




SOLID Principles

SOLID Principles in React “with examples”

created by: Zahra Mirzaei

The main purpose of SOLID principles is to work as a guideline for software professionals who care about their craft. People who take pride in building beautifully designed code bases that stand the test of time.





01 – Single Responsibility Principle (SRP)

“A React component should be only responsible for one actor. So, it should have one and only one reason to change.”

Example: Consider a component called “UserProfile” which is responsible for displaying the user’s profile information as well as fetching the data from the API. This violates the SRP as the component has two responsibilities, data retrieval, and display. To adhere to the SRP, we can split this component into two separate components, “UserProfileDisplay” and “UserProfileFetcher”.




```
1 import { useEffect, useState } from "react";
2
3 type Props = {
4   name: string
5   email: string
6 }
7
8 const UserProfile = () => {
9   const [user, setUser] = useState<Props>();
10  const [isLoading, setIsLoading] = useState<boolean>(true);
11  const [error, setError] = useState<null | string>(null);
12
13  useEffect(() => {
14    // fetch user data from API , you can also separate it as a function
15    fetch("https://api.example.com/users/example")
16      .then((response) => response.json())
17      .then((data) => {
18        setUser(data);
19        setIsLoading(false);
20      })
21      .catch((error) => {
22        setError(error);
23        setIsLoading(false);
24      });
25  }, []);
26
27  return (
28    <div>
29      {isLoading ? (
30        <p>Loading ... </p>
31      ) : error ? (
32        <p>Error: {error.message}</p>
33      ) : (
34        <UserProfileDisplay user={user} />
35      )}
36    </div>
37  );
38 };
39
40 const UserProfileDisplay = ({user}: Props) => {
41   const {name, email} = user
42   return (
43     <div>
44       <h2>User Profile</h2>
45       <p>Name: {name}</p>
46       <p>Email: {email}</p>
47     </div>
48   );
49 };
50
51 export { UserProfile, UserProfileDisplay };
52
```



02 – Open/Closed Principle (OCP)

“A React component should can be extended with new functionality using inheritance and composition, without having to modify the existing code.”

Example: Consider a component called “Form” that can be used to display different types of forms, such as a “login form” or “contact form”. To adhere to the OCP, we can create a base “Form” component and extend it for each specific form type, such as “LoginForm” and “ContactForm”. This way, the base “Form” component remains closed for modification but open for extension.




```
1  const Form = ({ name, email, onSubmit, onChange, children }) => {
2    return (
3      <div>
4        <form onSubmit={onSubmit}>
5          <input type="text" name="name" value={name} onChange={onChange} />
6          <input type="text" name="email" value={email} onChange={onChange} />
7
8          {children}
9
10         <button type="submit">Submit</button>
11       </form>
12     </div>
13   );
14 };
15
16 const LoginForm = ({ children, onChange, password, ...props }) => {
17   return (
18     <div>
19       <h2>Login Form</h2>
20       <input
21         type="password"
22         name="password"
23         value={password}
24         onChange={onChange}
25       />
26
27       <Form { ...props }>{children}</Form>
28     </div>
29   );
30 };
31
32 const ContactForm = ({ subject, onChange, message, children, ...props }) => {
33   return (
34     <div>
35       <h2>Contact Form</h2>
36
37       <input type="text" name="subject" value={subject} onChange={onChange} />
38       <textarea name="message" value={message} onChange={onChange}></textarea>
39
40       <Form { ...props }>{children}</Form>
41     </div>
42   );
43 };
44
45 export { LoginForm, ContactForm };
46
```

03 – Liskov Substitution Principle (LSP)

“A React component can be swapped with other components that have similar functionality. This makes the code more flexible and easier to scale.”

