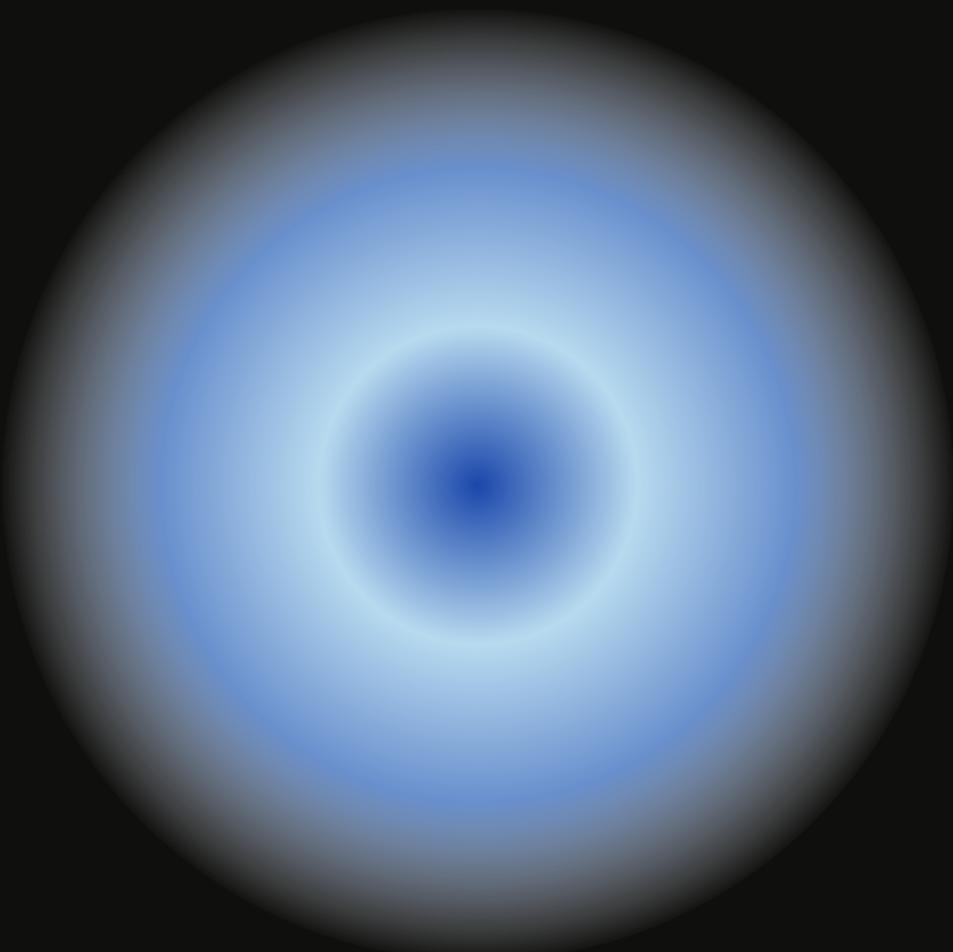


ARQUITECTURAS Y PATRONES

TypeScript



2025

Valentin Zoia



Tabla de Contenidos

Nota de Autor.....	8
<hr/>	
Introducción.....	9
<hr/>	
Capítulo 1.1: Los Fundamentos de la Arquitectura de Software.....	10
• ¿Qué Define una Arquitectura de Software?.....	10
• El Principio Rector: Separación de Responsabilidades.....	10
<hr/>	
Capítulo 1.2: Distinguiendo lo Esencial de lo Accidental.....	11
• La Clave del Éxito: Desacoplar la Lógica de Negocio... ..	11
• El Lenguaje Ubícuo: Construyendo un Vocabulario Común.....	13
<hr/>	



Tabla de Contenidos

Capítulo 1.3: Las Capas de la Arquitectura Limpia.....	14
• Capa Domain: El Corazón Inmutable.....	14
◦ Entidades: Los Pilares del Dominio.....	15
◦ Value Objects: Objetos Definidos por sus Atributos.....	15
◦ Aggregates: Manteniendo la Consistencia.....	15
◦ Interfaces: Los Contratos del Dominio.....	16
• Capa Application: Los Casos de Uso.....	16
◦ Principio de Responsabilidad Única.....	16
◦ CQRS: Separando Lecturas y Escrituras.....	17
◦ Inversión de Dependencias.....	17
• Capa Infrastructure: Las Implementaciones Concretas.....	17
• Capa Presentation: La Interfaz con el Mundo Exterior.....	18
Capítulo 1.4: La Regla de Oro de la Arquitectura Limpia.....	18
• La Regla de Dependencia.....	18
• Inyección de Dependencias.....	19
• Arquitectura de Plugins.....	19



Tabla de Contenidos

Capítulo 1.5: Patrones Arquitectónicos Relacionados 19

- Arquitectura Hexagonal..... 19
 - Domain-Driven Design (DDD)..... 20
 - Principios SOLID..... 20
-

Capítulo 1.6: Beneficios de la Arquitectura Limpia..... 21

- Ventajas Técnicas..... 21
 - Ventajas de Negocio..... 21
 - Comunicación Eficiente..... 21
-

Capítulo 1.7: Domain-Driven Design: Profundizando en el Dominio..... 22

- Introducción al DDD..... 22
 - DDD: Dos Caminos Complementarios..... 22
 - Parte Estratégica de DDD..... 22
 - Parte Táctica de DDD..... 23
 - Consideraciones Semánticas..... 24
-

Tabla de Contenidos

Capítulo 1.8: Patrones de Diseño en Arquitectura

Limpia.....	24
• Fundamentos de los Patrones de Diseño.....	24
• Principio Fundamental: No Forzar los Patrones.....	25
• La Base de los Patrones: Las Interfaces.....	25
◦ ¿Qué es una Interfaz?.....	25
◦ Diferencia entre Clase Abstracta e Interfaz.....	26
• Clasificación de Patrones.....	26
• Patrones Creacionales.....	27
• Patrón Builder - Creacional.....	27
◦ El Problema que Resuelve.....	27
◦ Implementación Básica del Builder.....	29
◦ Uso del Builder - La Magia del Encadenamiento..	32
• Patrón Factory - Creacional.....	33
• Patrón Abstract Factory - Creacional.....	35
• Patrón Prototype - Creacional.....	37
• Patrón Singleton - Creacional.....	38
• Patrones de Comportamiento.....	39
• Patrón Strategy - Comportamiento.....	39
• Patrón Command - Comportamiento.....	42
• Patrón State - Comportamiento.....	44
• Patrón Observer - Comportamiento.....	48



Tabla de Contenidos

• Patrones Estructurales.....	50
• Patrón Decorator - Estructural.....	50
• Patrón Adapter - Estructural.....	52
• Patrón Composite - Estructural.....	53
• Patrón Facade - Estructural.....	55
• Patrón Repository: - Estructural.....	57
◦ Motivación del Patrón.....	57
◦ Implementación del Patrón.....	58
<hr/>	
Capítulo 1.9: El Camino del Aprendizaje.....	61
• Conceptos Fundamentales.....	61
◦ Preguntas Clave.....	61
◦ Progresión Recomendada.....	61
<hr/>	
Capítulo 1.10: Bounded Contexts y Modularidad.....	62
• Definiendo Límites Conceptuales.....	62
• Módulos como Objetos de Dominio.....	62
• Escalabilidad Arquitectónica.....	63
• Conclusión.....	63

Tabla de Contenidos

Capítulo 2.1 - Parte Práctica.....	65
• Preparando el Proyecto Base.....	65
• Arquitectura del Proyecto.....	65
• Instalando y Configurando Prisma.....	67
• Entendiendo JWT (JSON Web Tokens).....	68
• Entendiendo Bcrypt.....	69
• Estructura Final del Proyecto.....	70
• Empezando con la Carpeta Shared.....	70
◦ Shared/Domain - Los Cimientos.....	71
◦ Shared/Application - La Base de los Servicios.....	72
◦ Shared/Infrastructure - Los Adapters.....	73
◦ Shared/Presentation - Middlewares Globales.....	76
• Módulo Users - Nuestro Primer Dominio.....	82
◦ Users/Domain - El Corazón del Negocio.....	82
◦ Users/Application - Los Casos de Uso.....	85
◦ Users/Infrastructure - La Persistencia.....	88
◦ Users/Presentation - La Interfaz HTTP.....	89
<hr/> Fin.....	92

```
if (parseInt(header1.css('padding-top'), 10) > header1_initialDistance) {  
    header1.css('padding-top', '' + $(window).scrollTop() - header1_initialDistance);  
}  
} else {  
    header1.css('padding-top', '' + header1_initialPadding + 'px');
```

Nota de Autor

Soy un joven principiante en el mundo del desarrollo de software, y este es el primer libro que escribo. No me considero un experto, ni pretendo que este libro sea una obra completa o definitiva. Simplemente busco compartir lo que aprendí recientemente sobre arquitecturas limpias y patrones de diseño.

Desde que comencé a interesarme por cómo estructurar mejor mis proyectos, me encontré con conceptos que cambiaron por completo mi forma de pensar el código. Lo que antes era caos, ahora podía tener orden. Lo que antes era improvisación, ahora tenía principios. Sentí la necesidad de organizar todo ese conocimiento, y de ahí nació este libro.

No soy partidario de escribir por escribir. Prefiero que el contenido tenga sentido, que pueda ser comprendido fácilmente, y que sirva como una herramienta útil para otros que también están empezando. Por eso, traté de hacer este libro lo más claro y visual posible, con ejemplos simples y explicaciones directas.

Mi objetivo es que esta guía pueda ayudarte a dar tus primeros pasos en el mundo de las arquitecturas y patrones, y que te acompañe en tu proceso de aprendizaje, como a mí me hubiese gustado que alguien lo hiciera cuando empecé.

¡Buena suerte en tu camino, y gracias por leer!

Valentin Zoia

Introducción

Arquitectura Limpia: Construyendo Software que Perdura

Imagina por un momento que sos un arquitecto encargado de diseñar un edificio de oficinas de tres pisos. ¿Comenzarías a colocar ladrillos sin planos? ¿Construirías a prueba y error, moviendo paredes cuando una oficina quede demasiado chica? Por supuesto que no. La arquitectura de software, al igual que la arquitectura tradicional, requiere una planificación cuidadosa y una estructura bien definida.

Una buena arquitectura debe gritar su propósito. Cuando observes los planos de un edificio, deberían transmitir inmediatamente si se trata de una oficina, un centro comercial o un departamento. Del mismo modo, la arquitectura de tu software debería proclamar claramente si estás construyendo un sistema médico, una plataforma de comercio electrónico o una red social.

Capítulo 1.1: Los Fundamentos de la Arquitectura de Software

¿Qué Define una Arquitectura de Software?

La arquitectura de software es el conjunto de reglas que nos autoimponemos al diseñar nuestras clases y estructurar nuestras aplicaciones. Su propósito fundamental es darnos una estructura, una organización que dé sentido a nuestro código para que este sea más escalable y mantenable.

Existe una distinción crucial entre arquitectura y diseño: La arquitectura opera a un nivel abstracto y te indica **QUÉ** se está haciendo. Te muestra en qué capas estás separando tu código y qué responsabilidades tiene cada una.

El diseño se centra en lo atómico y te indica **CÓMO** se están haciendo las cosas. Se enfoca en la implementación de patrones, comportamientos y estructuras específicas.

El Principio Rector: Separación de Responsabilidades

El objetivo primordial de toda arquitectura es lograr la separación de responsabilidades (separation of concerns). Este principio significa que cada componente debe tener una responsabilidad bien definida y separada, lo que resulta en un código más mantenable, escalable y eficiente.

