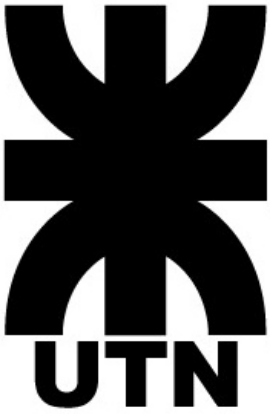
Test Driven Development (TDD)

Trabajo práctico 6



UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA

Ingeniería y Calidad de Software

GRUPO 3

Integrantes:

Barros, Enrique Agustín	98900
Capdevila Sauch, Catalina	77395
Figueroa, Francisco Emiliano	96809
Landi, Agostina Milena	94849
Lejtman, Uriel Benjamín	95903
Moreno, Matías Emanuel	95457
Moyano, Tomás	96846
Paris, Tomas	76791
Pedraza, Antonella	94868
Salzgeber, Ian	94173

Docentes: Ing. Laura Covaro - Ing. Cecilia Massano - Ing. Constanza Garnero

Curso: 4K3

Fecha de entrega: 24/10/2025

EcoHarmony Park

Decisiones de diseño y Flujo de Desarrollo

DECISIONES DE DISEÑO

Este bloque explica las decisiones de diseño más importantes y por qué las tomamos. Priorizamos contrato claro, testeabilidad (TDD), separación de responsabilidades y una UX consistente entre frontend y backend.

Convenciones y contrato
Elegimos usar **snake_case** de extremo a extremo en el backend y en el contrato HTTP (claves de request y response). Esto reduce ambigüedades y favorece la trazabilidad directa entre validadores, tests y payloads. En el frontend se preserva snake_case en el borde de la API y, cuando hace falta ergonomía para React/TS, se mapea a **camelCase** mediante funciones únicas (map_api_to_ui / map_ui_to_api) con pruebas que garantizan simetría. La razón es pragmática: el contrato es estable y testeado; la vista puede evolucionar sin forzar cambios en el backend.

Arquitectura y responsabilidades
Mantuvimos una separación clara: Express concentra el endpoint *POST /api/comprar-entrada*; la lógica de negocio vive en utilidades puras (utils/*) y un orquestador **comprar_entradas.js** que secuencia validaciones y cálculos. Esta elección favorece TDD: los validadores son funciones puras, baratas de probar y con mensajes exactos, mientras que la ruta se limita a coordinar y traducir códigos HTTP. También facilita mantenimiento: si cambia una regla, se modifica un validador sin tocar la ruta ni el esquema general.

Reglas de negocio y mensajes exactos
Todos los mensajes de validación son parte del contrato. Por eso las pruebas aceptan la igualdad exacta del string (por ejemplo, "No puede superar las 10 entradas", "No se pueden comprar entradas con más de un mes de anticipación"). Esta decisión elimina interpretaciones y evita regresiones silenciosas en el frontend. El costo es exigir disciplina al cambiar textos,
pero el beneficio es alto: cualquier cambio de copy rompe un test y obliga a revisar UI y documentación.

Cálculo de precios por edad y tipo
El precio se calcula por persona según edad y tipo de entrada (VIP o Regular) con las reglas acordadas: bebes y adultos mayores gratis, niños con 50% y el resto precio completo. Ubicamos el cálculo en una funcion pura **validar_edad_precio** para mantenerlo testeable y reutilizable. El frontend puede previsualizar totales con una versión espejo de estas reglas para UX, pero la última palabra la tiene el backend. Esto evita desalineaciones y dobles fuentes de lógica.

Fecha de apertura y compra anticipada

Se validan condiciones del calendario del parque (cerrado lunes, 25/12 y 1/1) y se restringe la compra a una ventana de anticipo de hasta un mes. Decidimos normalizar la fecha a Date local antes de validar para evitar bugs por zonas horarias. Esta decisión privilegia robustez y reduce "edge cases". Si la lógica de calendario creciera (feriados regionales, franjas horarias), el patrón de "validador puro + tests" nos permite extender sin romper el endpoint.

Gestión de errores y códigos HTTP

Optamos por 400 para errores de negocio (mensaje del validador) y 500 para excepciones no controladas. Cuando falla el envío de correo, devolvemos 200 con success: true, **correo_enviado**: false y una advertencia. El motivo es que la compra es una transacción de negocio ya validada; el correo es un canal secundario. Revertir por una falla de SMTP penaliza al usuario sin necesidad.

