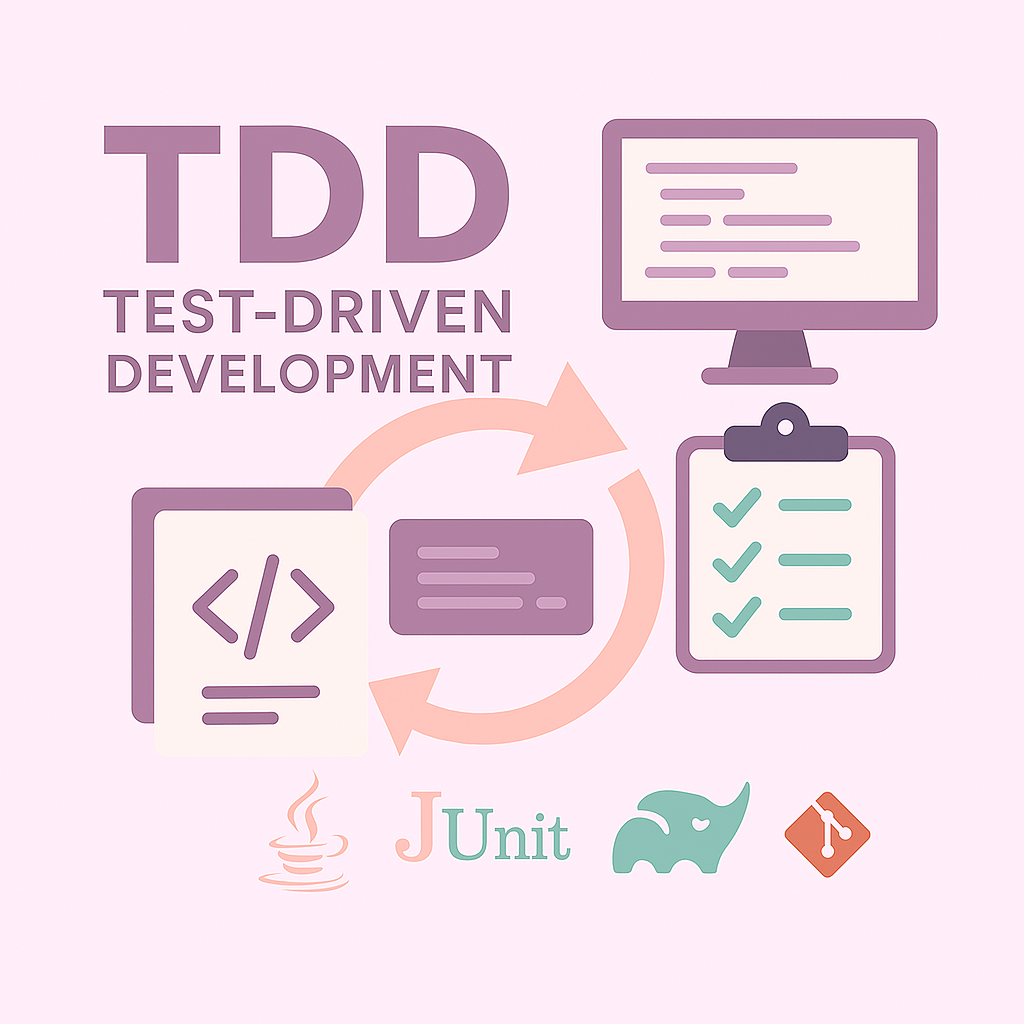
**OTEC - ECAS  
Diplomado de automatización**

**Informe Técnico - Proyecto TDD  
Desarrollo de un sistema CRUD con enfoque en Test Driven Development (TDD)**

****

*"El código que se puede probar es el código en el que se puede confiar." — Autor desconocido, frase promovida por Kent Beck.*

Paz Gutiérrez Portillo.

Lenguaje: Java 17.

Fecha de entrega: Miércoles 9 de julio 2025.

Indice.