La arquitectura limpia logra esta separación mediante capas, donde cada una tiene un propósito específico y relaciones claramente definidas con las demás.

Capítulo 1.2: Distinguiendo lo Esencial de lo Accidental

La Clave del Éxito: Desacoplar la Lógica de Negocio

La piedra angular de una arquitectura limpia reside en desacoplar completamente la lógica de negocio de los detalles de infraestructura. Esta separación determinará el éxito o fracaso de tu arquitectura.

Para lograr esto, primero debes dominar una distinción fundamental: ¿Qué constituye lógica de negocio? y ¿Qué pertenece a la infraestructura?

¿Qué es la Lógica de Negocio?

La lógica de negocio representa el corazón de tu dominio. Son las reglas, acciones y políticas que tienen vida fuera de tu aplicación. Estas reglas existirían incluso si tu software nunca hubiera sido creado.

Consideremos un ejemplo práctico: una plataforma de comercio electrónico. Los siguientes elementos forman parte de la lógica de negocio:

- Un producto tiene características específicas
- El proceso de compra de un producto
- Las políticas de devolución
- Los métodos de pago
- Las reglas de inventario (no vender sin stock)
- Los cálculos de precios y descuentos

Estas reglas son comprendidas por todos los involucrados en el proyecto: desarrolladores, especialistas en marketing, expertos del negocio y stakeholders. Son conceptos universales dentro del dominio que trascienden la implementación tecnológica.

¿Qué es la Infraestructura?

La infraestructura representa las herramientas que utilizas para ejecutar la lógica de negocio. Estos elementos son intercambiables y no afectan las reglas fundamentales del dominio.

Ejemplos de infraestructura incluyen:

- Bases de datos (PostgreSQL, MongoDB, MySQL)
- Frameworks web (Express, Django, Spring)
- Servicios de terceros (APIs de pago, servicios de email)
- Sistemas de almacenamiento
- Protocolos de comunicación

Lo crucial es entender que estos elementos no importan al negocio. Puedes cambiar de PostgreSQL a MongoDB, o de Express a Fastify, sin que las reglas de negocio se vean afectadas.

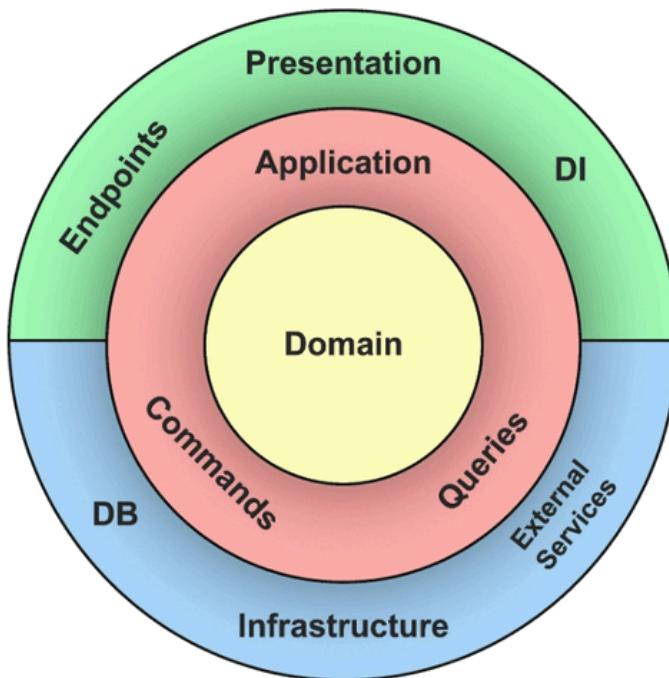
El Lenguaje Ubicuo: Construyendo un Vocabulario Común

Un aspecto fundamental para distinguir lo esencial de lo accidental es establecer un lenguaje ubicuo entre desarrolladores y expertos del dominio. Este concepto, introducido por Eric Evans en Domain-Driven Design, se refiere a un vocabulario compartido que todos los involucrados en el proyecto utilizan para describir el dominio.

El lenguaje ubicuo trasciende el código y se convierte en la herramienta de comunicación que garantiza que todos hablen de los mismos conceptos con la misma terminología. Cuando una persona de marketing habla de "envío gratuito", el desarrollador debe usar exactamente el mismo término en el código: `isFreeShipping`, no `noShippingCost` o `freeDelivery`.

Esta práctica asegura que el código sea auto-documentado y que las reglas de negocio sean evidentes para cualquier persona que lea el código, independientemente de su rol técnico.

Capítulo 1.3: Las Capas de la Arquitectura Limpia



Capa Domain: El Corazón Inmutable

La capa **Domain** constituye el núcleo de tu arquitectura. Es el código que no cambia porque el resto de la aplicación depende de él. Esta capa tiene características muy específicas:

Características del Domain:

- No depende de nada externo
- No tiene conocimiento de otras capas
- Contiene entidades y objetos de valor
- Define interfaces para servicios externos

Entidades: Los Pilares del Dominio

Las entidades representan objetos que tienen vida fuera de nuestro sistema. Cada entidad posee una identidad única y propiedades que tienen sentido para todos los expertos del dominio.

Una entidad `Product` en un e-commerce podría tener propiedades como `isFreeShipping`, `category`, `price`, `stockQuantity`. Estos nombres deben ser tan claros que incluso una persona del departamento de marketing pueda comprender qué representa cada propiedad.

Value Objects: Objetos Definidos por sus Atributos

Los Value Objects son objetos que se definen por sus atributos, no por su identidad. Dos Value Objects son iguales si todos sus atributos son iguales. Un ejemplo típico sería un objeto `Money` que contiene cantidad y moneda, o un objeto `Email` que encapsula la validación y formato de direcciones de correo electrónico.

Aggregates: Manteniendo la Consistencia

Los Aggregates son conjuntos de entidades y Value Objects que se tratan como una unidad para propósitos de consistencia de datos. Cada Aggregate tiene una raíz que actúa como el punto de entrada para todas las operaciones, asegurando que las reglas de negocio se mantengan íntegras.

Interfaces: Los Contratos del Dominio

En esta capa también residen las interfaces de los repositorios y servicios. Estas interfaces definen contratos que serán implementados por capas externas, permitiendo que el dominio permanezca independiente de los detalles de implementación.

Capa Application: Los Casos de Uso

La capa **Application** representa los casos de uso de la aplicación. Aquí reside la lógica de negocio transaccional que orquesta las interacciones entre entidades.

Esta capa coordina las diferentes interacciones de una tarea y delega el trabajo en los colaboradores de dominio. Es aquí donde existen los casos de uso que definen las acciones que el usuario puede realizar en la aplicación.

Principio de Responsabilidad Única

Cada caso de uso debe respetar el principio de responsabilidad única. Esto significa crear una clase específica para cada caso de uso, evitando clases "todopoderosas" que contengan múltiples responsabilidades.

Ejemplos de casos de uso:

- `CreateProductUseCase`
- `UpdateProductUseCase`
- `GetProductByIdUseCase`
- `RegisterUserUseCase`
- `LoginUserUseCase`

CQRS: Separando Lecturas y Escrituras

En muchos casos, se aplica el patrón CQRS (Command Query Responsibility Segregation), que separa las operaciones de escritura (comandos) de las operaciones de lectura (consultas). Esto proporciona mayor claridad y permite optimizar cada tipo de operación independientemente.

Inversión de Dependencias

Los casos de uso interactúan con servicios externos a través de interfaces definidas en la capa Domain. Esto implementa el principio de inversión de dependencias: los casos de uso dependen de abstracciones, no de implementaciones concretas.

Capa Infrastructure: Las Implementaciones Concretas

La capa **Infrastructure** contiene las implementaciones concretas de las interfaces definidas en el dominio. Aquí encontramos:

- Repositorios que implementan la persistencia de datos
- Servicios externos (APIs, sistemas de email)
- Configuraciones de bases de datos
- Integraciones con proveedores terceros

Esta capa es la más propensa a cambios y la más externa en términos de dependencias.

Capa Presentation: La Interfaz con el Mundo Exterior

La capa **Presentation** proporciona un puente entre el mundo exterior y los casos de uso. Es responsable de:

- Recibir solicitudes HTTP
- Validar datos de entrada
- Invocar casos de uso apropiados
- Formatear respuestas
- Manejar errores y excepciones

Incluye controladores, rutas, middlewares y toda la lógica relacionada con la interfaz de usuario o API.

Capítulo 1.4: La Regla de Oro de la Arquitectura Limpia

La Regla de Dependencia

La regla fundamental de la arquitectura limpia establece que **las dependencias de código solo pueden fluir desde las capas externas hacia las internas**. Las capas internas no deben tener conocimiento alguno de las capas externas. Esta regla garantiza que:

- La lógica de negocio permanezca pura e independiente
- Los cambios en capas externas no afecten el núcleo
- El sistema sea más testeable y mantenable

Inyección de Dependencias

Todas las dependencias se inyectan en lugar de instanciarse directamente. Esta inyección ocurre en el **Composition Root**, el lugar donde se configuran y conectan todos los componentes de la aplicación.

Arquitectura de Plugins

Este diseño crea lo que se conoce como "arquitectura de plugins", donde cada componente puede cambiar independientemente de los demás. La interfaz de usuario y la base de datos se convierten en plugins que se conectan al núcleo de la aplicación.

Capítulo 1.5: Patrones Arquitectónicos Relacionados

Arquitectura Hexagonal

La arquitectura limpia se basa en principios de la arquitectura hexagonal, que propone un núcleo central donde reside la lógica de negocio, rodeado por una capa externa de infraestructura.

Puertos y Adaptadores:

- Puerto: Una interfaz que define un contrato
- Adaptador: La implementación concreta de esa interfaz

Esta separación permite que la lógica de negocio permanezca completamente aislada de los detalles de implementación.

Domain-Driven Design (DDD)

La arquitectura limpia incorpora conceptos fundamentales del DDD:

- Bounded Contexts: Áreas de conocimiento que definen límites conceptuales
- Aggregates: Conjuntos de entidades que se tratan como una unidad
- Value Objects: Objetos que se definen por sus atributos, no por su identidad

Principios SOLID

Una arquitectura limpia es imposible sin adherirse a los principios SOLID:

- S: Responsabilidad Única
- O: Abierto/Cerrado
- L: Sustitución de Liskov
- I: Segregación de Interfaces
- D: Inversión de Dependencias

Capítulo 1.6: Beneficios de la Arquitectura Limpia

Ventajas Técnicas

1. **Testabilidad:** Cada componente puede probarse de forma aislada
2. **Mantenibilidad:** Los cambios se localizan en componentes específicos
3. **Escalabilidad:** Nuevas funcionalidades se agregan sin afectar el núcleo
4. **Flexibilidad:** Cambiar tecnologías no requiere reescribir la lógica de negocio

Ventajas de Negocio

1. **Reducción de costos:** Menos tiempo invertido en correcciones y mantenimiento
2. **Mayor productividad:** Desarrolladores pueden trabajar en paralelo sin interferencias
3. **Menor riesgo:** Cambios aislados reducen la probabilidad de introducir errores
4. **Longevidad:** El software perdura más tiempo sin necesidad de reescrituras completas

Comunicación Eficiente

Una arquitectura limpia hace que las necesidades operativas del sistema sean evidentes para todos los desarrolladores, facilitando la comunicación y el entendimiento del proyecto.

Capítulo 1.7: Domain-Driven Design: Profundizando en el Dominio

Introducción al DDD

Domain-Driven Design (DDD) es una técnica de desarrollo de software conceptualizada por Eric Evans. Su idea principal es capturar el dominio del modelo en los términos del mismo dominio, embeber este modelo en el código y protegerlo de influencias externas.

