**GUIA PORTAFOLIO**

Table of Contents

[📂 Cómo estructurar tu portafolio 1](#_Toc207452073)

[**1.README profesional** en cada proyecto: 1](#_Toc207452074)

[2.Currículum en tu repositorio principal (o link a tu CV en LinkedIn). 1](#_Toc207452075)

[**3.Demos desplegados** (en Railway, Render, Vercel, AWS, etc.) para que reclutadores puedan probarlos sin necesidad de clonar el repo. 1](#_Toc207452076)

[📑 Cómo presentar tu portafolio 1](#_Toc207452077)

[🚀 Proyectos que puedes incluir 2](#_Toc207452078)

[1. CRUD API simple (task-manager-api) 2](#_Toc207452079)

[2. API con autenticación y roles(user-auth-service) 2](#_Toc207452080)

[3. E-commerce básico (mini backend) 2](#_Toc207452081)

[4. Microservicios + comunicación(booking-microservices) 2](#_Toc207452082)

[5. Proyecto con DevOps / despliegue(integrations-api) 3](#_Toc207452083)

[6. Proyecto con integraciones externas(stock-market) 3](#_Toc207452084)

[🎯 Consejos extra 3](#_Toc207452085)

# 📂 Cómo estructurar tu portafolio

Tu portafolio debería estar en **GitHub/GitLab/Bitbucket** y organizado de manera clara. Incluye:

## **1.README profesional** en cada proyecto:

* + Breve descripción.
  + Tecnologías usadas.
  + Cómo instalar/ejecutar.
  + Screenshots o diagramas.
  + Explicación de arquitectura (si aplica).

## 2.Currículum en tu repositorio principal (o link a tu CV en LinkedIn).

## **3.Demos desplegados** (en Railway, Render, Vercel, AWS, etc.) para que reclutadores puedan probarlos sin necesidad de clonar el repo.

# 📑 Cómo presentar tu portafolio

* Un repositorio llamado **java-backend-portfolio** que tenga un **README principal** con:
  + ✨ Breve bio profesional.
  + 🚀 Lista de proyectos (con links).
  + 🛠️ Tecnologías que dominas.
  + 🌎 Links de despliegues (si los tienes).
  + 📧 Contacto (LinkedIn, correo, etc.).

Ejemplo de estructura:

📦 java-backend-portfolio

┣ 📂 task-manager-api

┣ 📂 user-auth-service

┣ 📂 ecommerce-backend

┣ 📂 booking-microservices

┣ 📂 integrations-api

┣ 📂 stock-market

┣ README.md

# 🚀 Proyectos que puedes incluir

Aquí te propongo una ruta de proyectos, cada uno agregando más complejidad:

## 1. CRUD API con autenticacion y auditoria(task-manager-api)

### Tecnologías

* Spring Boot
* Langchain4j
* Spring Data JPA
* H2/PostgreSQL
* Docker
* Render

### Idea

* API de gestión de tareas (to-do list).

### Objetivo

* mostrar que dominas CRUD + REST + conexión a DB.
* Despliegue de APIs

### 📝 Requerimientos del Cliente

#### 1. Objetivo principal

Quiero una **aplicación web de gestión de tareas** que permita a los usuarios organizar sus actividades de forma sencilla, colaborativa y accesible desde cualquier dispositivo.

#### 2. Requerimientos funcionales

##### Autenticación y usuarios

* + Registro de usuarios (correo y contraseña).
  + Inicio de sesión seguro con recuperación de contraseña.
  + Perfil básico (nombre, correo).

##### Gestión de tareas

* + Crear, editar, completar y eliminar tareas.
  + Posibilidad de asignar **fechas de vencimiento** y recordatorios.
  + Añadir **etiquetas o categorías** para organizar tareas (ej: trabajo, personal, urgente).
  + Adjuntar archivos a una tarea (PDF, imágenes, etc.).

##### Gestión de proyectos o listas

* + Los usuarios deben poder agrupar tareas en **listas** o **proyectos**.
  + Vista tipo **kanban** (como Trello) para arrastrar tareas entre estados (pendiente, en progreso, completado).
  + Vista tipo **lista simple** (como Todoist).