· **II. Objetivo del Proyecto**

· **III. Tecnologías Utilizadas**

· **IV. Estructura del Proyecto**

· **V. Desarrollo de Funcionalidades**

· **VI. Testing y Reportes de Cobertura**

· **VII. Instrucciones de Ejecución**

· **VIII. Pantallazos del Proyecto**

· **IX. Observaciones Finales**

· **X. Próximos Pasos**

· **XI. Reflexión Personal**

· **XII. Referencias y Buenas Prácticas TDD**

· **XIII. Conclusiones**

# 

# **I. Introducción**

Este informe documenta el desarrollo de un proyecto backend en **Java 17**, enfocado en la implementación de un sistema CRUD utilizando la metodología **Test Driven Development (TDD)**. El trabajo fue realizado en el contexto de formación técnica en **Fundación SOFOFA TECH**, aplicando estándares de organización, automatización, documentación y control de calidad.

A lo largo del proyecto se integraron tecnologías modernas como **Gradle**, **JUnit 5**, **Mockito** y **JaCoCo**, combinadas con buenas prácticas de versionado en Git y un enfoque sistemático para mejorar la cobertura de pruebas y mantener la legibilidad del código. Se priorizó el diseño de una estructura modular clara, la generación de reportes automáticos, y la entrega de documentación profesional acompañada de evidencia visual (pantallazos del entorno, resultados de pruebas y cobertura).

Este informe describe el resultado final, como también el proceso iterativo de codificación, testing, análisis y mejora continua, reflejando una experiencia cercana a proyectos reales en entornos profesionales.

# **II.** Objetivo del Proyecto

- Aplicar los principios del desarrollo guiado por pruebas desde la planificación hasta la implementación.

- Incorporar herramientas modernas de testing y análisis de código.

- Mantener una estructura limpia y extensible.

- Medir la calidad mediante cobertura de código y modularización efectiva.

# **III.** Tecnologías Utilizadas

|  |  |
| --- | --- |
| Tecnología | Rol en el Proyecto |
| Java 17 | Lenguaje principal |
| Gradle | Sistema de construcción y dependencias |
| JUnit 5 | Framework de testing unitario |
| Mockito | Mocking de clases y simulación de dependencias |
| JaCoCo | Herramienta de análisis de cobertura |
| H2 Database | Base de datos en memoria para pruebas |
| Git + GitHub | Control de versiones y repositorio remoto |

# 

# **IV. Estructura del Proyecto**

PROYECTO\_TDD/

├── src/ # Carpeta principal del código fuente

│ ├── main/java/ # Código funcional de la aplicación

│ │ ├── config/ # Configuración del sistema (por ejemplo, conexión a la base de datos)

│ │ │ └── DBConnection.java # Clase encargada de establecer la conexión con la base de datos

│ │ ├── model/ # Clases que representan entidades del dominio

│ │ │ └── Item.java # Entidad principal del sistema

│ │ └── repository/ # Lógica de negocio y persistencia de datos

│ │ ├── ItemRepository.java # Interfaz para operaciones con la base de datos

│ │ └── ItemService.java # Clase que contiene la lógica de negocio relacionada con Item

│ └── test/java/ # Código de pruebas automatizadas

│ └── repository/ # Pruebas unitarias de la capa de persistencia

│ └── ItemRepositoryTest.java # Test de funcionalidad para el repositorio de Items

├── docs/ # Documentación del proyecto

│ └── Informe\_TDD.docx # Informe técnico con enfoque en desarrollo guiado por pruebas (TDD)

├── build.gradle # Script de construcción del proyecto con Gradle

├── README.md # Archivo informativo con instrucciones y descripción general

· model/: entidades del dominio

· repository/: lógicade persistencia CRUD

· config/: configuraciones generales del entorno

· test/: pruebas unitarias siguiendo la metodología TDD

**V.Desarrollo de Funcionalidades**

Se implementó un módulo CRUD con operaciones de alta, lectura, actualización y eliminación de Item, aplicando el enfoque **Test Driven Development (TDD)** desde las primeras etapas del desarrollo. Cada funcionalidad fue construida siguiendo el ciclo Red → Green → Refactor, permitiendo diseñar pruebas antes de codificar la lógica correspondiente y asegurando que cada unidad cumpla con su propósito de forma aislada y verificable.

El repositorio ItemRepository gestiona la persistencia en memoria mediante H2, mientras que las pruebas en ItemRepositoryTest validan los métodos save, findById, update y deleteById. El desarrollo fue incremental, priorizando la cobertura y legibilidad del código.

