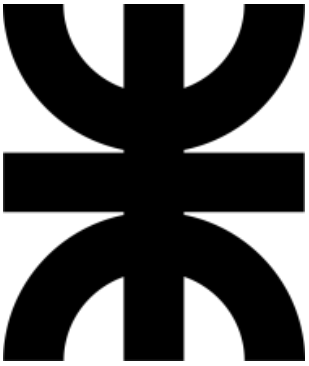
Universidad Tecnológica Nacional

Facultad Regional Córdoba

Ingeniería en Sistemas de Información



Documento de Implementación TDD - User Story: Comprar Entradas

Unidad: 04 - Aseguramiento de Calidad de Proceso y de Producto
Año: 2025
Curso: 4k1
Grupo Nro: 07
Integrantes:

Adragna, Jimena Sofía	94269
Baigorria, José Alejo	96269
Buchaillot, Julieta	95782
Depetris Mayne, Agustín	92141
Garcia, Florencia Daniela	94477
Lucini, Gabriel Alejandro	98023
Martinet, Agustina María Andrea	94674
Milhas Mac Dougall, Mariana	95257
Zeheiri, Micaela	91135

Índice de contenidos

Especificación técnica general

User Story: Comprar entradas (grupo impar)

Backend:

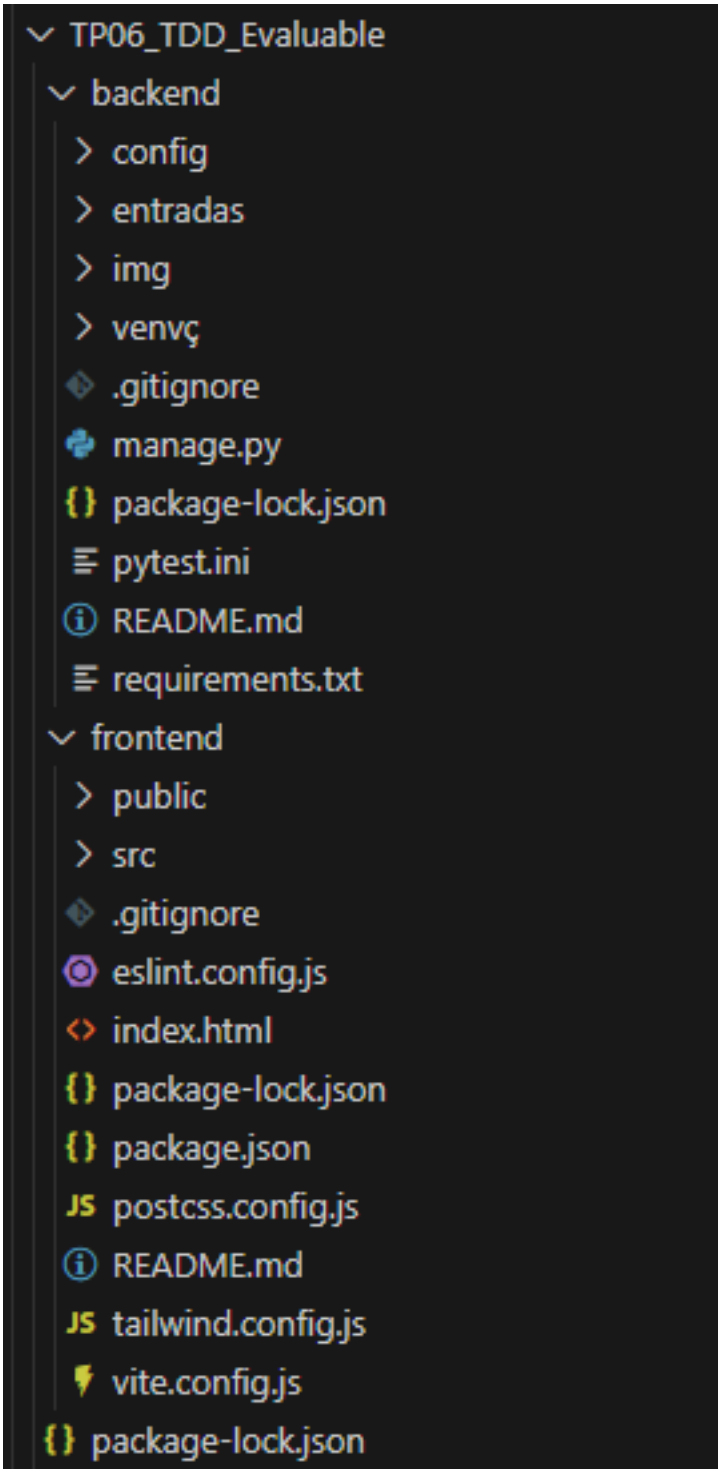
- **Lenguaje:** Python
- **Guía de Estilo:** PEP 8
- **Framework de Testing:** pytest

Frontend:

