



Gentleman Programming Book

“ A clean programmer is the best kind of programmer ”
- by *Alan Buscaglia*

Chapter 1 ^

Código limpio y agilidad

- ❖ Problemas del enfoque en cascada
- ❖ ¿Por qué ser ágil?
- ❖ Por qué crees que estás haciendo agile pero en realidad... no lo estás haciendo
- ❖ Programación extrema
- ❖ Desarrollo guiado por pruebas (TDD)
- ❖ Diseño atómico, desde la perspectiva del Front End
- ❖ Programación funcional
- ❖ Historias de usuario y TDD

Chapter 2 ^

Comunicación en primer lugar

- ❖ Las empresas no están limitadas a una ubicación en particular
- ❖ Coordinación a través de diferentes zonas horarias
- ❖ Existen otras formas sencillas de mejorar esta mentalidad
- ❖ Creando conexión

Chapter 3 ^

Arquitectura Hexagonal

- ❖ Hexágono y sus actores
- ❖ Puertos y recursos
- ❖ Tipos de lógica en un servicio
- ❖ Ejemplo de aplicación: La pizzería en forma de hexágono
- ❖ Pasos recomendados para trabajar en arquitectura hexagonal con ejemplo de aplicación

Chapter 4 ^

GoLang

- ❖ Cómo utilizar GoLang
- ❖ Ventajas
- ❖ Estructura recomendada
- ❖ ¿Cómo funciona GoLang?
- ❖ Tipos de datos

- » Estructuras
- » Arreglos
- » Método Make
- » Punteros
- » Valores Predeterminados
- » Bucle Range
- » Maps
- » Mutando Maps
- » Funciones
- » Valores de Funciones
- » Closure
- » Métodos
- » Interfaces
- » Valores de Interfaces con Nil
- » Interfaces Vacías
- » Aserción de tipo
- » Switch de tipo
- » Stringers
- » Errores
- » Lectores
- » Imágenes
- » GoRoutines
- » Canales
- » Canales con Búfer

- ❖ Range y Close
- ❖ GoRoutines Select
- ❖ Mutex

Chapter 5 ^

Guía Gentil de NVIM

- ❖ ¿Qué es NVIM?
- ❖ ¿Por qué amo NVIM?
- ❖ Instalación
- ❖ Configuración
- ❖ Conceptos Básicos
- ❖ Modo Visual
- ❖ Modo de Bloque Visual
- ❖ Modo de Línea Visual
- ❖ Modo de Inserción
- ❖ Modo de Comandos
- ❖ Movimientos en NVIM
- ❖ Buffers
- ❖ Marcas
- ❖ Grabaciones

Chapter 6 ^

Algoritmos a la Manera Caballerosa

- ❖ Notación Big O

- ❖ Conocimientos Previos Necesarios
- ❖ Tipos de Notación Big O
- ❖ Ejemplos Usando Código
- ❖ Complejidad en el Peor Caso, el Mejor Caso y el Caso Promedio
- ❖ Complejidad Espacial
- ❖ Arreglos
- ❖ ¿Cómo pensar?
- ❖ Búsqueda Lineal
- ❖ Búsqueda Binaria
- ❖ Bubble Sort

Chapter 7 ^

Gentleman del Código: Dominando la Clean Architecture

- ❖ Introducción al libro "Gentleman del Código: Dominando la Clean Architecture"
- ❖ Capítulo 1: "¡Descubriendo la Clean Architecture!"
- ❖ Capítulo 2: "Separación de preocupaciones: La clave de una arquitectura eficiente"
- ❖ Capítulo 3: "Patrones de diseño vs. Arquitecturas: Entendiendo las diferencias"
- ❖ Capítulo 4: "Mantenibilidad y Escalabilidad con Clean Architecture"
- ❖ Capítulo 5: "Plugins en la arquitectura: Flexibilidad y Adaptabilidad"
- ❖ Capítulo 6: "Desventajas y Consideraciones Temporales de Clean Architecture"

- ❖ Capítulo 7: "Estructura de la Clean Architecture: Una Visión en Capas"
- ❖ Capítulo 8: "Casos de Uso y Dominio en Clean Architecture: Diferencia entre Lógica de Negocios y Aplicación"
- ❖ Capítulo 9: "Implementación Práctica de Casos de Uso y Dominio en Clean Architecture"
- ❖ Capítulo 10: "Tecnologías y Herramientas: Alineación con Casos de Uso y Dominio en Clean Architecture"
- ❖ Capítulo 11: "Mantenimiento y Gestión a Largo Plazo en Clean Architecture"
- ❖ Capítulo 12: "Diferencias entre Lógica de Aplicación, Lógica de Dominio y Lógica de Empresa en Clean Architecture"
- ❖ Capítulo 13: "Adaptadores: Rompiendo Esquemas en Clean Architecture"
- ❖ Capítulo 14: "La Capa Externa en Clean Architecture: Conectando Tu Aplicación con el Mundo Exterior"
- ❖ Capítulo 15: "Rendimiento y Seguridad en la Capa Externa: Optimizando la Interacción con el Mundo Exterior"
- ❖ Conclusiones del Libro: "Gentleman del Código: Dominando la Clean Architecture"

Chapter 8 ^

Aplicación de Clean Architecture en el Front End

- ❖ Conceptos Clave de Clean Architecture
- ❖ Ejemplo
- ❖ Estructura Propuesta para la Aplicación Bancaria
- ❖ Dominio (Entities Models y Business Rules)

- » Casos de Uso (Use Cases)
- » Adaptadores (Interface Adapters)
- » Frameworks y Drivers
- » Enfoque Orgánico de la Estructura de Carpetas
- » Introducción a la Regla del Alcance
- » Aplicando Clean Architecture con la Regla del Alcance
- » Estructura Modular por Funcionalidad
- » Componente Contenedor
- » Componentes Específicos de Funcionalidad
- » Servicios y Adaptadores
- » Conclusión de la Implementación
- » ¿Qué es el Patrón Contenedor?
- » Estructura del Patrón Contenedor
- » Funcionamiento del Patrón Contenedor
- » Beneficios del Patrón Contenedor
- » Ejemplo con todo lo aprendido
- » Ejemplo de Estructura de Carpeta para una Aplicación de Comercio Electrónico
- » Detalle de Implementación
- » Beneficios de esta Estructura

Chapter 9 ^

Dominando React, la joya sin marco

- » React en vez de un framework completo

- ❖ Armando nuestro primer componente
- ❖ Uso de useState en un Componente
- ❖ Componentes Funcionales: Como una Receta de Cocina
- ❖ Virtual DOM
- ❖ Detección de Cambios: Comprendiendo el Flujo
- ❖ Dominando Custom Hooks
- ❖ Uso Correcto de useEffect: Evitando Errores Comunes
- ❖ Comunicación entre Componentes con children usando el Patrón Composition
- ❖ Comunicación entre Componentes: Composición vs Contexto vs Herencia
- ❖ Uso de Context
- ❖ Comprendiendo useRef, useMemo y useCallback
- ❖ Peticiones a una API y Manejo de Lógica Asíncrona
- ❖ Concepto de Portals
- ❖ Cómo Agregar Estilos a Componentes
- ❖ Routing con react-router-dom
- ❖ Control de Errores con Error Boundaries
- ❖ Mejorando tus Habilidades con Axios Interceptor

Chapter 10 ^

TypeScript Con De Tuti

- ❖ Introducción
- ❖ ¿Por Qué TypeScript?
- ❖ Conceptos Fundamentales de TypeScript

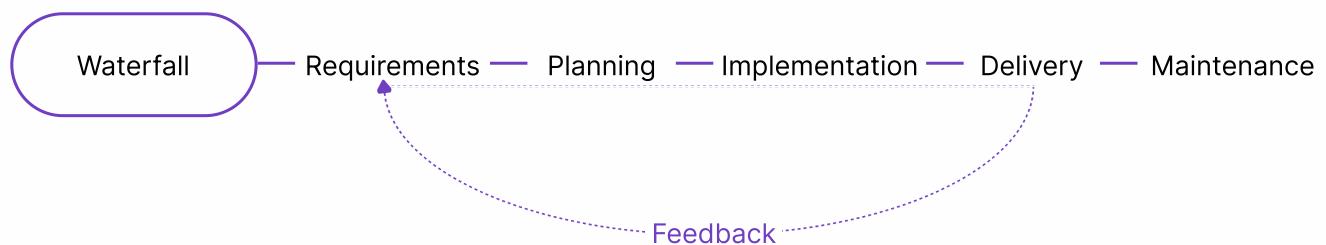
- » TypeScript en la Práctica
- » Mejores Prácticas y Patrones
- » ¿Qué es Transpilar?
- » Entendiendo el Tipado en JavaScript y TypeScript
- » TypeScript: El Superhéroe del Desarrollo
- » Ejemplo Práctico en TypeScript: El Uso de `any` y la Importancia del Tipado
- » Tipos Primitivos en TypeScript
- » Inferencia de Tipos en TypeScript
- » Clases, Interfaces, Enums y Const: ¿Cómo se Utilizan para Tipar en TypeScript?
- » Type vs Interface en TypeScript: Cuándo y Cómo Usarlos
- » El Concepto de Shape en TypeScript
- » Entendiendo union e intersección en TypeScript
- » Entendiendo `typeof` en TypeScript
- » Explorando `as const` en TypeScript
- » Aventura en TypeScript: Type Assertion y Casteo de Tipos
- » Functional Overloading en TypeScript: ¡Pura Magia!
- » Utilitarios de TypeScript: Helpers Esenciales
- » Generics en TypeScript
- » La Magia de los Enums

Código Limpio y Agilidad

¡Fantástico! Todo es ágil hoy en día, todas las empresas aman la agilidad, todo el mundo hace ágil, pero... ¿realmente lo hacen?

❖ Problemas del enfoque en cascada:

Vamos a hablar de la cascada, sí, el chico malo de la ciudad, el que todo el mundo odia. La idea principal de la metodología en cascada es:



- Obtenemos los requisitos, lo que queremos hacer, las necesidades de los interesados.
- Se planifica cómo hacerlo, a menudo con un análisis y diseño de la solución.
- Se implementa, creando software funcional.
- Se entrega el software y se espera el feedback, creando documentación durante el proceso.
- Mantenimiento de la solución funcional.

Una vez que entregamos la solución, pedimos feedback y si necesitamos cambiar algo, comenzamos todo el proceso nuevamente.

Esto es genial siempre y cuando los requisitos sean super estables y sepamos que no cambiarán durante la implementación, algo que en el mundo real es prácticamente imposible ya que SIEMPRE cambian.

Si fuéramos una tela, no tendríamos ninguno de estos problemas, ya que sabemos los materiales específicos necesarios para crear algo, los colocamos en la máquina, y el resultado siempre será el mismo.

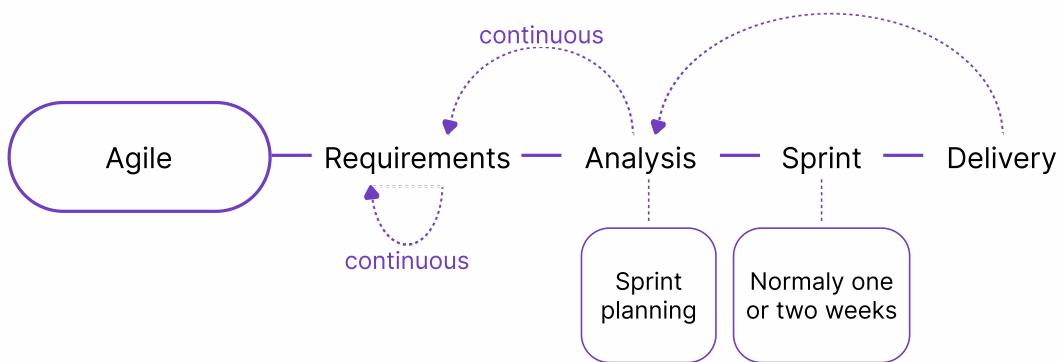
Pero trabajamos con soluciones de software para satisfacer las necesidades de las personas, y estas siempre cambian y evolucionan. Aquí viene el mayor problema con la metodología en cascada. Tenemos que esperar hasta el final de la implementación para recibir comentarios y luego comenzar todo el proceso nuevamente, ¿y qué pasa si las necesidades del usuario han cambiado en el ínterin? Acabamos de desperdiciar mucho tiempo útil.

Una gran analogía es la del piloto de un avión, le gustaría ser informado lo antes posible si hay algún problema con el avión y no esperar hasta que el motor falle, o peor aún, el avión se estrelle para recibir una notificación.

❖ ¿Por qué Agile?

Bueno, aquí está el punto: un proyecto es una sucesión siempre en evolución de eventos, ¡por ejemplo, el análisis nunca termina! Por lo tanto, recibir comentarios lo antes posible es la clave principal de la metodología Agile. Aquí, buscamos la participación de los interesados en todo el proceso, entregando la mínima cantidad de funcionalidades esperando una respuesta, esperando que sea positiva; y si es negativa, no hay problema, podemos abordarla lo antes posible sin tener que esperar al fin del mundo para hacerlo.

Entonces, la forma en que las empresas suelen usar Agile es la siguiente:



- Continuamente obtenemos requisitos, y al trabajar en pequeñas funcionalidades, podemos elegir cuáles son las necesidades más críticas e implementar un plan de acción para entregar pequeñas partes que puedan satisfacer este objetivo.
- Continuamos haciendo un análisis de los requisitos para preparar el trabajo futuro. La cantidad corresponderá al marco de tiempo que ya decidimos según las necesidades.
- Ahora, con todo preparado, estamos cómodos para comenzar a trabajar en nuestras tareas dentro de un sprint. Representa el marco de tiempo que decidimos en el que nos comprometemos a entregar una cierta cantidad de trabajo y puede ser variable según la necesidad.
- Entregamos las funcionalidades y continuamos con el proceso nuevamente. La principal diferencia es que comenzamos el trabajo con comentarios de los interesados de la iteración anterior.

Podemos fallar en la entrega dentro de un marco de tiempo ajustado, y eso no es un problema al principio, ya que medimos el equipo y recopilamos comentarios para ajustarnos al siguiente sprint. Después de algunas iteraciones, podemos estimar la cantidad correcta de trabajo que el equipo puede entregar en un cierto contexto.

❖ Por qué piensas que estás haciendo agile pero en realidad ... no lo estás haciendo

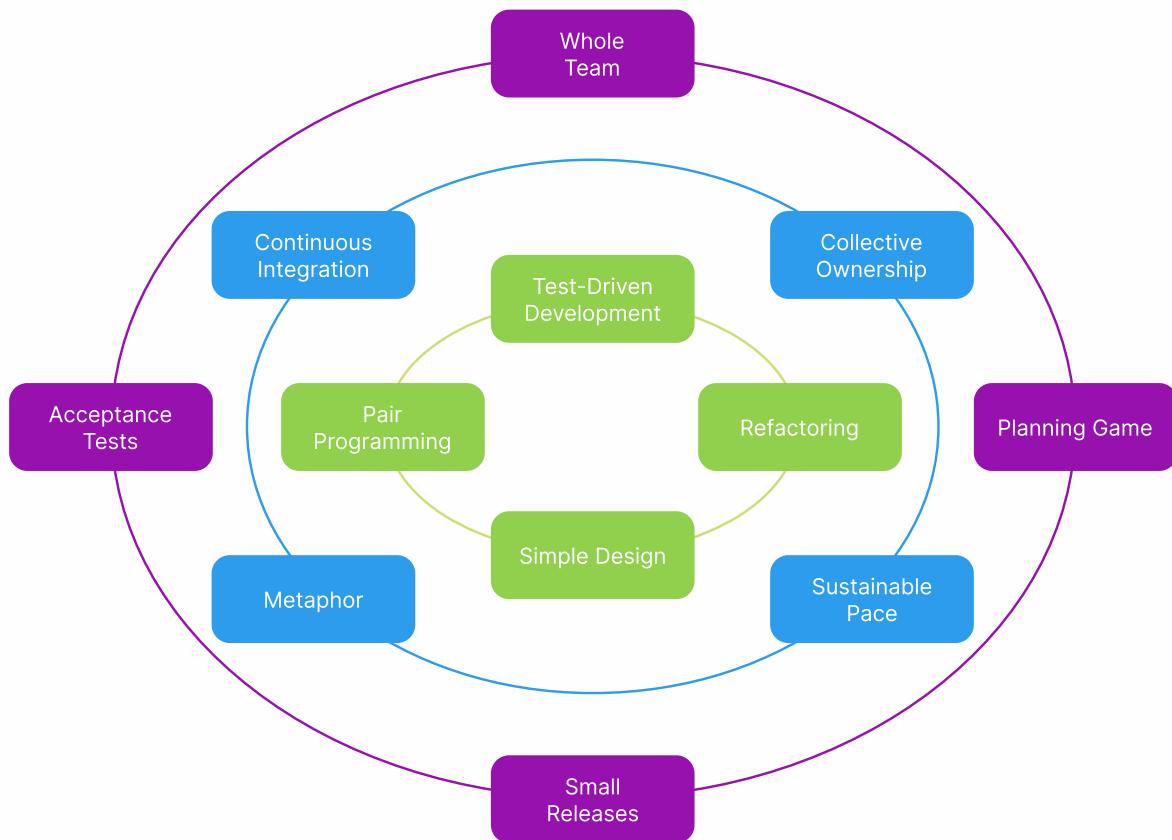
Esto es normal, piensas que estás haciendo agile porque tienes reuniones diarias y eso hace que el equipo sea ágil, pero en realidad esa es solo una ceremonia de muchas. No se puede definir si se es ágil o no por las ceremonias que tienen lugar dentro del proyecto, ya que es más bien una forma de pensar.

Puede que estés trabajando en sprints, utilizando scrum, haciendo retrospectivas y todas esas cosas increíbles, pero también puede que estés trabajando en características enormes, no entregando en cada sprint, no aceptando ningún cambio hasta que termines tu trabajo o incluso siendo dueño de tu conocimiento y no compartiéndolo con el equipo. Si te encuentras en cualquiera de estos últimos puntos ... eh ... no estás haciendo agile.

❖ Programación Extrema

Esta es una práctica impresionante que, según R.C. Martin, cofundador del manifiesto ágil y autor de Clean Agile, es la verdadera esencia del mismo.

Consiste en la organización de prácticas en tres anillos llamados el "Círculo de la Vida". Cada anillo representa un aspecto diferente del desarrollo de proyectos:



El "Anillo Exterior" representa el aspecto empresarial y contiene todas las prácticas enfocadas en los negocios que, juntas, crean el ambiente perfecto para el desarrollo de proyectos.

- **Juego de Planificación:** agarrar el proyecto y dividirlo en piezas más pequeñas para una mejor comprensión y organización. Características, historias, tareas, etc.
- **Pequeñas Liberaciones:** aquí es donde viene lo que estaba diciendo acerca de atacar una funcionalidad completa de una sola vez y cómo eso podría llevar a un enfoque en cascada a algo que debería ser ágil. Siempre debemos tratar de identificar y priorizar las piezas de valor más pequeñas y trabajar en torno a ellas, siendo el trabajo que queremos entregar lo antes posible. Cuanto más pequeña sea la pieza, más rápido recibiremos comentarios y actuaremos en consecuencia.
- **Pruebas de Aceptación:** ahora esto es fácil de entender pero difícil de implementar, necesitamos trabajar en lo que consideramos como equipo como "hecho", ¿cuáles son los requisitos para decir realmente que algo se ha completado por completo, o al menos en los límites acordados? Una recomendación es pensar de nuevo en el valor mínimo que queremos y podemos atacar con el equipo proporcionado. Si tomamos algo realmente grande, será difícil de implementar ya que debemos considerar demasiadas cosas, lo que resulta en requisitos omitidos, definiciones vagas y mala comunicación. Considere crear funcionalidades que tengan un inicio y un final claros, el resultado debe ser algo que proporcione valor por sí solo.
- **Todo el Equipo:** dentro de un equipo de proyecto adecuado, cada miembro proporciona una cierta funcionalidad, tenemos nuestros front-ends, back-ends, diseñadores, dueños de productos, gerentes de proyectos, etc. El problema principal siempre es el mismo, ¿cómo comunicar el trabajo cuando es tan diferente y, al mismo tiempo, depende de cada uno (volveremos a este punto en un momento)?

El "Anillo Intermedio" representa el aspecto del equipo y contiene todas las prácticas enfocadas en el equipo para mejorar la colaboración e interacción del equipo:

- **Ritmo Sostenible:** si preguntas, ¿cuándo quieres que se haga esto? ¡el resultado siempre será ... bueno, lo antes posible! Y, por supuesto, eso es realmente difícil de hacer, no porque el equipo no pueda hacerlo, la mayoría de ellos puede, el problema es hacerlo cada vez manteniendo el mismo ritmo, es imposible. Su equipo se quemará en la tercera o cuarta iteración y luego no se hará ningún trabajo, la velocidad de entrega se reducirá en gran medida. Nuevamente, piense en funcionalidades pequeñas y contenidas que puedan entregarse a una velocidad cómoda.
- **Propiedad Colectiva:** ¿Cuántas veces tienes que preguntarle a tus compañeros de equipo de qué está hablando el dueño del producto en una reunión porque no tienes la cantidad correcta de contexto? No estoy hablando de esas veces en las que estás jugando durante la reunión diaria, sino del torbellino que succiona toda la información que debería compartirse con el equipo y parece que nadie sabe lo que está sucediendo porque nunca se les informó. Este es un problema muy conocido en las empresas, la información se comparte en privado, por lo que solo las personas que estaban en la conversación saben lo que está sucediendo, y más tarde intentan comunicar lo mejor posible el resultado con el resto del equipo, pero en el proceso crean un juego telefónico roto. El proyecto necesita tener una estrategia de comunicación para enfrentar este tipo de situación.
- **Integración Continua:** Como programadores, deberíamos estar comprometidos a hacer cambios en el código lo más rápido posible, sin dejar pasar ni un solo día sin nuevas modificaciones en el repositorio. Siempre existe la posibilidad de trabajar juntos en una funcionalidad y no comunicar los esfuerzos entre sí. Por ejemplo, alguien puede crear un método que hace exactamente lo mismo que otro que ya fue creado por un compañero, pero no se informó al respecto. Hacer cambios lo más rápido posible entregará comentarios a tus compañeros de equipo y los mantendrá actualizados para que siempre

estén trabajando en los "últimos cambios". Si esperamos para entregar nuevo código después de que se complete la funcionalidad, entraremos nuevamente en el territorio de la metodología de cascada.

- **Metáfora:** Si vamos a trabajar juntos en un proyecto, todos debemos entender el contexto de la misma manera, teniendo definiciones claras de cada elemento. Tener el mismo nombre exacto para describir un elemento en particular brindará un mayor nivel de comprensión del equipo y evitará la confusión que podría generarse al referirse al mismo elemento de más de una manera. Podría pensar en esto como si cada proyecto fuera un país diferente, algunos de ellos se comunican en los mismos idiomas exactos, pero usan diferentes metáforas. Por ejemplo, Estados Unidos y Londres usan ambos el inglés como su idioma principal, pero para representar estar molesto, el inglés americano usa "disappointed" y el inglés británico usa "gutted".

El "Anillo Interior" representa el aspecto técnico, que contiene todas las prácticas relacionadas con la mejora del trabajo técnico.

- **Programación en Pareja:** Sentarse juntos para resolver un problema no solo hará que se llegue a una solución más rápido, sino que al mismo tiempo estás compartiendo tu punto de vista con tus compañeros y también ganando el suyo, con el beneficio adicional de llegar a un terreno medio y crear un conjunto de convenciones que el equipo seguirá después. La comunicación es clave cuando se trabaja en equipo, y tener la posibilidad de trabajar juntos para resolver un problema traerá retroalimentación y contexto en torno a la implementación.
- **Simple Design:** Una vez más, trabajemos en pequeñas cosas. Ya hemos hablado de esto antes, pero aquí hay un pequeño consejo: si queremos agregar una cierta funcionalidad que representa un gran desafío para el equipo, siempre debemos buscar una forma de proporcionar la misma cantidad de valor al dar una alternativa mucho más fácil. A veces nos desafiamos a nosotros mismos y entregamos una proposición de valor realmente compleja pero hermosa, pero el problema es que tal vez esa proposición no llegue a ninguna parte porque los requisitos cambian y podemos descubrir que en realidad el usuario no la quiere, por eso trabajar de la manera más simple posible es la mejor opción. Siempre se puede proporcionar una solución simple pero elegante para encontrar el valor adecuado y luego iterar algo mejor.
- **Refactorización:** Todos amamos la frase "si funciona, no lo toques", pero ese no es el enfoque correcto ya que entraremos en una espiral de código heredado al reutilizar código que ya no se puede mantener. Necesitamos refactorizar tanto como sea posible. La deuda técnica es prácticamente inevitable, siempre generamos algún código de mala calidad debido a las restricciones de tiempo de los plazos al implementar la solución rápida pero no tan correcta. Una buena manera de lidiar con esto es utilizar una parte del inicio o el final del sprint, según la prioridad, para refactorizar el código, y esto también se puede hacer utilizando la programación en parejas para aprovechar los beneficios descritos anteriormente.
- **Desarrollo guiado por pruebas:** Hablaremos más sobre esto más adelante, pero podemos definirlo como un proceso en el que escribimos nuestras pruebas antes de codificar una sola línea. La idea principal es que los requisitos de la tarea definen las pruebas que queremos realizar y, como resultado, guíen lo que codificamos. Puede ser un gran aliado al refactorizar, como veremos más adelante.

~~ TDD

TODO el mundo odia hacer pruebas, por ejemplo, los clientes odian PAGAR a las empresas por el "desperdicio" de tiempo de sus desarrolladores Front-End haciendo pruebas, y al final... dinero. Entonces, ¿por qué nosotros, los "desperdiciadores" de tiempo y dinero, deberíamos querer implementar pruebas, verdad?

Bueno, hay algunas cosas en mi mente que pueden compensar todo ese odio:

- Calidad del código
- Mantenimiento del código
- Velocidad de programación (sí, estás leyendo este punto correctamente)

Comprensible, ¿verdad? Escribimos pruebas para que pasen los casos de uso, para escribirlas debemos estar organizados porque si no... será imposible probar nuestro código. Pero hay cosas que considerar, ¿cómo escribimos pruebas de manera que realmente aumente la calidad de nuestro código de alguna manera significativa?

Primero, veamos qué significa calidad de código, y luego les diré lo que significa para MÍ la calidad del código.

Si buscas una respuesta, esta es la que encontrarás:

"Un código de calidad es uno que es claro, simple, bien probado, libre de errores, refactorizado, documentado y con buen rendimiento".

Ahora bien, la medida de la calidad depende de los requisitos de la empresa y los puntos clave suelen ser la confiabilidad, mantenibilidad, testabilidad, portabilidad y reutilización. Es realmente difícil crear código con un 100% de calidad, incluso Walter White no pudo crear metanfetamina con más del 99,1% de pureza; surgirán problemas de desarrollo, plazos y otras situaciones de contexto y tiempo que pondrán en peligro la calidad de tu código.

No puedes escribir código legible, mantenible, probable, portable y reutilizable si te apresuran a terminar una tarea de 4 puntos de historia en solo una mañana (realmente espero que no sea tu caso, y si lo es... ¡tú puedes!)

Entonces, aquí va lo que significa calidad de código para mí. Hacerlo es una mezcla de hacer lo mejor con las herramientas actuales, buenas prácticas y experiencia, contra los límites de contexto existentes para crear el código más limpio posible. Mi recomendación a todos mis estudiantes es que primero alcancen el objetivo y luego, si tienen tiempo, lo usen para mejorar la calidad tanto como sea posible. Es mejor entregar algo feo que una funcionalidad incompleta, pero hermosa. La calidad de tu código aumentará con tu experiencia en el camino, a medida que ganes más, sabrás los mejores pasos para alcanzar un objetivo en la menor cantidad de tiempo y con las mejores prácticas.

El código de calidad también se relaciona con el nivel de comunicación que puedes proporcionar a tus compañeros de equipo o cualquier persona con una simple mirada. Es fácil ver un código y decir "¡wow, esto es genial!" y también decir "¡wow, qué desorden!". Así que cuando codificas, debes pensar que no eres el único trabajando en ello, incluso si trabajas solo como un ejército de desarrollador único, eso ayudará mucho.

Permíteme darte algunas herramientas para escribir un mejor código, primero abramos un poco tu mente.

~~ Diseño atómico, punto de vista de Front End

Separa tu código en la cantidad mínima de lógica posible, cuanto más pequeño sea el código, más fácil será de probar. Esto también trae más beneficios, como la reutilización del código, un mejor mantenimiento e incluso un mejor rendimiento; a medida que el código se vuelve más pequeño y mejor organizado y dependiendo del lenguaje / marco que usemos, podríamos terminar con menos ciclos de procesamiento.

El mantenimiento mejorará enormemente, ya que estamos codificando pequeñas piezas de trabajo, cada una con el acoplamiento más suelto y la cohesión más alta posible, podemos rastrear y modificar el código con el mínimo número de problemas.

Déjame mostrarte cómo pensar atómicamente y cómo puedes llegar a una aplicación completa a partir de una pequeña entrada.

Primero tenemos nuestra entrada:

Cosas simples, eso es lo que llamamos un Átomo, la mínima pieza de lógica posible. Si lo codificas de forma atómica, puedes reutilizar esta entrada en cualquier lugar de tu aplicación y, más adelante, si necesitas modificar su comportamiento o apariencia, simplemente modificas un pequeño átomo con el resultado de tener un impacto en toda la aplicación.

Ahora, supongamos que agregas una etiqueta a esa entrada:

First name:

¡Felicidades! Ahora tienes lo que se llama una Molécula, la mezcla entre átomos, en este caso una etiqueta y una entrada. Podemos seguir avanzando y reduciendo la granularidad.

Podemos usar la entrada con la etiqueta dentro de un Formulario para crear un Organismo, la mezcla entre moléculas:

First name:

Last name:

Si mezclamos Organismos, obtendremos una Plantilla:

My amazing app

Log in

First name:

Last name:

Click the "Submit" button to see something amazing.

Y una colección de plantillas crea nuestra Página, y luego, usando la misma lógica, nuestra Aplicación.

Usar esta forma de pensar hará que tu código sea realmente mantenible, fácil de navegar para rastrear errores y, más que nada... ¡fácil de probar!

Si escribes algo que no sea un Átomo, sería muy difícil probar cualquier cosa, ya que la lógica tendría un alto acoplamiento y, por lo tanto, sería imposible separarla lo suficiente como para verificar casos específicos.

Un ejemplo sería probar un código altamente acoplado, para validar solo una cosa simple, uno tendría que comenzar a incluir una pieza de código... y luego otra... y otra, y después de que termine, verá que incluyó casi todo el código porque había demasiadas dependencias de un lugar a otro.

Y esa es la clave para incluir un jugador más valioso en todo esto.

~~ Programación Funcional

La programación funcional es un paradigma que especifica formas de programar de manera que dividimos nuestra lógica en métodos declarativos, sin efectos secundarios. De nuevo...piensa de manera atómica.

Cuando comenzamos a aprender a programar, normalmente lo hacemos de manera imperativa, donde la prioridad es el objetivo y no la forma en que lo alcanzamos. Aunque es más rápido que la programación funcional, que lo es, puede traer muchos dolores de cabeza aparte de dejar de lado todos los beneficios de la otra forma.

Escribamos una comparación en JavaScript.

Forma imperativa de buscar un elemento dentro de un array:

```
var exampleArray = [
  { name: 'Alan', age: 26 },
  { name: 'Axel', age: 23 },
```

```
];

function searchObject(name) {
  var foundObject = null;

  var index = 0;

  while (!foundObject && exampleArray.length > index) {
    if (exampleArray[index].name === name) {
      foundObject = exampleArray[index];
    }

    index += 1;
  }

  return foundObject;
}

console.log(searchObject('Alan'));

// { name: 'Alan', age: 26 }
```

Y ahora la manera funcional de alcanzar el mismo objetivo:

```
const result = exampleArrayMap.find(element => element.name === name);

console.log(result);
```

No solo es más corto, sino que también es escalable. El método map que estamos aplicando al array es declarativo de ECMAScript, lo que significa que cada vez que ejecutamos el método con los mismos parámetros, siempre obtendremos el mismo resultado. Tampoco modificaremos nada fuera del método, eso es lo que se llama efectos secundarios, el método devuelve un nuevo array con los elementos que cumplen la condición.

Por lo tanto, si creamos métodos que representen las unidades mínimas de lógica posibles, podemos reutilizar el código funcional y probado en toda la aplicación y mantenerlo si es necesario. Una vez más... piensa de manera atómica.

Ahora que sabemos la forma de pensar para crear un código de alta calidad y fácil mantenimiento, pasemos a lo que es una Historia de Usuario.

❖ Historia de Usuario y TDD

¡Vaya título! Todos conocen las historias de usuario, cómo definirlas, qué hacer con ellas, pero nadie sigue la misma forma de escribirlas o incluso su estructura.

Una historia de usuario es la explicación de una funcionalidad desde el punto de vista del usuario. Normalmente se ve así:

¿Qué? Como usuario (quién), quiero tener la posibilidad de... (qué) para... (por qué)

¿Cómo? (casos de uso) 1- paso 1 2- paso 2 3- paso 3 ...

Como puedes ver, definimos quién...el usuario, lo que quiere hacer...la funcionalidad, por qué queremos esta funcionalidad...lo que mientras lo escribimos incluso podemos descubrir que no tiene sentido crearla porque el objetivo no está claro, y cómo la crearemos...los casos de uso.

Los casos de uso representan el número de requisitos que necesitamos cumplir para decir claramente que una historia de usuario está terminada, normalmente cuentan la historia del camino feliz a seguir. También hay lugares donde se describen las entidades relacionadas con la historia de usuario y los casos extremos (camino triste) y creo que es una práctica realmente buena, pero al igual que al escribir código de alta calidad...necesitamos identificar los límites de nuestro contexto para ver cómo podemos escribir el contenido lo más específico posible sin transformar nuestra tarea en un documento difícil de seguir y consumir.

Ahora, TDD, Desarrollo Guiado por Pruebas, es un proceso donde definimos nuestras pruebas antes incluso de escribir una sola línea de código, así que...¿cómo probamos algo que ni siquiera está creado? Bueno, esa es la magia de esto, puedes tomar tus casos de uso y definir lo que necesitas para cumplir cada uno de ellos, crear pruebas alrededor de ellos, hacer que fallen y luego arreglarlos para que pasen...así de simple.

La idea principal detrás del TDD es pensar en:

- ¿Qué quieres hacer?
- ¿Cuáles son los requisitos principales?
- Escribir pruebas que fallarán a su alrededor.
- Crear tu código sabiendo lo que quieras hacer.
- Hacer que la prueba pase.

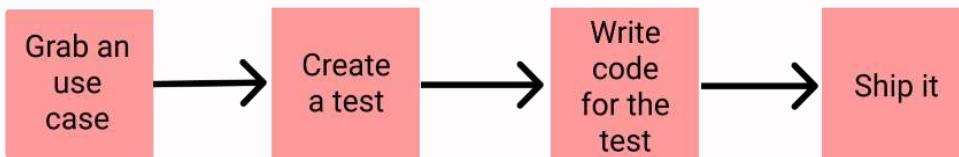
Si piensas que las pruebas son consumidoras de tiempo, bueno, lo son, pero porque tal vez lo hayas hecho previamente de la manera clásica y mala de codificar todo primero y luego intentar probar tu código. ¿Recuerdas de lo que hablábamos sobre la calidad del código, las buenas prácticas, etc.? Bueno, esos son los elementos principales que te ayudarán a probar tu código y si no los estás implementando correctamente, terminaremos con una funcionalidad imposible de separar y probar.

Es por eso que codificar sabiendo lo que quieras probar, utilizando la programación funcional y una forma de pensar atómica puede ser tan beneficioso, porque estarás creando lógica, identificando los requisitos y al final...aumentando la velocidad de codificación.

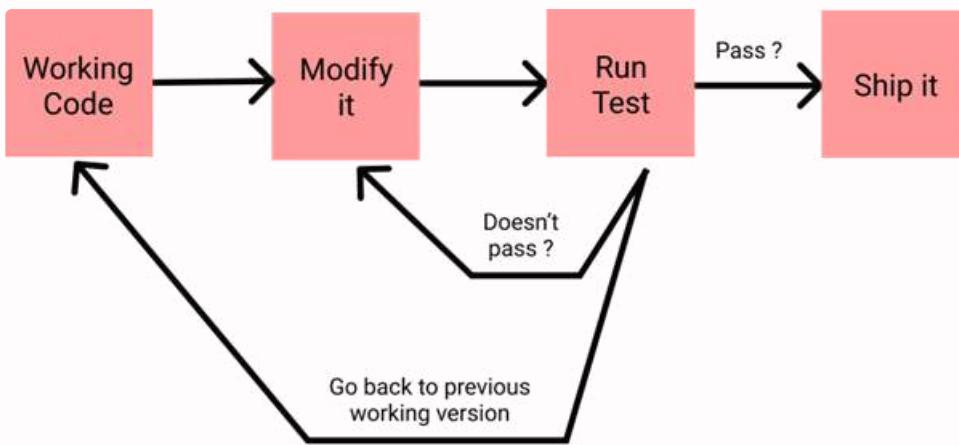
Así que aquí está, la prueba también ayuda a aumentar la velocidad de codificación, a medida que escribes código más manejable, es más fácil modificar un requisito (caso de uso) de tu funcionalidad, ya que lo has

identificado mediante una prueba que te dirá si tu refactorización fue correcta. También reduce la posibilidad de errores, por lo que se dedica menos tiempo a solucionar problemas más adelante.

Flujo del TDD:



Aquí hay un flujo TDD sobre cómo mejorar la calidad del código sin romper nada:



Comunicación ante todo

¡Estamos entrando en una nueva era! El trabajo remoto se está imponiendo y, a medida que aumentan las comodidades, también aumentan los problemas de comunicación entre equipos distribuidos.

❖ Las empresas no están limitadas a una ubicación determinada

Las empresas ya no están limitadas a la ubicación de sus oficinas, ya que ahora juegan con las reglas de todo el mundo, por lo que debemos cambiar la mentalidad para entender este nuevo paradigma. Un gran ejemplo es ofrecer un trabajo a un candidato, si consideramos un salario limitado a la ubicación del candidato, corremos el riesgo de que lo rechace porque puede haber recibido ofertas de todo el mundo, más tentadoras y mejor remuneradas.

Este concepto puede inducir problemas en cualquier nivel organizacional, las personas se compararán y lo que hacen con profesionales de todo el mundo y pueden pensar que no se les ofrece el mismo nivel de beneficios o que simplemente no se les paga lo suficiente. Entonces, ¿cómo se maneja este problema? haciéndoles sentir parte de algo increíble, ayudándoles a superar los obstáculos para el crecimiento personal y lo más importante de todo, manteniéndolos aprendiendo cosas nuevas.

No todo lo que brilla es oro, dice el refrán, y podemos aplicar el mismo concepto entre una empresa y sus empleados. ¡La gente no siempre busca dinero! el conocimiento es uno de los valores más grandes que se pueden proporcionar, como siempre predico lo siguiente: "El conocimiento primero, el dinero segundo; cuanto más sepas, más alguien estará dispuesto a pagarte por ello".

❖ Coordinación entre zonas horarias

Ser una empresa distribuida en primer lugar no es una tarea fácil, gestionar la sinergia laboral entre personas que ni siquiera están en la misma zona horaria puede resultar un desafío mayor de lo previsto.

Recomiendo aprender a trabajar de forma asincrónica, es solo poner las piezas del rompecabezas juntas, pero el desafío es encontrar cuáles son esas piezas. Junto con mi experiencia profesional, detecté que la pieza más importante y también la más difícil es generar una cantidad equilibrada de contexto entre el equipo.

La forma en que lo he estado haciendo a lo largo de los años es entendiendo que la comunicación es clave y secundaria a ninguna, su equipo necesita estar en la misma página o no funcionará.

En segundo lugar, debe conocer sus fuentes de verdad, un lugar donde pueda verificar para aclarar dudas y buscar contexto porque siempre está actualizado. Desde mi experiencia, descubrí que hay dos tipos diferentes:

- Uno representa la realidad de cada lógica empresarial, un gran ejemplo es el uso de Notion o Confluence como fuentes de verdad, donde detallamos lo que esperamos, por qué estamos haciendo lo que estamos haciendo, requisitos, casos especiales, etc.

- También existe la necesidad de un segundo tipo relacionado con las cargas de trabajo y lo que el equipo acordó hacer, y es uno que ya conoces, los tickets creados que el equipo completará con el tiempo.

Necesitamos dos tipos diferentes porque mientras que el primero nos proporciona todo el contexto que podríamos necesitar, también tenemos la necesidad de detenernos en algún momento y decidir cuándo comenzar el proceso de implementación. Este último detalle explica lo que el equipo ha acordado hacer con la cantidad correcta de contexto para que puedan proporcionar suficiente valor y no ser bloqueados en el proceso, y lo necesitamos para poder seguir trabajando en la mejora y evolución.

Pensemos en lo que sucede cuando se tienen dudas sobre una tarea determinada, muchas personas crearían un comentario dentro del ticket, dejarían un mensaje invitando a la comunicación a través de un canal de chat, etc. Esto generalmente resulta en una gran cantidad de conversaciones significativas, confusión y más dudas, ya que no podemos entender completamente la intención del texto. ¿Cómo solucionarlo? Solo cambie el orden de los elementos de una manera diferente:

- Solicite una llamada, hable entre sí y adapte de manera que las necesidades se cumplan en ambos lados.
- Documente los resultados en el ticket relacionado y deje un historial que se pueda rastrear para situaciones similares en el futuro.
- Está bien usar canales de chat para temas simples, pero pase a una reunión tan pronto como vea que la conversación no lleva a ninguna parte.
- Las reuniones no están destinadas a agregar contexto o información nueva, actualice la fuente de verdad y compártala como primera opción.
- Incluya el enlace de la fuente de verdad relacionado dentro de los tickets, pero también intente agregar toda la información necesaria de ella directamente en el ticket para dejar una definición de lo que se acordó en el momento de la creación.

❖ Hay algunas otras formas sencillas en las que podemos mejorar esta mentalidad

Podemos empezar por comunicar las decisiones a través de todas las geografías, esto resultará en que las personas comprendan lo que está sucediendo y el por qué.

Minimizar la fricción en la configuración del ambiente de trabajo, tener documentación y algún tipo de guía mejorará los procesos de nuevas incorporaciones al equipo para conocer la forma en que se espera que trabajen y llegar al nivel requerido para comenzar a trabajar.

Definir claramente la definición de "listo", esto se relaciona con la necesidad de tener una fuente de verdad para la carga de trabajo, cuáles son los criterios de aceptación que necesitamos completar para mover una tarea como "lista". Además, recuerda lo que dije antes, una funcionalidad necesita tener un inicio y un final propio de tal manera que pueda proporcionar valor mientras es independiente.

Usar algunos de los conceptos ya proporcionados en los capítulos anteriores, por ejemplo, hacer programación en pareja o revisiones de código ayuda a distribuir el conocimiento entre las oficinas, ayuda a generar una estructura entre los equipos globales y minimiza la cantidad de colaboración necesaria.

❖ Creando conexión

Alcanzar un nivel aceptado de afinidad entre los equipos distribuidos puede resultar desafiante, pero hay algunas cosas que podemos hacer para aumentar nuestras posibilidades:

- Comunicar incluso los detalles más pequeños hasta que ambas oficinas encuentren un ritmo saludable.
- Comunicar las decisiones.
- Todos necesitan comprender la decisión y por qué se tomó.
- No utilices los correos electrónicos ya que son una forma fácil de perder información.
- Usa un sistema de gestión de contenido, por ejemplo, una wiki.
- Crear canales para que los individuos y los equipos se comuniquen y vean las actualizaciones, otra gran idea sería crear canales para un futuro específico y las personas involucradas.
- Dedica tiempo a crear una guía simple pero efectiva de "Introducción".

Arquitectura Hexagonal

La arquitectura hexagonal es ampliamente utilizada y con razón. Esta arquitectura defiende la "separación de preocupaciones", lo que significa que la lógica de negocios se divide en diferentes servicios o hexágonos, los cuales se comunican entre sí a través de adaptadores a los recursos que necesitan para satisfacer las mismas.

~ Hexágono y sus actores

Se le llama hexagonal porque su forma se asemeja a un hexágono con una línea vertical que lo divide por la mitad. La mitad izquierda se refiere a los actores primarios, quienes llevan a cabo la acción inicial que da comienzo al funcionamiento del hexágono. Estos actores no se comunican directamente con el servicio, sino que utilizan un adaptador. La parte derecha representa a los actores secundarios, quienes proveen los recursos necesarios para que el hexágono pueda ejecutar la lógica interna.

Los adaptadores son piezas clave en la arquitectura hexagonal, ya que se encargan de mediar la comunicación entre dos entidades para que estas puedan interactuar de forma cómoda. Por ejemplo, en un servicio que provee información de los usuarios se utiliza el término "username" para identificar el nombre del usuario, mientras que en otro servicio se utiliza el término "userIdentifier" para la misma acción. Aquí es donde el adaptador interviene para realizar una serie de transformaciones y que la información sea utilizada de la forma más conveniente para cada entidad. En definitiva, los adaptadores facilitan la integración de los diferentes componentes del sistema y la interoperabilidad entre ellos.

Cuando un adaptador se comunica con un actor primario se le llama driver, mientras que cuando se comunica con un recurso necesario se le llama driven. Los drivers se encuentran a la derecha de la línea vertical en la imagen hexagonal y es importante destacar que también pueden representar a otros servicios. En este caso, la comunicación entre servicios debe ser a través de los adaptadores correspondientes, los drivers para el servicio que otorga el recurso y los drivens para el servicio que lo solicita. Así, un servicio puede actuar como actor primario de otro servicio.

~ Puertos y recursos

El siguiente concepto importante a comprender es el de los puertos. Estos indican las limitaciones que tienen tanto nuestro servicio como los adaptadores, y representan las diferentes funcionalidades que deben proporcionar a los actores primarios y secundarios para satisfacer las solicitudes y proveer los recursos necesarios.