Example: Consider a component called “Button” and a subclass called “SubmitButton”. Both of these components should have the same methods and properties so that they can be used interchangeably. This way, if we have a component that uses the “Button” component, we can substitute it with the “SubmitButton” component without affecting the functionality.



```
1  const Button = ({ text }) => {
2    const handleClick = () => console.log("Button clicked");
3    return <button onClick={handleClick}>{text}</button>;
4  };
5
6  const SubmitButton = ({ text }) => {
7    const handleClick = () => console.log("Submit Button clicked");
8    return <button onClick={handleClick}>{text}</button>;
9  };
10
11
12 const App = () => (
13   <>
14     <Button text="Click me" />
15     <SubmitButton text="Submit" />
16   </>
17 );
18
```


04 – Interface Segregation Principle (ISP)

“React components only implement the features they require. This helps keep the code efficient and reduces the risk of errors.”

Example: Consider a component called “Notification” that has methods for sending emails, SMS, and push notifications. However, not all implementations of this component need to send all three types of notifications. To adhere to the ISP, we can split the “Notification” component into three separate interfaces, “EmailNotification”, “SMSNotification”, and “PushNotification”. This way, a component that only needs to send emails can implement the “EmailNotification” interface without being forced to implement the other methods.



```

1  const EmailNotification = {};
2  const SMSNotification = {};
3  const PushNotification = {};
4
5  const Notification = () => {
6    const sendEmail = () => console.log("Sending EMAIL notification")
7    const sendSMS = () => console.log("Sending SMS notification")
8    const sendPush = () => console.log("Sending PUSH notification")
9    return (
10     <div>
11       <h1>Notification Component</h1>
12       <button onClick={sendEmail}>Send Email</button>
13       <button onClick={sendSMS}>Send SMS</button>
14       <button onClick={sendPush}>Send Push</button>
15     </div>
16   );
17 };
18
19 const EmailSender = () => {
20   const sendEmail = () => console.log("Sending email notification")
21   return (
22     <div>
23       <h1>Email Sender Component</h1>
24       <button onClick={sendEmail}>Send Email</button>
25     </div>
26   );
27 };
28
29 // Define a component that only needs to send emails and implements the EmailNotification interface.
30 Object.assign(EmailSender.prototype, EmailNotification);
31
32 const SMSSender = () => {
33   const sendSMS = () => console.log("Sending SMS notification")
34   return (
35     <div>
36       <h1>SMS Sender Component</h1>
37       <button onClick={sendSMS}>Send SMS</button>
38     </div>
39   );
40 };
41
42 // Define a component that only needs to send SMS notifications and implements the SMSNotification interface.
43 Object.assign(SMSSender.prototype, SMSNotification);
44
45 const PushSender = () => {
46   const sendPush = () => console.log("Sending push notification")
47   return (
48     <div>
49       <h1>Push Sender Component</h1>
50       <button onClick={sendPush}>Send Push</button>
51     </div>
52   );
53 };
54
55 // Define a component that only needs to send push notifications and implements the PushNotification interface.
56 Object.assign(PushSender.prototype, PushNotification);

```

D

05 – Dependency Inversion Principle (DIP)

“React components are not closely tied to each other but instead are connected through abstract interfaces. This makes the code more flexible and easier to change.”

Example: The `UserList` component depends on an abstract interface (`dataSource`), not on a concrete implementation of the low-level modules (`API` and `Database`). This allows the `UserList` component to be reused with different implementations of the data source without affecting its functionality.



```
1 // High-level module
2 const UserList = ({ dataSource }) => {
3   const users = dataSource.getUsers();
4
5   return (
6     <ul>
7       {users.map((user) => (
8         <li key={user.id}>{user.name}</li>
9       ))}
10    </ul>
11  );
12 };
13
14 // Low-level module
15 const API = () => {
16   const getUsers = () => {
17     // get users from API
18   };
19
20   return getUsers;
21 };
22
23 // Low-level module
24 const Database = () => {
25   const getUsers = () => {
26     // get users from database
27   };
28
29   return getUsers;
30 };
31
```