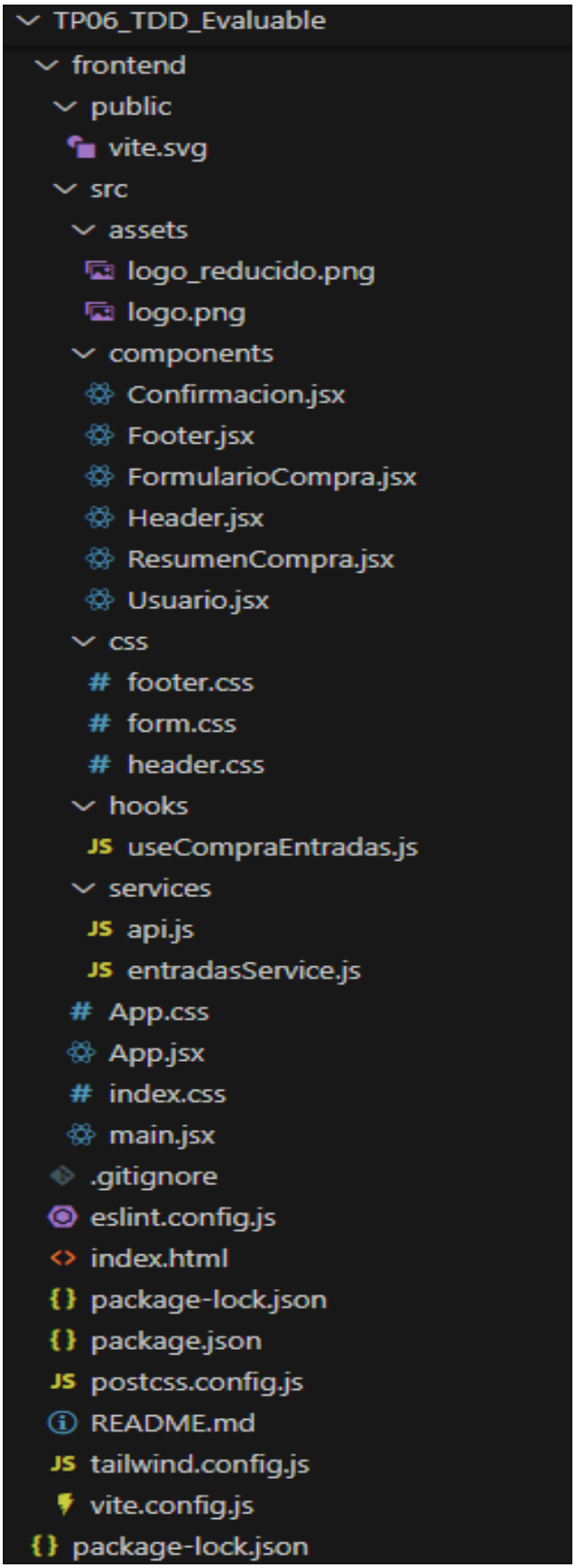
- **Lenguaje:** JavaScript
- **Guía de Estilo:** Airbnb React/JSX Style Guide

