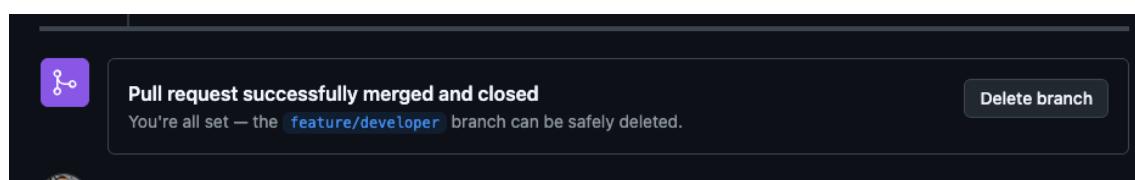
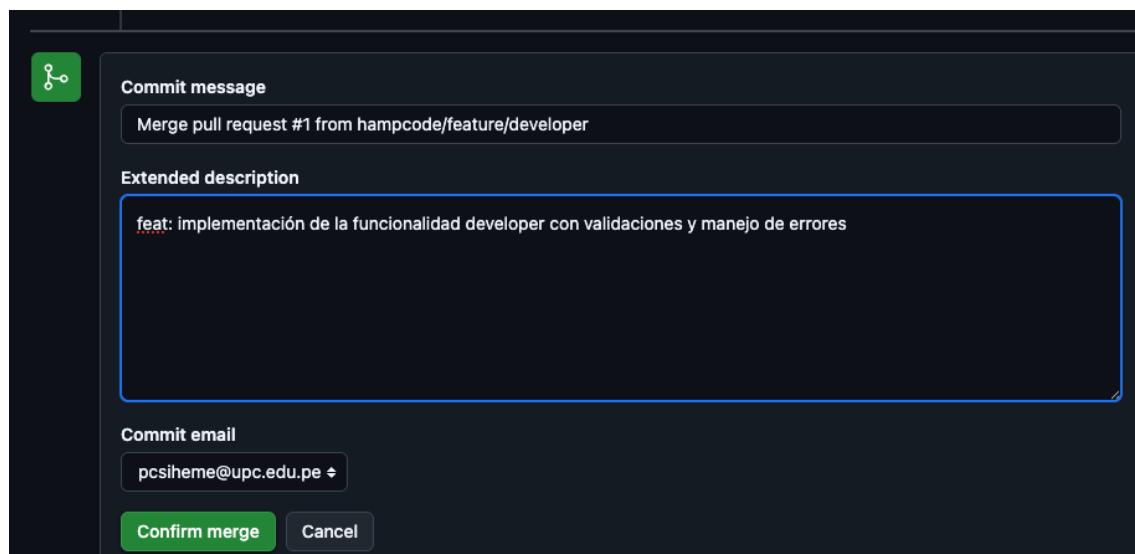
- Si el revisor deja observaciones:
 - Realiza los cambios en la misma rama feature/developer
 - Haz commit y push, los cambios se reflejan automáticamente en el PR

Aprobación del PR

- Una vez aprobado (por otro o por ti, si es un proyecto individual), puedes continuar con el merge.

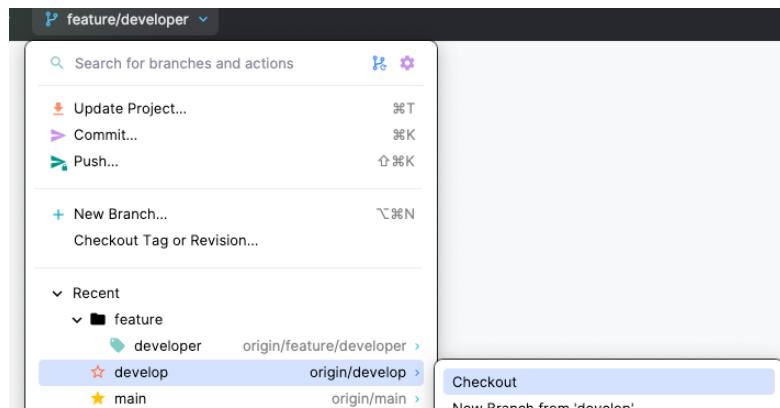
Merge del Pull Request a develop

- Haz clic en el botón "**Merge pull request**"
- Confirma con "**Confirm merge**"
- Opcionalmente, elimina la rama con "**Delete branch**" si ya no la necesitas.

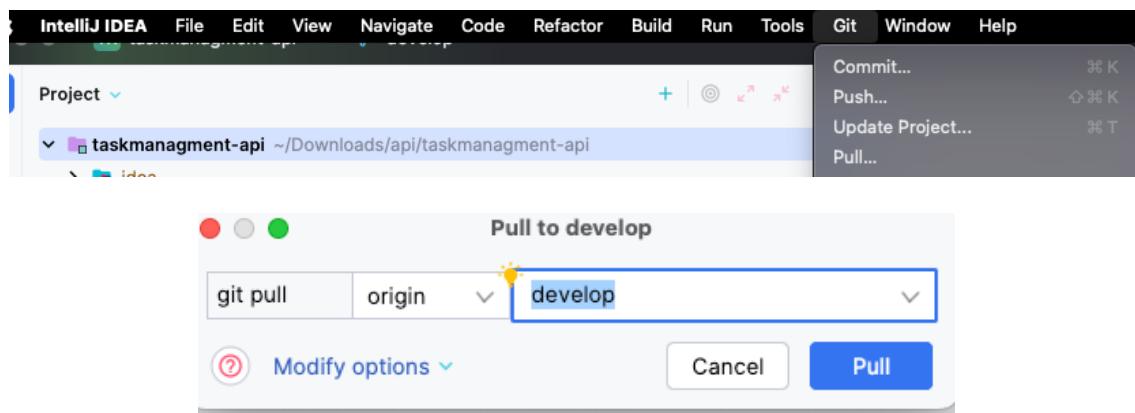


Actualizar localmente la rama develop

- Cambia a la rama develop en IntelliJ (Git > Branches > develop)

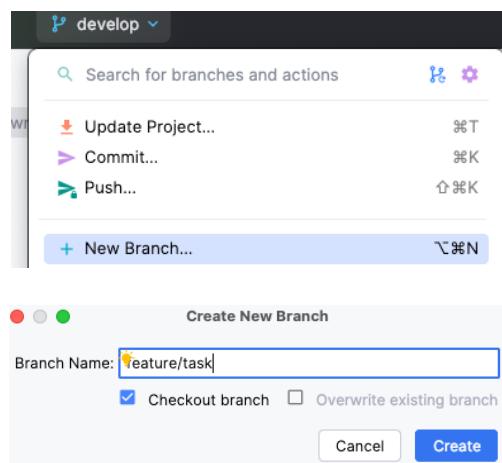


- Luego sincroniza con el remoto:

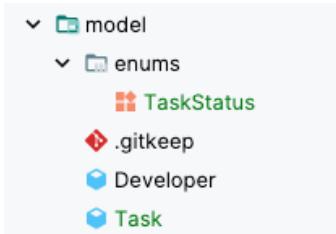


Desarrollo de la Funcionalidad Task

Esta sección estará dedicada a implementar la gestión de tareas (Task) dentro del sistema. Al igual que con Developer, se aplicará separación en capas, validaciones y control de estado, respetando reglas de negocio específicas como el límite de tareas activas por developer y transiciones válidas entre estados.