DDD es un enfoque en el desarrollo de software donde se centraliza todo en el dominio de negocio, utilizando un lenguaje ubicuo entre los desarrolladores y los expertos del dominio. El área temática en la que un usuario desarrolla un producto de software es el dominio. Un dominio es lo que hace una organización y el modo en que lo hace.

DDD: Dos Caminos Complementarios

Parte Estratégica de DDD

La parte estratégica se enfoca en el panorama general del dominio:

Bounded Contexts: Son áreas de conocimiento que funcionan como departamentos conceptuales. Cada contexto tiene su propio lenguaje ubicuo y sus propias reglas. Definen límites explícitos dentro de los cuales un modelo específico es válido.

Context Mapping: Define cómo se relacionan y comunican los diferentes Bounded Contexts entre sí. Establece las estrategias de integración y las formas de traducir conceptos entre contextos.

Core Domain: Identifica qué partes del dominio son fundamentales para el éxito del negocio y cuáles son de soporte o genéricas. El Core Domain recibe la mayor atención y los mejores recursos.

Parte Táctica de DDD

La parte táctica se enfoca en los patrones de diseño específicos:

Entidades: Objetos que tienen identidad y continuidad a través del tiempo. Su identidad las distingue, no sus atributos.

Value Objects: Objetos que se definen por sus atributos y no tienen identidad conceptual. Son inmutables y reemplazables.

Aggregates: Conjuntos de entidades y Value Objects que se tratan como una unidad. Tienen una raíz que controla el acceso y mantiene la consistencia.

Repositorios: Abstracciones que encapsulan la lógica de acceso a datos, proporcionando una interfaz orientada a colecciones.

Domain Services: Servicios que encapsulan lógica de dominio que no pertenece naturalmente a una entidad o Value Object.

Domain Events: Eventos que representan algo significativo que ocurrió en el dominio.

Consideraciones Semánticas

Al implementar DDD, es crucial dotar de mecanismos semánticos a la mutabilidad de los atributos. En lugar de utilizar funciones genéricas de asignación (setters), se debe emplear métodos con nombres auto-descriptivos en el lenguaje ubicuo.

Por ejemplo, en lugar de `product.setPrice(newPrice)`, sería más expresivo usar `product.updatePriceForSale(newPrice)` o `product.applyDiscount(discountPercentage)`.

Capítulo 1.8: Patrones de Diseño en Arquitectura Limpia

Fundamentos de los Patrones de Diseño

Si estás desarrollando software, seguramente te has enfrentado a problemas que parecen repetirse una y otra vez. ¿Te suena familiar esa sensación de estar escribiendo el mismo tipo de código para resolver situaciones similares? Bueno, no sos el único pa.

Un patrón de diseño es exactamente eso, es una solución probada a un problema común en el desarrollo de software. Es una técnica específica que ya ha sido validada y sabemos que funciona efectivamente.

Los patrones de diseño fueron concebidos para el paradigma orientado a objetos, utilizando interfaces, clases, clases abstractas, herencia e implementaciones. Son técnicas que pueden resolver problemas comunes, proporcionando soluciones ya acopladas a problemas específicos.

Principio Fundamental: No Forzar los Patrones

Es esencial entender que los patrones de diseño no deben forzarse ni aplicarse por moda. Si la aplicación de un patrón no es clara o no resuelve un problema específico, es mejor no utilizarlo. La simplicidad siempre es preferible a la complejidad innecesaria.

La Base de los Patrones: Las Interfaces

¿Qué es una Interfaz?

Una interfaz es un contrato, un protocolo por el cual los objetos van a interactuar entre ellos sin conocerse directamente. Es un mecanismo para establecer contratos que deben ser respetados por las implementaciones.

Cuando implementas una interfaz, debes cumplir tanto con los nombres de los métodos como con los tipos de parámetros y retorno especificados. Lo que importa es respetar el contrato; la implementación interna es responsabilidad del programador.

Diferencia entre Clase Abstracta e Interfaz

Interfaz: Es un contrato puro que debes respetar al implementarlo. No contiene implementación, solo definiciones.

Clase Abstracta: Tiene métodos ya implementados y desarrollados, además de propiedades y atributos. No se puede instanciar directamente, pero sí se pueden crear objetos que hereden de ella.

Limitaciones:

- Una clase abstracta permite herencia única
- Una interfaz puede tener múltiples implementaciones

Clasificación de Patrones

Los patrones de diseño se dividen en tres categorías principales:

- Creacionales: Se enfocan en la creación de objetos
- Estructurales: Se ocupan de la composición de clases y objetos
- Comportamentales: Se centran en la interacción entre objetos

Patrones Creacionales

Patrón Builder - Creacional

No has tenido la necesidad de que cuando creas un objeto y tiene un constructor enorme, y decis que clase de mierd! Que ocurriria si a este constructor le mando solo tres parametros en lugar de los 10 que tiene, o solo le mando 7, o mando los 10. Es decir este objeto para crearse es muy robusto, un constructor enorme. Siempre que tengamos un constructor enorme, ahí puede ser usado el patron builder. Lo que hace es separar esos parámetros/properties en métodos, que se pueden encadenar uno tras otro y asi vas construyendo tu objeto, no con el constructor, sino con la ejecución de métodos.

El Problema que Resuelve

Imagina una clase que tiene un constructor de 20 campos. Imagina que de ese objeto, vas a necesitar solo 3 campos en lugar de 20, o a lo mejor vas a necesitar todos los campos. Esto genera un problema serio de usabilidad y mantenimiento.

Veamos un ejemplo problemático en la siguiente página:

```

// ✗ Problema: Constructor gigante
class Pizza {
  constructor(
    public size: string,
    public dough: string,
    public sauce: string,
    public cheese: string,
    public pepperoni: boolean = false,
    public mushrooms: boolean = false,
    public onions: boolean = false,
    public sausage: boolean = false,
    public bacon: boolean = false,
    public ham: boolean = false,
    public pineapple: boolean = false,
    public olives: boolean = false,
    public peppers: boolean = false,
    public extraCheese: boolean = false,
    public thinCrust: boolean = false,
    public garlicButter: boolean = false,
    public spicyLevel: number = 0,
    public glutenFree: boolean = false,
    public vegan: boolean = false,
    public calories: number = 0
  ) {}
}

// Usar esta clase es una pesadilla:
const pizza1 = new Pizza(
  "large", "thick", "tomato", "mozzarella",
  true, false, false, false, false, false,
  false, false, false, false, false, false,
  0, false, false, 450
);

// ¿Qué representa cada boolean? ¡Imposible de leer!

```

¿Hay un patrón creacional que puede resolver esto? Sí, el patrón Builder.

Builder es un patrón de diseño que te va a resolver esto. Lo que hace es separar esos atributos y propiedades del objeto en métodos.

Esos métodos se van a encadenar según los que necesites. Es decir, tienes tu objeto, tu **new Object** y el método para agregar cierto campo, cierta propiedad, y eso puede encadenar con otra y otra propiedad, y al último un método llamado **build()**

Implementación Básica del Builder

Ver en la siguiente página:

```

// ✓ Solución: Patrón Builder
class Pizza {
    public size: string = "";
    public dough: string = "";
    public sauce: string = "";
    public cheese: string = "";
    public pepperoni: boolean = false;
    public mushrooms: boolean = false;
    public onions: boolean = false;
    public sausage: boolean = false;
    public bacon: boolean = false;
    public ham: boolean = false;
    public pineapple: boolean = false;
    public olives: boolean = false;
    public peppers: boolean = false;
    public extraCheese: boolean = false;
    public thinCrust: boolean = false;
    public garlicButter: boolean = false;
    public spicyLevel: number = 0;
    public glutenFree: boolean = false;
    public vegan: boolean = false;
    public calories: number = 0;

    describe(): string {
        return `Pizza ${this.size} con masa ${this.dough}, salsa ${this.sauce} y queso
        ${this.cheese}`;
    }
}

class PizzaBuilder {
    private pizza: Pizza;

    constructor() {
        this.pizza = new Pizza();
    }

    // Métodos para características básicas
    setSize(size: string): PizzaBuilder {
        this.pizza.size = size;
        return this; // Retorna this para permitir encadenamiento
    }

    setDough(dough: string): PizzaBuilder {
        this.pizza.dough = dough;
        return this;
    }
}

```

```

setSauce(sauce: string): PizzaBuilder {
  this.pizza.sauce = sauce;
  return this;
}

setCheese(cheese: string): PizzaBuilder {
  this.pizza.cheese = cheese;
  return this;
}

// Métodos para ingredientes adicionales
addPepperoni(): PizzaBuilder {
  this.pizza.pepperoni = true;
  return this;
}

addMushrooms(): PizzaBuilder {
  this.pizza.mushrooms = true;
  return this;
}

addOnions(): PizzaBuilder {
  this.pizza.onions = true;
  return this;
}

/*
...
Seguirian metodos como: addSausage(), addBacon(), addHam(), addPineapple(),
addOlives() ...
setSpicyLevel(level: number), makeGlutenFree(), makeVegan(), setCalories(), etc
*/

// Método final que construye el objeto
build(): Pizza {
  return this.pizza;
}
}

```

Uso del Builder - La Magia del Encadenamiento

```
// Ahora crear pizzas es intuitivo y legible:  
  
// Pizza simple  
const pizzaMargarita = new PizzaBuilder()  
.setSize("medium")  
.setDough("thin")  
.setSauce("tomato")  
.setCheese("mozzarella")  
.build();  
  
// Pizza compleja  
const pizzaSuprema = new PizzaBuilder()  
.setSize("large")  
.setDough("thick")  
.setSauce("tomato")  
.setCheese("mozzarella")  
.addPepperoni()  
.addMushrooms()  
.addOnions()  
.addSausage()  
.addPeppers()  
.addExtraCheese()  
.setSpicyLevel(2)  
.build();  
  
// Pizza vegana  
const pizzaVegana = new PizzaBuilder()  
.setSize("medium")  
.setDough("whole wheat")  
.setSauce("pesto")  
.setCheese("vegan cheese")  
.addMushrooms()  
.addOnions()  
.addOlives()  
.addPeppers()  
.makeVegan()  
.makeGlutenFree()  
.setCalories(320)  
.build();  
  
console.log(pizzaMargarita.describe());  
console.log(pizzaSuprema.describe());  
console.log(pizzaVegana.describe());
```

Activar Windows
Ve a Configuración para acti

Como pueden ver esta solución es muucho más legible, flexible y mantenible.

A esto también se le pueden agregar validaciones en la clase **PizzaBuilder** en cada seter como **setSize** se le podría agregar una validación para que por ejemplo se reciba un tipo de size válido, pienso tener un array como **const validSizes = ["small", "medium", "large", "extra-large"];**

y que valide si el parámetro size pasado incluye o no los valores de ese array. Acá podemos validar lo que queramos y gusten, es a gusto de cada uno.

Patrón Factory – Creacional

Imaginate que tenés una app de comida que va creciendo día a día. Cada vez agregas más platos al menú, y inevitablemente terminas metiendo **if** por todos lados (o si sos mas moderno, usas **switch**). El problema es que esto se vuelve imposible de mantener cuando la aplicación crece y crece.