Estructura del Proyecto

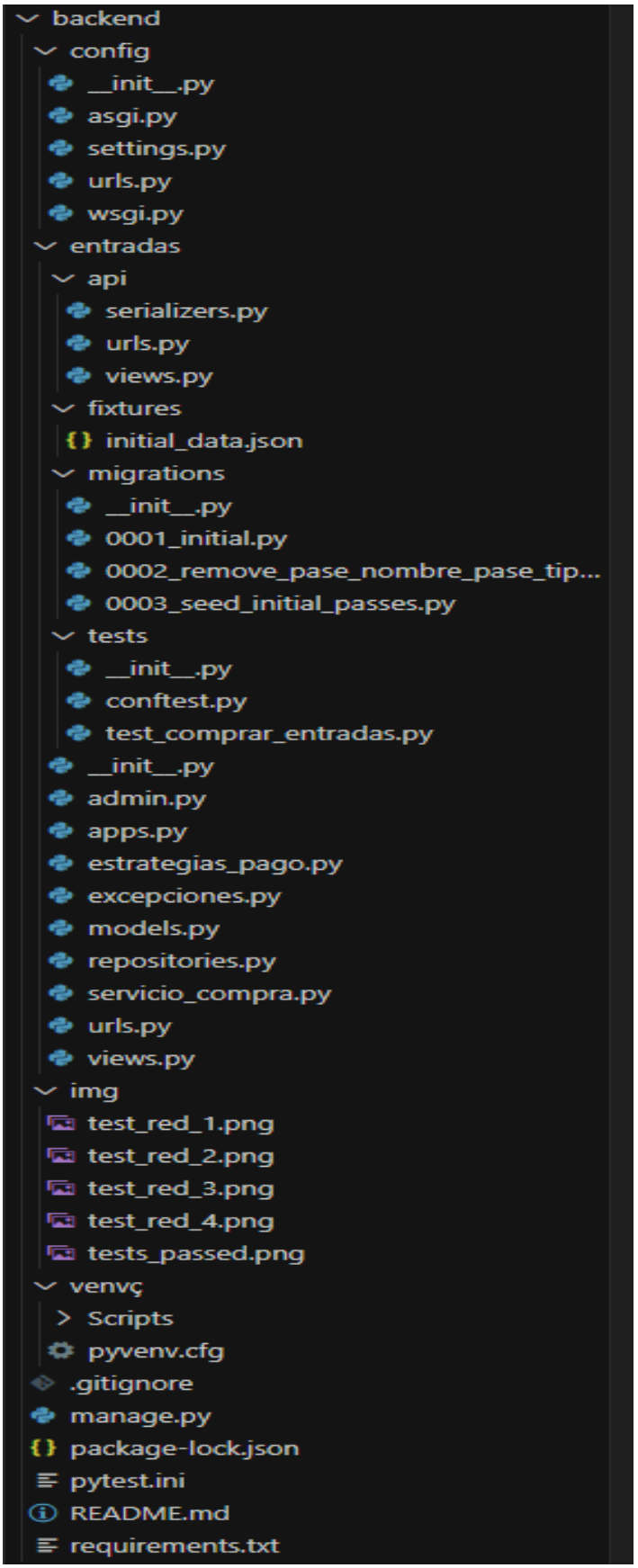
Vista general



Frontend


TP06_TDD_Evaluable
frontend
public
vite.svg
src
assets
logo_reducido.png
logo.png
components
Confirmacion.jsx
Footer.jsx
FormularioCompra.jsx
Header.jsx
ResumenCompra.jsx
Usuario.jsx
css
footer.css
form.css
header.css
hooks
useCompraEntradas.js
services
api.js
entradasService.js
App.css
App.jsx
index.css
main.jsx
.gitignore
eslint.config.js
index.html
package-lock.json
package.json
postcss.config.js
README.md
tailwind.config.js
vite.config.js
package-lock.json

Backend


backend
config
__init__.py
asgi.py
settings.py
urls.py
wsgi.py
entradas
api
serializers.py
urls.py
views.py
fixtures
initial_data.json
migrations
__init__.py
0001_initial.py
0002_remove_pase_nombre_pase_tip...
0003_seed_initial_passes.py
tests
__init__.py
conftest.py
test_comprar_entradas.py
__init__.py
admin.py
apps.py
estrategias_pago.py
excepciones.py
models.py
repositories.py
servicio_compra.py
urls.py
views.py
img
test_red_1.png
test_red_2.png
test_red_3.png
test_red_4.png
tests_passed.png
venvç
Scripts
pyvenv.cfg
.gitignore
manage.py
package-lock.json
pytest.ini
README.md
requirements.txt

Ciclo TDD Aplicado

Fase 1: RED - Definir modelo y excepciones

El archivo test_compra_entradas.py contiene un conjunto exhaustivo de pruebas para la User Story "Comprar Entradas", desarrolladas para guiar la implementación del código correspondiente. Las pruebas fueron diseñadas para fallar inicialmente (fase roja) y cubren tanto las unidades lógicas internas como los flujos de integración principales.

Enfoque y Herramientas

- **Justificación de herramientas:** Se eligió pytest para la estructura y ejecución de las pruebas debido a su sintaxis concisa y su potente sistema de Fixtures (@pytest.fixture). Este sistema nos permite definir y reutilizar datos de prueba consistentes (datos_compra_validos, usuario_valido_mock) y simulaciones de servicios (mocks_infraestructura) de manera modular.
- **Decisión de aislamiento (Mocking):** Se decidió simular las dependencias externas (Pasarela de Pagos, Servicio de Correo, Servicio de Calendario) usando MagicMock y Mock. Esta decisión es fundamental para lograr el aislamiento en las pruebas unitarias y de integración, asegurando que las pruebas sean rápidas, determinísticas (independientes de servicios externos) y se enfoquen exclusivamente en la lógica del ServicioCompraEntradas.
- **Estrategia de Cobertura (Unitaria vs. Integración):** Se optó por una estrategia de pruebas mixta. Se justifican las pruebas unitarias para métodos internos (_validar_cantidad, _validar_fecha_hora_visita, etc.) porque permiten validar reglas de negocio específicas y complejas de forma aislada y precisa. Las pruebas de integración (sobre comprar_entradas) son necesarias para asegurar que todos estos componentes colaboran correctamente en el flujo principal definido por la User Story.

Cobertura de pruebas

Las pruebas fueron diseñadas no solo para cubrir el "camino feliz", sino deliberadamente para validar los casos límite y de error, que es donde el TDD guía el diseño del manejo robusto de excepciones y reglas de negocio.

- **Validación de cantidad y estructura:** Las pruebas (ej. cantidad > 10, cantidad != len(visitantes), edad < 0) fueron diseñadas para asegurar que la implementación sea robusta contra datos de entrada malformados o inválidos, devolviendo las excepciones de negocio apropiadas (LimiteEntradasExcedidoError, ValueError).
- **Validación de reglas de negocio temporales (fecha y hora):** Se crearon pruebas específicas para los límites exactos de horario (ej. 8:59:59 vs 9:00:00) y días

especiales (Lunes, feriados). El propósito de esta granularidad es garantizar que la lógica de validación de fechas sea precisa y maneje correctamente todos los escenarios definidos.

