Juan Correa

React Principios SOLID

React y Principios SOLID

Los principios SOLID fueron recopilados y llamados de ese modo por **Robert C. Martin**, conocido como el tío Bob y autor de diversos libros como clean code y clean architecture (de lectura recomendada, por cierto).

Conocer estos principios fue un parte aguas de un antes y después en mi carrera como desarrollador de software.

Si seguimos estos principios, podemos crear un código más fácil de mantener y de leer **independientemente de la tecnología que estemos usando**.

Son cinco principios:

- **S**ingle Responsibility Principle.
- **O**pen-Closed Principle.
- **L**iskov Substitution Principle.
- **I**nterface Segregation Principle.
- **D**ependency Inversion Principle.

Que si leemos las siglas de cada uno, se forma la palabra **SOLID**.

Si bien estos principios han sido asociados y ejemplificados bajo el paradigma de la programación orientada a objetos (OOP por sus siglas en inglés), no significa que estén casados con este paradigma.

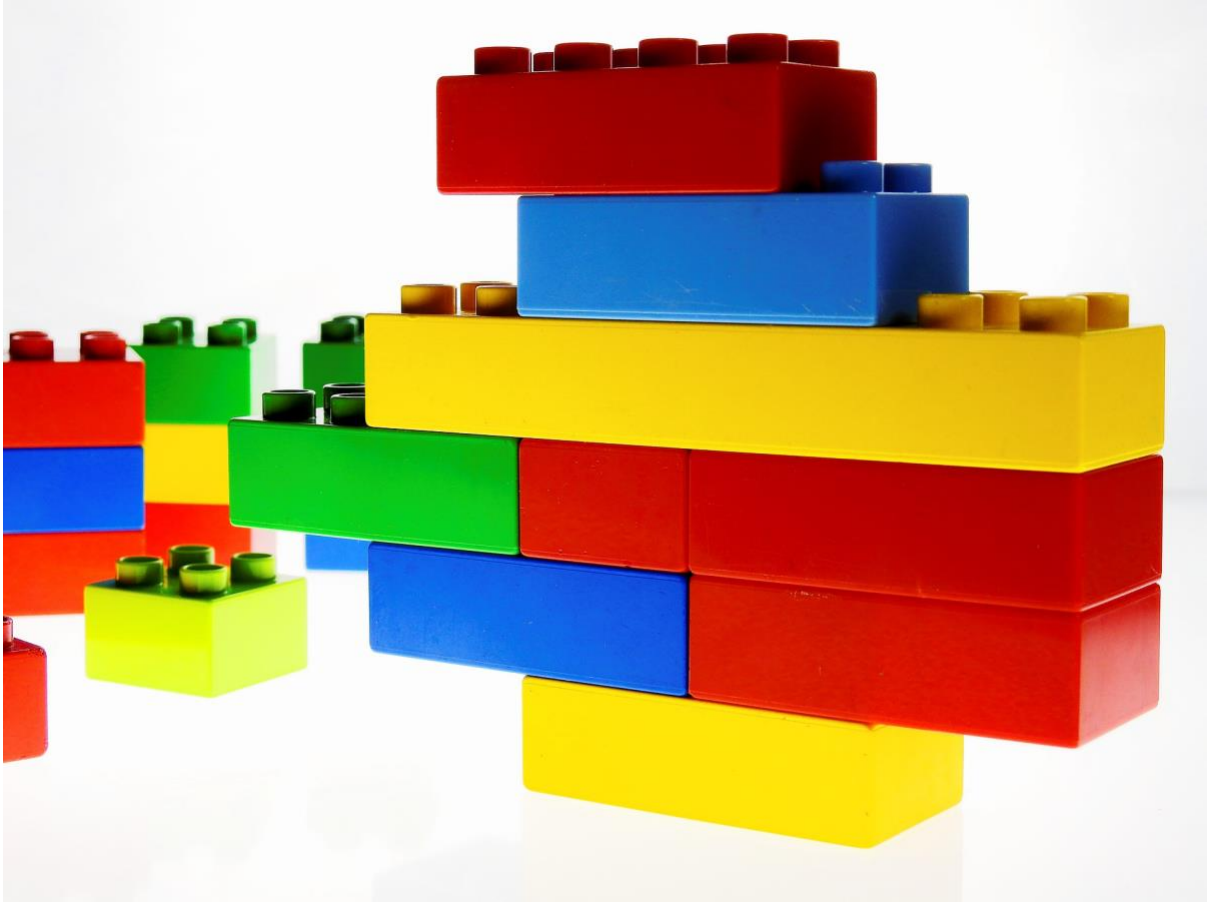
¿Cómo se relacionan los principios SOLID con React JS?

