Mastering SOLID

Principles In **Javascript**

With **React**

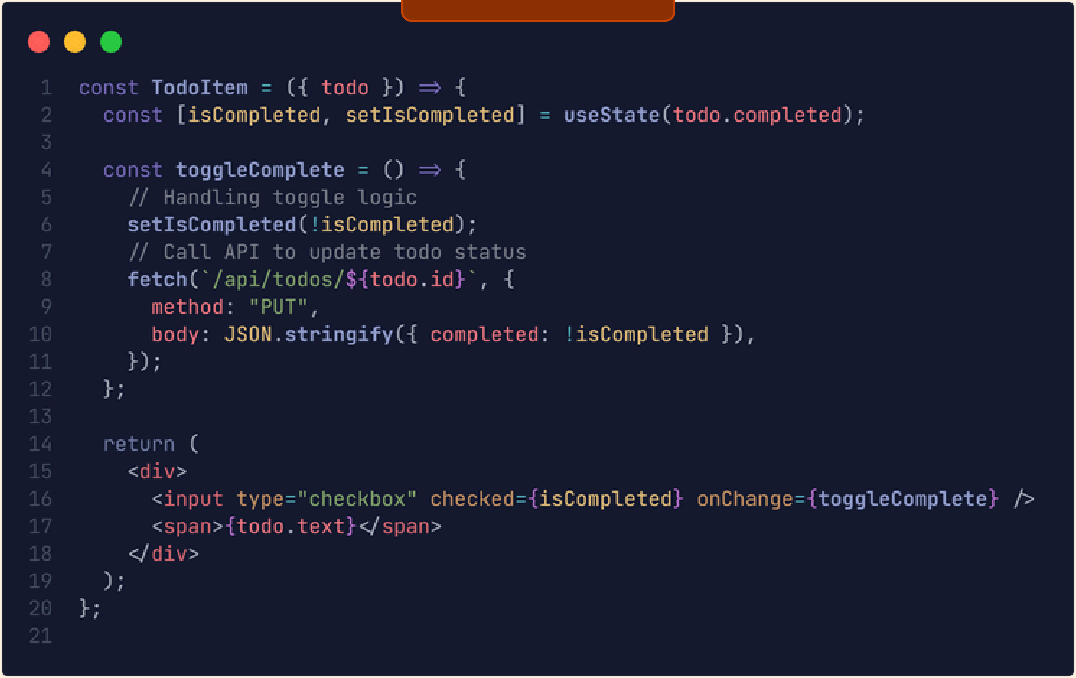
Start

→

1. Single Responsibility Principle

A TODO item component should follow SRP by focusing solely on rendering the UI. In the antipattern, it handles UI, state management, and API calls. By moving state management and API logic to a custom hook, the component now handles only UI rendering.

Anti Pattern

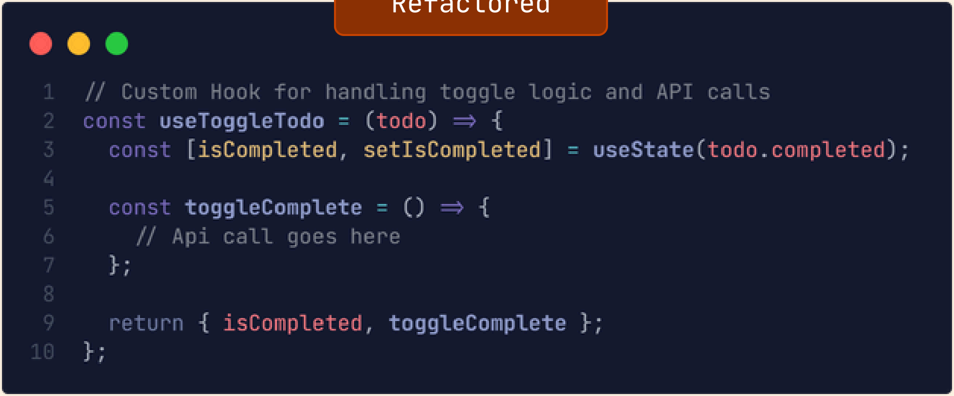


```jsx
const TodoItem = ({ todo }) => {
  const [isCompleted, setIsCompleted] = useState(todo.completed);

  const toggleComplete = () => {
    // Handling toggle logic
    setIsCompleted(!isCompleted);
    // Call API to update todo status
    fetch(`/api/todos/${todo.id}`, {
      method: "PUT",
      body: JSON.stringify({ completed: !isCompleted }),
    });
  };

  return (
    <div>
      <input type="checkbox" checked={isCompleted} onChange={toggleComplete} />
      <span>{todo.text}</span>
    </div>
  );
};
```