¿Cuándo usarlo? Cuando la creación de objetos depende de condiciones o tipos diferentes, y cuando querés evitar modificar el código base cada vez que agregas algo nuevo. El Factory Pattern encapsula la creación de objetos complejos, especialmente útil cuando la creación involucra lógica compleja o múltiples pasos.

```

// Ejemplo básico con Factory Method

interface Food {
    prepare(): void;
}

class Pizza implements Food {
    prepare(): void {
        console.log("Preparando Pizza...")
    }
}

class Burger implements Food {
    prepare(): void {
        console.log("Preparando Burger...")
    }
}

class Empanada implements Food {
    prepare(): void {
        console.log("Preparando Empanada...")
    }
}

enum FOODS_NAMES {
    PIZZA = 'pizza',
    BURGER = 'burger',
    EMPANADA = 'empanada'
}

const TiposDeComida: Record<FOODS_NAMES, any> = {
    pizza: Pizza,
    burger: Burger,
    empanada: Empanada,
} as const;

class FoodFactory {
    static createFood(type: keyof typeof TiposDeComida): Food {
        return new TiposDeComida[type]();
    }
}

// Uso
const factory = FoodFactory.createFood(FOODS_NAMES.EMPANADA);
factory.prepare(); // "Preparando Empanada..."

```

Patrón Abstract Factory - Creacional

Ahora, imaginemos que tu app de comida se expande internacionalmente. El problema es que dependiendo del país donde se haga la comida, esa misma comida tiene diferentes métodos de preparación. Vas a necesitar crear familias de productos/comida que van a variar según el país donde te encuentres.

La solución: Abstract Factory. Este patrón te permite crear familias de objetos relacionados sin especificar sus clases concretas.

```

// Productos específicos por país
class ArgentinaPizza extends Pizza {
    prepare(): void {
        console.log("Preparando Pizza Argentina...")
    }
}

class ItalianPizza extends Pizza {
    prepare(): void {
        console.log("Preparando Pizza Italiana...")
    }
}

// Factory abstracta
interface FoodFactory {
    createPizza(): Pizza;
    createBurger(): Burger;
}

// Factories concretas por país
class ArgentinaFoodFactory implements FoodFactory {
    createBurger(): Burger {
        return new ArgentinaBurger();
    }
    createPizza(): Pizza {
        return new ArgentinaPizza();
    }
}

class ItalianFoodFactory implements FoodFactory {
    createBurger(): Burger {
        return new ItalianBurger();
    }
    createPizza(): Pizza {
        return new ItalianPizza();
    }
};

const argFoodFactory = new ArgentinaFoodFactory();
const pizza_Arg = argFoodFactory.createPizza();
console.log(pizza_Arg.prepare());//Preparando Pizza Argentina...

```

Patrón Prototype - Creacional

Todos los objetos de JavaScript tienen un objeto más arriba del cual heredan, llamado prototype. ¿Para qué se utiliza? Para clonar objetos complejos sin tener que reconstruirlos a mano. Es perfecto para repetir pedidos, plantillas, o cualquier situación donde necesites duplicar objetos complejos.

```
interface Clonable<T> {
    clone(): T
}

class Order implements Clonable<Order> {
    items: string[];
    address: string;

    constructor(items: string[], address: string) {
        this.items = items;
        this.address = address;
    }

    clone(): Order {
        return new Order([...this.items], this.address);
    }
}

// Uso
const originalOrder = new Order(
    ["Coca Cola 2L", "Cheetos", "Pizza"],
    "Segurola 1407"
);

const clonedOrder = originalOrder.clone();
clonedOrder.items.push("Empanadas");

// La orden original no se ve afectada
console.log(originalOrder.items); // ["Coca Cola 2L", "Cheetos", "Pizza"]
console.log(clonedOrder.items); // ["Coca Cola 2L", "Cheetos", "Pizza", "Empanadas"]
```

Patrón Singleton - Creacional

El más famoso de los patrones creacionales. Digamos que necesitás una única instancia de configuración global para toda la app. El singleton te garantiza que solo existe una instancia de una clase en toda la aplicación.

¿Cuándo usarlo? Cuando necesitás una única instancia global. Por ejemplo: un configurador, un logger, cuando estás llamando un servicio, o cuando tenés que compartir información entre diferentes lugares y tiene que estar todo almacenado en el mismo lugar.

Advertencia: No hay que abusar del Singleton, porque puede convertirse en un ‘god object’, un objeto todopoderoso que termina haciendo demasiadas cosas.

```

class ConfigManager {
  private static instance: ConfigManager;
  private config: Record<string, any> = {};

  private constructor(){}

  static getInstance(): ConfigManager {
    if(!ConfigManager.instance){
      ConfigManager.instance = new ConfigManager();
    }
    return ConfigManager.instance;
  }

  set(key:string, value:any){
    this.config[key] = value;
  }

  get(key:string){
    return this.config[key]
  }
}

//Uso
const config1 = ConfigManager.getInstance();
const config2 = ConfigManager.getInstance();

config1.set("apiUrl", "https://api.google.com");
console.log(config2.get("apiUrl")); // "https://api.google.com"
console.log(config1 === config2); // true

```

Patrones de Comportamiento

Patrón Strategy – Comportamiento

Imaginate que tenés diferentes métodos para calcular el envío: por distancia, por monto, envío gratis, etc. El Strategy Pattern te permite definir una familia de algoritmos, encapsulados y hacerlos intercambiables.

¿Cuándo usarlo? Cuando tenés que cambiar el algoritmo de manera dinámica en tiempo de ejecución.

```

interface ShippingStrategy {
    calculate(amount: number): number;
}

class DistanceShipping implements ShippingStrategy {
    calculate(amount: number): number {
        return amount * 1.2; // 20% de recargo por distancia
    }
}

class FreeShipping implements ShippingStrategy {
    calculate(_amount: number): number {
        return 0; // Sin recargo
    }
}

class ShippingContext {
    constructor(private strategy: ShippingStrategy) {}

    getShippingCost(amount: number): number {
        return this.strategy.calculate(amount);
    }
}

// Uso
const context = new ShippingContext(new DistanceShipping());
console.log(context.getShippingCost(100)); // 120

const freeContext = new ShippingContext(new FreeShipping());
console.log(freeContext.getShippingCost(100)); // 0

```

Un ejemplo más divertido sería con diferentes estrategias de ataque en un juego: tenés diferentes peleadores, cada uno puede ser un mago, un luchador, un guerrero, arquero, etc. Y cada uno tiene que tener una estrategia de ataque diferente.

Veamos ése ejemplo:

```

/*
POSSIBLE EJEMPLO: Tienes diferentes peleadores,
cada peleador puede ser un mago, un luchador, un guerrero, arquero, etc
y cada uno tiene que tener una estrategia de ataque diferente.
Golpe fuerte, golpe rapido, hechizo, etc.
*/

// Ejemplo de uso con diferentes estrategias de ataque
interface AttackStrategy {
    attack(): string;
}

class StrongAttack implements AttackStrategy {
    attack(): string {
        return "Ataque fuerte!";
    }
}

class QuickAttack implements AttackStrategy {
    attack(): string {
        return "Ataque rápido!";
    }
}

class SpellAttack implements AttackStrategy {
    attack(): string {
        return "Lanzar hechizo!";
    }
}

class Archer {
    constructor(private strategy: AttackStrategy) {}

    performAttack(): string {
        return this.strategy.attack();
    }
}

// Uso de las estrategias de ataque
const strongArcher = new Archer(new StrongAttack());
const quickArcher = new Archer(new QuickAttack());
const spellArcher = new Archer(new SpellAttack());
console.log(strongArcher.performAttack()); // Ataque fuerte!
console.log(quickArcher.performAttack()); // Ataque rápido!
console.log(spellArcher.performAttack()); // Lanzar hechizo!

```

Patrón Command - Comportamiento

El Command Pattern te sirve para encapsular una petición en un objeto, de tal manera que podés parametrizar los objetos con diferentes peticiones, encolar o registrar las peticiones, y soportar operaciones que se puedan deshacer.

¿Cuándo usarlo? Cuando tenés operaciones de hacer y deshacer, cuando querés parametrizar acciones y tener logs de operaciones.

```

interface Command {
    execute(): void;
    undo():void;
}

class AddItemCommand implements Command {
    constructor(
        private order:string[],
        private item:string,
        private logs:string[] = []
    )
    {}

    execute(): void {
        this.order.push(this.item);
        console.log(`Item ${this.item} added to order.`);
        this.logs.push(`Added ${this.item}`);
    }

    undo(): void {
        const index = this.order.indexOf(this.item);
        if (index > -1) {
            this.order.splice(index, 1);
            console.log(`Item ${this.item} removed from order.`);
            this.logs.push(`Removed ${this.item}`);
        } else {
            console.log(`Item ${this.item} not found in order.`);
        }
    }
}

//Uso
const order:string[] = [];
const addPizza = new AddItemCommand(order, 'Pizza');
const addSoda = new AddItemCommand(order, 'Soda');
addPizza.execute(); // Item Pizza added to order.
addSoda.execute(); // Item Soda added to order.

console.log(order); // ['Pizza', 'Soda']
addPizza.undo(); // Item Pizza removed from order.
console.log(order); // ['Soda']
addSoda.undo(); // Item Soda removed from order.
console.log(order); // []

```

Patrón State - Comportamiento

Imaginate que tenés un pedido que puede estar en diferentes estados: en espera, preparando, listo para entregar, entregado. Querés poder cambiar el estado del pedido de manera controlada, y que cada estado tenga su propia lógica.

¿Cuándo usarlo? Cuando tenés un objeto que puede estar en diferentes estados, y querés encapsular la lógica de cada estado en una clase. También cuando el comportamiento de un objeto depende del estado, y cuando tenes la necesidad de evitar condicionales gigantes basados en el estado.

```

/*
Situacion: Tengo un pedido, que puede estar en diferentes estados:
 * - En espera - NewOrder
 * - Preparando - PreparingOrder
 * - Listo para entregar - ReadyForDelivery
 * - Entregado - DeliveredOrder
 *
 * Quiero poder cambiar el estado del pedido de manera controlada,
 * y que cada estado tenga su propia logica.
 *
 * Cuando usarlo? Cuando tengo un objeto que puede estar en diferentes estados,
 * y quiero encapsular la logica de cada estado en una clase.
 * Cuando Usarlo? Cuando el comportamiento de un objeto depende
 * del estado, cuando tenes la necesidad de evitar condicionales
 * gigantes por un estado

*/

```

```

interface OrderState {
    next(order:OrderContext): void;
    previous(order:OrderContext): void;
    status(): string;
}

class NewOrder implements OrderState {
    next(order: OrderContext): void {
        order.setState(new PreparingOrder());
    }
    previous(order: OrderContext): void {
        console.log("This is the first state, cannot go back.");
    }
    status(): string {
        return "Step 1: Order is new and waiting to be prepared.";
    }
}

```

```
class PreparingOrder implements OrderState{
    next(order: OrderContext): void {
        order.setState(new ReadyForDelivery());
    }
    previous(order: OrderContext): void {
        order.setState(new NewOrder());
    }
    status(): string {
        return "Step 2: Order is being prepared.";
    }
}

class ReadyForDelivery implements OrderState{
    next(order: OrderContext): void {
        order.setState(new DeliveredOrder());
    }
    previous(order: OrderContext): void {
        order.setState(new PreparingOrder());
    }
    status(): string {
        return "Step 3: Order is ready for delivery.";
    }
}

class DeliveredOrder implements OrderState{
    next(order: OrderContext): void {
        console.log("Order has already been delivered, cannot go forward.");
    }
    previous(order: OrderContext): void {
        order.setState(new ReadyForDelivery());
    }
    status(): string {
        return "Step 4: Order has been delivered.";
    }
}
```

```
class OrderContext{
    private state: OrderState;
    constructor(){
        this.state = new NewOrder(); // Initial state
    }
    setState(state: OrderState): void {
        this.state = state;
    }

    next(): void {
        this.state.next(this);
    }

    previous(): void {
        this.state.previous(this);
    }

    status(): string {
        return this.state.status();
    }
}

const orden = new OrderContext();
console.log(orden.status()); // Step 1: Order is new and waiting to be prepared.
orden.next();
console.log(orden.status()); // Step 2: Order is being prepared.
orden.next();
console.log(orden.status()); // Step 3: Order is ready for delivery.
orden.next();
console.log(orden.status()); // Step 4: Order has been delivered.
orden.previous();
console.log(orden.status()); // Step 3: Order is ready for delivery.
orden.previous();
console.log(orden.status()); // Step 2: Order is being prepared.
orden.previous();
console.log(orden.status()); // Step 1: Order is new and waiting to be prepared.
```