- **Validación de lógica de montos:** La cobertura exhaustiva del cálculo de montos, incluyendo todos los límites de edad (0, 2, 3, 9, 10, 60, 61), garantiza que la lógica de descuentos por edad esté implementada correctamente para todas las categorías.
- **Validación de flujos de pago:** Se diseñaron pruebas específicas para "Efectivo" y "Tarjeta" para asegurar que el sistema maneje correctamente los flujos condicionales (ej. no llamar a la pasarela si es efectivo) y los posibles fallos (pago rechazado).
- **Validación de usuario y datos de entrada:** Pruebas adicionales aseguran que el sistema maneje correctamente usuarios no registrados, emails inválidos, tipos de pase incorrectos, datos faltantes y tipos de datos incorrectos (ej. edad como string), forzando una validación de entrada completa.
- **Verificación del proceso completo (Integración):** Los tests de integración sirven para validar que el método público comprar_entradas orquesta correctamente las llamadas a los métodos internos y dependencias externas (mockeadas), devolviendo el resultado esperado o la excepción apropiada según el escenario.

Desglose de los Tests

El total se divide en dos categorías principales:

1. Pruebas de Integración

Estos tests verifican el flujo completo de la función principal comprar_entradas, incluyendo las interacciones con los mocks de los servicios externos (pagos, correo, calendario) y el manejo de fallos cruciales:

Ejemplos:

- test_comprar_entradas_flujo_completo_tarjeta_exitoso
- test_comprar_entradas_pago_tarjeta_rechazado_falla
- test_comprar_entradas_usuario_no_registrado_falla

En total tenemos nueve tests de integración.

2. Pruebas Unitarias

Estos tests se centran en verificar la lógica de negocio y las validaciones internas del servicio, aislando cada componente:

Subsección	Cantidad
Cálculo de Precios y Montos	30
Validación de Fecha y Hora de Visita	12
Validación de Formato de Edades	7
Validación de Formato de Usuario	8
Validación de Formato de Fecha y Hora	5
Validación de Formato de Cantidad de Entradas	5
Validación de Formato de Pases	5
Validación de Forma de Pago	9
Validación de Envío de Confirmación Vía Mail	3
Validación de Valor de Pases	2
Validación de Usuario Registrado	2
Validación de Cantidad de Entradas	6
Total Unitarias	**94**

Figura Nº1 “Tabla de pruebas unitarias”

Fase 2: Green - Implementación inicial del servicio

Implementamos el mínimo código necesario para que pasen los tests, principalmente definiendo y escribiendo el código de las funciones utilizadas en los tests previamente programados.

Fase 3: Refactor - Mejorar el diseño

Refactorización para mejor legibilidad y mantenibilidad del código. Además buscar mejorar el rendimiento y velocidad de ejecución del código.

Ejecución de Pruebas

Configuración para ejecutar las pruebas:

Instalar dependencias

En el README.md de la carpeta backend/ se indica como instalar lo necesario

Ejecutar todas las pruebas

pytest tests/test_compra_entradas.py -v

Ejecutar pruebas específicas

pytest tests/test_compra_entradas.py::TestTicketSystem::test_validar_cantidad_mayor_a_10_falla -v

Con cobertura

pytest tests/test_compra_entradas.py --cov=src

Justificación de Decisiones de Diseño

1. Arquitectura en Capas

Modelo: Clases de dominio y excepciones específicas.

Servicio: Lógica de negocio y coordinación.

Infraestructura: Servicios externos (pagos, email, calendario).

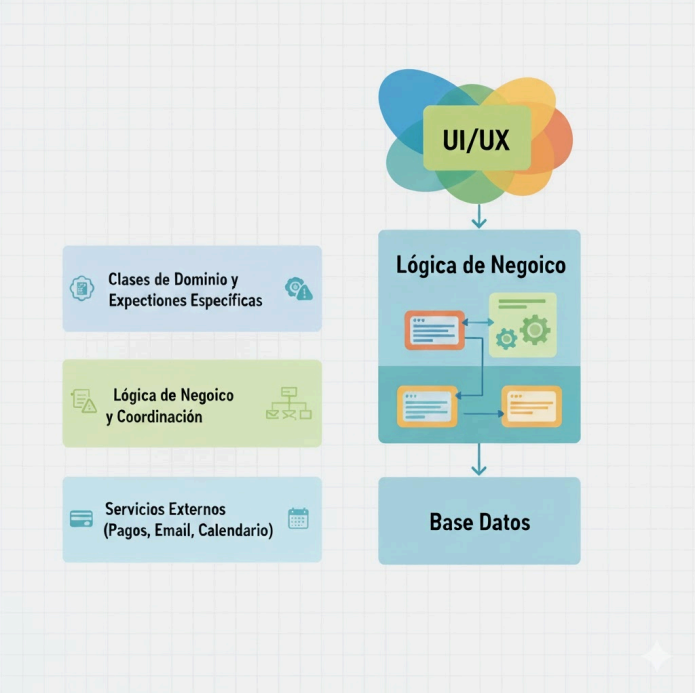


Figura Nº2: “Arquitectura en capas”

2. Principios SOLID Aplicados

Single Responsibility: Cada método tiene una única responsabilidad

Open/Closed: Fácil de extender con nuevos tipos de pase o descuentos

Liskov Substitution: Comportamiento predecible en herencia

Interface Segregation: Servicios externos acoplados débilmente

Dependency Inversion: Dependencias inyectadas en constructor

3. Patrones de Diseño

Service Layer: ServicioCompraEntradas como fachada principal

Strategy: Diferentes estrategias de cálculo de precios

Factory: Creación de objetos Compra

Observer: Notificaciones por email después de la compra

4. Manejo de Errores

Excepciones específicas: Cada error del dominio tiene su excepción

Validaciones tempranas: Fallar rápido con mensajes claros.