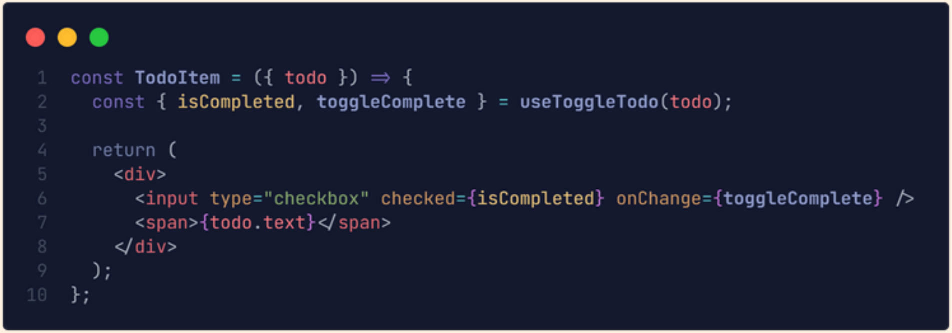
1. Single Responsibility Principle

A TODO item component should follow SRP by focusing solely on rendering the UI. In the antipattern, it handles UI, state management, and API calls. By moving state management and API logic to a custom hook, the component now handles only UI rendering.

Refactored



```
// Custom Hook for handling toggle logic and API calls
const useToggleTodo = (todo) => {
  const [isCompleted, setIsCompleted] = useState(todo.completed);

  const toggleComplete = () => {
    // Api call goes here
  };

  return { isCompleted, toggleComplete };
};
```

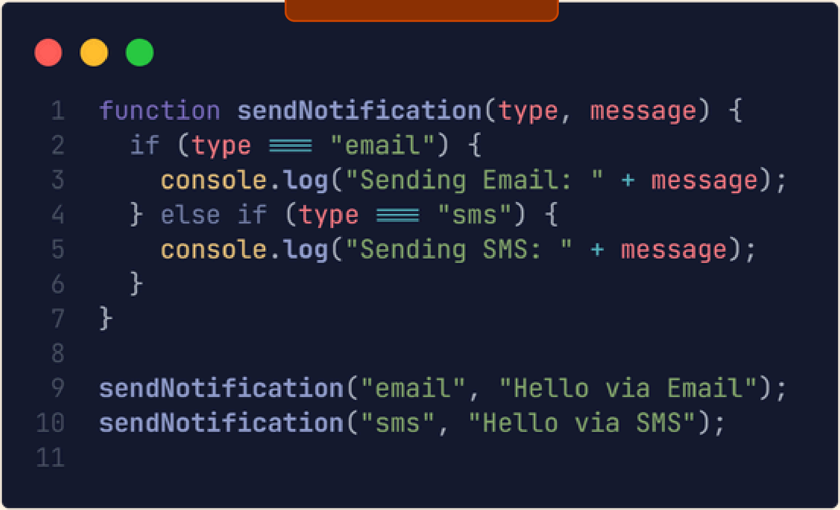


```
const TodoItem = ({ todo }) => {
  const { isCompleted, toggleComplete } = useToggleTodo(todo);

  return (
    <div>
      <input type="checkbox" checked={isCompleted} onChange={toggleComplete} />
      <span>{todo.text}</span>
    </div>
  );
};
```

2. Open/Closed Principle

A notification function violates the Open/Closed Principle if it needs modification for each new notification type. Instead, we can create separate methods for each notification type and pass them to a general sendNotification function. This way, new types are added without changing the core function, protecting the principle.