Estructura de paquetes del model



Enum TaskStatus

Define los posibles estados de una tarea (PENDING, IN_PROGRESS, COMPLETED, CANCELLED). Se usa para controlar el ciclo de vida de una tarea y validar las transiciones permitidas.

```
package com.org.model.enums;

public enum TaskStatus { no usages new *
    PENDING, no usages
    IN_PROGRESS, no usages
    COMPLETED, no usages
    CANCELLED no usages
}
```

Modelo Task

La clase Task representa una entidad del dominio encargada de modelar las tareas del sistema, con campos como título, descripción, estado, fechas y relación con un developer.

Uso de @Builder

Se usa @Builder para construir objetos Task de forma ordenada y legible. Su estructura es más compleja que Developer (que solo tiene id y name), ya que incluye estado, fechas y relaciones. Este patrón es útil cuando se desea construir objetos con múltiples campos sin necesidad de usar constructores con muchos parámetros o múltiples setters encadenados.

```
package com.org.model;

import com.org.model.enums.TaskStatus;
import jakarta.persistence.*;
import lombok.*;

import java.time.LocalDate;

@Entity new *
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Table(name = "tasks")
public class Task {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```

    @Column(nullable = false)
    private String title;

    @Column(nullable = false)
    private String description;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    private TaskStatus status;

    @ManyToOne(optional = false)
    @JoinColumn(name = "developer_id")
    private Developer developer;

    private LocalDate startDate;
    private LocalDate endDate;
}

```

Repositorio TaskRepository

Interface que extiende JpaRepository para acceder y consultar tareas. Incluye métodos personalizados para validar títulos únicos, contar tareas activas por developer, filtrar por fechas y paginar resultados.

```

package com.org.repository;

import com.org.model.Task;
import com.org.model.enums.TaskStatus;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.*;

import java.time.LocalDate;
import java.util.List;

public interface TaskRepository extends JpaRepository<Task, Long> { no usages new *

    // Verifica si ya existe una tarea con el mismo título
    boolean existsByTitle(String title); no usages new *

    // Cuenta cuántas tareas activas tiene un developer (por estados)
    @Query("SELECT COUNT(t) FROM Task t WHERE t.developer.id = :developerId AND t.status IN :statuses") no usages new *
    int countActiveTasksByDeveloperId(Long developerId, List<TaskStatus> statuses);

    // Lista todas las tareas con paginación
    Page<Task> findAll(Pageable pageable); new *

    // Lista tareas dentro de un rango de fechas con paginación
    @Query("SELECT t FROM Task t WHERE t.startDate >= :start AND t.endDate <= :end") no usages new *
    Page<Task> findTasksByDateRange(LocalDate start, LocalDate end, Pageable pageable);

    // Lista tareas de un developer según una lista de estados
    @Query("SELECT t FROM Task t WHERE t.developer.id = :developerId AND t.status IN :statuses") no usages new *
    List<Task> findTasksByDeveloperIdAndStatusIn(Long developerId, List<TaskStatus> statuses);
}

```

DTO TaskRequest

Este DTO se utiliza al **crear o actualizar** una tarea.

```
package com.org.dto.request;

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;

import java.time.LocalDate;

public record TaskRequest( 8 usages new *
    @NotBlank String title, 5 usages
    @NotBlank String description, 2 usages
    @NotNull Long developerId, 2 usages
    LocalDate startDate, 1 usage
    LocalDate endDate 1 usage
) {}
```

DTO TaskResponse

Este DTO se usa para retornar los datos completos de una tarea, incluyendo el estado y las fechas.

```
package com.org.dto.response;

import java.time.LocalDate;

public record TaskResponse( 3 usages new * 1 related problem
    Long id, no usages
    String title, no usages
    String description, no usages
    String status, no usages
    String developerName, no usages
    LocalDate startDate, no usages
    LocalDate endDate no usages
) {}
```

Mapper TaskMapper

Encapsula la conversión entre la entidad Task y sus DTOs (TaskRequest, TaskResponse). Utiliza el patrón Builder para construir objetos Task de forma clara y controlada, mejorando la legibilidad y evitando el uso de múltiples setters.

```
package com.org.mapper;

import com.org.dto.request.TaskRequest;
import com.org.dto.response.TaskResponse;
import com.org.model.Developer;
import com.org.model.Task;
import com.org.model.enums.TaskStatus;
import org.springframework.stereotype.Component;

@Component 2 usages new *
public class TaskMapper {

    public Task toEntity(TaskRequest request, Developer developer) { 1 usage new *
        return Task.builder()
            .title(request.title())
            .description(request.description())
            .developer(developer)
            .status(TaskStatus.PENDING)
            .startDate(request.startDate())
            .endDate(request.endDate())
            .build();
    }
}
```

```

    public TaskResponse toResponse(Task task) { no usages new *
        return new TaskResponse(
            task.getId(),
            task.getTitle(),
            task.getDescription(),
            task.getStatus().name(),
            task.getDeveloper().getName(),
            task.getStartDate(),
            task.getEndDate()
        );
    }
}

```

Excepciones

Con las excepciones que ya tienes:

- BusinessRuleException
- DuplicateResourceException
- ResourceNotFoundException
- GlobalExceptionHandler

estás cubriendo adecuadamente los escenarios principales.

ServicioTaskService

Servicio encargado de implementar la lógica de negocio para la gestión de tareas (`Task`). Utiliza `TaskRepository`, `DeveloperRepository` y `TaskMapper` para registrar, consultar, actualizar y cambiar el estado de las tareas.

Reglas de negocio y validaciones aplicadas:

1. Título único: no se puede registrar una tarea con un título ya existente.
2. Asignación controlada: solo se puede asignar una tarea a un developer si está en estado `PENDING`.
3. Límite de tareas activas: un developer no puede tener más de N tareas activas (`PENDING`, `IN_PROGRESS`).
4. Transición válida de estado:
 - o Solo se puede pasar a `IN_PROGRESS` desde `PENDING`.
 - o Solo se puede pasar a `COMPLETED` desde `IN_PROGRESS`.
5. Control de fechas:
 - o `startDate` se asigna al cambiar a `IN_PROGRESS`.
 - o `endDate` se asigna al cambiar a `COMPLETED` o `CANCELLED`.
6. Validación de existencia: se verifica que la tarea y el developer existan antes de operar.
7. Validación de rango de fechas: la fecha de inicio no puede ser posterior a la de fin (en búsqueda por rango).
8. Consulta de tareas activas: se permite listar tareas activas (`PENDING`, `IN_PROGRESS`) por developer para monitoreo.

@Transactional

es una anotación de Spring que indica que todos los métodos públicos de esta clase se ejecutarán dentro de una transacción. Si ocurre una excepción en cualquiera de ellos, la transacción se revertirá (rollback automático).

Se aplica `@Transactional(readOnly = true)` solo en métodos como `findAll()` y `findById()` porque **solo consultan datos**, no los modifican.

Esto mejora el rendimiento, evita operaciones innecesarias y optimiza el acceso a la base de datos.

