Informe de Implementación de Estrategia de Pruebas

Fecha: 22 de noviembre de 2025

Proyecto: Sistema de Gestión Academium

Elaborado por: Jefferson Perdomo, Santiago Barrera, Juliana Perez

Resumen

Este informe presenta el resultado del trabajo realizado para mejorar y consolidar el funcionamiento del Sistema de Gestión Académica. Durante esta entrega se aplicaron ajustes importantes en distintos componentes de la API, corrigiendo aspectos de tipado, validaciones, nombres de clases y métodos, manejo de errores, organización interna y consistencia entre capas. Asimismo, se optimizaron las consultas a la base de datos y se ordenó la estructura general del proyecto, logrando un sistema más claro, coherente y fácil de mantener.

Además, se implementó una estrategia completa de pruebas automatizadas que combina Pruebas Unitarias y Pruebas de Integración. Estas pruebas abarcan las entidades principales del sistema: Asignatura, Oferta, Período Académico, Plan de Estudio y Programa. También permiten verificar que las reglas de negocio se cumplan correctamente, que la lógica interna funcione como se espera y que todas las capas de la aplicación se comuniquen de manera adecuada.

En conjunto, estas mejoras fortalecen la calidad técnica del proyecto, aumentan la confiabilidad de la API y aseguran un comportamiento más estable y alineado con buenas prácticas de desarrollo profesional.

Ajustes Realizados

A continuación se presenta una descripción profunda de cada ajuste aplicado, organizada por capas arquitectónicas, con el objetivo de mostrar claramente **cómo mejoró el proyecto** en calidad, mantenibilidad y consistencia.

1. Configuración (common/configuración)

- **Problema original**

Las variables de entorno (PUERTO, credenciales DB, host, etc.) **no tenían validación**, y si faltaban, Fastify iniciaba con valores incorrectos (undefined, NaN), generando errores difíciles de diagnosticar.

🔸 **Ajustes realizados**

✔ Se agregó **validación de variables de entorno** usando Zod u otro validador.
✔ Se definieron **valores por defecto** cuando son razonables.
✔ Se centralizó toda la configuración en un módulo consistente.

🔸 **Por qué esto mejora el proyecto**

- Evita errores silenciosos.
- Garantiza que la aplicación falle explícitamente si falta una variable crítica.
- Permite escalar la aplicación sin confusiones.

2. Capa de dominio (modelos / reglas de negocio)

🔸 **Problemas detectados**

1. **Inconsistencia en nombres de clases** (asignatura, programa, etc. iniciaban con minúscula).
2. Algunas clases solo copiaban propiedades sin aportar lógica.
3. PeriodoAcademico implementaba *su propio nombre* en vez de la interfaz (implements PeriodoAcademico).
4. Inconsistencias de tipos (cupo: number vs. null en la clase).

🔸 **Ajustes aplicados**

✔ Estandarización: todas las clases comienzan con mayúscula (**Asignatura**, **Programa**, etc.).
✔ Corrección del tipado: las clases implementan **la interfaz correcta**.
✔ Alineación de tipos entre interfaces y clases.
✔ Revisión del uso de clases vacías:

- Se añadieron validaciones internas donde correspondía
- O se transformaron en DTOs si la clase no aportaba reglas de negocio

🔸 **Mejoras logradas**

- Código más legible, coherente y mantenible.

- Menos errores de tipado y más seguridad.
- El dominio representa correctamente la lógica del negocio.

3. Casos de uso (core/aplicación)

🔸 **Problemas detectados**

1. **Dependencias incorrectas** desde casos de uso hacia esquemas Zod de la capa de presentación.
2. Inconsistencia entre los tipos de entrada esperados por la interfaz y los usados en la clase.
3. Manejo heterogéneo de errores:
    - En algunos lugares se usa ValidacionError
    - En otros se usa Error genérico
4. Errores de escritura en nombres de métodos (ej. eliminiarPlanEstudio).

🔸 **Ajustes aplicados**

✔ Se corrigió la separación de capas:

- Ahora los casos de uso **NO dependen de Zod**.
- Se definieron **DTOs internos** para la capa de aplicación.
  ✔ Alineación de contratos entre interfaces y clases.
  ✔ Unificación del manejo de errores usando **ValidacionError** y errores específicos.
  ✔ Corrección de nombres de métodos para evitar ambigüedades.