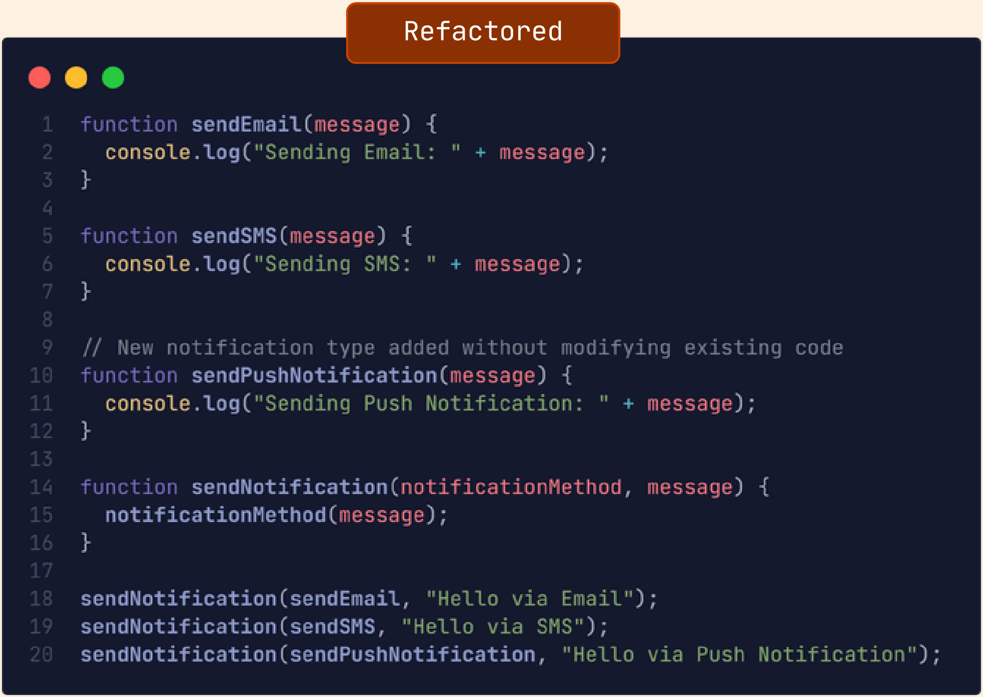
Anti Pattern



```javascript
function sendNotification(type, message) {
  if (type === "email") {
    console.log("Sending Email: " + message);
  } else if (type === "sms") {
    console.log("Sending SMS: " + message);
  }
}

sendNotification("email", "Hello via Email");
sendNotification("sms", "Hello via SMS");
```

2. Open/Closed Principle

A notification function violates the Open/Closed Principle if it needs modification for each new notification type. Instead, we can create separate methods for each notification type and pass them to a general sendNotification function. This way, new types are added without changing the core function, protecting the principle.