Correo y QR

Elegimos Nodemailer con credenciales vía .env y una plantilla HTML con branding del parque, código de entrada y QR de verificación. Usar un generador de QR remoto simplifica el stack y evita bibliotecas adicionales; si más adelante se requiere offline o mayor control estético, se puede reemplazar por una librería local sin cambiar el contrato, ya que el QR es un detalle de presentación.

Flujo end-to-end y resiliencia de UX

El formulario del frontend es multipaso para guiar al usuario y reducir errores: primero fecha y cantidad, luego detalles por visitante (edad y tipo) y finalmente pago y confirmación. Antes de llamar al backend, el service valida coherencia (longitudes de arrays, campos requeridos) para ahorrar round trips. Aun así, todas las decisiones definitivas se ejecutan en backend.

Esta doble validación (rápida en cliente, autoritativa en servidor) equilibra experiencia y seguridad.

TDD como motor de diseño

Diseñamos utilidades puras y orquestadores finos precisamente para testear primero: cada regla tiene su suite de pruebas con casos felices y bordes (nulo, vacío, inválido, límites de edad, calendario, formas de pago). Los mensajes exactos son asserts, y el flujo completo del endpoint tiene pruebas de integración. Este enfoque reduce el tiempo de depuración e incrementa la confianza al refactorizar. ***El costo es escribir más pruebas al inicio; el beneficio es iterar más rápido sin impactos inesperados.***

Seguridad y configuración

Se habilita CORS solo para los orígenes previstos y se usa JSON estricto. Las credenciales de SMTP van en variables de entorno; no se versionan. Las entradas del request se normalizan antes de validar (por ejemplo, trimming y parsing de fechas y números) para evitar errores por formatos heterogéneos.

FLUJO DE DESARROLLO

Tecnologías utilizadas: Backend (Express + Nodemailer) y Frontend (Next.js/React + Axios). Incluye arquitectura, flujo end-to-end, reglas de negocio, validaciones con mensajes exactos, TDD, errores/HTTP, y mejoras propuestas.

Contexto y objetivos

EcoHarmony Park implementa un flujo de compra de entradas en el que el backend valida reglas de negocio y calcula precios, y el frontend guía al usuario con validaciones de UX. El diseño prioriza modularidad y testeabilidad (TDD).

Arquitectura

Backend (Node.js/Express)

- **Servidor:** backend/app.js - configura CORS, JSON y monta la ruta en /api/comprar-entrada.
- **Ruta:** backend/routes/comprar-entrada.js - orquesta la compra: valida, calcula totales, genera codigo_entrada, intenta enviar correo y responde.
- **Dominio/reglas**: backend/utils/*.js - validadores unitarios y orquestador **comprar_entradas.js.**
- **Correo:** backend/mailer/enviar_correo.js - plantilla HTML con QR y detalles de la compra (SMTP por .env).

Frontend (Next.js/React/TypeScript)

- Formulario multipaso: frontend/components/ticket_purchase_form.tsx
- Lógica de envío y estado: frontend/hooks/use_comprar_entradas.ts
- Servicio HTTP (Axios): frontend/services/comprar_entradas.ts
- Utilidades/Reglas espejo: frontend/lib/park_data.ts

Endpoint único

POST /api/comprar-entrada (backend/app.js + backend/routes/comprar-entrada.js)

Body (JSON - ejemplo):

```
{
  "fecha_evento":
  "YYYY-MM-DD",
  "cantidad_entradas": 1,
  "edad_comprador":          [25],
  "tipo_entrada":       ["Regular"],
  "forma_pago":          "mercado
  pago",          "email_usuario":
  "user@example.com"
}
```

Respuesta (exito):
{
"success": true,
"message": "Compra realizada con exito. Se ha enviado un correo de confirmación.",
"codigo_entrada": "EHP-<timestamp>-<ID>",
"total": 5000,
"correo_enviado": true,
"advertencia": "El correo no se pudo enviar: <motivo>"
}
(*advertencia es opcional y aparece solo si el correo falla*)

Respuesta (error de validación):
{ "success": false, "message": "<mensaje correspondiente>" }

Respuesta (error interno):
{ "success": false, "message": "Error al procesar la compra", "error": "<detalle>" }

La ruta genera código entrada único (EHP-<Date.now()>-<random>) y calcula el total sumando el precio por persona con las reglas de edad/tipo. Si el email falla, el éxito no se revierte: se añade advertencia y correo_enviado=false.