Nuestras aplicaciones en React deberían ser como un edificio construido por piezas de lego.

Una pieza de lego es una unidad primaria que es utilizada en conjunto de otras unidades para crear figuras de manera creativa.

Cada pieza colocada en cada lugar bajo una razón específica, sin tener piezas de sobra ni faltantes, todas integradas en perfecta armonía.

Es por ello que se suelen usar las piezas de lego como una analogía de componentes en React JS.



Nuestros componentes son como piezas de lego: cada uno existe individualmente y pueden componerse para crear nuevas estructuras.

La diferencia es que nosotros somos artesanos que creamos las piezas, somos constructores a la hora de colocarlas para construir un bloque y a su vez somos lectores de las piezas que hacen otros.

Como artesanos y constructores, necesitamos de conocimientos para tomar mejores decisiones a la hora de crear una pieza que sea reusable y de forma que la podamos colocar estratégicamente para construir nuestro edificio.

A su vez, es importante hacer nuestras piezas de lego que también sean fácil de leer por nuestros colegas o terceros.

Si las piezas de Lego son como componentes de React, **los principios SOLID son la guía para construir bloques individuales de Lego** así como para poder usarlos en conjunto de otras piezas de manera armoniosa.

El momento en el que comiences a incorporar los principios SOLID en la manera en que resuelves los problemas de desarrollo, estarás creando un código más limpio, más fácil de mantener y de comprender.

Single Responsibility Principle

Este principio nos dice que:

“Un módulo debe tener una única razón para cambiar”.

Es muy común confundirlo con un principio que se aplica a funciones que son muy grandes: “una función debe hacer sólo una cosa y hacerla bien”.

Hay una gran diferencia entre “hacer solo una cosa” y “tener una razón para cambiar”.

Primero vamos a aclarar lo que significa que el "hacer solo una cosa" en componentes funcionales de React.

Si pensamos en la naturaleza de un componente como una pieza reutilizable, nuestros componentes de React deben estar limitados a hacer sólo una cosa y hacerla bien.

Ahora, si bien cada componente debe hacer solo una cosa, cuando uno o un conjunto de componentes conforman un módulo, este módulo **debe tener una sola razón para ser cambiado**.

¿Qué quiere decir “una razón para cambiar” en este contexto?¿Y cómo se determina o se conoce esa razón?

Es el **usuario final** quien determina las razones para que algo deba cambiar.

Por lo que “una razón para cambiar” significa la necesidad de actualizar nuestro código a raíz de una petición de un usuario final.

Un usuario final no significa referirse a una sola persona en concreto, sino al grupo de personas que comparten características en común que son quienes serán beneficiadas con el software que estamos creando. En UX son llamados “User Personas”.

Un ejemplo de esto es si tienes una app para publicar vacantes de trabajo y tienes por lo menos dos roles:

- Los que reclutan y publican las vacantes.
- Los que buscan y aplican a esas vacantes.

Por lo tanto, si tenemos los roles de reclutadores y aplicantes, si los requerimientos de los reclutadores nos hacen modificar partes de código destinados hacia los aplicantes, entonces estamos violando este principio.

Para que lo anterior haga sentido, vamos entrando a un ejemplo a nivel muy, muy general.

```
const JobOffer = ({ isRecruiter = true }) => {
 if (isRecruiter) {
   return "Oferta de trabajo con opciones para el reclutador";
 }

 return "Oferta de trabajo con opciones para el aplicante";
};
```

Digamos que este componente renderiza la información de la oferta de una vacante pero lo que va a mostrar dependerá del rol del usuario.