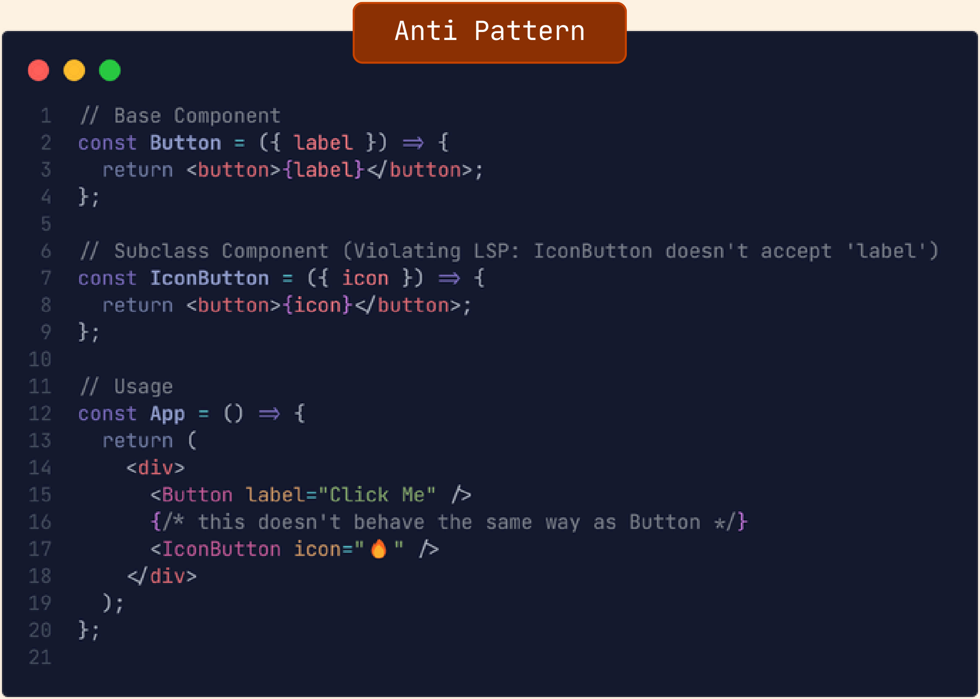
Refactored

```javascript
function sendEmail(message) {
  console.log("Sending Email: " + message);
}

function sendSMS(message) {
  console.log("Sending SMS: " + message);
}

// New notification type added without modifying existing code
function sendPushNotification(message) {
  console.log("Sending Push Notification: " + message);
}

function sendNotification(notificationMethod, message) {
  notificationMethod(message);
}

sendNotification(sendEmail, "Hello via Email");
sendNotification(sendSMS, "Hello via SMS");
sendNotification(sendPushNotification, "Hello via Push Notification");
```

3. Liskov Substitution Principle

In the Liskov Substitution Principle, `IconButton` should be able to replace `Button` without breaking the app. An antipattern occurs if `IconButton` changes the behavior by not accepting the `label` prop like `Button`. To follow the principle, `IconButton` should accept both `label` and `icon`, extending `Button` without errors.

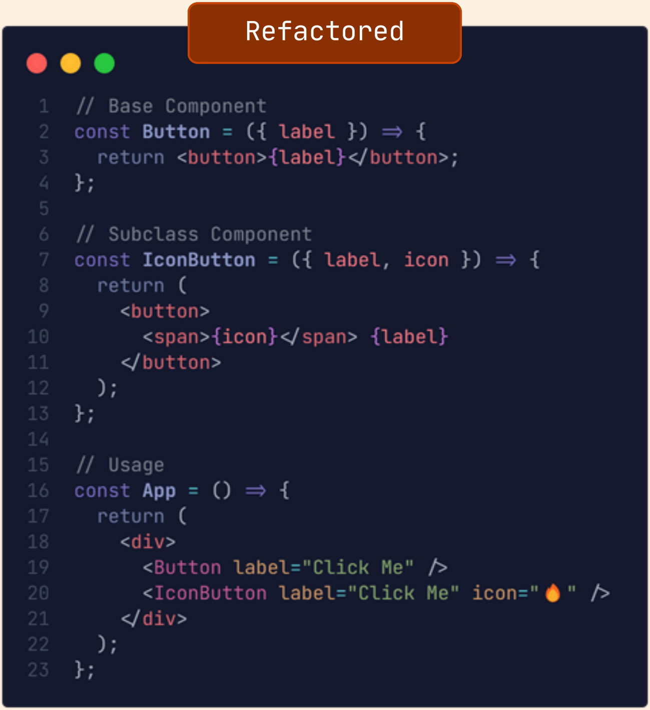
Anti Pattern



```jsx
// Base Component
const Button = ({ label }) => {
  return <button>{label}</button>;
};

// Subclass Component (Violating LSP: IconButton doesn't accept 'label')
const IconButton = ({ icon }) => {
  return <button>{icon}</button>;
};

// Usage
const App = () => {
  return (
    <div>
      <Button label="Click Me" />
      {/* this doesn't behave the same way as Button */}
      <IconButton icon="🔥" />
    </div>
  );
};