Los demás métodos (`create, update, delete, changeStatus`) sí modifican datos, por lo que **no deben marcarse como solo lectura**.

```
package com.org.service;

import com.org.dto.request.TaskRequest;
import com.org.dto.response.TaskResponse;
import com.org.exception.BusinessRuleException;
import com.org.exception.DuplicateResourceException;
import com.org.exception.ResourceNotFoundException;
import com.org.mapper.TaskMapper;
import com.org.model.Developer;
import com.org.model.Task;
import com.org.model.enums.TaskStatus;
import com.org.repository.DeveloperRepository;
import com.org.repository.TaskRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.time.LocalDate;
import java.util.List;

@Service no usages new *
@RequiredArgsConstructor
public class TaskService {

    private final TaskRepository taskRepository;
    private final DeveloperRepository developerRepository;
    private final TaskMapper taskMapper;

    private static final int MAX_ACTIVE_TASKS = 5; 2 usages
```

```

@Transactional no usages new *
public TaskResponse create(TaskRequest request) {
    // Regla 1: Título único
    if (taskRepository.existsByTitle(request.title())) {
        throw new DuplicateResourceException("Ya existe una tarea con ese título");
    }

    // Regla 6: Validación de existencia del developer
    Developer developer = developerRepository.findById(request.developerId())
        .orElseThrow(() -> new ResourceNotFoundException("Developer no encontrado"));

    // Regla 3: Límite de tareas activas por developer
    int activeCount = taskRepository.countActiveTasksByDeveloperId(
        developer.getId(), List.of(TaskStatus.PENDING, TaskStatus.IN_PROGRESS));
}

if (activeCount >= MAX_ACTIVE_TASKS) {
    throw new BusinessRuleException("El developer ya tiene el número máximo de tareas activas");
}

// Regla 2: Asignación controlada (solo si está en PENDING, se valida indirectamente en lógica)
Task task = taskMapper.toEntity(request, developer);
return taskMapper.toResponse(taskRepository.save(task));
}

@Transactional(readOnly = true) no usages new *
public Page<TaskResponse> findAll(Pageable pageable) {
    return taskRepository.findAll(pageable).map(taskMapper::toResponse);
}

@Transactional(readOnly = true) no usages new *
public TaskResponse findById(Long id) {
    // Regla 6: Validación de existencia
    Task task = taskRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Tarea no encontrada"));
    return taskMapper.toResponse(task);
}

@Transactional no usages new *
public TaskResponse update(Long id, TaskRequest request) {
    // Regla 6: Validación de existencia
    Task task = taskRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Tarea no encontrada"));

    // Regla 1: Título único
    if (!task.getTitle().equals(request.title()) && taskRepository.existsByTitle(request.title())) {
        throw new DuplicateResourceException("Ya existe una tarea con ese título");
    }

    // Regla 2: Solo actualizar tareas en estado PENDING
    if (task.getStatus() != TaskStatus.PENDING) {
        throw new BusinessRuleException("Solo se puede reasignar una tarea en estado PENDING");
    }

    // Regla 6: Validación de existencia del nuevo developer
    Developer newDeveloper = developerRepository.findById(request.developerId())
        .orElseThrow(() -> new ResourceNotFoundException("Developer no encontrado"));
}

```

```

// Regla 3: Límite de tareas activas (si cambia de developer)
if (!task.getDeveloper().getId().equals(newDeveloper.getId())) {
    int activeCount = taskRepository.countActiveTasksByDeveloperId(
        newDeveloper.getId(), List.of(TaskStatus.PENDING, TaskStatus.IN_PROGRESS));
}

if (activeCount >= MAX_ACTIVE_TASKS) {
    throw new BusinessRuleException("El nuevo developer tiene tareas activas al límite");
}

task.setDeveloper(newDeveloper);
}

task.setTitle(request.title());
task.setDescription(request.description());
return taskMapper.toResponse(taskRepository.save(task));
}

@Transactional no usages new *
public void delete(Long id) {
    // Regla 6: Validación de existencia
    Task task = taskRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Tarea no encontrada"));
    taskRepository.delete(task);
}

@Transactional 1 usage new *
public TaskResponse changeStatus(Long id, TaskStatus newStatus) {
    // Regla 6: Validación de existencia
    Task task = taskRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Tarea no encontrada"));

    TaskStatus current = task.getStatus();

    // Regla 4: Transición válida de estado
    if (newStatus == TaskStatus.COMPLETED & current != TaskStatus.IN_PROGRESS) {
        throw new BusinessRuleException("Solo se puede completar una tarea que estuvo en progreso");
    }

    if (newStatus == TaskStatus.IN_PROGRESS && current != TaskStatus.PENDING) {
        throw new BusinessRuleException("Solo se puede iniciar una tarea que esté pendiente");
    }

    task.setStatus(newStatus);
    return taskMapper.toResponse(taskRepository.save(task));
}

```

```

@Transactional(readOnly = true) no usages new *
public Page<TaskResponse> findByDateRange(LocalDate start, LocalDate end, Pageable pageable) {
    // Regla 7: Validar que la fecha de inicio no sea mayor que la fecha fin
    if (start.isAfter(end)) {
        throw new BusinessRuleException("La fecha de inicio no puede ser posterior a la fecha de fin");
    }

    return taskRepository.findTasksByDateRange(start, end, pageable)
        .map(taskMapper::toResponse);
}

@Transactional(readOnly = true) no usages new *
public List<TaskResponse> findActiveTasksByDeveloper(Long developerId) {
    // Regla 8: Consulta de tareas activas por developer
    List<Task> tasks = taskRepository.findTasksByDeveloperIdAndStatusIn(
        developerId, List.of(TaskStatus.PENDING, TaskStatus.IN_PROGRESS)
    );
    return tasks.stream().map(taskMapper::toResponse).toList();
}

```

Controller TaskController

Controlador REST responsable de gestionar el recurso Task. Expone endpoints para crear, consultar, actualizar, eliminar y cambiar el estado de tareas. También permite filtrar tareas por rango de fechas y listar tareas activas asignadas a un developer. Interactúa con TaskService, utiliza paginación (Pageable) y maneja validaciones mediante @Valid.

```

package com.org.controller;

import com.org.dto.request.TaskRequest;
import com.org.dto.response.TaskResponse;
import com.org.model.enums.TaskStatus;
import com.org.service.TaskService;
import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.time.LocalDate;
import java.util.List;

@RestController new *
@RequestMapping("/tasks")
@RequiredArgsConstructor
public class TaskController {

    private final TaskService taskService;

    // Crear una nueva tarea
    @PostMapping new *
    public ResponseEntity<TaskResponse> create(@Valid @RequestBody TaskRequest request) {
        return ResponseEntity.ok(taskService.create(request));
    }
}

```

```

// Obtener todas las tareas con paginación
@GetMapping("new")
public ResponseEntity<Page<TaskResponse>> findAll(Pageable pageable) {
    return ResponseEntity.ok(taskService.findAll(pageable));
}

// Obtener una tarea por su ID
@GetMapping("/{id}")
public ResponseEntity<TaskResponse> findById(@PathVariable Long id) {
    return ResponseEntity.ok(taskService.findById(id));
}

// Actualizar una tarea
@PutMapping("/{id}")
public ResponseEntity<TaskResponse> update(@PathVariable Long id, @Valid @RequestBody TaskRequest request) {
    return ResponseEntity.ok(taskService.update(id, request));
}

// Eliminar una tarea
@DeleteMapping("/{id}")
public ResponseEntity<Void> delete(@PathVariable Long id) {
    taskService.delete(id);
    return ResponseEntity.noContent().build();
}

// Cambiar el estado de una tarea
@PatchMapping("/{id}/status")
public ResponseEntity<TaskResponse> changeStatus(@PathVariable Long id, @RequestParam TaskStatus status) {
    return ResponseEntity.ok(taskService.changeStatus(id, status));
}

// Buscar tareas por rango de fechas
@GetMapping("/range")
public ResponseEntity<Page<TaskResponse>> findByDateRange(
    @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE) LocalDate start,
    @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE) LocalDate end,
    Pageable pageable
) {
    return ResponseEntity.ok(taskService.findByDateRange(start, end, pageable));
}

// Obtener tareas activas (PENDING o IN_PROGRESS) por developer
@GetMapping("/developer/{developerId}/active")
public ResponseEntity<List<TaskResponse>> findActiveByDeveloper(@PathVariable Long developerId) {
    return ResponseEntity.ok(taskService.findActiveTasksByDeveloper(developerId));
}
}

```

Pruebas Funcionales con Postman

En esta sección se realizarán pruebas sobre los endpoints del recurso **Task** usando Postman. Estas pruebas permiten verificar el comportamiento de la API en operaciones clave como crear tareas, cambiar su estado, listar por fechas y validar reglas de negocio.

