

Họ và tên: Nguyễn Chấn Hưng

Lĩnh vực: Software Engineering

## BÁO CÁO MINI PROJECT

**ĐỀ TÀI:** Nghiên cứu khả năng tái sử dụng và mở rộng của Custom Hooks trong kiến trúc ứng dụng ReactJS: Xây dựng thư viện hooks có tính module hóa cao

# 1. Giới thiệu về Custom Hooks và tầm quan trọng của thiết kế kiến trúc

## 1.1. ReactJS và vấn đề quản lý logic trạng thái

Trong những năm gần đây, ReactJS đã trở thành một trong những thư viện JavaScript phổ biến nhất để xây dựng giao diện người dùng hiện đại và phức tạp. Với mô hình dựa trên component, React thúc đẩy việc chia nhỏ giao diện thành các phần độc lập và tái sử dụng được. Tuy nhiên, khi ứng dụng phát triển, việc quản lý logic trạng thái xuyên suốt các component trở thành một thách thức đáng kể. Các phương pháp truyền thống như "render props" hay "Higher-Order Components (HOCs)" dù có thể giải quyết được một phần nhưng lại thường dẫn đến các vấn đề như có quá nhiều lớp bọc component, hoặc khó khăn trong việc chia sẻ logic giữa các component. Điều này làm giảm khả năng tái sử dụng, gây khó khăn trong việc đọc hiểu và bảo trì mã nguồn, đồng thời ảnh hưởng đến hiệu suất và khả năng mở rộng của ứng dụng.

## 1.2. Giới thiệu Custom Hooks:

Để giải quyết những hạn chế trên, React đã giới thiệu một tính năng mạnh mẽ vào phiên bản 16.8: "Hooks". Hooks cho phép các nhà phát triển sử dụng state và các tính năng khác của React mà không cần viết Class Components. Đặc biệt, "Custom Hooks" là một cơ chế cho phép chúng ta trích xuất và tái sử dụng logic trạng thái có thể dùng chung giữa các component, một cách hiệu quả.

Một Custom Hook chỉ đơn giản là một hàm trong JavaScript có tên bắt đầu bằng "use" và có thể gọi các React Hooks tiêu chuẩn (như "useState", "useEffect", "useMemo", "useCallback", ...) bên trong nó. Và thay vì trả ra một thành phần UI như một component, thì custom hook sẽ trả về các giá trị, thường là các biến trạng thái, và thường sẽ đi kèm với các hàm setter của các trạng thái đó. Vai trò chính của Custom Hooks bao gồm:

- **Tái sử dụng logic trạng thái:** Cho phép chia sẻ logic giữa các component mà không cần lặp đi lặp lại các đoạn mã giống nhau.

- **Tách biệt logic:** Giúp phân tách logic nghiệp vụ và logic trạng thái khỏi giao diện người dùng, làm cho component trở nên đơn giản hơn và dễ quản lý hơn.
- **Cải thiện tính dễ đọc hiểu (readability) của mã nguồn:** Giảm thiểu sự phức tạp trong component bằng cách trừu tượng hóa các logic phức tạp vào các hàm Custom Hook có tên rõ ràng.
- **Tăng cường khả năng kiểm thử:** Logic được đóng gói trong Custom Hooks thường dễ dàng kiểm thử độc lập hơn so với việc kiểm thử khi nó gắn liền với component UI.

### 1.3. Tầm quan trọng của khả năng tái sử dụng, mở rộng và module hóa trong phát triển ứng dụng ReactJS

Trong bối cảnh phát triển phần mềm Agile và liên tục thay đổi, khả năng tái sử dụng (reusability), mở rộng (extensibility) và module hóa (modularity) trở thành những yếu tố then chốt quyết định sự thành công và bền vững của một dự án.

- **Khả năng tái sử dụng** giúp giảm thiểu việc viết lặp lại các đoạn mã tương tự nhau, tiết kiệm thời gian và công sức phát triển, đồng thời giảm thiểu lỗi do nhất quán trong việc sử dụng lại các thành phần đã được kiểm chứng.
- **Khả năng mở rộng** đảm bảo rằng ứng dụng có thể dễ dàng thích nghi với các yêu cầu mới, thêm tính năng mới mà không cần phải viết lại hoặc thay đổi đáng kể các phần hiện có của hệ thống. Đây là yếu tố đóng vai trò quan trọng trong việc phát triển một ứng dụng có vòng đời dài.
- **Tính module hóa** đề cập đến việc chia nhỏ ứng dụng thành các đơn vị độc lập, có vai trò rõ ràng. Mỗi module có thể được phát triển, kiểm thử và bảo trì một cách riêng biệt mà không ảnh hưởng đến các module khác. Điều này giúp quản lý độ phức tạp, đặc biệt trong các dự án lớn với nhiều nhà phát triển.