Además, se aplicó separación de responsabilidades mediante paquetes organizados (model, repository, config) y se integraron herramientas como JaCoCo para medir la cobertura de cada componente. Se documentaron casos de uso con evidencia visual, facilitando la revisión y comprensión del ciclo de construcción y validación de cada operación.

# **VI.** Testing y Reportes de Cobertura

Ubicación del reporte: build/reports/jacoco/test/html/index.html

Cobertura actual:

|  |  |  |
| --- | --- | --- |
| Paquete | Instrucciones | Ramas |
| model | 100 % | 100 % |
| repository | 90 % | 100 % |
| config | 50 % | — |

# **VII.Instrucciones de Ejecución**

./gradlew clean test jacocoTestReport

- Limpia proyectos previos

- Ejecuta tests JUnit

- Genera reporte JaCoCo HTML

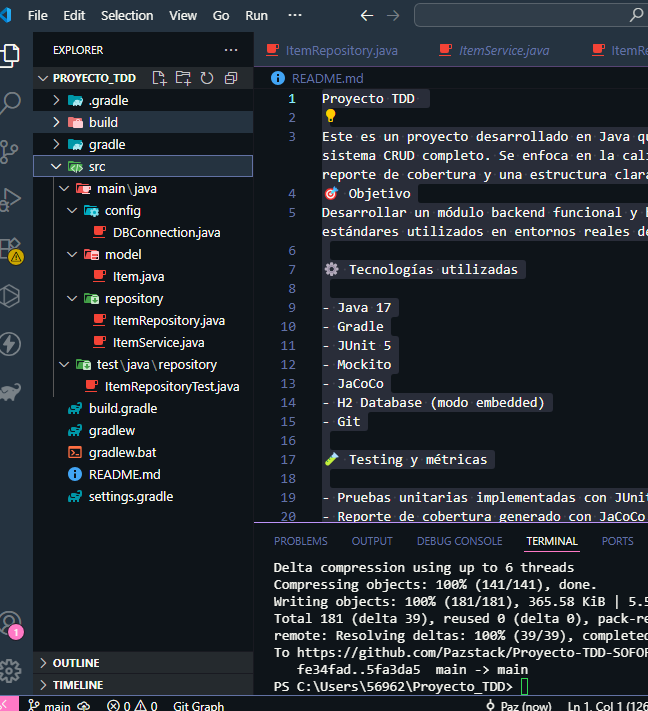
Ubicación del reporte generado por JaCoCo:

build/reports/jacoco/test/html/index.html

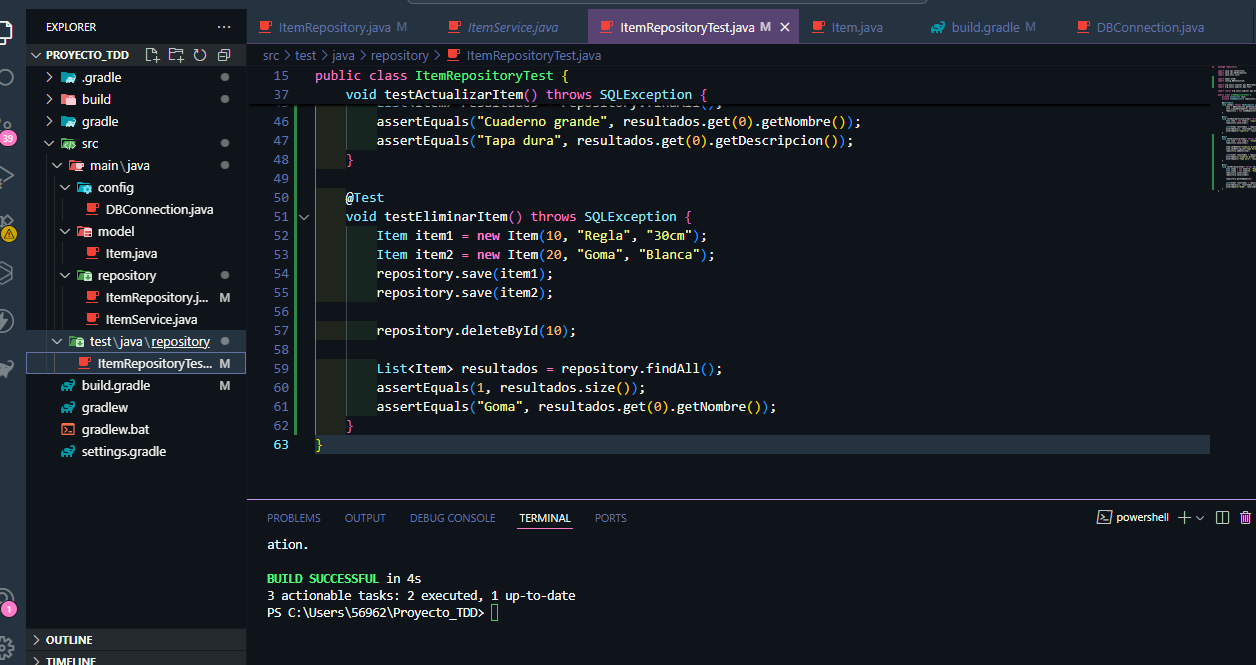
* Permite visualizar gráficamente la cobertura de código
* Muestra métricas por paquete, clase, método y línea
* Usa colores (verde/amarillo/rojo) para indicar nivel de cobertura
* Se abre en cualquier navegador web