```

3. Liskov Substitution Principle

In the Liskov Substitution Principle, `IconButton` should be able to replace `Button` without breaking the app. An antipattern occurs if `IconButton` changes the behavior by not accepting the `label` prop like `Button`. To follow the principle, `IconButton` should accept both `label` and `icon`, extending `Button` without errors.



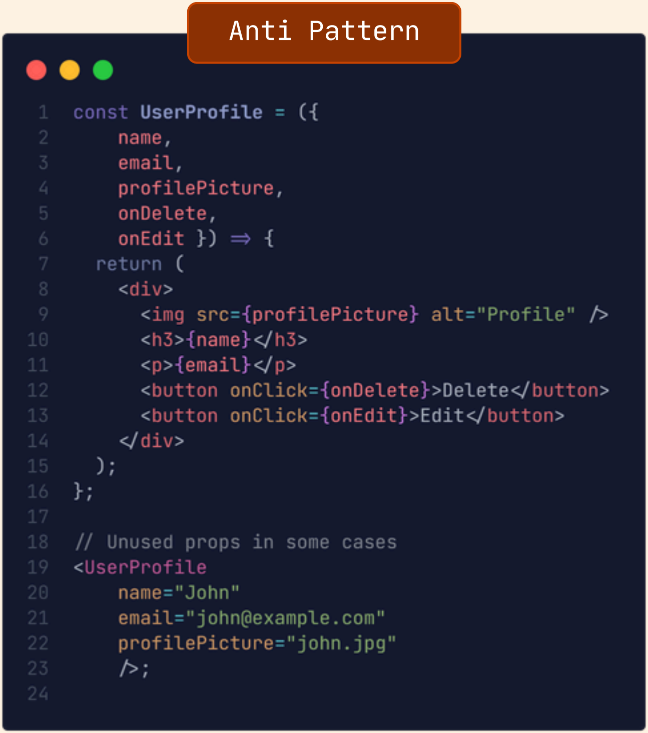
Refactored

```jsx
// Base Component
const Button = ({ label }) => {
  return <button>{label}</button>;
};

// Subclass Component
const IconButton = ({ label, icon }) => {
  return (
    <button>
      <span>{icon}</span> {label}
    </button>
  );
};

// Usage
const App = () => {
  return (
    <div>
      <Button label="Click Me" />
      <IconButton label="Click Me" icon="🔥" />
    </div>
  );
};
```

4. Interface Segregation Principle

The UserProfile component violates the Interface Segregation Principle by accepting unnecessary props like `onDelete` and `onEdit`, making it harder to reuse. To fix this, we can split it into smaller, specific components so each handles only the required props, making the component easier to use and maintain.