Jerarquía de excepciones: Desde errores de entrada hasta errores de sistema.

Esta estrategia de manejo de errores se justifica porque prioriza la claridad diagnóstica y la robustez del sistema. Al usar excepciones específicas, se permite que el código que captura el error (el handler) sepa exactamente qué salió mal (ej. ParqueCerradoError en lugar de un genérico ValueError), facilitando la lógica de recuperación o la presentación de mensajes precisos al usuario. Las validaciones tempranas garantizan el principio de "fallar rápido", evitando la ejecución de lógica costosa o compleja si una condición de entrada básica ya es incorrecta, lo que reduce el procesamiento innecesario del sistema, mejorando su rendimiento y evitando que se pueda entrar en estados inconsistentes por errores a media ejecución.

5. Cumplimiento PEP 8

Elegimos seguir la guía de estilo para el código del backend la guía de estilo PEP8 ya que es el estándar más difundido en la comunidad de Python, tanto así que es lo que se sigue en las librerías estándar de Python. Destaca por su alta consistencia y legibilidad lo cuál simplifica su lectura y disminuye el tiempo de compresión del mismo.

Algunos ejemplos de los puntos que seguimos son:

- Uso de sangría tabulación (representativa de 4 espacios) para mayor legibilidad tanto del código como de las indentaciones
- Nombres descriptivos: _calcular_precio_entrada en lugar de calc_precio
- Docstrings completos: Documentación Google-style
- Type hints: Tipado en métodos públicos
- Line length: 79 caracteres respetados
- Organización imports: Agrupados y ordenados según si son de librerías estándar de terceros o específicas locales respectivamente.
- Uso de cadenas de documentación para funciones: escritas luego de la definición de la función
- Uso de CapWords para los nombres de las clases y excepciones y el uso de el sufijo Error en estos últimos
- Captura de excepciones específicas siempre que se pueda por encima de una captura general.

6. Testabilidad

Inyección de dependencias: Mocks fáciles de proporcionar

Métodos pequeños: Fáciles de probar unitariamente

Separación concerns: Lógica separable para testing

Estas características nos permiten hacer un testing legible, comprobable y de rápida ejecución permitiéndonos identificar rápidamente puntos de falla del código para su pronta corrección.

7. Cumplimiento Airbnb React/JSX Style Guide

Elegimos seguir la guía de estilo para el código del frontend Airbnb React/JSX Style Guide ya que es el estándar más difundido en la comunidad de React, sirviendo como la configuración base para numerosas empresas y proyectos a gran escala. También encamina hacia código de gran consistencia y legibilidad, lo cual simplifica la lectura de los componentes, disminuye el tiempo de comprensión del código, y permite la aplicación rigurosa de buenas prácticas y patrones idiomáticos de React a través de herramientas de linting como ESLint, lo que resulta en un frontend más robusto y mantenible.

Algunos ejemplos de puntos que seguimos son:

- Un componente React por archivo
- Reglas de nombrado como la extensión .jsx para los componentes React, el uso de PascalCase para los nombres de archivos y camelCase para las instancias de los componentes
- Usar un espacio para el cierre de etiquetas, y que si estas son de tipo self-close (`<Foo variant="stuff" />`) no tengan un cierre en etiquetado aparte (que no se cierre así: `<Foo variant="stuff"></Foo>`)

8. Consultas al Product Owner

Algunas consideraciones hechas en la lógica y presentación de la funcionalidad se deben a consultas, y las respectivas respuestas, que se le han hecho al PO designado por el parque (en este caso simulado por los profesores).

A continuación detallamos las preguntas y respuestas:

- **¿Tipo de letra para el frontend?**

  Montserrat.

- **¿Qué paleta de color usar?**

https://coolors.co/134611-3e8914-3da35d-96e072-e8fccf

- **¿Logo del parque?**

  Diseño nuestro y enviarlo para validación.

- **¿Qué días el parque está abierto?**

  Abre todos los días, menos los lunes y días festivos (25 de diciembre y 1 de enero). El parque abre de 9:00hs a 19:00hs.

- **¿La edad se carga por rango?**

  Se carga como un entero.

- **¿Precio de las entradas?**

  $10000 VIP¨y $5000 regular. Los menores de 3 no pagan, los menores a 10 pagan la mitad y los mayores a 60 pagan la mitad

- **Confirmación del logo del parque**

- **¿La categoría de las entradas en una compra es para todas la misma o son independientes?**

  El tipo es por cada entrada, en una compra puede comprarse algunas entradas vips y otras regular.

- **¿En la pantalla de compra se deben mostrar los datos del usuario?**

  No hace falta, pueden mockear el nombre de usuario logueado arriba a la derecha.

- **Al confirmar, ¿Se debe enviar un mail con los datos generados?**

  No hace falta, con un mensaje que informe los datos de la compra basta.