Se utilizará una colección con una variable `base_url` para facilitar la configuración del entorno de pruebas y mantener consistencia en las peticiones.

Crear Tarea

POST Crear tarea +

taskmanagement-api / task / **Crear tarea**

POST {{base_url}} /tasks

Params Authorization Headers (8) Body Scripts Settings

none form-data x-www-form-urlencoded raw binary GraphQL **JSON**

```
1 {
2   "title": "Diseñar pantalla de login",
3   "description": "Maquetar y definir estilos del login",
4   "developerId": 2,
5   "startDate": "2025-05-20",
6   "endDate": "2025-05-24"
7 }
8
```

Crear Tarea – Fallo por título duplicado

POST Crear Tarea – Fallo por título duplicado +

taskmanagement-api / task / **Crear Tarea – Fallo por título duplicado**

POST {{base_url}} /tasks

Params Authorization Headers (8) Body Scripts Settings

none form-data x-www-form-urlencoded raw binary GraphQL **JSON**

```
1 {
2   "title": "Diseñar pantalla de login",
3   "description": "Tarea repetida",
4   "developerId": 2,
5   "startDate": "2025-06-01",
6   "endDate": "2025-06-05"
7 }
8
```

Body Cookies Headers (5) Test Results |

{ } JSON Preview Visualize |

```
1 {
2   "type": "about:blank",
3   "title": "Recurso duplicado",
4   "status": 409,
5   "detail": "Ya existe una tarea con ese título",
6   "instance": "/api/v1/tasks",
7   "path": "uri=/api/v1/tasks"
8 }
```

Crear Tarea – Fallo por developer inexistente

The screenshot shows a Postman request to create a task. The URL is `taskmanagement-api / task / Crear Tarea – Fallo por developer inexistente`. The method is POST. The body contains the following JSON:

```
1 {
2   "title": "Nueva tarea",
3   "description": "Developer no existe",
4   "developerId": 999,
5   "startDate": "2025-06-01",
6   "endDate": "2025-06-10"
7 }
```

The response body is:

```
1 {
2   "type": "about:blank",
3   "title": "Recurso no encontrado",
4   "status": 404,
5   "detail": "Developer no encontrado",
6   "instance": "/api/v1/tasks",
7   "path": "uri=/api/v1/tasks"
8 }
```

Crear Tarea – Fallo por exceso de tareas activas

- Repetir 5 veces la creación de tareas en estado PENDING para el mismo developer, luego un 6to intento.

The screenshot shows a Postman request to create a task. The URL is `taskmanagement-api / task / Crear Tarea – Fallo por exceso de tareas activas`. The method is POST. The body contains the following JSON:

```
1 {
2   "title": "Nueva tarea 6",
3   "description": "Developer no existe",
4   "developerId": 2,
5   "startDate": "2025-06-01",
6   "endDate": "2025-06-10"
7 }
```

The response body is:

```
1 {
2   "type": "about:blank",
3   "title": "Error interno",
4   "status": 500,
5   "detail": "El developer ya tiene el número máximo de tareas activas",
6   "instance": "/api/v1/tasks",
7   "path": "uri=/api/v1/tasks"
8 }
```

Cambiar estado – Correcto (PENDING → IN_PROGRESS)

The screenshot shows the Postman interface for a PATCH request. The URL is `taskmanagement-api / task / Cambiar estado – Correcto (PENDING → IN_PROGRESS)`. The method is set to PATCH. In the Headers section, there are 7 items. The Body section contains a JSON object representing a task:

```
1 {
2   "id": 3,
3   "title": "Diseñar pantalla de login",
4   "description": "Maquetar y definir estilos del login",
5   "status": "IN_PROGRESS",
6   "developerName": "Henry Mendoza Puerta",
7   "startDate": "2025-05-20",
8   "endDate": "2025-05-24"
9 }
```

Cambiar estado – Incorrecto (COMPLETED sin pasar por IN_PROGRESS)

The screenshot shows the Postman interface for a PATCH request. The URL is `taskmanagement-api / task / Cambiar estado – Incorrecto (COMPLETED sin pasar por IN_PROGRESS)`. The method is set to PATCH. In the Headers section, there are 4 items. The Body section contains a JSON object representing a task with status COMPLETED:

```
1 {
2   "type": "about:blank",
3   "title": "Error interno",
4   "status": 500,
5   "detail": "Solo se puede completar una tarea que estuvo en progreso",
6   "instance": "/api/v1/tasks/4/status",
7   "path": "uri=/api/v1/tasks/4/status"
8 }
```

A red banner at the top right indicates a **500 Internal Server Error** response.

Cambiar estado – Incorrecto (IN_PROGRESS desde COMPLETED)

The screenshot shows a POSTMAN interface with the following details:

- Method:** PATCH
- URL:** {{base_url}}/tasks/4/status?status=IN_PROGRESS
- Headers:** (7) - includes Content-Type: application/json
- Params:** (1) - includes status (Value: IN_PROGRESS)
- Body:** (JSON) - empty

The screenshot shows a POSTMAN interface with the following details:

- Method:** GET
- URL:** {{base_url}}/tasks/4
- Headers:** (4) - includes Accept: application/json
- Body:** (JSON) - empty

The response body is a JSON object:

```
1 {  
2   "id": 4,  
3   "title": "Diseñar pantalla de login",  
4   "description": "Maquetar y definir estilos del login",  
5   "status": "IN_PROGRESS",  
6   "developerName": "Henry Mendoza Puerta",  
7   "startDate": "2025-05-20",  
8   "endDate": "2025-05-24"  
9 }
```

Buscar Tarea por ID – Correcto

The screenshot shows a POSTMAN interface with the following details:

- Method:** GET
- URL:** {{base_url}}/tasks/3
- Headers:** (6) - includes Accept: application/json
- Params:** (1) - includes id (Value: 3)
- Body:** (JSON) - empty

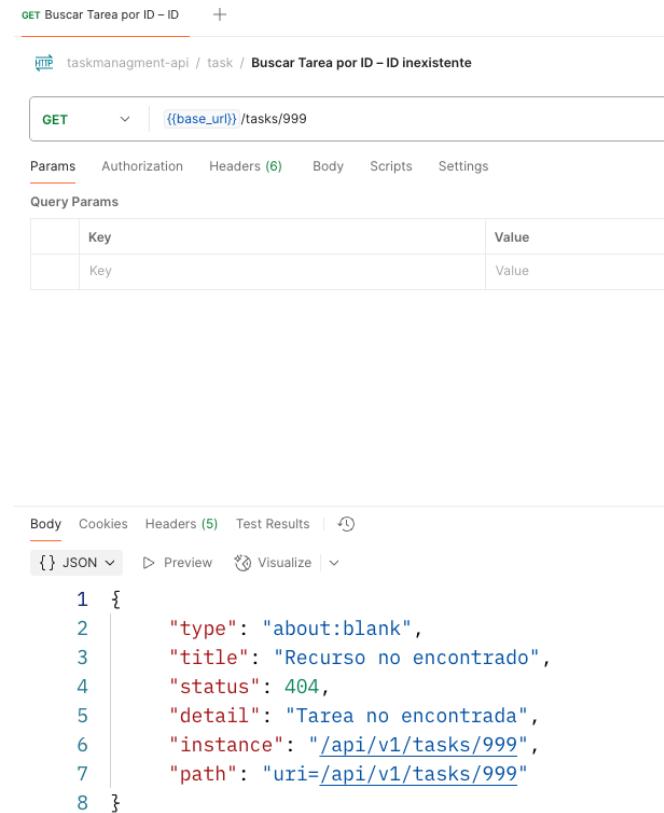
The screenshot shows a POSTMAN interface with the following details:

- Method:** GET
- URL:** {{base_url}}/tasks/3
- Headers:** (5) - includes Accept: application/json
- Body:** (JSON) - empty

The response body is a JSON object:

```
1 {  
2   "id": 3,  
3   "title": "Diseñar pantalla de login",  
4   "description": "Maquetar y definir estilos del login",  
5   "status": "IN_PROGRESS",  
6   "developerName": "Henry Mendoza Puerta",  
7   "startDate": "2025-05-20",  
8   "endDate": "2025-05-24"  
9 }
```

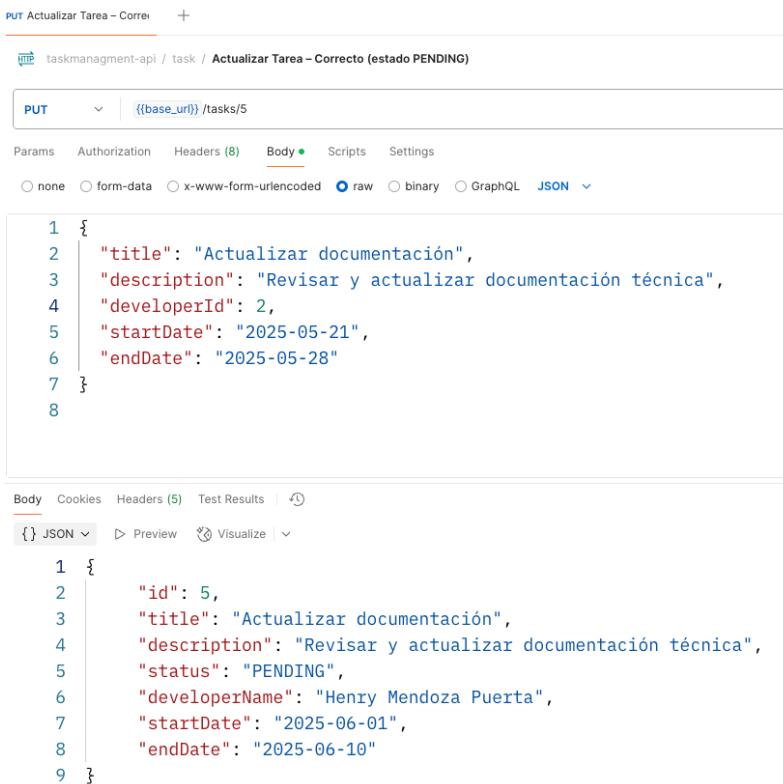
Buscar Tarea por ID – ID inexistente



The screenshot shows a Postman request for "Buscar Tarea por ID – ID inexistente". The URL is `taskmanagement-api / task / Buscar Tarea por ID – ID inexistente`. The method is GET, and the path is `base_url /tasks/999`. The Headers tab shows `Content-Type: application/json`. The Body tab displays the following JSON response:

```
1 {  
2   "type": "about:blank",  
3   "title": "Recurso no encontrado",  
4   "status": 404,  
5   "detail": "Tarea no encontrada",  
6   "instance": "/api/v1/tasks/999",  
7   "path": "uri=/api/v1/tasks/999"  
8 }
```

Actualizar Tarea – Correcto (estado PENDING)



The screenshot shows a Postman request for "Actualizar Tarea – Correcto (estado PENDING)". The URL is `taskmanagement-api / task / Actualizar Tarea – Correcto (estado PENDING)`. The method is PUT, and the path is `base_url /tasks/5`. The Headers tab shows `Content-Type: application/json`. The Body tab displays the following JSON request body:

```
1 {  
2   "title": "Actualizar documentación",  
3   "description": "Revisar y actualizar documentación técnica",  
4   "developerId": 2,  
5   "startDate": "2025-05-21",  
6   "endDate": "2025-05-28"  
7 }  
8
```

The response is a JSON object of the updated task with id 5:

```
1 {  
2   "id": 5,  
3   "title": "Actualizar documentación",  
4   "description": "Revisar y actualizar documentación técnica",  
5   "status": "PENDING",  
6   "developerName": "Henry Mendoza Puerta",  
7   "startDate": "2025-06-01",  
8   "endDate": "2025-06-10"  
9 }
```

Actualizar Tarea – Incorrecto (estado distinto a PENDING)

The screenshot shows a POST request to `taskmanagement-api / task / Actualizar Tarea – Incorrecto (estado distinto a PENDING)`. The Body tab contains the following JSON:

```
1 {
2   "title": "Optimizar consultas SQL",
3   "description": "Mejorar el rendimiento de la base de datos",
4   "developerId": 2,
5   "startDate": "2025-05-22",
6   "endDate": "2025-05-29"
7 }
```

The response status is **500 Internal Server Error** with a duration of 32 ms. The Body tab shows the error message:

```
1 {
2   "type": "about:blank",
3   "title": "Error interno",
4   "status": 500,
5   "detail": "Solo se puede reasignar una tarea en estado PENDING",
6   "instance": "/api/v1/tasks/4",
7   "path": "uri=/api/v1/tasks/4"
8 }
```

Buscar por Rango de Fechas – Correcto

The screenshot shows a GET request to `taskmanagement-api / task / Buscar por Rango de Fechas – Correcto`. The Params tab has the following query parameters:

Key	Value	Description
start	2025-05-01	
end	2025-06-10	

The response status is **200 OK** with a duration of 48 ms and a size of 1.05 Ki. The Body tab shows the following JSON:

```
1 {
2   "content": [
3     {
4       "id": 3,
5       "title": "Diseñar pantalla de login",
6       "description": "Maquetar y definir estilos del login",
7       "status": "IN_PROGRESS",
8       "developerName": "Henry Mendoza Puerta",
9       "startDate": "2025-05-20",
10      "endDate": "2025-05-24"
11    },
12    {
13      "id": 4,
14      "title": "Nueva tarea",
15      "description": "Developer no existe",
16      "status": "COMPLETED",
17      "developerName": "Henry Mendoza Puerta",
18      "startDate": "2025-06-01",
19      "endDate": "2025-06-10"
20    }
21  ]
22 }
```

Buscar por Rango – Incorrecto (start > end)

The screenshot shows a POST request to `taskmanagement-api / task / Buscar por Rango – Incorrecto (start > end)`. The URL is `http://{{base_url}}/tasks/range?start=2025-06-01&end=2025-05-01`. The 'Query Params' table shows 'start' and 'end' parameters both set to '2025-05-01'. The response status is 500 Internal Server Error, with a body containing an error message in JSON format:

```
1 {  
2   "type": "about:blank",  
3   "title": "Error interno",  
4   "status": 500,  
5   "detail": "La fecha de inicio no puede ser posterior a la fecha de fin",  
6   "instance": "/api/v1/tasks/range",  
7   "path": "uri=/api/v1/tasks/range"  
8 }
```

Eliminar Tarea – Correcto

The screenshot shows a DELETE request to `taskmanagement-api / task / Eliminar Tarea – Correcto`. The URL is `http://{{base_url}}/tasks/3`. The 'Query Params' table shows a 'Key' parameter. The response status is 204 No Content, with a body containing the number 1.