Custom Hooks, với bản chất là một cơ chế đóng gói logic có thể tái sử dụng, đóng vai trò trung tâm trong việc hiện thực hóa các nguyên tắc này trong kiến trúc ứng dụng ReactJS. Báo cáo này sẽ đi sâu vào cách tận dụng các mẫu thiết kế phần mềm (như Composite, Adapter, Factory ) kết hợp với Custom Hooks để xây dựng một thư viện hooks có tính tái sử dụng cao, dễ dàng mở rộng và có cấu trúc module hóa.

## 2. Các mẫu thiết kế trong Custom Hooks

Trong lập trình hướng đối tượng, “Design Patterns” là những giải pháp tổng quát, có thể tái sử dụng cho các vấn đề chung thường gặp trong thiết kế phần mềm. Mặc dù React Hooks không hoàn toàn là lập trình hướng đối tượng, nhưng tự duy nhất sau các Design Patterns vẫn cực kỳ hữu ích và có thể được chuyển thể một cách linh hoạt để giải quyết các vấn đề trong cách lập trình liên quan đến hàm và trạng thái của React. Việc áp dụng các mẫu thiết kế này vào “Custom Hooks” không chỉ giúp giải quyết các vấn đề cụ thể mà còn định hình một kiến trúc ứng dụng bền vững, dễ bảo trì và mở rộng.

### 2.1. Composition Pattern

#### 2.1.1. Khái niệm và nguyên lý

Composition Pattern là nguyên lý kết hợp nhiều đối tượng hoặc hàm nhỏ, đơn lẻ, có vai trò rõ ràng để tạo nên một đối tượng hoặc hàm lớn hơn, phức tạp hơn. Thay vì xây dựng một khối chức năng monolithic làm nhiều việc, Composition tập trung vào việc tạo ra các building blocks nhỏ gọn, chuyên biệt, sau đó ghép nối chúng lại với nhau để đạt được một mục đích lớn hơn. Nguyên lý cốt lõi của mẫu thiết kế này là một đối tượng lớn "sở hữu" các đối tượng nhỏ hơn làm thành phần của nó.

Trong bối cảnh của React Custom Hooks, Composition được thể hiện thông qua việc một Custom Hook lớn gọi và sử dụng nhiều Custom Hooks (hoặc Hooks có sẵn của React) nhỏ hơn bên trong nó. Mỗi hook nhỏ đóng gói một phần logic cụ thể (ví dụ: quản lý trạng thái, xử lý đầu vào), và hook lớn hơn sẽ điều phối các hook nhỏ này để cung cấp một chức năng phức tạp hơn cho component.

#### 2.1.2. Áp dụng trong Custom Hooks

Để minh họa Composition Pattern, ta sẽ tạo một Custom Hook `useValidation`. Hook này được thiết kế để quản lý lỗi và cung cấp các hàm xác thực cho các trường dữ liệu. Thay vì chứa logic kiểm tra tất cả các quy tắc xác thực, `useValidation` tận dụng Composition Pattern bằng cách kết hợp nhiều hàm validator đơn lẻ (ví dụ:

`required, minLength, isNumber`). Mỗi hàm validator này có một trách nhiệm duy nhất: kiểm tra một quy tắc cụ thể và trả về thông báo lỗi nếu quy tắc bị vi phạm, hoặc `undefined` nếu hợp lệ.

Trong ví dụ dưới đây, một hook `useValidation` nhận vào một đối tượng ánh xạ validator ứng với từng trường thông tin cần validate (ví dụ: `{fieldName: [validator1, validator2]}`)

### Hook `useValidation`:

```
export const useValidation = (validators = {}) => {

  const [errors, setErrors] = useState({});

  const validateAll = useCallback((formValues) => {

    let isValid = true;

    let newErrors = {};

    for (const fieldName in formValues) {

      let fieldErrors = [];

      if (validators[fieldName]) {

        validators[fieldName].forEach((validator => {

          const error = validator(formValues[fieldName]);

          if (error) {

            fieldErrors.push(error);

          }

        }));

      }

      if (fieldErrors.length > 0) {

        isValid = false;

      }

    }

    return {isValid, errors: fieldErrors};

  });

  return [errors, validateAll];
}
```

```

        newErrors[fieldName] = fieldErrors.length > 0 ? fieldErrors :
undefined;

    }

    setErrors(newErrors);

    return isValid;
}, [validators]);

const validate_errors = errors;

return {validate_errors, validateAll, setErrors};

};