~ Tipos de lógica en un servicio

Para comprender las diferentes tipos de lógica que se pueden encontrar dentro de un servicio, es útil distinguir entre la lógica de negocio, la de organización y los casos de uso. Un ejemplo que ilustra esta distinción es una aplicación que administra cuentas bancarias de usuarios y que debe permitir el registro de usuarios mayores de 18 años.

- **Lógica de negocio:** esta es la lógica que proviene del producto y no está afectada por cambios externos. En este ejemplo, el requisito de que los usuarios deban ser mayores de 18 años no tiene su origen en una limitación técnica, sino en una necesidad propia de la aplicación que estamos desarrollando. También el hecho de que se deba crear un registro de usuario es una lógica de negocio, ya que es una necesidad específica de la aplicación.
- **Lógica de organización:** es similar a la lógica de negocio, pero se reutiliza en más de un proyecto de una misma organización. Por ejemplo, la metodología utilizada para validar y registrar tarjetas de crédito en nuestra aplicación podría ser una lógica de organización que se utiliza en varios proyectos dentro de la misma empresa.
- **Casos de uso:** son aquellos que tienen una limitación técnica y pueden cambiar si el uso de la aplicación cambia. Por ejemplo, los requisitos para registrar un usuario podrían no ser un caso de uso, ya que no existe ninguna limitación técnica para validar los campos de un formulario. Sin embargo, la disposición del mensaje de error, su color, tamaño, etc. sí pueden afectar el uso de la aplicación, lo que convierte a estos elementos en casos de uso. La posición y forma de mostrar los campos en la pantalla también pueden ser un caso de uso, ya que afecta al SEO de la aplicación si se produce un cambio en el content layout shift.

~~ Ejemplo de aplicación: La pizzería en forma de hexágono

Uno de los ejemplos más conocidos es el de una pizzería donde una persona quiere hacer un pedido. Para ello, se fijará en el menú las diferentes opciones, pedirá al cajero su pedido, éste comunicará a la cocina el encargo, la cocina realizará la serie de procedimientos necesarios para cumplir con el requisito y devolverá el pedido ya completo al cajero para que éste se entregue al comprador. Si pensamos detenidamente en cada una de las entidades del ejemplo, podremos encontrar que el comprador es el actor principal, el menú con las diferentes opciones es el puerto, el cajero es el adaptador y la cocina es nuestro servicio.

El consumidor pedirá un producto viendo el menú y solo podrá pedir lo que vea en el mismo. A su vez, ese mismo pedido que de cara al público puede tener un nombre llamativo, seguramente para la cocina se llame de una manera más simplificada para aumentar la eficiencia del proceso. El cajero conoce esta nomenclatura y es el encargado de poder gestionar una correcta comunicación entre el consumidor y la cocina. Lo que para uno es una pizza margarita, para otro es la número 53.

Una parte que no vemos es que la cocina en sí necesita recursos para poder completar el pedido. Esto quiere decir que tanto el queso, el tomate y el resto de ingredientes deben ser solicitados para poder gestionar las órdenes. Para esto, seguramente hay un encargado que se encuentre entre el restaurante y un proveedor de materia prima. Y nuevamente, lo que para uno es tomate, para otro es el producto ABC. Aquí vemos el claro ejemplo de un actor secundario, el proveedor, que se comunica por medio de un intermediario, el encargado, para proveer los recursos necesarios a nuestro hexágono.

~~ Pasos recomendados para trabajar en arquitectura hexagonal con ejemplo de aplicación

- 1- Piensa atentamente en los requisitos. Un ejemplo que ilustra esta distinción es una aplicación que administra cuentas bancarias de usuarios y que debe permitir el registro de usuarios mayores de 18 años.
- 2- Identifica la lógica de negocios que debes cumplir. En el ejemplo del sistema de reservas de vuelos, la lógica de negocios podría incluir la validación de la disponibilidad de los vuelos y la asignación de asientos a los pasajeros, así como la gestión de pagos y la emisión de boletos.
- 3- Identifica qué acciones debe proveer tu hexágono para poder satisfacer la lógica requerida. En el ejemplo del sistema de reservas de vuelos, las acciones que el hexágono debe proveer podrían incluir: buscar vuelos disponibles, reservar un vuelo, asignar asientos a los pasajeros, gestionar pagos y emitir boletos.
- 4- Identifica los recursos necesarios para poder satisfacer dicha lógica y quién puede proporcionarlos. En el ejemplo del sistema de reservas de vuelos, los recursos necesarios podrían incluir: una base de datos de vuelos y asientos, un proveedor de pagos en línea y un servicio de emisión de boletos. En este paso, es importante identificar quién puede proporcionar estos recursos y cómo se integrarán en el sistema.
- 5- Crea los puertos necesarios para los drivers y drivens. En el ejemplo del sistema de reservas de vuelos, los puertos necesarios podrían incluir: un puerto de búsqueda de vuelos, un puerto de reservas, un puerto de asignación de asientos, un puerto de gestión de pagos y un puerto de emisión de boletos. Es importante que estos puertos estén diseñados para interactuar con los drivers (interfaces de usuario) y los drivens (bases de datos, proveedores de pagos, etc.) de forma clara y coherente.
- 6- Crea adaptadores de tipo Stub/Mocked para satisfacer de forma inmediata las solicitudes y poner a prueba tu hexágono. En este paso, se crean adaptadores para los drivers y drivens que imitan su comportamiento real, pero que se pueden usar para probar el hexágono de forma inmediata sin depender de los sistemas externos. Por ejemplo, se podrían crear adaptadores Stub/Mocked para la base de datos de vuelos y asientos, el proveedor de pagos en línea y el servicio de emisión de boletos.
- 7- Crea las pruebas necesarias que deben pasar de manera satisfactoria para que tu hexágono satisfaga la solicitud. En este paso, se crean pruebas para validar que el hexágono funciona según lo esperado y que satisface los requisitos del sistema. Por ejemplo, se podrían crear pruebas para validar que se pueden buscar vuelos disponibles, reservar un vuelo, asignar asientos a los pasajeros, gestionar pagos y emitir boletos de forma exitosa.
- 8- Crea la lógica dentro de tu hexágono para poder satisfacer los casos de uso. En este paso, se crea la lógica de negocio dentro del hexágono para poder satisfacer los casos de uso identificados en el paso 2. En el ejemplo del sistema de reservas de vuelos, se podría crear la lógica para validar la disponibilidad de los vuelos y asientos, asignar los asientos a los pasajeros y gestionar el proceso de pago y emisión de boletos. Esta lógica debe estar diseñada de tal manera que sea fácilmente modificable y escalable en el futuro si se requieren cambios o mejoras en el sistema. También es importante que esta lógica esté separada de la lógica específica de los drivers y drivens para facilitar el mantenimiento y la evolución del sistema en el futuro.

Para trabajar en arquitectura hexagonal, es fundamental seguir un proceso metódico y cuidadoso. En primer lugar, es crucial leer detenidamente los requerimientos para poder pensar en la mejor forma de resolverlos. Es esencial reconocer la lógica de negocios y los casos de uso necesarios para satisfacerla. Como recordarás de la explicación anterior, esto es fundamental para asegurarnos de que estamos cubriendo todas las necesidades.

Una vez que hayamos identificado los requisitos y los recursos necesarios, es momento de crear nuestros puertos. Para hacerlo, debemos reconocer los métodos esenciales que deben estar disponibles para nuestros actores primarios y secundarios. Esto nos permitirá controlar los accesos y salidas al servicio. Por convención, los nombres de los puertos deben comenzar con la palabra "For", seguida de la acción que deben realizar. Por ejemplo, si necesitamos un puerto para realizar una acción de registro, podríamos llamarlo "ForRegistering". También podemos reducir la cantidad de puertos si asociamos diferentes acciones relacionadas, como "ForAuthenticating", que proveerá las acciones de registro y login.

Siguiendo estos pasos y manteniendo una atención cuidadosa a los detalles, podremos trabajar de forma efectiva en arquitectura hexagonal y lograr resultados óptimos en nuestros proyectos.

A continuación, debemos crear nuestros adaptadores driver y driven. Para ello, recomiendo utilizar como nombre del adaptador el realizador de la acción, seguido de la acción en sí. Por ejemplo, en nuestro caso, podríamos llamarlos Registerer o Authenticator, respectivamente.

Los adaptadores, en primer lugar, deben ser del tipo stub y proporcionar información controlada que pueda utilizarse para satisfacer la lógica de negocios y los tests de la misma. De esta manera, podremos cerrar nuestro hexágono y dejarlo listo para la implementación.

Con nuestros adaptadores completos, procedemos a utilizar TDD (Desarrollo Guiado por Pruebas, por sus siglas en inglés). Crearemos los tests necesarios para cumplir con los casos de uso, de manera que se pueda comprobar el correcto funcionamiento de la lógica al implementarla.

Nuestro servicio debe disponer de las entidades necesarias para el tipado, satisfacer los métodos proporcionados en los puertos primarios y cumplir con los casos de uso. El servicio es responsable de recibir una solicitud, buscar los recursos necesarios mediante los adaptadores secundarios y utilizarlos para cumplir con los casos de uso y, por lo tanto, con la lógica de negocios.

~~ Ejemplo de código siguiendo la temática de aplicación bancaria:

```
// Clase que representa el hexágono
export class BankAccountService {
    private bankAccountPort: BankAccountPort;

    constructor(bankAccountPort: BankAccountPort) {
        this.bankAccountPort = bankAccountPort;
    }

    /**
     * Método para crear una nueva cuenta bancaria.
     * @param name - El nombre del titular de la cuenta.
     * @param age - La edad del titular de la cuenta.
     * @throws AgeNotAllowedException si la edad no es permitida.
     */
    public createBankAccount(name: string, age: number): void | AgeNotAllowedException {
        if (age >= 18) {
            const bankAccount = new BankAccount(name, age);
            this.bankAccountPort.saveBankAccount(bankAccount);
        }
    }
}
```

```

} else {
    throw new AgeNotAllowedException("La edad mínima para crear una cuenta bancaria es de 18 años.");
}
}

// Archivo bank-account-port.ts
// Interfaz que define el puerto para acceder a la base de datos de cuentas bancarias
export interface BankAccountPort {
    saveBankAccount(bankAccount: BankAccount): void;
}

// Archivo bank-account.ts
// Clase que representa la entidad de una cuenta bancaria
export class BankAccount {
    private name: string;
    private age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

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

    public getAge(): number {
        return this.age;
    }
}

// Archivo age-not-allowed-exception.ts
// Excepción personalizada para cuando la edad no está permitida
export class AgeNotAllowedException extends Error {
    constructor(message: string) {
        super(message);
        this.name = "AgeNotAllowedException";
    }
}

// Archivo bank-account-repository.ts
// Clase que implementa el puerto para acceder a la base de datos de cuentas bancarias
export class BankAccountRepository implements BankAccountPort {
    private bankAccounts: BankAccount[] = [];

    public saveBankAccount(bankAccount: BankAccount): void {
        this.bankAccounts.push(bankAccount);
    }
}

// Archivo bank-account-controller.ts

```

```
// Clase que representa el driver para la creación de cuentas bancarias
import { BankAccountService } from "./bank-account-service";
import { AgeNotAllowedException } from "./age-not-allowed-exception";

export class BankAccountController {
    private bankAccountService: BankAccountService;

    constructor(bankAccountService: BankAccountService) {
        this.bankAccountService = bankAccountService;
    }

    /**
     * Método para crear una nueva cuenta bancaria.
     * @param name - El nombre del titular de la cuenta.
     * @param age - La edad del titular de la cuenta.
     */
    public createBankAccount(name: string, age: number): void {
        try {
            this.bankAccountService.createBankAccount(name, age);
            console.log("La cuenta bancaria se ha creado correctamente.");
        } catch (e) {
            if (e instanceof AgeNotAllowedException) {
                console.log(e.message);
            } else {
                console.log("Ha ocurrido un error al crear la cuenta bancaria.");
            }
        }
    }
}

// Archivo main.ts
// Ejemplo de uso
import { BankAccountRepository } from "./bank-account-repository";
import { BankAccountService } from "./bank-account-service";
import { BankAccountController } from "./bank-account-controller";

const bankAccountPort = new BankAccountRepository();
const bankAccountService = new BankAccountService(bankAccountPort);
const bankAccountController = new BankAccountController(bankAccountService);

bankAccountController.createBankAccount("John Doe", 20);
bankAccountController.createBankAccount("Jane Doe", 16);
```

Cómo usar GoLang

Un LENGUAJE ASOMBROSO creado por Google en colaboración con Rob Pike, Ken Thomson y Robert Griesemer.

❖ Ventajas:

- Rápido, compila directamente a código máquina sin necesidad de usar un intérprete.
- Fácil de aprender, muy buena documentación y muchas cosas simplificadas.
- Escala muy bien, soporta la programación concurrente a través de "GoRoutines".
- Recolector de basura automático, gestión automática de memoria.
- Motor de formateo incluido, no es necesario usar terceros.
- No se necesitan bibliotecas para pruebas o benchmarks porque ya están incluidas.
- Muy poco boilerplate para crear aplicaciones.
- Tiene una API para programación de redes, incluida como biblioteca estándar.
- MUY rápido, en algunas pruebas de rendimiento es más rápido que las aplicaciones backend hechas en Java y Rust.
- Sistema de plantillas incorporado, GENIAL para trabajar con HTMX.

❖ Estructura Recomendada:

- **ui** (contenido relacionado con el frontend en caso de renderización en el servidor)
 - *html* (plantillas)
 - *static* (contenido estático multimedia y de estilo)
 - *assets*
 - *css*
- **internal** (contenido relacionado con herramientas y entidades reutilizables en todo el proyecto)
 - *models*
 - *utils*
- **cmd**
 - *web* (contiene la lógica de la aplicación)
 - *domain* (lógica de negocio)

- *routes* (rutas disponibles)

~~ ¿Cómo funciona GoLang?

GoLang utiliza un archivo base llamado go.mod, que contendrá el módulo principal que se llamará igual que el proyecto, y también la versión de Go utilizada. Luego, cada archivo tendrá la extensión ".go" para identificar que es un paquete perteneciente al lenguaje.

Pero... ¿qué es un paquete? Si vienes de JavaScript, puedes pensar en él de la misma manera que un módulo ES, ya que se usa para encapsular lógica relacionada. Pero a diferencia de los módulos ES, el paquete se identifica por las líneas de código "package packageName" en camel case el nombre del paquete en cuestión e importa la ubicación del paquete. Diferentes archivos que contienen lógica perteneciente al mismo paquete pueden organizarse por separado PERO deben estar bajo la misma carpeta principal, ya que esto es de suma importancia para luego importar dicho paquete en otros.

Para importar un paquete diferente se hace a través de la palabra "import" más la ruta a la que pertenece el paquete.

```
import "miProyecto/cmd/web/routes"
```

Si necesitas más de un paquete a la vez, no es necesario repetir la línea de código ya que se pueden agrupar usando "()" los diversos paquetes:

```
import (  
    "miProyecto/cmd/web/routes"  
    "miProyecto/internal/models"  
)
```

Vale la pena mencionar que luego Go relacionará el nombre final de la ruta con el uso del paquete, por lo que para usar la lógica contenida en él se hará pensando en él como si fuera un objeto, donde cada propiedad representa un elemento lógico del paquete:

```
routes.MiRuta
```

Métodos de paquete privados y públicos:

Si el método comienza con minúscula, es un método privado, no se puede acceder desde fuera del paquete en sí mismo.

```
func miFuncion()
```

Si el método comienza con mayúscula, es un método público, se puede acceder importando el paquete desde otro.

```
func MiFuncion()
```

Ámbito del paquete

Veamos un archivo Go

```
package main - nombre del paquete

import "fmt" - paquete fmt importado, sin ruta porque es propio de Go

var numero int = 2

func main() {
    i, j := 42, 2701 - variables locales al método, i con valor 42 y j con valor 2701

    fmt.Println(i) - usando el método "Println" del paquete "fmt"
}
```

Seguro que has notado algo, "numero" tiene un tipo "int" precediendo la asignación de valor, mientras que "i" y "j" no lo tienen, esto se debe a que al igual que Typescript, Go infiere el tipo para esos primitivos. Veamos cómo trabajar con tipos.

❖ Tipos de Datos

- **bool** = true / false
- **string** = cadena de caracteres
- **int, int8, int16, int32, int64, uint, uint8, uint16, uint32, uint64, uintptr** = valores numéricos enteros con sus límites, generalmente son 32 bits en sistemas de 32 bits y 64 para sistemas de 64 bits. Debería usarse entero a menos que haya una razón específica para usar un valor restringido.
- **byte** === uint8
- **rune** === int32
- **float32, float64** = representa valores numéricos reales
- **complex64, complex128** = números complejos que tienen una parte real y una parte imaginaria.

❖ Structs

Representa una colección de propiedades, puedes pensar en ella como una interfaz de Typescript, ya que representa el contrato que debe seguir al crear una propiedad. Importante, si deseas que la propiedad sea

accesible fuera del paquete, recuerda que debe comenzar con "mayúscula".

```
type Persona struct {
    Nombre string
    Apellido string
    Edad int
}

var persona = Persona {
    Nombre: "Caballero",
    Apellido: "Programador",
    Edad: 31
}

fmt.Println(persona.Nombre)
```

Otra forma:

```
var persona2 = Persona{"Caballero", "Programador", 31}
```

❖ Arrays

Ahora comienza la diversión, los arrays son bastante diferentes de lo que estamos acostumbrados ya que DEBEN tener el número máximo de elementos que van a contener dentro:

```
var a [10]int - crea un array de 10 elementos de tipo int
a[0] = "Caballero"
```

O también

```
var a = [2]int{2, 3}
fmt.Println(a) - [2 3]
```

Si necesitamos que sea dinámico podemos hablar de "slices". Un "slice" es una porción de un array existente o una representación de una colección de elementos de un cierto tipo.

```
var primos = [6]int{2, 3, 5, 7, 11, 13}
var s []

int = primos[1:4] - crea un slice usando "primos" como base desde la posición 1 hasta la 4

fmt.Println(s) - [3 5 7]
```

```
s = append(s , 14)

fmt.Println(s) - [3 5 7 14]

fmt.Println(primos) - [2 3 5 7 14 13]
```

También puedes omitir valores para rangos máximo y mínimo haciéndolos tener valores predeterminados:

```
var a [10]int

es lo mismo que

a[0:10]
a[:10]
a[0:]
a[:]
```

❖ Método Make

Para crear slices dinámicos puedes usar el método "make" incluido, esto creará un array lleno de elementos vacíos y devolverá un slice que se refiere a él. El método "len" se puede usar para ver cuántos elementos contiene actualmente y "cap" para ver su capacidad, es decir, cuántos elementos puede contener.

```
a := make([]int, 0, 5) // len(a)=0 cap(a)=5
```

❖ Punteros

Si vienes de Javascript...

esto llevará un poco de tiempo, pero veamos juntos el siguiente ejemplo:

```
type TipoElemento struct {
    nombre string
}

var ejemploElemento = TipoElemento {
    nombre: "Caballero",
}

func MiFuncion(elemento TipoElemento) {
    ...
}
```

Aquí podrías pensar que estamos trabajando en el elemento "ejemploElemento", pero es todo lo contrario.

Entonces, si queremos trabajar con el mismo elemento pasado como parámetro a la función, se debe usar un poco de magia:

```
```go
var a = 1
```

Crea un espacio de memoria que dentro contiene el valor "1" y creamos una referencia a ese espacio de memoria llamada "a". ¡La diferencia con Javascript es que esta referencia no se pasa al método a menos que hayamos creado un puntero a ella!

```
var p *int // puntero "p" que referenciará una propiedad de tipo "int"

i := 42
p = &i // crea un puntero directo a la propiedad "i"

// Si queremos acceder al valor referenciado por el puntero "p", usamos el nombre del puntero precedido por un asterisco
fmt.Println(*p) // 42

*p = 21

fmt.Println(*p) // 21
```

Donde esto cambia es si apuntamos a una "estructura", ya que sería un poco engorroso hacer `(*p).Propiedad`, se reduce a usarlo como si fuera la estructura misma:

```
v := Persona{"Caballero"}
p := &v
p.Nombre = "Programador"
fmt.Println(v) // {Programador}
```

## ❖ Valores Predeterminados

En Go, cuando declaras una variable sin asignar explícitamente un valor, toma un valor predeterminado basado en su tipo. Aquí tienes una tabla que resume los valores predeterminados:

### Valores Predeterminados para Tipos de Datos:

- **bool:** `false`
- **string:** `""` (cadena vacía)
- **Tipos Numéricos:** `0`

- **array**: [0...] (valores cero)
- **map**: nil (no inicializado)
- **slice**: nil (no inicializado)
- **puntero**: nil (no inicializado)
- **función**: nil (no inicializado)

## ~~ Bucle Range

El bucle **range** es una construcción poderosa para iterar sobre secuencias como slices, arrays, maps y strings. Proporciona dos componentes: el índice (**i**) y el valor (**v**) de cada elemento. Aquí tienes tres variaciones comunes:

- **Iteración Completa:**

```
var arr = []int{5, 4, 3, 2, 1}

for i, v := range arr {
 fmt.Printf("índice: %d, valor: %d\n", i, v)
}
```

Este enfoque itera sobre tanto el índice como el valor de cada elemento en **arr**.

- **Ignorando Índice:**

```
for _, v := range arr {
 fmt.Printf("valor: %d\n", v)
}
```

El guion bajo (**\_**) descarta la información del índice, centrándose solo en los valores de los elementos.

- **Ignorando Valor:**

```
for i, _ := range arr {
 fmt.Printf("índice: %d\n", i)
}
```

De manera similar, puedes usar un guion bajo para omitir el valor y acceder solo a los índices.

## ~ Maps

Los maps son colecciones desordenadas que asocian claves únicas (de cualquier tipo hashable) con valores. Go proporciona dos formas de crear y trabajar con maps:

- Usando la Función `make`:

```
type Persona struct {
 DNI, Nombre string
}

var m map[string]Persona

func main() {
 m = make(map[string]Persona)
 m["123"] = Persona{"123", "pepe"}
 fmt.Println(m["123"])
}
```

- Literal de Mapa:

```
type Persona struct {
 DNI, Nombre string
}

var m = map[string]Persona{
 "123": Persona{"123", "pepe"},
 "124": Persona{"124", "jorge"},
}

func main() {
 fmt.Println(m)
}
```

Los literales de mapas ofrecen una forma concisa de inicializar maps con pares clave-valor.

## ~ Mutando Maps

- Inserción:

```
m[clave] = elemento
```

Agrega un nuevo par clave-valor al map m.

- **Recuperación:**

```
elemento = m[clave]
```

## ~~ Funciones

Las funciones son bloques de código reutilizables que realizan tareas específicas.

Se declaran con la palabra clave `func`, seguida del nombre de la función, la lista de parámetros (si los hay), el tipo de retorno (si lo hay) y el cuerpo de la función encerrado entre llaves.

Aquí tienes un ejemplo:

```
func saludar(nombre string) string {
 return "¡Hola, " + nombre + "!"
}

func main() {
 mensaje := saludar("Golang")
 fmt.Println(mensaje)
}
```

## ~~ Valores de Funciones

Las funciones pueden asignarse a variables, lo que te permite pasárselas como cualquier otro valor. Esto permite técnicas poderosas como las funciones de orden superior.

Aquí tienes un ejemplo que muestra cómo pasar una función como argumento y llamarla indirectamente:

```
func LlamarCallback(retroceso func(float64, float64) float64 {
 return retroceso(3, 4)
}

func hipotenusa(x, y float64) float64 {
 return math.Sqrt(x*x + y*y)
}

func main() {
 fmt.Println(hipotenusa(5, 12))
 fmt.Println(LlamarCallback(hipotenusa))
}
```

## ❖ Closure

Los closures son un tipo especial de función que captura variables de su entorno circundante. Esto permite que el cierre acceda y manipule estas variables incluso después de que la función que lo rodea haya devuelto.

Aquí tienes un ejemplo de un cierre que crea una función "sumador" con una suma persistente:

```
func sumador() func(int) int {
 suma := 0

 return func(x int) int {
 suma += x
 return suma
 }
}

func main() {
 pos, neg := sumador(), sumador()

 for i := 0; i < 10; i++ {
 fmt.Println(
 pos(i),
 neg(-2*i),
)
 }
}
```

## ❖ Métodos

Go no tiene clases, pero permite definir métodos en tipos (structs, interfaces). Un método es una función asociada con un tipo, que toma un argumento receptor (generalmente el tipo mismo) que se refiere implícitamente al objeto sobre el que se llama el método.

Aquí tienes un ejemplo de una estructura Persona con un método Saludar:

```
type Persona struct {
 Nombre, Apellido string
}

func (p Persona) Saludar() string {
 return "Hola " + p.Nombre
}

func main() {
```

```
p := Persona{"Pepe", "Perez"}
fmt.Println(p.Saludar())
}
```

Los métodos también se pueden definir en tipos que no son estructuras:

```
type Nombre string

func (n Nombre) Saludar() string {
 return "Hola " + string(n)
}

func main() {
 nombre := Nombre("Pepe")
 fmt.Println(nombre.Saludar())
}
```

Los métodos pueden aceptar punteros como receptores, lo que permite modificaciones al objeto original:

```
type Persona struct {
 nombre, apellido string
}

func (p *Persona) cambiarNombre(n string) {
 p.nombre = n
}

func main() {
 p := Persona{"pepe", "perez"}
 p.cambiarNombre("juan")
 fmt.Println(p) // Salida: {juan perez}

 pp := &Persona{"puntero", "persona"}
 pp.cambiarNombre("punteroNuevoNombre")
 fmt.Println(*pp) // Salida: {punteroNuevoNombre persona}
}
```

Go automáticamente desreferencia receptores de puntero cuando es necesario, por lo que no siempre necesitas usar el operador `*` explícitamente.

## ~~ Interfaces

Las interfaces definen un conjunto de métodos que un tipo debe implementar. Proporcionan una manera de lograr polimorfismo, permitiendo que diferentes tipos se usen de manera intercambiable siempre que implementen los métodos requeridos.

Aquí tienes un ejemplo de una **Interfaz** que define dos métodos, **Saludar** y **Moverse**:

```

type Persona interface {
 Saludar() string
 Moverse() string
}

type Alumno struct {
 Nombre string
}

func (a Alumno) Saludar() string {
 return "Hola " + a.Nombre
}

func (a Alumno) Moverse() string {
 return "Estoy caminando"
}

func main() {
 var persona Persona = Alumno{
 "Pepe",
 }

 fmt.Println(persona.Saludar())
 fmt.Println(persona.Moverse())
}

```

## ~~ Valores de Interfaces con Nil

Los valores de las interfaces pueden ser `nil`, lo que indica que no tienen una referencia a ningún objeto específico. Aquí tienes un ejemplo que muestra cómo manejar valores de interfaces `nil`:

```

type I interface {
 M()
}

type T struct {
 S string
}

func (t *T) M() {
 if t == nil {
 fmt.Println("<nil>")
 return
 }
 fmt.Println(t.S)
}

```

```
func main() {
 var i I

 var t *T
 i = t
 describe(i)
 i.M() // Salida: <nil>

 i = &T{"hola"}
 describe(i)
 i.M() // Salida: hola
}

func describe(i I) {
 fmt.Printf("(%v, %T)\n", i, i)
}
```

## ~~ Interfaces Vacías

Si no conoces los métodos específicos que una interfaz podría requerir de antemano, puedes crear una interfaz vacía usando el tipo `interface{}`. Esto te permite almacenar cualquier valor en la interfaz, pero no podrás llamar métodos directamente sobre él.

```
var i interface{}
```

## ~~ Aserción de tipo

Cuando usamos una interfaz vacía `interface{}`, podemos utilizar cualquier tipo de dato, PERO, esto también viene con problemas. ¿Cómo sabemos si el parámetro de un método es del tipo esperado si es una interfaz vacía? Aquí es donde las Aserciones de Tipo resultan útiles, ya que proporcionan la posibilidad de probar si la interfaz vacía es del tipo esperado.

```
t := i.(T)
```

Esto significa que el valor de la interfaz `i` tiene el tipo concreto `T` y asigna el valor subyacente `T` a la variable `t`.

Si `i` no tiene un tipo `T`, esto provocará un error en tiempo de ejecución (panic).

Puedes probar si la interfaz contiene un tipo específico usando un segundo parámetro, al igual que lo hacemos con `err`:

```
t, ok := i.(T)
```

Esto guardará `true` o `false` dentro de `ok`. Si es `false`, `t` guardará un valor cero y no ocurrirá ningún error en tiempo de ejecución.

```
func main() {
 var i interface{} = "hello"

 s := i.(string)
 fmt.Println(s) // hello

 s, ok := i.(string)
 fmt.Println(s, ok) // hello true

 f, ok := i.(float64)
 fmt.Println(f, ok) // 0 false

 f = i.(float64) // panic: conversión de interfaz: la interfaz {} es string, no float64
 fmt.Println(f) // nada, generará un error antes
}
```

## ~~ Switch de tipo

Proporciona la posibilidad de hacer más de una aserción de tipo en serie.

Al igual que un switch regular, pero usamos tipos en lugar de valores, y los últimos se compararán con el tipo del valor contenido por la interfaz dada.

```
switch v := i.(type) {
 case T:
 // si v tiene tipo T
 case S:
 // si v tiene tipo S
 default:
 // si v no tiene ni tipo T ni tipo S, tendrá el mismo tipo que "i"
}
```

Aclaración: al igual que las Aserciones de Tipo, usamos un tipo como parámetro `i.` (`T`), pero en lugar de usar `T`, necesitamos usar la palabra clave `type`.

Esto es excelente cuando ejecutamos lógicas diferentes que dependen del tipo del parámetro:

```
type Saludador interface {
 Saludar()
}

type Persona struct {
```

```

Nombre string
}

func (p Persona) Saludar() {
 fmt.Printf("¡Hola, mi nombre es %s!\n", p.Nombre)
}

type Numero int

func (n Numero) Saludar() {
 if n%2 == 0 {
 fmt.Printf("¡Hola, soy un número par: %d!\n", n)
 } else {
 fmt.Printf("¡Hola, soy un número impar: %d!\n", n)
 }
}

func main() {
 saludadores := []Saludador{
 Persona{"Alice"},
 Persona{"Bob"},
 Numero(3),
 Numero(4),
 }

 for _, saludador := range saludadores {
 switch valor := saludador.(type) {
 case Persona:
 valor.Saludar()
 case Numero:
 valor.Saludar()
 }
 }
}

```

## ~~ Stringers

Es un tipo que se define a sí mismo como `string`, está definido por el paquete `fmt` y se utiliza para imprimir valores.

```

type Persona struct {
 Nombre string
 Edad int
}

func (p Persona) String() string {
 return fmt.Sprintf("%v (%v años)", p.Nombre, p.Edad)
}

```

```

func main() {
 a := Persona{"Gentleman Programming", 32}
 z := Persona{"Alan Buscaglia", 32}

 fmt.Println(a, z) // Gentleman Programming (32 años) Alan Buscaglia (32 años)
}

```

Ejemplo usando Stringers para modificar la forma en que mostramos una dirección IP al usar `fmt.Println`:

```

type IPAddr [4]byte

func (ip IPAddr) String() string {
 str := ""

 for i, ipValor := range ip {
 str += fmt.Sprint(ipValor)

 if i < len(ip)-1 {
 str += "."
 }
 }

 return fmt.Sprintf("%s", str)
}

// TODO: Agregar un método "String() string" a IPAddr.

func main() {
 hosts := map[string]IPAddr{
 "loopback": {127, 0, 0, 1},
 "googleDNS": {8, 8, 8, 8},
 }

 for _, ip := range hosts {
 fmt.Println(ip)
 }
}

```

## ❖ Errores

Para mostrar errores, Go utiliza valores de error para expresar estados de error, y para esto, existe el tipo `error` y es similar a la interfaz `fmt.Stringer`:

```

type error interface {
 Error() string
}

```

Exactamente como con `fmt.Stringer`, el paquete `fmt` busca la interfaz `error` al imprimir valores. Normalmente, los métodos devuelven un valor de `error` y deberíamos usarlo para gestionar qué hacer en caso de que sea diferente a `nil`:

```
i, err := strconv.Atoi("42")

if err != nil {
 fmt.Println("no se pudo convertir el número: %v\n", err)
 return
}

fmt.Println("Entero convertido: ", i)
```

## 先进技术

Otra gran interfaz que representa el extremo de lectura de un flujo de datos, estos datos pueden ser transmitidos a través de archivos, conexiones de red, compresores, cifrados, etc.

Y tiene un método `Read`:

```
func (T) Read(b []byte) (n int, err error)
```

Este método poblará el array de bytes con datos y devolverá el número de bytes poblados y un valor de error. Devuelve un error de `io.EOF` cuando termina el flujo.

```
func main() {
 data := "Gentleman Programming"

 // crea un nuevo io.Reader leyendo desde data
 lector := strings.NewReader(data)

 // crea un buffer para almacenar los datos copiados
 var buffer strings.Builder

 // copia los datos del lector a un buffer. io.Copy lee del lector y escribe en el escritor hasta que
 n, err := io.Copy(&buffer, lector)

 if err != nil {
 fmt.Println("Error:", err)

 } else {
 fmt.Println("\n%d bytes copiados exitosamente. \n", n)

 // accede a los datos copiados en el buffer
```

```

 fmt.Println("Datos copiados:", buffer.String())
 }
}

```

Ejemplo, ¡obtengamos una cadena cifrada y descifrémosla!

```

package main

import (
 "io"
 "os"
 "strings"
)

type rot13Reader struct {
 r io.Reader
}

func (rr *rot13Reader) Read(p []byte) (n int, err error) {
 n, err = rr.r.Read(p)
 for i := 0; i < n; i++ {
 if (p[i] >= 'A' && p[i] <= 'Z') || (p[i] >= 'a' && p[i] <= 'z') {
 if p[i] <= 'Z' {
 // p[i] - 'A' calcula la posición del carácter actual en relación con 'A', luego agregamos 13 y
 // luego aplicamos '%26' para asegurarnos de que el resultado esté dentro del rango del alfabeto
 // y al final agregamos 'A' que convierte el resultado nuevamente al valor ASCII de una letra
 p[i] = (p[i]-'A'+13)%26 + 'A'
 } else {
 p[i] = (p[i]-'a'+13)%26 + 'a'
 }
 }
 }
 return
}

func main() {
 s := strings.NewReader("Lbh penpxrq gur pbqr!")
 r := rot13Reader{s}
 io.Copy(os.Stdout, &r)
}

```

## ❖ Imágenes

El paquete `image` define la interfaz `Image`, que es una herramienta poderosa para trabajar con imágenes ya que puedes crear, manipular y decodificar varios tipos de imágenes como PNG, JPEG, GIF, BMP, y más:

```
package image
```

```
type Image interface {
 ColorModel() color.Model
 Bounds() Rectangle
 At(x, y int) color.Color
}
```

Creemos una imagen pequeña y negra con un píxel rojo en el centro:

```
package main

import (
 "image"
 "image/color"
 "image/png"
 "os"
)

func main() {
 // Crea una nueva imagen RGBA con dimensiones 100x100
 img := image.NewRGBA(image.Rect(0, 0, 100, 100))

 // Establece todos los pixeles en negro
 for x := 0; x < 100; x++ {
 for y := 0; y < 100; y++ {
 img.Set(x, y, color.Black)
 }
 }

 // Establece el píxel en el centro en rojo
 img.Set(50, 50, color.RGBA{255, 0, 0, 255})

 // Crea un archivo PNG para guardar la imagen
 archivo, err := os.Create("imagen_simple.png")
 if err != nil {
 panic(err)
 }
 defer archivo.Close()

 // Codifica la imagen al formato PNG y guárdala en el archivo
 err = png.Encode(archivo, img)
 if err != nil {
 panic(err)
 }

 println("Imagen simple generada exitosamente!")
}
```

## ❖ GoRoutines

Como mencionamos anteriormente, Go es un lenguaje que admite la programación concurrente a través de "GoRoutines". Una GoRoutine es un hilo ligero gestionado por el tiempo de ejecución de Go, lo que te permite ejecutar múltiples funciones concurrentemente.

PERO es diferente a otros lenguajes, ya que es un hilo virtual que se ejecuta en un hilo real y es gestionado por el tiempo de ejecución de Go.

Para ejecutar una función como GoRoutine, solo necesitas agregar la palabra clave `go` antes de la llamada a la función:

```
go f(x, y, z)
```

Esto ejecutará la función `f(x, y, z)` concurrentemente en una nueva GoRoutine. Los parámetros se evalúan en el momento de la llamada a la función, por lo que si cambian más tarde, la GoRoutine usará los valores actualizados.

Veamos un ejemplo:

```
package main

import (
 "fmt"
)

func say(s string) {
 for i := 0; i < 3; i++ {
 fmt.Println(s)
 }
}

func main() {
 // Lanza una nueva goroutine para ejecutar la función say con "Hello"
 go say("Hello")

 // Imprime "World" 3 veces en la función principal
 for i := 0; i < 3; i++ {
 fmt.Println("Gentleman")
 }
}
```

Cuando ejecutas este código, la salida no será necesariamente "Hello" seguido de "Gentleman" tres veces cada uno. Esto se debe a que las goroutines se ejecutan concurrentemente. Podrás ver "Hello" y "Gentleman" mezclados.

Las goroutines son ligeras, por lo que puedes crear miles de ellas sin ningún problema de rendimiento. Son gestionadas por el tiempo de ejecución de Go, que las programa eficientemente en hilos reales del sistema operativo.

Otra gran característica es que se ejecutan en el mismo espacio de direcciones, por lo que pueden comunicarse entre sí utilizando canales para compartir memoria, pero esto también debe ser gestionado y sincronizado.

Para hacerlo, podemos usar los canales:

## ~~ Canales

Serán nuestra forma de comunicarnos entre goroutines, son tipados y se pueden utilizar para enviar y recibir datos con el operador de canal <-:

```
ch <- v // Envía v al canal ch.
v := <-ch // Recibe de ch y asigna el valor a v.
```

Los datos fluirán en la dirección de la flecha, por lo que si quieras enviar datos a un canal, debes usar la flecha que apunta al canal, y si quieras recibir datos de un canal, debes usar la flecha que apunta desde el canal.

También puedes crear un canal con la función make:

```
ch := make(chan int)
```

Esto creará un canal que enviará y recibirá enteros.

Por defecto, los envíos y recepciones bloquean hasta que el otro lado esté listo. Esto permite que las goroutines se sincronicen sin que tengamos que gestionar manualmente esa sincronización.

```
package main

import (
 "fmt"
)

func say(s string, ch chan string) {
 for i := 0; i < 3; i++ {
 fmt.Println(s)
 ch <- s // Envía "Hello" al canal después de cada impresión
 }
}

func main() {
 // Crea un canal para almacenar cadenas
 ch := make(chan string)

 // Lanza una nueva goroutine para ejecutar la función say
```

```
go say("Hello", ch)

// Espera infinitamente mensajes en el canal (asegura que se impriman todos los "Hello")
for {
 msg := <-ch // Recibe mensaje del canal
 fmt.Println("Received:", msg)
}

fmt.Println("Gentleman") // Imprime "Gentleman" después de recibir todos los mensajes
}
```

## ~~ Canales con Búfer

Todos los canales pueden tener un búfer, esto significa que pueden contener un número limitado de valores sin que haya un receptor correspondiente para esos valores.

Cuando el canal está lleno, el remitente se bloqueará hasta que el receptor haya recibido un valor. Esto es extremadamente útil cuando quieras enviar múltiples valores y no quieras perderlos si el receptor no está listo.

```
ch := make(chan int, 100)
```

Esto creará un canal que puede contener hasta 100 enteros.

Si envías más de 100 valores al canal, el remitente se bloqueará hasta que el receptor haya recibido algunos valores.

```
func main() {
 ch := make(chan int, 2)
 ch <- 1
 ch <- 2
 ch <- 3 // error fatal: todas las goroutines están dormidas - ;deadlock!

 fmt.Println(<-ch)
 fmt.Println(<-ch)
 fmt.Println(<-ch)
}
```

## ~~ Range y Close

¡Puedes cerrar un canal en cualquier momento! Un momento recomendado para cerrar un canal es cuando deseas señalar que no se enviarán más valores en él y quien debe hacerlo debería ser el remitente, nunca el receptor, ya que enviar en un canal cerrado causará un pánico:

```
v, ok := <-ch
ok será false si no hay más valores para recibir y el canal está cerrado.
```

Para recibir valores de un canal hasta que se cierre, puedes usar `range`:

```
for i:= range ch {
 fmt.Println(i)
}
```

¿Necesitamos cerrarlos? No necesariamente, solo si el receptor necesita saber que no se enviarán más valores, o si el remitente necesita decirle al receptor que ha terminado de enviar valores, de esta manera terminaremos el bucle `range`.

Ejemplo:

```
func say(s string, ch chan string) {
 for i := 0; i < 3; i++ {
 fmt.Println(s)
 ch <- s // Envía "Hello" al canal después de cada impresión
 }
 close(ch) // Cierra el canal después de enviar los mensajes
}

func main() {
 // Crea un canal para almacenar cadenas
 ch := make(chan string)

 // Lanza una nueva goroutine para ejecutar la función say
 go say("Hello", ch)

 // Bucle para recibir e imprimir mensajes hasta que el canal se cierre
 for {
 msg, ok := <-ch // Recibe el mensaje y verifica el estado abierto del canal
 if !ok {
 break // Salir del bucle si el canal está cerrado
 }
 fmt.Println("Received:", msg)
 }

 fmt.Println("Mensajes recibidos. Saliendo.")
}
```

## ~~ Selección de GoRoutines

La declaración `select` permite que una goroutine espere en múltiples operaciones de comunicación. Se bloquea hasta que uno de sus casos pueda ejecutarse, luego ejecuta ese caso.

Es útil cuando deseas esperar en múltiples canales y realizar acciones diferentes según qué canal esté listo.

```
func say(s string, ch chan string) {
 for i := 0; i < 3; i++ {
 fmt.Println(s)
 ch <- s // Envía "Hello" al canal después de cada impresión
 }
 // Cierra el canal después de que el bucle termine de enviar mensajes
 close(ch)
}

func main() {
 // Crea un canal para almacenar cadenas
 ch := make(chan string)

 // Lanza una nueva goroutine para ejecutar la función say
 go say("Hello", ch)

 // Usa select para manejar mensajes desde el canal o un tiempo de espera
 for {
 select {
 case msg, ok := <-ch: // Recibe el mensaje y verifica el estado abierto del canal
 if !ok {
 fmt.Println("Canal cerrado. Saliendo.")
 break
 }
 fmt.Println("Received:", msg)
 case <-time.After(1 * time.Second): // Expira después de 1 segundo si no se recibe ningún mensaje
 fmt.Println("Tiempo de espera esperando mensaje.")
 break
 }
 }
}
```

También puedes usar un caso `default` en una declaración `select`, esto se ejecutará si ningún otro caso está listo:

```
func say(s string, ch chan string) {
 for i := 0; i < 3; i++ {
 fmt.Println(s)
 ch <- s // Envía "Hello" al canal después de cada impresión
 }
 close(ch) // Cierra el canal después de enviar mensajes
}

func main() {
```

```

// Crea un canal para almacenar cadenas
ch := make(chan string)

// Lanza una nueva goroutine para ejecutar la función say
go say("Hello", ch)

// Usa select con un caso predeterminado
for {
 select {
 case msg, ok := <-ch:
 if !ok {
 fmt.Println("Canal cerrado. Saliendo.")
 break
 }
 fmt.Println("Received:", msg)
 case <-time.After(1 * time.Second):
 fmt.Println("Tiempo de espera esperando mensaje.")
 break
 default:
 fmt.Println("Nada para recibir o aún no se ha agotado el tiempo de espera.")
 }
}
}

```

Ahora hagamos un ejercicio donde comprobaremos si dos árboles de nodos tienen la misma secuencia de valores:

```

package main

import (
 "fmt"
)

type TreeNode struct {
 Val int
 Left *TreeNode
 Right *TreeNode
}

func isSameSequence(root1, root2 *TreeNode) bool {
 seq1 := make(map[int]bool)
 seq2 := make(map[int]bool)

 traverse(root1, seq1)
 traverse(root2, seq2)

 return equal(seq1, seq2)
}

func traverse(node *TreeNode, seq map[int]bool) {
 if node == nil {

```

```

 return
 }

 seq[node.Val] = true
 traverse(node.Left, seq)
 traverse(node.Right, seq)
}

func equal(seq1, seq2 map[int]bool) bool {
 if len(seq1) != len(seq2) {
 return false
 }

 for val := range seq1 {
 if !seq2[val] {
 return false
 }
 }

 return true
}

func main() {
 // Construyendo el primer árbol binario
 root1 := &TreeNode{
 Val: 3,
 Left: &TreeNode{
 Val: 1,
 Left: &TreeNode{
 Val: 1,
 },
 Right: &TreeNode{
 Val: 2,
 },
 },
 Right: &TreeNode{
 Val: 8,
 Left: &TreeNode{
 Val: 5,
 },
 Right: &TreeNode{
 Val: 13,
 },
 },
 }
}

// Construyendo el segundo árbol binario
root2 := &TreeNode{
 Val: 8,
 Left: &TreeNode{
 Val: 3,
 Left: &TreeNode{

```

```

 Val: 1,
 Left: &TreeNode{
 Val: 1,
 },
 Right: &TreeNode{
 Val: 2,
 },
},
Right: &TreeNode{
 Val: 5,
},
},
Right: &TreeNode{
 Val: 13,
},
}
}

fmt.Println(isSameSequence(root1, root2)) // Salida: true

```

## ❖ Mutex

Algo de lo que debemos cuidarnos al trabajar con GoRoutines es el acceso a la memoria compartida, donde más de una GoRoutine puede acceder al mismo espacio de memoria al mismo tiempo, lo que puede provocar grandes conflictos.

Este concepto se llama exclusión mutua, y se resuelve mediante el uso de mutexes, que se utilizan para sincronizar el acceso a la memoria compartida.

```

import ("sync")

var mu sync.Mutex

```

Tiene dos métodos, **Lock** y **Unlock**, que se utilizan para proteger la memoria compartida:

```

func safeIncrement() {
 mu.Lock() // bloquea la memoria compartida
 defer mu.Unlock() // desbloquea la memoria compartida cuando la función retorna
 count++ // incrementa la memoria compartida
}

```

Aquí estamos utilizando la declaración **defer** para asegurar que el mutex se desbloquee cuando la función retorne, incluso si entra en pánico.