🔸 **Mejoras conseguidas**

- Arquitectura más limpia (cumple DDD y Clean Architecture).
- Si cambia Fastify o Zod, la lógica del negocio no se rompe.
- Errores más consistentes y fáciles de manejar desde los controladores.

4. Infraestructura — Repositorios PostgreSQL

🔸 **Problemas encontrados**

1. Diferencias entre nombres retornados por PostgreSQL (idasignatura) y los esperados en el código (idAsignatura).
2. Uso de Object.keys() para generar columnas dinámicas, frágil ante cambios.
3. Fechas manipuladas según el orden de keys → muy inseguro.
4. Código duplicado para convertir fechas a YYYY-MM-DD.

🔸 **Ajustes aplicados**

✔ Se agregaron **alias SQL consistentes**:

```
SELECT idasignatura AS "idAsignatura"
```

✔ Se eliminaron dependencias peligrosas de Object.keys().
✔ Se creó una **utilidad común para manejo de fechas**.
✔ Se normalizó la lectura de resultados del pool a camelCase.
✔ Se revisó toda la capa de acceso a datos para evitar errores silenciosos.

🔸 **Beneficios**

- La capa de infraestructura es más robusta.
- Menor probabilidad de errores por columnas mal nombradas.
- Código más fácil de testear y extender.

5. Capa de presentación (Fastify)

🔸 **Problemas encontrados**

1. Inconsistencia en validaciones:
    - POST usando Zod

    - PUT aceptando datos sin validación

2. Controladores extraían substrings de error.message para decidir el código HTTP.

3. Falta de un **manejador global de errores**.

🔸 **Ajustes aplicados**

✔ Se usó Zod también en actualizaciones usando .partial().
✔ Se creó un **error handler global** para:

- ValidacionError → 400
- NotFoundError → 404
- ConflictError → 409
- Errores inesperados → 500
  ✔ Se estandarizaron mensajes y estructuras JSON.
  ✔ Se corrigieron nombres de rutas y controladores.

🔸 **Resultado**

- API más consistente y profesional.
- Controladores más limpios (menos try/catch repetidos).
- Respuestas uniformes para cualquier cliente (web, móvil, Postman, etc.).

6. Validación y modelo de errores

🔸 **Problemas iniciales**

- Mezcla de enfoques: ZodError, ValidacionError, Error genérico…
- Controladores interpretando mensajes en texto plano.
- Falta de jerarquía clara de errores del dominio.

🔸 **Ajustes aplicados**

✔ Se definieron errores propios del dominio:

- ValidacionError
- EntidadNoEncontradaError
- ReglaNegocioError
- ConflictError

✔ Se unificó la estrategia de validación en toda la API.
✔ Se conectó todo al manejador global de errores.

🔸 **Beneficios**

- Respuestas HTTP predecibles.
- Diagnóstico más claro.
- Código más profesional y mantenible.

Ajustes finales que fortalecen la calidad global

✔ **Separación estricta entre capas**

✔ **Contratos de entrada y salida definidos claramente**

✔ **Dominio más expresivo**

✔ **Infraestructura limpia y coherente**

✔ **Presentación desacoplada del negocio**

✔ **Pruebas más fáciles de mantener y expandir**

Resultado final

Después de aplicar estos ajustes, el proyecto:

- Cumple los principios de **arquitectura limpia**
- Tiene una base **fuerte, escalable y profesional**
- Su manejo de errores es **claro y unificado**
- La estructura es **consistente y mantenible**
- Supera la calidad de entregas anteriores por amplio margen

Ejecución de scripts para validación de pruebas

Mediante los siguientes scripts se puede validar y evidenciar que los casos de prueba se ejecutan correctamente, garantizando el correcto funcionamiento de las pruebas unitarias e integrales del sistema:

- npm run unit-test: Ejecuta las pruebas unitarias, permitiendo verificar el comportamiento individual de los componentes.
- npm run integration-test: Ejecuta las pruebas de integración, asegurando la interacción correcta entre los distintos módulos del sistema.