Por ejemplo, un reclutador no podrá ver la opción para aplicar a la vacante y un potencial aplicante no podrá ver las opciones para editar la información.

Este componente hace dos cosas diferentes.

Este tipo de prácticas también las puedes identificar cuando se pasan varios props booleanos. Es un indicativo de que *probablemente* el componente tiene más de una responsabilidad.

Primero vamos a refactorizar este componente para que reciba por props lo que debe mostrar sin preocuparse del rol de usuario.

```
const JobOffer = ({ title, description, renderActions }) => {
 return (
   <section>
     <h2>{title}</h2>
     <p>{description}</p>
     {renderActions()}
   </section>
 );
};
```

Spoiler Alert: Estamos usando un prop llamado *renderActions*. Así es como luce el patrón **Render Props**. Veremos más a detalle sobre este patrón en su capítulo correspondiente.

Con esto, nuestro componente JobOffer ya hace sólo una cosa y la hace bien.

Para usarlo sería:

```
<JobOffer
  title="React Developer"
  description="lorem ipsum"
  renderActions={() => <button>Apply</button>}
/>
```

Ya tenemos nuestro componente que hace sólo una cosa y la hace bien, ahora vamos a pasar al pricipio de responsabilidad única continuando con nuestra app para trabajos.

Ya que es un componente sin lógica cuya responsabilidad es mostrar elementos en la UI, necesitamos un componente que tenga la lógica necesaria para poder pasarle por props la información adecuada según lo que se necesite.

Digamos que tenemos un componente JobOfferContainer que tiene la lógica necesaria para ello.

```
const JobOfferContainer = ({ isRecruiter = true }) => {
  // sólo un reclutador puede editar una oferta
  const editJobOffer = () => {
    // código para editar
  };

  // sólo un aplicante puede aplicar a una oferta
  const applyToJobOffer = () => {
    // código para aplicar
  };

  const renderApplyAction = () => (
    <button onClick={applyToJobOffer}>Apply</button>
  );
```

```
 const renderEditAction = () => (
   <button onClick={editJobOffer}>Edit Content</button>
 );

 return (
   <div className="App">
     <JobOffer
       title="React Developer"
       description="lorem ipsum"
       renderActions={isRecruiter ? renderEditAction :
renderApplyAction}
     />
   </div>
 );
};
```

Vemos que aparece de nuevo el booleano isRecruiter y que las funciones que tiene dentro se refieren a aplicar a vacante o editar vacante, dependiendo del rol de usuario.

Pues bien, nuestro componente JobOfferContainer tiene dos razones para cambiar debido a que depende de las necesidades de usuarios diferentes.

Es decir, cuando cambien las necesidades de un candidato para aplicar, probablemente terminemos modificando este componente. Y cuando cambien las necesidades de un reclutador para editar, también.

Una solución a esto es crear dos componentes de responsabilidades (o razones para cambiar) separadas: Un componente con la lógica para reclutador y otro para la lógica del aplicante.

```
const RecluiterJobOfferContainer = () => {
 const editJobOffer = () => {
   // código para editar
 };

 const renderEditAction = () => (
   <button onClick={editJobOffer}>Edit Content</button>
```

```
  );

  return (
    <div className="App">
      <JobOffer
        title="React Developer"
        description="lorem ipsum"
        renderActions={renderEditAction}
      />
    </div>
  );
};
```

Este componente hace sólo una cosa y tiene una sóla razón para cambiar.

Lo mismo aplica para el rol de aplicante, crearíamos un JobOfferApplicantContainer con su lógica para aplicar y reutilizando el componente JobOffer.

Por el momento, el mensaje que te quiero transmitir es que usando esta solución podemos reutilizar componentes, customizar el contenido y estructura en base a la composición de elementos.

Lo mismo que hemos aplicado a nivel de un componente que muestra información, también puede ser aplicado a un componente más global como a una o un grupo de páginas.

El mismo principio aplica: **Cada módulo debe tener una sola razón para cambiar.**

Open Close Principle

Este principio fue hecho por Bertrand Meyer en 1980 y se enuncia como:

“Una entidad de software debe estar abierta para su extensión, pero cerrada para su modificación”.