```

package main

import (

```

```

 "fmt"
 "sync"
)

type TreeNode struct {
 Left *TreeNode
 Right *TreeNode
 Val int
}

type SequenceCollector struct {
 sequence map[int]bool
 mutex sync.Mutex
}

func isSameSequence(root1, root2 *TreeNode) bool {
 seq1 := &SequenceCollector{sequence: make(map[int]bool)}
 seq2 := &SequenceCollector{sequence: make(map[int]bool)}

 var wg sync.WaitGroup
 wg.Add(2)

 go func() {
 defer wg.Done()
 traverse(root1, seq1)
 }()

 go func() {
 defer wg.Done()
 traverse(root2, seq2)
 }()

 return equal(seq1.sequence, seq2.sequence)
}

func traverse(node *TreeNode, seq *SequenceCollector) {
 if node == nil {
 return
 }

 seq.mutex.Lock()
 seq.sequence[node.Val] = true
 seq.mutex.Unlock()

 traverse(node.Left, seq)
 traverse(node.Right, seq)
}

func equal(seq1, seq2 map[int]bool) bool {
 if len(seq1) != len(seq2) {
 return false
 }
}

```

```
 for val := range seq1 {
 if !seq2[val] {
 return false
 }
 }

 return true
 }

func main() {
 root1 := &TreeNode{
 Val: 3,
 Left: &TreeNode{
 Val: 1,
 Left: &TreeNode{
 Val: 1,
 },
 Right: &TreeNode{
 Val: 2,
 },
 },
 Right: &TreeNode{
 Val: 8,
 Left: &TreeNode{
 Val: 5,
 },
 Right: &TreeNode{
 Val: 13,
 },
 },
 }
 root2 := &TreeNode{
 Val: 8,
 Left: &TreeNode{
 Val: 3,
 Left: &TreeNode{
 Val: 1,
 Left: &TreeNode{
 Val: 1,
 },
 Right: &TreeNode{
 Val: 2,
 },
 },
 Right: &TreeNode{
 Val: 5,
 },
 },
 Right: &TreeNode{
 Val: 13,
 },
 },
}
```

```
}

 fmt.Println(isSameSequence(root1, root2)) // true
}
```

# Guía Gentil de NVIM !

¡Primero que nada...

## ~ ¿Qué es NVIM?

Vim fue inventado por primera vez el 2 de noviembre de 1991 por Bram Moolenaar. Es un editor de texto altamente configurable construido para permitir una edición eficiente de texto.

Es una versión mejorada del editor vi distribuido con la mayoría de los sistemas UNIX. Vim a menudo se llama un "editor para programadores", y es tan útil para la programación que muchos lo consideran un IDE completo.

Pero no es solo para programadores. Vim es perfecto para todo tipo de edición de texto, desde componer correos electrónicos hasta editar archivos de configuración.

Entonces... ¿Qué es NVIM entonces?

NVIM es un fork de Vim, con muchas características nuevas, mejor rendimiento y muchos nuevos complementos y configuraciones.

Es un editor de texto altamente configurable que se puede usar para programar, escribir y editar archivos de texto. ¡Lo más divertido es que ni siquiera ha sido lanzado todavía! En el momento de escribir esta guía, NVIM todavía está en la fase beta 0.9.5.

Pero es lo suficientemente estable como para usarse a diario.

## ~ ¿Por qué amo NVIM?

Eficiencia, Velocidad y Personalización. Permíteme hablarte un poco de mí, soy un ingeniero de software y paso la mayor parte de mi tiempo escribiendo código.

He probado muchos editores de texto y entornos de desarrollo integrado (IDE), pero siempre vuelvo a Vim. ¿Por qué? Porque es rápido, eficiente y altamente personalizable.

Pero Vim tiene sus propios problemas, es difícil de configurar y tiene una curva de aprendizaje pronunciada para los principiantes, pero como amante de Dark Souls... Me encanta el desafío.

Una vez que dominas la bestia, nunca volverás atrás. Tener el poder de no alejarme de mi teclado, aprender algo nuevo todos los días, jugar con nuevos complementos y crear mis propias configuraciones es lo que me hace amar Vim.

Puede ejecutarse en cualquier lugar, en cualquier plataforma, es rápido, es ligero, puedes compartir tu configuración muy fácilmente y es de código abierto. Y la mejor parte...

te ves increíble usándolo, mucha gente te preguntará "¿qué es eso?" y te sentirás como un hacker en una película, ejecutando comandos y atajos y mostrando el poder de tu eficiencia.

Bien, comenzemos esta guía, te mostraré cómo instalar NVIM, configurarlo y usarlo como tu editor principal. Te mostraré cómo instalar complementos, crear tus propias configuraciones y hacer que se vea increíble.

Comencemos con la instalación.

## ~ Instalación

### Requisitos previos

(Ejecuta todos los comandos usando el terminal predeterminado del sistema, lo cambiaremos más tarde)

USUARIOS DE WINDOWS:

Primero, instalamos WSL (<https://learn.microsoft.com/en-us/windows/wsl/install>), esto es imprescindible para los usuarios de Windows, te permitirá ejecutar una distribución completa de Linux en tu máquina con Windows y recomiendo usar la versión 2 ya que utiliza todos los recursos de la máquina.

```
wsl --install
wsl --set-default-version 2
```

Como ahora estamos ejecutando una distribución completa de Linux en nuestra máquina con Windows, los siguientes pasos serán los mismos para todas las plataformas, ya sea Windows, macOS o Linux.

1- Instala Homebrew, este es un administrador de paquetes para macOS y Linux, te permitirá instalar muchos paquetes y herramientas fácilmente y siempre está actualizado.

```
set install_script $(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)
```

Incluye la ruta de HomeBrew en tu perfil de shell

```
Cambia 'TuNombreDeUsuario' por el nombre de usuario del dispositivo

(echo; echo 'eval "$(($(/home/linuxbrew/.linuxbrew/bin/brew shellenv))")' >> /home/YourUserName/.bashrc
eval "$(($(/home/linuxbrew/.linuxbrew/bin/brew shellenv))")"
```

2- Instala build-essential, este es un paquete que contiene una lista de paquetes esenciales para compilar software y lo necesitaremos para compilar algunos complementos. Este paso no es necesario para los

usuarios de macOS.

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install build-essential
```

3- Instala NVIM, instalaremos NVIM usando Homebrew, esto instalará la última versión de NVIM y todas sus dependencias.

```
brew install nvim
```

4- Instala Node y NPM, necesarios para los complementos de desarrollo web y algunos servidores de lenguaje.

```
brew install node
brew install npm
```

5- Instala GIT, necesitaremos GIT para clonar repositorios.

```
brew install git
```

6- Instala FISH, este es un shell altamente personalizable que tiene muchas funciones, te recomiendo usarlo como tu shell predeterminado. Algunas de sus increíbles características son la autocompletación y el resaltado de sintaxis.

```
```bash  
brew install fish  
// establecer como predeterminado:  
  
which fish  
// esto devolverá una ruta, llamémosla whichFishResultingPath  
  
// agregarlo como un shell disponible  
echo whichFishResultingPath | sudo tee -a /etc/shells  
  
// establecerlo como predeterminado  
sudo chsh -s whichFishResultingPath
```

7- Instala Oh My Fish, este es un marco para el shell fish, te permitirá instalar temas, complementos y configurar tu shell fácilmente.

```
curl https://raw.githubusercontent.com/oh-my-fish/oh-my-fish/master/bin/install | fish
```

8- Instala las siguientes dependencias necesarias para ejecutar LazyVim

```
brew install gcc  
brew install fzf  
brew install fd  
brew install ripgrep
```

9- Instala Zellij, este es un multiplexor de terminal, te permitirá dividir tu terminal en múltiples paneles y ejecutar múltiples comandos al mismo tiempo.

```
brew install zellij
```

10 - Instala Wezterm, este es un emulador de terminal, es altamente personalizable y tiene muchas características, te recomiendo usarlo como tu terminal predeterminado. Una de las características más fuertes es la aceleración de GPU, hará que tu terminal sea más rápida y receptiva y está escrito en lua, el mismo lenguaje que utiliza LAZYVIM.

<https://wezfurlong.org/wezterm/index.html>

11- Instala Iosevka Term Nerd Font, esta es una fuente altamente personalizable y tiene muchas características, te recomiendo usarla como tu fuente predeterminada. Tiene muchas ligaduras y caracteres especiales que harán que tu terminal se vea increíble. Una fuente nerd es una fuente que tiene muchos caracteres especiales y ligaduras que harán que tu terminal se vea increíble y se necesita para renderizar iconos.

<https://github.com/ryanoasis/nerd-fonts/releases/download/v3.1.1/IosevkaTerm.zip>

12- Ahora permíteme compartirte mi repositorio personalizado que contiene todas mis configuraciones para NVIM, FISH, Wezterm y Zellij.

<https://github.com/Gentleman-Programming/Gentleman.Dots>

Solo sigue los pasos y tendrás un terminal y un editor de código completamente personalizados y gentlemanizados. Una última cosa antes de continuar, usaremos algunos complementos que ya están configurados dentro del repositorio y están gestionados por LazyVim, un increíble administrador de paquetes que te permitirá instalar y actualizar complementos fácilmente.

Ahora que tenemos todo... ¡comencemos a aprender cómo configurar NVIM!

❖ Configuración

Como estamos utilizando mi repositorio personalizado, todas las configuraciones ya están hechas, pero explicaré cómo configurar NVIM y cómo instalar complementos y crear tus propias configuraciones.

Como se mencionó anteriormente, estamos utilizando LazyVim, <http://www.lazyvim.org/>, un increíble administrador de paquetes que te permitirá instalar y actualizar complementos sin problemas como la mantequilla, también proporciona complementos ya configurados que harán tu vida más fácil.

Pero primero, aprendamos cómo instalar complementos manualmente.

Si observas la estructura de carpetas de nvim, encontrarás una carpeta "plugins", que contendrá varios archivos que representan cada uno de los complementos instalados manualmente.

Cada archivo contendrá el nombre del complemento y la URL del repositorio. Para instalar un complemento manualmente, necesitarás crear un nuevo archivo dentro de la carpeta "plugins" y agregar el siguiente contenido.

```
return {  
    "url-del-repositorio",  
}
```

Y eso es todo, la próxima vez que abras NVIM, el complemento se instalará automáticamente.

Para acceder a una ventana de gestión de LazyVim, simplemente abre nvim usando el comando "nvim" en tu terminal y luego escribe el siguiente comando ":LazyVim", esto abrirá una ventana con todos los complementos instalados y su estado, puedes instalar, actualizar y eliminar complementos usando esta ventana.

Ahora, para acceder a todos los complementos extra ya configurados proporcionados por LazyVim, simplemente escribe el siguiente comando ":LazyVimExtra", esto abrirá una ventana con todos los complementos disponibles, puedes instalar, actualizar y eliminar complementos usando esta ventana.

Para instalar un nuevo lenguaje de programación, escribe ":MasonInstall" y selecciona el lenguaje que deseas instalar, esto instalará todos los complementos y configuraciones necesarios para ese lenguaje y eso es todo, estás listo para comenzar.

Para establecer nuevos atajos de teclado, simplemente abre el archivo "keymaps.lua" dentro de la carpeta "lua" y agrega el siguiente contenido.

```
vim.keymap.set('modo', 'quéQuieresPresionar', 'QuéQuieresHacer')
```

El modo representa el modo que vas a usar, puede ser "n" para el modo normal, "i" para el modo de inserción, "v" para el modo visual y "c" para el modo de comando. El segundo parámetro representa la tecla que deseas presionar y el tercer parámetro representa la acción que deseas realizar. Debes regresar a esta parte más tarde después de que toquemos los conceptos básicos de los modos de vim.

❖ Conceptos Básicos

Nvim tiene 4 modos: Normal, Inserción, Visual y Comando. Cada uno de ellos tiene su propio propósito y atajos.

Modo Normal

En este modo, puedes navegar por el texto, eliminar, copiar, pegar y ejecutar comandos. Puedes ingresar a este modo presionando la tecla "ESC". En resumen, este es el modo en el que pasarás la mayor parte de tu tiempo y donde nos moveremos a través de nuestro código.

Movimiento Horizontal

Para navegar, NO usaremos las teclas de flecha, usaremos la tecla "h" para moverse hacia la izquierda, la tecla "j" para moverse hacia abajo, la tecla "k" para moverse hacia arriba y la tecla "l" para moverse hacia la derecha. Esta es la forma más eficiente de navegar por el texto y te hará lucir como un profesional.

Realmente te recomiendo usar las teclas "hjkl" para navegar, te hará más eficiente y no necesitarás mover las manos desde la fila de inicio. La eficiencia es el nombre del juego.

Para saltar al principio de la línea, usa la tecla "0", para saltar al final de la línea, usa la tecla "\$". Para saltar al principio del archivo, usa las teclas "gg", para saltar al final del archivo, usa la tecla "G". Siempre es lo mismo, si presionas un comando, harás algo, y si presionas "Shift" mientras lo haces, harás lo contrario.

Para moverte horizontalmente correctamente a través de una línea, necesitarás usar la tecla "w" para saltar al principio de la siguiente palabra, la tecla "b" para saltar al principio de la palabra anterior, la tecla "e" para saltar al final de la siguiente palabra y la tecla "ge" para saltar al final de la palabra anterior. También puedes usar la increíble tecla "f" para saltar a un carácter específico en la línea, simplemente presiona "f" y luego el carácter al que deseas saltar, y eso es todo, estás allí. Y como dije antes, si usas "Shift" mientras lo haces, harás lo contrario, moviéndote a la ocurrencia anterior. También puedes usar la tecla "s" para buscar un carácter, esto utiliza un complemento llamado "Sneak", que te permitirá buscar un carácter y saltar a él después de presionar la tecla que aparecerá junto a todas las ocurrencias.

Movimiento Vertical

Para navegar verticalmente, puedes usar la tecla "Ctrl" con la tecla "u" para moverte hacia arriba medio página, la tecla "Ctrl" con la tecla "d" para moverte hacia abajo medio página, la tecla "Ctrl" con la tecla "b" para moverte hacia arriba una página y la tecla "Ctrl" con la tecla "f" para moverte hacia abajo una página.

Esta es la forma más eficiente de navegar por el texto, ya que no sabemos dónde está esa parte específica de lógica, así que podemos movernos bastante rápido de esta manera y encontrar lo que estamos buscando.

Otra excelente manera de navegar verticalmente es usar la tecla "Shift" con las teclas "y", esto te permitirá saltar al siguiente o al párrafo anterior, y esto es una de las cosas que me hace amar NVIM, si tu código está limpio y correctamente indentado, podrás saltar a través del código realmente rápido y encontrar lo que estás buscando, ¡TE ESTÁ ENSEÑANDO A ESCRIBIR CÓDIGO LIMPIO!

Si deseas saltar a una línea específica, puedes usar la tecla ":" , esto abrirá el modo de comando y luego puedes escribir el número de línea al que deseas saltar, y eso es todo, estás allí.

Modo Visual

Este modo se utiliza para seleccionar texto, puedes ingresar a este modo presionando la tecla "v". Puedes usar los mismos comandos que en el modo normal, pero ahora puedes seleccionar texto.

También puedes usar la tecla "Shift" con la tecla "v" para seleccionar toda la línea. Nuevamente, podemos usar movimientos para seleccionar texto, por ejemplo, si queremos seleccionar las próximas 10 líneas, podemos usar el comando "10j" y si queremos seleccionar las próximas 10 palabras, podemos usar el comando "10w". Y si queremos seleccionar los próximos 10 caracteres, podemos usar el comando "10l".

Modo de Bloque Visual

Este modo se utiliza para seleccionar un bloque de texto, puedes ingresar a este modo presionando la tecla "Ctrl" junto con la tecla "v". Puedes usar los mismos comandos que en el modo normal, pero ahora puedes seleccionar un bloque de texto.

Nuevamente, podemos usar movimientos para seleccionar texto, por ejemplo, si queremos seleccionar las próximas 10 líneas, podemos usar el comando "10j", y si queremos seleccionar las próximas 10 palabras, podemos usar el comando "10w". Y si queremos seleccionar los próximos 10 caracteres, podemos usar el comando "10l".

Un bloque de texto es un rectángulo de texto, y puedes copiar, pegar y eliminarlo. También puedes usar la tecla "Shift" con la tecla "I" para insertar texto en un bloque, y la tecla "Shift" con la tecla "A" para agregar texto en un bloque.

También es útil para escribir muchas líneas al mismo tiempo, por ejemplo, si quieras escribir un comentario en varias líneas, puedes usar la tecla "Ctrl" con la tecla "v" para seleccionar las líneas donde deseas escribir el comentario, y luego usar la tecla "Shift" con la tecla "I" para insertar el comentario, y eso es todo, estás hecho después de presionar la tecla "ESC".

Modo de Línea Visual

Este modo se utiliza para seleccionar una línea de texto, puedes ingresar a este modo presionando la tecla "Shift" junto con la tecla "v".

Puedes usar los mismos comandos que en el modo normal, pero ahora puedes seleccionar una línea de texto. Nuevamente, podemos usar movimientos para seleccionar texto, por ejemplo, si queremos seleccionar las próximas 10 líneas, podemos usar el comando "10j", y si queremos seleccionar las próximas 10 palabras, podemos usar el comando "10w". Y si queremos seleccionar los próximos 10 caracteres, podemos usar el comando "10l".

Modo de Inserción

Este es el modo que utilizarás para escribir texto, puedes ingresar a este modo presionando la tecla "i". Puedes usar los mismos comandos que en el modo normal, pero ahora puedes escribir texto. También puedes usar la tecla "Shift" con la tecla "i" para insertar texto al principio de la línea, y la tecla "A" para agregar texto

al final de la línea, y lo mismo si quieras comenzar a escribir en un carácter específico, puedes usar "i" para insertar antes del carácter y "a" para agregar después del carácter.

También puedes usar la tecla "o" para insertar una nueva línea debajo de la línea actual, y la tecla "O" para insertar una nueva línea encima de la línea actual. Usar la tecla "Ctrl" con la tecla "w" eliminará la última palabra, y usar la tecla "Ctrl" con la tecla "u" eliminará la última línea mientras estás en modo de inserción.

Otro comando útil es la tecla "Ctrl" con la tecla "n", esto autocompletará el texto que estás escribiendo, y es realmente útil cuando estás escribiendo código. Y si quieres salir del modo de inserción, puedes usar la tecla "ESC".

Modo de Comando

Este modo se utiliza para ejecutar comandos, puedes ingresar a este modo presionando la tecla ":". ¡Aquí es donde podemos salir de NVIM! Simplemente escribe ":q" ¡y listo! si tienes cambios, primero guárdalos usando ":w" y si quieres forzar la salida ":q!".

Otra cosa genial es que puedes hacer más de un comando a la vez, por ejemplo, si quieres guardar y salir, puedes usar ":wq".

Una configuración que recomiendo es establecer el número de líneas de manera relativa haciendo ":set relativenumber", esto te permitirá ver el número de línea en relación con la línea en la que te encuentras, y es realmente útil para saber dónde estás en el archivo. Puedes hacer esto escribiendo el siguiente comando ":set relativenumber", y queremos esto ya que podemos movernos rápidamente a una línea específica usando un número y la dirección a la que queremos ir, por ejemplo, si queremos saltar a la décima línea arriba de nosotros, podemos usar el comando "10k", y si queremos saltar a la décima línea debajo de nosotros, podemos usar el comando "10j".

~ Movimientos en NVIM

Y esto introduce el concepto de "Movimientos" en NVIM, cada comando que escribimos es un movimiento, y se crea combinando un número, una dirección y un comando.

Por ejemplo, si queremos eliminar las próximas 10 líneas, podemos usar el comando "10dd", y si queremos copiar las próximas 10 líneas, podemos usar el comando "10yy". Esta es la forma más eficiente de navegar por el texto y una de las características más sólidas de NVIM.

Ahora usemos lo que hemos aprendido para eliminar, copiar y pegar texto.

Para eliminar texto, podemos usar la tecla "d", y luego el movimiento que queramos usar, por ejemplo, si queremos eliminar las próximas 10 líneas, podemos usar el comando "10dd", y si queremos eliminar las próximas 10 palabras, podemos usar el comando "10dw". Y si queremos eliminar los próximos 10 caracteres, podemos usar el comando "10dl", y si queremos eliminar toda la línea podemos usar el comando "dd".

Para copiar texto, podemos usar la tecla "y", y luego el movimiento que queramos usar, por ejemplo, si queremos copiar las próximas 10 líneas, podemos usar el comando "10yy", y si queremos copiar las próximas

10 palabras, podemos usar el comando "10yw". Y si queremos copiar los próximos 10 caracteres, podemos usar el comando "10yl", y si queremos copiar toda la línea podemos usar el comando "yy".

Para pegar texto, podemos usar la tecla "p", esto pegará el texto después del cursor, y si queremos pegar el texto antes del cursor, podemos usar la tecla "P".

~~ RegistrOS

Y ahora viene lo divertido, ¿has visto qué sucede cuando eliminamos o copiamos texto? el texto se guarda en un registro, y podemos acceder a él usando la tecla "p", y podemos acceder al último texto eliminado usando la tecla "P". Esto es algo que muchos principiantes odian porque no saben qué es un registro o cómo acceder a él, así que déjame explicártelo.

Un registro es un lugar donde se guarda el texto, y podemos acceder a él usando el "Leader" (normalmente "Space") y la tecla de comillas dobles, a veces necesitamos hacer "Leader" y comillas dobles dos veces si tu distribución es Internacional, y aparecerá un panel con todos los registros, y verás que el texto copiado más reciente se guarda en el registro "0", así que ahora que sabemos esto, puedes acceder a él usando el comando "0p". Y si quieras acceder al último texto eliminado, puedes usar el comando "1p".

~~ Buffers

Un buffer es un lugar donde se guarda el texto, y puedes acceder a él usando la tecla "Leader" y "be", y aparecerá un panel con todos los buffers, y puedes navegar por ellos usando las teclas "j" y "k".

También puedes usar la tecla "d" para eliminar un buffer. Una forma de pensar en los buffers es como pestañas, puedes tener varios buffers abiertos al mismo tiempo, y puedes navegar por ellos, cada vez que abres un archivo se crea un nuevo buffer y se guarda en la memoria, y si abres el mismo buffer en dos lugares al mismo tiempo verás que si cambias algo en un buffer, cambiará en el otro buffer también.

Hay un comando especial que creé para que puedas borrar todos los buffers excepto el actual para esos momentos especiales en los que has estado programando durante horas y el rendimiento es un poco lento, puedes hacer "Leader" y "bq".

~~ Marcas

Las marcas son increíbles, puedes crear una nueva marca usando la tecla "m" y luego una letra, por ejemplo, si quieras crear una nueva marca en la línea actual, puedes usar el comando "ma", y si quieras saltar a esa marca, puedes usar el comando ``a".

Si quieres eliminar una marca, haz ":delm letraDeLaMarca", y para eliminar TODAS las marcas ":delm!". Las marcas se guardan en el buffer actual, y puedes usarlas para navegar rápidamente por el texto.

~ Grabaciones

Ahora esto es increíble y súper útil, digamos que necesitamos hacer una acción múltiples veces y es súper tedioso hacerlo, lo que NVIM proporciona es una forma de replicar un conjunto de comandos creando una macro, puedes comenzar a grabar usando la tecla "q" y luego una letra, por ejemplo, si quieras comenzar a grabar una macro en el registro "a", puedes usar el comando "qa", y luego puedes hacer las acciones que quieras replicar, y luego puedes detener la grabación usando la tecla "q".

Para reproducir la macro, puedes usar la tecla "@" y luego la letra, por ejemplo, si quieres reproducir la macro en el registro "a", puedes usar el comando "@a".

Esto es súper útil y te hará más eficiente.

Y nuevamente puedes usar movimientos con tus grabaciones, por ejemplo, si quieres eliminar las próximas 10 líneas y copiarlas, puedes usar el comando "qad10jyy", y luego puedes reproducir la macro usando el comando "@a", y también puedes replicar la macro varias veces usando el comando "10@a".

Algoritmos a la Manera Caballerosa

❖ Notación Big O

"Cómo el código se ralentiza a medida que crece la cantidad de datos"

- El rendimiento de un algoritmo depende de la cantidad de datos que se le proporciona.
- Número de pasos necesarios para completar. Algunas máquinas ejecutan algoritmos más rápido que otras, por lo que simplemente tomamos el número de pasos necesarios.
- Ignorar operaciones más pequeñas, constantes. $O(N + 1) \rightarrow O(N)$ donde N representa la cantidad de datos.

```
function sumar(n: number): number {
  let suma = 0;

  for(let i = 0; i < n; i++) {
    suma += i;
  }

  return suma;
}

// si n es igual a 10, entonces O(N) son 10 pasos, si n es igual a 100, entonces O(N) son 100 pasos
```

Aquí podemos ver que $O(N)$ es lineal, lo que significa que la cantidad de pasos depende del número de datos que se nos proporciona.

```
function saludar(nombre: string): string{
  return `¡Hola ${nombre}!`

}

// si n es igual a 10, entonces O(1) son 3 pasos... ¿3? ¡SÍ, 3 pasos!

// 1 - Crear nuevo objeto de cadena para almacenar el resultado (asignando memoria para la nueva cadena)
// 2 - Concatenar la cadena '¡Hola' con el resultado.
// 3 - Devolver la cadena concatenada.
```

Pero ahora tenemos $O(1)$ ya que la cantidad de pasos no depende de la cantidad de datos que se nos proporciona, siempre será 1.

Conocimientos Previos Necesarios

- El 'Logaritmo' de un número es la potencia a la que se debe elevar la base para producir ese número. Por ejemplo, el logaritmo base 2 de 8 es 3 porque $2^3 = 8$.
- 'Lineal' significa que el número de pasos crece linealmente con la cantidad de datos.
- El 'Cuadrático' de un número es el cuadrado de ese número. Por ejemplo, el cuadrado de 3 es 9 porque $3^2 = 9$.
- 'Exponencial' de un número es la potencia de la base elevada a ese número. Por ejemplo, el exponencial de 2 elevado a la potencia de 3 es 8 porque $2^3 = 8$.
- 'Factorial' de un número es el producto de todos los enteros positivos menores o iguales a ese número. Por ejemplo, el factorial de 3 es 6 porque $3! = 3 * 2 * 1 = 6$.
- 'Quicksort' es un algoritmo de ordenación que utiliza la estrategia de dividir y conquistar para ordenar una matriz. Es una ordenación por comparación y no es estable. Divide y vencerás es una estrategia para resolver un problema dividiéndolo en partes más pequeñas y resolviendo cada parte individualmente.

Tipos de Notación Big O

- $O(1)$ - Tiempo constante - Siempre el mismo número de pasos independientemente de la cantidad de datos
- $O(\log N)$ - Tiempo logarítmico - El número de pasos crece logarítmicamente (búsqueda binaria)
- $O(N)$ - Tiempo lineal - El número de pasos crece linealmente (bucles)
- $O(N \log N)$ - Tiempo linealítmico - El número de pasos crece linealítmicamente (ordenación rápida)
- $O(N^2)$ - Tiempo cuadrático - El número de pasos crece cuadráticamente (bucles anidados)
- $O(2^N)$ - Tiempo exponencial - El número de pasos crece exponencialmente (algoritmos recursivos)
- $O(N!)$ - Tiempo factorial - El número de pasos crece factorialmente (algoritmos de fuerza bruta, aquel

Ejemplo con N igual a 1000:

- $O(1)$ - 1 paso
- $O(\log N)$ - 10 pasos
- $O(N)$ - 1000 pasos, mil pasos
- $O(N \log N)$ - 10000 pasos, diez mil pasos
- $O(N^2)$ - 1000000 pasos, un millón de pasos
- $O(2^N)$ - 2^{1000} pasos
- $O(N!)$ - $1000!$ pasos, factorial de 1000

La idea principal es que queremos evitar los algoritmos de tiempo exponencial y factorial ya que crecen muy rápido y no son eficientes en absoluto, A MENOS que estemos seguros de que la cantidad de datos que se nos proporciona es muy pequeña, ya que en realidad puede ser más rápido que otros algoritmos.

Calificación de letras para la Notación Big O, de mejor a peor, teniendo en cuenta que estamos utilizando un gran conjunto de datos:

- $O(1)$ - Tiempo constante - A
- $O(\log N)$ - Tiempo logarítmico - B

- $O(N)$ - Tiempo lineal - C
- $O(N \log N)$ - Tiempo linealítmico - D
- $O(N^2)$ - Tiempo cuadrático - F
- $O(2^N)$ - Tiempo exponencial - F
- $O(N!)$ - Tiempo factorial - F

Ejemplos usando código

$O(1)$ - Tiempo constante

```
function sayHi(n: string): string{
    return `Hola ${n}`
}
```

Aquí está por qué es $O(1)$:

- El algoritmo realiza una cantidad constante de trabajo, independientemente del tamaño de la entrada.
- El número de pasos necesarios para completar el algoritmo no depende del tamaño de la entrada.

Por lo tanto, la complejidad temporal del algoritmo es $O(1)$ en todos los casos.

$O(\log N)$ - Tiempo logarítmico

```
// teniendo el siguiente array que representa los números del 0 al 9 en orden
const arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

// queremos encontrar el índice de un número en el array ordenado
function binarySearch(arr: number[], target: number): number {
    // inicializamos los punteros izquierdo y derecho
    let left = 0;
    let right = arr.length - 1;

    // mientras que left sea menor o igual que right, seguimos buscando el target
    while (left <= right) {
        // obtenemos el medio del array para compararlo con el target
        // iteramos usando el medio del array para encontrar el target porque sabemos que el array está
        const mid = Math.floor((left + right) / 2); // índice medio
        const midValue = arr[mid]; // valor medio

        // si el valor medio es el target, devolvemos el índice
        if (midValue === target) {
            return mid;
        }

        // si el valor medio es menor que el target, buscamos el lado derecho del array actualizando el p
        if (midValue < target) {
```

```
        left = mid + 1;
    } else {
        right = mid - 1;
    }
}
return -1; // target no encontrado
}
```

En la búsqueda binaria, el algoritmo divide continuamente el intervalo de búsqueda a la mitad hasta que se encuentra el elemento objetivo o el intervalo de búsqueda se vacía. Con cada iteración, el algoritmo descarta la mitad del espacio de búsqueda en función de una comparación con el elemento medio del intervalo actual.

Aquí está por qué es $O(\log N)$:

- En cada iteración del bucle while, el espacio de búsqueda se divide a la mitad.
- Este proceso de división continúa hasta que el espacio de búsqueda se reduce a un solo elemento o se encuentra el objetivo.
- Dado que el espacio de búsqueda se divide a la mitad en cada iteración, el número de iteraciones requeridas para alcanzar el elemento objetivo crece de forma logarítmica con el tamaño del array de entrada.

Por lo tanto, la complejidad temporal de la búsqueda binaria es $O(\log N)$ en promedio.

$O(N)$ - Tiempo lineal

```
function sum(n: number): number {
    let sum = 0;
    for(let i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}
```

Aquí está por qué es $O(N)$:

- El algoritmo itera sobre el array de entrada una vez, realizando una cantidad constante de trabajo para cada elemento.
- El número de iteraciones es directamente proporcional al tamaño del array de entrada.
- A medida que aumenta el tamaño de la entrada, el número de pasos necesarios para completar el algoritmo crece linealmente.

Por lo tanto, la complejidad temporal del algoritmo es $O(N)$ en el peor de los casos.

O(N log N) - Tiempo linealítmico

```
// teniendo el siguiente array
const arr = [5, 3, 8, 4, 2, 1, 9, 7, 6];

// queremos ordenar el array usando el algoritmo quick sort
function quickSort(arr: number[]): number[] {
    // primero verificamos si el array tiene solo un elemento o ningún elemento
    if (arr.length <= 1) {
        return arr;
    }

    // obtenemos el pivote como el último elemento del array, el pivote es el elemento con el que vamos
    const pivot = arr[arr.length - 1];

    // creamos dos arrays, uno para los elementos menores que el pivote y otro para los elementos mayor
    const left = [];
    const right = [];

    // iteramos sobre el array y comparamos cada elemento con el pivote
    for (let i = 0; i < arr.length - 1; i++) {
        // si el elemento es menor que el pivote, lo agregamos al array izquierdo
        if (arr[i] < pivot) {
            left.push(arr[i]);
        } else {
            // si el elemento es mayor que el pivote, lo agregamos al array derecho
            right.push(arr[i]);
        }
    }

    // llamamos recursivamente a la función quickSort en los arrays izquierdo y derecho y concatenamos
    return [...quickSort(left), pivot, ...quickSort(right)];
}
```

Aquí está por qué es O(N log N):

- El algoritmo divide el array en dos subarrays basados en un elemento pivote y ordena recursivamente estos subarrays.
- Cada paso de particionamiento implica iterar sobre todo el array una vez, lo que lleva un tiempo O(N). Sin embargo, el array suele dividirse de tal manera que el tamaño de los subarrays se reduce con cada llamada recursiva. Esto resulta en una complejidad temporal de O(N log N) en promedio.

O(N^2) - Tiempo cuadrático

```
// Dado el siguiente arreglo
const arr = [5, 3, 8, 4, 2, 1, 9, 7, 6];
```

```

// Queremos ordenar el arreglo usando el algoritmo de ordenamiento burbuja
function bubbleSort(arr: number[]): number[] {

    // Iteramos sobre el arreglo
    for (let i = 0; i < arr.length; i++) {

        // Iteramos sobre el arreglo nuevamente
        for (let j = 0; j < arr.length - 1; j++) {

            // Comparamos elementos adyacentes y los intercambiamos si están en el orden incorrecto
            if (arr[j] > arr[j + 1]) {

                // Intercambiamos los elementos
                const temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    return arr;
}

```

Aquí está por qué es $O(N^2)$:

- El ordenamiento burbuja funciona repasando la lista repetidamente, comparando elementos adyacentes y cambiándolos si están en el orden incorrecto.
- En el peor de los casos, donde el arreglo está en orden inverso, el ordenamiento burbuja necesitará hacer N pasadas a través del arreglo, cada pasada requiriendo $N-1$ comparaciones e intercambios.
- Esto resulta en un total de $N * (N-1)$ comparaciones e intercambios, lo que se simplifica a $O(N^2)$ en términos de complejidad temporal.

O(2^N) - Tiempo exponencial

```

// Queremos calcular el enésimo número de Fibonacci usando un algoritmo recursivo
function fibonacci(n: number): number {

    // Verificamos si n es 0 o 1 como el caso base de la recursión porque la secuencia de Fibonacci com
    if (n <= 1) {
        return n;
    }

    // Llamamos recursivamente a la función fibonacci para calcular el enésimo número de Fibonacci
    return fibonacci(n - 1) + fibonacci(n - 2);
}

```

Aquí está por qué es $O(2^N)$:

- En cada llamada recursiva a la función fibonacci, se realizan dos llamadas recursivas adicionales con $n - 1$ y $n - 2$ como argumentos.
- Esto conduce a un crecimiento exponencial en el número de llamadas recursivas a medida que n aumenta.
- Cada nivel de recursión se bifurca en dos llamadas recursivas, lo que resulta en una estructura de llamadas recursivas similar a un árbol binario.
- El número de llamadas de función se duplica con cada nivel de recursión, lo que lleva a un total de 2^N llamadas de función al calcular el enésimo número de Fibonacci.

Por lo tanto, la complejidad temporal del algoritmo es $O(2^N)$ en el peor de los casos.

O(N!) - Tiempo factorial

```
// Dado el siguiente arreglo
const arr = [1, 2, 3];

// Queremos generar todas las permutaciones de un arreglo dado usando un algoritmo recursivo
function permute(arr: number[]): number[][] {
    // Caso base: si el arreglo tiene solo un elemento, devolverlo como una sola permutación
    if (arr.length <= 1) {
        return [arr];
    }

    // Inicializamos un arreglo vacío para almacenar las permutaciones
    const result: number[][] = [];

    // Iteramos sobre cada elemento en el arreglo
    for (let i = 0; i < arr.length; i++) {
        // Generamos todas las permutaciones del arreglo excluyendo el elemento actual
        const rest = arr.slice(0, i).concat(arr.slice(i + 1));
        const permutations = permute(rest);

        // Agregamos el elemento actual al principio de cada permutación
        for (const perm of permutations) {
            result.push([arr[i], ...perm]);
        }
    }
    return result;
}
```

Aquí está por qué es $O(N!)$:

- En cada llamada recursiva a la función permute, el algoritmo genera permutaciones seleccionando cada elemento del arreglo como el primer elemento y luego generando recursivamente permutaciones de los elementos restantes.

- El número de permutaciones crece factorialmente con el tamaño del arreglo de entrada.
- Para cada elemento en el arreglo, hay $(N-1)!$ permutaciones de los elementos restantes, donde N es el número de elementos en el arreglo.
- Por lo tanto, el número total de permutaciones es $N * (N-1) * (N-2) * \dots * 1$, que es factorial de N ($N!$).

Por lo tanto, la complejidad temporal del algoritmo es $O(N!)$ en el peor de los casos.

Complejidad en el Peor Caso, el Mejor Caso y el Caso Promedio

- La complejidad temporal en el peor de los casos representa el número máximo de pasos que tarda un algoritmo en completarse para un tamaño de entrada dado. Proporciona un límite superior sobre el rendimiento del algoritmo. Es la medida más comúnmente utilizada de la complejidad temporal en entrevistas de trabajo.
- La complejidad temporal en el mejor de los casos representa el número mínimo de pasos que tarda un algoritmo en completarse para un tamaño de entrada dado. Proporciona un límite inferior sobre el rendimiento del algoritmo. Es menos informativa que la complejidad en el peor de los casos y rara vez se usa en la práctica.
- La complejidad temporal en el caso promedio representa el número esperado de pasos que tarda un algoritmo en completarse para un tamaño de entrada dado, promediado sobre todas las entradas posibles. Proporciona una estimación más realista del rendimiento de un algoritmo que la complejidad en el peor de los casos. Sin embargo, calcular la complejidad en el caso promedio puede ser desafiantre y a menudo se evita a favor de la complejidad en el peor de los casos.

Complejidad espacial

La complejidad espacial de un algoritmo es una medida de la cantidad de memoria que requiere para ejecutarse en función del tamaño de la entrada. Se suele expresar en términos de la cantidad máxima de memoria utilizada por el algoritmo en cualquier momento durante su ejecución.

Es importante distinguir entre la complejidad temporal y la complejidad espacial, ya que un algoritmo con buena complejidad temporal puede tener una complejidad espacial deficiente, y viceversa. Por ejemplo, un algoritmo recursivo con complejidad temporal exponencial también puede tener complejidad espacial exponencial debido a las llamadas recursivas que consumen memoria.

Pero algo a tener en cuenta es que la complejidad espacial no es tan importante como la complejidad temporal, ya que la memoria suele ser más económica que la potencia de procesamiento y, en escenarios de la vida real, generalmente omitimos el análisis de complejidad espacial y nos enfocamos en la complejidad temporal.

Imagina que estás en un asado tradicional argentino. Tienes espacio limitado en la parrilla (similar a la memoria limitada en la informática), y quieres optimizar cuánta carne puedes cocinar a la vez.

Ahora, comparemos la carne con los datos en un algoritmo. Cuando estás cocinando, tienes que considerar cuánto espacio ocupa cada corte de carne en la parrilla. De manera similar, en la informática, los algoritmos

tienen que considerar cuánto espacio en memoria necesitan para almacenar y procesar datos.

Pero aquí está la cuestión: en un asado, lo más importante suele ser cuán rápido puedes cocinar la carne y servirla a tus invitados. De manera similar, en la informática, el tiempo que tarda un algoritmo en ejecutarse (complejidad temporal) suele ser el factor más crítico para el rendimiento.

Por lo tanto, si bien es esencial tener en cuenta cuánto espacio utiliza tu algoritmo, generalmente es más interesante centrarse en qué tan eficientemente puede resolver un problema en términos de tiempo.

Por supuesto, en algunas situaciones, como si estás cocinando en un balcón pequeño o cocinando para una multitud, el espacio se vuelve más importante. De manera similar, en informática, si estás trabajando con recursos de memoria limitados o en un dispositivo con restricciones estrictas de memoria, deberás prestar más atención a la complejidad espacial.

Pero en general, al igual que en un asado argentino, el equilibrio entre la complejidad temporal y espacial es clave para crear un resultado delicioso (o eficiente)!

Sin embargo, hablemos de cómo se calcula la complejidad espacial, o "cuánto espacio ocupas" en el caso de nuestra analogía de asado. Al igual que evaluarías cuánto espacio ocupa cada corte de carne en la parrilla, en informática, necesitas considerar cuánta memoria consume cada estructura de datos o variable en tu algoritmo.

Aquí hay un enfoque básico para calcular la complejidad espacial:

- **Identifica las Variables y Estructuras de Datos:** Observa el algoritmo e identifica todas las variables y estructuras de datos que utiliza. Estas podrían ser matrices, objetos u otros tipos de variables.
- **Determina el Espacio Utilizado por Cada Variable:** Para cada variable o estructura de datos, determina cuánto espacio ocupa en memoria. Por ejemplo, una matriz de enteros ocupará espacio proporcional al número de elementos multiplicado por el tamaño de cada entero.
- **Suma el Espacio:** Una vez que hayas determinado el espacio utilizado por cada variable, súmalos todos para obtener el espacio total utilizado por el algoritmo.
- **Considera el Espacio Auxiliar:** No olvides tener en cuenta cualquier espacio adicional utilizado por estructuras de datos auxiliares o llamadas de funciones. Por ejemplo, si tu algoritmo utiliza recursión, deberás considerar el espacio utilizado por la pila de llamadas.
- **Expresa la Complejidad Espacial:** Finalmente, expresa la complejidad espacial utilizando la notación Big O, al igual que haces con la complejidad temporal. Por ejemplo, si el espacio utilizado crece linealmente con el tamaño de la entrada, lo expresarías como $O(N)$. Si crece cuadráticamente, lo expresarías como $O(N^2)$, y así sucesivamente.

Entonces, al igual que gestionas cuidadosamente el espacio en tu parrilla para que quepa la mayor cantidad de carne posible sin amontonarla, en informática, quieres optimizar el uso de memoria para almacenar y procesar datos de manera eficiente. Y al igual que encontrar el equilibrio perfecto entre carne y espacio en un asado argentino, encontrar el equilibrio adecuado entre la complejidad temporal y espacial en tu algoritmo es clave para crear un resultado delicioso (o eficiente)!

¡Ejemplo de tiempo!

Utilicemos un algoritmo simple para encontrar la suma de elementos en una matriz como ejemplo para calcular la complejidad espacial.