Eliminar Tarea – Tarea inexistente

The screenshot shows a Postman request to delete a task with ID 999. The URL is `http://taskmanagement-api/task/((base_url))/tasks/999`. The response status is 404 Not Found, and the response body is a JSON object indicating the task was not found.

Key	Value	Description
Key	Value	Description

The screenshot shows a GET request to retrieve active tasks for developer ID 2. The URL is `http://taskmanagement-api/task/((base_url))/tasks/developer/2/active`. The response status is 200 OK, and the response body is a JSON array containing two task objects.

Key	Value	Description
Key	Value	Description

Tareas activas por Developer – Correcto

The screenshot shows a GET request to retrieve active tasks for developer ID 2. The URL is `http://taskmanagement-api/task/((base_url))/tasks/developer/2/active`. The response status is 200 OK, and the response body is a JSON array containing two task objects.

Key	Value	Description
Key	Value	Description

The screenshot shows a GET request to retrieve active tasks for developer ID 2. The URL is `http://taskmanagement-api/task/((base_url))/tasks/developer/2/active`. The response status is 200 OK, and the response body is a JSON array containing two task objects.

Key	Value	Description
Key	Value	Description

Tareas activas – Developer no existe

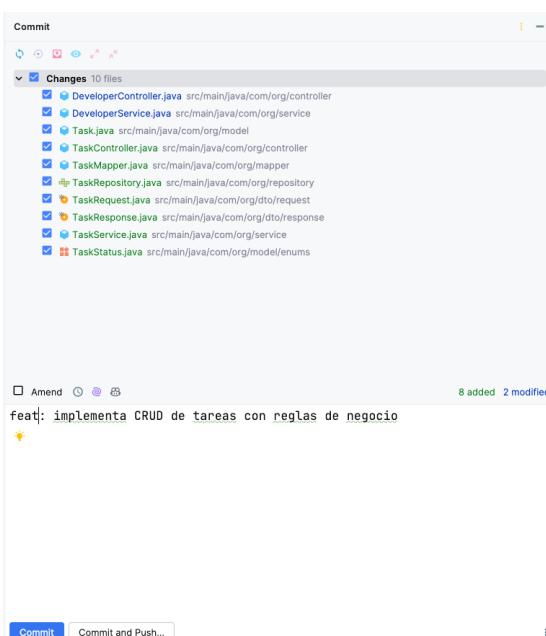
The screenshot shows a REST API client interface. At the top, it says "GET Tareas activas – Develop". Below that is the URL "taskmanagement-api / task / Tareas activas – Developer no existe". The method dropdown is set to "GET". The "Headers (6)" tab is selected. Under "Query Params", there is a table with one row and two columns: "Key" and "Value". The "Key" column has "Key" and the "Value" column has "Description". Below the table, there are tabs for "Body", "Cookies", "Headers (5)", and "Test Results". The "Body" tab shows a JSON response: "1 []". The status bar at the bottom right indicates "200 OK" with a green background, "33 ms", and "166".

Trabajo Colaborativo con Git – Integración de la Funcionalidad Task

Una vez finalizada la implementación de la rama `feature/task`, el siguiente paso es integrarla a la rama `develop` para consolidar los avances. Esta operación se realiza desde IntelliJ IDEA para facilitar el control visual de los cambios.

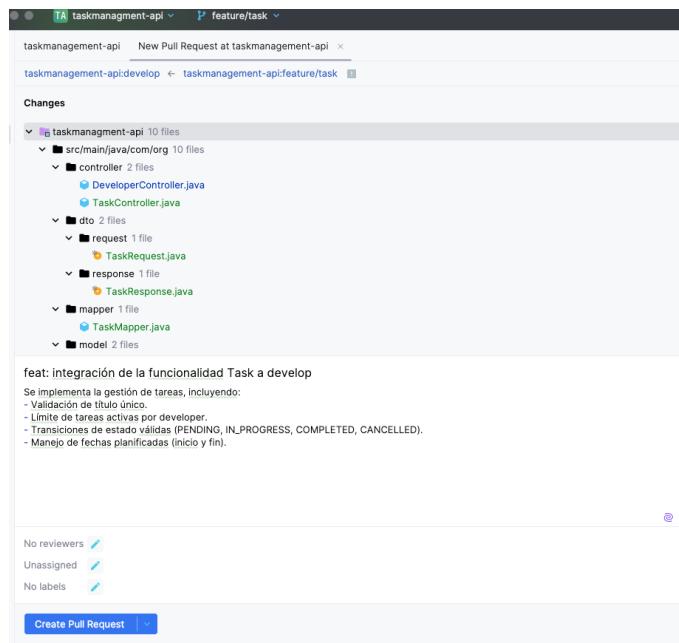
Hacer push de la rama local `feature/task` a GitHub

- Asegúrate de tener todos los cambios guardados.
- Desde `Git > Commit...` confirma el mensaje:
- Realiza el **Commit & Push** hacia la rama remota `feature/task`.



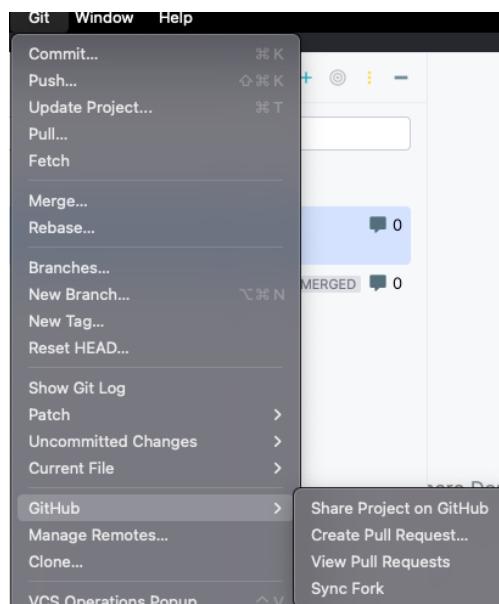
Crear Pull Request (PR) desde feature/task hacia develop

- Abre GitHub desde IntelliJ (Git > GitHub > Create Pull Requests o desde navegador como se realizó con feature/developer).
- Selecciona:
 - **Base branch:** develop
 - **Compare:** feature/task



Revisar y hacer merge a develop

- Verifica que no haya conflictos.
- Selecciona la opción **Merge Pull Request**.
- Mensaje del merge:



taskmanagement-api #2

feat: integración de la funcionalidad Task a develop

#2 · created Today, by hampcode

feat: implementa CRUD con validaciones y manejo de errores

#1 · created Today, by hampcode

MERGED 0

Debes dar doble clic en el PR que quieras revisar

taskmanagement-api #2

feat: integración de la funcionalidad Task a develop

View Timeline

Changes

feature/task

AI Assistant

Summary

This Pull Request introduces features for managing tasks, such as CRUD operations, task state transitions, pagination, filtering, and date range queries. Additionally, it refactors existing pagination in the DeveloperController and the DeveloperService to use Pageable.