En mi experiencia, este principio describe lo que significa que **un componente sea realmente reutilizable** y es un común denominador en todos los patrones avanzados en React JS.

En el contexto de React, aplicar este patrón se traduce en crear componentes cuyo comportamiento y estructura de UI puedan ser extendidos o modificados sin necesidad de actualizar el código fuente del componente.

Tan solo imagina una app hecha en React en la que cada vez que necesitemos agregar una nueva funcionalidad, necesitemos modificar gran parte de los componentes existentes.

Eso sería un síntoma de deuda técnica que incrementa los costos por mantener el software así como impactar en la experiencia de desarrollo por parte del equipo, lo cual casi nunca es tomado en cuenta pero que es crucial para mantener a las personas motivadas (en mi experiencia personal).

Por lo tanto, si no seguimos las buenas prácticas de desarrollo podemos terminar creando un monstruo de componentes difíciles de mantener y cerrados al cambio, **incrementando la deuda técnica y complejidad accidental.**

En el mundo de React, este principio puede ser violado en los siguientes casos:

- Cuando necesitamos modificar el formato o apariencia de un componente y tenemos que modificarlo directamente o duplicarlo en uno nuevo (ese componente debería ser extensible tanto en estilos como en estructura).
- Cuando la lógica a implementar en un componente va a variar de un escenario a otro y también la duplicamos (esa lógica debería ser abstraída y proveída según se necesite).

Esto aplica tanto en componentes que solo representan parte de la UI (como un simple button), como componentes que manejan lógica de llamadas a apis y suscripciones a eventos u otras apis.

Por ejemplo, un componente Button puede lucir como sigue:

```
const Button = ({ onClick, children }) => (
 <button onClick={onClick}>{children}</button>
);
```

Nota que pasamos por props lo que queremos que haga cuando el usuario haga click en él y su texto a mostrar. Esto es muy común de ver en los componentes simples como un botón.

No tiene sentido crear diferentes componentes Button para que al hacer click hagan cosas diferentes, estaríamos desaprovechando y violando el fundamento principal de los componentes.

Mejor pasamos por un prop lo que queremos que haga cuando hagamos click en él.

Para componentes que manejan lógica, una manera de dejarlos abiertos a su extensión es por medio de **inyección de dependencias** usando los props, algo muy común cuando usamos el patrón de **High Order Components**, pero también podemos aplicarlo con los **Custom Hooks**, el patrón **Props Getters** e incluso con usando la funcionalidad de **React Context**.

Profundizaremos más sobre estos patrones en sus capítulos correspondientes.

Para ejemplificar este principio en código, digamos que queremos agregar lógica a nuestros componentes en la que haya un handler para mostrar un alert, independientemente del componente donde lo queramos usar.

Una manera de hacerlo sería agregar una función como la siguiente en cada uno de los componentes que queramos que muestre el alert:

```
const handleCall = () => alert("success!");
```

¿Te imaginas qué pasaría si queremos modificar esa función y la hemos copiado y pegado en 50 componentes? Tendríamos que ir de uno en uno modificando, y lo mismo cada vez que cambie en el futuro.

Una manera de resolverlo puede ser con un **High Order Component (HOC)** que inyecte por props la lógica deseada, en este caso, el handleCall.

```
const WithCall = (Component) => {
  // función que queremos inyectar
  const handleCall = () => alert("success!");

  // aquí la inyectamos
```

```
  const Wrapper = (props) =>
    <Component {...props} call={handleCall} />;

  return Wrapper;
};
```

La manera de usar nuestro HOC en un componente sería:

```
const MyComponent = ({ call }) => {
  return <button onClick={call}>Click me!</button>;
 };

const MyComponentWithCall = WithCall(MyComponent);
```

Si no estás familiarizado con los HOC, no te preocupes. Veremos este patrón paso a paso y desde cero en su capítulo correspondiente.

Lo más importante a tener en mente cuando creamos nuestras aplicaciones en React (y en cualquier otra tecnología) es mantener el sistema lo más fácil de extender posible sin impactar en grandes cambios.

Esto es posible creando componentes abiertos al cambio como hemos visto en los ejemplos anteriores y como seguiremos viendo en los patrones.