```
function sumArray(arr: number[]): number {
    let sum = 0; // Espacio utilizado por la variable sum: O(1)

    for (let num of arr) { // Espacio utilizado por la variable de bucle: O(1)
        sum += num; // Espacio utilizado por la variable temporal: O(1)
    }

    return sum; // Espacio utilizado por el valor devuelto: O(1)
}
```

En este ejemplo:

- Tenemos una variable **sum** para almacenar la suma de los elementos, que ocupa una cantidad constante de espacio, denotada como $O(1)$.
- Tenemos una variable de bucle **num** que itera a través de cada elemento de la matriz. También ocupa una cantidad constante de espacio, $O(1)$.
- Dentro del bucle, tenemos una variable temporal para almacenar la suma de cada elemento con **sum**, que nuevamente ocupa una cantidad constante de espacio, $O(1)$.
- El valor devuelto de la función es la suma, que también ocupa una cantidad constante de espacio, $O(1)$.

Dado que cada variable y estructura de datos en este algoritmo ocupa una cantidad constante de espacio, la complejidad espacial total de este algoritmo es $O(1)$.

En resumen, la complejidad espacial de este algoritmo es constante, independientemente del tamaño de la matriz de entrada.

Ahora consideremos un ejemplo donde creamos una nueva matriz para almacenar la suma acumulativa de elementos de la matriz de entrada.

Aquí está el algoritmo:

```
function cumulativeSum(arr: number[]): number[] {
    const result = []; // Espacio utilizado por la matriz resultante: O(N), donde N es el tamaño de la
    let sum = 0; // Espacio utilizado por la variable sum: O(1)
    for (let num of arr) { // Espacio utilizado por la variable de bucle: O(1)
        sum += num; // Espacio utilizado por la variable temporal: O(1)
        result.push(sum); // Espacio utilizado por el nuevo elemento en la matriz resultante: O(1), p
    }
}
```

```
return result; // Espacio utilizado por el valor devuelto (la matriz resultante): O(N)
```

En este ejemplo:

- Tenemos una variable `result` para almacenar la suma acumulativa de elementos, que crece linealmente con el tamaño de la matriz de entrada `arr`. Cada elemento agregado a `result` contribuye a la complejidad espacial. Por lo tanto, el espacio utilizado por `result` es $O(N)$, donde N es el tamaño de la matriz de entrada.
- Tenemos una variable de bucle `num` que itera a través de cada elemento de la matriz de entrada `arr`, que ocupa una cantidad constante de espacio, $O(1)$.
- Dentro del bucle, tenemos una variable temporal `sum` para almacenar la suma acumulativa de elementos, que ocupa una cantidad constante de espacio, $O(1)$.
- Dentro del bucle, agregamos un nuevo elemento a la matriz `result` para cada elemento en la matriz de entrada. Cada operación de agregado agrega un elemento a la matriz, por lo que también contribuye a la complejidad espacial. Sin embargo, como se ejecuta N veces (donde N es el tamaño de la matriz de entrada), el espacio utilizado por las operaciones de agregado es $O(N)$.
- El valor devuelto de la función es la matriz `result`, que ocupa $O(N)$ espacio.

En general, la complejidad espacial de este algoritmo es $O(N)$, donde N es el tamaño de la matriz de entrada. Esto se debe a que el espacio utilizado por la matriz `result` crece linealmente con el tamaño de la entrada.

❖ Arreglos

Cuando hablamos de arreglos, generalmente pensamos en colecciones ordenadas de elementos, ¿verdad? Pero en JavaScript, los arreglos son en realidad objetos. Entonces, ¿qué es un arreglo real, podrías preguntar? Bueno, un verdadero arreglo es un bloque contiguo de memoria donde cada elemento ocupa la misma cantidad de espacio.

En un arreglo real, acceder a un elemento es súper rápido: es una operación de tiempo constante, lo que significa que lleva la misma cantidad de tiempo sin importar cuán grande sea el arreglo. ¿Por qué? Porque puedes calcular exactamente dónde está cada elemento en la memoria.

Ahora, contrastemos eso con los arreglos de JavaScript. Se implementan como objetos, donde los índices son las claves. Entonces, cuando accedes a un elemento en un arreglo de JavaScript, en realidad estás accediendo a una propiedad de un objeto. Esto significa que acceder a elementos no es tan rápido; es una operación de tiempo lineal porque el motor de JavaScript tiene que buscar a través de las claves del objeto para encontrar la correcta.

Para encontrar la ubicación en memoria de un elemento en un arreglo real, usas una fórmula simple:

```
const index = base_address + offset * size_of_element;
```

Aquí, el **índice** es lo que usualmente llamamos el índice, pero es más como un desplazamiento. La **base_address** es el punto de inicio del arreglo en la memoria, y **size_of_element** es, bueno, el tamaño de cada elemento.

Con esta fórmula, cada vez que buscas un elemento, estás realizando una operación de tiempo constante porque no importa cuán grande sea el arreglo, las matemáticas permanecen iguales.

Ahora, ilustremos esto con algo de código:

```
// Creamos un nuevo arreglo con 5 elementos
const a = [1, 2, 3, 4, 5];

// Calculemos el espacio total que ocupa el arreglo en memoria
const totalSpace = array.length * 4; // asumiendo que cada elemento ocupa 4 bytes

// Elijamos un índice
const index = 3;

// Calculemos la ubicación en memoria del elemento
const sizeOfEachElement = 4; // cada elemento ocupa 4 bytes
const baseAddress = array; // la referencia al propio arreglo
const offset = index * sizeOfEachElement;
const memoryLocation = baseAddress + offset;

// Accedamos al elemento en la ubicación en memoria calculada
const elementAtIndex = memoryLocation;

console.log("El valor en el índice", index, "es:", elementAtIndex); // El valor en el índice 3 es: 4
```

En este ejemplo, estamos simulando cómo funciona un arreglo real bajo el capó. Calculamos la ubicación en memoria de un elemento usando el índice, y luego accedemos a esa ubicación en memoria para obtener el elemento.

Ahora, visualicemos esto con Node.js, donde podemos echar un vistazo a lo que está sucediendo en la memoria:

```
// Creamos un nuevo búfer de arreglo en Node.js
const buffer = new ArrayBuffer(16); // 16 bytes de memoria

// Verifiquemos el tamaño del búfer (en bytes)
console.log("buffer.byteLength: " + buffer.byteLength); // Salida: buffer.byteLength: 16

// Ahora creamos una nueva Int8Array, un arreglo tipado de enteros con signo de 8 bits
const int8Array = new Int8Array(buffer);

// Registremos el contenido de la Int8Array (todos ceros inicialmente)
console.log(int8Array); // Salida: Int8Array(16) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

// Cada elemento en la Int8Array representa un solo byte en el búfer
```

```

// Ahora cambiemos el valor del primer elemento del arreglo a 1
int8Array[0] = 1;

// Registremos nuevamente el contenido de la Int8Array y del búfer
console.log(int8Array); // Salida: Int8Array(16) [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
console.log(buffer); // Observa cómo el búfer subyacente también se modifica (el primer byte se convierte)
// Salida: Búfer después del cambio: ArrayBuffer { [Uint8Contents]: <01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00> }

// Las TypedArrays proporcionan una vista diferente de la misma memoria

// Ahora creamos un arreglo de enteros con signo de 16 bits (usa 2 bytes por elemento)
const int16Array = new Int16Array(buffer);

// Registremos el contenido de la Int16Array (valores iniciales basados en el búfer)
console.log(int16Array); // Salida: Int16Array(8) [1, 0, 0, 0, 0, 0, 0, 0]

// La Int16Array tiene menos elementos porque usa más bytes por elemento, 2 bytes

// Nuevamente, cambiemos un valor (tercer elemento) y observemos los efectos
int16Array[2] = 4000;

// Registremos el contenido de los tres arreglos y el búfer
console.log(int16Array); // Salida: Int16Array(8) [1, 0, 4000, 0, 0, 0, 0, 0]
console.log(buffer); // Observa cómo se modifican múltiples bytes en el búfer (quinto y sexto bytes)
// Salida: ArrayBuffer { [Uint8Contents]: <01 00 00 00 a0 0f 00 00 00 00 00 00 00 00 00 00> } (los cambios afectan los bytes 5 y 6)
console.log(int8Array); // La vista de la Int8Array también se ve afectada (los quinto y sexto bytes cambian)
// Salida: Int8Array(16) [1, 0, 0, 0, -96, 15, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] (el quinto byte es -96, el sexto es 15)

// Ahora creamos un arreglo de enteros con signo de 32 bits (usa 4 bytes por elemento)
const int32Array = new Int32Array(buffer);

// Registremos el contenido de la Int32Array (valores iniciales basados en el búfer)
console.log(int32Array); // Salida: Int32Array(4) [1, 4000, 0, 0]

// Incluso menos elementos debido al mayor tamaño de cada elemento, 4 bytes

// Cambiemos el valor y observemos los efectos
int32Array[2] = 100000;

// Registremos el contenido de los tres arreglos y el búfer
console.log(int32Array); // Salida: Int32Array(4) [1, 4000, 100000, 0]
console.log(buffer); // ArrayBuffer { [Uint8Contents]: <01 00 00 00 a0 0f 00 00 10 27 00 00 00 00 00 00> }
console.log(int16Array); // Int16Array(8) [1, 0, 4000, 0, 100000, 0, 0, 0] (el cuarto elemento ahora es 100000)
console.log(int8Array); // Int8Array(16) [1, 0, 0, 0, -96, 15, 0, 0, 16, 39, 0, 0, 0, 0, 0, 0] (los nuevos valores aparecen en los bytes 4 y 5)

```

En este ejemplo de Node.js, estamos creando un búfer de 16 bytes, que es como un bloque de memoria. Luego, creamos diferentes arreglos tipados para ver esta memoria de manera diferente, como arreglos de

diferentes tipos de enteros. Modificar uno de estos arreglos también cambia el búfer subyacente, mostrando cómo son vistas diferentes de la misma memoria.

Ahora, volvamos al por qué estamos viendo decimales. Necesitaremos entender primero la representación de complemento a dos.

Entonces, el complemento a dos es como el truco mágico que usamos en la informática para manejar tanto números positivos como negativos usando el mismo sistema binario. Imagina esto: en nuestros números binarios, el primer dígito desde la izquierda (el jefe grande, ¿sabes?) decide si el número es positivo o negativo. Si es 0, es positivo, y si es 1, es negativo.

Ahora, para los números positivos, es pan comido. Solo los escribes en binario, como de costumbre. Por ejemplo, el número 3 en binario de 8 bits es 00000011. No hay problema allí, ¿verdad?

Pero cuando se trata de números negativos, necesitamos un truco. Tomamos la representación binaria del número positivo, invertimos todos los bits (los 0 se convierten en 1 y los 1 se convierten en 0), y luego sumamos 1 al resultado. Esto nos da el complemento a dos del número negativo. Digamos que queremos representar -3. Primero, empezamos con la representación binaria de 3, que es 00000011. Luego, invertimos todos los bits para obtener 11111100, y finalmente, sumamos 1 a este resultado para obtener 11111101. Ese es el complemento a dos de -3 en binario de 8 bits.

Ahora, ¿por qué hacemos todo esto? Bueno, es porque en informática nos gusta que las cosas estén ordenadas y sean consistentes. Con el complemento a dos, podemos usar las mismas reglas para sumar y restar tanto números positivos como negativos, sin necesidad de preocuparnos por casos especiales. Mantiene nuestra matemática bonita y limpia, como tomar mate en un día soleado en Buenos Aires.

Ahora podemos ver por qué estamos viendo decimales.

Cuando trabajas con 8 bits, los valores se representan en decimal, ya que el rango de un entero de 8 bits con signo es de -128 a 127. Si el valor se desborda, se envuelve dentro del rango:

Cuando lidias con 8 bits, los valores se muestran en decimal porque, ya sabes, un entero de 8 bits con signo solo puede contener números de -128 a 127. Es como tener un espacio limitado en un autobús lleno: solo puedes meter a tanta gente.

Ahora, imagina que estás tratando de meter el número 4000. En binario, eso es 111110100000. Pero aquí está la cosa: nuestro autobús solo llega hasta 127. Entonces, cuando intentas meter 4000 allí, es como intentar meter a un equipo de fútbol en un auto pequeño, simplemente no va a suceder.

Cuando intentas meter 4000 en nuestro autobús de 8 bits, se desborda. Pero en lugar de causar caos, hace algo bastante genial: se envuelve. Ves, la ruta del autobús comienza de nuevo desde el principio, como un bucle interminable.

Este "envolver" es donde las cosas se ponen interesantes. El bit más a la izquierda, el gran jefe del número, cambia su signo. Entonces, en lugar de ser positivo, se vuelve negativo. Es como dar la vuelta al autobús yendo en la dirección opuesta.

Ahora, usamos nuestro truco anterior llamado complemento a dos para averiguar el nuevo número. Primero, invertimos todos los bits de **111110100000** para obtener **000001011111**. Luego, sumamos 1 a este número invertido, dándonos **000001100000**.

¡Y voilà! Ese número binario **000001100000** representa -96 en decimal. Entonces, aunque intentaste meter 4000, nuestro pequeño autobús lo maneja como un campeón y te dice que es -96. ¡Esa es la magia del desbordamiento y el envolvimiento en un mundo de 8 bits!

~~ ¿Cómo pensar?

Cuando estás hasta el cuello de código, entender los conceptos y aplicarlos para resolver esos problemas es la clave. ¿Mi consejo? Echa el resto comentando tu código, explicando cada paso dentro del método y DESPUÉS láñate de lleno a la implementación.

Así sabrás qué hacer y solo necesitarás descubrir cómo hacerlo.

Ejemplo:

```
function sumArray(arr: number[]): number {
    // Inicializa la variable de suma para almacenar la suma de elementos
    let sum = 0; // O(1)

    // Itera sobre cada elemento en el array
    for (let num of arr) { // O(N)
        // Suma el elemento actual a la suma
        sum += num; // O(1)
    }

    // Devuelve la suma final
    return sum; // O(1)
}
```

~~ Búsqueda Lineal

Es el pan de cada día de los algoritmos, pero ¿realmente sabes cómo se las arregla para lucirse?

Aquí está la jugada: estamos bailando a través de cada elemento de una colección, preguntando si el elemento que estamos buscando es el que tenemos frente a nosotros. ¿El gran jefe de JavaScript que usa este algoritmo? ¡El método `indexOf`, pibe!

```
function indexOf(arr: number[], target: number): number {
    // Itera sobre cada elemento en el array
    for (let i = 0; i < arr.length; i++) {
```

```

    // Comprueba si el elemento actual es igual al objetivo
    if (arr[i] === target) {

        // Si lo es, devuelve el índice del elemento
        return i;
    }

}

// Si el objetivo no aparece por ningún lado, devuelve -1
return -1;
}

```

Entonces... ¿cuál es el peor escenario aquí que nos dará la Notación Big O? Que volvamos con las manos vacías o que el elemento que buscamos esté paseando al final del array, haciendo que este algoritmo sea O(N). A medida que N crece, también lo hace la complejidad, es como ver tu cintura después de devorar una montaña de alfajores.

❖ Búsqueda Binaria

Ahora, este es el rey de reyes, el mejor de los mejores y uno de los titanes en entrevistas profesionales y desafíos de programación. Veamos cuándo y cómo soltar su poder:

¿El truco? ¡Dividir y conquistar, che! En lugar de jugar al escondite con cada elemento del array, cortamos esa cosa en dos y preguntamos, "¿Eh, estás a la izquierda o a la derecha?" Luego, seguimos cortando hasta que atrapemos a nuestro esquivo objetivo.

```

function binarySearch(arr: number[], target: number): number {
    // Inicializa los punteros izquierdo y derecho
    // izquierdo comienza al principio del array
    // derecho comienza al final del array
    let left = 0;
    let right = arr.length - 1;

    // Continúa avanzando mientras el puntero izquierdo sea menor o igual al puntero derecho
    while (left <= right) {
        // Calcula el índice medio del espacio de búsqueda actual
        const mid = Math.floor((left + right) / 2);

        // Comprueba si el elemento medio es el objetivo
        if (arr[mid] === target) {

            // Si lo es, devuelve el índice del elemento
            return mid;
        } else if (arr[mid] < target) {

            // Si el elemento medio es menor que el objetivo, ve hacia la derecha
            left = mid + 1;
        } else {

```

```

        // Si el elemento medio es mayor que el objetivo, ve hacia la izquierda
        right = mid - 1;
    }
}
}

```

Y con este ejemplo verás, claro como una luz que nos guía en las noches más oscuras, que no estoy mintiendo:

```

// necesitamos encontrar el índice del número objetivo dentro de un array de 1024 elementos

1024 / 2 = 512 // lo dividimos a la mitad y vemos que el número objetivo está en la mitad derecha
512 / 2 = 256 // lo dividimos a la mitad nuevamente y vemos que el número objetivo está en la mitad derecha
256 / 2 = 128 // lo dividimos a la mitad nuevamente y vemos que el número objetivo está en la mitad derecha
128 / 2 = 64 // lo dividimos a la mitad nuevamente y vemos que el número objetivo está en la mitad derecha
64 / 2 = 32 // lo dividimos a la mitad nuevamente y vemos que el número objetivo está en la mitad derecha
32 / 2 = 16 // lo dividimos a la mitad nuevamente y vemos que el número objetivo está en la mitad derecha
16 / 2 = 8 // lo dividimos a la mitad nuevamente y vemos que el número objetivo está en la mitad derecha
8 / 2 = 4 // lo dividimos a la mitad nuevamente y vemos que el número objetivo está en la mitad derecha
4 / 2 = 2 // lo dividimos a la mitad nuevamente y vemos que el número objetivo está en la mitad derecha
2 / 2 = 1 // lo dividimos a la mitad nuevamente y vemos que el número objetivo está en la mitad derecha
1 / 2 = 0.5 // ya no podemos dividirlo más, así que nos detenemos

// aquí viene la magia, si contamos el número de pasos tenemos un total de 10 pasos para encontrar el número
// ¿sabes cuál es el logaritmo de 1024 en base 2? ¡es 10!
log2(1024) = 10

```

Ejercicio de la bola de cristal

El problema de la bola cristal es un problema matemático clásico que se toma en entrevistas de programación. Se trata de encontrar la posición en la que dos bolas cristales se encuentran y se rompen al caer de una determinada altura. Lo importante aquí no es saber que son bolas de cristal, de que caen de una altura o cualquier otro detalle, tenemos que entender las bases y generalizar para que funcione en cualquier caso. Si realizamos esta tarea, podremos decir que la altura en verdad es un arreglo de n elementos, siendo 0 el inicio de la caída, n la altura máxima y index el lugar donde chocan las bolas.

```

// f = no se encuentran, t = se encuentran o se han encontrado previamente
const altura = [f, f, f, f, f, f, t, t, t, t, t];
//           inicio          choque          fin de la caída

```

Solución

Veamos las herramientas que ya disponemos de para resolver este problema:

- **Arrays:** Arrays son una estructura de datos que contiene un conjunto de elementos de un mismo tipo. En este caso, la altura de las bolas.

- **Loop**: Loop es una estructura de control que permite repetir una sección de código un número de veces. En este caso, el número de veces que se deben repetir es el número de elementos en el arreglo.
- **If**: If es una estructura de control que permite ejecutar una sección de código si una condición es verdadera. En este caso, la condición es si el elemento en el arreglo es verdadero o falso.
- **Linear Search**: Linear Search es una técnica de búsqueda en el que se busca un elemento en un arreglo mediante la búsqueda lineal, buscando el elemento en cada posición del arreglo y así sucesivamente.
- **Binary Search**: Binary Search es una técnica de búsqueda en el que se busca un elemento en un arreglo mediante la búsqueda binaria, particionando el arreglo en dos partes y buscando el elemento en la parte que más se adecúe a lo que estamos buscando y así sucesivamente.

```
// Búsqueda lineal
const altura = [f, f, f, f, f, f, t, t, t, t, t];

function findCrystalBall(altura: number[]): number {
  for (let i = 0; i < altura.length; i++) {
    if (altura[i] === true) {
      return i;
    }
  }

  return -1;
}
```

El problema con esta solución es que no se puede optimizar para que funcione en cualquier caso, ya que la búsqueda lineal busca en cada posición del arreglo y así sucesivamente, lo que significa que si el arreglo tiene muchos elementos, el tiempo de ejecución será muy grande.

Así que vamos a buscar una solución más eficiente.

```
// Búsqueda binaria
function findCrystalBall(altura: number[]): number {
  let inicio = 0;
  let fin = altura.length - 1;

  while (inicio <= fin) {
    const index = Math.floor((inicio + fin) / 2);

    if (altura[index] === true) {
      return index;
    } else if (altura[index] === false) {
      inicio = index + 1;
    } else {
      fin = index - 1;
    }
  }
}
```

```
    return -1;
```

Funciona ? la verdad que no ! porque no estamos encontrando el momento justo en el que las bolas se encuentran, sino que estamos retornando el primer true que encontramos en vez del primero de todos. Además de esto, si sumamos la condición de que una vez que una de las bolas se rompa, ya no podrá volver a utilizarse, el tiempo de ejecución se mantendrá linear con el tamaño del arreglo. Ya que el peor de los casos haremos $1/2$ de la altura y si encontramos un true, tendremos que volver a la posición anterior y buscar cual es el punto de quiebre lo cual sigue siendo $O()$.

Así que vamos a combinar todo lo que aprendimos hasta ahora para implementar una solución increíble y mucho más eficiente:

1- Vamos a reducir el tamaño del corte para que en vez de la mitad del arreglo, sea la raíz de N , y de esta manera se reduce considerablemente el tiempo de ejecución.

2- Una vez que encontramos el primer true, vamos a volver al punto donde empieza la sección de corte $SQRT(N)$ y buscamos el primer true que se encuentre de manera LINEAR.

```
function findCrystalBall(altura: number[]): number {
    const N = altura.length;
    const SQRT = Math.floor(Math.sqrt(N));

    // ahora vamos a reducir el tamaño del corte para que en vez de la mitad del arreglo, sea la raíz de
    // y lo recorremos de manera LINEAR, raíz de  $N$  cada vez
    let i = SQRT;

    for (; i < N; i += SQRT) {
        if (altura[i] === true) {
            break;
        }
    }

    i -= SQRT;

    for (let j = 0; j < SQRT && i < N; i++, j++) {
        if (altura[i] === true) {
            return i;
        }
    }

    return -1;
}

// Búsqueda binaria
function findCrystalBall(altura: number[]): number {
    let inicio = 0;
    let fin = altura.length - 1;

    while (inicio <= fin) {
        const index = Math.floor((inicio + fin) / 2);
```

```
// comprueba si el valor en la posición media es 'true' y
// si es el primer 'true' del arreglo (el elemento anterior debe ser 'false' si index no es 0)
if (altura[index] === true)
    if (index === 0 || altura[index - 1] === false) return index;
    else fin = index - 1;
else inicio = index + 1;
}

return -1;
}
```

Aunque las dos soluciones funcionan y parecen efectivas, la búsqueda lineal es más eficiente !

- $N = 1000$ pasos
- $\text{SQRT}(N) = 100$ pasos
- $\log_2(N) = 10$ pasos

$[0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1]$

0

$N = 16$

$i = \text{sqrt}(N) = 4$

$\text{start} = 0 \quad \text{end} = 15$

1- $[0, 0, 0, 0, 0] \times i = 4$

1- $[0, 0, 0, 0, 0, 0]$

$\text{start} \leq \text{end}$

2- $[0, 0, 1, 1, 1] \checkmark \quad i = 8$

true? \times

$j = 0 \quad i = 4 = 4 \quad j < \text{sqrt}$
 $[0, 0, 1, 1, 1]$

2- $\text{start} = 4 \checkmark \quad \text{start} = 15 \checkmark$

1- $0 = \text{true? } \times \quad i < N$

$[0, 0, 1, 1, 1, 1]$

2- $j = 1 \times i = 5 \times$

true? \checkmark

$0 = \text{true? } \times$

$\text{index} \equiv 0 \times$

3- $j = 2 \times i = 6 \checkmark$

$\text{altura}[\text{index} - 1] = \text{false? } \times$

$0 = \text{true? } \times$

3- $\text{start} = 4 \checkmark \quad \text{end} = 7 \checkmark$

$[0, 0, 1]$

true? \times

4- $j = 3 \quad i = 7$

4- $\text{start} = 7 \checkmark \quad \text{end} = 7 \checkmark$

$[1]$

true? \checkmark

$| i = 7 |$

❖ Bubble Sort

Sorting es una de las cosas que más hacemos en el día a día como programadores, PERO, mucha gente no sabe lo que realmente pasa y depende de las famosas funciones mágicas que son parte de todos los lenguajes de programación.

Aunque no lo crean, hacer sorting es una de las cosas más fáciles que podemos hacer ! y lo mejor es que es también uno de los algoritmos más cortos a escribir.

La idea principal del sorting por el método de burbujeo es que se recorrerá un arreglo, comparando cada elemento con el próximo, si el elemento es mayor que el próximo, se intercambiarán lugares. Lo importante a saber es que al final de la primera iteración, el elemento más grande se encontrará en el final del arreglo. Esto nos permite volver a iterar el arreglo para seguir ordenando el resto de elementos pero de una manera más eficiente, ya que no debemos incluir el último elemento del resultado de la anterior iteración y así se irá reduciendo la cantidad de elementos ante cada pasada hasta que el arreglo sea ordenado.

```
const unsortedArray = [3, 1, 4, 8, 2];

// primera iteración
// comparamos el primero con el segundo y los cambiamos
const unsortedArray = [1, 3, 4, 8, 2];
// comparamos el segundo con el tercero y los dejamos
const unsortedArray = [1, 3, 4, 8, 2];
// comparamos el tercero con el cuarto y los dejamos
const unsortedArray = [1, 3, 4, 8, 2];
// comparamos el cuarto con el quinto y los cambiamos
const unsortedArray = [1, 3, 4, 2, 8]; // queda el 8 al final, el número más grande

// segunda iteración donde excluimos el 8 y volvemos a iterar

// comparamos el primero con el segundo y los dejamos
const unsortedArray = [1, 3, 4, 2];
// comparamos el segundo con el tercero y los dejamos
const unsortedArray = [1, 3, 4, 2];
// comparamos el tercero con el cuarto y los cambiamos
const unsortedArray = [1, 3, 2, 4]; // el 4 queda al final, el número más grande

// tercera iteración donde excluimos el 4 y volvemos a iterar

// comparamos el primero con el segundo y los dejamos
const unsortedArray = [1, 3, 2];
// comparamos el segundo con el tercero y los cambiamos
const unsortedArray = [1, 2, 3]; // el 3 queda al final, el número más grande

// cuarta iteración donde excluimos el 3 y volvemos a iterar

// comparamos el primero con el segundo y los dejamos
const unsortedArray = [1, 2]; // FIN
```

Si nos fijamos podemos descubrir un patrón encubierto en el código, ante cada iteración el resultado es $N - i$, donde N es el tamaño del arreglo y i es el número de iteraciones que hemos realizado hasta ahora.

```
const unsortedArray = [3, 1, 4, 8, 2];

// primera iteración N - i = 4
// resultado de la primera iteración = [1, 3, 4, 2]

// segunda iteración N - i = 3
```

```
// resultado de la segunda iteración = [1, 3, 2]

// tercera iteración N - i = 2
// resultado de la tercera iteración = [1, 2]
```

Si generalizamos el patrón, podemos decir que el resultado final sería $N - N + 1$ ya que la forma generalizada de calcular la sumatoria de elementos de un arreglo es: $((N + 1) * N) / 2$

```
Sumatoria entre 1 y 10 es 11
Sumatoria entre 2 y 9 es 11
Sumatoria entre 3 y 8 es 11
Sumatoria entre 4 y 7 es 11
.
.
.
Sumatoria entre 5 y 6 es 11
Por lo que podemos decir que N siendo 5 y N + 1 siendo 6 el resultado final sería
(6 + 1) * 5 / 2 = 11
```

Ahora si generalizamos el patrón dentro de la Big O, podemos decir que:

```
1- N (N + 1) / 2
2- N (N + 1) // eliminamos constantes elevando todo a 2
3- N^2 + N
// en Big O removemos valores insignificantes, si hablamos de números MUY grandes,
// + N es un MUY chico en comparación
4- N^2

Quedando O (N^2) en Big O
```

Código

```
function bubbleSort(array: number[]): number[] {
    // definimos el tamaño del arreglo
    const N = array.length;

    // iteramos sobre el arreglo
    for (let i = 0; i < N; i++) {
        // iteramos sobre el arreglo
        for (let j = 0; j < N - 1 - i; j++) {
            if (array[j] > array[j + 1]) {
                // si es mayor, cambiamos el elemento
                const temp = array[j];

                // cambiamos el elemento con el siguiente
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}
```

```
}

return array;
}
```

Gentleman del Código: Dominando la Clean Architecture

♪ Introducción al libro "Gentleman del Código: Dominando la Clean Architecture"

¡Hola, genios del código! Soy Gentleman, y les traigo un recorrido apasionante por el universo de la arquitectura de software y las metodologías ágiles, utilizando el famoso patrón de Clean Architecture. Este libro es un compilado basado en mis discursos y charlas, especialmente diseñado para guiarlos desde la conceptualización hasta la implementación práctica de sistemas escalables y fáciles de mantener. ¡Prepárense para convertirse en Gentleman del código!

♪ Capítulo 1: "¡Descubriendo la Clean Architecture!"

Para los que recién se asoman a este mundo, les cuento que esto no es puro cuento chino, ¡es una de las mejores cosas que le pudo pasar al desarrollo de software! Así que, pongan primera, que arrancamos con todo.

¿Qué es la Clean Architecture?

Primero lo primero: ¿Qué onda con la Clean Architecture? Bueno, muchachos, esto no es un jueguito de Lego, aunque se le parezca en que todo encaja perfecto. Clean Architecture es un patrón arquitectónico, y lo que la hace especial es que te ayuda a organizar tu proyecto de una manera que, por más que pase el tiempo, lo puedas mantener y expandir sin querer tirarte de los pelos. Es decir, no es un simple patrón de diseño, es una filosofía completa que te ayuda a separar lo que es código de negocio de infraestructura o de la UI, haciendo todo más modular y testeable.

De dónde sale y para dónde vamos

Esta joyita no salió de la nada. Fue popularizada por el tío Bob Martin, que es algo así como el Messi del desarrollo de software. El tipo este la pensó como una forma de evitar que los proyectos se conviertan en un monstruo que nadie quiere tocar. Con la Clean Architecture, cada pieza del sistema tiene su lugar y su responsabilidad claramente definidas, haciendo que incluso los proyectos más grandes sean fáciles de manejar.

¿Y para dónde vamos con esto? Hacia el futuro, ¡claro está! Implementar la Clean Architecture en tus proyectos significa que estás pensando en grande, en sistemas que no solo funcionen bien hoy, sino que se puedan adaptar y expandir en el futuro sin dramas.

¿Quién soy y dónde estoy?

Bueno, para los que no me conocen, soy Gentleman, un apasionado del código limpio y las buenas prácticas en el desarrollo de software. Estoy aquí para guiarlos en este camino de descubrimiento y para asegurarme de que al final de este libro no solo entiendan qué es la Clean Architecture, sino que puedan implementarla y defenderla como verdaderos gurús del código.

Conclusión del capítulo

Entonces, chicos, lo que vamos a estar haciendo a lo largo de este libro es aplicar esta clínica de la

arquitectura limpia. Ya verán que es un concepto hermoso y super útil. No solo es diferente a lo que muchos piensan, sino que realmente es una solución arquitectónica que les va a cambiar la forma de construir software. Prepárense, porque esto recién empieza y les prometo que va a ser un viaje increíble.

En el próximo capítulo, vamos a hincarle el diente a la "Separación de preocupaciones", que es el corazón de la Clean Architecture. Vamos a desmenuzar cómo esta práctica no solo es fundamental para mantener nuestro código limpio y ordenado, sino que también es la clave para hacer nuestros proyectos escalables y fáciles de mantener. Así que no se despeguen, ¡que esto se pone bueno!

❖ Capítulo 2: "Separación de preocupaciones: La clave de una arquitectura eficiente"

En este capítulo, vamos a profundizar en uno de los principios más fundamentales y poderosos: la Separación de Preocupaciones. Así que, ajusten sus cinturones porque vamos a desglosar cómo este concepto puede hacer que sus proyectos sean una belleza en cuanto a mantenimiento y escalabilidad.

¿Qué es la Separación de Preocupaciones?

Primero que nada, pongámonos serios y definamos qué es esto. La Separación de Preocupaciones no es más que una estrategia pro para organizar nuestro código. Imaginen que están organizando una fiesta y tienen que mantener separados a los fans de Boca y River... bueno, algo así es lo que hacemos aquí, pero con nuestro código. Esta técnica nos ayuda a mantener cada parte de nuestro proyecto enfocada en una tarea específica, sin meterse en los asuntos de las demás. Esto, mis amigos, es la esencia de construir software que no solo es fácil de entender, sino también de modificar y expandir.

¿Cuál es el objetivo de aplicarla en Clean Architecture?

En Clean Architecture, la Separación de Preocupaciones es como el as de espadas en la manga de un mago. Nos permite estructurar el proyecto en capas, donde cada una tiene una responsabilidad clara. Por ejemplo, tenemos una capa para la lógica de negocio, otra para la interfaz de usuario, y otra más para el acceso a datos. Esto hace que si mañana queremos cambiar la base de datos o la interfaz de usuario, podemos hacerlo sin que el resto del sistema sufra un colapso nervioso.

Las Ventajas: Mantenibilidad y Escalabilidad

Ahora, hablemos de las ventajas. Implementar la Separación de Preocupaciones en nuestras arquitecturas no solo hace que el código sea más bonito estéticamente (que también cuenta, ¡eh!), sino que hace maravillas por la mantenibilidad. Cuando cada parte del sistema se ocupa solo de lo suyo, actualizar o arreglar bugs se vuelve mucho más sencillo. Y no solo eso, sino que también prepara el terreno para que el sistema crezca sin dramas. A medida que añadimos más funcionalidades, cada una se integra en su respectiva capa sin alterar las demás, lo que hace que el sistema sea super escalable.

Recomendaciones Prácticas

Para cerrar, les voy a dejar algunas recomendaciones de oro. Si están implementando Clean Architecture, no se olviden de definir bien las interfaces entre las capas. Esto es como decir, "vos, ocupate de esto y no te metas en aquello". Mantengan estas interfaces limpias y claras, y se van a ahorrar un montón de dolores de cabeza. Además, no tengan miedo de reorganizar las capas si con el tiempo ven que algo podría mejorar. La flexibilidad es clave en este juego.

Conclusión del capítulo

Bueno, espero que ahora tengan una idea clara de por qué la Separación de Preocupaciones es tan crucial en la Clean Architecture. En el próximo capítulo, vamos a comparar los patrones de diseño con las arquitecturas, para que vean cómo todo encaja en este gran puzzle del desarrollo de software. Así que no se pierdan el próximo episodio, porque vamos a seguir desmenuzando estos temas para que se conviertan en verdaderos maestros del código. ¡Nos vemos en el próximo capítulo, cracks!

~~ Capítulo 3: "Patrones de diseño vs. Arquitecturas: Entendiendo las diferencias"

Sigamos profundizando en este apasionante mundo de la programación. En este capítulo, vamos a meterle pata a la diferencia entre patrones de diseño y arquitecturas de software. Mucha gente confunde estos dos términos, pero hoy vamos a dejar todo más claro que el agua. ¡Así que vamos al ruedo!

¿Qué son los patrones de diseño?

Empecemos por lo básico: los patrones de diseño son soluciones probadas a problemas comunes que nos encontramos en el desarrollo de software. Piensen en ellos como las recetas de cocina de la abuela, cada una con su secreto para resolver un problema específico en la cocina. En programación, por ejemplo, tenemos el patrón Factory, que nos ayuda a crear objetos sin especificar la clase exacta del objeto que se creará. Esto es puro oro cuando queremos flexibilidad en nuestro sistema.

¿Y qué es una arquitectura de software?

Por otro lado, cuando hablamos de arquitectura de software, estamos hablando del plan maestro, la estructura completa sobre la cual construimos nuestras aplicaciones. La arquitectura define cómo se organiza el sistema, cómo interactúan los componentes, y cómo se gestionan los datos y el control del flujo en la aplicación. Es, digamos, el plano de la casa en la que vamos a meter esos muebles (los patrones de diseño).

Las diferencias clave

La diferencia fundamental entre un patrón de diseño y una arquitectura es el alcance. Mientras que un patrón de diseño se aplica a un problema específico dentro de una parte del sistema, la arquitectura se extiende a lo largo de todo el proyecto. Es como comparar cambiar el tapizado de un sillón (patrón de diseño) con diseñar toda la sala de estar (arquitectura).

Integración de patrones en la arquitectura

En la Clean Architecture, los patrones de diseño juegan un rol crucial, pero siempre dentro del marco que la arquitectura define. Usamos patrones como herramientas que nos ayudan a implementar la separación de preocupaciones y a asegurar que cada parte del sistema se pueda mantener y escalar de manera eficiente. Los patrones nos dan las tácticas, pero la arquitectura nos da la estrategia.

Consejos prácticos

Mi consejo aquí es simple: aprendan y dominen varios patrones de diseño, pero siempre tengan en mente cómo estos se ajustan dentro de la arquitectura general de su aplicación. No se dejen llevar por la tentación de aplicar un patrón solo porque sí; cada patrón tiene su lugar y su momento, y saber cuándo y cómo usarlo es lo que separa a los buenos programadores de los verdaderos maestros.

Conclusión del capítulo

Espero que ahora tengan una mejor comprensión de lo que separa a los patrones de diseño de las

arquitecturas de software y cómo ambos se complementan para crear aplicaciones robustas y mantenibles. En nuestro próximo capítulo, vamos a explorar más a fondo cómo la Clean Architecture facilita la mantenibilidad y escalabilidad de nuestros proyectos. ¡Así que no se lo pierdan, porque vamos a seguir desgranando estos conceptos para que ustedes se conviertan en los próximos arquitectos estrella del mundo del software! ¡Nos vemos en el próximo capítulo, genios del código!

~~ Capítulo 4: "Mantenibilidad y Escalabilidad con Clean Architecture"

Vamos a sumergirnos en dos de las joyas máspreciadas cuando hablamos de arquitecturas de software: la mantenibilidad y la escalabilidad. Estos son los pilares que hacen que la Clean Architecture brille con luz propia. Vamos a explorar cómo esta arquitectura nos ayuda a lograr un código que no solo es un placer mantener sino también fácil de escalar. ¡Arrancamos!

La Mantenibilidad en Clean Architecture

Imaginémonos que tenemos un auto donde cada parte está tan enredada con las demás que cambiar una bujía requiere desmontar medio motor. Suena como una pesadilla, ¿no? Bueno, eso es lo que Clean Architecture busca evitar en el desarrollo de software. Al mantener una separación clara y definida de las responsabilidades, cada componente del sistema puede ser entendido, probado y modificado de manera independiente. Esto es crucial porque cuando llega el momento de arreglar bugs o añadir mejoras, podemos ir directo al grano sin temor a romper otras partes del sistema.

¿Cómo se traduce esto a la práctica?

Para ponerlo en práctica, pensemos en un ejemplo simple. Supongamos que tenemos una aplicación con una interfaz de usuario, una lógica de negocio y una base de datos. En una arquitectura bien diseñada, podríamos cambiar la interfaz de usuario sin tocar la lógica de negocio o la base de datos. Esto no solo reduce el riesgo de bugs, sino que también hace que el desarrollo sea más rápido y menos propenso a errores.

Escalabilidad con Clean Architecture

Ahora, hablemos de escalabilidad. En el mundo del software, escalar no solo significa manejar más datos o usuarios, sino también añadir nuevas funcionalidades sin colapsar bajo el propio peso del sistema. Clean Architecture nos facilita esto al permitirnos agregar nuevas características como si fueran módulos o bloques independientes. Cada bloque se comunica con los demás a través de interfaces bien definidas, lo que significa que podemos expandir nuestra aplicación sin tener que reescribir o ajustar excesivamente lo que ya funciona.

Un ejemplo concreto de escalabilidad

Pongamos un caso práctico. Si decidimos que queremos añadir un sistema de pagos a nuestra aplicación, en una arquitectura limpia, este se integraría a través de una interfaz de servicios, sin alterar directamente la lógica de negocio existente. Esto nos permite no solo integrar el nuevo módulo de forma eficiente, sino también probarlo y modificarlo sin impactar el resto de la aplicación.

Conclusión del capítulo

Entender y aplicar la mantenibilidad y la escalabilidad con Clean Architecture puede parecer un desafío al principio, pero les aseguro que las recompensas valen la pena. Un sistema bien diseñado no es solo más fácil de mantener y escalar, sino también más robusto y adaptable a los cambios, lo que en el mundo del software, amigos míos, es oro puro.

En nuestro próximo capítulo, vamos a profundizar en cómo la arquitectura de plugins puede aumentar aún más la flexibilidad y adaptabilidad de nuestros sistemas. Así que no se pierdan este próximo capítulo porque vamos a seguir construyendo sobre esta base sólida para convertirlos en verdaderos arquitectos del software. ¡Nos vemos allí, maestros del código!

~ Capítulo 5: "Plugins en la arquitectura: Flexibilidad y Adaptabilidad"

En este capítulo, vamos a adentrarnos en un concepto super clave: la arquitectura de plugins. Este enfoque nos da una flexibilidad y adaptabilidad tremendas en nuestros proyectos. Preparados, listos, ¡ya!

¿Qué es la Arquitectura de Plugins?

Imaginemos que nuestro software es un equipo de fútbol. En un equipo, cada jugador tiene su posición y función específica, pero podemos hacer cambios según lo necesite el partido. Un delantero puede salir y entrar otro para dar frescura al ataque. Bueno, algo similar pasa en la arquitectura de plugins: tratamos partes de nuestro sistema como componentes o módulos que se pueden "enchufar" y "desenchufar" fácilmente, sin alterar el funcionamiento global del sistema.

Flexibilidad y Adaptabilidad

Este enfoque es una maravilla cuando el proyecto necesita adaptarse rápidamente a nuevas condiciones o requerimientos. Supongamos que queremos añadir una nueva función analítica o modificar el proceso de pago. En lugar de reescribir grandes porciones de código, simplemente "enchufamos" el nuevo módulo que maneja esta función. Si más adelante necesitamos actualizarlo o reemplazarlo, podemos hacerlo sin afectar el resto de la aplicación.

Implementación Práctica de Plugins

Para implementar esta arquitectura, cada módulo o plugin debe definirse claramente con interfaces estandarizadas. Esto significa que cada parte sabe cómo comunicarse con las demás sin necesidad de saber exactamente qué hay dentro de cada una. Es como decir, "Hey, necesito que hagas esto, no me importa cómo lo hagas, solo hazlo". Esta separación ayuda a mantener nuestro código ordenado, testeable y, sobre todo, fácil de cambiar.

Ventajas de Usar Plugins

Una de las mayores ventajas de utilizar la arquitectura de plugins es la posibilidad de probar nuevos features o servicios de forma aislada. Podemos desarrollar, probar, y desplegar nuevos componentes de forma independiente, lo que acelera los ciclos de desarrollo y reduce el riesgo asociado con cambios en sistemas complejos.

Un Ejemplo Real

Pensemos en una aplicación que gestiona reservas en un hotel. Si queremos añadir una funcionalidad para gestionar eventos especiales, podemos desarrollar un plugin específico para eso. Este módulo se encargará de todo lo relacionado con los eventos, desde la reserva hasta la gestión de invitados, y se integrará con el sistema existente a través de una interfaz definida.

Conclusión del capítulo

La arquitectura de plugins es un ejemplo brillante de cómo la Clean Architecture promueve la adaptabilidad y la escalabilidad. Nos permite mantener sistemas complejos y en constante evolución sin perder la cabeza en

el proceso. En el próximo capítulo, vamos a hablar sobre las desventajas y consideraciones temporales de la Clean Architecture. Así que, ¡no se lo pierdan porque vamos a poner las cartas sobre la mesa y hablar claro sobre cuándo y cómo utilizar esta arquitectura a nuestro favor!

Nos vemos en el próximo capítulo, ¡sigamos adelante, equipo!

~~ Capítulo 6: "Desventajas y Consideraciones Temporales de Clean Architecture"

Como todo en la vida, incluso las mejores prácticas tienen sus desventajas y limitaciones. En este capítulo, vamos a ser brutales y honestos sobre las desventajas de la Clean Architecture y las consideraciones temporales que implica su implementación. Así que, afilá los lápices, que vamos a anotar los pros y contras de esta poderosa herramienta.

Las Desventajas de la Estructuración Rigurosa

Si bien la separación clara y definida de responsabilidades en Clean Architecture ofrece muchos beneficios, también puede ser su talón de Aquiles. Esta estructura requiere una planificación y una disciplina rigurosas, lo que puede resultar en un aumento del tiempo de desarrollo inicial. Sí, amigos, no todo lo que brilla es oro. Esta rigurosidad puede ser vista como una desventaja, especialmente en proyectos con plazos ajustados o donde la velocidad de entrega es crucial.

Verbosidad y Complejidad

Otra posible desventaja es la verbosidad. Clean Architecture puede llevar a la creación de muchos archivos y directorios, lo que puede complicar el proyecto más de lo necesario, especialmente para quienes son nuevos en este enfoque. A veces, uno puede terminar sintiendo que está escribiendo más código de 'andamiaje' que código que realmente aporta al negocio.

Consideraciones Temporales: ¿Cuándo usar Clean Architecture?

No todo proyecto necesita una bazúca para matar una mosca. Clean Architecture es ideal para aplicaciones grandes y complejas que requerirán mantenimiento a largo plazo y donde la escalabilidad es un factor crítico. Sin embargo, si estás trabajando en un prototipo rápido o un proyecto pequeño con un ciclo de vida corto, implementar Clean Architecture podría ser un exceso. En estos casos, la simplicidad debería ser la prioridad.

El Costo del Tiempo

El tiempo es dinero, y esto nunca ha sido más cierto que en el desarrollo de software. Adoptar Clean Architecture significa invertir tiempo en aprender y aplicar sus principios correctamente. Esto puede ser un desafío para los equipos bajo presión para entregar rápidamente. Además, cualquier cambio arquitectónico importante, como pasar de una arquitectura monolítica a una basada en Clean, requerirá un esfuerzo significativo y posiblemente una curva de aprendizaje empinada.

Conclusión del capítulo

Entonces, ¿vale la pena? Absolutamente, pero solo si el zapato le calza al pie. Clean Architecture no es una solución universal, pero cuando se utiliza en el contexto adecuado, puede transformar un proyecto caótico en un sistema bien engrasado que es un placer mantener y escalar.

Así cerramos nuestro libro, mis queridos desarrolladores. Espero que hayan disfrutado este viaje tanto como yo y que estén listos para aplicar estos conocimientos en sus propios proyectos. Recuerden, la mejor

arquitectura es aquella que sirve a las necesidades del proyecto y del equipo. ¡Sigan programando, sigan aprendiendo y, sobre todo, sigan disfrutando el proceso!

Nos vemos en el próximo código, ¡Gentleman fuera!

❖ Capítulo 7: "Estructura de la Clean Architecture: Una Visión en Capas"

En este capítulo adicional, vamos a profundizar en la estructura de la Clean Architecture y cómo esta se desglosa en capas distintas. Esta arquitectura, desarrollada por Robert C. Martin, también conocido como Uncle Bob, se basa en principios de diseño que buscan promover la separación de intereses y la independencia de la infraestructura. Vamos a desmenuzar cada capa para que entiendas cómo cada una contribuye a crear un sistema de software robusto, mantenible y escalable.

1. Capa de Entidades (Entities): La capa de entidades o "Entities" representa el corazón de la aplicación, donde se define la lógica de negocio y las reglas que son fundamentales para el funcionamiento del sistema. Estas entidades son agnósticas respecto al resto del sistema y deben ser independientes de cualquier framework o base de datos que se utilice. Incluyen objetos de dominio cuya responsabilidad es contener datos y ejecutar las reglas del negocio que son críticas y centrales para la aplicación.

2. Capa de Casos de Uso (Use Cases): La capa de casos de uso, o "Use Cases", encapsula y implementa todas las acciones que un usuario puede realizar con el sistema. Es aquí donde se modelan los procesos de negocio específicos de la aplicación. Los casos de uso orquestan el flujo de datos hacia y desde las entidades, y dirigen esos datos hacia las capas exteriores. Esta capa no debe saber nada sobre cómo se presentan los datos al usuario final o cómo se almacenan los datos; su único objetivo es manejar la lógica de aplicación requerida para ejecutar cada caso de uso.

3. Capa de Adaptadores (Interface Adapters): La capa de adaptadores, también conocida como "Interface Adapters", actúa como un puente entre los casos de uso internos y el mundo exterior. Esta capa convierte los datos de la forma más conveniente para los casos de uso y entidades, a la forma más conveniente para algún agente externo como una base de datos o un navegador web. Por ejemplo, un controlador MVC en una aplicación web se ubicaría aquí, tomando datos del usuario, convirtiéndolos a un formato que los casos de uso puedan entender, y luego pasando estos datos a un caso de uso apropiado.

4. Capa de Frameworks y Controladores (Frameworks & Drivers): La capa externa, o "Frameworks & Drivers", es donde todos los detalles acerca de la implementación con frameworks específicos o bases de datos se manejan. Esta capa es completamente independiente del negocio y se encarga de cosas como la base de datos, la interfaz de usuario y cualquier framework que se utilice. El objetivo de esta capa es minimizar la cantidad de código que tienes que cambiar si decides implementar otra base de datos, otro framework o incluso una interfaz de usuario diferente.

Conclusión del capítulo: Entender la estructura en capas de la Clean Architecture es crucial para diseñar una aplicación que sea fácil de mantener, testear y expandir. Al mantener estas capas bien definidas y separadas, los desarrolladores pueden asegurar que cambios en una área del software (como la interfaz de usuario o la base de datos) tengan el mínimo impacto posible en las otras áreas, especialmente en la lógica de negocio central.

Con esta estructura, estamos mejor equipados para enfrentar los desafíos del desarrollo de software moderno, proporcionando sistemas que no solo satisfacen las necesidades actuales sino que también son capaces de adaptarse a los cambios futuros con facilidad.

❖ Capítulo 8: "Casos de Uso y Dominio en Clean Architecture: Diferencia entre Lógica de Negocios y Aplicación"

En este nuevo capítulo vamos a explorar las diferencias entre casos de uso y dominio dentro de la Clean Architecture. Es crucial entender estos conceptos para diseñar aplicaciones robustas y bien estructuradas. Así que ajusten esos cerebros, ¡porque arrancamos con todo!

¿Qué es el Dominio y por qué es tan central?

El dominio en Clean Architecture es el núcleo, el corazón de nuestra aplicación. ¿Por qué? Porque aquí reside toda la lógica de negocio que define cómo funciona nuestro proyecto. Todo lo demás en nuestro sistema puede cambiar, pero el dominio es sagrado, es la esencia que no puede ser alterada por capas externas. Es intocable y todas las demás capas dependen de él.

La Regla del Dominio

Existe una regla de oro en Clean Architecture conocida como "la regla del dominio", que establece que el dominio es autónomo y no debe depender de nada más allá de sí mismo. Todo en nuestro sistema gira en torno al dominio; él no se ajusta a las otras capas, sino que son las otras capas las que deben adaptarse y servir al dominio.

Casos de Uso: La conexión entre el exterior y el dominio

Mientras que el dominio se enfoca en las reglas de negocio, los casos de uso son los encargados de implementar esa lógica en escenarios específicos de la aplicación. Los casos de uso saben lo que sucede en el dominio, pero no al revés. Actúan como intermediarios entre el mundo exterior y las reglas de negocio puras contenidas en el dominio.

¿Cómo se comunica la información?

En Clean Architecture, la información fluye desde el centro (el dominio) hacia fuera. Las capas externas, como los adaptadores o la interfaz de usuario, pueden interactuar con los casos de uso, pero siempre respetando la autonomía del dominio. Esta estructura asegura que las dependencias estén bien organizadas y que la lógica de negocio central permanezca protegida y clara.

El Error Común en la División de Responsabilidades

Un punto crítico que a menudo confunde incluso a los desarrolladores experimentados es cómo dividir correctamente las responsabilidades entre el dominio y los casos de uso. Un ejemplo clásico es la validación de reglas de negocio: estas deben residir en el dominio si son fundamentales para la esencia de la aplicación, sin importar la tecnología o la plataforma utilizada.

Ejemplo Práctico: Aplicación Bancaria

Consideremos una aplicación bancaria donde se establece que para abrir una cuenta, el cliente debe ser mayor de 18 años. Esta es una regla de negocio y, por lo tanto, pertenece al dominio porque define un requisito fundamental de la aplicación, no influenciado por consideraciones técnicas externas.

Conclusión del capítulo

Dominar la distinción entre el dominio y los casos de uso es esencial para cualquier desarrollador que aspire a construir sistemas robustos y mantenibles utilizando Clean Architecture. Al mantener esta clara separación, aseguramos que nuestra aplicación no solo sea funcional sino también adaptable a largo plazo.

En el próximo capítulo, profundizaremos aún más en cómo implementar estas teorías en ejemplos prácticos y código, asegurándonos de que estas estructuras teóricas se traduzcan en aplicaciones reales y eficientes. ¡No se lo pierdan, vamos que se pone mejor!

~ Capítulo 9: "Implementación Práctica de Casos de Uso y Dominio en Clean Architecture"

Vamos a meter de lleno en cómo implementar efectivamente los casos de uso y el dominio en nuestros proyectos. Vamos a transformar la teoría en práctica, y para eso, ¡vamos a ver algo de código!

Estructura Básica de un Caso de Uso

Los casos de uso son estructuras que definen cómo se lleva a cabo una operación específica en el sistema, guiados por las reglas del dominio. Por ejemplo, si tenemos una aplicación bancaria, un caso de uso podría ser "Crear Cuenta", que implementaría la validación de edad que mencionamos antes. Veamos cómo se podría ver esto en código:

```
class CrearCuentaUseCase {  
    constructor(private cuentaRepository: CuentaRepository) {}  
  
    execute(datosCuenta: DatosCuenta) {  
        if (datosCuenta.edad < 18) {  
            throw new Error("El cliente debe ser mayor de 18 años para abrir una cuenta.");  
        }  
        const cuenta = new Cuenta(datosCuenta);  
        this.cuentaRepository.save(cuenta);  
    }  
}
```

Integración del Dominio

El dominio, por su parte, se encargaría de las entidades y las reglas de negocio centrales. En nuestro ejemplo, la entidad sería la **Cuenta**, y una regla de negocio sería que el cliente debe ser mayor de edad. Esto podría implementarse en la clase **Cuenta** así:

```
class Cuenta {  
    constructor(private datos: DatosCuenta) {}  
    if (datos.edad < 18) {  
        throw new Error("El cliente debe ser mayor de 18 años para abrir una cuenta.");  
    }  
}
```

Diferenciación entre Capas

Es vital diferenciar claramente entre las capas de dominio y de casos de uso. El dominio contiene la lógica que es fundamental para el negocio y no depende de la capa de aplicación o infraestructura. Los casos de uso interactúan con estas entidades del dominio y orquestan cómo se ejecutan las operaciones en respuesta a las acciones del usuario.

Manejo de la Interfaz de Usuario

La interfaz de usuario, o la capa más externa, debería ser totalmente agnóstica respecto al dominio y los casos de uso. Su única función es presentar datos al usuario y enviar las acciones del usuario a la capa de aplicación. Por ejemplo, un botón en una interfaz gráfica que permite al usuario abrir una nueva cuenta simplemente invocaría el caso de uso `CrearCuentaUseCase`.

Conclusión del capítulo

La clave para implementar correctamente la Clean Architecture es asegurarse de que cada parte del sistema haga solo lo que le corresponde, y nada más. Esto no solo simplifica el mantenimiento y la expansión del sistema, sino que también facilita la prueba y la depuración de cada componente de manera aislada.

En el próximo capítulo, exploraremos cómo las reglas de negocio y los casos de uso influyen en la elección de tecnologías y herramientas para el desarrollo de nuestra aplicación. Así que manténganse en sintonía, ¡porque vamos a seguir profundizando en cómo hacer que nuestras aplicaciones sean más limpias, eficientes y, sobre todo, magníficas!

~~ Capítulo 10: "Tecnologías y Herramientas: Alineación con Casos de Uso y Dominio en Clean Architecture"

Ahora que tenemos una sólida comprensión de los casos de uso y el dominio, es hora de hablar sobre cómo elegir las tecnologías y herramientas que mejor se alinean con nuestra arquitectura. Preparados, que empezamos con todo.

El Rol de las Tecnologías en Clean Architecture

En Clean Architecture, la elección de tecnologías debe ser cuidadosa y estratégica. Lo fundamental aquí es que la tecnología sirva a la arquitectura, y no al revés. Esto significa que las decisiones tecnológicas deben apoyar y no complicar nuestra estructura de dominio y casos de uso.

Independencia Tecnológica del Dominio

El dominio debe ser completamente agnóstico respecto a la tecnología. Esto es crucial porque las reglas de negocio no deben estar contaminadas por limitaciones o especificidades tecnológicas. Por ejemplo, si decidimos cambiar nuestra base de datos de SQL a NoSQL, esto no debería afectar la lógica del dominio de nuestra aplicación.

Selección de Tecnologías para Casos de Uso

Los casos de uso, aunque más cercanos a la tecnología que el dominio, aún deben mantenerse lo suficientemente flexibles para adaptarse a cambios. Al seleccionar herramientas para implementar los casos de uso, consideremos aquellas que ofrecen la mayor flexibilidad y facilidad de integración. Por ejemplo, en un backend podríamos optar por frameworks como Express.js o Spring Boot, que son robustos pero ofrecen la flexibilidad necesaria para adaptarse a diferentes casos de uso sin imponer restricciones severas.

Ejemplo Práctico: Implementando la Interfaz de Usuario

Consideremos la interfaz de usuario en un sistema que utiliza Clean Architecture. Si bien la interfaz puede ser construida con cualquier framework de frontend como React, Angular o Vue, es esencial que estas tecnologías se utilicen de manera que respeten la separación del dominio y los casos de uso. Esto se logra a través de adaptadores o puentes que comunican la UI con los casos de uso, manteniendo la independencia del dominio.

```
// Ejemplo de adaptador en React para el caso de uso CrearCuenta
function CrearCuentaComponent({ crearCuentaUseCase }) {
  const [edad, setEdad] = useState('');

  const handleCrearCuenta = async () => {
    try {
      await crearCuentaUseCase.execute({ edad });
      alert('Cuenta creada exitosamente!');
    } catch (error) {
      alert(error.message);
    }
  };

  return (
    <div>
      <input
        type="number"
        value={edad}
        onChange={e => setEdad(e.target.value)}
        placeholder="Ingrese su edad"
      />
      <button onClick={handleCrearCuenta}>Crear Cuenta</button>
    </div>
  );
}
```

Conclusión del capítulo

La elección de tecnologías en Clean Architecture no es solo una cuestión de preferencia personal o tendencia de mercado, sino una decisión estratégica que debe alinearse con la estructura y principios de nuestra arquitectura. Asegúémonos de que nuestras herramientas y tecnologías fortalezcan nuestros casos de uso y dominio, no que los compliquen.

En el próximo capítulo, vamos a abordar la gestión y mantenimiento de estos sistemas a largo plazo, considerando cómo la Clean Architecture facilita estos procesos. Así que no se despeguen, ¡porque el aprendizaje continúa y lo mejor siempre está por venir!

~~ Capítulo 11: "Mantenimiento y Gestión a Largo Plazo en Clean Architecture"

¡Hola de nuevo, compañeros del código! Aquí Gentleman, y hoy nos adentramos en uno de los temas más cruciales en el desarrollo de software: el mantenimiento y la gestión a largo plazo de nuestras aplicaciones

construidas con Clean Architecture. Preparémonos para entender cómo esta metodología no solo ayuda a construir sistemas robustos sino también a mantenerlos eficientemente a lo largo del tiempo.

¿Por Qué es Importante el Mantenimiento en Clean Architecture?

Clean Architecture, por su diseño, facilita enormemente el mantenimiento de software. Gracias a la clara separación entre el dominio y los casos de uso, y entre estos y las capas externas, cada parte del sistema puede ser actualizada, mejorada, o incluso reemplazada sin que esto cause un efecto dominó en las otras partes. Esta independencia es vital para la sustentabilidad a largo plazo de cualquier aplicación.

Estrategias para un Mantenimiento Efectivo

- **Pruebas Automatizadas:** Implementar un sólido conjunto de pruebas automatizadas es fundamental. Estas pruebas deben cubrir tanto los casos de uso como las entidades del dominio. Al mantener el dominio aislado de cambios externos y tecnológicos, podemos asegurarnos de que las pruebas sean estables y confiables a lo largo del tiempo.
- **Documentación Clara:** Mantener una documentación detallada y actualizada de cada capa, especialmente del dominio y los casos de uso, ayuda a nuevos desarrolladores y equipos a entender rápidamente la lógica y estructura del sistema, facilitando el mantenimiento y la escalabilidad.
- **Refactorización Continua:** La refactorización no es solo para corregir errores, sino para mejorar continuamente la estructura del código conforme el software evoluciona y crece. Clean Architecture hace que la refactorización sea más manejable y menos riesgosa, dado que las dependencias son controladas y claras.

Gestión de Dependencias y Actualizaciones

Uno de los grandes beneficios de Clean Architecture es cómo maneja las dependencias. Al tener capas bien definidas y separadas, actualizar una librería o cambiar una herramienta en una capa específica (como la capa de infraestructura o la interfaz de usuario) no impacta el dominio ni los casos de uso. Esto reduce significativamente el riesgo durante las actualizaciones y minimiza el downtime del sistema.

Ejemplo Práctico: Actualización de la Capa de Persistencia

Supongamos que decidimos cambiar nuestra base de datos de MySQL a PostgreSQL. En una arquitectura limpia, esta transición afectaría principalmente a la capa de infraestructura, donde los adaptadores de la base de datos residen. El dominio y los casos de uso, que contienen la lógica de negocio, permanecerían intactos, asegurando que la funcionalidad principal de la aplicación no se vea comprometida.

```
// Adaptador de base de datos antes de la actualización
class MySQLCuentaRepository implements CuentaRepository {
    save(cuenta: Cuenta) {
        // Lógica para guardar la cuenta en MySQL
    }
}

// Adaptador de base de datos después de la actualización
class PostgreSQLCuentaRepository implements CuentaRepository {
    save(cuenta: Cuenta) {
        // Lógica para guardar la cuenta en PostgreSQL
    }
}
```

Conclusión del capítulo

La mantenibilidad y la gestión efectiva de las aplicaciones a largo plazo son quizás los mayores beneficios que ofrece Clean Architecture. Al adherirse a sus principios, podemos garantizar que nuestros sistemas no solo sean robustos en el lanzamiento, sino que continúen siéndolo a medida que escalan y evolucionan.

En nuestro próximo capítulo, discutiremos cómo liderar equipos y proyectos utilizando Clean Architecture, asegurando que la integridad del diseño se mantenga desde la concepción hasta la implementación final. ¡Estén atentos, que aún hay mucho más por aprender!

~~ Capítulo 12: "Diferencias entre Lógica de Aplicación, Lógica de Dominio y Lógica de Empresa en Clean Architecture"

En este capítulo, profundizaremos en las diferencias clave entre la lógica de aplicación, la lógica de dominio y la lógica de empresa, tres conceptos fundamentales en la arquitectura de software que pueden confundirse fácilmente pero que son críticos para el diseño y mantenimiento de sistemas robustos y eficientes.

Lógica de Dominio: El Corazón de la Aplicación

La lógica de dominio es el núcleo central de cualquier sistema en Clean Architecture. Contiene las reglas y procesos de negocio esenciales que definen cómo opera la empresa a nivel más fundamental. Esta lógica es independiente de la plataforma y la tecnología utilizada para implementar la aplicación. Su principal característica es que es puramente sobre el "qué" y el "por qué" de las operaciones, no sobre el "cómo".

Por ejemplo, si consideramos un sistema bancario, la regla de que "los clientes deben ser mayores de 18 años para abrir una cuenta" es parte de la lógica de dominio.

Lógica de Aplicación: Orquestando el Flujo de Trabajo

La lógica de aplicación se refiere al "cómo" se llevan a cabo las operaciones definidas en la lógica de dominio dentro de una aplicación específica. Actúa como el mediador entre la interfaz de usuario y la lógica de dominio, gestionando el flujo de datos y asegurando que las operaciones se ejecuten en el orden correcto. Esta capa también maneja la lógica necesaria para interactuar con la base de datos y otros servicios externos.

Continuando con el ejemplo del sistema bancario, la lógica de aplicación decidiría cuándo y cómo se verifica la edad del cliente y qué hacer si el cliente no cumple con esta regla.

Lógica de Empresa: Reglas a Nivel Organizacional

La lógica de empresa abarca las reglas y políticas que no están limitadas a una aplicación específica sino que son comunes a varias aplicaciones o a toda la organización. Estas reglas pueden incluir políticas de cumplimiento, regulaciones de seguridad y otros estándares operativos que afectan a múltiples sistemas dentro de la empresa.

Por ejemplo, en nuestra analogía del sistema bancario, una regla de empresa podría ser que todas las transacciones financieras deben ser auditadas anualmente, lo cual es una política que afectaría a todas las aplicaciones de gestión financiera de la organización.

Implementación de las Capas de Lógica en Clean Architecture

En la práctica, la separación de estas tres lógicas permite a los desarrolladores y diseñadores de sistemas asegurar que cada parte del software se ocupe de aspectos específicos y bien definidos del negocio. Esto no solo clarifica el desarrollo y el mantenimiento, sino que también facilita la escalabilidad y la adaptabilidad del sistema a largo plazo.

```
// Ejemplo de cómo podrían interactuar estas lógicas en código

// Lógica de Dominio
class Cuenta {
    constructor(private edad: number) {
        if (!this.esMayorDeEdad()) {
            throw new Error("Debe ser mayor de 18 años para abrir una cuenta.");
        }
    }

    private esMayorDeEdad() {
        return this.edad >= 18;
    }
}

// Lógica de Aplicación
class CrearCuentaService {
    constructor(private cuentaRepo: CuentaRepository) {}

    crearCuenta(datosCliente: { edad: number }) {
        const cuenta = new Cuenta(datosCliente.edad);
        this.cuentaRepo.guardar(cuenta);
    }
}
```

En resumen, la diferenciación clara entre la lógica de aplicación, la lógica de dominio y la lógica de empresa es esencial para la construcción de sistemas informáticos que sean tanto eficientes en su ejecución como en su mantenimiento. Al entender y aplicar adecuadamente estas separaciones, los equipos pueden desarrollar software que no solo satisface los requisitos actuales sino que también es robusto frente a los cambios futuros.

» Capítulo 13: "Adaptadores: Rompiendo Esquemas en Clean Architecture"

En este capítulo, nos metemos de lleno en una de las capas más revolucionarias y dinámicas de la Clean Architecture: los adaptadores. Vamos a desglosar cómo esta capa, que puede parecer complicada por su naturaleza dual, es en realidad el puente vital entre el núcleo interno de nuestra aplicación y el caótico mundo exterior. ¡Acompáñame en este viaje para entender mejor su rol disruptivo!

Los Adaptadores: Una Doble Función Vital Los adaptadores en la Clean Architecture tienen una tarea compleja y crítica: comunican la aplicación con el mundo exterior y viceversa. Son comparables a las

membranas celulares en biología, que controlan qué sustancias pueden entrar y salir de la célula, asegurando su supervivencia y funcionamiento óptimo.

- **Ingreso de Datos:** Cuando la información llega del exterior, como datos de un endpoint en el frontend, el adaptador la transforma para que se ajuste a las necesidades de los casos de uso internos. Por ejemplo, puede recibir información en un formato no ideal y debe mapearla a un formato que la lógica de negocio pueda procesar eficientemente.
- **Salida de Datos:** De forma similar, cuando los casos de uso generan datos que necesitan ser enviados al exterior, el adaptador también maneja esta transformación, asegurando que los datos sean comprensibles y utilizables para otros sistemas o usuarios finales.

```
// Ejemplo de adaptador en acción
class UserInfoAdapter {
    convertToInternalFormat(externalData: { name: string; lastName: string }) {
        // Transformación de datos externos a formato interno
        return {
            firstName: externalData.name,
            lastName: externalData.lastName
        };
    }

    convertToExternalFormat(internalData: { firstName: string; lastName: string }) {
        // Preparar datos internos para enviar al exterior
        return {
            name: internalData.firstName,
            lastName: internalData.lastName
        };
    }
}
```

Por Qué los Adaptadores Son Clave Estos componentes no solo facilitan la interacción entre diferentes sistemas y formatos, sino que también protegen la integridad de la lógica interna. Al aislar los cambios y las peculiaridades del mundo exterior en una capa específica, los adaptadores permiten que el núcleo de la aplicación permanezca limpio y enfocado en la lógica de negocio, sin ser afectado por las variabilidades externas.

Conclusión del capítulo Entender y implementar efectivamente los adaptadores en Clean Architecture es crucial para el desarrollo de sistemas robustos y mantenibles que interactúan con un entorno complejo y en constante cambio. En el próximo capítulo, vamos a explorar ejemplos reales de cómo los adaptadores han permitido a las aplicaciones adaptarse rápidamente a nuevas demandas y tecnologías sin grandes sobresaltos, manteniendo la lógica de negocio intacta y funcionando sin problemas. ¡Seguí conmigo en esta aventura para ver cómo esta teoría se aplica en la práctica!

~~ Capítulo 14: "La Capa Externa en Clean Architecture: Conectando Tu Aplicación con el Mundo Exterior"

En este capítulo, vamos a desmenuzar la capa externa en Clean Architecture, una capa crucial que nos conecta con el mundo exterior. A menudo esta capa es subestimada, pero hoy, ¡vamos a darle el protagonismo que se merece!

Importancia de la Capa Externa La capa externa en Clean Architecture incluye todo lo que interactúa con el entorno exterior de nuestra aplicación, como interfaces de usuario, bases de datos, y APIs externas. Es fundamental entender que, aunque es la capa más cercana al usuario o al servicio externo, su diseño y funcionamiento deben ser cuidadosamente gestionados para mantener la integridad de nuestra arquitectura interna.

Componentes de la Capa Externa

- **Frontend:** Ya sea que estés trabajando con React, Vue, Angular o cualquier otro framework, el frontend se considera parte de la capa externa. Su tarea principal es presentar información al usuario y capturar sus interacciones, que luego se comunican a las capas internas a través de adaptadores.
- **Backend:** Servicios que procesan la lógica de negocios y manejan las solicitudes del cliente también residen aquí, actuando como el puente entre el frontend y el dominio de la aplicación.
- **Base de Datos:** Si bien muchos podrían pensar que las bases de datos son el núcleo de una aplicación, en Clean Architecture, son simplemente otro detalle de implementación, gestionadas a través de adaptadores para mantener la independencia del dominio.

Detalles de Implementación Aunque nos pasamos días y noches aprendiendo y discutiendo sobre qué tecnología usar, al final del día, todas estas son simplemente detalles de implementación. Lo que realmente importa es cómo estos detalles se integran y contribuyen a las reglas de negocio centrales sin afectarlas directamente.

```
// Ejemplo de un adaptador para una base de datos MongoDB
class MongoDBUserAdapter implements UserDatabaseAdapter {
    constructor(private db: DatabaseConnection) {}

    async fetchUser(userId: string) {
        const userDocument = await this.db.collection('users').findOne({ id: userId });
        return new User(userDocument.name, userDocument.email);
    }

    async saveUser(user: User) {
        await this.db.collection('users').insertOne({
            id: user.id,
            name: user.name,
            email: user.email
        });
    }
}
```

El Poder de los Adaptadores La magia real ocurre con los adaptadores, que permiten que elementos de la capa externa como bases de datos y APIs cambien sin tener un impacto devastador en la lógica de negocio.

Cambiar de MongoDB a DynamoDB, por ejemplo, simplemente requeriría ajustes en el adaptador correspondiente, no una reescritura completa de la lógica de negocios o casos de uso.

Conclusión del capítulo Entender y aplicar correctamente la capa externa en Clean Architecture nos permite aprovechar al máximo las tecnologías que elegimos sin comprometer la integridad de nuestra aplicación. A través de los adaptadores, podemos garantizar que nuestra aplicación sea tan flexible como necesitamos que sea, sin estar casados con ninguna tecnología específica.

En el próximo capítulo, exploraremos cómo estas integraciones afectan el rendimiento y la seguridad de nuestras aplicaciones, asegurando que no solo son funcionales sino también robustas y seguras. ¡Así que no se pierdan este próximo viaje, porque seguimos enriqueciendo nuestro conocimiento juntos!

» Capítulo 15: "Rendimiento y Seguridad en la Capa Externa: Optimizando la Interacción con el Mundo Exterior"

Ahora que hemos explorado la estructura y la función de la capa externa en Clean Architecture, es crucial abordar cómo estas interacciones afectan el rendimiento y la seguridad de nuestras aplicaciones. Este capítulo se enfoca en optimizar estos aspectos críticos para asegurarnos de que no solo nuestra arquitectura sea sólida sino también rápida y segura.

Optimización del Rendimiento en la Capa Externa El rendimiento es clave cuando se trata de la experiencia del usuario y la eficiencia operativa. En la capa externa, donde nuestra aplicación se encuentra con el mundo, cada milisegundo cuenta. Aquí es donde los adaptadores juegan un rol crucial, no solo en la transformación de datos sino también en asegurar que estas transformaciones sean eficientes.

- **Caching:** Implementar estrategias de caché en adaptadores puede reducir significativamente la latencia y la carga en nuestros servidores. Por ejemplo, caching los resultados de las consultas comunes a las bases de datos puede evitar operaciones costosas repetidas veces.
- **Asincronía:** Utilizar operaciones asincrónicas para manejar las solicitudes puede mejorar enormemente el rendimiento al no bloquear el procesamiento mientras esperamos respuestas del servidor o de la base de datos.

Garantizando la Seguridad en la Capa Externa La seguridad es otro aspecto vital, especialmente cuando nuestra aplicación interactúa con el mundo exterior. La capa externa es a menudo el primer punto de contacto para ataques, por lo que es imperativo fortalecerla.

- **Validación y Sanitización de Datos:** Es fundamental validar y sanitizar todos los datos que entran a través de la capa externa para proteger nuestra aplicación de entradas maliciosas. Los adaptadores deben asegurarse de que todos los datos entrantes se ajusten a las expectativas antes de pasarlo a las capas internas.
- **Manejo de Errores:** Implementar un manejo de errores robusto en los adaptadores puede prevenir la propagación de errores que podrían exponer vulnerabilidades o información sensible.

Ejemplo Práctico: Mejorando el Rendimiento y la Seguridad Consideremos un adaptador que interactúa con una API externa. Este adaptador no solo debe mapear los datos para los casos de uso internos, sino también optimizar el acceso y asegurar que los datos sean seguros.

```
class ExternalAPIAdapter {  
    constructor(private apiClient: APIClient) {}  
  
    async fetchSecureData(userId: string) {  
        const userData = await this.apiClient.get(`/users/${userId}`);  
        if (!userData) {  
            throw new Error('User not found');  
        }  
        return this.sanitize(userData);  
    }  
  
    private sanitize(data: any) {  
        // Remove any unwanted or dangerous fields  
        delete data.internalId;  
        return data;  
    }  
}
```

Conclusión del capítulo Optimizar el rendimiento y garantizar la seguridad en la capa externa son aspectos esenciales que requieren atención cuidadosa y continua. Al aplicar las estrategias adecuadas y utilizar adaptadores efectivamente, podemos asegurar que nuestra aplicación no solo funcione sin problemas sino que también sea segura frente a amenazas externas.

~~ Conclusiones del Libro: "Gentleman del Código: Dominando la Clean Architecture"

Al llegar al final de este viaje a través de la Clean Architecture, hemos explorado cada capa, desglosado cada componente y descubierto cómo cada parte contribuye al éxito de una aplicación moderna y escalable. Pero, ¿qué hemos aprendido y cómo podemos aplicar estos conocimientos en nuestros proyectos? Aquí te dejo las conclusiones clave para que te lleves y comiences a aplicar desde hoy.

1. Claridad en la Separación de Concerns: La Clean Architecture nos enseña la importancia de mantener una clara separación entre las diferentes preocupaciones de nuestras aplicaciones. Esto no solo facilita la mantenibilidad y la escalabilidad sino que también permite que equipos de diferentes disciplinas colaboren efectivamente sin pisarse los pies.

2. La Independencia del Dominio: El corazón de nuestra aplicación, el dominio, debe permanecer puro y libre de influencias externas. Esta independencia nos asegura que las reglas de negocio se mantienen consistentes y protegidas, independientemente de los cambios en las tecnologías periféricas o en las interfaces de usuario.

3. Adaptadores y Capas Externas: Los adaptadores y la capa externa actúan como los guardianes de nuestro sistema, protegiendo la lógica de negocio de las impurezas y variabilidades del mundo exterior.

Aprender a implementar correctamente estos componentes es esencial para construir sistemas que puedan evolucionar sin constantes refactorizaciones internas.

4. Flexibilidad y Adaptabilidad: Una de las grandes lecciones de la Clean Architecture es su enfoque en la flexibilidad. Al aislar las dependencias y permitir que los detalles de implementación residan en la periferia, nuestras aplicaciones pueden adaptarse rápidamente a nuevas tecnologías o requerimientos del negocio sin grandes revisiones del núcleo del sistema.

5. Mantenimiento y Evolución: Finalmente, hemos visto cómo la Clean Architecture facilita el mantenimiento y la evolución de las aplicaciones a lo largo del tiempo. Las pruebas automatizadas, la documentación clara y las estrategias de refactorización continua son solo algunas de las prácticas que ayudan a que los sistemas construidos con esta arquitectura perduren y se adapten sin desmoronarse bajo el peso de su propio éxito.

Conclusión General: Como líderes y desarrolladores, nuestro objetivo es construir software que no solo cumpla con los requisitos actuales sino que también se adapte y responda a los desafíos futuros. La Clean Architecture ofrece un marco robusto y probado para lograrlo. Al entender y aplicar sus principios, nos posicionamos para liderar proyectos que no solo resuelven problemas técnicos sino que también impulsan el éxito del negocio y la innovación.

Este libro no marca el final de tu aprendizaje, sino que, espero, sea el comienzo de una nueva forma de pensar y construir software. Con estas herramientas en tu arsenal, estás más que equipado para enfrentar los desafíos del desarrollo moderno. ¡Adelante, Gentleman, y a transformar el mundo del software con cada línea de código que escribas!

Aplicación de Clean Architecture en el Front End

¡Buenas, gente! Hoy vamos a charlar sobre cómo llevar la Clean Architecture al front end, pero con un giro especial. Vamos a ver cómo podemos hacerlo de una manera más orgánica y flexible, sin estructuras de carpetas demasiado rígidas, y cómo la regla del alcance juega un papel fundamental en todo esto. ¡Acompáñenme en este recorrido por un enfoque más natural y adaptable!

~~ Conceptos Clave de Clean Architecture

La idea detrás de la Clean Architecture es desacoplar el software de las tecnologías de UI, bases de datos, y cualquier otro elemento externo, concentrándonos en la lógica de negocio pura. Esto se logra mediante capas que separan responsabilidades claramente:

- **Dominio:** Aquí definimos nuestras entidades y reglas de negocio que son completamente independientes de la interfaz de usuario y tecnologías externas. Son los conceptos fundamentales sobre los que se construye la aplicación.
- **Casos de Uso:** Se encargan de implementar la lógica de negocio necesaria para cumplir con los requisitos funcionales del sistema. Operan sobre el modelo de dominio y utilizan adaptadores para comunicarse con la capa de infraestructura.
- **Adaptadores:** Conectan los casos de uso con el mundo externo, ya sea presentando datos al usuario o comunicándose con una base de datos.
- **Frameworks y Drivers:** Esta es la capa más externa, donde interactuamos directamente con frameworks específicos y bibliotecas.

Ejemplo

Para el ejemplo de una aplicación bancaria donde el requerimiento de negocio es que solo se pueden registrar personas mayores de 18 años, vamos a diseñar una estructura siguiendo los principios de la Clean Architecture. Esta estructura asegurará que las reglas de negocio, como la restricción de edad, estén claramente definidas y desacopladas de la interfaz de usuario y otros componentes externos.

Estructura Propuesta para la Aplicación Bancaria

Imaginemos cómo podríamos organizar nuestro proyecto en las capas sugeridas por la Clean Architecture:

Dominio (Entities Models y Business Rules)

Aquí se define el modelo de User que incluye atributos como nombre, fecha de nacimiento, dirección, etc. Además, en esta capa residirán las reglas de negocio, como la verificación de la mayoría de edad.

- User.js

```
class User {
  constructor(name, dateOfBirth) {
    this.name = name;
    this.dateOfBirth = dateOfBirth;
  }

  isAdult() {
    const today = new Date();
    const age = today.getFullYear() - this.dateOfBirth.getFullYear();
    return age >= 18;
  }
}
```

- UserBusinessRules.js

```
function validateUser(user) {
  return user.isAdult();
}
```

Casos de Uso (Use Cases)

Esta capa contiene la lógica específica que implementa los requisitos funcionales, utilizando las entidades del dominio. En nuestro caso, un caso de uso sería "Registrar Usuario".

- RegisterUser.js

```
class RegisterUser {
  constructor(userRepository, user) {
    this.userRepository = userRepository;
    this.user = user;
  }

  execute() {
    if (validateUser(this.user)) {
      this.userRepository.add(this.user);
      return true;
    } else {
      throw new Error("User must be at least 18 years old");
    }
  }
}
```

Adaptadores (Interface Adapters)

Estos adaptadores incluirán controladores y presentadores que interactúan con la capa de dominio y transforman datos para la UI o para servicios externos.

- UserAdapter.js

```
class UserAdapter {  
    static toDTO(user) {  
        return {  
            name: user.name,  
            dateOfBirth: user.dateOfBirth.toISOString().split("T")[0],  
        };  
    }  
}
```

Frameworks y Drivers

Aquí es donde se implementan detalles específicos de tecnología como componentes React para la UI, la configuración de rutas, y servicios que interactúan con bases de datos o APIs externas.

- UserComponent.js (React Component)

```
import React, { useState } from "react";  
  
function UserComponent({ onSubmit }) {  
    const [name, setName] = useState("");  
    const [dateOfBirth, setDateOfBirth] = useState("");  
  
    const handleSubmit = () => {  
        const user = new User(name, new Date(dateOfBirth));  
        onSubmit(user);  
    };  
  
    return (  
        <form onSubmit={handleSubmit}>  
            <input  
                type="text"  
                value={name}  
                onChange={(e) => setName(e.target.value)}  
            />  
            <input  
                type="date"  
                value={dateOfBirth}  
                onChange={(e) => setDateOfBirth(e.target.value)}  
            />  
            <button type="submit">Register</button>  
        </form>  
    );  
}  
export default UserComponent;
```

```
    );
}
```

- App.js (Setting up the application)

```
import React from "react";
import UserComponent from "./UserComponent";
import UserRepository from "./UserRepository";

function App() {
  const userRepository = new UserRepository();

  const handleUserSubmit = (user) => {
    try {
      const registerUser = new RegisterUser(userRepository, user);
      registerUser.execute();
      alert("User registered successfully!");
    } catch (error) {
      alert(error.message);
    }
  };

  return <UserComponent onSubmit={handleUserSubmit} />;
}


```

Enfoque Orgánico de la Estructura de Carpetas

A diferencia de los enfoques tradicionales que estructuran las carpetas de manera muy rígida desde el principio, prefiero un enfoque más orgánico. La idea es permitir que la estructura del proyecto evolucione naturalmente a medida que crece y cambian los requisitos. No me gusta sobreestructurar las carpetas porque creo que la estructura debería surgir de las necesidades del proyecto y del equipo, no al revés.

❖ Introducción a la Regla del Alcance

La regla del alcance es esencial en este enfoque orgánico. Define cómo organizamos y reutilizamos componentes basados en su visibilidad y uso dentro de la aplicación:

- Componentes en Root: Estos son componentes y servicios que son accesibles y reutilizables a través de toda la aplicación. Por ejemplo, componentes de UI genéricos o servicios de autenticación que son necesarios en múltiples partes del sistema.
- Componentes en Funcionalidades Específicas: Localizados en contenedores o módulos específicos, estos componentes solo se utilizan dentro de un contexto o funcionalidad particular. Son perfectos para aplicar lazy loading, ya que solo se cargan cuando se necesita la funcionalidad correspondiente.

Aplicando Clean Architecture con la Regla del Alcance

Aquí está cómo podemos aplicar estos principios juntos para construir una aplicación de front end robusta y mantenible:

- Definir Claramente el Modelo de Dominio: Comenzamos definiendo nuestro modelo de dominio de manera agnóstica, sin preocuparnos por la UI o la infraestructura.
- Desarrollar Casos de Uso: Implementamos la lógica de negocio en forma de casos de uso, que manipulan el modelo de dominio y se comunican con adaptadores.
- Implementar Adaptadores de Forma Flexible: Utilizamos adaptadores para conectar nuestros casos de uso con componentes específicos de la interfaz de usuario y servicios externos. Aquí es donde aplicamos la regla del alcance para decidir si un componente es global o específico de un módulo.
- Usar Lazy Loading para Mejorar el Rendimiento: Cargamos perezosamente módulos o contenedores específicos de funcionalidades según sean necesitados, lo cual es gestionado fácilmente mediante rutas en frameworks modernos como React, Angular o Vue.

Perfecto, vamos a expandir sobre cómo cada módulo o funcionalidad en el front end puede ser organizada para reflejar claramente su propósito y estructura interna, siguiendo la filosofía de la Clean Architecture y la regla del alcance que hemos discutido.

Estructura Modular por Funcionalidad

En un enfoque de desarrollo front end basado en la Clean Architecture, cada característica o funcionalidad de la aplicación se organiza en su propia carpeta, nombrada exactamente igual que la característica que representa. Esta estructura no solo facilita la navegación a través del código y mejora la comprensión del mismo, sino que también encapsula la funcionalidad de manera efectiva.

Componente Contenedor

Dentro de cada carpeta de funcionalidad, se crea un componente principal que lleva el mismo nombre que la carpeta. Este componente actúa como un "contenedor" y tiene dos responsabilidades principales:

- Estructura de la Presentación: Define cómo se estructuran y visualizan los componentes en la pantalla. Este contenedor determina el layout y la composición visual de los componentes hijos que forman la interfaz de la funcionalidad específica.
- Lógica de Negocios y Obtención de Datos: Integra la lógica de negocios que es relevante para el layout, gestionando el estado necesario y realizando las operaciones necesarias para obtener los datos de las entidades del dominio. Esto incluye interactuar con los servicios para recuperar o enviar datos a fuentes externas.

Componentes Específicos de Funcionalidad

Cada componente dentro de la carpeta de funcionalidad maneja su propia funcionalidad específica. Estos componentes están diseñados para ser lo más autónomos posible, interactuando con las entidades del dominio y aplicando las reglas de negocio pertinentes. Su diseño modular y bien definido facilita su reutilización y mantenimiento.

Servicios y Adaptadores

Los servicios son utilizados por los componentes para comunicarse con entidades externas, como backends o APIs. Estos servicios residen generalmente en su propia subcarpeta dentro del módulo de funcionalidad y son responsables de enviar y recibir datos desde y hacia el exterior.

Los adaptadores, por otro lado, se encuentran también en una carpeta propia llamada `adapters` dentro del módulo. Su función es mapear los datos entre la forma esperada por los servicios externos y la forma utilizada por las entidades del dominio. Este mapeo bidireccional asegura que los datos se puedan intercambiar de manera fluida y coherente, respetando las abstracciones impuestas por la arquitectura.

Conclusión de la Implementación

La organización de cada funcionalidad en su propio módulo, con un componente contenedor que gestiona la presentación y la lógica asociada, junto con componentes específicos que se encargan de detalles más granulares, crea un sistema altamente modular y escalable. Esta estructura no solo mejora la legibilidad y el mantenimiento del código, sino que también optimiza el rendimiento mediante técnicas como el lazy loading, cargando solo los módulos necesarios cuando son requeridos por el usuario.

Implementar la Clean Architecture en el front end con este enfoque detallado y organizado prepara el terreno para aplicaciones robustas, mantenibles y adaptables, capaces de evolucionar y expandirse con las necesidades del negocio y del mercado.

¿Qué es el Patrón Contenedor?

El patrón contenedor, también conocido en algunos círculos como "Container Pattern", es una técnica de arquitectura de software que consiste en encapsular o agrupar varios elementos que están relacionados entre sí dentro de un mismo módulo o contenedor. Este contenedor actúa como un límite lógico que define la autonomía y la responsabilidad sobre una porción específica de la funcionalidad de la aplicación.

Estructura del Patrón Contenedor

Imaginemos que estamos trabajando en una aplicación web. En un enfoque tradicional, podrías tener todos tus componentes, lógica de negocio, y llamadas a servicios dispersos por todo el proyecto. En cambio, con el patrón contenedor, organizas estos elementos en grupos lógicos que reflejan sus funciones dentro de la aplicación.

Por ejemplo, si tenemos una sección de la aplicación dedicada a la gestión de usuarios, podríamos tener una estructura de carpetas como esta bajo un directorio `UserManagement`:

```
UserManagement/  
|--- components/
```

```
|   └── UserProfile.js
|   └── UserList.js
├── hooks/
|   └── useUserSearch.js
|   └── useUserProfile.js
└── models/
|   └── UserModel.js
└── services/
|   └── UserService.js
└── UserContainer.js // Este es el contenedor principal
```

Funcionamiento del Patrón Contenedor

- Encapsulación: Cada contenedor maneja su propio estado y dependencias. Esto significa que el contenedor de gestión de usuarios maneja todo lo que es específico para esa función, desde la lógica de negocio hasta la interacción con la API correspondiente.
- Independencia: Los contenedores son independientes entre sí, lo que facilita el desarrollo, testing, y mantenimiento. Si necesitas trabajar en la gestión de usuarios, todo lo que necesitas está en un solo lugar, sin tener que tocar otras partes de la aplicación.
- Reusabilidad: Al encapsular la lógica y los componentes de manera coherente, puedes reutilizar estos contenedores en diferentes partes de la aplicación o incluso en diferentes proyectos, siempre que la funcionalidad sea requerida.
- Integración con Lazy Loading: Cuando utilizamos frameworks modernos como React, Angular o Vue, podemos cargar estos contenedores de forma perezosa (lazy loading). Esto significa que el código relacionado con la gestión de usuarios solo se carga cuando el usuario accede a esa parte específica de la aplicación, mejorando el rendimiento y la velocidad de carga inicial de la app.

Beneficios del Patrón Contenedor

- Mejor Organización: Al tener un límite claro de qué código hace qué, se mejora la legibilidad y se simplifica la estructura del proyecto.
- Desacoplamiento: Reduce las dependencias cruzadas entre diferentes partes de la aplicación, lo que disminuye el riesgo de efectos secundarios indeseados durante las actualizaciones o cambios.
- Escalabilidad: Facilita la gestión del crecimiento de la aplicación. A medida que la aplicación crece, puedes seguir agregando contenedores sin afectar significativamente a los existentes.
- Mantenimiento Facilitado: Actualizar, testear o arreglar bugs se vuelve más fácil porque todo lo que afecta a un área funcional está contenido dentro de su propio módulo.

Implementar el patrón contenedor es como organizar un conjunto de herramientas en cajas específicas en tu taller; cada herramienta tiene su lugar y sabes exactamente dónde encontrar lo que necesitas para cada trabajo. Esto no solo hace tu vida más fácil, sino que también hace que el trabajo sea más eficiente y menos propenso a errores. En el desarrollo de software, especialmente en aplicaciones complejas, adoptar este patrón puede marcar la diferencia entre un proyecto manejable y uno que es un dolor de cabeza constante.

¡Así que dale, probalo y mirá cómo se transforma tu flujo de trabajo!

Adoptar la Clean Architecture en nuestros proyectos no solo mejora la calidad del código sino que también nos prepara para crecer y adaptarnos con facilidad a nuevas demandas y tecnologías. Implementar técnicas como el lazy loading y el patrón contenedor nos asegura una aplicación robusta, mantenible y eficiente.

❖ Ejemplo con todo lo aprendido

¡Claro! Vamos a profundizar en cómo podríamos estructurar un ejemplo práctico usando la Clean Architecture en el front end, aplicando el concepto de módulos y contenedores que discutimos. Supongamos que estamos construyendo una aplicación de comercio electrónico que incluye funcionalidades como listar productos, añadir productos al carrito, y realizar pagos.

Ejemplo de Estructura de Carpeta para una Aplicación de Comercio Electrónico

La siguiente estructura de carpetas refleja cómo podríamos organizar el código de la aplicación:

```
src/
  └── products/
      ├── ProductsContainer.js      // Contenedor para la lista de productos
      ├── components/
      |   ├── ProductList.js        // Muestra la lista de productos
      |   └── ProductItem.js        // Muestra un producto individual
      ├── services/
      |   ├── ProductService.js    // Comunica con la API para obtener productos
      └── adapters/
          └── ProductAdapter.js    // Adapta los datos de productos para la vista
  └── cart/
      ├── CartContainer.js        // Contenedor para el carrito de compras
      ├── components/
      |   ├── CartView.js          // Vista principal del carrito
      |   └── CartItem.js          // Componente para un ítem individual en el carrito
      ├── services/
      |   ├── CartService.js       // Maneja la lógica de agregar/eliminar ítems del carrito
      └── adapters/
          └── CartAdapter.js       // Adapta los datos del carrito para la vista
  └── checkout/
      ├── CheckoutContainer.js    // Contenedor para el proceso de checkout
      ├── components/
      |   ├── PaymentForm.js       // Formulario para detalles de pago
      |   └── OrderSummary.js      // Resumen de la orden antes de la compra
      ├── services/
      |   ├── PaymentService.js    // Procesa los pagos
      └── adapters/
          └── PaymentAdapter.js    // Adapta los datos de pago para ser enviados a la API
```

Detalle de Implementación

Contenedores:

- **ProductsContainer.js:** Este archivo sería responsable de cargar los productos desde el servicio, manejar cualquier estado relacionado con la visualización de los productos y pasar los datos necesarios a los componentes para su visualización.
- **CartContainer.js:** Gestiona el estado del carrito de compras, incluyendo productos añadidos, cantidades seleccionadas y comunicación con servicios para actualizar el carrito.
- **CheckoutContainer.js:** Controla el flujo del proceso de checkout, incluyendo la recolección de información de pago y finalización de la compra.

Componentes:

- Cada componente como **ProductList.js**, **CartItem.js**, y **PaymentForm.js** se encarga de la lógica de aplicación y de los casos de uso, acercando a las entidades con las reglas de negocio.

Servicios y Adaptadores:

- **ProductService.js**, **CartService.js**, y **PaymentService.js** interactúan con APIs externas para enviar y recibir datos.
- Los adaptadores como **ProductAdapter.js** y **PaymentAdapter.js** transforman los datos recibidos de la API al formato que los componentes pueden utilizar más eficientemente y viceversa.

Beneficios de esta Estructura

- **Modularidad:** Cada funcionalidad de la aplicación es autocontenido con su propio conjunto de lógica y presentación, facilitando la mantenibilidad y escalabilidad.
- **Reusabilidad:** Los componentes dentro de cada módulo pueden ser reutilizados en diferentes partes de la aplicación si es necesario.
- **Mantenimiento:** Actualizar o arreglar bugs en una parte específica de la aplicación es más sencillo porque el código afectado está aislado.
- **Rendimiento:** Con técnicas como el lazy loading, solo se cargan los recursos necesarios cuando son realmente necesarios, lo cual puede mejorar significativamente el tiempo de carga de la aplicación.

Dominando React, la joya sin marco

~ React en vez de un framework completo

Queridos lectores y futuros maestros del front-end, en este capítulo vamos a sumergirnos en el fascinante mundo de React, una librería que, aunque a veces la confundimos con un framework, es en realidad una herramienta esencial y flexible para la construcción de interfaces de usuario.

React, desarrollado por Facebook (ahora Meta), se ha ganado un lugar privilegiado en el corazón de los desarrolladores gracias a su simplicidad y eficacia. Pero, ¿cuándo es adecuado usar React solo y no optar por un framework más robusto como Angular o incluso Vue?

Condiciones ideales para usar solo React

- **Proyectos de pequeña a mediana escala:** React es increíblemente eficiente para proyectos que no requieren una gran cantidad de características backend integradas o complejidades adicionales que un framework completo podría manejar mejor.
- **Equipos con experiencia en JavaScript moderno:** Si tu equipo tiene sólidos conocimientos de JavaScript moderno y no quiere lidiar con la curva de aprendizaje de TypeScript (aunque React también se lleva de maravillas con TS), React ofrece una base excelente y flexible para construir sin mucha estructura predefinida.
- **Aplicaciones con necesidad de alta personalización:** Sin un framework dictando la estructura, React permite una personalización y flexibilidad extremas. Esto es ideal para aplicaciones que necesitan una arquitectura única o específica que un framework podría no soportar tan fácilmente.
- **Uso intensivo de componentes reutilizables:** React se centra en la composición de componentes, lo que facilita la reutilización de los mismos. Si tu proyecto se beneficia de un alto grado de reutilización de componentes, React podría ser tu mejor opción.
- **Es posible que no necesites un framework:** Si tu proyecto no requiere de SEO ya que es privado, optar por React Vanilla pudo ser muy beneficioso ya que podemos elegir de forma exclusiva qué tecnologías y herramientas utilizar. Por ejemplo, si tu aplicación es privada y no se beneficiará de las bondades de Server Side Rendering, entonces NextJs podría ser mucho más que lo que realmente necesitas.
- **Un referente en el equipo con experiencia:** Siguiendo lo que hablamos antes, la increíble flexibilidad de React también es un arma de doble filo. Ante un problema hay MUCHAS soluciones, por lo que hay que tener a alguien con experiencia para saber cuál opción es la mejor de acuerdo al contexto en el que se encuentra el equipo y el proyecto.

Integrando React en tu equipo de desarrollo

Integrar React en un equipo de desarrollo requiere considerar tanto las habilidades técnicas como la cultura del equipo. Aquí algunos tips para que la adopción sea un éxito rotundo:

- **Capacitación continua:** Asegúrate de que tu equipo entienda las bases de React y sus patrones de diseño más comunes. Como Gentleman Programming, te recomiendo realizar sesiones de pair programming y code reviews centradas en las mejores prácticas de React.
- **Establece estándares de código:** React es muy flexible, pero esa flexibilidad puede llevar a inconsistencias en el código si no se establecen normas claras. Define guías de estilo y arquitectura desde el principio.
- **Aprovecha la comunidad:** La comunidad de React es vasta y siempre dispuesta a ayudar. Fomenta la participación de tu equipo en foros, seminarios web y conferencias para mantenerse actualizado con las últimas tendencias y mejores prácticas.
- **Prioriza la calidad sobre la velocidad:** Aunque React puede permitir un desarrollo rápido, es esencial que no se sacrifique la calidad del código. Implementa pruebas unitarias y de integración desde el principio para asegurar que las aplicaciones sean robustas y mantenibles.

Conclusión

Usar React sin un framework adicional es perfectamente viable y, en muchas situaciones, la mejor decisión que podrías tomar. Alienta a tu equipo a experimentar con esta librería, adaptándola a las necesidades del proyecto, siempre con un ojo crítico hacia la calidad y la sostenibilidad del código.

~~ Armando nuestro primer componente

Entendiendo JSX

Antes de meter mano en nuestros componentes, es clave que entendamos qué es JSX. JSX es una extensión de sintaxis para JavaScript que nos permite escribir elementos de React de una manera que se parece mucho al HTML, pero con el poder de JavaScript. Esto hace que la escritura de nuestras interfaces sea intuitiva y eficiente.

Por ejemplo, si queremos mostrar un simple "Hola, mundo!", lo haríamos así:

```
function Saludo() {  
  return <h1>Hola, mundo!</h1>;  
}
```

Aunque parece HTML, en realidad JSX es convertido por Babel (un compilador de JavaScript) en llamadas a funciones de React, como `React.createElement`. Así que el ejemplo anterior es en esencia lo mismo que hacer:

```
function Saludo() {  
  return React.createElement("h1", null, "Hola, mundo!");  
}
```

Estructura de un Componente como Función

Un componente funcional es simplemente una función de JavaScript que retorna un elemento de React, que puede ser una simple descripción de la UI o puede incluir más lógica y otros componentes. Es la forma más directa y moderna de definir componentes, especialmente desde la introducción de Hooks, que permiten usar estado y otros features de React sin escribir una clase.

Veamos la estructura básica de un componente funcional:

```
// Definimos un componente funcional que acepta props
function Bienvenida(props) {
  // Podemos acceder a las propiedades enviadas al componente a través de `props`
  return <h1>Bienvenido, {props.nombre}</h1>;
}

// Uso del componente con una prop 'nombre'
<Bienvenida nombre="Gentleman" />;
```

En este ejemplo, `Bienvenida` es un componente que recibe `props`, un objeto que contiene todas las propiedades que pasamos al componente. Usamos `{}` dentro del JSX para insertar valores de JavaScript, en este caso para mostrar dinámicamente el nombre que recibimos.

Componentes Stateful vs Stateless

Volvamos a la distinción entre componentes con estado (stateful) y sin estado (stateless):

- **Stateful Components:** Manejan algún tipo de estado interno o datos cambiantes. Aunque aún no hablaremos de Hooks, es importante saber que estos componentes en el futuro podrán usar cosas como `useState` para gestionar su estado interno.
- **Stateless Components:** Simplemente aceptan datos a través de `props` y muestran algo en la pantalla. No mantienen ningún estado interno y son típicamente usados para mostrar UI:

```
function Mensaje({ texto }) {
  return <p>{texto}</p>;
}
```

❖ UseState

En React, manejar el estado de nuestros componentes es crucial para controlar su comportamiento y datos en tiempo real. La función `useState` es una herramienta esencial en este proceso, similar a cómo gestionamos las decisiones diarias en nuestras vidas.

¿Qué es `useState`?

Imagina useState como una caja fuerte en tu casa, donde guardas un objeto valioso que puede cambiar con el tiempo, como el dinero que decides gastar o ahorrar. Esta caja fuerte tiene una forma especial de mostrarte cuánto dinero hay dentro (get) y una manera de actualizar esta cantidad (set).

Estructura de useState como una Clase:

Si pensamos en useState como si fuera una clase, podría verse de la siguiente manera:

```
class State {  
  constructor(initialValue) {  
    this.value = initialValue; // El valor inicial se almacena aquí  
  }  
  
  getValue() {  
    return this.value; // Método para obtener el valor actual  
  }  
  
  setValue(newValue) {  
    this.value = newValue; // Método para actualizar el valor  
  }  
}
```

En esta estructura, value representa el estado actual, y tenemos métodos `getValue()` y `setValue(newValue)` para interactuar con este estado.

Uso de useState en un Componente

Para entenderlo mejor, vamos a compararlo con algo cotidiano: ajustar la temperatura de un aire acondicionado en tu casa.

Supongamos que quieres mantener una temperatura agradable mientras estás en casa. Usarás un control (como el useState) para ajustar esta temperatura. Aquí te muestro cómo:

```
import React, { useState } from "react";  
  
function AirConditioner() {  
  const [temperature, setTemperature] = useState(24); // 24 grados es la temperatura inicial  
  
  const increaseTemperature = () => setTemperature(temperature + 1);  
  const decreaseTemperature = () => setTemperature(temperature - 1);  
  
  return (  
    <div>  
      <h1>Temperatura Actual: {temperature}°C</h1>  
      <button onClick={increaseTemperature}>Subir Temperatura</button>  
      <button onClick={decreaseTemperature}>Bajar Temperatura</button>  
    </div>  
  );  
}  
  
export default AirConditioner;
```

```
});  
}
```

En este ejemplo, `temperature` es el estado que estamos manejando. Iniciamos con una temperatura de 24 grados. Los métodos `increaseTemperature` y `decreaseTemperature` actúan como los botones de subir o bajar la temperatura en el control del aire acondicionado.

~~ Componentes Funcionales: Como una Receta de Cocina

Imagina que un componente funcional en React es como seguir una receta de cocina. Cada vez que decides cocinar algo, sigues los pasos para preparar tu plato. De forma similar, un componente funcional "sigue los pasos" cada vez que React decide que necesita actualizar lo que se muestra en pantalla.

Preparación de Ingredientes (Props)

Cuando cocinas, primero recoges todos los ingredientes que necesitas. En un componente funcional, estos ingredientes son las `props`. Las `props` son datos o información que pasas al componente para que haga su trabajo, como los ingredientes en tu receta que determinan cómo sale el plato.

```
function Sandwich({ relleno, pan }) {  
  return (  
    <div>  
      Un sandwich de {relleno} en pan de {pan}.  
    </div>  
  );  
}
```

En este ejemplo, `relleno` y `pan` son las `props`, los ingredientes que necesitas para hacer tu sandwich.

Ejecución de la Receta (Función del Componente)

Cada vez que haces la receta, sigues los pasos para combinar los ingredientes y cocinarlos. Cada ejecución puede variar ligeramente, por ejemplo, puedes decidir poner más especias o menos sal. En un componente funcional, la "ejecución" es cuando React llama a la función del componente para que genere el JSX basado en las `props` actuales.

Cada vez que las `props` cambian, es como si decidieras ajustar la receta. React "cocina" de nuevo el componente, es decir, ejecuta la función del componente para ver cómo debe verse ahora con los nuevos "ingredientes".

Presentación del Plato (Renderizado)

La presentación final es cuando pones el plato cocinado en la mesa. En React, esto es lo que ves en la pantalla después de que el componente se ejecuta. El JSX que retorna la función del componente determina cómo se "presenta" el componente en la interfaz de usuario.

Ejemplo de Uso

Usar el componente es como servir tu plato cocinado a alguien para que lo disfrute.

```
<Sandwich relleno="jamón y queso" pan="integral" />
```

Cada vez que los detalles del sandwich cambian (digamos, cambias `relleno` a "pollo y tomate"), React realiza el proceso nuevamente para asegurarse de que la presentación en pantalla coincida con los ingredientes dados.

Este enfoque te ayuda a ver cada componente como una receta individual, donde las props son tus ingredientes y la función del componente es la guía de cómo combinarlos para obtener el resultado final en la pantalla, siempre fresco y actualizado según los ingredientes que proporcionas.

~~ Virtual DOM

Imaginá que el DOM (Document Object Model) es un escenario donde cada elemento HTML es un actor. Cada vez que algo cambia en tu página web (por ejemplo, un usuario interactúa con ella o los datos recibidos de un servidor alteran el estado de la página), podrías tener que reorganizar a los actores en este escenario para reflejar esos cambios. Sin embargo, reorganizar estos actores (elementos del DOM) directamente y con frecuencia es muy costoso en términos de rendimiento.

Aquí es donde entra en juego el Virtual DOM. React mantiene una copia ligera de este escenario en memoria, una especie de boceto o script de cómo está organizado el escenario en un momento dado. Este boceto es lo que llamamos Virtual DOM.

Funcionamiento del Virtual DOM

- **Actualización del estado o las props:** Cada vez que hay un cambio en el estado o las props de tu aplicación, React actualiza este Virtual DOM. No hay cambios en el escenario real todavía, solo en el boceto.
- **Comparación con el DOM real:** React compara este Virtual DOM actualizado con una versión anterior del Virtual DOM (la última vez que el estado o las props fueron actualizados).
- **Detección de diferencias:** Este proceso de comparación se conoce como "reconciliación". React identifica qué partes del Virtual DOM han cambiado (por ejemplo, un actor necesita moverse de un lado del escenario al otro).
- **Actualización eficiente del DOM real:** Una vez que React sabe qué cambios son necesarios, actualiza el DOM real de la manera más eficiente posible. Esto es como dar instrucciones específicas a los actores sobre cómo reubicarse en el escenario sin tener que reconstruir toda la escena desde cero.

Ventajas del Virtual DOM

- **Eficiencia:** Al trabajar con el Virtual DOM, React puede minimizar el número de manipulaciones costosas del DOM real. Solo hace los cambios necesarios y los hace de manera que afecte lo menos posible el rendimiento de la página.
- **Rapidez:** Como las operaciones con el Virtual DOM son mucho más rápidas que las operaciones directas con el DOM real, React puede manejar cambios a alta velocidad sin degradar la experiencia del usuario.
- **Simplicidad en el desarrollo:** Como desarrolladores, no tenemos que preocuparnos por cómo y cuándo actualizar el DOM. Nos centramos en el estado de la aplicación, y React se encarga del resto.

~~ Detección de Cambios: Comprendiendo el Flujo

En el universo de React, la detección de cambios es como el radar en un partido de fútbol; está constantemente monitoreando y asegurándose de que todo lo que sucede en el campo de juego se maneja adecuadamente. Vamos a desglosar cómo funciona este proceso, enfocándonos en el concepto de triggers y cómo estos influyen en el renderizado de los componentes.

¿Qué es un Trigger?

Un trigger en React es cualquier evento que inicia el proceso de renderizado. Esto puede ser tan simple como un clic en un botón, un cambio en el estado del componente, o incluso una respuesta a una llamada API que llega de forma asíncrona.

Imagínate que estás en una cocina: cada acción que realizas, desde encender la estufa hasta cortar verduras, puede ser vista como un trigger. En React, cada uno de estos triggers tiene el potencial de actualizar la interfaz de usuario, dependiendo de cómo esté configurada la lógica en tus componentes.

Tipos de Triggers

Hay dos tipos fundamentales de triggers en React:

- **Inicial:** Es como el silbato inicial de un partido. Se da cuando el componente se monta por primera vez en el DOM. En términos técnicos, esto se refiere a cuando se crea la raíz de tu aplicación en React y se carga el componente inicial.
- **Re-renders:** Estos ocurren después del montaje inicial. Cada vez que hay una actualización en el estado o las props, React decide si necesita re-renderizar el componente para reflejar esos cambios. Es como hacer ajustes en tu estrategia de juego en tiempo real.

El Proceso de Renderizado

Renderizar en React es como preparar y presentar un plato. Cuando se invoca un render, React prepara la UI basada en el estado actual y las props, y luego la sirve en la pantalla. Este proceso se repite cada vez que un

trigger activa un cambio.

El "render" no es más que la función que compone tu componente. Cada vez que esta función se ejecuta, React evalúa el JSX returned y actualiza el DOM en consecuencia, siempre y cuando detecte diferencias entre el DOM actual y el output del render.

Commit: Actualizando el DOM

Una vez que React ha preparado la nueva vista en memoria (a través del Virtual DOM), se realiza un "commit". Este es el proceso de aplicar cualquier cambio detectado al DOM real. Es como si, después de preparar el plato en la cocina, finalmente lo llevas a la mesa. React compara el nuevo Virtual DOM con el anterior y realiza las actualizaciones necesarias para que el DOM real refleje estos cambios.

Este proceso asegura que solo se actualicen las partes del DOM que realmente necesitan cambios, optimizando el rendimiento y evitando renderizados innecesarios.

Recapitulando

Cada componente en React actúa como un pequeño chef en la cocina de una gran restaurante, preparando su parte del plato. React, como el chef principal, se asegura de que cada componente haga su parte solo cuando es necesario, basándose en los triggers recibidos. Este enfoque asegura que la cocina (tu aplicación) funcione de manera eficiente y efectiva, respondiendo adecuadamente a las acciones del usuario y otros eventos.

~ Dominando Custom Hooks

Introducción a los Custom Hooks

En el universo de React, los Custom Hooks son como recetas personalizadas en la cocina: nos permiten mezclar ingredientes comunes de maneras nuevas y emocionantes para crear platos (o componentes) únicos y reutilizables. A lo largo de este capítulo, exploraremos por qué los Custom Hooks son esenciales para simplificar la lógica y mejorar la reusabilidad en nuestras aplicaciones.

La Charla Técnica: Ciclos de Vida y Custom Hooks

Para comprender mejor los Custom Hooks, pensemos en una cafetería. Al igual que un barista que prepara tu café favorito exactamente cuando lo pides, los Custom Hooks nos permiten "servir" lógica específica en el momento justo del ciclo de vida de un componente en React.

Ahora, supongamos que quieres que algo suceda automáticamente en tu aplicación, como encender una luz cuando oscurece. Aquí te muestro cómo un Custom Hook puede manejar este "encendido automático":

```
function useAutoLight() {  
  const [isDark, setDark] = useState(false);  
}
```

```

useEffect(() => {
  const handleDarkness = () => {
    const hour = new Date().getHours();
    setDark(hour > 18 || hour < 6);
  };

  window.addEventListener("timeChange", handleDarkness);
  return () => window.removeEventListener("timeChange", handleDarkness);
}, []); // Se ejecuta una vez al montar y al desmontar

return isDark;
}

```

Este Hook encapsula la lógica para detectar si está oscuro y actuar en consecuencia, similar a un sensor de luz automático en tu casa.

Aplicaciones Prácticas de Custom Hooks

Explorando cómo los Custom Hooks se implementan en situaciones cotidianas, podemos pensar en ellos como atajos en un dispositivo móvil, permitiéndonos realizar tareas comunes con mayor rapidez y eficiencia.

Custom Hook para Manejo de Formularios:

Considera el proceso de llenar un diario personal. Quieres que sea fácil registrar tus pensamientos sin distracciones. Mira cómo este Custom Hook simplifica el manejo de un "diario digital":

```

function useDiary(initialEntries) {
  const [entries, setEntries] = useState(initialEntries);

  const addEntry = (newEntry) => {
    setEntries([...entries, newEntry]);
  };

  return {
    entries,
    addEntry,
  };
}

```

Este Hook actúa como un asistente personal para tu diario, ayudándote a añadir nuevos pensamientos de manera organizada y eficiente.

❖ Uso Correcto de useEffect: Evitando Errores Comunes

En este capítulo, vamos a adentrarnos en el uso correcto del hook `useEffect` en React, una herramienta fundamental para manejar los efectos secundarios en tus componentes. Aunque `useEffect` es muy potente,

es crucial saber cuándo y cómo usarlo para evitar problemas de rendimiento y mantener el código limpio y manejable.

~ Introducción al useEffect

El `useEffect` se usa para gestionar efectos secundarios en tus componentes de React. Pero, ¿qué es un efecto secundario? Imagina que `useEffect` es como un cronómetro en tu cocina que se activa cuando metes una pizza al horno. No importa lo que estés haciendo, cuando el cronómetro suena, sabes que la pizza está lista y necesitas sacarla del horno.

Estructura Básica de useEffect

El `useEffect` tiene una estructura sencilla pero poderosa:

```
import React, { useEffect, useState } from "react";

function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    console.log("El componente se ha montado o actualizado.");

    return () => {
      console.log("El componente se va a desmontar.");
    };
  }, []);

  return <h1>Hola React!</h1>;
}

}
```

En este ejemplo, `useEffect` se ejecuta después de que el componente se renderiza por primera vez, similar a poner un cronómetro cuando metes la pizza al horno. La función de limpieza (`return`) es como sacar la pizza y apagar el cronómetro cuando terminas.

Evitando el Uso Incorrecto de useEffect

El `useEffect` no debe ser usado para todo. Utilizarlo incorrectamente puede llevar a problemas de rendimiento y lógica compleja innecesaria. Veamos algunos ejemplos comunes de errores y cómo evitarlos.

1. Evitar Bucles Infinitos

Un error común es causar un bucle infinito al actualizar el estado dentro de `useEffect` sin manejar correctamente las dependencias:

```
const [count, setCount] = useState(0);

useEffect(() => {
  setCount(count + 1);
}, [count]);
```

En este ejemplo, cada vez que count cambia, `useEffect` se vuelve a ejecutar, actualizando count nuevamente, lo que causa un bucle infinito. Es como si cada vez que sacaras la pizza del horno, volvieras a meterla y reiniciar el cronómetro.

Solución: Ajusta las dependencias correctamente y evita actualizar el estado dentro del mismo `useEffect` que depende de ese estado.

```
const [count, setCount] = useState(0);

useEffect(() => {
  const interval = setInterval(() => {
    setCount((c) => c + 1);
  }, 1000);

  return () => clearInterval(interval);
}, []);
```

Aquí, `useEffect` solo se ejecuta una vez al montar el componente, y el estado se actualiza cada segundo sin causar un bucle infinito.

2. Evitar Ejecutar Lógica en Cambios de Estado con `useEffect`

Un caso incorrecto de uso de `useEffect` es cuando se ejecuta una lógica al cambiar una variable del componente. Para esto, es mejor ejecutar la lógica en el momento de la acción, como un clic.

Incorrecto:

```
const [value, setValue] = useState("");

useEffect(() => {
  console.log("El valor ha cambiado:", value);
}, [value]);
```

En lugar de usar `useEffect` para detectar cambios en `value`, es más eficiente y claro ejecutar la lógica directamente en el manejador del evento.

Correcto:

```
const handleChange = (newValue) => {
  setValue(newValue);
  console.log("El valor ha cambiado:", newValue);
};
```

Esto es como si en lugar de usar un cronómetro para sacar la pizza del horno, simplemente prestaras atención al horno y sacaras la pizza cuando suena la alarma.

Casos Correctos para useEffect

1. Llamadas a APIs

Un uso correcto de `useEffect` es para llamadas a APIs, donde necesitas realizar una acción asíncrona cuando se monta el componente:

```
useEffect(() => {
  const fetchData = async () => {
    const response = await fetch("https://api.example.com/data");
    const result = await response.json();
    setData(result);
  };

  fetchData();
}, []);
```

Aquí, `useEffect` actúa como tu recordatorio de sacar la pizza justo cuando el temporizador suena, asegurando que los datos se obtienen correctamente al montar el componente.

2. Suscripciones y Limpieza

`useEffect` es útil para suscribirse a servicios externos y limpiar esas suscripciones cuando el componente se desmonta:

```
useEffect(() => {
  const subscription = someService.subscribe((data) => {
    setData(data);
  });

  return () => {
    subscription.unsubscribe();
  };
}, []);
```

Es como suscribirse a un boletín y cancelar la suscripción cuando ya no te interesa.

3. Sincronización de Datos

Otra aplicación útil de `useEffect` es la sincronización de datos entre componentes o con servicios externos. Por ejemplo, sincronizar el estado local con el almacenamiento local del navegador:

```
useEffect(() => {
  const savedData = localStorage.getItem("data");
  if (savedData) {
    setData(JSON.parse(savedData));
  }
}, []);

useEffect(() => {
  localStorage.setItem("data", JSON.stringify(data));
}, [data]);
```

Aquí, el primer `useEffect` actúa como un asistente que verifica si hay pizza en la nevera al abrir la puerta, y el segundo asegura que cualquier cambio en los ingredientes se guarda automáticamente.

~ Comunicación entre Componentes con `children` usando el Patrón Composition

En este capítulo, vamos a explorar cómo manejar la comunicación entre componentes en React utilizando el patrón de composición y los `children`. Estos conceptos permiten crear componentes flexibles y reutilizables, facilitando la transferencia de datos y la gestión del estado entre componentes padres e hijos.

Introducción a la Composición de Componentes

El patrón de composición en React nos permite construir componentes complejos a partir de componentes más pequeños y específicos. Es como construir un automóvil a partir de diversas piezas: motor, ruedas, chasis, etc. Cada una de estas piezas tiene una función específica, pero juntas forman un automóvil funcional.

Entendiendo los `children` en React

Podemos utilizar las `props.children` para pasar contenido de un componente padre a un componente hijo de manera dinámica.

Ejemplo Básico de `props.children`

```
const Container = ({ children }) => {
  return <div className="container">{children}</div>;
};

const App = () => {
  return (
```

```
<Container>
  <h1>Hola, Mundo!</h1>
  <p>Este es un párrafo dentro del contenedor.</p>
</Container>
);
};
```

En este ejemplo, **Container** es un componente que envuelve a sus hijos, que son pasados dinámicamente desde el componente **App**.

Patrón de Composición

El patrón de composición se basa en la idea de componer componentes más pequeños para crear interfaces de usuario complejas. Imagina que estás construyendo una página web como si fuera un lego. Cada bloque es un componente que puedes combinar para crear algo más grande y funcional.

Ejemplo de Composición con Múltiples slots

```
const Layout = ({ header, main, footer }) => {
  return (
    <div className="layout">
      <header className="layout-header">{header}</header>
      <main className="layout-main">{main}</main>
      <footer className="layout-footer">{footer}</footer>
    </div>
  );
};

const App = () => {
  return (
    <Layout
      header={<h1>Bienvenido</h1>}
      main={<p>Este es el contenido principal.</p>}
      footer={<small>© 2024 Mi Sitio Web</small>}
    />
  );
};
```

En este ejemplo, el componente **Layout** actúa como un contenedor que organiza sus hijos en diferentes secciones (**header**, **main**, **footer**). Cada sección es pasada como una prop específica.

Comunicación entre Componentes Padres e Hijos

La comunicación entre componentes en React se maneja principalmente a través de props. Los componentes padres pueden pasar datos y funciones a los componentes hijos, permitiéndoles controlar el comportamiento y el estado de los hijos desde el padre.

Ejemplo de Comunicación entre Componentes

```

const Button = ({ onClick, children }) => {
  return <button onClick={onClick}>{children}</button>;
};

const App = () => {
  const handleClick = () => {
    alert("¡Botón clickeado!");
  };

  return (
    <div>
      <Button onClick={handleClick}>Click Me</Button>
    </div>
  );
};

```

En este ejemplo, el componente `Button` recibe una función `onClick` como prop desde el componente padre `App`. Cuando el botón se clickea, se ejecuta la función `handleClick` definida en el parent.

Ejemplo Completo con Patrón de Composición y Slots

Para ilustrar todo lo que hemos aprendido, veamos un ejemplo más completo que utiliza el patrón de composición y maneja la comunicación entre componentes padres e hijos de manera efectiva.

Ejemplo Completo

```

const Panel = ({ title, content, actions }) => {
  return (
    <div className="panel">
      <div className="panel-header">
        <h2>{title}</h2>
      </div>
      <div className="panel-content">{content}</div>
      <div className="panel-actions">{actions}</div>
    </div>
  );
};

const App = () => {
  const handleSave = () => {
    alert("Guardado!");
  };

  const handleCancel = () => {
    alert("Cancelado");
  };

  return (

```

```
<Panel
  title="Configuración"
  content={<p>Aquí puedes configurar tus preferencias.</p>}
  actions={[
    <div>
      <button onClick={handleSave}>Guardar</button>
      <button onClick={handleCancel}>Cancelar</button>
    </div>
  ]}
/>
);
};
```

En este ejemplo, el componente `Panel` se compone de tres secciones (`title`, `content`, `actions`) que son pasadas desde el componente `App`. Esto permite una gran flexibilidad y reutilización de componentes.

Casos Prácticos y Beneficios

- **Flexibilidad:** Puedes diseñar componentes altamente configurables y reutilizables.
- **Separación de Concerns:** Mantiene la lógica de cada componente separada, facilitando el mantenimiento.
- **Reusabilidad:** Componentes bien diseñados pueden ser reutilizados en múltiples lugares de la aplicación.

Comparación con la Vida Cotidiana

Imagina que estás organizando una fiesta y decides delegar tareas a diferentes personas (componentes). Tienes a alguien encargado de las bebidas, otro de la música y otro de la comida. Cada uno trabaja de manera independiente, pero al final, todos se coordinan para que la fiesta sea un éxito. Este es el poder del patrón de composición en React.

~~ Comunicación entre Componentes: Composición vs Contexto vs Herencia

Cuando trabajamos con React, uno de los desafíos más comunes es compartir información entre componentes que no tienen una relación directa. En este capítulo, vamos a explorar tres enfoques principales para resolver este problema: Prop Drilling, Context y Composición. Cada enfoque tiene sus ventajas y desventajas, y es importante comprender cuándo y cómo usarlos para crear aplicaciones eficientes y mantenibles.

Estructura Básica de Componentes

Primero, visualicemos una estructura básica de componentes para entender mejor los ejemplos:

```
<Father>
  <Child1 />
</Father>

**Child1:**
<Child1>
  <Child2 />
</Child1>
```

~~ Compartiendo Información Entre Componentes Sin Relación Directa

Prop Drilling

Prop Drilling es el método más directo para pasar información desde un componente padre a un componente hijo, y luego a un nieto. Sin embargo, puede volverse problemático a medida que la aplicación crece.

Ejemplo de Prop Drilling:

```
const Father = () => {
  const sharedProp = "Información Compartida";
  return <Child1 sharedProp={sharedProp} />;
};

const Child1 = ({ sharedProp }) => {
  return <Child2 sharedProp={sharedProp} />;
};

const Child2 = ({ sharedProp }) => {
  return <div>{sharedProp}</div>;
};
```

Problemas de Prop Drilling:

- **Acoplamiento Alto:** Los componentes están fuertemente acoplados, lo que dificulta su reutilización.
- **Mantenimiento Difícil:** A medida que la cantidad de componentes aumenta, se vuelve complicado mantener el código.
- **Entendimiento Complicado:** Es difícil seguir la trayectoria de los props a través de muchos niveles de componentes.

Gestión del Estado con Context

El Context API de React proporciona una forma más limpia y escalable de compartir información entre componentes que no tienen una relación directa. Utiliza un proveedor (provider) que mantiene el estado compartido, y consumidores (consumers) que pueden acceder a ese estado.

Ejemplo de Context API:

```
const MyContext = React.createContext();

const Father = () => {
  const [sharedState, setSharedState] = React.useState("Estado Compartido");

  return (
    <MyContext.Provider value={sharedState}>
      <Child1 />
    </MyContext.Provider>
  );
};

const Child1 = () => {
  return <Child2 />;
};

const Child2 = () => {
  const sharedState = React.useContext(MyContext);
  return <div>{sharedState}</div>;
};
```

Ventajas del Context API:

- **Entendible:** Es fácil seguir el flujo de datos.
- **Escalable:** Puede manejar aplicaciones de cualquier tamaño.
- **Directo:** Los componentes pueden acceder directamente al estado compartido sin pasar props innecesarios.

Desventajas del Context API:

- **Dependencia de la Ubicación del Proveedor:** Los componentes solo pueden acceder al contexto si están dentro del proveedor.

```
<MyContext.Provider>
  <Father>
    <Child1 />
  </Father>
</MyContext.Provider>
```

```
<FatherOutsideContextProvider>
  <Child2 /> // No puede acceder al estado del contexto
</FatherOutsideContextProvider>
```

Composición para la Victoria

La composición en React permite construir componentes más complejos a partir de componentes más simples. En lugar de pasar props a través de muchos niveles, podemos utilizar la propiedad `children` para pasar elementos directamente.

Ejemplo de Composición:

```
const Father = () => {
  const sharedProp = "Información Compartida";

  return (
    <Child1>
      <Child2 sharedProp={sharedProp} />
    </Child1>
  );
};

const Child1 = ({ children }) => {
  return <div>{children}</div>;
};

const Child2 = ({ sharedProp }) => {
  return <div>{sharedProp}</div>;
};
```

Ventajas de la Composición

- **Sencillo:** Menos código repetitivo y más claro.
- **Escalable:** Fácil de escalar a medida que la aplicación crece.
- **Sin Dependencias:** No depende de la ubicación del proveedor como en el Context API.
- **Código Reutilizable:** Los componentes pueden ser reutilizados fácilmente en diferentes partes de la aplicación.
- **Lógica Individual:** Cada componente maneja su propia lógica de manera independiente.

❖ Uso de Context

En el desarrollo de aplicaciones con React, a menudo necesitamos compartir información entre componentes sin una relación directa entre ellos. El Context API de React proporciona una forma más robusta y escalable para manejar este tipo de situaciones.

¿Qué es el Context?

El Context API de React permite compartir valores entre componentes sin tener que pasar explícitamente props por cada nivel del árbol de componentes. Es particularmente útil para temas, configuraciones de idioma, o cualquier otro tipo de dato que deba ser accesible en muchas partes de la aplicación.

Creando un Context

Primero, necesitamos crear un contexto. Esto se hace con la función `createContext` de React:

```
import { createContext, useContext, useState } from "react";

export const GentlemanContext = createContext();
```

El `GentlemanContext` ahora puede ser utilizado para proveer y consumir valores en nuestra aplicación.

Proveedor del Contexto

El proveedor (**Provider**) del contexto es el componente que envuelve a aquellos componentes que necesitan acceder al contexto. Define el valor del contexto que será compartido.

```
export const GentlemanProvider = ({ children }) => {
  const [gentlemanContextValue, setGentlemanContextValue] = useState("");
  return (
    <GentlemanContext.Provider
      value={{ gentlemanContextValue, setGentlemanContextValue }}
    >
      {children}
    </GentlemanContext.Provider>
  );
};
```

En este ejemplo, `GentlemanProvider` utiliza el hook `useState` para mantener y actualizar el valor del contexto. Este valor puede ser cualquier tipo de dato, como un string, un número, un objeto o una función.

Consumiendo el Contexto

Para acceder al valor del contexto en un componente hijo, utilizamos el hook `useContext`:

```
export const useGentlemanContext = () => {
  const context = useContext(GentlemanContext);
```

```
if (context === undefined) {
  throw new Error("GentlemanContext must be used within a GentlemanProvider");
}
return context;
};
```

El hook `useGentlemanContext` encapsula el uso de `useContext` y se asegura de que el contexto sea utilizado correctamente dentro de un `GentlemanProvider`.

Ejemplo Completo

Veamos un ejemplo completo de cómo utilizar este contexto en una aplicación:

```
import React from "react";
import ReactDOM from "react-dom";
import { GentlemanProvider, useGentlemanContext } from "./GentlemanContext";

const Father = () => {
  const { gentlemanContextValue, setGentlemanContextValue } =
    useGentlemanContext();

  return (
    <div>
      <h1>Father Component</h1>
      <button onClick={() => setGentlemanContextValue("Valor compartido")}>
        Set Context Value
      </button>
      <Child1 />
    </div>
  );
};

const Child1 = () => {
  return (
    <div>
      <h2>Child1 Component</h2>
      <Child2 />
    </div>
  );
};

const Child2 = () => {
  const { gentlemanContextValue } = useGentlemanContext();

  return (
    <div>
      <h3>Child2 Component</h3>
      <p>Context Value: {gentlemanContextValue}</p>
    </div>
  );
};
```

```
};

const App = () => (
  <GentlemanProvider>
    <Father />
  </GentlemanProvider>
);

ReactDOM.render(<App />, document.getElementById("root"));
```

Ventajas del Context API

- 1. Entendible:** El flujo de datos es claro y fácil de seguir. Los componentes pueden acceder directamente al contexto sin la necesidad de pasar props innecesarias.
- 2. Escalable:** El Context API puede manejar aplicaciones de cualquier tamaño sin los problemas de mantenimiento asociados al Prop Drilling.
- 3. Centralizado:** El estado compartido está centralizado, lo que facilita su gestión y actualización.

Desventajas del Context API

- 1. Ubicación del Proveedor:** Los componentes solo pueden acceder al contexto si están dentro del proveedor. Si el proveedor está mal colocado, puede que algunos componentes no tengan acceso al contexto.

```
<MyContext.Provider>
  <Father>
    <Child1 />
  </Father>
</MyContext.Provider>

<FatherOutsideContextProvider>
  <Child2 /> // No puede acceder al estado del contexto
</FatherOutsideContextProvider>
```

~~ Comprendiendo useRef, useMemo y useCallback

En este capítulo, vamos a desmitificar el uso de algunos de los hooks más potentes y a menudo mal entendidos en React: `useRef`, `useMemo` y `useCallback`. Veremos cómo y cuándo utilizarlos correctamente, y los compararemos con el conocido `useState` para entender sus diferencias y similitudes.

useRef: La Referencia Constante

El hook `useRef` nos permite crear una referencia mutable que puede persistir a lo largo del ciclo de vida completo del componente sin causar re-renderizados.

Ejemplo práctico con useRef:

Supongamos que tenemos un componente con un botón que, al ser clicado, activa automáticamente otro botón:

```
import { useRef } from "react";

const Clicker = () => {
  const buttonRef = useRef(null);

  const handleClick = () => {
    buttonRef.current.click();
  };

  return (
    <div>
      <button ref={buttonRef} onClick={() => alert("Button clicked!")}>
        Hidden Button
      </button>
      <button onClick={handleClick}>Click to trigger Hidden Button</button>
    </div>
  );
};

export default Clicker;
```

En este ejemplo, `useRef` se utiliza para acceder directamente al botón oculto y simular un clic programáticamente. `useRef` no causa re-renderizados del componente cuando su valor cambia, lo que lo hace ideal para almacenar referencias a elementos DOM o a cualquier valor que necesite persistir sin causar renders adicionales.

useMemo: Memorización de Cálculos

`useMemo` se utiliza para memorizar valores calculados costosos y solo recalcularlos cuando una de las dependencias ha cambiado. Esto es especialmente útil para mejorar el rendimiento de componentes que realizan cálculos intensivos.

Ejemplo práctico con useMemo:

```
import { useMemo, useState } from "react";

const ExpensiveCalculationComponent = ({ number }) => {
  const expensiveCalculation = (num) => {
    console.log("Calculating...");
    return num * 2; // Simula una operación costosa
  };
};
```

```

const memoizedValue = useMemo(() => expensiveCalculation(number), [number]);

return (
  <div>
    <h2>Resultado: {memoizedValue}</h2>
  </div>
);
};

const ParentComponent = () => {
  const [number, setNumber] = useState(1);

  return (
    <div>
      <ExpensiveCalculationComponent number={number} />
      <button onClick={() => setNumber(number + 1)}>Incrementar</button>
    </div>
  );
};

export default ParentComponent;

```

En este ejemplo, `useMemo` memoriza el resultado de `expensiveCalculation` y solo recalcula el valor cuando `number` cambia, evitando ejecuciones innecesarias de una función costosa.

useCallback: Memorización de Funciones

`useCallback` es similar a `useMemo`, pero en lugar de memorizar el resultado de una función, memoriza la propia función. Esto es útil para evitar re-creaciones innecesarias de funciones, especialmente cuando se pasan como props a componentes hijos que podrían causar re-renderizados adicionales.

Ejemplo práctico con `useCallback`:

```

import { useCallback, useState } from "react";

const ChildComponent = ({ handleClick }) => {
  console.log("Child rendered");
  return <button onClick={handleClick}>Click me</button>;
};

const ParentComponent = () => {
  const [count, setCount] = useState(0);

  const handleClick = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, []);

  return (
    <div>
      <h2>Count: {count}</h2>
    </div>
  );
};

```

```
        <ChildComponent handleClick={handleClick} />
      </div>
    );
};

export default ParentComponent;
```

En este ejemplo, `useCallback` asegura que `handleClick` mantenga la misma referencia entre renderizados, evitando re-renderizados innecesarios del `ChildComponent`.

Comparación entre `useState`, `useRef`, `useMemo` y `useCallback`

`useState`:

- **Propósito:** Manejo del estado interno del componente.
- **Re-render:** Causa re-render del componente cuando el estado cambia.
- **Ejemplo:** Contadores, toggles.

`useRef`:

- **Propósito:** Crear referencias mutables que persisten a través del ciclo de vida del componente.
- **Re-render:** No causa re-render cuando el valor cambia.
- **Ejemplo:** Referencias a elementos DOM, almacenar valores persistentes.

`useMemo`:

- **Propósito:** Memorizar valores calculados costosos para evitar recalculaciones innecesarias.
- **Re-render:** No causa re-render, memoriza el resultado del cálculo.
- **Ejemplo:** Calcular valores derivados de estados complejos.

`useCallback`:

- **Propósito:** Memorizar funciones para evitar re-creaciones innecesarias.
- **Re-render:** No causa re-render, memoriza la referencia de la función.
- **Ejemplo:** Pasar callbacks a componentes hijos que dependen de funciones memoizadas.

~ Peticiones a una API y Manejo de Lógica Asíncrona

En este capítulo, vamos a explorar cómo realizar peticiones a una API de manera correcta en React, manejar la lógica asíncrona, y cómo cachear resultados en el Local Storage para mejorar el rendimiento de nuestra aplicación.

Realizando Peticiones a una API

Vamos a desglosar cómo realizar peticiones a una API utilizando `fetch`, explicar las partes que lo componen y entender por qué utilizamos funciones asíncronas (`async`) en lugar de hacer las peticiones directamente dentro de `useEffect`.

La Función Fetch

`fetch` es una función nativa de JavaScript utilizada para realizar peticiones HTTP a una API. Su uso básico es el siguiente:

```
fetch("https://api.example.com/data")
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error("Error:", error));
```

En este ejemplo, `fetch` realiza una petición a la URL especificada. La respuesta (`response`) es convertida a JSON utilizando el método `.json()`, y luego los datos (`data`) son manipulados en el siguiente `.then`. Si ocurre un error durante la petición, es capturado y manejado en el bloque `.catch`.

Partes de `fetch`

- **URL**: La dirección a la que se realiza la petición.
- **Método HTTP**: Por defecto, `fetch` usa el método GET. Podemos especificar otros métodos (POST, PUT, DELETE, etc.) pasando un objeto de configuración como segundo argumento.
- **Encabezados (Headers)**: Información adicional que se envía con la petición, como tipo de contenido (Content-Type) o tokens de autenticación.
- **Cuerpo (Body)**: Datos que se envían con la petición, especialmente en métodos POST o PUT.

Ejemplo con configuración adicional:

```
fetch("https://api.example.com/data", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify({ key: "value" }),
```

```
})
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error("Error:", error));
```

Funciones Asincrónicas (async/await)

Las funciones **async/await** son una forma moderna y más legible de trabajar con promesas. Un ejemplo básico de una función asincrónica es:

```
const fetchData = async () => {
  try {
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error:", error);
  }
};
```

En este ejemplo, **await** pausa la ejecución de la función **fetchData** hasta que la promesa de **fetch** se resuelve. Si la promesa se rechaza, el error es capturado en el bloque **catch**.

Uso de **async** en **useEffect**

No podemos declarar directamente una función **async** en el argumento de **useEffect** debido a que **useEffect** espera que el retorno sea una función de limpieza o **undefined**. Las funciones **async** devuelven implícitamente una promesa, lo cual no es compatible con **useEffect**. Para resolver esto, encapsulamos la función **async** dentro de **useEffect**.

Ejemplo de **fetchApi** en **useEffect**:

```
import React, { useState, useEffect } from "react";

const fetchApi = async (setData, setLoading, setError) => {
  try {
    const response = await fetch("https://api.example.com/data");
    if (!response.ok) {
      throw new Error("Network response was not ok");
    }
    const data = await response.json();
    localStorage.setItem("apiData", JSON.stringify(data));
    setData(data);
    setLoading(false);
  } catch (error) {
    setError(error);
    setLoading(false);
  }
};
```

```

    }

};

const ApiComponent = () => {
  const [data, setData] = useState(() => {
    const cachedData = localStorage.getItem("apiData");
    return cachedData ? JSON.parse(cachedData) : null;
  });
  const [loading, setLoading] = useState(!data);
  const [error, setError] = useState(null);

  useEffect(() => {
    const getData = async () => {
      if (!data) {
        await fetchApi(setData, setLoading, setError);
      }
    };
    getData();
  }, [data]);

  const clearCache = () => {
    localStorage.removeItem("apiData");
    setData(null);
    setLoading(true);
    setError(null);
  };

  if (loading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return (
      <div>
        Error: {error.message}
        <button onClick={clearCache}>Retry</button>
      </div>
    );
  }

  return (
    <div>
      <h1>Data from API:</h1>
      <pre>{JSON.stringify(data, null, 2)}</pre>
      <button onClick={clearCache}>Clear Cache and Retry</button>
    </div>
  );
};

export default ApiComponent;

```

Creando un Custom Hook para Manejar el Cache

Para mejorar la reutilización de código y la organización, podemos crear un custom hook que se encargue de almacenar el estado en un contexto, utilizando el cache del Local Storage si está disponible. Para separar mejor el contexto, vamos a crear un archivo **DataProvider.js**.

DataProvider.js

```
import React, { createContext, useContext, useState, useEffect } from "react";

const DataContext = createContext();

const DataProvider = ({ children }) => {
  const [data, setData] = useState(() => {
    const cachedData = localStorage.getItem("apiData");
    return cachedData ? JSON.parse(cachedData) : null;
  });
  const [loading, setLoading] = useState(!data);
  const [error, setError] = useState(null);

  useEffect(() => {
    const getData = async () => {
      if (!data) {
        await fetchApi(setData, setLoading, setError);
      }
    };
    getData();
  }, [data]);
}

return (
  <DataContext.Provider value={{ data, loading, error, setData }}>
    {children}
  </DataContext.Provider>
);
};

const useData = () => {
  const context = useContext(DataContext);
  if (context === undefined) {
    throw new Error("useData must be used within a DataProvider");
  }
  return context;
};

const fetchApi = async (setData, setLoading, setError) => {
  try {
    const response = await fetch("https://api.example.com/data");
    if (!response.ok) {
      throw new Error("Network response was not ok");
    }
    const data = await response.json();
    setData(data);
    setLoading(false);
  } catch (error) {
    setError(error.message);
  }
};
```

```

localStorage.setItem("apiData", JSON.stringify(data));
setData(data);
 setLoading(false);
} catch (error) {
 setError(error);
 setLoading(false);
}
};

export { DataProvider, useData };

```

Utilizando el Custom Hook en un Componente

Ahora podemos utilizar el custom hook `useData` dentro de nuestros componentes para acceder a los datos de la API de manera eficiente.