# 

# **VIII. Pantallazos del Proyecto**



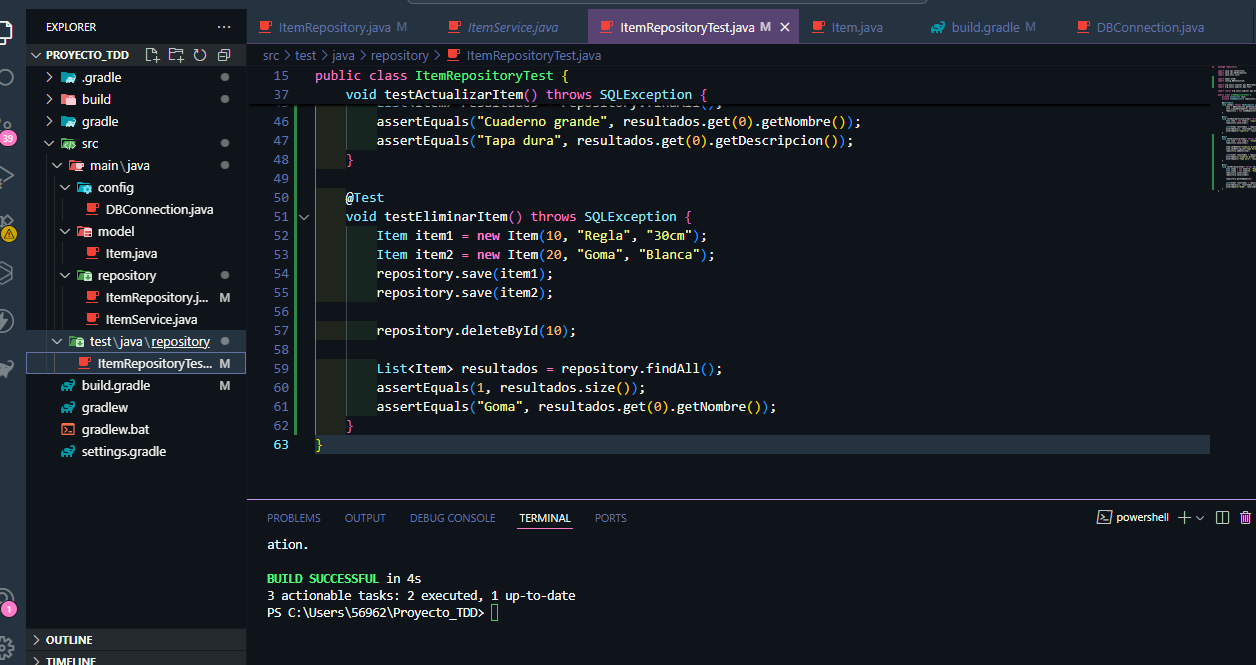
Estructura del proyecto en Visual Studio Code → Estructura de carepetas.