La ejecución de estos comandos permite comprobar de manera clara y controlada que el sistema cumple con los criterios de calidad definidos en el proceso de desarrollo.

1. Alcance de las Pruebas

La estrategia se centró en asegurar la calidad en las siguientes entidades, fundamentales para la operación del sistema:

1. **Periodo Académico:** Gestión de fechas de inicio, fin y transiciones de estado.
2. **Programa:** Definición y estructura de los programas académicos.

3. **Plan de Estudio:** Estructura curricular, relación entre asignaturas y prerrequisitos.
4. **Asignatura:** Definición de créditos, tipo de materia y códigos.
5. **Oferta Académica:** Disponibilidad de asignaturas en periodos específicos.

2. Pruebas Unitarias (Capa Lógica)

Las pruebas unitarias fueron implementadas en las capas de **Dominio** y **Aplicación**, utilizando un enfoque de aislamiento para cada componente.

Cobertura por Entidad y Capa

Se han cubierto tres componentes esenciales por cada entidad:

Entidad	Casos de Uso (Aplicación)	Controlador (Presentación)	Esquema (Dominio/Presentación)	Foco de la Prueba
Asignatura	Lógica de negocio (CRUD, validación de requisitos).	Manejo de peticiones HTTP, código de respuesta.	Validación y serialización de datos de entrada.	Reglas de Negocio y Lógica de Petición.
Oferta	Lógica de negocio específica del módulo.	Manejo de peticiones HTTP, código de respuesta.	Validación y serialización de datos de entrada.	Reglas de Negocio y Lógica de Petición.
Periodo Académico	**Validación de Transiciones de Estado** y traslape de fechas.	Manejo de peticiones HTTP, código de respuesta.	Validación y serialización de datos de entrada.	**Reglas de Transición de Estado**.

Plan de Estudio	Lógica de negocio (CRUD, validación de estructura).	Manejo de peticiones HTTP, código de respuesta.	Validación y serialización de datos de entrada.	Reglas de Negocio y Lógica de Petición.
Programa	Lógica de negocio (CRUD, validación de datos).	Manejo de peticiones HTTP, código de respuesta.	Validación y serialización de datos de entrada.	Reglas de Negocio y Lógica de Petición.

3. Pruebas de Integración (Flujo Completo)

Las pruebas de integración validan el flujo completo de la información, desde la capa de Presentación (Controlador) hasta la capa de Infraestructura (Repositorio) y viceversa, asegurando que todos los componentes trabajen armónicamente.

Estas pruebas confirman la correcta persistencia de datos y la coherencia en la manipulación de las entidades a través de los servicios expuestos.

Entidades con Cobertura de Integración

Los siguientes módulos tienen pruebas de integración implementadas, asegurando la comunicación completa con el sistema de persistencia (simulado o real):

Entidad	Archivo de Prueba	Foco de la Prueba
Asignatura	asignatura.int.te st.ts	Creación y recuperación de asignaturas en el flujo completo.
Oferta	oferta.int.test.t s	Integración de la lógica de oferta con dependencias.

Periodo Académico	periodo.int.test. ts	Integración del control de traslape y persistencia de estados.
Programa	programa.int.test .ts	Creación y modificación de programas en el flujo de aplicación.

4. Conclusión

La estrategia de pruebas implementada proporciona una alta confianza en la estabilidad y corrección del código base. La separación entre las pruebas unitarias, enfocadas en validar la lógica de negocio y las reglas internas, y las pruebas de integración, orientadas a comprobar la correcta comunicación entre las capas del sistema, permite identificar y aislar fallas de forma rápida y precisa. Esto facilita considerablemente el mantenimiento y el desarrollo futuro.

Además, durante esta entrega se integraron de manera completa los ajustes y mejoras detallados en el informe técnico, incluyendo la corrección de tipados, validaciones, nombres de clases y métodos, manejo de errores, coherencia entre capas, optimización de consultas y reorganización de la estructura de la API. La combinación de estos cambios con la estrategia de pruebas automatizadas fortalece significativamente la calidad técnica del proyecto y proporciona una base más estable, coherente y profesional para su evolución.