Patrón Observer - Comportamiento

El Observer Pattern te permite notificar a múltiples objetos cuando hay cambios en otros. Por ejemplo, queres notificar a la cocina y al repartidor cuando hay un nuevo pedido. Define una dependencia de uno a muchos: un pedido puede notificar a múltiples interesados.

```

/*
Notificar a la cocina y al repartidor cuando hay un nuevo pedido.
Define una dependencia de 1 -> n
    1 pedido -> (cocina y repartidor)
Notificar a objetos cuando hay cambios en otros.
*/

//quiero actualizar con numero de orden a los observadores
interface Observer {
    update(orderId: string): void;
}

//Entidades que van a escuchar el pedido
class Kitchen implements Observer {
    update(orderId: string): void {
        console.log(`Cocina ha recibido el pedido con ID: ${orderId}`);
    }
}

class Delivery implements Observer {
    update(orderId: string): void {
        console.log(`Repartidor ha recibido el pedido con ID: ${orderId}`);
    }
}

class OrderSubject{
    private observers: Observer[] = [];

    public addObserver(observer:Observer){
        this.observers.push(observer)
    }

    notify(orderId:string){
        this.observers.forEach((o) => o.update(orderId));
    }
}

//uso
const subject = new OrderSubject();
const kitchen = new Kitchen();
const delivery = new Delivery();

subject.addObserver(kitchen);
subject.addObserver(delivery);

// Simular un nuevo pedido
const orderId = "12345";
subject.notify(orderId); // Notifica a la cocina y al repartidor

```

Patrones Estructurales

Patrón Decorator - Estructural

¿Qué es un decorador? Te permite extender la funcionalidad de algo que antes no la tenía. Por ejemplo, queres agregar toppings directamente a las comidas sin modificar las clases base.

Objetivo: Agregar cosas de manera dinámica sin modificar la estructura original de una clase.

¿Cuándo usarlo? Cuando necesitas agregar funcionalidades a objetos de manera flexible y dinámica, y queres evitar crear sub-clases para cada combinación posible.

```

interface IFood {
    getDescription():string;
    getCost():number;
}

class BasicFood implements IFood{
    getDescription(): string {
        return "Pizza"
    }
    getCost(): number {
        return 10
    }
}

class CheeseDecorator implements IFood{
    constructor(private food:IFood){}

    getDescription(): string {
        return `${this.food.getDescription()} con Queso Extra`
    }
    getCost(): number {
        return this.food.getCost() + 2;
    }
}

class BaconDecorator implements IFood{
    constructor(private food:IFood){}

    getDescription(): string {
        return `${this.food.getDescription()} con Bacon Extra `
    }
    getCost(): number {
        return this.food.getCost() + 4;
    }
}

//Uso

let pizza: IFood = new BasicFood()
pizza = new CheeseDecorator(pizza)
pizza = new BaconDecorator(pizza);

console.log(pizza.getDescription())// Output: "Pizza con Queso Extra con Bacon Extra"
console.log(pizza.getCost())//Output: "16"

```

Patrón Adapter - Estructural

Digamos que tenés que integrar un sistema de pagos con una API que es diferente a la tuya. El patrón Adapter va a convertir la interfaz de una clase en lo que el cliente espera. Es como un traductor entre dos sistemas que hablan diferentes idiomas.

```
// Interfaz esperada por tu sistema
interface Payment {
    pay(amount: number): void;
}

// Servicio externo con interfaz diferente
class StripeService {
    makePayment(value: number) {
        console.log(`Pagando ${value} con Stripe`);
    }
}

// Adapter que traduce entre las dos interfaces
class StripeAdapter implements Payment {
    constructor(private stripe: StripeService) {}

    pay(amount: number): void {
        this.stripe.makePayment(amount);
    }
}

// Uso
const payment: Payment = new StripeAdapter(new StripeService());
payment.pay(100); // "Pagando $100 con Stripe"
```

Patrón Composite - Estructural

El Composite Pattern nos permite tratar objetos individuales y grupos de objetos de manera uniforme. Es perfecto para crear estructuras jerárquicas como menús, combos, o cualquier situación donde tengas elementos individuales y agrupaciones de elementos.

```

interface FoodItem {
    getName(): string;
    getPrice(): number;
}

class SimpleFood implements FoodItem {
    constructor(
        private name: string,
        private price: number
    ) {}

    getName(): string {
        return this.name;
    }

    getPrice(): number {
        return this.price;
    }
}

class Combo implements FoodItem {
    private items: FoodItem[] = [];

    public addItem(item: FoodItem): void {
        this.items.push(item);
    }

    getName(): string {
        return `Combo: ${this.items.map(item => item.getName()).join(", ")}`;
    }

    getPrice(): number {
        return this.items.reduce((total, item) => total + item.getPrice(), 0);
    }
}

// Uso
const burger = new SimpleFood("Hamburguesa", 5.99);
const fries = new SimpleFood("Papas Fritas", 2.99);
const drink = new SimpleFood("Refresco", 1.99);

const combo = new Combo();
combo.addItem(burger);
combo.addItem(fries);
combo.addItem(drink);
console.log(combo.getName());// "Combo: Hamburguesa, Papas Fritas, Refresco"
console.log(combo.getPrice());// 10.97

```

Patrón Facade - Estructural

Este patrón se utiliza mucho dentro de la arquitectura hexagonal. ¿Qué problema resuelve? El problema de tener que interactuar con subsistemas complejos o con múltiples pasos.

El Facade Pattern te proporciona una interfaz simplificada para un conjunto de interfaces en un subsistema. Es como tener un control remoto universal para tu televisor, reproductor de DVD y sistema de sonido.

```

class OrderService {
    createOrder() {
        console.log("Orden creada");
    }
}

class PaymentService {
    processPayment() {
        console.log("Pago procesado");
    }
}

class DeliveryService {
    dispatchOrder() {
        console.log("Pedido despachado");
    }
}

class OrderFacade {
    constructor(
        private orderService: OrderService,
        private paymentService: PaymentService,
        private deliveryService: DeliveryService
    ) {}

    placeOrder() {
        this.orderService.createOrder();
        this.paymentService.processPayment();
        this.deliveryService.dispatchOrder();
        console.log("Pedido realizado con éxito");
    }
}

// Uso
const facade = new OrderFacade(
    new OrderService(),
    new PaymentService(),
    new DeliveryService()
);

facade.placeOrder();
// Output:// Orden creada// Pago procesado// Pedido despachado// Pedido realizado con éxito

```

Patrón Repository: - Estructural

El patrón Repository es especialmente útil en arquitecturas limpias. Es un patrón para trabajar la persistencia de datos de manera independiente y desacoplada del dominio.

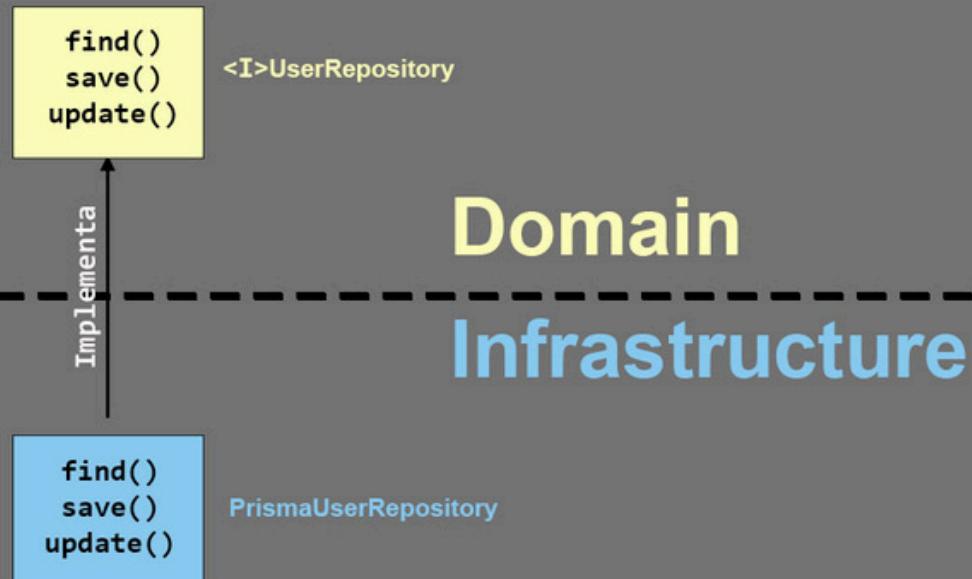
Motivación del Patrón

El patrón Repository permite trabajar la persistencia de manera separada y desacoplada de tu proyecto principal. Es un conjunto de clases que realizan toda la tarea de acceso a datos, ya sea a bases de datos, archivos, servicios en la nube, etc.

Esta abstracción queda transparente para tu proyecto principal. Puedes tener múltiples implementaciones para diferentes proveedores:

- `PrismaUserRepository`
- `MySqlUserRepository`
- `RedisUserRepository`
- `MongoDBUserRepository`

Todos estos repositorios siguen y respetan una misma interfaz, haciendo que el cambio de uno a otro sea completamente transparente para tu aplicación.



github.com/ValentinZoia

Implementación del Patrón

Paso 1: Definir la Interfaz

Primero, creamos una interfaz en la capa Domain:

```
// User/domain/interfaces/IUserRepository.ts
export interface IUserRepository {
  findById(id: string): Promise<User | null>;
  findByEmail(email: string): Promise<User | null>;
  save(user: User): Promise<void>;
  update(user: User): Promise<void>;
  delete(id: string): Promise<void>;
}
```

Paso 2: Crear el Caso de Uso e injectar la interfaz.

```
// User/application/use-cases/UserRegisterUseCase.ts
export class UserRegisterUseCase {
    constructor(private readonly repository: IUserRepository) {}

    async register(id: string, email: string): Promise<void> {
        const user = new User(id, email);
        await this.repository.save(user);
    }
}
```

Ahora solo quedaría crear las implementaciones que queramos usar, por ejemplo las que dije antes.

PrismaUserRepository , MySqlUserRepository , etc. Lo importante es que implementen la interfaz **IUserRepository**

Paso 3: Crear Implementaciones Concretas

```
// User/infrastructure/repositories/PrismaRepositoryImpl.ts
export class PrismaRepositoryImpl implements IUserRepository {
    async save(user: User): Promise<void> {
        // Implementación específica de Prisma
    }

    async update(user: User): Promise<void> {
        // Implementación específica de Prisma
    }

    async findByEmail(email: string): Promise<User | null> {
        // Implementación específica de Prisma
    }

    async findById(id: string): Promise<User | null> {
        // Implementación específica de Prisma
    }

    async delete(id: string): Promise<void> {
        // Implementación específica de Prisma
    }
}
```

Las implementaciones deben cumplir con todos los métodos de la interfaz, en el ejemplo de arriba no los completé porque no viene al caso. Pero recordá eso, deben usar todos los métodos, save, findById, findByEmail, delete. Y dentro de cada ellos, estará el código correspondiente a el proveedor que estas usando, en prisma será el cliente de prisma, y en sql será la creación de query, etc. Pero eso no importa, lo importante es que cumpla con los nombres de los métodos y los tipos de parámetros y retornos establecidos en la interfaz.