- **¿El sistema debe verificar que el usuario que realiza la compra sea mayor de edad? Y, en caso de compra de entradas, ¿debería requerirse al menos un ticket asociado a un adulto?**

  No se habla en el dominio, ni en la US sobre eso. Así que no hay problema.

Vista de la implementación en pantallas

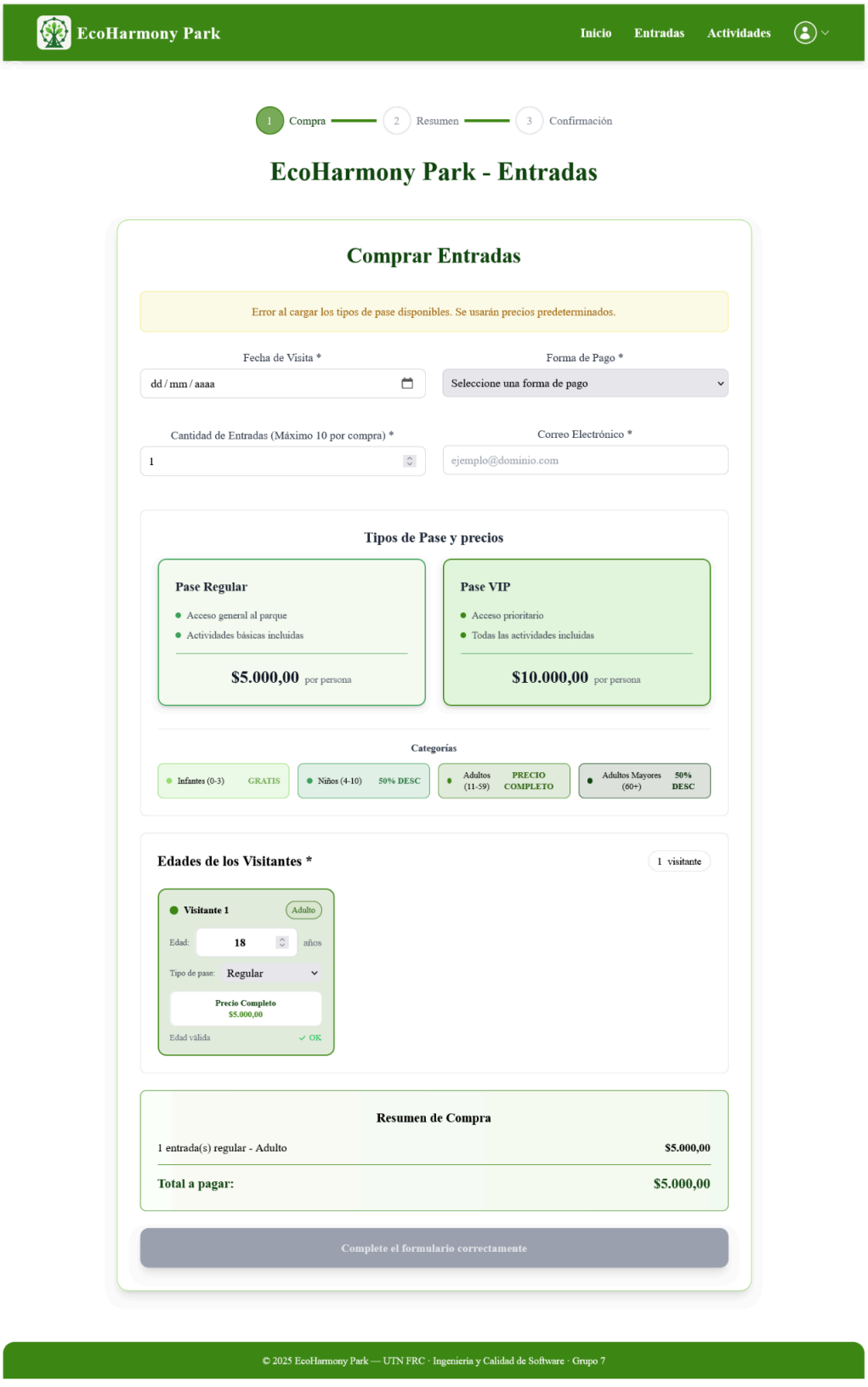


Figura Nº3: “Pantalla completa de compra de entradas”

Figura Nº4: “Logo del parque”

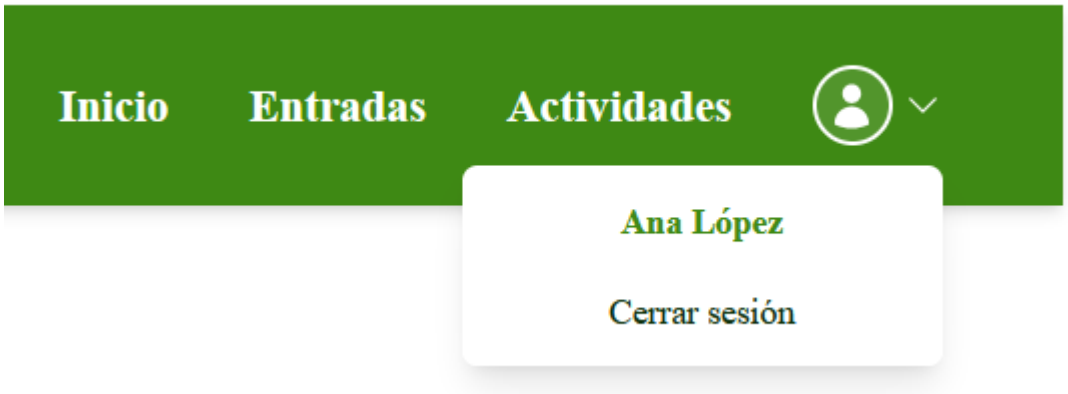


Figura Nº5: “Desplegable de usuario en el menú superior”