Main Changes

- DeveloperController: Refactored getPaginated method to use Pageable instead of custom page and size parameters.
- New TaskController: Added endpoints for task CRUD operations, status updates, and advanced queries (date range and active tasks).
- New DTOs: Added TaskRequest and TaskResponse for task data structure.
- New TaskMapper: Handles conversions between entities and DTOs for tasks.
- New TaskStatus: Added a data model for tasks including fields like title, description, status, and related developer.
- New TaskRepository: Added repository methods for custom queries related to tasks.
- New TaskService: Encapsulates business logic for task creation, updates, deletion, and querying.
- DeveloperService: Modified the findPaginated method to support Pageable.

Regenerate

Henry A. Mendoza Puerla 2 minutes ago

Se implementa la gestión de tareas, incluyendo:

- Validación de título único.
- Límite de tareas activas por desarrollador.
- Transiciones de estado válidas (PENDING, IN_PROGRESS, COMPLETED, CANCELLED).
- Manejo de fechas planificadas (inicio y fin).

Darth Vader 10 minutes ago

added a commit to this pull request

feat: Implementa CRUD de tareas con reglas de negocio

Request Review...

Para realizar Merge seleccionar los 3 puntos que están al costado de Request Review

Missing reviewer

Request Review...

Merge Review... >

Close Pull Request

Merge...

Squash and Merge...

Rebase

Merge Review

Merge commit message:

Merge pull request #2

feat: funcionalidad integrada a develop

Cancel Merge

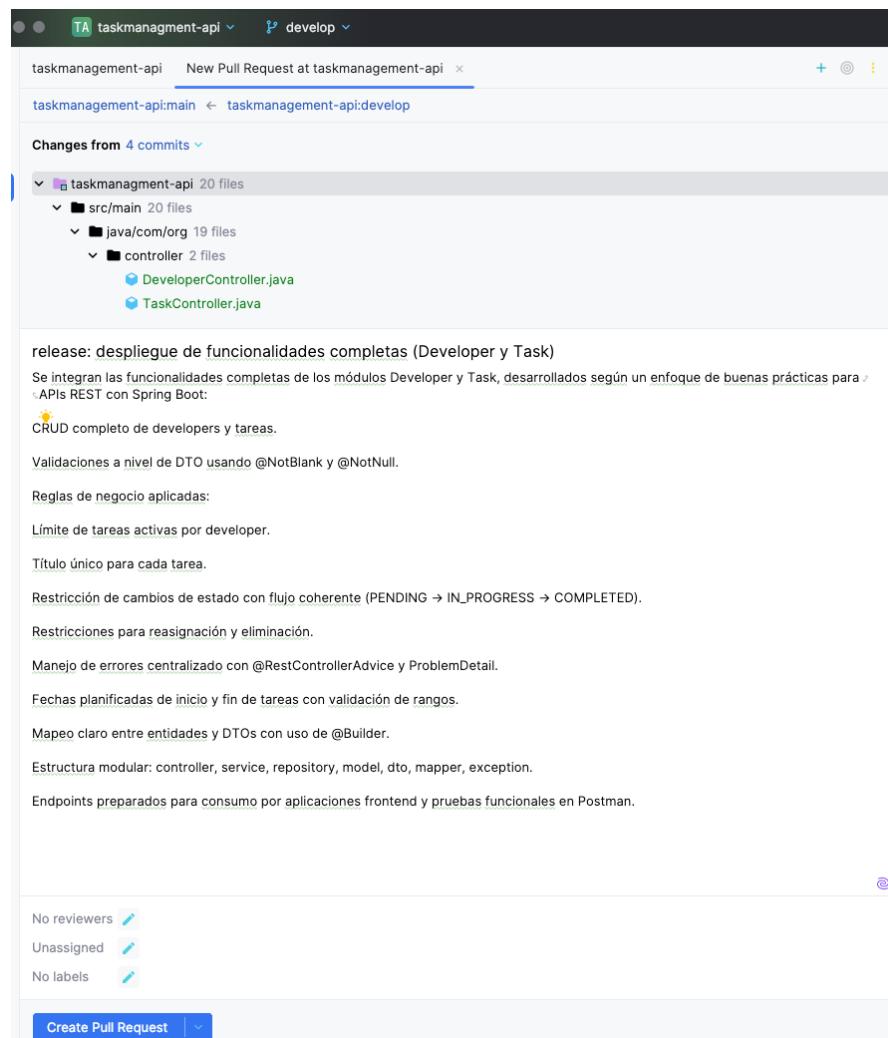
Eliminar rama feature/task

Solo elimina la rama si **ya fue integrada correctamente** a `develop` o `main`. Puedes mantenerla un par de días si estás haciendo validaciones finales, pero lo ideal es no dejar ramas feature antiguas que ya cumplieron su ciclo.

Pull Request `develop` → `main`

Después de integrar las funcionalidades `developer` y `task` en `develop`, se procede con su consolidación en `main` como una versión lista para pruebas o despliegue.

- **Base:** `main`
- **Compare:** `develop`

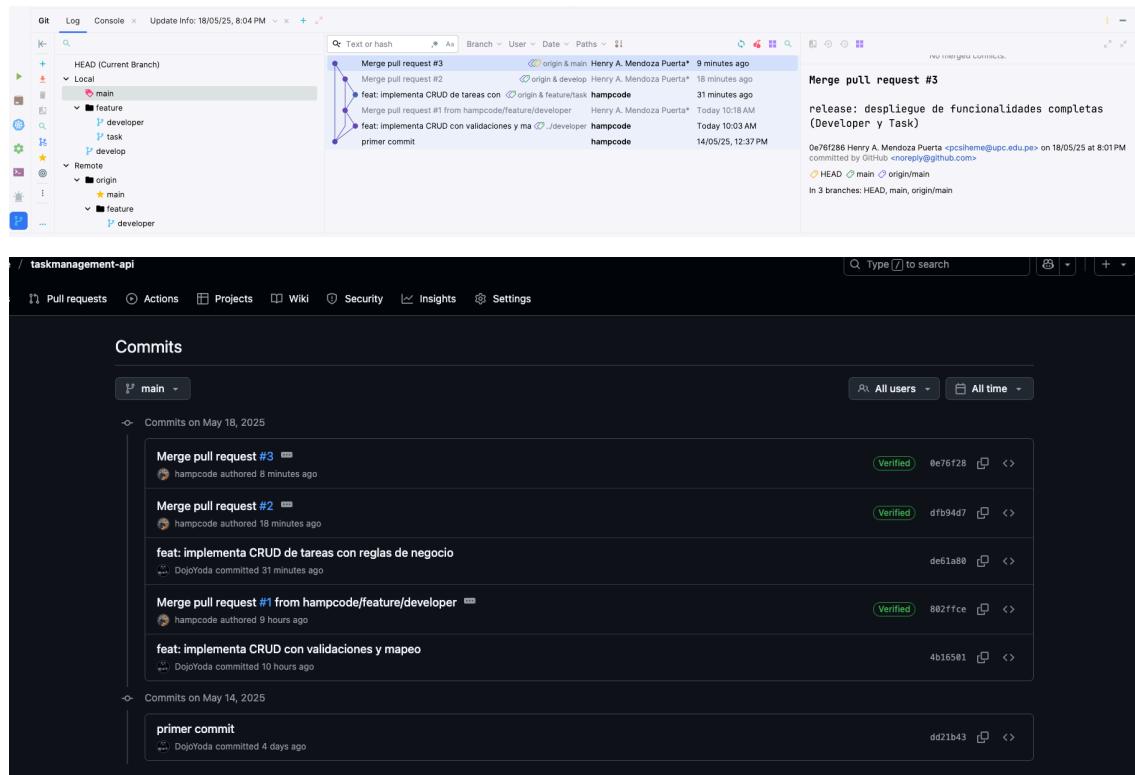


Confirmar Merge

- Una vez revisado, realizar el merge hacia `main`.
 - El proyecto queda listo para pruebas integradas o despliegue.