Anti Pattern

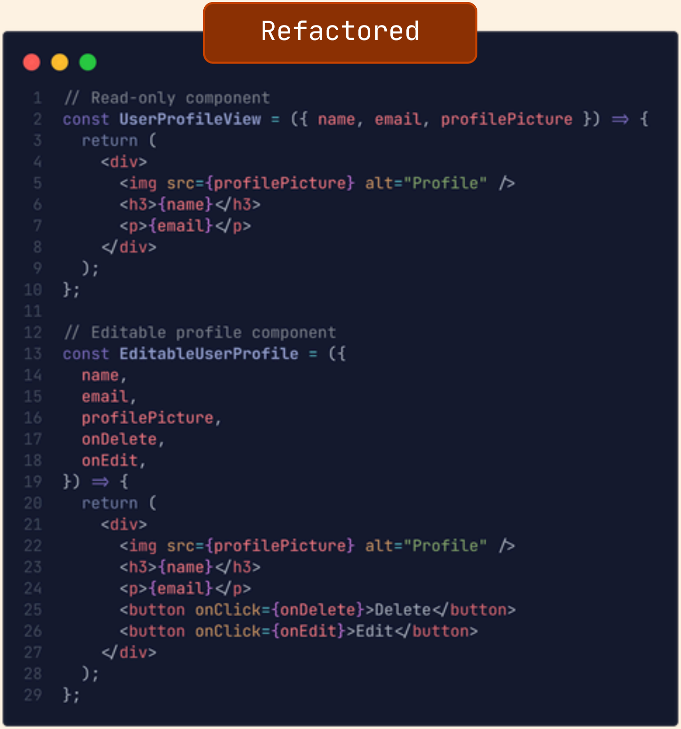


```
const UserProfile = ({
    name,
    email,
    profilePicture,
    onDelete,
    onEdit }) => {
  return (
    <div>
      <img src={profilePicture} alt="Profile" />
      <h3>{name}</h3>
      <p>{email}</p>
      <button onClick={onDelete}>Delete</button>
      <button onClick={onEdit}>Edit</button>
    </div>
  );
};

// Unused props in some cases
<UserProfile
    name="John"
    email="john@example.com"
    profilePicture="john.jpg"
    />;
```

4. Interface Segregation Principle

The UserProfile component violates the Interface Segregation Principle by accepting unnecessary props like `onDelete` and `onEdit`, making it harder to reuse. To fix this, we can split it into smaller, specific components so each handles only the required props, making the component easier to use and maintain.

Refactored

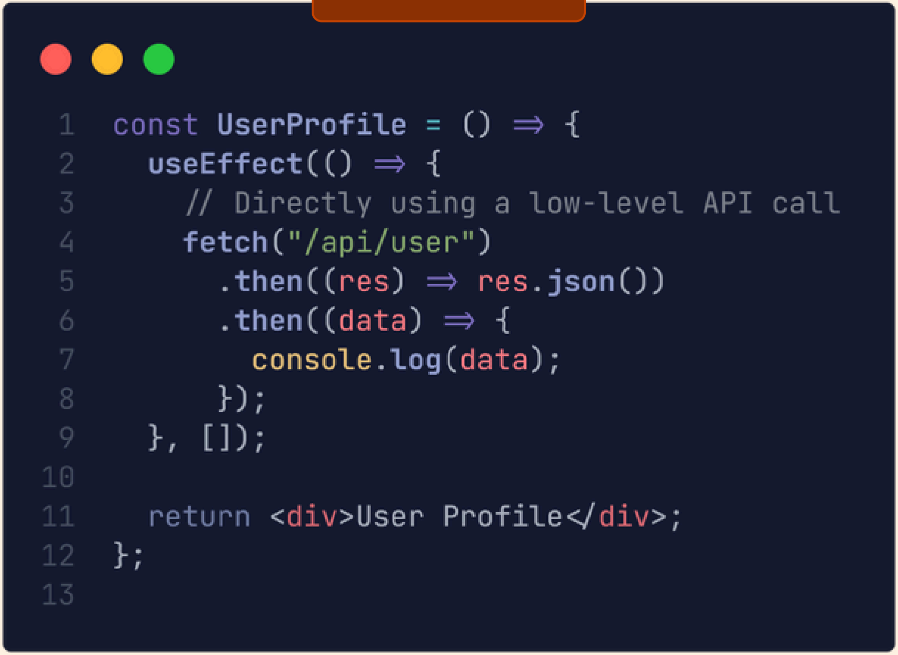


```jsx
// Read-only component
const UserProfileView = ({ name, email, profilePicture }) => {
  return (
    <div>
      <img src={profilePicture} alt="Profile" />
      <h3>{name}</h3>
      <p>{email}</p>
    </div>
  );
};

// Editable profile component
const EditableUserProfile = ({
  name,
  email,
  profilePicture,
  onDelete,
  onEdit,
}) => {
  return (
    <div>
      <img src={profilePicture} alt="Profile" />
      <h3>{name}</h3>
      <p>{email}</p>
      <button onClick={onDelete}>Delete</button>
      <button onClick={onEdit}>Edit</button>
    </div>
  );
};
```

