**Dependency Inversion Principle (DIP)**

The *dependency inversion* principle is one of the famous SOLID principles. Also, it is one of the most important ones.

Today, we will see how to solve a very common mistake that novice React developers make using this principle.

I will try to keep it very simple. Let’s get started!

**What Does This Principle Tell Us?**

In terms of object-oriented programming, the main idea behind this principle is to always have a high-level code interface with abstraction rather than an implementation detail.

Hold on! I know what you are thinking: “I am a simple frontend developer. Why are you bothering me with these complex terms?”

Let me state it simply for you. For a React application, this principle means:

“No component or function should care about how a particular thing is done.”

Still not clear? OK, let’s get our hands dirty with some code!

**A Practical Example**

Let’s take a very common use case. We are going to make an API call from our component to get some data from a remote source. An implementation can look like this:

Users.js

Look at this component. It depends on some remote data that is fetched right inside the component.

Our Users component’s main responsibility is to render the data. It should not care about how data is fetched or where the data comes from.

This component knows too much — and that’s a problem.

**Why?**

Well, let’s say you have ten other components and all of them fetch their own data.

Now your manager comes along and tells you to use axios instead of fetch

You are in trouble! Now you have to go into each file and refactor the logic to use axios.

But life is not so simple! After a few days, your manager comes again and tells you to implement caching.

You have to do the same thing once again.

Thus, it increases the chance of introducing a bug in your software. Also, the code becomes unmaintainable and valuable time is wasted.

**So What Should We Do Then?**

Let’s introduce a data-fetching Hook and abstract away our logic outside our component because that’s exactly what this principle tells us. To depend on abstraction, remember?

useFetch.ts

Now use this Hook inside our Users component:

User.tsx

Notice a great thing about this solution: Your useFetch Hook doesn’t care about who is calling it. It just takes a URL as an input and returns the data.

Now all other components can take advantage of the Hook that we just wrote. And our Users component no longer depends on the concrete details on how the data is coming back or which library is being used!

**More Advanced Usage**

Now let’s satisfy your manager with basic caching functionality:

useFetch.ts

You have to change the code in only one place now. That’s great! Let’s say you need to show API errors as a toast. Can you do that now? If so, then you got my point.

**The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules. Instead, both should depend on abstractions. This helps to decouple the code and make it more flexible and scalable. Here's an example of a custom API call hook using axios or fetch that follows the DIP in React JS:**

**javascript**

**Copy code**

**import { useState, useEffect } from 'react';**

**import axios from 'axios';**

**const useFetch = (url) => {**

**const [data, setData] = useState(null);**

**const [isLoading, setIsLoading] = useState(true);**

**const [error, setError] = useState(null);**

**useEffect(() => {**

**const fetchData = async () => {**

**try {**

**const response = await axios.get(url);**

**setData(response.data);**

**setIsLoading(false);**

**} catch (error) {**

**setError(error);**

**setIsLoading(false);**

**}**

**};**

**fetchData();**

**}, [url]);**

**return { data, isLoading, error };**

**};**

**export default useFetch;**

**In this example, we use axios to make the API call, but you can also use fetch. The useFetch hook accepts a URL and returns an object that contains the fetched data, a loading state, and an error state. We've created an abstraction of the API call by using a third-party library and returning the data, loading state, and error state in a standard format. This allows any higher-level components to use this hook without worrying about the implementation details of the API call.**

Bad Code:

javascript

Copy code

// UserService.js

export default class UserService {

constructor() {

this.users = [

{ name: 'Alice', email: 'alice@example.com' },

{ name: 'Bob', email: 'bob@example.com' },

];

}

getUsers() {

return this.users;

}

}

// UserList.js

import React, { useState } from 'react';

import UserService from './UserService';

function UserList() {

const [users, setUsers] = useState(new UserService().getUsers());

return (

<div>

<h1>User List</h1>

<ul>

{users.map(user => (

<li key={user.email}>

{user.name} ({user.email})

</li>

))}

</ul>

</div>

);

}

export default UserList;

In the above example, the UserList component directly depends on the UserService class to fetch the list of users. This violates the DIP as the UserList component should not depend on a concrete implementation of the UserService class.

Good Code:

typescript

Copy code

// IUserService.js

export default interface IUserService {

getUsers: () => Promise<User[]>;

}

// UserService.js

export default class UserService implements IUserService {

constructor() {

this.users = [

{ name: 'Alice', email: 'alice@example.com' },

{ name: 'Bob', email: 'bob@example.com' },

];

}

getUsers() {

return Promise.resolve(this.users);

}

}

// UserList.js

import React, { useState, useEffect } from 'react';

import IUserService from './IUserService';

function UserList({ userService }: { userService: IUserService }) {

const [users, setUsers] = useState<User[]>([]);

useEffect(() => {

userService.getUsers().then(setUsers);

}, [userService]);

return (

<div>

<h1>User List</h1>

<ul>

{users.map(user => (

<li key={user.email}>

{user.name} ({user.email})

</li>

))}

</ul>

</div>

);

}

export default UserList;

In the above example, we have fixed the violation of the DIP by introducing an interface IUserService that defines the contract for the UserService class. The UserList component now depends on the IUserService interface instead of the concrete UserService class. The UserList component receives an instance of the IUserService interface as a prop and uses it to fetch the list of users asynchronously. This decouples the UserList component from the implementation details of the UserService class and makes it easier to switch to a different implementation of the IUserService interface in the future.