Reporte HTML de cobertura → resumen de cobertura de código generada con JaCoCo. En la siguiente imagen se muestra el reporte generado por **JaCoCo**, accesible en build/reports/jacoco/test/html/index.html. Aquí se visualiza la cobertura detallada de cada paquete del proyecto. Los colores indican nivel de cobertura:

* Verde: 100% cubierto
* Amarillo: cobertura parcial
* Rojo (ausente): no cubierto

Este reporte evidencia que los paquetes model y repository cuentan con una cobertura sólida tanto en instrucciones como en ramas, mientras que config presenta un área de mejora.

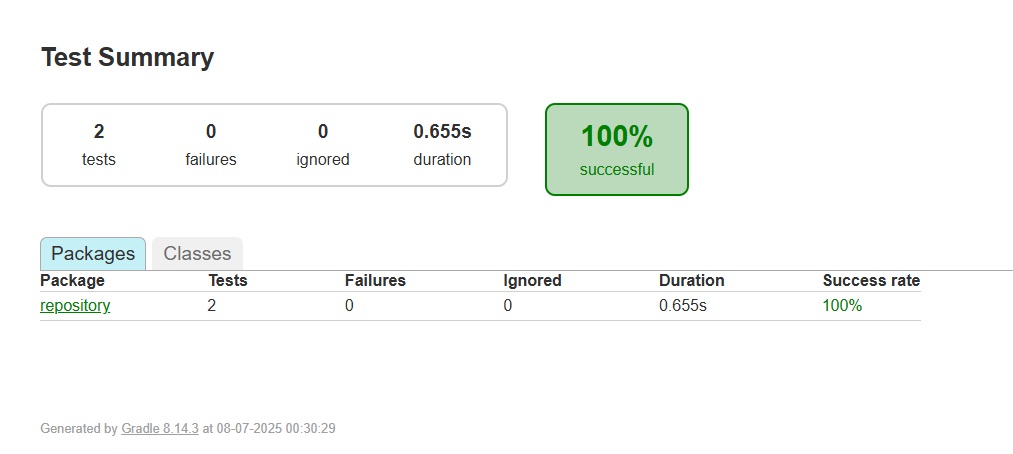


Pruebas ejecutadas exitosamente → (testEliminarItem) Test de actualización y eliminación de ítems ejecutado exitosamente.

La imagen muestra el archivo ItemRepositoryTest.java, donde se definen y ejecutan dos pruebas:

* testActualizarItem(): valida que los atributos de un Item sean modificados correctamente.
* testEliminarItem(): confirma que, al eliminar un elemento por su ID, el restante se conserva correctamente en el repositorio.

El terminal al pie refleja que ambas pruebas se ejecutaron sin errores, reforzando la confiabilidad del módulo repository. Este flujo representa con claridad una implementación completa del ciclo **Red → Green → Refactor** en contexto real.

