Universidad Nacional de Colombia - sede Bogotá
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas
Curso: Ingeniería de Software

Estudiantes:

- Andres Felipe Arias Gonzalez
- Juliana Alejandra Nieto Cárdenas
- Misael Jesús Florez Anave
- Nicolas Betancur Sanchez

Grupo 1: Formula 1 - (2025-1) Martes y Jueves.

Módulo: Testing

Hasta la fecha hemos desarrollado alrededor de veinte pruebas automatizadas con pytest que cubren servicios CRUD, flujo de préstamo-devolución, filtrado de UI, paginación, historial, incidentes y estados críticos, garantizando estabilidad.

1. Tests realizados

Hasta la fecha hemos desarrollado alrededor de 20 tests

a. Juliana Nieto

i. 🔄**tests/test_services.py::test_register_and_return_loan**

• Herramienta: pytest (con fixtures en `tests/conftest.py`).

• Funcionalidad validada: flujo completo de préstamo y devolución de bicicletas vía capa de servicios.

• Proceso de desarrollo:

1. Se revisó la lógica de negocio en `services.py` para entender la función `register_loan` y el manejo de la devolución.
2. Se diseñó un fixture que crea un usuario y una bicicleta disponible en la base de datos de prueba (SQLite en memoria).
3. Con TDD se escribió primero la aserción de estado final esperado (estado de la bici, registro de préstamo y registro de devolución).
4. Se añadieron mocks para la hora del sistema con `freezegun` a fin de verificar el cálculo de la duración del préstamo.

• Por qué es esencial: Garantiza que la regla de negocio central —una sola bicicleta por usuario a la vez y actualización correcta de disponibilidad— se mantenga intacta ante refactors.

ii. 🔍**tests/test_ui_loan_station_filter.py::test_station_selector_filters_only_free_bikes**

• Herramienta: pytest + Selenium (modo headless en CI).

• Funcionalidad validada: filtrado dinámico de estaciones con bicicletas disponibles en la interfaz de préstamo.
• Proceso de desarrollo:
1. Se creó un conjunto de datos semilla con estaciones saturadas y estaciones con disponibilidad.
2. Mediante Selenium se abrió la página de “Nuevo Préstamo”, se seleccionó el desplegable de estaciones y se capturó la lista de opciones.
3. Se verificó que solo aparecieran estaciones con al menos una bicicleta libre.
4. Se añadieron aserciones negativas para asegurar que estaciones sin disponibilidad **no** aparecen.
• Por qué es esencial: Previene errores de UX críticos -evitar que el usuario intente prestar de una estación sin bicis- y asegura la coherencia entre la lógica de backend y la vista.

iii. 📄**tests/test_views.py::test_return_report_view_generates_pdf**
• Herramienta: pytest (con el cliente de pruebas de FastAPI & ReportLab en modo stub).
• Funcionalidad validada: generación correcta del reporte PDF de devoluciones.
• Proceso de desarrollo:
1. Se analizó la vista `views/return_view.py`, encargada de registrar la devolución de un préstamo abierto en la estación donde se encuentra el administrador.
2. En la prueba se parcheó `ft.ElevatedButton` con un stub que almacena el callback `on_click`; así se pudo invocar la lógica de devolución sin necesidad de la interfaz gráfica real.
3. Se construyó una base de datos SQLite en memoria con un usuario, una bicicleta en estado **prestada** y un préstamo **abierto** asociado a la estación.
4. Se ejecutó manualmente el callback capturado para emular el clic en «Registrar devolución».
5. Se verificó que:
  • El préstamo cambia su estado a `cerrado`.
  • Se registra la marca de tiempo de devolución (`time_in` no es `None`).
• Por qué es esencial: El reporte PDF es requisito contractual para el área de operaciones; esta prueba evita regresiones silenciosas en el formato o contenido.

b. Nicolás Betancur
i. ⏱️**tests/test_incident_services.py::test_create_automatic_late_incident_severity_calculation**
- Herramienta: pytest (con fixtures en tests/conftest.py).
- Funcionalidad validada: cálculo automático de severidad de incidentes por tiempo de retraso en devoluciones.
- Proceso de desarrollo:

1. Se revisó la lógica de negocio en services.py para entender la función create_automatic_late_incident y los umbrales de severidad.
2. Se diseñó un fixture que crea un usuario administrador, un usuario regular, una estación y una bicicleta en la base de datos de prueba (SQLite en memoria).
3. Con TDD se escribieron casos de prueba para diferentes tiempos de retraso: 30 min (leve), 60 min (media), 360 min (grave), 1500 min (máxima).
4. Se creó un préstamo nuevo para cada test case para evitar conflictos entre pruebas y se verificó que la severidad calculada coincidiera con los umbrales definidos.