```

import React from "react";
import { DataProvider, useData } from "./DataProvider";

const ApiComponent = () => {
  const { data, loading, error } = useData();

  const clearCache = () => {
    localStorage.removeItem("apiData");
    window.location.reload();
  };

  if (loading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return (
      <div>
        Error: {error.message}
        <button onClick={clearCache}>Retry</button>
      </div>
    );
  }
}

return (
  <div>
    <h1>Data from API:</h1>
    <pre>{JSON.stringify(data, null, 2)}</pre>
    <button onClick={clearCache}>Clear Cache and Retry</button>
  </div>
);
};

const App = () => (
  <DataProvider>

```

```
<ApiComponent />
</DataProvider>
);

export default App;
```

Importancia de la Seguridad en el Local Storage

Es fundamental recalcar que el Local Storage no es un lugar seguro para almacenar información sensible, como tokens de autenticación, contraseñas o cualquier dato personal. El Local Storage es accesible por cualquier script que se ejecute en la misma página, lo que lo hace vulnerable a ataques XSS (Cross-Site Scripting).

Por lo tanto, se recomienda almacenar en el Local Storage solo datos que no sean críticos para la seguridad de la aplicación y del usuario.

❖ Concepto de Portals

En este capítulo, vamos a explorar el concepto de Portals en React, entender qué son, cómo utilizarlos y algunos ejemplos prácticos para ilustrar su uso.

¿Qué es un Portal?

Un Portal en React es una forma de renderizar un componente hijo en un nodo del DOM diferente al de su componente padre. En lugar de seguir la estructura habitual del DOM donde los componentes hijos se renderizan dentro de sus padres, los Portals permiten montar un componente en un nodo completamente separado del árbol del DOM.

¿Por qué usar Portals?

Los Portals son útiles en situaciones donde necesitamos que un componente se renderice fuera del flujo normal del DOM, como en los siguientes casos:

- **Modales y Diálogos:** Aseguran que el modal se superponga correctamente sobre otros contenidos.
- **Tooltips y Popovers:** Facilitan la gestión de superposiciones y posicionamientos dinámicos.
- **Menús Contextuales:** Evitan problemas de z-index y de overflow de los padres.

Cómo Utilizar un Portal

Para crear un Portal, utilizamos la función `createPortal` del módulo `react-dom`. Esta función toma dos argumentos:

- El contenido que queremos renderizar.
- El nodo del DOM donde queremos que se monte el contenido.

Ejemplo básico:

```
import React from "react";
import ReactDOM from "react-dom";

const portalRoot = document.getElementById("portal-root");

const Modal = ({ children }) => {
  return ReactDOM.createPortal(
    <div className="modal">{children}</div>,
    portalRoot,
  );
};

const App = () => {
  const [showModal, setShowModal] = React.useState(false);

  const toggleModal = () => {
    setShowModal(!showModal);
  };

  return (
    <div>
      <button onClick={toggleModal}>Toggle Modal</button>
      {showModal && (
        <Modal>
          <h1>Modal Content</h1>
          <button onClick={toggleModal}>Close</button>
        </Modal>
      )}
    </div>
  );
};

export default App;
```

En este ejemplo, el componente **Modal** se renderiza dentro del nodo **portal-root**, que está fuera del nodo del componente **App**.

Crear un Nodo de Portal en el DOM

Para que el ejemplo anterior funcione, debemos asegurarnos de tener un nodo en nuestro HTML donde el Portal se monte. Generalmente, se agrega en el archivo **index.html**.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>React App</title>
  </head>
  <body>
    <div id="root"></div>
    <div id="portal-root"></div>
    <script src="bundle.js"></script>
  </body>
</html>
```

Ejemplos Prácticos

Ejemplo 1: Modal Básico

Vamos a crear un modal básico utilizando Portals. Este modal contendrá contenido sencillo y un botón para cerrarlo.

```
import React from "react";
import ReactDOM from "react-dom";
import "./Modal.css";

const portalRoot = document.getElementById("portal-root");

const Modal = ({ children, onClose }) => {
  return ReactDOM.createPortal(
    <div className="modal-overlay">
      <div className="modal-content">
        {children}
        <button onClick={onClose}>Close</button>
      </div>
    </div>, 
    portalRoot,
  );
};

const App = () => {
  const [showModal, setShowModal] = React.useState(false);

  const toggleModal = () => {
    setShowModal(!showModal);
  };

  return (
    <div>
      <button onClick={toggleModal}>Open Modal</button>
      {showModal && (
        <Modal>
          <p>This is a simple modal!</p>
        </Modal>
      )}
    </div>
  );
};

ReactDOM.render(<App />, root);
```

```

        <Modal onClose={toggleModal}>
          <h1>This is a modal</h1>
          <p>Content inside the modal goes here.</p>
        </Modal>
      )}
    </div>
  );
};

export default App;

```

Ejemplo 2: Tooltip

Vamos a crear un tooltip que se muestra al pasar el cursor sobre un botón. El tooltip se renderiza utilizando Portals para asegurar su correcta posición y visibilidad.

```

import React from "react";
import ReactDOM from "react-dom";
import "./Tooltip.css";

const portalRoot = document.getElementById("portal-root");

const Tooltip = ({ children, position }) => {
  return ReactDOM.createPortal(
    <div className="tooltip" style={{ top: position.top, left: position.left }}>
      {children}
    </div>,
    portalRoot,
  );
};

const App = () => {
  const [tooltip, setTooltip] = React.useState({
    visible: false,
    position: { top: 0, left: 0 },
    content: "",
  });

  const showTooltip = (event) => {
    const { top, left } = event.target.getBoundingClientRect();
    setTooltip({
      visible: true,
      position: { top: top + window.scrollY, left: left + window.scrollX },
      content: "This is a tooltip",
    });
  };

  const hideTooltip = () => {
    setTooltip({ visible: false, position: { top: 0, left: 0 }, content: "" });
  };
}


```

```

return (
  <div>
    <button onMouseEnter={showTooltip} onMouseLeave={hideTooltip}>
      Hover me
    </button>
    {tooltip.visible && (
      <Tooltip position={tooltip.position}>{tooltip.content}</Tooltip>
    )}
  </div>
);
};

export default App;

```

Ejemplo 3: Menú Contextual

Crearemos un menú contextual que aparece al hacer clic derecho en un área específica de la pantalla.

```

import React from "react";
import ReactDOM from "react-dom";
import "./ContextMenu.css";

const portalRoot = document.getElementById("portal-root");

const ContextMenu = ({ position, onClose }) => {
  return ReactDOM.createPortal(
    <div
      className="context-menu"
      style={{ top: position.top, left: position.left }}
    >
      <ul>
        <li onClick={onClose}>Option 1</li>
        <li onClick={onClose}>Option 2</li>
        <li onClick={onClose}>Option 3</li>
      </ul>
    </div>,
    portalRoot,
  );
};

const App = () => {
  const [contextMenu, setContextMenu] = React.useState({
    visible: false,
    position: { top: 0, left: 0 },
  });

  const handleContextMenu = (event) => {
    event.preventDefault();
    const { top, left } = event.target.getBoundingClientRect();
    setContextMenu({

```

```

        visible: true,
        position: { top: top + window.scrollY, left: left + window.scrollX },
    });
};

const closeContextMenu = () => {
    setContextMenu({ visible: false, position: { top: 0, left: 0 } });
};

return (
    <div
        onContextMenu={handleContextMenu}
        style={{ width: "100vw", height: "100vh" }}
    >
        Right-click anywhere in this area.
        {contextMenu.visible && (
            <ContextMenu
                position={contextMenu.position}
                onClose={closeContextMenu}
            />
        )}
    </div>
);
};

export default App;

```

~~ Cómo Agregar Estilos a Componentes

En este capítulo, exploraremos cómo agregar estilos a los componentes de React utilizando varias técnicas, incluyendo CSS tradicional, CSS-in-JS, y SCSS modules. Cada enfoque tiene sus propias ventajas y desventajas, y elegir el adecuado dependerá de las necesidades específicas de tu proyecto.

Estilos CSS Tradicionales

Paso 1: Creación del Objeto `styles`

En lugar de utilizar archivos CSS externos, vamos a definir nuestros estilos directamente dentro del componente usando un objeto `styles`. Este enfoque es útil para estilos locales y se integra fácilmente con JSX.

```

// Button.js
import React from "react";

const styles = {
    button: {
        backgroundColor: "#007bff",

```

```

color: "white",
border: "none",
padding: "10px 20px",
borderRadius: "4px",
cursor: "pointer",
transition: "background-color 0.3s ease",
},
buttonHover: {
  backgroundColor: "#0056b3",
},
};

const Button = ({ children, onClick }) => {
  return (
    <button
      style={styles.button}
      onMouseOver={(e) =>
        (e.currentTarget.style.backgroundColor =
          styles.buttonHover.backgroundColor)
      }
      onMouseOut={(e) =>
        (e.currentTarget.style.backgroundColor = styles.button.backgroundColor)
      }
      onClick={onClick}
    >
      {children}
    </button>
  );
};

export default Button;

```

Ventajas y Desventajas

- **Ventajas:** Fácil de entender y utilizar, excelente para proyectos pequeños.
- **Desventajas:** Puede volverse difícil de mantener en proyectos grandes debido a la falta de encapsulación y posibilidad de conflictos de nombres.

CSS-in-JS con Styled Components

Paso 1: Instalación de Styled Components

Primero, instalamos la librería **styled-components**.

```
npm install styled-components
```

Paso 2: Crear Componentes Estilizados

Creamos nuestros componentes con estilos integrados utilizando **styled-components**.

```
// Button.js
import React from "react";
import styled from "styled-components";

const StyledButton = styled.button`
background-color: #007bff;
color: white;
border: none;
padding: 10px 20px;
border-radius: 4px;
cursor: pointer;
transition: background-color 0.3s ease;

&:hover {
  background-color: #0056b3;
}
`;

const Button = ({ children, onClick }) => {
  return <StyledButton onClick={onClick}>{children}</StyledButton>;
};

export default Button;
```

Ventajas y Desventajas

- **Ventajas:** Estilos encapsulados, soporte para temas, excelente para componentes reutilizables.
- **Desventajas:** Puede aumentar el tamaño del paquete, la sintaxis puede ser menos familiar para algunos desarrolladores, y se realiza en tiempo de ejecución (runtime), lo que puede afectar el rendimiento.

Soluciones Alternativas

Aunque **styled-components** realiza el procesamiento en tiempo de ejecución, existen otras soluciones como **panda css** que realizan el procesamiento en tiempo de build, eliminando este problema de rendimiento.

SCSS Modules

Paso 1: Instalación de SASS

Instalamos sass para permitir el uso de SCSS en nuestro proyecto.

```
npm install sass
```

Paso 2: Creación del Archivo SCSS Module

Creamos un archivo SCSS para nuestro componente. Utilizando SCSS modules, podemos asegurarnos de que los estilos son locales al componente.

```
/* Button.module.scss */
.button {
  background-color: #007bff;
  color: white;
  border: none;
  padding: 10px 20px;
  border-radius: 4px;
  cursor: pointer;
  transition: background-color 0.3s ease;

  &:hover {
    background-color: #0056b3;
  }
}
```

Paso 3: Importar el SCSS Module en el Componente

Importamos el SCSS module en nuestro componente y aplicamos los estilos.

```
// Button.js
import React from "react";
import styles from "./Button.module.scss";

const Button = ({ children, onClick }) => {
  return (
    <button className={styles.button} onClick={onClick}>
      {children}
    </button>
  );
};

export default Button;
```

Ventajas y Desventajas

- **Ventajas:** Estilos locales y encapsulados, soporte completo para SCSS.
- **Desventajas:** Configuración adicional, puede ser complejo para proyectos muy grandes.

Comparación de Enfoques

CSS Tradicional

- **Uso:** Ideal para proyectos pequeños y rápidos.
- **Simplicidad:** Alta.
- **Mantenimiento:** Puede ser difícil en proyectos grandes debido a la falta de encapsulación.

CSS-in-JS (Styled Components)

- **Uso:** Ideal para componentes altamente reutilizables y proyectos que utilizan mucho JavaScript.
- **Simplicidad:** Media, la sintaxis puede ser un obstáculo para algunos.
- **Mantenimiento:** Alto, gracias a la encapsulación y la capacidad de usar temas.
- **Rendimiento:** Puede ser menos eficiente debido a que se procesa en tiempo de ejecución. Alternativas como `panda css` pueden mitigar este problema al realizar el procesamiento en tiempo de build.

SCSS Modules

- **Uso:** Ideal para proyectos que requieren un fuerte uso de SCSS y encapsulación de estilos.
- **Simplicidad:** Media-Alta, familiaridad con SCSS es un plus.
- **Mantenimiento:** Alto, estilos locales y potentes características de SCSS.

~~ Routing con react-router-dom

Introducción a React Router

En este capítulo, vamos a explorar cómo manejar el routing en una aplicación de React utilizando `react-router-dom`. El enrutamiento (routing) es el proceso de definir y manejar las distintas rutas dentro de una aplicación web. Esto permite que los usuarios puedan navegar entre diferentes vistas o páginas sin necesidad de recargar toda la aplicación.

`react-router-dom` es una librería popular en el ecosistema de React para implementar routing. Facilita la creación de rutas, la navegación entre ellas y la protección de rutas basadas en la autenticación y otros factores.

Conceptos Básicos de React Router

Antes de adentrarnos en los ejemplos, repasemos algunos conceptos clave que necesitamos conocer sobre `react-router-dom`:

- **Router**: Es el contenedor que envuelve la aplicación y maneja la historia de navegación. Utilizamos `BrowserRouter` para una aplicación web estándar.
- **Routes**: Son definiciones de rutas que especifican qué componente se debe renderizar para una URL particular.
- **Route**: Es un componente que se usa para definir una ruta. Cada `Route` se asocia a una URL y a un componente específico.
- **Navigate**: Es un componente utilizado para redirigir programáticamente al usuario a una nueva ruta.
- **Private Routes**: Rutas que solo pueden ser accedidas si se cumplen ciertas condiciones, como estar autenticado.

Instalación de `react-router-dom`

Antes de empezar, necesitamos instalar `react-router-dom` en nuestra aplicación. Podemos hacerlo utilizando `npm` o `yarn`:

```
npm install react-router-dom
```

o

```
yarn add react-router-dom
```

Configuración Básica del Router

Para configurar el router en nuestra aplicación, primero envolvemos nuestra aplicación en un `BrowserRouter` y definimos las rutas utilizando el componente `Routes` y `Route`.

```
import React from "react";
import {
  BrowserRouter as Router,
  Routes,
  Route,
  Navigate,
} from "react-router-dom";
import Login from "./pages/Login";
import Dashboard from "./pages/Dashboard";

const App = () => {
  return (
    <Router>
```

```

<Routes>
  <Route path="/login" element={<Login />} />
  <Route path="/dashboard" element={<Dashboard />} />
  <Route path="/" element={<Navigate to="/login" replace />} />
</Routes>
</Router>
);
};

export default App;

```

En este ejemplo, hemos definido tres rutas:

- **/login** para el componente **Login**.
- **/dashboard** para el componente **Dashboard**.
- ***** para redirigir cualquier ruta no definida a **/login**.

Rutas Privadas

Las rutas privadas son rutas que solo pueden ser accedidas si el usuario está autenticado. Para manejar esto, podemos crear un componente **PrivateRoute** que verifique si el usuario está autenticado antes de permitirle el acceso a la ruta.

```

// PrivateRoute.js
import React from "react";
import { Navigate } from "react-router-dom";
import { useUserContext } from "../UserContext";

const PrivateRoute = ({ children }) => {
  const { user } = useUserContext();
  return user.id ? children : <Navigate to="/login" replace />;
};

export default PrivateRoute;

```

Utilizamos el contexto **UserContext** para verificar si el usuario está autenticado. Si no lo está, redirigimos al usuario a la página de login.

Uso del Contexto para Manejar la Autenticación

En lugar de Redux, utilizaremos el Context API de React para manejar el estado de la autenticación del usuario.

```

// UserContext.js
import { createContext, useContext, useState, useEffect } from "react";

```

```

export const UserContext = createContext();

export const UserProvider = ({ children }) => {
  const [user, setUser] = useState(() => {
    const storedUser = localStorage.getItem("user");
    return storedUser
      ? JSON.parse(storedUser)
      : { id: null, name: "", email: "", role: "user" };
  });

  const login = (userData) => {
    setUser(userData);
    localStorage.setItem("user", JSON.stringify(userData));
  };

  const logout = () => {
    setUser({ id: null, name: "", email: "", role: "user" });
    localStorage.removeItem("user");
  };

  return (
    <UserContext.Provider value={{ user, login, logout }}>
      {children}
    </UserContext.Provider>
  );
};

export const useUserContext = () => {
  const context = useContext(UserContext);
  if (context === undefined) {
    throw new Error("useUserContext must be used within a UserProvider");
  }
  return context;
};

```

Implementación Completa del Ejemplo

Finalmente, vamos a integrar todos estos componentes en nuestra aplicación principal y a manejar la autenticación y las rutas protegidas utilizando `react-router-dom` y el Context API.

```

// App.js
import React from "react";
import {
  BrowserRouter as Router,
  Routes,
  Route,
  Navigate,
} from "react-router-dom";
import { UserProvider, useUserContext } from "./UserContext";

```

```

import Login from "./pages/Login";
import Dashboard from "./pages/Dashboard";
import PrivateRoute from "./components/PrivateRoute";
import RoleRoute from "./components/RoleRoute";

const App = () => {
  return (
    <UserProvider>
      <Router>
        <Routes>
          <Route path="/login" element={<Login />} />
          <Route
            path="/dashboard"
            element={
              <PrivateRoute>
                <Dashboard />
              </PrivateRoute>
            }
          />
          <Route
            path="/admin"
            element={
              <RoleRoute requiredRole="admin">
                <AdminPage />
              </RoleRoute>
            }
          />
          <Route path="/" element={<Navigate to="/login" replace />} />
        </Routes>
      </Router>
    </UserProvider>
  );
};

export default App;

```

Routing Anidado y Lazy Loading

Ventajas de las Rutas Anidadas

Las rutas anidadas permiten organizar mejor las secciones de la aplicación, facilitando el mantenimiento y la escalabilidad. Con rutas anidadas, podemos definir rutas dentro de otras rutas, permitiendo que una sección de la aplicación tenga su propio conjunto de rutas.

Configuración Básica de Rutas Anidadadas

Supongamos que tenemos una aplicación con un dashboard que tiene múltiples secciones como Home, Profile y Settings. Vamos a definir estas rutas anidadas en nuestro componente principal.

```

import React from "react";
import {
  BrowserRouter as Router,
  Routes,
  Route,
  Navigate,
  Outlet,
} from "react-router-dom";
import Login from "./pages/Login";
import Dashboard from "./pages/Dashboard";
import Home from "./pages/Home";
import Profile from "./pages/Profile";
import Settings from "./pages/Settings";
import PrivateRoute from "./components/PrivateRoute";
import RoleRoute from "./components/RoleRoute";

const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/login" element={<Login />} />
        <Route
          path="/dashboard"
          element={
            <PrivateRoute>
              <Dashboard />
            </PrivateRoute>
          }
        >
          <Route path="home" element={<Home />} />
          <Route path="profile" element={<Profile />} />
          <Route path="settings" element={<Settings />} />
        </Route>
        <Route path="/" element={<Navigate to="/login" replace />} />
      </Routes>
    </Router>
  );
};

export default App;

```

En este ejemplo, hemos definido rutas anidadas dentro de `/dashboard`. Las rutas `home`, `profile`, y `settings` son accesibles solo cuando el usuario está en `/dashboard`.

Utilizando Lazy Loading

Lazy loading es una técnica para cargar componentes solo cuando son necesarios, en lugar de cargarlos todos al inicio. Esto mejora el rendimiento de la aplicación, especialmente si tiene muchas rutas o componentes pesados.

Para implementar lazy loading, usamos la función `React.lazy()` y el componente `Suspense` de React.

```
import React, { Suspense, lazy } from "react";
import {
  BrowserRouter as Router,
  Routes,
  Route,
  Navigate,
} from "react-router-dom";
import Login from "./pages/Login";
import PrivateRoute from "./components/PrivateRoute";

const Dashboard = lazy(() => import("./pages/Dashboard"));
const Home = lazy(() => import("./pages/Home"));
const Profile = lazy(() => import("./pages/Profile"));
const Settings = lazy(() => import("./pages/Settings"));

const App = () => {
  return (
    <Router>
      <Suspense fallback={<div>Loading...</div>}>
        <Routes>
          <Route path="/login" element={<Login />} />
          <Route
            path="/dashboard"
            element={
              <PrivateRoute>
                <Dashboard />
              </PrivateRoute>
            }
          >
            <Route path="home" element={<Home />} />
            <Route path="profile" element={<Profile />} />
            <Route path="settings" element={<Settings />} />
          </Route>
          <Route path="/" element={<Navigate to="/login" replace />} />
        </Routes>
      </Suspense>
    </Router>
  );
};

export default App;
```

Aquí hemos envuelto nuestras rutas con el componente `Suspense` y especificado un componente `fallback` que se muestra mientras los componentes anidados se cargan.

Alcance Lógico de los Elementos

La estructura de rutas anidadas nos permite definir el alcance lógico de los elementos de forma clara. Cada sección de la aplicación puede tener su propio conjunto de rutas, lo que facilita la comprensión y el mantenimiento del código.

- **Rutas Principales:** Las rutas principales como `/login` y `/dashboard` se definen en el nivel superior.
- **Rutas Secundarias:** Dentro del `Dashboard`, definimos rutas secundarias como `home`, `profile` y `settings`.

Esta organización jerárquica ayuda a mantener un código limpio y modular. Cada sección de la aplicación está encapsulada dentro de su propio ámbito lógico, lo que hace más fácil entender cómo se estructuran y cargan los componentes.

Ejemplo Completo

Vamos a mostrar un ejemplo completo que combina rutas anidadas, lazy loading y el manejo de autenticación utilizando Context API.

```
// App.js
import React, { Suspense, lazy } from "react";
import {
  BrowserRouter as Router,
  Routes,
  Route,
  Navigate,
} from "react-router-dom";
import { UserProvider } from "./UserContext";
import Login from "./pages/Login";
import PrivateRoute from "./components/PrivateRoute";

const Dashboard = lazy(() => import("./pages/Dashboard"));
const Home = lazy(() => import("./pages/Home"));
const Profile = lazy(() => import("./pages/Profile"));
const Settings = lazy(() => import("./pages/Settings"));

const App = () => {
  return (
    <UserProvider>
      <Router>
        <Suspense fallback={<div>Loading...</div>}>
          <Routes>
            <Route path="/login" element={<Login />} />
            <Route
              path="/dashboard"
              element={
                <PrivateRoute>
                  <Dashboard />
                </PrivateRoute>
              }
            />
          </Routes>
        </Suspense>
      </Router>
    </UserProvider>
  );
}

export default App;
```

```
        }
      >
      <Route path="home" element={<Home />} />
      <Route path="profile" element={<Profile />} />
      <Route path="settings" element={<Settings />} />
    </Route>
    <Route path="*" element={<Navigate to="/login" replace />} />
  </Routes>
</Suspense>
</Router>
</UserProvider>
);
};

export default App;
```

En este ejemplo:

- Utilizamos `React.lazy()` para cargar los componentes de manera perezosa.
- `Suspense` envuelve nuestras rutas para mostrar un fallback mientras los componentes se cargan.
- `PrivateRoute` asegura que solo los usuarios autenticados puedan acceder a las rutas anidadas bajo `/dashboard`.

Esta estructura permite una carga eficiente y una organización clara del código, facilitando la navegación y la experiencia del usuario en la aplicación.

❖ Control de Errores con Error Boundaries

Introducción

En este capítulo, aprenderemos a manejar errores de manera eficiente en nuestras aplicaciones React utilizando Error Boundaries. Esta técnica nos permitirá capturar errores en componentes específicos sin afectar el resto de la aplicación. Vamos a explorar cómo implementar Error Boundaries y cómo podemos manejarlos para asegurar que nuestra aplicación continúe funcionando de manera estable, incluso cuando ocurren errores.

¿Qué es un Error Boundary?

Un Error Boundary es un componente de React que detecta errores en cualquier componente hijo durante el ciclo de renderizado. Similar a cómo un Provider engloba otros componentes para proporcionarles un contexto, un Error Boundary engloba otros componentes para manejar los errores que puedan surgir.

La idea es que si un componente dentro del Error Boundary falla, solo esa parte específica de la aplicación se verá afectada. El Error Boundary puede mostrar un mensaje de error o una UI de respaldo, sin detener el funcionamiento del resto de la aplicación.

Implementación Básica de un Error Boundary

Para empezar, vamos a implementar un Error Boundary básico. Usaremos clases, ya que actualmente, los Error Boundaries solo pueden ser implementados en componentes de clase.

```
import React, { Component } from "react";

class ErrorBoundary extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    console.log(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Oops! Algo salió mal.</h1>;
    }

    return this.props.children;
  }
}

export default ErrorBoundary;
```

En este código:

- **getDerivedStateFromError:** Este método se llama cuando un error es lanzado en un componente hijo. Actualiza el estado para indicar que ha ocurrido un error.
- **componentDidCatch:** Este método se usa para registrar errores o realizar tareas adicionales.
- **render:** Si hay un error, se muestra un mensaje de error; de lo contrario, se renderizan los componentes hijos normalmente.

Usando el Error Boundary

Para usar el Error Boundary, simplemente envolvemos los componentes que queremos monitorear con nuestro Error Boundary.

```

import React from "react";
import ReactDOM from "react-dom";
import App from "./App";
import ErrorBoundary from "./ErrorBoundary";

ReactDOM.render(
  <ErrorBoundary>
    <App />
  </ErrorBoundary>,
  document.getElementById("root"),
);

```

De esta forma, cualquier error que ocurra dentro de App será capturado por ErrorBoundary, y en lugar de romper toda la aplicación, solo mostrará el mensaje de error.

Manejo de Errores en Fetch Requests

Los Error Boundaries no capturan errores asíncronos, como los que ocurren en solicitudes fetch. Para manejar estos errores, necesitamos usar un enfoque diferente.

```

import React, { useState, useEffect } from "react";

const DataFetcher = () => {
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch("https://rickandmortyapi.com/api/character")
      .then((response) => {
        if (!response.ok) {
          throw new Error("Network response was not ok");
        }
        return response.json();
      })
      .then((data) => setData(data))
      .catch((error) => setError(error));
  }, []);

  if (error) {
    return <div>Oops! Algo salió mal: {error.message}</div>;
  }

  return (
    <div>
      {data ? (
        data.results.map((character) => (
          <div key={character.id}>{character.name}</div>
        ))
      ) : (
        <div>No se encontraron resultados.</div>
      )}
    </div>
  );
}

export default DataFetcher;

```

```

        <div>Loading...</div>
    )}
</div>
);
};

export default DataFetcher;

```

En este componente:

- `useEffect` se usa para hacer la solicitud `fetch`.
- Si ocurre un error durante la solicitud, se establece el estado de error y se muestra un mensaje de error.

Integración de Error Boundaries con Fetch Requests

Podemos combinar los Error Boundaries con el manejo de errores de `fetch` para una solución más robusta.

```

import React, { Component } from "react";

class ErrorBoundary extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    console.log(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Oops! Algo salió mal.</h1>;
    }

    return this.props.children;
  }
}

const DataFetcher = () => {
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch("https://rickandmortyapi.com/api/character")

```

```

        .then((response) => {
          if (!response.ok) {
            throw new Error("Network response was not ok");
          }
          return response.json();
        })
        .then((data) => setData(data))
        .catch((error) => setError(error));
      }, []);
    }

    if (error) {
      throw error;
    }

    return (
      <div>
        {data ? (
          data.results.map((character) => (
            <div key={character.id}>{character.name}</div>
          ))
        ) : (
          <div>Loading...</div>
        )}
      </div>
    );
  };

const App = () => (
  <ErrorBoundary>
    <DataFetcher />
  </ErrorBoundary>
);

export default App;

```

En este ejemplo, cualquier error durante el fetch lanzará una excepción que será capturada por el Error Boundary, asegurando que la UI de respaldo se muestre cuando sea necesario.

Beneficios del Lazy Loading

El lazy loading, o carga perezosa, nos permite cargar componentes solo cuando se necesitan. Esto puede mejorar el rendimiento de nuestra aplicación al reducir el tiempo de carga inicial y el uso de memoria.

```

import React, { Suspense, lazy } from "react";

const LazyComponent = lazy(() => import("./LazyComponent"));

const App = () => (
  <Suspense fallback={<div>Loading...</div>}>
    <LazyComponent />
  </Suspense>
);

```

```
</Suspense>
);

export default App;
```

En este ejemplo, `LazyComponent` solo se cargará cuando se necesite, y mientras tanto, se mostrará el mensaje de "Loading...".

Integrando Lazy Loading y Error Boundaries

Combinar lazy loading con Error Boundaries puede ofrecer una solución muy eficiente y robusta.

```
import React, { Suspense, lazy } from "react";
import ErrorBoundary from "./ErrorBoundary";

const LazyComponent = lazy(() => import("./LazyComponent"));

const App = () => (
  <ErrorBoundary>
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
  </ErrorBoundary>
);

export default App;
```

De esta manera, manejamos tanto errores como la carga perezosa de componentes, asegurando una experiencia de usuario fluida y sin interrupciones.

❖ Mejorando tus Habilidades con Axios Interceptor

Introducción

En este capítulo, vamos a adentrarnos en el mundo de los interceptores de Axios en React. Aprenderemos cómo utilizarlos para manejar de manera eficiente las peticiones HTTP y las respuestas, dándonos la flexibilidad y el control que necesitamos para construir aplicaciones robustas y seguras. Vamos a crear un proyecto desde cero, configurar Axios e implementar interceptores para manejar autenticaciones, errores y más.

¿Qué es Axios y por qué usarlo?

Axios es una popular librería para hacer peticiones HTTP en JavaScript. Facilita la comunicación con APIs, ofreciendo una sintaxis limpia y numerosas funcionalidades adicionales que Fetch, la API nativa de JavaScript, no proporciona de manera tan directa. Una de estas funcionalidades clave es el uso de interceptores, que nos permiten interceptar y modificar peticiones y respuestas antes de que lleguen al servidor o al cliente.

Creando un Proyecto con Vite

Vamos a comenzar creando un nuevo proyecto de React utilizando Vite, un bundler rápido y ligero.

- **Crear el Proyecto:**

```
npm create vite@latest
```

Elije un nombre para tu proyecto y selecciona "React" y "TypeScript" como las opciones deseadas.

- **Instalar Axios:**

```
npm install axios
```

Configurando Axios Interceptor

Ahora que tenemos nuestro proyecto configurado, vamos a crear un interceptor para manejar nuestras peticiones y respuestas.

- **Crear el Interceptor:** Crea un archivo `axios.interceptor.ts` en una carpeta `services`.

```
import axios from "axios";

const axiosInstance = axios.create();

axiosInstance.interceptors.request.use(
  (config) => {
    // Modificar la petición antes de enviarla
    const token = localStorage.getItem("token");
    if (token) {
      config.headers["Authorization"] = `Bearer ${token}`;
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  },
);

axiosInstance.interceptors.response.use(
  (response) => {
    // Modificar la respuesta antes de enviarla al cliente
    return response;
  },
  (error) => {
```

```

    if (error.response.status === 401) {
      // Manejar errores de autenticación
      console.log("No autorizado, redirigiendo al login...");
    }
    return Promise.reject(error);
  },
);

export default axiosInstance;

```

En este código, interceptamos todas las peticiones para añadir un token de autenticación si está disponible, y manejamos errores de respuesta específicos, como una autenticación fallida.

- **Utilizar el Interceptor:** En tu componente principal (App.tsx), importa y utiliza el interceptor para realizar una petición.

```

import React, { useEffect, useState } from "react";
import axiosInstance from "./services/axios.interceptor";

const App = () => {
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axiosInstance.get(
          "https://rickandmortyapi.com/api/character/1",
        );
        setData(response.data);
      } catch (error) {
        setError(error);
      }
    };

    fetchData();
  }, []);

  if (error) return <div>Error: {error.message}</div>;
  if (!data) return <div>Loading...</div>;

  return (
    <div>
      <h1>{data.name}</h1>
      <p>Status: {data.status}</p>
    </div>
  );
};

export default App;

```

Ventajas de Usar Interceptores

- **Autenticación Centralizada:** Los interceptores nos permiten añadir headers de autenticación a todas las peticiones desde un único lugar.
- **Manejo de Errores:** Podemos capturar y manejar errores de manera global, mostrando mensajes personalizados o redirigiendo al usuario según sea necesario.
- **Logging y Depuración:** Podemos registrar las peticiones y respuestas para monitorear la actividad de la red y depurar problemas.

Manejo Avanzado de Errores

Vamos a mejorar nuestro interceptor para manejar diferentes tipos de errores y refrescar el token de autenticación cuando sea necesario.

```
axiosInstance.interceptors.response.use(
  (response) => {
    return response;
  },
  async (error) => {
    if (error.response.status === 401) {
      // Intentar refrescar el token
      try {
        const refreshToken = localStorage.getItem("refreshToken");
        const response = await axios.post("/auth/refresh-token", {
          token: refreshToken,
        });
        localStorage.setItem("token", response.data.token);
        error.config.headers["Authorization"] = `Bearer ${response.data.token}`;
        return axiosInstance(error.config);
      } catch (e) {
        // Redirigir al login si el refresco del token falla
        console.log("Redirigiendo al login...");
      }
    }
    return Promise.reject(error);
  );
};
```

Con esta configuración, si una petición recibe un error 401 (No autorizado), intentamos refrescar el token. Si el refresco falla, redirigimos al usuario a la página de login.

TypeScript Con De Tuti

~ Introducción

¡Hola a todos! Aquí Gentleman al teclado, trayéndoles un análisis pormenorizado sobre TypeScript y cómo este lenguaje puede revolucionar la forma en que trabajamos en equipos de desarrollo. Este libro es una expansión de un vídeo que subí a YouTube, donde hablamos de las bases de TypeScript, las ventajas y cómo puede ayudar en un equipo. Vamos a sumergirnos no solo en el contenido del vídeo, sino que ampliaremos con ejemplos prácticos, código y reflexiones clave para que ustedes, mis queridos desarrolladores, puedan llevar su código a un nivel superior.

~ ¿Por Qué TypeScript?

¿Qué es TypeScript?

TypeScript es un "superset" de JavaScript, lo que significa que tiene todo lo que JavaScript ofrece, pero añade más funcionalidades que son especialmente útiles en proyectos grandes o en equipos. Como mencioné en el vídeo, si JavaScript es bueno, TypeScript es JavaScript con esteroides.

```
// Ejemplo básico de TypeScript
let mensaje: string = "¡Hola, TypeScript!";
console.log(mensaje);
```

Ventajas de Usar TypeScript en Equipos

- **Seguridad de Tipos:** Reduce los errores comunes en JavaScript permitiendo especificar tipos de variables.
- **Mantenibilidad:** El código es más fácil de entender y mantener.
- **Refactorización:** Segura y fácil de realizar gracias al sistema de tipos.

~ Conceptos Fundamentales de TypeScript

Variables y Tipos

Uno de los pilares de TypeScript es su capacidad de tipado. Esto evita muchos errores comunes en JavaScript.

```
// Ejemplo de tipado en TypeScript
let esActivo: boolean = true;
```

```
let cantidad: number = 123;
```

Interfaces y Clases

TypeScript permite definir interfaces y clases, lo que facilita la implementación de patrones de diseño avanzados y la organización del código.

```
interface Usuario {
    nombre: string;
    edad: number;
}

class Empleado implements Usuario {
    constructor(
        public nombre: string,
        public edad: number,
    ) {}
}
```

~~ TypeScript en la Práctica

Análisis de Casos Prácticos

Vamos a analizar el segmento del vídeo donde discutimos la mutabilidad de las variables y cómo TypeScript puede ayudar a controlarla.

```
// Ejemplo de inmutabilidad en TypeScript
let x: number = 10;
// x = "Cambio de tipo"; // Esto generará un error en TypeScript.
```

Creando Métodos Efectivos

Discutimos también cómo la falta de claridad en los tipos puede llevar a errores en métodos que parecen simples.

```
function suma(a: number, b: number): number {
    return a + b;
}
```

~~ Mejores Prácticas y Patrones

Trabajando con Equipos

- **Claridad:** Usa tipos siempre.
- **Documentación:** Aprovecha las características de TypeScript para documentar el código.
- **Revisión de Código:** Fomenta las revisiones de código que se centren en la mejora del tipado.

Herramientas y Extensiones

Hablaremos de herramientas que pueden integrarse con TypeScript para mejorar aún más el flujo de trabajo, como linters, formateadores de código, y más.

❖ ¿Qué es Transpilar?

Transpilar, en el mundo de la programación, es el proceso de convertir código escrito en un lenguaje (o versión de un lenguaje) a otro lenguaje (o versión de ese lenguaje). En nuestro caso, a menudo hablamos de convertir TypeScript a JavaScript. Básicamente, transpilar es como traducir.

¿Por Qué Necesitamos Transpilar?

TypeScript es increíble porque nos da superpoderes: tipos, interfaces, y un montón de ayudas para evitar errores. Pero los navegadores y Node.js no entienden TypeScript, ellos solo hablan JavaScript. Entonces, necesitamos un traductor, y ese traductor es el compilador de TypeScript (tsc).

Ejemplo Práctico

Vamos a ver esto en acción con un ejemplo sencillo. Imaginemos que tenemos un archivo TypeScript `script.ts` con el siguiente código:

```
// script.ts
let mensaje: string = "Hola, mundo";
console.log(mensaje);
```

Este archivo contiene una variable `mensaje` de tipo `string` y un `console.log` para mostrarla. Ahora, para que nuestro navegador entienda este código, necesitamos transpilarlo a JavaScript. Esto lo hacemos con el comando `tsc` (TypeScript Compiler):

```
tsc script.ts
```

Después de ejecutar este comando, obtenemos un archivo `script.js` con el siguiente contenido:

```
// script.js
var mensaje = "Hola, mundo";
```

```
console.log(mensaje);
```

Como podés ver, el compilador de TypeScript ha convertido (o "transpilado") el código TypeScript a JavaScript.

Un Poco Más de Magia: Configuración del Compilador

Podemos hacer mucho más con la configuración de nuestro compilador TypeScript. Por ejemplo, podemos definir cómo queremos que se comporte el proceso de transpilar mediante un archivo `tsconfig.json`. Aquí te dejo un ejemplo básico:

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "es6", // Indica a qué versión de JavaScript queremos transpilar
    "outDir": "./dist", // Carpeta donde se guardarán los archivos transpilados
    "strict": true // Activa todas las comprobaciones estrictas
  },
  "include": ["src/**/*.ts"], // Archivos a incluir en la transpilation
  "exclude": ["node_modules"] // Archivos o carpetas a excluir
}
```

Haciendo Magia con `tsc --watch`

Si querés llevar tu flujo de trabajo al siguiente nivel, podés usar el comando `tsc --watch`, que va a estar atento a cualquier cambio en tus archivos TypeScript y automáticamente los va a transpilar a JavaScript. Es como tener un asistente personal que siempre está atento para ayudarte.

```
tsc --watch
```

En Resumen

Transpilar es un proceso esencial que nos permite escribir en lenguajes modernos y con mejores características, y luego convertir ese código a un lenguaje que los navegadores y Node.js pueden entender. Es como tener un traductor que convierte nuestras palabras en algo que todo el mundo puede entender.

Así que la próxima vez que escuches "transpilar", ya sabés que no es más que un proceso de traducción, asegurando que nuestras genialidades en TypeScript lleguen intactas y claras a cualquier entorno JavaScript.

~~ Entendiendo el Tipado en JavaScript y TypeScript

JavaScript: Un Lenguaje con Tipado Dinámico

Aunque a simple vista no lo parezca, JavaScript sí cuenta con un sistema de tipos, pero es dinámico. Esto significa que el tipo de una variable puede cambiar a lo largo de la ejecución del programa, lo que introduce flexibilidad pero también cierta propensión a errores difíciles de rastrear. Aquí es donde JavaScript muestra tanto su flexibilidad como sus limitaciones, ya que esta característica puede llevar a confusiones en proyectos grandes o cuando se trabaja en equipo.

El motor V8 de JavaScript, que es el que utilizan la mayoría de los navegadores modernos como Chrome y Node.js, maneja el tipado dinámico de una manera muy particular. Cuando se define una variable o un método, V8 asigna un tipo inicial basado en el valor asignado. Esta información se guarda en un buffer de memoria.

```
// Ejemplo de tipado dinámico en JavaScript
let valor = "Hola"; // Inicialmente es una cadena
valor = 100; // Ahora es un número
```

En este ejemplo, `valor` cambia de un string a un número. ¡Es como si tu perro de repente decidiera ser gato! Esto, aunque útil a veces, puede traer problemas.

Cuando el tipo de una variable cambia, V8 debe reestructurar la forma en que almacena esta variable en memoria. Esto implica recalcular y reasignar la memoria para la nueva forma del dato, lo cual puede ser costoso en términos de rendimiento.

```
let a = 1; // 'a' es un número
a = "uno"; // 'a' ahora es un string
```

El motor V8 detecta el cambio de tipo y ajusta la memoria y las referencias internas para adaptarse al nuevo tipo. Este proceso puede ralentizar la ejecución si ocurre con frecuencia.

TypeScript: Estabilidad y Seguridad a Través del Tipado Estático

En contraste, TypeScript introduce un sistema de tipado estático, que obliga a definir el tipo de dato de las variables y funciones desde el comienzo. Esto ayuda a evitar muchos errores comunes en JavaScript al hacer que el código sea más predecible y más fácil de debuggear. Al utilizar TypeScript, se puede tener un control mucho más estricto sobre cómo se manejan los datos en las aplicaciones, lo que se traduce en un código más robusto y seguro.

```
let valor: number = 100;
// valor = "Hola"; // Esto causará un error en TypeScript
```

¡Y listo! Ahora `valor` no puede cambiar de tipo y te aseguras que siempre será un número. Como tener un perro que siempre será perro.

❖ TypeScript: El Superhéroe del Desarrollo

Primero, imaginemos que TypeScript es un superhéroe. Su misión: salvarnos de los errores y las pesadillas del código JavaScript. Pero, como todo buen superhéroe, tiene sus límites y áreas de operación. En este caso, TypeScript solo usa sus superpoderes durante el desarrollo.

¿Qué Hace TypeScript?

TypeScript, como dijimos anteriormente, es JavaScript con esteroides. Te permite agregar tipos a tus variables y funciones, lo que ayuda a evitar errores comunes. Pero aquí viene el truco: cuando tu aplicación se ejecuta en el navegador o en Node.js, todo ese código de TypeScript se ha transformado (o transpilado) en JavaScript. Es como si nuestro superhéroe se quitara el traje y se pusiera un uniforme común y corriente.

Linters: Los Compañeros de Batalla

Ahora, hablemos de los linters. Son como los compañeros de equipo de nuestro superhéroe. Los linters, como ESLint, trabajan codo a codo con TypeScript para mantener tu código limpio y libre de errores. Mientras que TypeScript se enfoca en los tipos y la estructura del código, los linters se ocupan de las reglas de estilo y buenas prácticas.

Ejemplo Práctico

Vamos a ver un ejemplo para que esto quede más claro. Supongamos que estamos trabajando en una aplicación súper cool:

```
// TypeScript
function saludar(nombre: string): string {
  return `Hola, ${nombre}`;
}

const saludo = saludar("Gentleman");
console.log(saludo);
```

Acá tenemos una función `saludar` que toma un `nombre` de tipo `string` y devuelve un saludo. TypeScript nos asegura que siempre pasemos un `string` a esta función. Pero, cuando llega el momento de la verdad, esto es lo que se ejecuta en tu navegador:

```
// JavaScript transpilado
function saludar(nombre) {
  return "Hola, " + nombre;
}

var saludo = saludar("Gentleman");
console.log(saludo);
```

¡Voilá! Todo el código TypeScript se ha transformado en JavaScript.

Integración con ESLint

Ahora, sumemos a nuestro linter al equipo. Imagina que tenemos una regla que dice que siempre debemos usar comillas simples. Así se ve el archivo `.eslintrc.json`:

```
{  
  "rules": {  
    "quotes": ["error", "single"]  
  }  
}
```

Si alguien se pone rebelde y usa comillas dobles en lugar de simples, ESLint nos va a tirar de las orejas y nos va a recordar seguir las reglas:

```
// Código incorrecto según ESLint  
function saludar(nombre: string): string {  
  return "Hola, " + nombre; // ESLint nos va a avisar de este error  
}
```

Con ESLint integrado, nuestro código va a mantenerse limpio y consistente, haciendo equipo con TypeScript para asegurarnos de que todo esté en orden.

En Resumen

TypeScript es nuestro superhéroe durante el desarrollo, ayudándonos a escribir código más seguro y predecible. Pero una vez que todo está listo para la acción, se transforma en JavaScript. Y con los linters como compañeros de batalla, mantenemos nuestro código en la línea.

Así que, la próxima vez que alguien te diga que TypeScript "solo sirve durante el desarrollo", sabrás que, aunque es cierto, es precisamente ahí donde marca la diferencia.

~~ Ejemplo Práctico en TypeScript: El Uso de `any` y la Importancia del Tipado

Vamos a ver por qué, aunque TypeScript nos da herramientas poderosas, usarlas incorrectamente puede llevarnos a caer en los mismos problemas que podríamos tener en JavaScript. ¡Prepárense para un pequeño desafío mental!

Paso 1: Declarar una variable con un tipo específico

Vamos a comenzar con algo simple. En TypeScript, podemos especificar el tipo de una variable para asegurarnos de que siempre contenga el tipo correcto de valor. Miren este ejemplo:

```
let numero: number = 5;
```

Ahora, si intentamos asignar un valor de un tipo diferente, como un `string`, TypeScript nos mostrará un error. Esto es genial porque previene errores en tiempo de compilación. Vamos a ver:

```
numero = "esto debería fallar"; // Error: Type 'string' is not assignable to type 'number'.
```

Paso 2: Uso del tipo any

El tipo `any` en TypeScript nos permite asignar cualquier tipo de valor a una variable, similar a lo que ocurre por defecto en JavaScript. Veamos:

```
let variableFlexible: any = 5;
variableFlexible = "puedo ser un string también";
variableFlexible = true; // ¡Y ahora un booleano!
```

Con `any`, no hay errores de compilación, independientemente del tipo de dato que asignemos. Esto nos da mucha flexibilidad, pero también elimina las garantías de seguridad de tipo que TypeScript ofrece.

Pregunta provocativa

Ahora, aquí viene la pregunta clave: Si estamos usando `any`, ¿te parece que está bien? Si piensan que no... entonces no te gusta Javascript ! ya que sería lo mismo que usar '`any`' en todas nuestras variables.

¡Exacto! La mayoría dirá que no es una buena práctica usar `any`, ya que perdemos todos los beneficios del tipado estático que TypeScript ofrece. Es como si volviéramos a JavaScript, donde podemos cometer fácilmente errores de tipo.

Usar `any` nos lleva de vuelta a la flexibilidad (y los peligros) de JavaScript. Si bien `any` puede ser útil en situaciones donde necesitamos una solución temporal o cuando trabajamos con bibliotecas de terceros para las cuales no tenemos tipos, deberíamos evitarlo en nuestro código principal. En TypeScript, la meta es aprovechar el sistema de tipos para escribir código más seguro y mantenable.

~ Tipos Primitivos en TypeScript

TypeScript enriquece el conjunto de tipos primitivos de JavaScript, proporcionando un control más robusto y opciones para la declaración de variables. Aquí están los principales tipos primitivos:

- **Boolean**: Valor verdadero o falso.