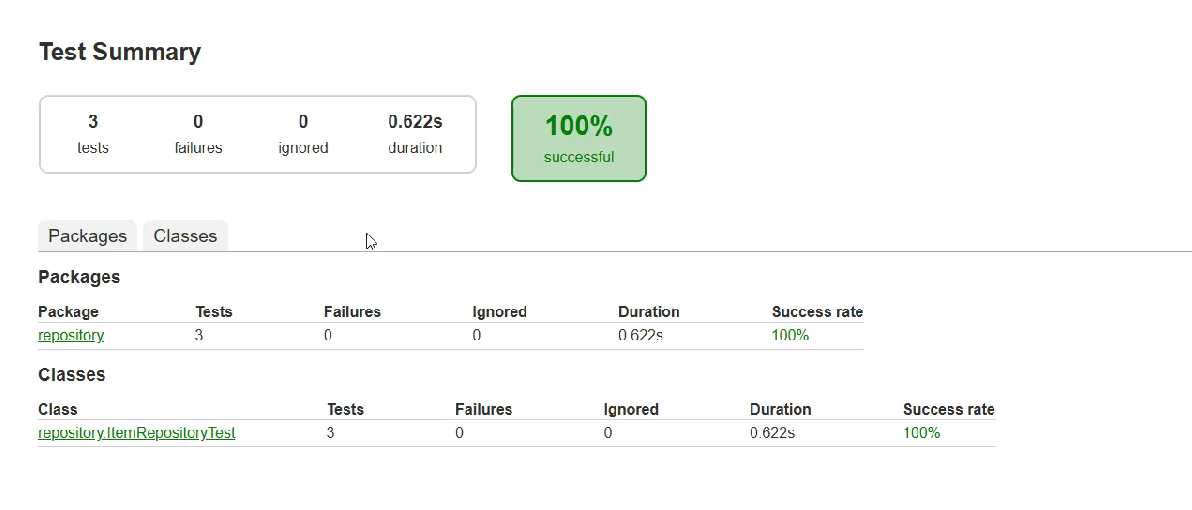


Resumen de pruebas del paquete repository → La imagen muestra el reporte de testeo generado tras ejecutar las pruebas unitarias con Gradle.

Se evidencia que:

* Se ejecutaron **3 pruebas** en total
* **0 fallos**, **0 pruebas ignoradas**
* Tiempo total: **0.622 segundos**
* **Éxito del 100%**

Este resumen corresponde a la clase repository.ItemRepositoryTest, validando que los métodos CRUD implementados en ese módulo están correctamente cubiertos por pruebas unitarias, lo que contribuye a la confiabilidad del sistema.

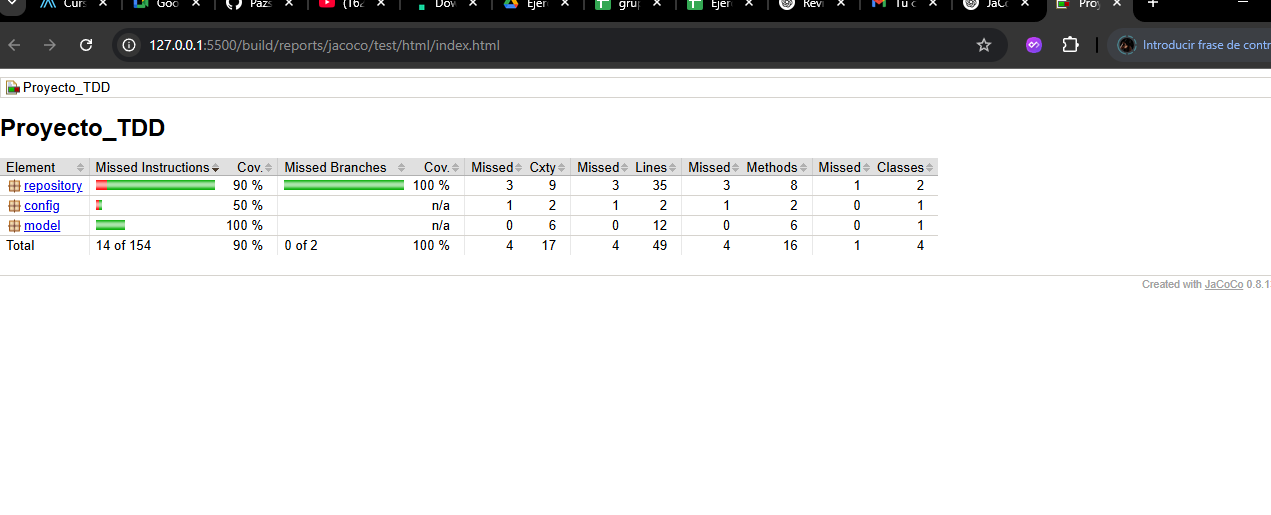


Reporte HTML de cobertura. → **reporte completo de cobertura de código** generado por **JaCoCo.**

La siguiente tabla fue obtenida desde el reporte generado por JaCoCo y resume métricas clave por paquete:

| **Paquete** | **Instrucciones** | **Ramas** | **Métodos** | **Clases** | **Complejidad** |
| --- | --- | --- | --- | --- | --- |
| model | 100% | N/A | 100% | 100% | 0 |
| repository | 90% | 100% | 87% | 67% | Media |
| config | 50% | — | 50% | 100% | Baja |

Este nivel de detalle permite identificar con claridad las áreas del proyecto que podrían ser reforzadas con nuevos tests, especialmente el paquete config.

****

Detalle de cobertura en paquete config. → Métricas detalladas por clase según el reporte JaCoCo.