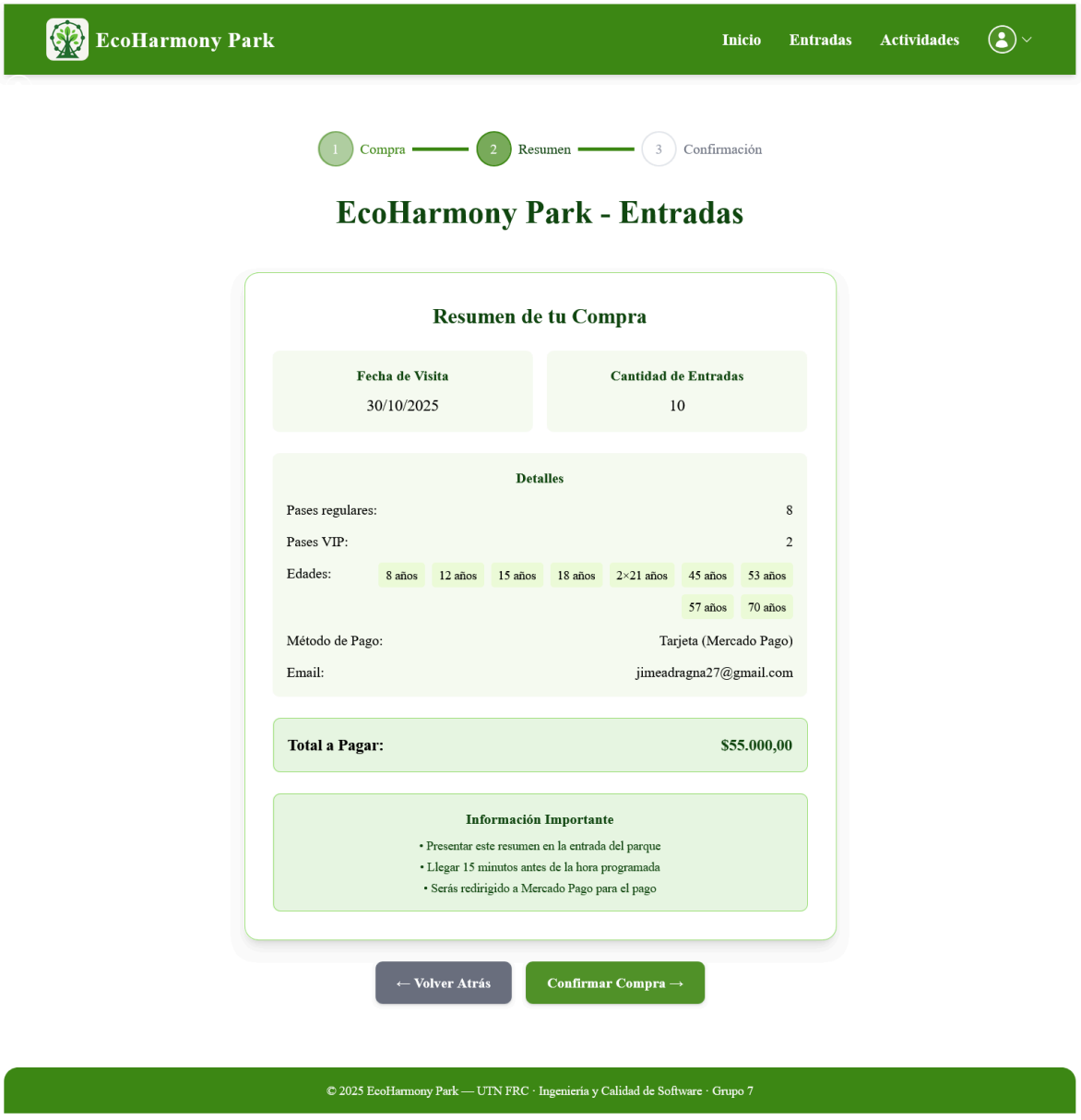


Figura Nº6: “Resumen de compra”

