Tp final arquitectura de software

Elmer Alan Cornejo Quito

1. **Descripción de la aplicación**

   Este sistema como la consigna pide permite a los usuarios registrarse gestionar reservas de espacios y ver su disponibilidad (ejemplo. Salones, auditorios, canchas), además el administrador podrá revisar las reservas existentes.

   La aplicación esta desarrollada en Django con una organización modular que separa la lógica de negocio, la interfaz y el acceso a datos, sus módulos clave son:

   - Usuarios
   - Reservas
   - Espacios

   Facilitando así la mantenibilidad y la escalabilidad del sistema en el tiempo. Además se desarrollaron casos de uso encapsulados que representan acciones del negocio, como "crear una reserva" o "ver reservas ".

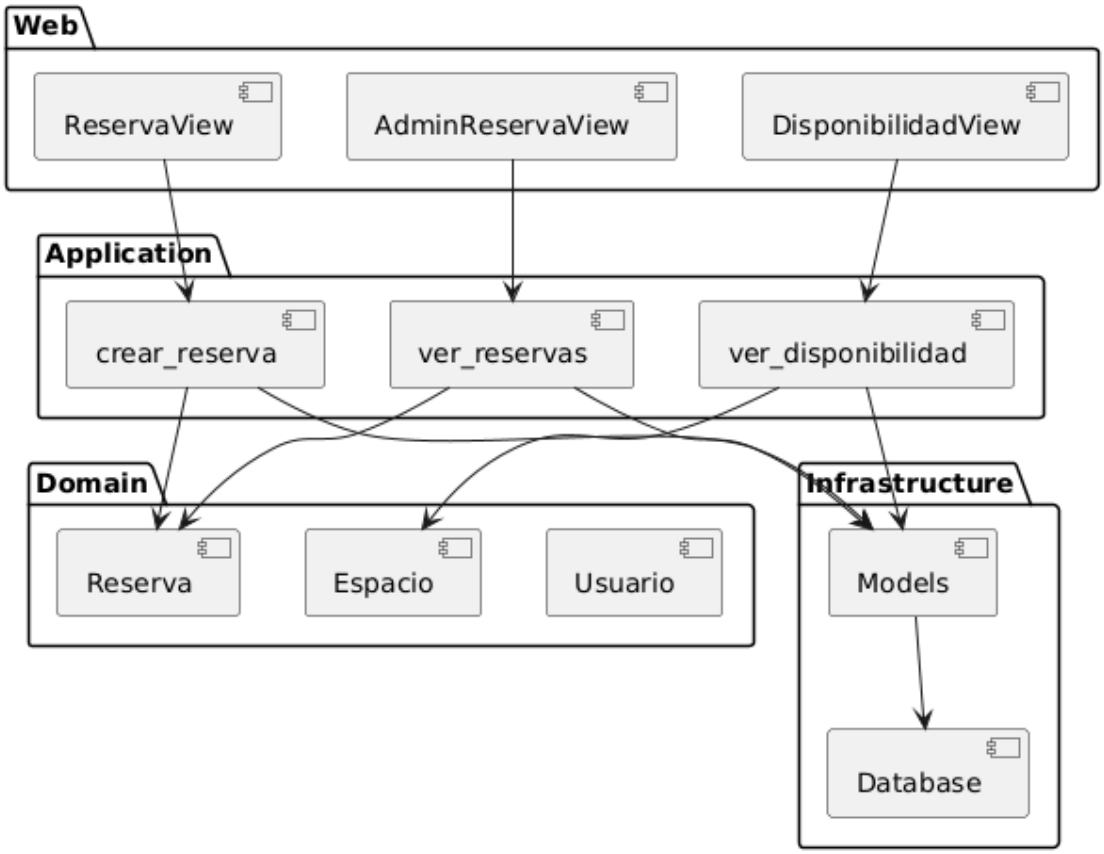
2. **Estilos arquitectónicos aplicados y justificación**

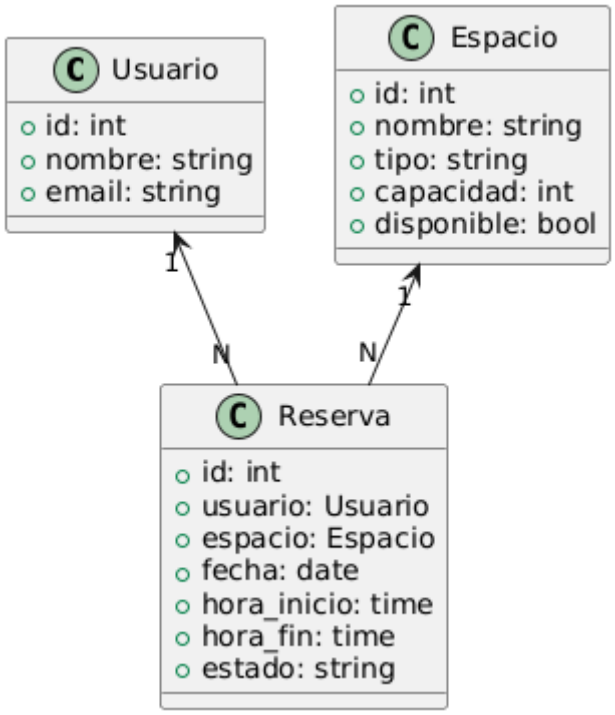
   Se opto por la arquitectura hexagonal o también conocida como Ports and adapters, ya que permiten separar la lógica del negocio de los detalles técnicos, manteniendo así un bajo acoplamiento entre capas facilitando así pruebas unitarias.