Paso 4: Inyección de Dependencias

```
// En tu composition root
const repository = new PrismaRepositoryImpl();
const userRegisterUseCase = new UserRegisterUseCase(repository);

// Para cambiar a MySQL:
const repository = new MySqlRepositoryImpl();
const userRegisterUseCase = new UserRegisterUseCase(repository);
```

Con solo cambiar el repositorio estaremos usando el proveedor de persistencia de datos que nos plazca, sin errores en ninguna parte del código, así de fácil.

Beneficios del Patrón Repository

- 1. Desacoplamiento:** La lógica de negocio no depende de detalles de persistencia
- 2. Testabilidad:** Puedes crear implementaciones mock para testing
- 3. Flexibilidad:** Cambiar proveedores de datos sin afectar el código de negocio
- 4. Mantenibilidad:** Cada implementación está aislada y es fácil de mantener

Capítulo 1.9: El Camino del Aprendizaje

Conceptos Fundamentales

Para dominar la arquitectura de software, es esencial comprender conceptos que van más allá del código:

- Atributos de calidad: Escalabilidad, seguridad, mantenibilidad, performance
- Características arquitectónicas: Cómo estos atributos se manifiestan en el código
- Patrones arquitectónicos: Soluciones probadas para problemas comunes

Preguntas Clave

Desarrolla el hábito de hacerte preguntas arquitectónicas:

- ¿Qué patrón arquitectónico tiene mi aplicación?
- ¿Qué atributos de calidad son prioritarios?
- ¿Por qué se eligió esta tecnología y no otra?
- ¿Cómo escalaría este sistema bajo carga?

Progresión Recomendada

1. Comienza con arquitectura por capas: Es la más sencilla de implementar
2. Practica los principios SOLID: Especialmente la responsabilidad única
3. Explora patrones de diseño: Repository, Builder, Factory, Strategy, Observer
4. Implementa monolitos modulares: Cada módulo puede convertirse en un microservicio

5. Avanza hacia arquitecturas más complejas: Hexagonal, DDD, Event-Driven

Capítulo 1.10: Bounded Contexts y Modularidad

Definiendo Límites Conceptuales

Los **Bounded Contexts** son áreas de conocimiento que funcionan como departamentos conceptuales dentro de tu aplicación. Cada contexto tiene su propio lenguaje ubicuo y sus propias reglas.

En un sistema de e-commerce, podrías tener contextos como:

- **Catálogo:** Productos, categorías, inventario
- **Ventas:** Órdenes, pagos, facturación
- **Usuarios:** Autenticación, perfiles, preferencias
- **Logística:** Envíos, tracking, devoluciones

Módulos como Objetos de Dominio

Los módulos representan los objetos centrales de tu dominio: Product, User, Order, Payment. Cada módulo encapsula toda la lógica relacionada con su concepto específico.

Escalabilidad Arquitectónica

El objetivo es definir una arquitectura que escale a lo largo del tiempo, tolerante a la incorporación de nuevos elementos y altamente testeable. Esta escalabilidad se logra mediante:

- Límites bien definidos entre módulos
- Interfaces estables entre capas
- Principios de diseño consistentes
- Separación clara de responsabilidades

Conclusión

La arquitectura limpia no es simplemente una moda tecnológica; es una filosofía de desarrollo que reconoce que el software exitoso debe ser construido para durar. Al centrar nuestros esfuerzos en el dominio y separar claramente las responsabilidades, creamos sistemas que pueden evolucionar con el tiempo sin requerir reescrituras completas.

El precio de esta durabilidad es la complejidad inicial: más abstracciones, más archivos, más interfaces. Sin embargo, esta inversión se amortiza rápidamente cuando el sistema crece y los cambios se vuelven más frecuentes.

Recuerda que tener una arquitectura es mejor que no tener ninguna. La arquitectura limpia, junto con DDD y la arquitectura hexagonal, representan algunas de las mejores prácticas disponibles para construir software empresarial robusto.

El verdadero valor de una arquitectura limpia se manifiesta cuando puedes realizar cambios significativos en una parte del sistema sin afectar otras partes. Cuando puedes cambiar de base de datos, framework o servicio externo sin tocar tu lógica de negocio. Cuando nuevos desarrolladores pueden entender rápidamente qué hace tu sistema simplemente observando su estructura. En última instancia, la arquitectura limpia es una inversión en el futuro de tu software y en la productividad de tu equipo. Es la diferencia entre construir un sistema que perdure décadas y uno que requiera reescrituras constantes.

Capítulo 2.1 - Parte Práctica

En esta parte práctica, vamos a construir un sistema de autenticación completo usando **Node + Express + TypeScript + JWT + Prisma + MongoDB + Clean Architecture + Validaciones con Zod**. Vamos a aplicar todo lo que vimos en el capítulo anterior, construyendo un proyecto mantenable, escalable y legible.

Preparando el Proyecto Base

Antes de empezar, tengo una plantilla base en mi GitHub que ya tiene configurado Node + Express + TypeScript + Clean Architecture. Podés usarla como punto de partida: github.com/ValentinZoia/plantilla-ts-clean-architecture

Esta plantilla ya incluye:

- Node.js + Express + TypeScript configurado
- JWT y Bcrypt instalados
- Estructura de carpetas base
- Scripts de desarrollo

Lo único que necesitamos agregar es Prisma para manejar la base de datos.

Arquitectura del Proyecto

Primero hablemos de la estructura. Hay muchas formas de organizar un proyecto usando Clean Architecture, pero la mejor para mí es separar en primer nivel por Dominio (quién eres: User, Product, Order, etc.) y por dentro de cada una separar por capas (Domain, Application, Infrastructure, Presentation).

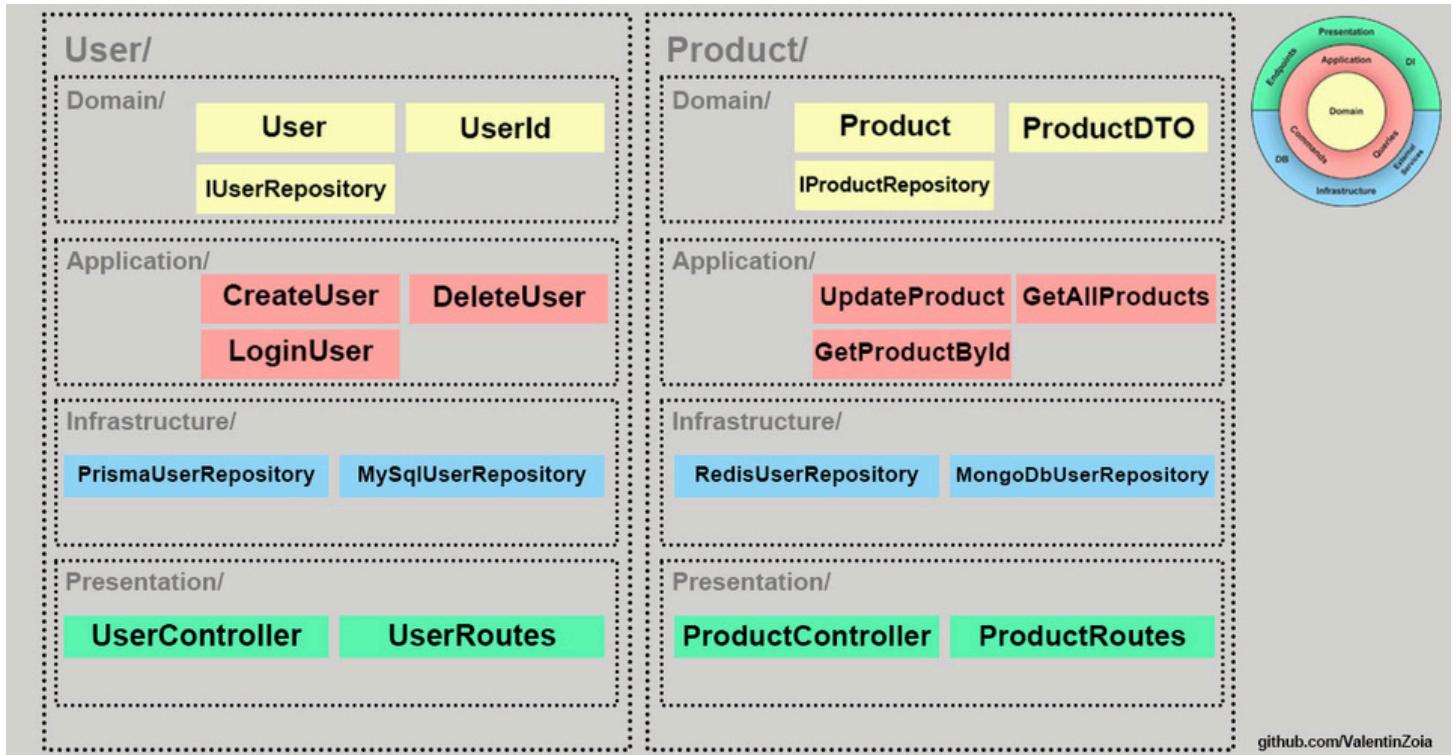
Esta forma de estructurar aporta mucha más semántica de negocio y hace que todo escale mejor. Tendremos tantas carpetas a primer nivel como entidades tengamos en nuestro negocio. Si querés hacer una nueva feature para usuarios, sabés que todo lo que tenés que tocar está dentro de la carpeta **Users**

✗ Forma incorrecta: Separar por capas primero

```
src/
├── domain/
│   ├── User.ts
│   ├── Product.ts
│   └── Order.ts
└── application/
    ├── UserService.ts
    ├── ProductService.ts
    └── OrderService.ts
```

✓ Forma correcta: Separar por dominio primero

```
src/
├── Users/
│   ├── domain/
│   ├── application/
│   ├── infrastructure/
│   └── presentation/
└── Products/
    ├── domain/
    ├── application/
    ├── infrastructure/
    └── presentation/
```



Instalando y Configurando Prisma

Prisma es un ORM (Object-Relational Mapping) moderno que nos permite interactuar con la base de datos de manera más intuitiva. En lugar de escribir SQL crudo, podemos usar JavaScript/TypeScript.

```
# Instalar Prisma
npm install prisma @prisma/client

# Inicializar Prisma
npx prisma init
```

Después de ejecutar `npx prisma init`, Prisma crea:

- Un archivo `prisma/schema.prisma` para definir tu modelo de datos
- Un archivo `.env` para las variables de entorno
- Una carpeta `src/generated` que contiene el cliente de Prisma generado automáticamente

¿Qué hace Prisma?

- **Mapeo de objetos:** Convierte filas de la base de datos en objetos JavaScript
- **Type Safety:** Genera tipos TypeScript automáticamente
- **Migraciones:** Maneja cambios en la estructura de la base de datos
- **Query Builder:** Permite hacer consultas complejas sin SQL

Entendiendo JWT (JSON Web Tokens)

JWT es un estándar para transmitir información de manera segura entre partes. Imaginate que es como un carnet de identidad digital.

¿Cómo funciona?