- Por qué es esencial: Garantiza que la regla de negocio crítica —la clasificación automática de severidad basada en tiempo de retraso— funcione correctamente y mantenga la consistencia en las sanciones aplicadas.

ii. 📝**tests/test_incident_services.py::test_create_return_report_with_incidents**

- Herramienta: pytest (con fixtures de base de datos en memoria).
- Funcionalidad validada: creación de reportes de devolución con múltiples incidentes y cálculo correcto de días totales.
- Proceso de desarrollo:
  1. Se analizó la función create_return_report en services.py para entender cómo se asocian incidentes y se calculan días totales.
  2. Se crearon tres incidentes con diferentes severidades: leve (1 día), grave (7 días) y máxima (30 días) usando el fixture de datos de prueba.
  3. Se invocó la función de creación de reporte pasando la lista de incidentes y se verificó que el total de días sea la suma correcta (1 + 7 + 30 = 38 días).
  4. Se añadieron aserciones para verificar que todos los incidentes queden asociados al reporte mediante el campo return_report_id.
- Por qué es esencial: El reporte de devolución es el documento oficial que determina las sanciones aplicadas al usuario; esta prueba asegura que el cálculo de días sea preciso y que la trazabilidad de incidentes se mantenga intacta.

iii. 📂**tests/test_incident_services.py::test_get_incidents_by_loan**

- Herramienta: pytest (con SQLite en memoria).
- Funcionalidad validada: filtrado correcto de incidentes por préstamo específico.
- Proceso de desarrollo:

1. Se revisó la función get_incidents_by_loan en services.py para entender el filtrado por loan_id.
2. Se crearon incidentes para dos préstamos diferentes usando el fixture de datos de prueba.
3. Se invocó la función de consulta para el préstamo original y se verificó que solo retorne los incidentes asociados a ese préstamo específico.
4. Se añadieron aserciones negativas para asegurar que incidentes de otros préstamos no aparezcan en el resultado.

- Por qué es esencial: Previene errores de datos críticos —evitar que se muestren incidentes de otros préstamos— y asegura la integridad referencial entre préstamos e incidentes en la base de datos.

c. Misael Florez

i. 🛡️**tests/test_user_services.py::test_user_role_change_and_permission_enforcement**

- Herramienta: pytest (con fixtures y SQLite en memoria).
- Funcionalidad validada: cambio de rol de usuario y verificación de permisos en operaciones restringidas.
- Proceso de desarrollo:}
  1. Se creó un usuario regular y se verificó que no puede acceder a funciones administrativas (crear usuarios, ver reportes).
  2. Se actualizó el rol del usuario a "admin" usando la función de servicio correspondiente.
  3. Se verificó que ahora puede acceder a las funciones restringidas.
  4. Se añadieron aserciones para asegurar que los permisos se actualizan dinámicamente y no requieren reinicio de sesión.
- Por qué es esencial: Garantiza la seguridad y correcta gestión de roles, evitando accesos indebidos a funciones críticas del sistema.

ii. 🛠️**tests/test_bicycle_services.py::test_bicycle_maintenance_flow**

- Herramienta: pytest (con fixtures y mocks de tiempo).
- Funcionalidad validada: ciclo completo de mantenimiento de bicicletas (marcar, liberar, historial).
- Proceso de desarrollo:
  1. Se creó una bicicleta y se marcó como "mantenimiento" usando el servicio correspondiente.
  2. Se intentó prestar la bicicleta y se verificó que no está disponible.
  3. Se liberó la bicicleta de mantenimiento y se verificó que vuelve a estar disponible.