   Las capas del sistema son los siguientes:

   - Domain: contiene las entidades de negocio y sus atributos principales. Por ejemplo, una reserva tiene fecha, hora de inicio, hora de fin, usuario y espacio asignado.
   - Application: aquí se encapsulan los casos de uso como crear_reserva, ver_reservas y ver_disponibilidad, que combinan lógica y validaciones del negocio.
   - Infrastructure: contiene los modelos conectados a la base de datos y se comunica con la capa de aplicación.
   - Interfaces Web: a través de vistas HTTP se exponen los casos de uso al usuario (consumidos por postman).

   Este estilo permite adaptabilidad futura.





3. Atributos de calidad abordados

Se priorizaron los siguientes atributos de calidad:

1. Mantenibilidad

Justificación: Se usó una estructura modular con capas separadas (domain, application, infrastructure, web), lo que facilita realizar cambios sin afectar todo el sistema.

Estilo asociado: Clean Architecture (separación de responsabilidades, inversión de dependencias).

2. Reusabilidad

Justificación: Las funciones en application pueden usarse desde distintas interfaces , evitando duplicación de lógica.

Estilo asociado: Arquitectura por capas, que favorece componentes reutilizables.

3. Escalabilidad

Justificación: El diseño desacoplado permite agregar nuevas funciones o usuarios sin romper el sistema.

Estilo asociado: Diseño orientado a servicios (como base futura para microservicios).

4. **Principales decisiones arquitectónicas tomadas (ADR)**

- **ADR 001: Elección del Framework Django**

  Estado

  Aceptado

  Contexto

  Al iniciar el proyecto, necesitaba un framework web que me permitiera desarrollar de forma rápida pero ordenada. Como mi lenguaje base era Python, opté por evaluar frameworks dentro de ese ecosistema.

  Decisión

  Elegí Django porque ya contaba con experiencia básica usándolo, tiene una comunidad activa, y ofrece muchas herramientas integradas como ORM, gestión de rutas, formularios, autenticación, etc. Esto me permitió enfocarme en la arquitectura del sistema y no en configurar herramientas externas.

  Consecuencias

  Gracias a esta elección pude avanzar con rapidez. Sin embargo, decidí no usar el patrón MTV de Django y opté por separar la lógica según una arquitectura hexagonal, para cumplir con los requisitos del curso y mejorar la modularidad.

- **ADR 002: Arquitectura Hexagonal**

  Estado

  Aceptado

Contexto

Quería separar la lógica del negocio del framework, hacer pruebas sin depender de Django y cumplir con los criterios del curso.

Decisión

Apliqué arquitectura hexagonal, separando dominio, casos de uso, vistas web y adaptadores externos.

- domain/ para las entidades del negocio (Reserva, Espacio, Usuario)
- application/ para los casos de uso
- web/ para las vistas HTTP (adaptadores de entrada)
- infrastructure/ para posibles adaptadores como base de datos o notificaciones

Consecuencias

La estructura fue más compleja al principio, pero me dio más control y claridad para mantener, probar y escalar el sistema.

- **ADR 003: Separación por módulos**

Estado

Aceptado

Contexto

La app tenía funcionalidades distintas (usuarios, espacios, reservas, notificaciones) y necesitaba mantenerlas organizadas.

Decisión

Dividí el sistema por módulos en todas las capas, tanto para el dominio como para las vistas y los casos de uso.

- En domain/models/ tengo usuario.py, espacio.py, reserva.py
- En application/ los casos de uso están por funcionalidad
- En web/views/ cada vista HTTP representa a un módulo funcional

Consecuencias

Me permitió trabajar ordenadamente, con ramas específicas y pruebas más simples por funcionalidad.

5. **Trade-offs que surgieron durante el diseño**
   - **Separación de capas vs complejidad inicial**

Se leigio una estructura modular con capas inspirada en clean architecture. Aportando claridad y orden al código, pero también aumento la complejidad inicial generando varios errores por importaciones cruzadas.

- **Consistencia en nombres y atributos**

Hubo multiples errores generados por diferencias en los nombres de atributos entre modelos, lo que causo fallos al hacer POST o cargar datos en los tests.