5. Dependency Inversion Principle

In the UserProfile example, directly including an API call breaks the Dependency Inversion Principle, creating tight coupling. To fix this, we can pass the API service as a prop, making the component easier to test and modify without changing UserProfile itself.

Anti Pattern

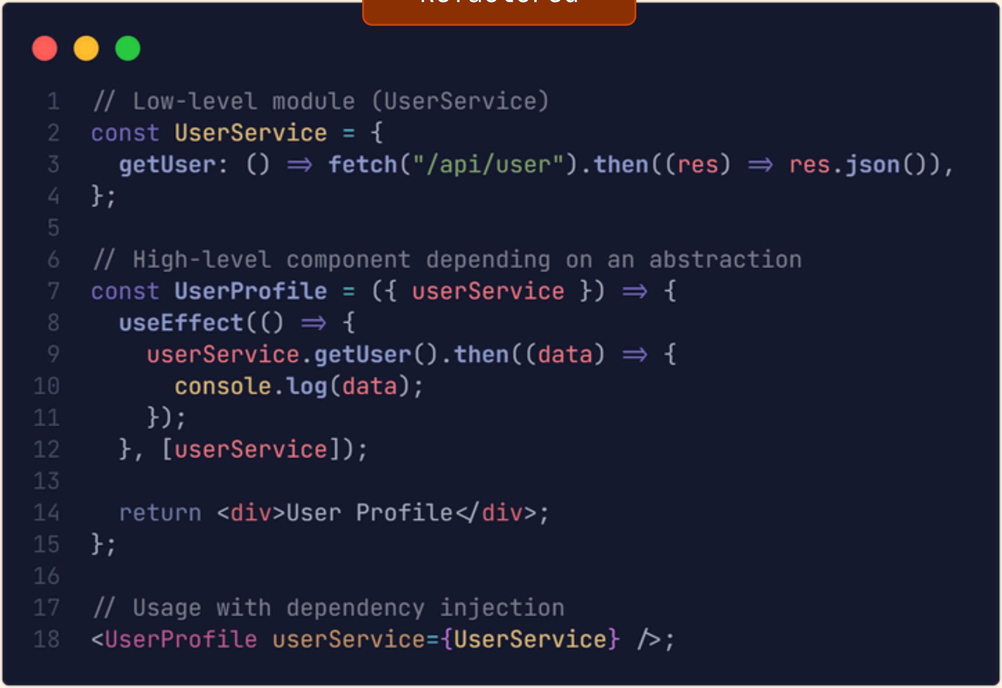


```
const UserProfile = () => {
  useEffect(() => {
    // Directly using a low-level API call
    fetch("/api/user")
      .then((res) => res.json())
      .then((data) => {
        console.log(data);
      });
  }, []);

  return <div>User Profile</div>;
};
```

5. Dependency Inversion Principle

In the UserProfile example, directly including an API call breaks the Dependency Inversion Principle, creating tight coupling. To fix this, we can pass the API service as a prop, making the component easier to test and modify without changing UserProfile itself.

Refactored



```
// Low-level module (UserService)
const UserService = {
  getUser: () => fetch("/api/user").then((res) => res.json()),
};

// High-level component depending on an abstraction
const UserProfile = ({ userService }) => {
  useEffect(() => {
    userService.getUser().then((data) => {
      console.log(data);
    });
  }, [userService]);

  return <div>User Profile</div>;
};

// Usage with dependency injection
<UserProfile userService={UserService} />;
```

Follow For More

Learn Together, Grow Together

Dinuka Nilupul

@dinukanilupul