El siguiente reporte generado por **JaCoCo** expone métricas específicas para las clases ItemService y ItemRepository, dos componentes esenciales del proyecto:

| **Clase** | **Instrucciones** | **Ramas** | **Métodos** | **Clases** |
| --- | --- | --- | --- | --- |
| ItemService | 100% | N/A | Missed: 3 | 1 no cubierta |
| ItemRepository | 90% | 100% | Missed: 5 | 1 no cubierta |

* ✅ ItemService presenta 100% de cobertura en instrucciones, aunque aún tiene métodos no probados (indicador de posibles responsabilidades acopladas).
* ItemRepository tiene una cobertura sólida, pero aún se detectan líneas y métodos sin ejecutar directamente en pruebas.

# **IX.** Observaciones Finales

- Dominio de herramientas esenciales de desarrollo backend.

- Excelente cobertura en model y repository, mostrando aplicación efectiva de TDD.

- Estructura modular y uso de Gradle listos para CI/CD.

- Consistencia en convenciones Java y nomenclatura.

# **X.** Próximos Pasos

- Refactorizar ItemService para mayor desacoplamiento.

- Incrementar cobertura en config con pruebas de inicialización.

- Agregar validaciones en modelos.

- Integrar GitHub Actions para testeo por commit.

# **XI.** Reflexión Personal

Este proyecto representó un verdadero punto de inflexión en mi proceso de aprendizaje como desarrolladora. Adoptar la metodología **Test Driven Development (TDD)** no solo transformó mi forma de escribir código, sino que también fortaleció mi capacidad para anticipar errores, diseñar con mayor claridad y confiar en la solidez de mis implementaciones.

Al comenzar con las pruebas antes de escribir cualquier lógica de negocio, aprendí a pensar en términos de **interfaces limpias, responsabilidades bien definidas y modularidad**, lo que me permitió mantener el enfoque durante el desarrollo. Cada ciclo **Red → Green → Refactor** fue una oportunidad para crecer tanto técnica como mentalmente, enfrentando el código desde una perspectiva más madura y profesional.

Uno de los aprendizajes más significativos fue entender que **las pruebas no son una carga**, sino una herramienta poderosa que habilita el cambio, la refactorización y la evolución del software sin miedo. Ver el reporte de cobertura, con altos porcentajes en los módulos más críticos, fue una validación tangible del esfuerzo invertido.

Además, este trabajo me hizo consciente de la importancia de herramientas como **JaCoCo**, **Mockito** y el uso de bases de datos en memoria como **H2**, que me permitieron simular entornos reales sin complicaciones adicionales.

En resumen, más allá de construir un sistema CRUD funcional, este proyecto me permitió desarrollar una nueva forma de pensar: más ordenada, estratégica y enfocada en la calidad.

# **XII. Referencias y Buenas Prácticas TDD**

- El ciclo Red → Green → Refactor impulsa diseño modular y calidad de código.

- Buenas prácticas: tests pequeños e independientes; refactor frecuente; pruebas positivas y negativas.

- TDD obliga a pensar en interfaces antes del código, mejorando mantenibilidad y desacoplamiento.

## 

## **XIII.** Conclusiones

La implementación inicial del sistema CRUD bajo la metodología **Test Driven Development (TDD)** permitió desarrollar funcionalidades básicas con una buena cobertura de pruebas, aunque aún existen oportunidades para mejorar y refactorizar.

La estructura modular y el desacoplamiento entre capas facilitan la evolución del proyecto, lo que permitirá perfeccionar la lógica y aumentar la calidad en próximas iteraciones.

Herramientas como **JUnit 5**, **Mockito** y **JaCoCo** aportaron un soporte fundamental para el control de calidad y monitoreo de cobertura.

La base de datos en memoria **H2** facilitó las pruebas automatizadas, permitiendo un desarrollo ágil y sin dependencias complejas.

La disciplina del ciclo **Red → Green → Refactor** fue clave para mantener un enfoque sistemático y preparar el terreno para futuras mejoras.

Este proyecto reafirma que el desarrollo guiado por pruebas es una práctica esencial para construir software confiable, y aunque el CRUD no está aún completamente perfeccionado, la planificación de refactorizaciones asegura un camino de mejora continua y calidad sostenible.