##### Colaboración

* + Invitar a otros usuarios a un proyecto.
  + Asignar tareas a diferentes miembros del equipo.
  + Comentarios dentro de una tarea (chat ligero).

##### Notificaciones

* + Notificaciones por correo y en la app cuando:
    - Se les asigna una tarea.
    - Una tarea está próxima a vencerse.
    - Alguien comenta en una tarea asignada.

##### Búsqueda y filtros

* + Buscar tareas por nombre, etiquetas o miembros asignados.
  + Filtrar por vencimiento (hoy, esta semana, vencidas).

#### 3. Requerimientos no funcionales

* **Accesibilidad**: La aplicación debe funcionar en desktop y móvil (responsive).
* **Seguridad**: Contraseñas encriptadas, conexión HTTPS, permisos según roles (usuario normal vs. administrador de proyecto).
* **Escalabilidad**: Preparada para soportar equipos grandes en un futuro (ej. empresas).
* **Rendimiento**: Las tareas deben cargarse en menos de 2 segundos incluso en proyectos grandes.
* **Usabilidad**: Interfaz limpia e intuitiva, sin necesidad de tutorial largo.

#### 4. Extras deseables (a futuro)

* Integración con **Google Calendar** o **Outlook**.
* Soporte para **subtareas**.
* Vista tipo **timeline / calendario** (como Asana).
* Aplicación móvil nativa.

👉 En resumen, busco algo que combine la **simplicidad de Todoist**, la **colaboración de Asana**, y la **visualización tipo tablero de Trello**.

### Creación base de datos

#### 🔑 Autenticación (ya hecho, pero ajustamos)

* **User**
  + id (PK)
  + username / email (unique)
  + password (hashed)
  + enabled, created\_at, updated\_at
* **Role**
  + id (PK)
  + name (ej: ADMIN, USER, MODERATOR)
* **User\_Role** (tabla intermedia si quieres many-to-many o un role\_id directo en User si es uno-a-muchos)

✔️ Con esto tienes autenticación y autorización lista.

#### 📌 Entidades principales estilo Trello

1. **Board (Tablero)**
   * id (PK)
   * name
   * description
   * created\_at, updated\_at
   * owner\_id (FK → User)

🔗 Relaciones:

* + Un User puede tener muchos Board (uno-a-muchos)
  + Pero también varios usuarios pueden participar en un mismo Board → entonces necesitamos **Board\_Member**

1. **Board\_Member**
   * id (PK)
   * board\_id (FK → Board)
   * user\_id (FK → User)
   * role\_in\_board (ej: OWNER, EDITOR, VIEWER)

🔗 Relaciones:

* + Many-to-Many entre User y Board con atributos extra (rol dentro del tablero).

1. **List (Lista de tareas, como en Trello: ToDo, Doing, Done)**
   * id (PK)
   * name
   * position (para ordenar listas dentro de un tablero)
   * board\_id (FK → Board)

🔗 Relaciones:

* + Un Board tiene muchas List.

1. **Task (Tarea/Tarjeta)**
   * id (PK)
   * title
   * description
   * due\_date
   * position (para ordenar dentro de la lista)
   * list\_id (FK → List)
   * created\_by (FK → User)
   * assigned\_to (FK → User) → opcional (quién trabaja en la tarea)
   * status (ej: OPEN, IN\_PROGRESS, DONE)

🔗 Relaciones:

* + Una List tiene muchas Task.

1. **Comment (Comentarios en tareas)**
   * id (PK)
   * content
   * created\_at
   * user\_id (FK → User)
   * task\_id (FK → Task)

🔗 Relaciones:

* + Una Task puede tener muchos Comment.

1. **Attachment (Archivos en tareas)**
   * id (PK)
   * file\_url (o ruta en almacenamiento externo tipo S3, GCP, local)
   * uploaded\_at
   * task\_id (FK → Task)
   * user\_id (FK → User, quien subió el archivo)

### README

**Breve descripción**

**aplicación web de gestión de tareas** que permite a los usuarios organizar sus actividades de forma sencilla, colaborativa y accesible desde cualquier dispositivo. Con auditoria sobre que usuarios hicieron cambios e las tareas y su estado actual, también se incluye autenticación con los usuarios que tiene la base de datos y también de acuerdo al role que este tenga en los boards que contienen las tareas.