Flujo end-to-end

1. Frontend (TicketPurchaseForm) valida paso a paso: fecha, cantidad, visitantes y pago. Muestra mensajes de UX.
2. Hook use_comprar_entradas arma el payload coherente y llama al service.
3. Service (frontend/services/comprar_entradas.ts):
   - Verifica requeridos y coherencia local: longitud de edad_comprador/tipo_entrada === cantidad_entradas.
   - POST a /api/comprar-entrada vía Axios.
4. Backend comprar-entrada.js:
   - Llama a utils/comprar_entradas.js. Si NO retorna "Compra realizada con exito" -> 400 con mensaje.
   - Genera codigo_entrada, calcula total con validar_edad_precio.
   - Intenta enviar_correo_responsable(email_usuario, datos).
   - Devuelve 200 con { success, message, codigo_entrada, total, correo_enviado, advertencia?
   - }.
5. Frontend muestra confirmación / redirección.

Reglas de negocio (desde los validadores)

Cantidad de entradas - utils/validar_cantidad_entradas.js

- Si cantidad > 10 -> "No puede superar las 10 entradas"
- Si cantidad < 1 -> "La cantidad de entradas debe ser al menos 1"
- Si valido -> "Se ingreso una cantidad válida de entradas"

Forma de pago - utils/validar_seleccion_forma_pago.js

- Aceptadas: "mercado pago" | "efectivo"
- Vacio/espacios/u otro -> "Forma de pago no válida"
- Valido -> "Forma de pago válida"

Fecha de apertura del parque - utils/validar_fecha_apertura.js

- null/undefined -> "La fecha es obligatoria"
- Fecha invalida -> "La fecha no es válida"
- Lunes -> "Parque cerrado los dias lunes"
- 25 de diciembre -> "Parque cerrado el 25 de diciembre"
- 1 de enero -> "Parque cerrado el 1 de enero"
- Valida -> "Se ingreso correctamente la fecha del evento"

Compra anticipada - utils/validar_compra_anticipada.js

- null/undefined/'' -> "La fecha del evento es obligatoria"
- Fecha invalida -> "La fecha del evento no es valida"
- Menor a hoy -> "La fecha del evento debe ser igual o mayor a la actual"
- > 1 mes -> "No se pueden comprar entradas con más de un mes de anticipacion"
- Ok -> "Se ingreso correctamente la fecha del evento"

Precio por edad y tipo - utils/validar_edad_precio.js

- VIP: $10.000 - Regular: $5.000 (según comentarios en el archivo).

- <= 3 y >= 60 -> gratis ($0).

4-15 -> 50% del precio base.

- Resto -> precio completo.
- Errores controlados: "La edad es obligatoria"; "La edad debe ser un número válido mayor o igual a 0"; "El tipo de entrada es obligatorio"; "Tipo de entrada invalido".

Calculo total: la ruta recorre las personas y suma validar_edad_precio(edad_i, tipo_i) para construir entradas_detalle[] y total.

Orquestador utils/comprar_entradas.js

- Entrada: (fecha_evento: string YYYY-MM-DD, cantidad_entradas: number, edad_comprador: number[], tipo_entrada: string[], forma_pago: string)
- Normalización: parsea el string de fecha a Date (local).
- Flujo (en orden): validar_cantidad_entradas -> validar_fecha_apertura -> validar_compra_anticipada -> loop validar_edad_precio -> validar_seleccion_forma_pago -> "Compra realizada con exito" o mensaje de error.

Envío de correo - backend/mailer/enviar_correo.js

- Nodemailer con .env: SMTP_HOST, SMTP_PORT, SMTP_USER, SMTP_PASS.
- Plantilla HTML: branding, **codigo_entrada**, QR (api.qrserver.com), detalle por entrada, forma_pago, total.
- Manejo de error: la compra no se bloquea; propaga **correo_enviado=false** y advertencia cuando falle.

Frontend: UX + resiliencia

- Formulario multipaso.
- Hook (**use_comprar_entradas**) centraliza estado y llamada al service.
- Service (**comprar_entradas.ts**) valida localmente y normaliza mensajes de error si el backend responde 4xx/5xx.
- Reglas espejo (lib/park_data.ts) para UX: is_park_open, is_christmas, is_new_year, is_within_one_month, calculate_total.

TDD - cobertura funcional

- Flujo: backend/tests/test_comprar_entradas.test.js.
- Validadores: suites para cantidad, apertura, anticipación, selección de pago, edad/precio (casos felices + bordes).

Errores y códigos HTTP (matriz)

- Backend: 400 en validación de negocio (mensaje del validador).
- Backend: 500 en excepción no controlada ("Error al procesar la compra" + error).
- Backend: 200 con advertencia si falla el correo (exito, **correo_enviado=false**).
- Frontend: manejar timeout/red de cliente con mensaje amigable.

Seguridad, configuración y despliegue

- CORS + JSON en app.js.
- .env para credenciales SMTP.