Lisset Dolores Sánchez Juárez | Academia- Xideral Fecha: 19 de septiembre de 2025

Galería-JUnit

Explicación

En este proyecto me enfoqué en probar la clase GaleriaServiceImpl mediante tres pruebas unitarias diferentes, utilizando el método assertEquals de JUnit. También, integré la anotación @BeforeEach para inicializar el servicio antes de cada prueba, evitando repetir código y haciendo que cada test comience con un estado limpio.
A continuación, se detalla cada una de ellas:

Test1- testAgregarYListarObras

Valida la funcionalidad básica de agregar y recuperar obras. Para ello, se crea un repositorio en memoria (ObraRepositoryImpl) junto con el servicio. Posteriormente, se agrega una pintura y se comprueba que la lista tenga exactamente un elemento. Además, se valida que el nombre de la primera obra sea "La noche estrellada".

Test 2- testAgregarVariasObras

Verifica que se pueden manejar múltiples obras sin pérdida de datos. En este caso, se agregan dos obras distintas (una pintura y una escultura) y se comprueba que la lista final contenga exactamente dos elementos.

Test 3- testDescripcionPolimorfica

Comprueba el polimorfismo en acción. Aunque la variable utilizada es del tipo genérico Obra, la salida del método descripcion() depende del tipo concreto de la obra (Pintura o Escultura). En este test, se agregan ambos tipos de obras y se verifica que las descripciones generadas sean diferentes según la clase específica.

Lisset Dolores Sánchez Juárez | Academia- Xideral Fecha: 19 de septiembre de 2025

Pintores-Mockito

Explicación

En este proyecto, para asegurar que la aplicación funciona correctamente, implementé pruebas unitarias juntando JUnit 5 y Mockito. Esto me permitió probar de manera controlada tanto la lógica de negocio como los endpoints de la API sin depender de una base de datos real o de peticiones externas. Se probaron principalmente la capa de servicio y la de controlador, asegurando que cada componente se comportara como se esperaba en distintos escenarios.

Pruebas del Servicio (PintorServiceTest)

En la capa de servicio, se simuló el repositorio de datos utilizando @Mock de Mockito, e inyectando este mock en el servicio con @InjectMocks. Se probaron métodos como save, findById, findAll y update. Con Mockito, se definieron comportamientos específicos mediante when(...).thenReturn(...) y al final se verificó que los métodos se llamaran la cantidad correcta de veces usando verify(...).

Pruebas del Controlador (PintorControllerTest)

En la capa de controlador, se simuló el servicio para probar los endpoints REST. El controlador se probó con @InjectMocks, lo que significa que se crea una instancia real del controlador, pero con el servicio simulado inyectado. Se verificaron todos los endpoints: GET, POST, PUT y DELETE. Por ejemplo, el GET /pintores/123 que devuelve un 200 OK con el objeto esperado, mientras que el GET /pintores/999 devuelve un 404 Not Found si el pintor no existe. Estas pruebas permitieron asegurar que tanto la lógica como las respuestas de la API son correctas y consistentes.

Lisset Dolores Sánchez Juárez | Academia- Xideral Fecha: 19 de septiembre de 2025

Películas-SpingBatch

Explicación

En este proyecto la temática elegida fue un catálogo de películas, cuyo objetivo principal es leer información desde un archivo CSV que contiene datos de diferentes películas, filtrarlas por un criterio específico (en este caso, conservar únicamente aquellas con un rating mayor o igual a 7.0) y finalmente almacenarlas en una base de datos MySQL.

Estructura y funcionamiento

El sistema sigue la arquitectura típica de Spring Batch, que se compone regularmente de cuatro elementos principales:

1. El ItemReader. Este se encarga de leer los datos de un archivo CSV que contiene más de 100 registros de películas, incluyendo campos como título, género, director, año de estreno y calificación.
2. El ItemProcessor. En esta etapa se procesan los datos para aplicar una regla de negocio. En este caso, solo se guardan en la base de datos las películas que tienen una calificación (rating) mayor o igual a 7.0.
3. El ItemWriter. Finalmente, las películas que pasaron el filtro son escritas en una tabla de MySQL llamada MOVIES.
4. Job y Steps: Todo el flujo está controlado por un Job que ejecuta un Step, el cual define la lectura, procesamiento y escritura en bloques o chunks de 10 registros para optimizar el rendimiento

Ejecución y resultados

La aplicación expone un endpoint REST que permite iniciar el proceso de importación de las películas.
Al enviar una petición POST a http://localhost:9191/movies/jobs/importMovies, el Job comienza a ejecutarse de manera controlada.
Spring Batch lee el archivo CSV, filtra los datos según la calificación y guarda únicamente las películas válidas en la base de datos MySQL configurada en application.properties.
Gracias a la integración con Spring Data JPA, la creación de la tabla MOVIES se realiza automáticamente, sin necesidad de escribir las sentencias SQL manualmente.