**Tecnologías usadas**

* Spring Boot
* Langchain4j
* Spring Data JPA
* H2/PostgreSQL

**Cómo instalar/ejecutar**

**Ejecutar con el deploy en render**

**Screenshots o diagrama**

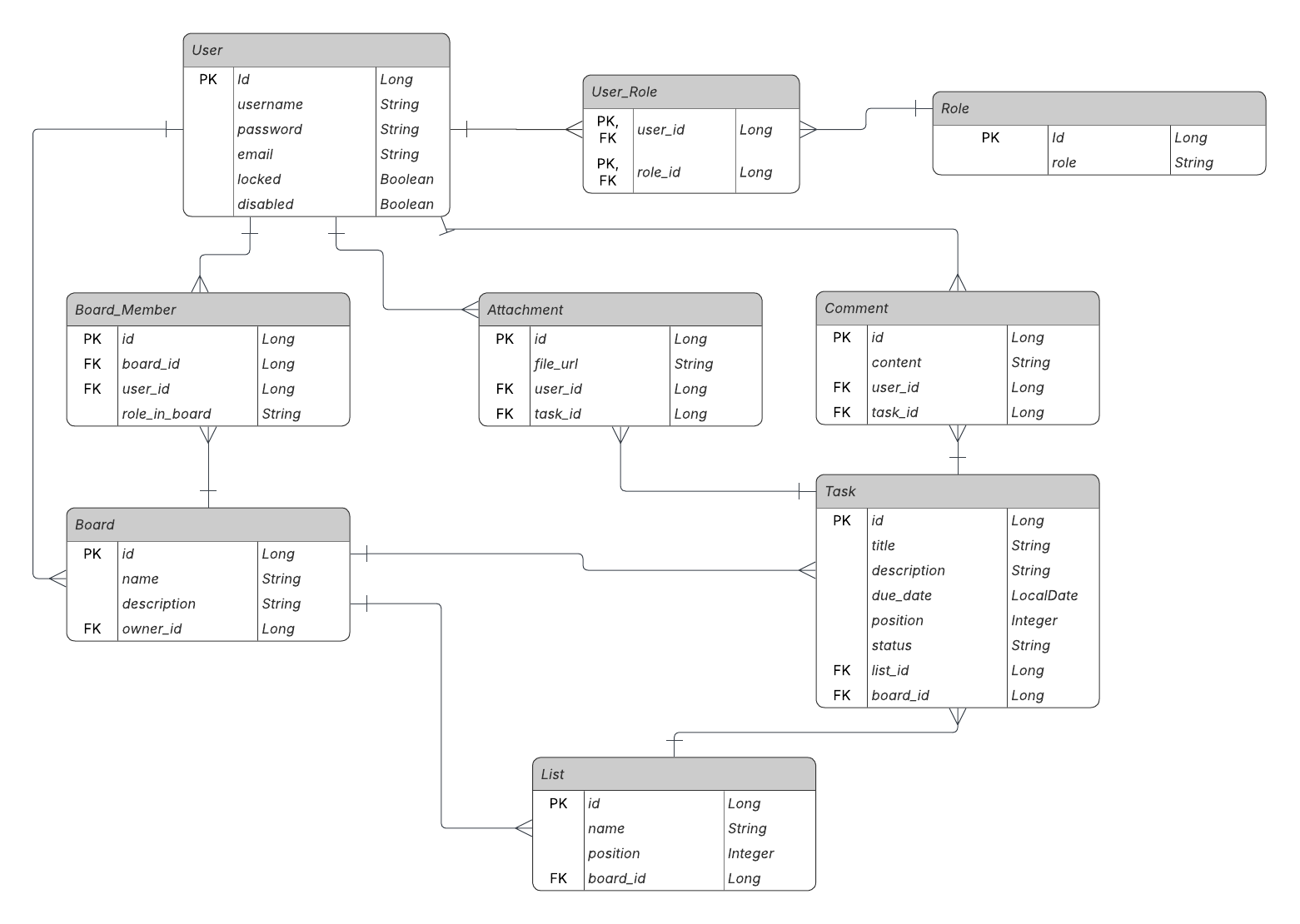
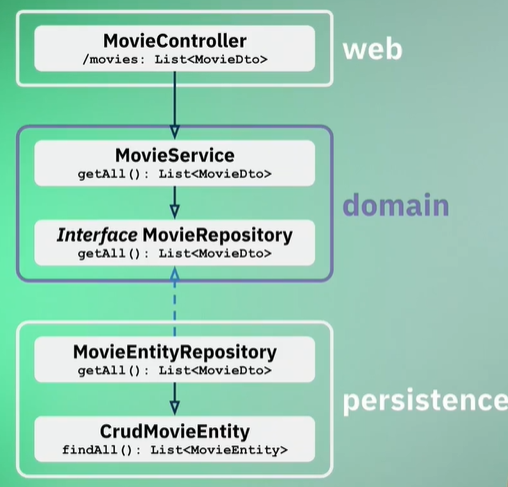