1. El usuario se loguea con credenciales
2. El servidor genera un JWT firmado digitalmente
3. El cliente guarda este token (generalmente en cookies)
4. En cada petición, el cliente envía el token
5. El servidor verifica la firma y extrae la información

Estructura de un JWT:

`header.payload.signature`

¿Por qué usar JWT?

- Stateless: No necesitás guardar sesiones en el servidor
- Portable: Funciona entre diferentes dominios
- Seguro: Está firmado digitalmente
- Escalable: Perfecto para microservicios

Entendiendo Bcrypt

Bcrypt es una función de hash diseñada específicamente para hashear contraseñas de manera segura.

¿Por qué no guardar contraseñas en texto plano?

- Si hackean tu base de datos, todas las contraseñas quedan expuestas
- Incluso los desarrolladores pueden ver las contraseñas

¿Cómo funciona Bcrypt?

1. Toma la contraseña original
2. Agrega un "salt" (datos aleatorios)
3. Aplica el algoritmo de hash múltiples veces
4. Genera un hash único y prácticamente irreversible

Contraseña original: "miPassword123"

Hash generado:

"\$2b\$10\$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p921d
Gxad68LJZdL17lhWy"

¿Por qué es seguro?

- **Irreversible:** No podés obtener la contraseña original del hash
- **Salt único:** Cada contraseña tiene un salt diferente
- **Slow by design:** Tarda tiempo en procesarse, dificultando ataques de fuerza bruta

Estructura Final del Proyecto

```
src/
|
+-- Users/           # Dominio de usuarios
|   +-- domain/      # Lógica de negocio pura
|   +-- application/ # Casos de uso
|   +-- infrastructure/ # Persistencia de datos
|   +-- presentation/ # Controllers y rutas
|
+-- Shared/          # Utilidades compartidas
|   +-- domain/      # Errores y entidades base
|   +-- application/ # Servicios base
|   +-- infrastructure/ # Adapters y base de datos
|   +-- presentation/ # Middlewares y router principal
|
+-- generated/       # Generado automáticamente por Prisma
+-- app.ts            # Configuración de Express
+-- server.ts         # Punto de entrada
```

Empezando con la Carpeta Shared

Antes de crear el módulo **Users**, empecemos por la carpeta **Shared**. Esta va a contener todas las utilidades que pueden ser reutilizadas por cualquier módulo de la aplicación.

¿Por qué empezar por Shared? Aunque ahora solo tengamos el módulo **Users**, estamos pensando en escalabilidad. Mañana podríamos agregar **Products**, **Orders**, **Posts**, etc. Tener una base sólida de utilidades compartidas nos va a ahorrar mucho tiempo.

Shared/Domain - Los Cimientos

La capa domain de Shared contiene nuestros errores personalizados:

```
//src/Shared/domain/errors/custom.error.ts
export class CustomError extends Error {
    constructor(
        public readonly statusCode: number,
        public readonly message:string,
        readonly type: string,
    ) {
        super(message);
        this.name = "CustomError";
    }

    static badRequest(message: string) {
        return new CustomError(400, message, 'BadRequestError');
    }
    static unauthorized(message: string) {
        return new CustomError(401, message, 'UnauthorizedError');
    }
    static forbidden(message: string) {
        return new CustomError(403, message, 'ForbiddenError');
    }
    static notFound(message: string ) {
        return new CustomError(404, message, 'NotFoundError');
    }
    static internalServerError(message: string = 'Internal Server Error') {
        return new CustomError(500, message, 'InternalServerError');
    }
}
```

Esta clase nos permite crear errores HTTP de manera consistente. En lugar de retornar strings o números, tenemos objetos con información estructurada.

```
//src/Shared/domain/errors/validation.error.ts
export class ValidationError extends Error{
    constructor(
        public readonly errors: Record<string, string[]>,
        message: string = 'Validation failed',
        readonly statusCode: number = 400,
    ) {
        super(message);
        this.name = 'ValidationError';
    }
}
```

ValidationError es específico para errores de validación. Almacena múltiples errores por campo, perfecto para formularios.

Shared/Application - La Base de los Servicios

```
//src/Shared/application/base/Service.ts
export abstract class Service<Input extends any[], Output>{
    abstract execute(...data:Input): Promise<Output>;
}
```

Esta clase abstracta define el contrato que deben cumplir todos nuestros casos de uso. Usando generics, podemos especificar los tipos de entrada y salida para cada servicio.

Shared/Infrastructure - Los Adapters

Los adapters son el corazón de la capa de infraestructura.
Nos permiten abstraer dependencias externas:

```
//src/Shared/infrastructure/adapters/envs.ts
import 'dotenv/config';
import {get} from 'env-var';

export const envs ={
  PORT: get('PORT').required().asPortNumber(),
  JWT_SECRET: get('JWT_SECRET').required().asString(),
  CORS_ORIGIN: get('CORS_ORIGIN').required().asString(),
  SALT_ROUNDS: get('SALT_ROUNDS').default('10').asIntPositive(),
  DATABASE_URL: get('DATABASE_URL').required().asString(),
  NODE_ENV: get('NODE_ENV').default('development').asString(),
}
```

Este adapter centraliza todas las variables de entorno y las valida. La librería `env-var` nos asegura que las variables tengan el tipo correcto.

```
//src/Shared/infrastructure/adapters/bcrypt.adapter.ts
import { compareSync, hashSync } from "bcryptjs"
import { envs } from "./envs";

const SALT_ROUNDS = envs.SALT_ROUNDS;

export class BcryptAdapter {
    static hash(password:string): string {
        return hashSync(password, SALT_ROUNDS);
    }

    static compare(password:string, hashed:string): boolean{
        return compareSync(password, hashed);
    }
}
```

El **BcryptAdapter** encapsula toda la lógica de hashing. Si mañana queremos cambiar de bcrypt a otra librería, solo modificamos este archivo.

```

//src/Shared/infrastructure/adapters/jwt.adapter.ts
import jwt from 'jsonwebtoken';
import { envs } from './envs';

//Esto es una dependencia oculta, mejor inyectarla desde el constructor
const JWT_SECRET = envs.JWT_SECRET;

export class JwtAdapter{
    static async generateToken(
        payload: Record<string, any>,
        options: Record<string, any> = { expiresIn: '5h' }
    ): Promise<string | null> {
        return new Promise((resolve)=>{
            jwt.sign(payload, JWT_SECRET,options,(err, token) =>{
                if(err) return resolve(null);
                resolve(token!);
            });
        })
    }

    static validateToken<T>(token:string): Promise<T | null>{
        return new Promise((resolve)=>{
            jwt.verify(token, JWT_SECRET , (err, decoded) =>{
                if (err) return resolve(null)

                resolve(decoded as T)
            })
        })
    }
}

```

El **JwtAdapter** maneja toda la lógica de tokens JWT. Fijate que usamos Promises en lugar de callbacks para mantener consistencia con el resto del código.

Shared/Presentation - Middlewares Globales

```
//src/Shared/presentation/middlewares/validationMiddleware.ts
import { Request, Response, NextFunction } from "express";
import { z } from "zod";

export class ValidationMiddleware {
    static validateBody(schema: z.ZodSchema) {
        return (req: Request, res: Response, next: NextFunction) => {
            try {
                // Validar el cuerpo de la solicitud
                req.body = schema.parse(req.body);
                next();
            } catch (error) {
                throw error;
            }
        };
    }
}
```

Este middleware valida el cuerpo de las peticiones HTTP usando esquemas de Zod. Es reutilizable para cualquier endpoint.

```

//src/Shared/presentation/middlewares/errorHandler.ts
import { PrismaClientKnownRequestError } from ".....generated/prisma/runtime/library";
import { Request, Response, NextFunction } from "express";
import { ValidationError, CustomError } from "../../domain/errors";
import {z} from 'zod';

export const errorHandler = (
  err: Error,
  req: Request,
  res: Response,
  next: NextFunction
): void => {

  if (err instanceof z.ZodError) {
    const formattedErrors: Record<string, string[]> = {};
    err.errors.forEach((_err) => {
      const path = _err.path.join(".");
      if (!formattedErrors[path]) {
        formattedErrors[path] = [];
      }
      formattedErrors[path].push(_err.message);
    });
    throw new ValidationError(formattedErrors);
  }

  //Error de validacion personalizado
  if (err instanceof ValidationError) {
    res.status(400).json({
      type: "ValidationError",
      statusCode: err.statusCode || 400,
      message: err.message,
      errors: err.errors,
    });
    return;
  }

  if(err instanceof CustomError){

    res.status(err.statusCode).json({
      type: err.type || 'CustomError',
      message: err.message,
      statusCode: err.statusCode ,
    })
  }
}

```

```

//Error prisma
if (err instanceof PrismaClientKnownRequestError) {
// P2002 es el código para violación de restricción única
  if (err.code === "P2002") {
    res.status(409).json({
      type: "UniqueConstraintError",
      message: "Ya existe un registro con este valor",
    });
    return;
  }

// P2003 es el código para violación de restricción de clave foránea
  if (err.code === 'P2003') {
    res.status(400).json({
      type: 'ForeignKeyConstraintError',
      message: 'El valor de referencia no existe',
    });
    return;
  }

// P2025 es el código para error de no encontrado
  if (err.code === 'P2025') {
    res.status(404).json({
      type: 'NotFoundError',
      message: 'Recurso no encontrado'
    });
    return;
  }

  if(err.code === 'P2023'){
    res.status(404).json({
      type: 'InconsistentColumnData',
      message: 'Valor inválido de identificador',
    });
    return;
  }
}

// Error general del servidor
res.status(500).json({
  message: `Error del Servidor`
});

}

```

Este middleware maneja todos los errores de la aplicación de manera centralizada. Identifica el tipo de error y devuelve una respuesta HTTP apropiada.

```

//src/Shared/presentation/middlewares/authMiddleware.ts
import { IUserRepository } from "../../Users/domain/interfaces";
import { Request, Response, NextFunction } from "express";
import { CustomError } from "../../domain/errors/custom.error";
import { JwtAdapter } from "../../infrastructure/adapters";

type JwtUserPayload = {
  id: string;
  username: string;
}

export class AuthMiddleware {
  constructor(private readonly userRespository: IUserRepository) {}

  authenticate = async (
    req: Request,
    res: Response,
    next: NextFunction
  ): Promise<void> => {
    try {
      //extrae el token de la cookie con nombre 'access_token'
      const token = req.cookies.access_token;

      //si no existe ese token, o esa cookie. Devuelve un error con acceso denegado.
      if (!token) {
        throw CustomError.unauthorized(
          "Acceso denegado. No se proporcionó un token de autenticación."
        );
      }

      //si hay token, lo verifica. mediante esta inyección de dependencia oculta.
      const payload = await JwtAdapter.validateToken<JwtUserPayload>(token);

      //si esa verificación da error, lo devolvemos.
      if (!payload) {
        throw CustomError.unauthorized(
          "Token de autenticación inválido o expirado."
        );
      }

      const user = await this.userRespository.findUserByUsername(payload.username);
      if (!user) throw CustomError.unauthorized("Token invalido - Usuario no encontrado.");

      //Si todo es correcto, Añadir información del admin al objeto request (id, username).
      req.user = { id: payload.id, username: payload.username } as JwtUserPayload;
      next();
    } catch (error) {

```

```

        } catch (error) {
            next(error);
        }
    };
}

// Extendemos el tipo Request de Express para incluir el admin
declare global {
    namespace Express {
        interface Request {
            user?: JwtUserPayload
        }
    }
}

```

Este middleware verifica que el usuario esté autenticado. Extrae el token JWT de las cookies, lo valida y agrega la información del usuario al objeto **request**.

```

//src/Shared/presentation/routes/AppRouter.route.ts
import { AuthUserRoutes } from '../../../../../Users/presentation/routes';
import { Router } from 'express';

export class AppRoutes {

    static get routes(): Router {

        const router = Router();

        // Definir todas mis rutas principales
        router.use('/api/users', AuthUserRoutes.routes )

        //Aca seguirian las rutas dependiendo de tu app. Ejemplo:
        //router.use('/api/products', ProductRoutes.routes)//router.use('/api/categories', CategoryRoutes.routes)

        return router;
    }
}

```

El **AppRouter** centraliza todas las rutas de la aplicación. Cada módulo registra sus rutas aquí.