Liskov Substitution Principle

Debemos a Barbara Liskov la definición de este patrón en 1988. Nos dice:

“Para construir sistemas de software con partes intercambiables, esas partes se deben adherir a un contrato que permita que esas partes puedan ser reemplazadas por otras”.

En la programación orientada a objetos, un contrato sería una interfaz en la que definimos lo que una clase debe implementar.

Pero Javascript no es un lenguaje tipado ni estático en el que podamos definir interfaces. Es aquí donde entra Typescript como alternativa para poder definir contratos por medio de interfaces. Por ejemplo:

```
interface User {
 name: String;
 run: Function;
}
```

Esta interfaz llamada User nos dice que su contrato es: la propiedad name que debe ser de tipo cadena y la propiedad run que debe ser de tipo función.

Esta es una buena razón para considerar usar Typescript en tus proyectos.

Pero también existen más alternativas como usar en conjunto Prop Types, Eslint y Flow, que son herramientas de desarrollo que nos ayudan a validar que nuestro código Javascript cumple con los tipados especificados.

En estae contenido no vamos a profundizar en estas herramientas ya que no es su propósito, pero es bueno que las tengas en mente.

Aunque la mayoría de los ejemplos son usando clases y herencia, en React la herencia no es recomendada debido a que su filosofía es crear **componentes declarativos junto la composición.**

En apps de la vida real se aplica mejor cuando usamos **contratos en los props y states** en los componentes como veremos a continuación.

```
const Greetings = ({ name }) => <h2>Hello {name}</h2>;
```

Este componente recibe por props un valor name y lo renderiza. Para usarlo sería:

```
export default function App() {
 const name = "John Doe";
 return (
   <div className="App">
     <h1>Liskov Substitution Principle</h1>
     <Greetings name={name} />
   </div>
 );
}
```

Esto funciona bien, pero ahora vamos a intercambiar el componente Greetings por uno diferente.

```
const AnotherGreetings = ({ firstname }) =>
  <h2>Hello {firstname}</h2>;

export default function App() {
 const name = "John Doe";
 return (
   <div className="App">
     <h1>Liskov Substitution Principle</h1>
     <AnotherGreetings name={name} />
   </div>
 );
}
```

Vemos que AnotherGreetings en realidad espera recibir el prop firstname pero el componente App le pasa el prop name, por lo que no va a funcionar como esperamos.

Usando Typescript, podemos definir el contrato de los props del siguiente modo:

```
interface UserProps {
 name: String;
}
```

```
const Greetings = ({ name }: UserProps) => <h2>Hello {name}</h2>;

const AnotherGreetings = ({ firstname }: UserProps) => (
 <h2>Hello {firstname}</h2>
);

export default function App() {
 const name = "John Doe";
 return (
   <div className="App">
     <h1>Liskov Substitution Principle</h1>
     <AnotherGreetings name={name} />
   </div>
 );
}
```

Y en tiempo de ejecución, si no respetamos los props definidos por UserProps, nos va a salir un error de transpilación.

Recuerda que si quieres asegurarte que tus componentes cumplen el principio de Liskov por medio de los props y el state, vale la pena aplicar las opciones antes mencionadas:

- Typescript.
- Prop Types, Eslint y Flow.

Este principio es útil tanto en patrones de UI como de lógica.

En los patrones de UI, si queremos maximizar la customización de un grupo de componentes por medio de la composición, es bueno considerar un contrato en los props para que podamos componer con uno u otro componente.

Lo anterior tendrá más sentido una vez que veas los patrones Render Props y Compund Pattern a detalle en sus capítulos correspondientes.

Interface Segregation Principle

Este principio se enuncia como:

“Un sistema no debe depender en cosas que no necesita”.

Un caso de uso en React es cuando un componente se le pasa por props valores de los que realmente necesita. Por ejemplo:

```
const Greetings = ({ user }) => <h2>Hello {user.name}</h2>;

export default function App() {
 const user = {
   name: "John Doe",
   age: 27,
   jobTitle: "React Developer"
 };
 return (
   <div className="App">
     <h1>Liskov Substitution Principle</h1>
     <Greetings user={user} />
   </div>
 );
}
```

En este ejemplo tenemos que el componente Greetings recibe el props user que es un objeto con tres propiedades, pero en realidad sólo necesita la propiedad name.