Diagrama Base de datos

**Explicación de arquitectura**

Se uso una arquitectura por capas orientada al dominio para poder separar la persistencia de los datos del motor especifico que se use en la aplicación y de esta manera se garantiza que si por alguna razón el motor se llegue a cambiar se pueda hacer de manera más eficiente puesto que el dominio está separado de la persistencia, y también se tiene la capa web que será la encargada de recibir las peticiones para realizar las funciones correspondientes.  
  


En esta imagen tomamos como ejemplo una clase Entity que representa una tabla llamada Movie en una base de datos, con este ejemplo explicaremos la arquitectura que estamos usando.

En la capa de dominio la interfaz MovieRepository nos ayuda a desacoplar nuestra lógica del Entity que representa la tabla en la base de datos y así empezamos a hablar en términos del dominio, y nos permite flexibilidad y escalabilidad, puesto que ya no dependemos de una única base de datos, sino que si quisiéramos cambiar algo de esta o reemplazarla con otro motor de base de datos podríamos hacerlo. Y esta interfaz lleva nuestro Entity a un DTO (Data Transfer Object). Ahora también tenemos una clase MovieEntityRepository que será la que implemente esta interfaz y también se comunica con CrudMovieEntity que será la interfaz que se conecta directamente con la base de datos. Luego la interfaz MovieRepository será usada por MovieService que es la clase encargada de contener la lógica de caso de uso y por último el controlador MovieController usara ese MovieService para obtener los datos, pero esta vez de forma que sea orientada al dominio y sin acoplamiento con la base de datos.

## 2. API con autenticación y roles(user-auth-service)

* **Tecnologías:** Spring Boot, Spring Security, JWT.
* **Ejemplo:** Sistema de usuarios con login/registro y roles (admin/user).
* Objetivo: demostrar autenticación, seguridad y autorización.

## 3. E-commerce básico (mini backend)

* **Tecnologías:** Spring Boot, JPA, PostgreSQL/MySQL, Swagger.
* **Funcionalidades:**
  + Productos
  + Carrito
  + Órdenes
  + Usuarios con roles (admin puede crear productos)
* Objetivo: mostrar modelado de datos, relaciones, endpoints RESTful.

## 4. Microservicios + comunicación(booking-microservices)

* **Tecnologías:** Spring Cloud, Eureka, API Gateway, Docker.
* **Ejemplo:** Sistema de reservas (usuarios, reservas, pagos).
* Objetivo: mostrar que entiendes microservicios, escalabilidad y comunicación entre servicios.

## 5. Proyecto con DevOps / despliegue(integrations-api)

* **Tecnologías:** Docker, Jenkins/GitHub Actions, Railway/Render/AWS.
* **Ejemplo:** Desplegar tu e-commerce en la nube con CI/CD.
* Objetivo: mostrar que sabes **llevar tu app a producción**.

## 6. Proyecto con integraciones externas(stock-market)

* **Ejemplo:** API de clima, conversor de divisas o stock market.
* **Objetivo:** demostrar que puedes consumir APIs externas.

# 🎯 Consejos extra

* Documenta bien tu código (JavaDocs, README).
* Usa **Swagger/OpenAPI** en tus APIs para documentación automática.
* Crea **tests unitarios** (JUnit, Mockito).
* Mantén commits claros en Git.
* Opcional: hacer un **blog técnico** en GitHub Pages o Medium para mostrar lo que aprendes.