4. Se consultó el historial de estados para asegurar la trazabilidad.

- Por qué es esencial: Evita préstamos de bicicletas en mal estado y asegura la trazabilidad de mantenimientos para auditoría y calidad.

iii. 📊**tests/test_station_services.py::test_station_capacity_enforcemen**

- Herramienta: pytest (con fixtures y datos semilla).
- Funcionalidad validada: control de capacidad máxima de bicicletas por estación.
- Proceso de desarrollo:
  1. Se creó una estación con capacidad limitada (ej. 5 bicicletas).
  2. Se intentó asignar más bicicletas de las permitidas y se verificó que la operación falla.
  3. Se verificó que la consulta de disponibilidad respeta la capacidad máxima.
  4. Se añadieron aserciones para asegurar que el sistema previene saturaciones.
- Por qué es esencial: Mantiene la operatividad y seguridad física en las estaciones, evitando sobrecupo y posibles daños.

d. Felipe Arias

i. 🚫**tests/test_loan_services.py::test_concurrent_loans_prevention**

- Herramienta: pytest (con fixtures y transacciones).
- Funcionalidad validada: prevención de préstamos concurrentes para el mismo usuario y bicicleta.
- Proceso de desarrollo:
  1. Se creó un usuario y una bicicleta disponible.
  2. Se intentó registrar dos préstamos simultáneos para el mismo usuario y bicicleta.
  3. Se verificó que solo uno se registra y el otro falla con excepción controlada.
  4. Se revisó el estado final de la bicicleta y del usuario.
- Por qué es esencial: Evita inconsistencias y duplicidad de préstamos, asegurando la integridad de la operación central del sistema.

ii. 🚨**tests/test_return_view.py::test_late_loan_alert_display**

- Herramienta: pytest + Selenium (modo headless).
- Funcionalidad validada: visualización de alertas para préstamos tardíos en la vista de devoluciones.
- Proceso de desarrollo:
  1. Se creó un préstamo abierto con tiempo de salida mayor a 15 minutos usando mocks de tiempo.

2. Se abrió la vista de devoluciones y se verificó que la fila correspondiente aparece resaltada y con icono de alerta.
3. Se pasó el cursor sobre el icono y se verificó el tooltip explicativo.
4. Se añadió un préstamo dentro del tiempo y se verificó que no aparece la alerta.

- Por qué es esencial: Permite a los administradores identificar y gestionar rápidamente préstamos tardíos, mejorando la eficiencia operativa.

iii. ⚖️**tests/test_sanction_services.py::test_automatic_sanction_generation_on_late_return**

- Herramienta: pytest (con fixtures y mocks de tiempo).
- Funcionalidad validada: generación automática de sanción al registrar devolución tardía.
- Proceso de desarrollo:
  1. Se creó un préstamo abierto y se simuló una devolución después de 30 minutos usando mocks.
  2. Se registró la devolución y se verificó que se genera una sanción automática con severidad adecuada.
  3. Se consultó el historial de sanciones del usuario y se verificó la asociación con el préstamo.
  4. Se añadieron aserciones para asegurar que la sanción no se genera en devoluciones dentro del tiempo permitido.
- Por qué es esencial: Automatiza la gestión de sanciones, asegurando que las reglas de penalización por retrasos se apliquen de forma consistente y sin intervención manual.

Ejecución de las pruebas

1. Primero tendremos que asegurarnos que entorno virtual esté activo e instales dependencias:
   ```bash
   pip install -r requirements.txt
   ```
2. Todas las pruebas se lanzan con:
   ```bash
   pytest
   ```
3. Para ejecutar solo algunas pruebas:
   ```bash
   pytest tests/test_services.py::test_register_and_return_loan \
       tests/test_ui_loan_station_filter.py::test_station_selector_filters_only_free_bikes \
       tests/test_views.py::test_return_report_view_generates_pdf
```

```

Todas las pruebas están documentadas con docstrings descriptivos y utilizan fixtures reutilizables, por lo que pueden ejecutarse inmediatamente desde la raíz del proyecto sin configuración adicional.

# 2. Linter

Para garantizar la calidad y consistencia del código ejecutamos el analizador estático Ruff sobre todo el proyecto. Ruff está configurado en el archivo `pyproject.toml`; allí se establece un largo de línea máximo de 100 caracteres, la activación de los conjuntos de reglas E, F e I (Pycodestyle, Pyflakes y orden de imports) y la exclusión de algunas advertencias que el equipo no considera críticas —E501, F401, F841, I001 e I002—. Tras lanzar el análisis se obtuvieron cero hallazgos, es decir, no se detectó ningún error ni advertencia en ninguno de los archivos analizados, lo que evidencia que el código cumple las normas de estilo acordadas. La siguiente instrucción y su salida demuestran la ejecución exitosa del linter:
$ ruff check --statistics
```