Para corregirlo, hacemos:

```
const Greetings = ({ name }) => <h2>Hello {name}</h2>;

export default function App() {
 const user = {
   name: "John Doe",
   age: 27,
```

```
    jobTitle: "React Developer"
  };

  return (
    <div className="App">
      <h1>Liskov Substitution Principle</h1>
      <Greetings name={user.name} />
    </div>
  );
}
```

Ahora sólo estamos pasando por props lo que el componente necesita.

Al igual que en Liskov Substitution Principle, en este principio también podemos utilizar las siguientes tecnologías para validar los props:

- Typescript.
- Prop Types, Eslint y Flow.

Y listo, con esto podemos aplicar este principio.

Dependency Inversion Principle

La esencia de este principio la podemos resumir como:

El código que implementa lógica de alto nivel no debe depender de código que implementa los detalles a bajo nivel. Ambos deben depender de abstracciones.

“¡¿Que?!”

Esa fue mi reacción la primera vez que leí esta definición. Lo podemos reformular como:

“Nuestro código debe depender de abstracciones, no de concretos.”

¿A qué nos referimos con abstracciones y concretos en este contexto?

Un concreto es una función o clase que implementa una funcionalidad final en nuestra aplicación. Por ejemplo: un componente funcional que renderiza un botón o un componente que renderiza una página completa.

Una abstracción es una interfaz o clase que va a indicar qué es lo que deben implementar los elementos concretos. Por ejemplo: una interfaz usando Typescript.

Con esto en mente, podemos reformular de nuevo como:

“El código de nuestra aplicación debería depender de interfaces en vez de funciones o clases concretas”.

Si aún suena confuso, no te preocupes. Es normal si es la primera vez que lees sobre este principio.

Vamos a ir paso a paso con ejemplos para que sea más fácil comprender la esencia de este principio y su aplicación.

En React, la inversión de dependencias se logra mediante el uso de interfaces en las dependencias.

Para empezar, definamos una dependencia como aquello que nuestro componente necesita para funcionar.

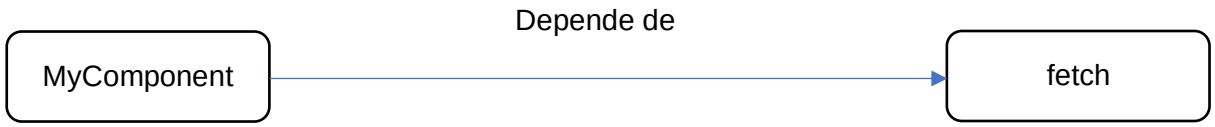
Por ejemplo: Es común ver en blogs ejemplos de componentes que hacen llamadas a apis como el siguiente:

```
const MyComponent = () => {
 const [data, setData] = useState(null);

 useEffect(() => {
   fetch("my-endpoint")
     .then((response) => response.json())
     .then((data) => setData(data));
 }, []);

 if (!data) {
   return "loading...";
 }

 return <div>{data.content}</div>;
};
```

En este caso, fetch es una dependencia porque el componente depende de él para que pueda funcionar correctamente (tiene alto acoplamiento).

Podemos también notar lo siguiente en calidad de código:

- Este componente hace dos cosas: representa datos y tiene lógica de llamada a api.
- No está abierto a su extensión: Si en un futuro cambiamos nuestra api por otra fuente de datos como firebase, tendremos que cambiar nuestro componente aunque en la UI no cambie nada.
- Para hacer unit tests de este componente, tendremos que hacer un mock de fetch.



Como puedes ver, los principios de responsabilidad única y de abierto - cerrado se manifiestan también, lo cual es natural.

“Vale Juan, pero ¿Cómo lo puedo corregir?”.

¡Vamos a ello!

El primer cambio que haremos es crear una función separada que solo haga el fetch como lo necesitamos.

```
const fetchData = async () => {
 const response = await fetch("my-endpoint");
 const data = await response.json();
 return data;
};
```

Nota que ahora estamos resolviendo las promesas con la sintaxis async await y retornamos la respuesta de nuestro endpoint.
Para mantener el ejemplo simple, no consideraremos los escenarios de fallos en la llamada a la api.