Módulo Users - Nuestro Primer Dominio

Ahora que tenemos la base compartida, podemos crear nuestro primer módulo: **Users**.

Users/Domain - El Corazón del Negocio

```
//src/Users/domain/entities/User.entity.ts
export class User {
    constructor(
        public readonly id: string | null ,//mongodb otorga una id, por eso el null.
        public readonly username: string,
        public readonly password: string,
    ) {}
}
```

La entidad **User** representa un usuario en nuestro dominio. Es independiente de la base de datos y contiene solo la lógica de negocio pura.

```

//src/Users/domain/dtos/CreateUser.dto.ts
import { z } from 'zod';

export const createAdminSchema = z.object({
  username: z.string()
    .min(3, 'Username debe tener al menos 3 caracteres')
    .max(20, 'Username no puede exceder 20 caracteres')
    .regex(/^[a-zA-Z0-9_]+$/, 'Username solo puede contener letras, números y guiones bajos')
    .trim()
    .refine((val) => val === val.toLowerCase(), {
      message: "El username debe estar en minúsculas.",
    })
    .refine((val) => !/\s/.test(val), {
      message: "El username no debe contener espacios.",
    }),
  password: z.string()
    .min(8, 'Password debe tener al menos 8 caracteres')
    .regex(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/, 'Password debe contener mayúscula, minúscula y número'),
  id: z.string().uuid().optional() // Si lo generas automáticamente
});

export type CreateAdminDTO = z.infer<typeof createAdminSchema>

export class CreateAdminDto {
  private constructor(
    public username: string,
    public password: string,
    public id?: string // Opcional si se genera automáticamente
  ) {}

  static create(data: unknown): CreateAdminDto{
    const parsedData = createAdminSchema.parse(data);
    return new CreateAdminDto(parsedData.username, parsedData.password);
  }
}

```

Los DTOs (Data Transfer Objects) definen cómo se transfieren los datos entre capas. Usamos Zod para validar y transformar la información de entrada.

```
//src/Users/domain/interfaces/IUserRepository.ts
import { CreateUserDto } from "../dtos/CreateUser.dto";
import { User } from "../entities/User.entity";

export interface IUserRepository {
    create(user:CreateUserDto):Promise<User>
    findUserByUsername(username: string): Promise<User | null>;
}
```

La interfaz **IUserRepository** define el contrato para la persistencia de usuarios. La capa de dominio no sabe cómo se implementa, solo define qué operaciones necesita.

Users/Application - Los Casos de Uso

```
//src/Users/application/services/CreateUserService.ts
import { IUserRepository } from "../../domain/interfaces";
import { ValidationError } from "../../shared/errors";
import { BcryptAdapter } from "../../shared/adapters";
import { CreateUserDto } from "../../domain/dtos";
import { Service } from "../../shared/base";

interface PublicUserResponse {
  id: string;
  username: string;
}

type HashFunction = (password:string) => string

export class CreateUserService extends Service<
  [CreateUserDto],
  PublicUserResponse
> {
  constructor(
    private userRepository: IUserRepository,
    private readonly hashPassword: HashFunction = BcryptAdapter.hash,
  ) {
    super();
  }

  async execute(createUserDto: CreateUserDto): Promise<PublicUserResponse> {
    const { username, password, id } = createUserDto;

    // 1. Validar que el username no este en uso
    const userFound = await this.userRepository.findUserByUsername(username);
    if (userFound !== null) throw new ValidationError({ username: ["El username ya esta en uso"] });

    // 2. Hashear contraseña
    const hashedPassword = this.hashPassword(password);

    //3. Crear usuario
    const createdUser = await this.userRepository.create({
      id,
      username,
      password: hashedPassword,
    });

    return {
      id: createdUser.id || "",
      username: createdUser.username,
    };
  }
}
```

Este servicio maneja la lógica de crear un usuario. Fíjate que recibe las dependencias por constructor (inyección de dependencias) y sigue un flujo claro: validar, hashear, crear.

```

//src/Users/application/services/LogInUserService.ts
import { CustomError, ValidationError } from "../../../../shared/errors";
import { BcryptAdapter, JwtAdapter } from "../../../../shared/adapters";
import { IUserRepository } from "../../domain/interfaces";
import { LogInUserDto } from "../../domain/dtos";
import { Service } from "../../../../shared/base";

type CompareFunction = (password: string, hashed: string) => boolean;
type TokenResponse = string;

export class LogInUserService extends Service<[LogInUserDto], TokenResponse> {
  constructor(
    private readonly userRepository: IUserRepository,
    private readonly comparePassword: CompareFunction = BcryptAdapter.compare
  ) {
    super();
  }

  async execute(logInUserDto: LogInUserDto): Promise<TokenResponse> {
    const { username, password } = logInUserDto;

    //1. Busco si existe un usuario con ese username, si no existe, retorno un error.
    const userFound = await this.userRepository.findUserByUsername(username);
    if (!userFound) throw new ValidationError({ credentials: ["Credenciales incorrectas"] });

    //2. Comparo las contraseñas, si no coinciden, retorno un error.
    const isPasswordMatching = this.comparePassword(
      password,
      userFound.password
    );
    if (!isPasswordMatching) throw new ValidationError({ credentials: ["Credenciales incorrectas"] });

    //3. Si todo esta correcto, creo el token de sesion con la informacion publica del usuario.
    const token = await JwtAdapter.generateToken({
      id: userFound.id,
      username: userFound.username,
    });
    if (!token) throw CustomError.internalServerError("Error al generar JWT");

    return token;
  }
}

```

El servicio de login verifica las credenciales y devuelve un token JWT. El flujo es: buscar usuario, comparar contraseña, generar token.

Users/Infrastructure - La Persistencia

```
//src/Users/infrastructure/repositories/PrismaUserRepositoryImpl.ts
import prisma from "../../shared/database/prismaClient";
import { IUserRepository } from "../../domain/interfaces";
import { ValidationError } from "../../shared/errors";
import { CreateUserDTO } from "../../domain/dtos";
import { User } from "../../domain/entities";

export class PrismaUserRepositoryImpl implements IUserRepository {
    async create(user: CreateUserDTO): Promise<User> {
        const { username, password, id } = user;

        const createdUser = await prisma.user.create({
            data: {
                id: id,
                username: username,
                password: password,
            },
        });
        //Mapear el objeto creado a la entidad User
        return this.mapPrismaToUser(createdUser);
    }

    async findUserByUsername(username: string): Promise<User | null> {
        const existingUser = await prisma.user.findUnique({
            where: { username },
        });
        if (!existingUser) return null;
        //Mapear el objeto encontrado a la entidad User
        return this.mapPrismaToUser(existingUser);
    }

    private mapPrismaToUser(object: { [key: string]: any }): User {
        const { id, _id, username, password } = object;

        if (!_id || !id) throw new ValidationError({ id: ["Missing id"] });

        if (!username) throw new ValidationError({ username: ["Missing username"] });

        if (!password) throw new ValidationError({ password: ["Missing password"] });

        return new User(_id || id, username, password);
    }
}
```

Este repositorio implementa la interfaz IUserRepository usando Prisma. El método mapPrismaToUser convierte los objetos de Prisma a entidades de dominio.

Users/Presentation - La Interfaz HTTP

```
//src/Users/presentation/controllers/UserController.ts
import { CreateUserService, LogInUserService } from "../../application/services";
import { CreateUserDto } from "../../domain/dto";
import { LoginUserDto } from "../../domain/dtos";
import { Request, Response, NextFunction } from "express";

export class UserController {
  // DI
  constructor(
    private readonly createUserService: CreateUserService,
    private readonly logInUserService: LogInUserService,
  ) {}

  createUser = async (
    req: Request,
    res: Response,
    next: NextFunction
  ): Promise<void> => {
    try {
      //realmente aca la request ya esta validada por el middleware, pero lo hacemos igual.

      const createUserDto = CreateUserDto.create(req.body);

      const newUser = await this.createUserService.execute(createUserDto);
      res.status(201).json(newUser);
    } catch (error) {
      next(error);
    }
  };
}
```

```

login = async (
  req: Request,
  res: Response,
  next: NextFunction
): Promise<void> => {
  try {
    const loginUserDto = LoginUserDto.create(req.body);

    const token = await this.logInUserService.execute(loginUserDto);
    // Configurar la cookie con el token JWT
    res.cookie('access_token', token, {
      httpOnly: true,
      secure: process.env.NODE_ENV === 'production',// Solo en HTTPS en producción
      maxAge: 5 * 60 * 60 * 1000,// 5 horas
      sameSite: 'strict'
    });

    res.status(200).json({
      success: true,
      message: 'Login exitoso',
    });
  } catch (error) {
    next(error);
  }
};

async logout(req: Request, res: Response): Promise<void> {
// Eliminar la cookie
  res.clearCookie('access_token');
  res.status(200).json({
    success: true,
    message: 'Logout exitoso'
  });
}
}

```

El controlador maneja las peticiones HTTP. Recibe los servicios por constructor y delega la lógica de negocio a los casos de uso.

```
//src/Users/presentation/routes/UserRoute.ts

import { CreateUserService, LogInUserService } from "../../application/services";
import { AuthMiddleware, ValidationMiddleware } from "../../../../shared/middlewares";
import { PrismaUserRepositoryImpl } from "../../infrastructure/repositories";
import { createUserSchema, loginUserSchema } from "../../domain/dtos";
import { BcryptAdapter } from "../../../../shared/adapters";
import { UserController } from "../controllers";
import { Router } from "express";

export class AuthUserRoutes {
  static get routes(): Router {
    const router = Router();
    const repository = new PrismaUserRepositoryImpl();
    const createUserService = new CreateUserService(
      repository,
      BcryptAdapter.hash
    );
    const logInUserService = new LogInUserService(
      repository,
      BcryptAdapter.compare
    );
    const authMiddleware = new AuthMiddleware(repository);
    const controller = new UserController(
      createUserService,
      logInUserService
    );

    //crear usuario
    router.post(
      "/",
      ValidationMiddleware.validateBody(createUserSchema),
      controller.createUser.bind(controller)
    );

    //login
    router.post(
      "/login",
      ValidationMiddleware.validateBody(loginUserSchema),
      controller.login.bind(controller)
    );
  }
}
```

```

// a estas rutas solo deberan acceder usuarios logeados. por eso el uso del middleware para
// verificar su session.
router.post(
  "/logout",
  authMiddleware.authenticate,
  controller.logout.bind(controller)
);

router.get("/", authMiddleware.authenticate, (req, res) => {
  res.json({ user: req.user });
});

return router;
}
}

```

En este último archivo, tendremos todos los endpoints del módulo **Users**. Es el archivo ‘composition root’, donde se instancian todas las clases y se inyectan las correspondientes dependencias.

Fin

Este es el final de este pequeño libro tanto teórico como práctico sobre las arquitecturas limpias empleadas actualmente en el software, algunos patrones de diseños y buenas prácticas. Fue escrito y pensado desde mi más humilde conocimiento, así que espero que les haya servido.