```
let estaActivo: boolean = true;
```

- **Number**: Cualquier número, incluyendo decimales, hexadecimales, binarios y octales.

```
let cantidad: number = 56;
let hexadecimal: number = 0xf00d;
let binario: number = 0b1010;
let octal: number = 0o744;
```

- **String**: Cadenas de texto.

```
let nombre: string = "Gentleman";
```

- **Array**: Arreglos que pueden ser tipados.

```
let listaDeNumeros: number[] = [1, 2, 3];
let listaDeStrings: Array<string> = ["uno", "dos", "tres"];
```

- **Tuple**: Permiten expresar un arreglo con número fijo y tipos de elementos conocidos, pero no necesariamente del mismo tipo.

```
let tupla: [string, number] = ["hola", 10];
```

- **Enum**: Un medio para dar nombres más amigables a conjuntos de valores numéricos.

```
enum Color {
  Rojo,
  Verde,
  Azul,
}
let c: Color = Color.Verde;
```

- **Any**: Para valores que pueden cambiar de tipo en el tiempo, es una forma de decirle a TypeScript que maneje la variable como en JavaScript puro.

```
let noEstoySeguro: any = 4;
noEstoySeguro = "quizás sea una cadena";
noEstoySeguro = false; // ahora es un booleano
```

- **Void**: Ausencia de tener cualquier tipo, usado comúnmente como tipo de retorno en funciones que no retornan nada.

```
function advertirUsuario(): void {
  console.log("Este es un aviso!");
}
```

- **Null y Undefined**: Son subtipos de todos los otros tipos.

```
let u: undefined = undefined;
let n: null = null;
```

~ Inferencia de Tipos en TypeScript

TypeScript es inteligente cuando se trata de inferir los tipos de las variables basándose en la información disponible, como el valor inicial de las variables. Sin embargo, esta inferencia puede ser complicada.

```
let mensaje = "Hola, mundo";
// `mensaje` es automáticamente inferido como `string`
```

Trampas de la Inferencia en TypeScript: Un Ejemplo Práctico

¡Hola, comunidad! Hoy vamos a profundizar en un tema fascinante y a veces complicado de TypeScript: las trampas de la inferencia de tipos, utilizando un ejemplo práctico que nos mostrará cómo TypeScript maneja la inferencia de tipos dentro de estructuras de control complejas.

Ejemplo Problemático

Consideremos el siguiente código TypeScript:

```
const arregloDeValores = [
  {
    numero: 1,
    label: "label1",
  },
  {
    numero: 2,
  },
];

const metodo = (param: typeof arregloDeValores) => {
  const indexArray = [1, 2];

  indexArray.forEach((index) => {
    if (param[index].label) {
      // param[index].label => string | undefined
      console.log(param[index].label); // param[index].label => string | undefined
    }
  });
};
```

En este ejemplo, tenemos un arreglo `arregloDeValores` que contiene objetos con las propiedades `numero` y `label`. Sin embargo, notarán que el segundo objeto en el arreglo no tiene definida la propiedad `label`. Esto hace que el tipo de `label` sea inferido como `string | undefined`.

Problema de Inferencia

Cuando pasamos `arregloDeValores` a la función `metodo` y usamos `forEach` para iterar sobre un arreglo de índices, hacemos una comprobación en cada iteración para ver si `label` está presente. Si bien dentro del bloque `if`, uno podría esperar que TypeScript entienda que `label` no es `undefined` debido a la comprobación, la realidad es que TypeScript aún considera que el tipo de `param[index].label` es `string | undefined` tanto dentro como fuera del `if`.

¿Por qué ocurre esto?

TypeScript no lleva un "estado" del tipo a través del flujo del código de la misma manera que lo haría en un contexto más simple. Aunque dentro del bloque `if` ya verificamos que `label` existe, TypeScript no tiene una "memoria" de esta comprobación para futuras referencias en el mismo bloque de código. Esto es especialmente cierto cuando estamos iterando o utilizando estructuras más complejas como `forEach`, `for`, etc., donde las comprobaciones de tipo no se "propagan" más allá del ámbito inmediato en el que se realizan.

Consejo para Manejar la Inferencia en Bloques Iterativos

Para manejar mejor estos casos y asegurar que el código sea tipo-seguro sin depender de la inferencia de TypeScript, podrías considerar las siguientes prácticas:

- **Asignación a Variables Temporales:** A veces, asignar a una variable temporal dentro del bloque puede ayudar.

```
indexArray.forEach((index) => {
  const label = param[index].label;
  if (label) {
    console.log(label); // TypeScript entiende que label es string aquí
  }
});
```

- **Refinamiento de Tipos con Tipos de Guardia:** Utiliza tipos de guardia para refinar los tipos dentro de los bloques iterativos o condicionales.

```
indexArray.forEach((index) => {
  if (typeof param[index].label === "string") {
    console.log(param[index].label); // Ahora es seguro que es un string
  }
});
```

~~ Clases, Interfaces, Enums y Const: ¿Cómo se Utilizan para Tipar en TypeScript?

Cada uno de estos elementos tiene su propia magia para ayudarnos a escribir código más limpio, escalable y seguro. Vamos a desglosar cada uno y ver cómo y cuándo utilizarlos. 🎉✨

1. Clases

Las clases en TypeScript no solo son una plantilla para crear objetos, sino que también pueden ser utilizadas como tipos.

```
class Automovil {  
    constructor(  
        public marca: string,  
        public modelo: string,  
    ) {}  
}  
  
let miAuto: Automovil = new Automovil("Toyota", "Corolla");
```

En el ejemplo, `Automovil` no solo define una clase sino también un tipo. Cuando decimos `let miAuto: Automovil`, estamos utilizando la clase como un tipo, asegurando que `miAuto` cumpla con la estructura y comportamiento definidos en la clase `Automovil`.

2. Interfaces

Las interfaces son potentes en TypeScript por su habilidad de definir contratos de estructuras para clases, objetos y funciones sin generar JavaScript al compilar. Son ideales para definir formas de datos y se usan extensamente en la programación orientada a objetos y en la integración con bibliotecas.

```
interface Vehiculo {  
    marca: string;  
    arrancar(): void;  
}  
  
class Camion implements Vehiculo {  
    constructor(  
        public marca: string,  
        public capacidadCarga: number,  
    ) {}  
    arrancar() {  
        console.log("El camión está arrancando...");  
    }  
}
```

Las interfaces no solo permiten tipar objetos y clases, también pueden ser extendidas y combinadas, lo cual es excelente para mantener el código organizado y reutilizable.

3. Enums

Los enums o enumeraciones permiten definir un conjunto de constantes nombradas. Son útiles para manejar un conjunto de valores relacionados, proporcionando una forma de agruparlos bajo un mismo tipo.

```
enum Color {
  Rojo,
  Verde,
  Azul,
}

let colorFavorito: Color = Color.Verde;
```

Utilizar enums mejora la legibilidad del código y reduce la posibilidad de errores al restringir los valores que una variable puede tomar.

4. Const Assertions

En TypeScript, `const` no solo define una constante a nivel de ejecución, sino que también puede ser utilizada para hacer afirmaciones de tipo (type assertions). Usando `as const`, podemos decirle a TypeScript que trate el tipo de manera más específica y literal.

```
let config = {
  nombre: "Aplicación",
  version: 1,
} as const;

// config.nombre = "Otra App"; // Error, porque nombre es una constante.
```

Esto es especialmente útil para definir objetos con propiedades que nunca cambiarán sus valores una vez asignados.

~~ Type vs Interface en TypeScript: Cuándo y Cómo Usarlos

Aunque ambos se pueden usar para definir tipos en TypeScript, tienen sus particularidades y casos de uso ideales. Vamos a desglosar las diferencias y entender cuándo es mejor usar cada uno. 

¿Qué es interface?

Una `interface` en TypeScript se utiliza principalmente para describir la forma que deben tener los objetos. Es una manera de definir contratos dentro de tu código así como también contratos con código externo al tuyo.

```
interface Usuario {
  nombre: string;
  edad?: number;
}

function saludar(usuario: Usuario) {
```

```
    console.log(`Hola, ${usuario.nombre}`);
}
```

Las interfaces son ideales para la programación orientada a objetos en TypeScript, donde puedes usarlas para asegurar que ciertas clases implementen métodos y propiedades específicos.

Ventajas de usar interface:

- **Extensibilidad:** Las interfaces son extendibles y pueden ser extendidas por otras interfaces. Usando `extends`, una interfaz puede heredar de otra, lo cual es excelente para mantener grandes bases de código bien organizadas.
- **Fusión de Declaraciones:** TypeScript permite que las declaraciones de `interface` sean fusionadas automáticamente. Si defines la misma interfaz en diferentes lugares, TypeScript las combina en una sola interfaz.

¿Qué es type?

El alias de tipo `type` se puede utilizar para crear un tipo personalizado y puede ser asignado a cualquier tipo de dato, no sólo a objetos. Los `type` son más versátiles que las interfaces en ciertos aspectos.

```
type Punto = {
  x: number;
  y: number;
};

type D3Punto = Punto & { z: number };
```

Ventajas de usar type:

- **Tipos Unión e Intersección:** Con `type`, puedes fácilmente utilizar tipos unión e intersección para combinar tipos existentes de formas complejas y útiles.
- **Tipos Primitivos y Tuplas:** Los `type` pueden ser utilizados para alias de tipos primitivos, uniones, intersecciones, y tuplas.

¿Cuándo usar interface o type?

- **Usa interface cuando:**
 - Necesitas definir un 'contrato' para clases o para la forma de objetos.
 - Quieres aprovechar las capacidades de extensión y fusión de interfaces.

- Estás creando una librería de definiciones de tipo o una API que será usada en otros proyectos TypeScript.
- **Usa type cuando:**
 - Necesitas usar uniones o intersecciones.
 - Quieres usar tuplas y otros tipos que no pueden ser expresados con una interface.
 - Prefieres trabajar con tipos más flexibles y no necesitas extender o implementarlos desde clases.

~~ El Concepto de Shape en TypeScript

Ahora vamos a hablar de un concepto fundamental en TypeScript que nos ayuda a manejar la estructura y el tipo de nuestros objetos: el **shape**. Este concepto es crucial para entender cómo TypeScript maneja la tipificación y cómo podemos sacarle el máximo provecho a nuestro código. Así que, ¡Con de Tuti! 🚀

¿Qué es el Shape?

El concepto de **shape** (o forma) en TypeScript se refiere a la estructura que debe tener un objeto para ser considerado de un cierto tipo. Básicamente, cuando definimos un tipo o una interfaz, estamos definiendo el shape que cualquier objeto de ese tipo debe seguir.

```
interface Usuario {  
    nombre: string;  
    edad: number;  
}  
  
let usuario: Usuario = {  
    nombre: "Juan",  
    edad: 25,  
};
```

En este ejemplo, **Usuario** define el shape que el objeto **usuario** debe tener: debe tener las propiedades **nombre** y **edad** de los tipos **string** y **number**, respectivamente.

Inferencia de Tipos y Shape

TypeScript es muy bueno inferiendo tipos basados en los valores que proporcionamos. Sin embargo, la inferencia de tipos también se basa en el shape de los objetos.

```
let otroUsuario = {  
    nombre: "Ana",  
    edad: 30,  
};
```

Aquí, TypeScript inferirá que `otroUsuario` tiene el shape `{ nombre: string; edad: number; }` sin necesidad de que lo especifiquemos explícitamente.

Trampas de la Inferencia con Shape

A veces, confiar en la inferencia de tipos puede llevar a situaciones complicadas, especialmente cuando trabajamos con objetos complejos y arrays. Vamos a volver a ver un ejemplo de cómo esto puede ser problemático:

```
const arregloDeValores = [
  {
    numero: 1,
    label: "label1",
  },
  {
    numero: 2,
  },
];

const metodo = (param: typeof arregloDeValores) => {
  const indexArray = [1, 2];

  indexArray.forEach((index) => {
    if (param[index].label) {
      console.log(param[index].label);
    }
  });
};
```

En este caso ya antes visto, `param[index].label` sigue siendo `string | undefined` tanto fuera como dentro del `if`, por más que hemos comprobado su existencia. ¿Por qué pasa esto? Porque TypeScript no puede garantizar que el shape se mantendrá constante a lo largo de la iteración sin almacenar la comprobación en una variable.

Manejo Correcto del Shape

Para manejar correctamente estas situaciones, es mejor guardar las comprobaciones en una variable, lo cual le da a TypeScript una pista más clara sobre el shape:

```
const metodo = (param: typeof arregloDeValores) => {
  const indexArray = [1, 2];

  indexArray.forEach((index) => {
    const item = param[index];
    if (item.label) {
      console.log(item.label);
    }
  });
};
```

```
});
```

Ahora, TypeScript entiende que `item.label` dentro del `if` es un `string` y no `undefined`.

❖ Entendiendo union e intersección en TypeScript

Vamos a explorar dos operadores fundamentales en TypeScript que nos permiten manejar tipos de manera flexible y poderosa: `|` (unión) y `&` (intersección). Estos operadores son clave para definir tipos complejos y manejar diferentes escenarios en nuestros programas. ¡Vamos a sumergirnos en ellos!

Operador | (Unión)

El operador `|` en TypeScript se utiliza para combinar tipos de manera que un valor pueda ser de uno de esos tipos. Es decir, si tenemos `TipoA | TipoB`, estamos diciendo que una variable puede ser de tipo `TipoA` o de tipo `TipoB`.

```
type Resultado = "éxito" | "error";  
  
let estado: Resultado;  
  
estado = "éxito"; // válido  
estado = "error"; // válido  
estado = "otro"; // inválido, TypeScript marcará un error
```

En este ejemplo, `estado` puede ser `"éxito"` o `"error"`, pero no puede ser otro valor.

Combinación de Tipos con Propiedades Compartidas

Cuando utilizamos el operador `|` para combinar tipos que comparten algunas propiedades, estas propiedades se conservan en la unión solo si son comunes a todos los tipos incluidos. Veamos un ejemplo para entender mejor este concepto:

```
interface Perro {  
  tipo: "perro";  
  ladra: boolean;  
}  
  
interface Gato {  
  tipo: "gato";  
  maulla: boolean;  
}  
  
type Animal = Perro | Gato;  
  
function procesarAnimal(animal: Animal) {
```

```

// Solo podemos acceder a la propiedad 'tipo' común a ambos tipos
console.log(animal.tipo);

// Esto generaría un error, ya que 'ladra' o 'maulla' dependen del tipo específico
// console.log(animal.ladra); // Error: 'ladra' no existe en el tipo 'Animal'.
// console.log(animal.maulla); // Error: 'maulla' no existe en el tipo 'Animal'.
}

let miPerro: Perro = { tipo: "perro", ladra: true };
let miGato: Gato = { tipo: "gato", maulla: true };

procesarAnimal(miPerro); // Salida esperada: "perro"
procesarAnimal(miGato); // Salida esperada: "gato"

```

En este ejemplo, **Animal** es una unión de **Perro** y **Gato**. Aunque ambos tipos tienen la propiedad **tipo**, las propiedades específicas como **ladra** y **maulla** solo están disponibles cuando se trabaja con un tipo específico (**Perro** o **Gato**), no en el tipo **Animal** como un todo.

Operador & (Intersección)

Por otro lado, el operador **&** en TypeScript se utiliza para crear un tipo que tenga todas las propiedades de los tipos que estamos combinando. Es decir, **TipoA & TipoB** significa un tipo que tiene todas las propiedades de **TipoA** y todas las propiedades de **TipoB**.

```

interface Persona {
  nombre: string;
}

interface Empleado {
  salario: number;
}

type EmpleadoConNombre = Persona & Empleado;

let empleado: EmpleadoConNombre = {
  nombre: "Juan",
  salario: 3000,
};

```

En este caso, **EmpleadoConNombre** es un tipo que tiene tanto **nombre** como **salario**, combinando las propiedades de **Persona** y **Empleado**.

Uso de | y & juntos

Podemos combinar **|** y **&** para crear tipos aún más complejos y específicos según nuestras necesidades:

```
type Opciones = { modo: "modoA" | "modoB" } & { tamaño: "pequeño" | "grande" };
```

```
let configuracion: Opciones = {
  modo: "modoA",
  tamaño: "pequeño",
};
```

En este ejemplo, `configuración` debe tener tanto `modo` (que puede ser "modoA" o "modoB") como `tamaño` (que puede ser "pequeño" o "grande").

Diferencias Clave

- **| (Unión)**: Se usa para combinar tipos donde un valor puede ser de cualquiera de esos tipos.
- **& (Intersección)**: Se usa para combinar tipos donde un valor debe tener todas las propiedades de esos tipos.

~ Entendiendo `typeof` en TypeScript

Ahora vamos a ver el uso del operador `typeof` en TypeScript y cómo puede ayudarnos a manejar tipos complejos de manera más eficiente.

Concepto de `typeof`

En TypeScript, `typeof` es un operador que nos permite referirnos al tipo de una variable, propiedad o expresión en tiempo de compilación. Este operador devuelve el tipo estático de la expresión a la que se aplica. Es muy útil cuando necesitamos referirnos a un tipo existente en lugar de definirlo explícitamente.

```
let x = 10;
let y: typeof x; // y será del tipo 'number'
```

En el ejemplo anterior, `typeof x` se evalúa como el tipo de la variable `x`, que es `number`. Esto nos permite asignar el tipo de `x` a otra variable `y` sin tener que especificarlo manualmente.

Utilización para Tipos Complejos

Una de las mayores ventajas de `typeof` es su capacidad para manejar tipos complejos de manera más clara y concisa. Por ejemplo, cuando trabajamos con tipos que son el resultado de uniones o intersecciones complejas, podemos utilizar `typeof` para capturar esos tipos de manera eficiente.

```
interface Persona {
  nombre: string;
  edad: number;
}

type Empleado = {
```

```
id: number;
puesto: string;
} & typeof miPersona; // Captura el tipo de 'miPersona'

const miPersona = { nombre: "Juan", edad: 30 };

let empleado: Empleado;

empleado = { id: 1, puesto: "Desarrollador", nombre: "Juan", edad: 30 };
```

En este ejemplo, `typeof miPersona` captura el tipo de `miPersona`, que es `{ nombre: string; edad: number; }`. Luego, este tipo se combina (`&`) con las propiedades adicionales de `Empleado`. Esto nos permite definir `Empleado` de una manera que aprovecha directamente el tipo de `miPersona` sin tener que repetir su estructura.

Beneficios de `typeof`

- **Refleja Cambios Automáticamente:** Si modificamos `miPersona`, el tipo de `Empleado` se ajustará automáticamente para reflejar esos cambios.
- **Evita Duplicación de Código:** No necesitamos definir manualmente la estructura de `Persona` dos veces; `typeof` se encarga de mantener la consistencia.
- **Mantenimiento Simplificado:** Cuando el tipo de `miPersona` cambia, los usos de `typeof` se actualizan automáticamente, reduciendo errores y tiempos de mantenimiento.

~~ Explorando `as const` en TypeScript

Este operador puede ayudarnos a definir valores constantes inmutables, mejorando la seguridad y precisión de nuestro código. Vamos a ver cómo funciona y cómo podemos usarlo para sacarle el máximo provecho.

¿Qué es `as const`?

El operador `as const` le dice a TypeScript que trate el valor de una expresión como una constante literal. Esto significa que cada valor se considerará inmutable y su tipo se reducirá a su forma más específica posible. Esto es particularmente útil cuando queremos asegurarnos de que los valores no cambien en el futuro.

Ejemplo Básico

Comencemos con un ejemplo simple para ver cómo funciona `as const`:

```
let colores = ["rojo", "verde", "azul"] as const;
```

Si no usáramos `const`, `colores` sería del tipo `string[]`, lo que permite cualquier cadena en el array. Pero al usar `as const`, `colores` se convierte en un tipo literal específico: `readonly ["rojo", "verde", "azul"]`. Ahora, TypeScript sabe que `colores` contiene exactamente esos tres elementos y nada más.

Aplicación en Objetos

El uso de `as const` no se limita a arrays; también puede ser aplicado a objetos. Esto es especialmente útil cuando trabajamos con configuraciones o datos que no deberían cambiar.

```
const configuracion = {
  modo: "producción",
  version: 1.2,
  opciones: {
    depuración: false,
  },
} as const;
```

En este caso, el objeto `configuracion` tiene un tipo inmutable con los valores exactos que hemos definido. Esto significa que `configuracion.modo` es del tipo "producción", `configuracion.version` es del tipo 1.2, y `configuracion.opciones.depuracion` es del tipo `false`.

Beneficios de `as const`

- **Inmutabilidad:** Los valores no pueden ser cambiados, lo que previene errores accidentales.
- **Tipos Literales:** Los tipos se reducen a sus formas más específicas, mejorando la precisión del tipado.
- **Seguridad de Tipo:** Garantiza que los valores no se modifiquen en tiempo de ejecución, proporcionando mayor seguridad en el código.

Uso en Funciones

Veamos cómo `as const` puede mejorar la precisión en el contexto de funciones:

```
function obtenerConfiguracion() {
  return {
    modo: "producción",
    version: 1.2,
    opciones: {
      depuración: false,
    },
  } as const;
}

const config = obtenerConfiguracion();
// config.modo es "producción", no string
```

```
// config.version es 1.2, no number  
// config.opciones.depuracion es false, no boolean
```

Aquí, la función `obtenerConfiguracion` devuelve un objeto cuyo tipo es inmutable gracias a `as const`. Esto asegura que los valores devueltos tengan los tipos más específicos posibles.

❖ Aventura en TypeScript: Type Assertion y Casteo de Tipos

Vamos a explorar cómo utilizarlos correctamente, los cuidados que debemos tener, y la diferencia crucial entre `unknown` y `any`. ¡Vamos a darle!

¿Qué es Type Assertion?

Type Assertion es una forma de indicarle a TypeScript que trate una variable como si fuera de un tipo específico. Es como decirle al compilador: "Confía en mí, sé lo que estoy haciendo". Esto puede ser útil en situaciones donde estamos seguros del tipo de una variable, pero TypeScript no puede inferirlo correctamente.

Hay dos sintaxis principales para Type Assertion en TypeScript:

- Usando el operador `as`:

```
let valor: any = "Este es un string";  
let longitud: number = (valor as string).length;
```

- Usando el operador `<type>`:

```
let valor: any = "Este es un string";  
let longitud: number = (<string>valor).length;
```

Ambas sintaxis logran lo mismo, pero `as` es más comúnmente utilizada en código moderno de TypeScript, especialmente cuando se trabaja con JSX en React.

❖ Cuidados con Type Assertion

Type Assertion es poderoso, pero también puede ser peligroso si se usa incorrectamente. Aquí hay algunas cosas a tener en cuenta:

- **Confianza en el Tipo:** Asegúrate de que la aserción sea válida. Si te equivocas, puedes introducir errores difíciles de detectar.

```
let valor: any = "Este es un string";
let numero: number = valor as number; // ¡Error en tiempo de ejecución!
```

- **Evitar aserciones innecesarias:** No uses Type Assertion si TypeScript puede inferir el tipo correctamente.

```
let valor = "Este es un string"; // TypeScript infiere que 'valor' es un string
let longitud: number = valor.length; // No se necesita Type Assertion
```

❖ Casteo de Tipos en TypeScript

El casteo de tipos es similar a Type Assertion, pero a menudo se refiere a convertir un tipo a otro en tiempo de ejecución, algo más común en lenguajes como C# o Java. En TypeScript, el casteo generalmente se logra mediante funciones de conversión.

Ejemplo:

```
let valor: any = "123";
let numero: number = Number(valor); // Casteo de string a number
```

Aunque TypeScript es un superset de JavaScript, no agrega características de casteo explícito, sino que se basa en funciones de conversión de JavaScript.

❖ unknown vs any: Conoce la Diferencia

any y **unknown** son dos tipos especiales en TypeScript que permiten trabajar con valores de cualquier tipo, pero tienen diferencias clave en su uso y seguridad.

- **any:**

- Permite que cualquier valor sea asignado a una variable.
- Desactiva todas las comprobaciones de tipo, lo que puede llevar a errores en tiempo de ejecución.
- Debe usarse con moderación.

```
let valor: any = "Este es un string";
valor = 42; // No hay error, pero puede causar problemas en tiempo de ejecución
valor.metodoInexistente(); // No hay error en tiempo de compilación, pero fallará en tiempo de ejecución
```

- **unknown:**

- También permite que cualquier valor sea asignado a una variable.

- Obliga a realizar comprobaciones de tipo antes de acceder a las propiedades o métodos, haciendo el código más seguro.

```
let valor: unknown = "Este es un string";

if (typeof valor === "string") {
  console.log(valor.length); // Safe, TypeScript sabe que es un string
}

// valor.metodoInexistente(); // Error en tiempo de compilación
```

❖ Ejemplo Combinado

Veamos un ejemplo que combine Type Assertion, `any`, y `unknown`:

```
function procesarValor(valor: unknown) {
  if (typeof valor === "string") {
    let longitud = (valor as string).length;
    console.log(`La longitud del string es ${longitud}`);
  } else if (typeof valor === "number") {
    let doble = (valor as number) * 2;
    console.log(`El doble del número es ${doble}`);
  } else {
    console.log("El valor no es ni un string ni un número");
  }
}

let valorAny: any = "Texto";
procesarValor(valorAny);

valorAny = 100;
procesarValor(valorAny);

valorAny = true;
procesarValor(valorAny); // El valor no es ni un string ni un número
```

En este ejemplo, usamos `unknown` para recibir valores de cualquier tipo, luego verificamos su tipo antes de realizar operaciones específicas. También mostramos cómo `any` puede ser flexible, pero debe manejarse con cuidado para evitar errores.

❖ Functional Overloading en TypeScript: ¡Pura Magia!

Vamos a explorar cómo podemos definir múltiples firmas para una función y cómo utilizar tipos para que nuestras funciones cambien su output según el tipo de parámetro de entrada. ¡Vamos a darle!

¿Qué es Functional Overloading?

En TypeScript, functional overloading (sobrecarga de funciones) nos permite definir múltiples firmas para una función, de modo que pueda aceptar diferentes tipos de argumentos y comportarse de manera distinta según el tipo de entrada.

Esto es particularmente útil cuando tenemos una función que puede operar de diferentes maneras dependiendo de los parámetros que reciba.

Sintaxis Básica

La sintaxis básica para definir una sobrecarga de funciones en TypeScript incluye varias firmas de función seguidas de una implementación que cubre todos los casos.

```
function miFuncion(param: string): string;
function miFuncion(param: number): number;
function miFuncion(param: boolean): boolean;

// Implementación que cubre todas las sobrecargas
function miFuncion(
    param: string | number | boolean,
): string | number | boolean {
    if (typeof param === "string") {
        return `String recibido: ${param}`;
    } else if (typeof param === "number") {
        return param * 2;
    } else {
        return !param;
    }
}

// Uso de la función sobrecargada
console.log(miFuncion("Hola")); // String recibido: Hola
console.log(miFuncion(42)); // 84
console.log(miFuncion(true)); // false
```

En este ejemplo, `miFuncion` puede aceptar un `string`, un `number` o un `boolean`, y se comportará de manera diferente según el tipo del argumento.

Ejemplos Prácticos

- Función para manipular arrays y strings:

```
function manipular(data: string): string[];
function manipular(data: string[]): string;
function manipular(data: string | string[]): string | string[] {
```

```
if (typeof data === "string") {
  return data.split("");
} else {
  return data.join("");
}

// Uso de la función sobrecargada
console.log(manipular("Hola")); // ['H', 'o', 'l', 'a']
console.log(manipular(["H", "o", "l", "a"])); // "Hola"
```

- Función para manejar diferentes tipos de entradas y producir diferentes salidas:

```
function calcular(input: number): number;
function calcular(input: string): string;
function calcular(input: number | string): number | string {
  if (typeof input === "number") {
    return input * input;
  } else {
    return input.toUpperCase();
  }
}

// Uso de la función sobrecargada
console.log(calcular(5)); // 25
console.log(calcular("hola")); // "HOLA"
```

Consideraciones Importantes

- **Implementación Unificada:** La implementación de la función debe ser capaz de manejar todos los tipos de parámetros definidos en las firmas de sobrecarga.
- **Retorno Compatible:** El tipo de retorno debe ser compatible con todos los tipos definidos en las firmas de sobrecarga.
- **Uso Apropriado de Type Guards:** Es fundamental usar correctamente los type guards (`typeof`, `instanceof`) para asegurar que la implementación maneje adecuadamente cada tipo.

Caso con tipos complejos

Sobrecarga de Funciones con Tipos Complejos

Primero, definimos nuestras interfaces para los tipos complejos `Gato` y `Perro`, que extienden una interfaz base `Animal`.

```

interface Animal {
  tipo: string;
  sonido(): void;
}

interface Gato extends Animal {
  tipo: "gato";
  raza: string;
}

interface Perro extends Animal {
  tipo: "perro";
  color: string;
}

```

Luego, definimos las declaraciones de sobrecarga de la función `procesarAnimal` para especificar los tipos de entrada y los tipos de salida.

```

function procesarAnimal(animal: Gato): string;
function procesarAnimal(animal: Perro): number;

```

A continuación, implementamos la función `procesarAnimal` utilizando la sobrecarga de funciones. Dependiendo de si el parámetro es un `Gato` o un `Perro`, la función devolverá un `string` o un `number`, respectivamente.

```

function procesarAnimal(animal: Gato | Perro): string | number {
  if ("raza" in animal) {
    // El objeto es un gato
    console.log(`Es un gato de raza ${animal.raza}`);
    animal.sonido();
    return animal.raza;
  } else {
    // El objeto es un perro
    console.log(`Es un perro de color ${animal.color}`);
    animal.sonido();
    return animal.color.length;
  }
}

```

Implementación de los Tipos

Creamos instancias de `Gato` y `Perro` y utilizamos la función `procesarAnimal` para procesar estos objetos. Dependiendo del tipo de objeto, la función devolverá un `string` o un `number`.

```

const miGato: Gato = {
  tipo: "gato",

```

```

raza: "Siamés",
sonido: () => console.log("Miau"),
};

const miPerro: Perro = {
  tipo: "perro",
  color: "Negro",
  sonido: () => console.log("Guau"),
};

const resultadoGato = procesarAnimal(miGato); // Output: Es un gato de raza Siamés \n Miau
const resultadoPerro = procesarAnimal(miPerro); // Output: Es un perro de color Negro \n Guau

console.log(resultadoGato); // Output: Siamés
console.loa(resultadoPerro); // Output: 5

```

En este ejemplo, `procesarAnimal(miGato)` devolverá la raza del gato como un `string`, mientras que `procesarAnimal(miPerro)` devolverá la longitud del color del perro como un `number`.

Ejemplo Adicional: Sobrecarga con Verificación de Propiedades

Ahora, veamos otro ejemplo utilizando sobrecarga de funciones y la verificación de propiedades con el operador `in`.

```

interface Vehiculo {
  tipo: string;
  velocidadMaxima(): void;
}

interface Coche extends Vehiculo {
  tipo: "coche";
  marca: string;
}

interface Bicicleta extends Vehiculo {
  tipo: "bicicleta";
  tipoDeFreno: string;
}

function describirVehiculo(vehiculo: Coche): string;
function describirVehiculo(vehiculo: Bicicleta): boolean;

function describirVehiculo(vehiculo: Coche | Bicicleta): string | boolean {
  if ("marca" in vehiculo) {
    // El objeto es un coche
    console.log(`Es un coche de marca ${vehiculo.marca}`);
    vehiculo.velocidadMaxima();
    return vehiculo.marca;
  } else {
    // El objeto es una bicicleta
    console.log(`Es una bicicleta con freno de tipo ${vehiculo.tipoDeFreno}`);
    vehiculo.velocidadMaxima();
  }
}

```

```

        return vehiculo.tipoDeFreno.length > 5;
    }
}

const miCoche: Coche = {
    tipo: "coche",
    marca: "Toyota",
    velocidadMaxima: () => console.log("200 km/h"),
};

const miBicicleta: Bicicleta = {
    tipo: "bicicleta",
    tipoDeFreno: "disco",
    velocidadMaxima: () => console.log("30 km/h"),
};

const resultadoCoche = describirVehiculo(miCoche); // Output: Es un coche de marca Toyota \n 200 km/h
const resultadoBicicleta = describirVehiculo(miBicicleta); // Output: Es una bicicleta con freno de tipo disco

console.log(resultadoCoche); // Output: Toyota
console.log(resultadoBicicleta); // Output: false

```

En este ejemplo, `describirVehiculo(miCoche)` devolverá la marca del coche como un `string`, mientras que `describirVehiculo(miBicicleta)` devolverá un `boolean` indicando si la longitud del tipo de freno de la bicicleta es mayor a 5 caracteres.

~~ Utilitarios de TypeScript: Helpers Esenciales

TypeScript ofrece una variedad de tipos utilitarios que facilitan la manipulación y gestión de tipos complejos. Estos helpers permiten transformar, filtrar y crear nuevos tipos basados en otros tipos existentes. A continuación, exploraremos algunos de los helpers más comunes y cómo se pueden utilizar en el desarrollo diario.

Partial

`Partial<T>` convierte todas las propiedades de un tipo `T` en opcionales. Es útil cuando queremos trabajar con versiones incompletas de un tipo.

```

interface Usuario {
    nombre: string;
    edad: number;
    email: string;
}

const usuarioParcial: Partial<Usuario> = {
    nombre: "Juan",
};

```

Required

`Required<T>` convierte todas las propiedades de un tipo `T` en requeridas. Es el opuesto de `Partial`.

```
interface Configuracion {
  modoOscuro?: boolean;
  notificaciones?: boolean;
}

const configuracionCompleta: Required<Configuracion> = {
  modoOscuro: true,
  notificaciones: true,
};
```

Readonly

`Readonly<T>` convierte todas las propiedades de un tipo `T` en propiedades de solo lectura.

```
interface Libro {
  titulo: string;
  autor: string;
}

const libro: Readonly<Libro> = {
  titulo: "1984",
  autor: "George Orwell",
};

// libro.titulo = 'Rebelión en la granja'; // Error: no se puede asignar a 'titulo' porque es una propiedad de solo lectura
```

Record

`Record<K, T>` construye un tipo de objeto cuyas propiedades son claves del tipo `K` y valores del tipo `T`.

```
type Rol = "admin" | "usuario" | "invitado";

const permisos: Record<Rol, string[]> = {
  admin: ["leer", "escribir", "borrar"],
  usuario: ["leer", "escribir"],
  invitado: ["leer"],
};
```

Pick

`Pick<T, K>` crea un tipo seleccionando un subconjunto de las propiedades `K` de un tipo `T`.

```
interface Persona {  
    nombre: string;  
    edad: number;  
    direccion: string;  
}  
  
const personaNombreEdad: Pick<Persona, "nombre" | "edad"> = {  
    nombre: "Maria",  
    edad: 30,  
};
```

Omit

`Omit<T, K>` crea un tipo omitiendo un subconjunto de las propiedades K de un tipo T.

```
interface Producto {  
    id: number;  
    nombre: string;  
    precio: number;  
}  
  
const productoSinId: Omit<Producto, "id"> = {  
    nombre: "Laptop",  
    precio: 1500,  
};
```

Exclude

`Exclude<T, U>` excluye de T los tipos que son asignables a U.

```
type NumerosOString = string | number | boolean;  
  
type SoloNumerosOString = Exclude<NumerosOString, boolean>; // string | number
```

Extract

`Extract<T, U>` extrae de T los tipos que son asignables a U.

```
type Tipos = string | number | boolean;  
  
type SoloBooleanos = Extract<Tipos, boolean>; // boolean
```

NonNullable

NonNullable<T> elimina **null** y **undefined** de un tipo T.

```
type PosiblementeNulo = string | number | null | undefined;  
  
type SinNulos = NonNullable<PosiblementeNulo>; // string | number
```

ReturnType

ReturnType<T> obtiene el tipo de retorno de una función T.

```
function obtenerUsuario(id: number) {  
  return { id, nombre: "Juan" };  
}  
  
type Usuario = ReturnType<typeof obtenerUsuario>; // { id: number, nombre: string }
```

Ejemplo Completo

Vamos a ver un ejemplo práctico utilizando varios de estos helpers juntos:

```
interface Usuario {  
  id: number;  
  nombre: string;  
  email?: string;  
  direccion?: string;  
}  
  
// Convertir todas las propiedades a opcionales  
type UsuarioParcial = Partial<Usuario>;  
  
// Convertir todas las propiedades a requeridas  
type UsuarioRequerido = Required<Usuario>;  
  
// Crear un tipo de solo lectura  
type UsuarioSoloLectura = Readonly<Usuario>;  
  
// Seleccionar solo algunas propiedades  
type UsuarioBasico = Pick<Usuario, "id" | "nombre">;  
  
// Omitir algunas propiedades  
type UsuarioSinId = Omit<Usuario, "id">;  
  
// Crear un registro de roles a permisos  
type Rol = "admin" | "editor" | "lector";  
const permisos: Record<Rol, string[]> = {  
  admin: ["crear", "leer", "actualizar", "eliminar"],  
  editor: ["crear", "leer", "actualizar"],
```

```
    lector: ["leer"],  
};  
  
// Excluir tipos  
type ID = string | number | boolean;  
type IDSinBooleanos = Exclude<ID, boolean>; // string | number  
  
// Extraer tipos  
type SoloBooleanos = Extract<ID, boolean>; // boolean  
  
// Eliminar null y undefined  
type PuedeSerNulo = string | null | undefined;  
type NoNulo = NonNullable<PuedeSerNulo>; // string
```

❖ Generics en TypeScript

Los genéricos en TypeScript son una poderosa herramienta que permite crear componentes reutilizables y altamente flexibles. Los genéricos proporcionan una forma de definir tipos de una manera que aún no está determinada, lo que permite que las funciones, clases y tipos trabajen con cualquier tipo especificado en el momento de la llamada o de la instancia. A continuación, exploraremos los conceptos básicos y avanzados de los genéricos en TypeScript, incluyendo ejemplos prácticos.

Conceptos Básicos

Los genéricos se declaran utilizando la notación de ángulo `<T>`, donde `T` es un parámetro de tipo genérico. Este parámetro de tipo puede ser cualquier letra o palabra, aunque `T` es comúnmente usado.

Funciones Genéricas

Las funciones genéricas permiten trabajar con cualquier tipo de dato sin sacrificar la tipificación.

```
function identidad<T>(valor: T): T {  
  return valor;  
}  
  
const numero = identidad<number>(42); // 42  
const texto = identidad<string>("Hola Mundo"); // 'Hola Mundo'
```

Clases Genéricas

Las clases genéricas permiten crear estructuras de datos que pueden trabajar con cualquier tipo.

```
class Caja<T> {  
  contenido: T;  
  
  constructor(contenido: T) {
```

```

    this.contenido = contenido;
}

obtenerContenido(): T {
    return this.contenido;
}
}

const cajaDeNumero = new Caja<number>(123);
console.log(cajaDeNumero.obtenerContenido()); // 123

const cajaDeTexto = new Caja<string>("Texto");
console.log(cajaDeTexto.obtenerContenido()); // 'Texto'

```

Interfaces Genéricas

Las interfaces genéricas permiten definir contratos que pueden adaptarse a diferentes tipos.

```

interface Par<K, V> {
    clave: K;
    valor: V;
}

const parNumeroTexto: Par<number, string> = { clave: 1, valor: "Uno" };
const parTextoBooleano: Par<string, boolean> = { clave: "activo", valor: true };

```

❖ Uso Avanzado de Genéricos

Restricciones en Genéricos

Podemos restringir los tipos que un genérico puede aceptar usando `extends`.

```

interface ConNombre {
    nombre: string;
}

function saludar<T extends ConNombre>(obj: T): void {
    console.log(`Hola, ${obj.nombre}`);
}

saludar({ nombre: "Juan" }); // Hola, Juan
// saludar({ apellido: 'Perez' }); // Error: el objeto no tiene la propiedad 'nombre'

```

Genéricos en Funciones de Orden Superior

Podemos utilizar genéricos en funciones que aceptan y retornan otras funciones.

```
function procesar<T>(elementos: T[], callback: (elemento: T) => void): void {
  elementos.forEach(callback);
}

procesar<number>([1, 2, 3], (numero) => console.log(numero * 2)); // 2, 4, 6
```

❖ Ejemplo Completo con Tipos Complejos y `in`

A continuación, vamos a combinar lo aprendido sobre genéricos con una comprobación avanzada de tipos utilizando la palabra clave `in`.

```
interface Animal {
  tipo: string;
  sonido(): void;
}

interface Gato extends Animal {
  tipo: "gato";
  raza: string;
}

interface Perro extends Animal {
  tipo: "perro";
  color: string;
}

function procesarAnimal<T extends Animal>(animal: T): string {
  if ("raza" in animal) {
    return `Es un gato de raza ${animal.raza}`;
  } else if ("color" in animal) {
    return `Es un perro de color ${animal.color}`;
  } else {
    return `Es un animal de tipo desconocido`;
  }
}

const miGato: Gato = {
  tipo: "gato",
  raza: "Siamés",
  sonido: () => console.log("Miau"),
};

const miPerro: Perro = {
  tipo: "perro",
  color: "Negro",
  sonido: () => console.log("Guau"),
};
```

```
console.log(procesarAnimal(miGato)); // Output: Es un gato de raza Siamés
console.log(procesarAnimal(miPerro)); // Output: Es un perro de color Negro
```

En este ejemplo, `procesarAnimal` es una función genérica que puede procesar cualquier tipo de animal que extienda de `Animal`. Utilizamos la palabra clave `in` para verificar la existencia de una propiedad y así determinar el tipo exacto del objeto.

❖ La Magia de los Enums

Primero, definimos dos enums. Los enums son esos amigos que siempre traen algo útil a la fiesta. Nos permiten agrupar constantes con nombre para que no tengamos que andar adivinando qué significa cada valor.

```
enum Numbers1 {
  "NUMBER1" = "number1",
  "NUMBER2" = "number2",
}

enum Numbers2 {
  "NUMBER3" = "number3",
}
```

❖ Combinando Superpoderes

Ahora, mezclamos estos dos enums en un solo objeto. Para esto, usamos el operador de propagación (...). Y ojo, porque `as const` es la clave acá para que TypeScript trate este objeto como una constante inamovible.

```
const myNumbers = { ...Numbers1, ...Numbers2 } as const;
const mixValues = Object.values(myNumbers);
```

❖ Tipos Derivados de los Enums Combinados

¿Y ahora qué? Bueno, ahora usamos `typeof` y `[number]` para crear un tipo que represente los valores combinados de los enums. ¿Cómo es esto?

```
type MixNumbers = (typeof mixValues)[number];
```

Pero, ¿por qué `[number]`?

Buena pregunta, querido lector. Cuando hacemos `Object.values(myNumbers)`, obtenemos un array de valores. Entonces, `typeof mixValues` nos da el tipo de este array, que es `string[]` en nuestro caso. Al usar `[number]`, estamos diciendo "quiero el tipo de los elementos dentro de este array". Es como decirle a TypeScript: "Che, dame el tipo de lo que hay adentro, no del contenedor".

Ahora, la razón más técnica y precisa: los enums en TypeScript generan una estructura interna que usa tanto las claves como los valores para crear una especie de bi-direccionalidad. Esto significa que en el objeto enum, cada valor tiene una clave numérica asociada automáticamente. Cuando usamos `[number]`, estamos aprovechando esta característica para obtener el tipo de los valores que están siendo indexados numéricamente.

❖ Creando un Tipo Basado en Nuestros Valores

Finalmente, creamos un tipo `Enums` que utiliza un índice mapeado para definir propiedades basadas en los valores de `MixNumbers`. Cada propiedad puede ser de cualquier tipo (`any`), porque a veces la vida es así de flexible.

```
type Enums = {
  [key in MixNumbers]: any;
};
```

❖ El Ejemplo Completo

Vamos a ver el código completo en acción:

```
enum Numbers1 {
  "NUMBER1" = "number1",
  "NUMBER2" = "number2",
}

enum Numbers2 {
  "NUMBER3" = "number3",
}

const myNumbers = { ...Numbers1, ...Numbers2 } as const;
const mixValues = Object.values(myNumbers);

type MixNumbers = (typeof mixValues)[number];

type Enums = {
  [key in MixNumbers]: any;
};

// Ejemplo de uso
const example: Enums = {
```

```
number1: "Este es el número 1",
number2: 42,
number3: { detalle: "Número 3 como objeto" },
};

console.log(example);
```

❖ Desglose del Código

- **Definición de Enums:** Numbers1 y Numbers2 son nuestros superhéroes iniciales, cada uno con sus propios poderes.
- **Combinación de Enums:** Mezclamos los poderes de nuestros héroes en un solo equipo utilizando ... y as const.
- **Creación de Tipos Derivados:** Utilizamos typeof y [number] para crear un tipo que representa los valores combinados.
- **Definición del Tipo Enums:** Usamos un índice mapeado para definir propiedades basadas en MixNumbers.