La manera de usarlo en el componente ahora es:

```
const MyComponent = () => {
 const [data, setData] = useState(null);

 useEffect(() => {
   const load = async () => {
     const data = await fetchData();
     setData(data);
   };

   load();
 }, []);

 if (!data) {
   return "loading...";
 }
```

```
 return <div>{data.content}</div>;
};
```

Nota: para usar async await dentro de useEffect, necesitamos crear dentro una función debido que el callback de useEffect debe ser una función normal.

Ahora nuestro componente no está directamente acoplado con fetch, pero aun sigue teniendo la dependencia por medio de la función fetchData que hemos creado.

Para no estar usando la dependencia fetchData directamente vamos a aplicar la inyección de de dependencias antes de hacer la inversión de dependencias.

La inyección de dependencias simplemente consiste en inyectar por parámetro o props las dependencias que requerimos en lugar de usarlas directamente.

Para eso podemos usar un HOC o un React Context. Usaremos la segunda opción para variar en ejemplos.

```
const FetchContext = React.createContext(null);

export default function App() {
 return (
   <div className="App">
     <h1>Dependency Inversion</h1>
     <FetchContext.Provider value={fetchData}>
       <MyComponent />
     </FetchContext.Provider>
   </div>
 );
}
```

Con nuestro FetchContext estamos inyectando la función fetchData que definimos antes. Para usarlo seria:

```
const MyComponent = () => {
 const fetchDataFromContext = useContext(FetchContext);
 const [data, setData] = useState(null);

 useEffect(() => {
   const load = async () => {
     const data = await fetchDataFromContext();
     setData(data);
   };

   load();
 }, [fetchDataFromContext]);

 if (!data) {
   return "loading...";
 }

 return <div>{data.content}</div>;
};
```

Con esto ya estamos inyectando nuestra función de llamada a api por medio de React Context.

Si en un futuro cambiamos la llamada a api por una llamada a firebase, lo único que tenemos que hacer es crear una nueva función con la lógica de llamada a firebase y reemplazarla en el value que pasamos por provider.

Por poner un ejemplo:

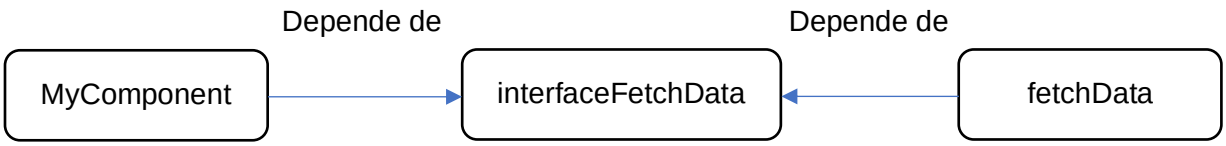
```
export default function App() {
 return (
   <div className="App">
     <h1>Dependency Inversion</h1>
     <FetchContext.Provider value={myAnotherDataFuncion}>
       <MyComponent />
     </FetchContext.Provider>
```

```
    </div>
  );
}
```

Y MyComponent no sabrá si estamos consumiendo una api, Firebase o lo que sea. Ese es parte de nuestro objetivo.

Esto es un buen cambio, pero todavía tenemos una dependencia de un concreto, que es la función fetchData que si bien la estamos inyectando, sigue siendo un concreto.

Una manera de ya no depender de un concreto es actualizar nuestro código con Typescript, declarar una interfaz y usar la interfaz en lugar el concreto.



Y por eso se llama “inversión de dependencias”, porque invertimos la dependencia como se muestra en la figura anterior.

Ahora nuestro componente depende de una abstracción, en este caso, la abstracción es representada por una interfaz de Typescript. Esa misma interfaz la usamos en fetchData para que su definición sea acorde con la firma de la interfaz.

De ese modo, si en el futuro hago una nueva función que consuma Firebase, al FetchContext.Provider no le va a importar porque dependerá de una abstracción, no de un concreto.