Revisar los comits realizados



Versionado del Proyecto con Git Tags

Para marcar versiones estables del proyecto y facilitar su seguimiento, se utiliza el sistema de **etiquetado (tags)** de Git. Un tag permite identificar un punto específico en el historial, como una versión lista para pruebas, integración o despliegue.

Convención recomendada

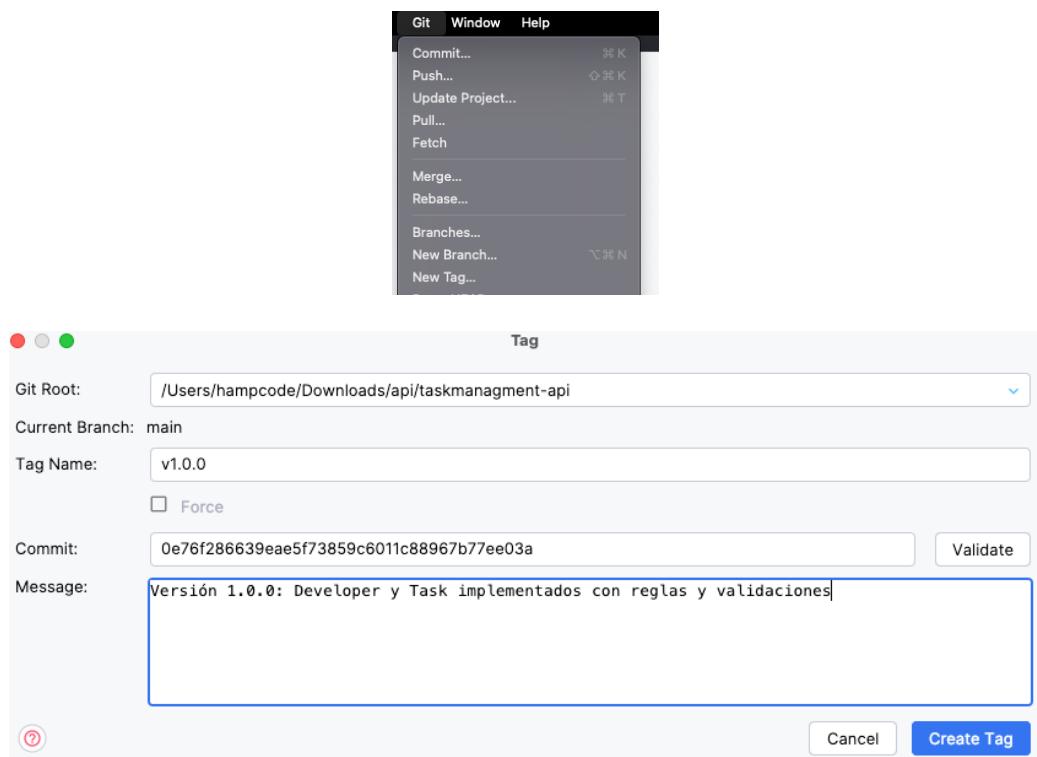
Usa el formato **semántico**: v[MAJOR].[MINOR].[PATCH]

Esta convención ayuda a comunicar claramente el tipo de cambios realizados en una versión. Cada parte tiene un significado:

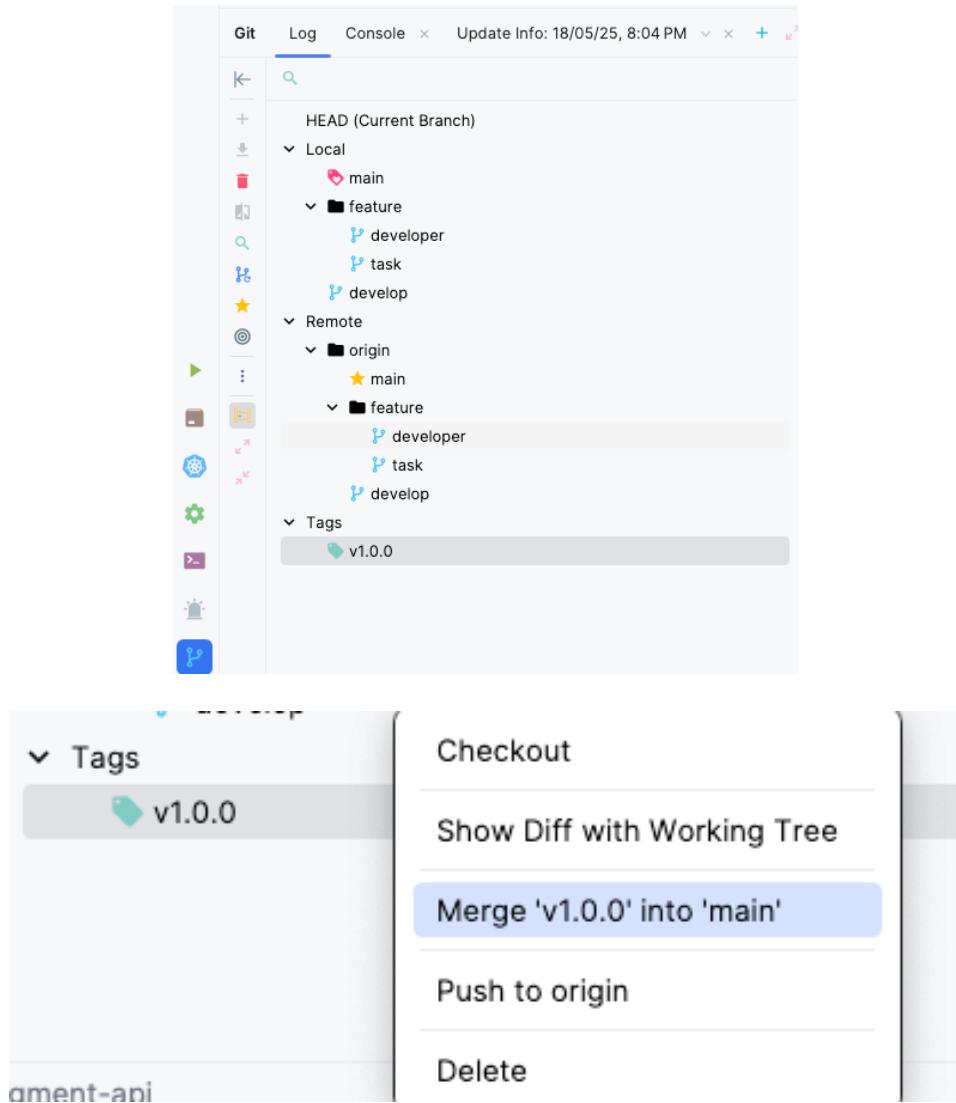
- **MAJOR** (v1.0.0 → v2.0.0)
Se incrementa cuando se hacen cambios incompatibles con versiones anteriores (rompe la compatibilidad).
- **MINOR** (v1.0.0 → v1.1.0)
Se incrementa cuando se agregan funcionalidades nuevas compatibles con versiones anteriores.
- **PATCH** (v1.0.0 → v1.0.1)
Se incrementa cuando se hacen correcciones de errores sin cambiar la funcionalidad existente.

Ejemplo práctico:

- v1.0.0 – Primera versión estable.
- v1.1.0 – Se agregó el módulo de reportes.
- v1.1.1 – Se corrigió un bug en el endpoint de tareas.



Realizar Push to origin del Tag V1.0.0



Hemos finalizado.

HAMP