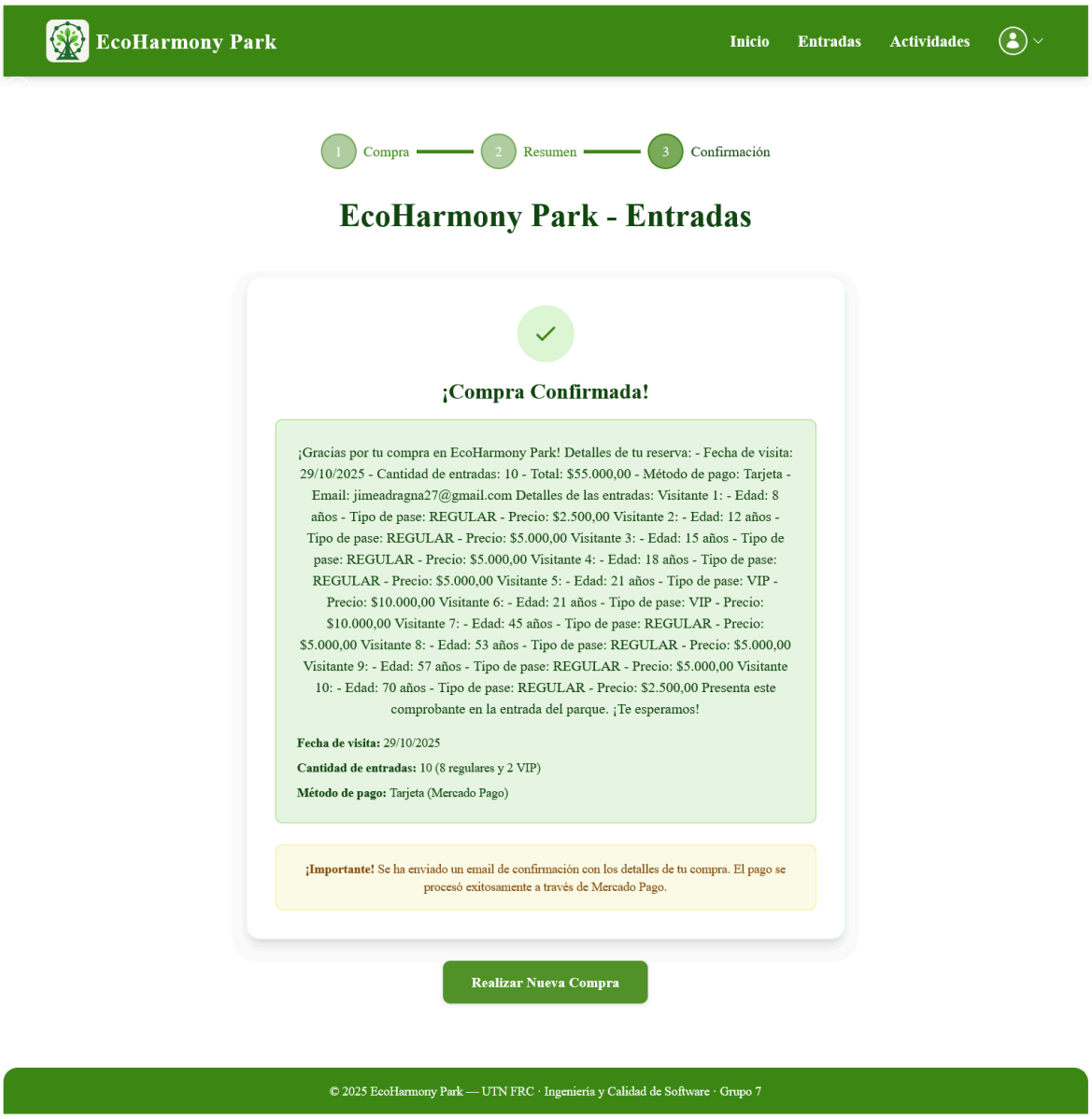


Figura Nº7: “Confirmación de compra”

Conclusión

La implementación del proyecto siguió un riguroso ciclo de Desarrollo Guiado por Pruebas (TDD), lo que aseguró la calidad y robustez del sistema. Este proceso se dividió en tres fases principales:

- **Fase Red:** Se definieron 103 pruebas exhaustivas que inicialmente fallaron, cubriendo tanto unidades lógicas internas como flujos de integración. Esto permitió guiar el diseño del manejo de excepciones y reglas de negocio.
- **Fase Green:** Se implementó el código mínimo necesario para que todas las pruebas pasarán, enfocándose en la funcionalidad principal de la User Story "Comprar Entradas".
- **Fase Refactor:** Se llevó a cabo una mejora continua del diseño, refactorizando el código para optimizar la legibilidad, mantenibilidad y rendimiento, siempre manteniendo las pruebas en verde para asegurar que no se introdujera regresiones.

Además, se tomaron decisiones de diseño clave, como la arquitectura en capas, la aplicación de principios SOLID, el uso de patrones de diseño (Service Layer, Strategy, Factory, Observer) y una estrategia robusta de manejo de errores con excepciones específicas. Se siguió la guía de estilo PEP 8 para el código backend y la guía Airbnb React/JSX Style Guide para el frontend, garantizando consistencia y legibilidad. La testabilidad fue una prioridad, facilitada por la inyección de dependencias y métodos pequeños.

Referencias

Universidad Tecnológica Nacional, Facultad Regional Córdoba. (2025). *Material de cátedra: Ingeniería y Calidad de Software, curso 4K1*. UTN FRC.

Python Software Foundation. (2024). *PEP 8 – Style Guide for Python Code*. https://peps.python.org/pep-0008/

Airbnb. (2024). *Airbnb JavaScript style guide – React/JSX*. https://airbnb.io/javascript/react/

Link del repositorio GitHub: https://github.com/marimilhas/ICS_GRUPO7