```

## Các hàm validator:

```

export const required = (value) => {

    if (!value || value.trim() === '') {

        return 'Trường này là bắt buộc.';

    }

    return null;
};

export const minLength = (length) => (value) => {

    if (value && value.length < length) {

        return `Cần ít nhất ${length} ký tự.`;

    }

    return null;
};

```

```
export const isEmail = (value) => {

  const emailRegex = /^[^@\s]+@[^\s@]+\.\[^@\s]+\$/;

  if (value && !emailRegex.test(value)) {

    return 'Địa chỉ email không hợp lệ.';

  }

  return null;

};

export const isNumber = (value) => {

  if (value === null || value === undefined || value === '') {

    return 'Trường này bắt buộc phải là số.';

  }

  if (isNaN(Number(value)) || !isFinite(Number(value))) {

    return 'Phải là một số hợp lệ.';

  }

  return null;

};

export const isPositive = (value) => {

  if (Number(value) < 0) {

    return 'Giá trị phải là số dương.';

  }

  return null;

};
```

## Sử dụng hook useValidation:

```
const validators = { // validators

    productId: [required, minLength(3)],

    productName: [required, minLength(5)],

    itemPrice: [required, isNumber, isPositive],

    quantityInStock: [required, isNumber, isPositive],

    productCategory: [required],

    productDescription: [required]

};

const { validate_errors, validateAll, setErrors } =
useValidation(validators);
```

Trong ví dụ này, useValidation không chứa trực tiếp logic của các quy tắc xác thực (ví dụ: kiểm tra độ dài tối thiểu hay định dạng số). Thay vào đó, nó nhận vào một đối tượng ánh xạ validator, với mỗi key là tên trường và value là một mảng các hàm validator. useValidation sau đó kết hợp các hàm validator này để thực hiện quá trình xác thực tổng thể cho toàn bộ form.

### 2.1.3. Lợi ích và đánh giá

#### Lợi ích:

- **Có tính tái sử dụng:** Hook useValidation có thể được áp dụng cho bất kỳ form nào bằng cách truyền vào các bộ quy tắc xác thực khác nhau.
- **Tính module hóa và tách biệt logic:** Mỗi quy tắc xác thực được đóng gói trong một hàm nhỏ. Hook useValidation chỉ đóng vai trò là một hàm composite, gọi các hàm này và quản lý trạng thái lỗi.
- **Dễ mở rộng:** Việc thêm một quy tắc xác thực mới cho một trường chỉ đơn giản là thêm một hàm vào mảng validator. Thêm một trường cần xác thực mới chỉ là thêm một thuộc tính vào đối tượng

validators. Vì vậy khi thay đổi yêu cầu, ta không cần sửa đổi logic cốt lõi bên trong hook `useValidation`.

## Đánh giá:

Việc áp dụng Composition Pattern trong custom hook là một phương pháp thiết kế cơ bản và hiệu quả trong React. Mẫu thiết kế này thúc đẩy các nguyên tắc của lập trình hàm và tách biệt mối quan tâm, dẫn đến một kiến trúc mã nguồn sạch sẽ, dễ quản lý và có khả năng mở rộng cao.

## 2.2. Factory Pattern

### 2.2.1. Khái niệm và nguyên lý

Factory Pattern là một mẫu thiết kế khởi tạo (creational design pattern), theo nguyên tắc hướng đối tượng, cung cấp một giao diện để tạo đối tượng trong một lớp cha nhưng cho phép các lớp con thay đổi loại đối tượng sẽ được tạo. Mục tiêu chính của nó là “tách rời quá trình khởi tạo đối tượng khỏi mã sử dụng chúng”. Thay vì gọi trực tiếp constructor của một đối tượng, chúng ta gọi một “factory method” để nó tạo ra đối tượng cho chúng ta. Điều này giúp hệ thống trở nên linh hoạt hơn, vì mã sử dụng không cần biết chi tiết về cách các đối tượng được tạo ra.

Trong ngữ cảnh của React Custom Hooks, Factory Pattern được thể hiện thông qua việc tạo ra một hàm cấp cao để trả về một Custom Hook khác. Hàm factory này sẽ nhận các tham số (ví dụ: một hàm xử lý logic, cấu hình ban đầu) để tùy chỉnh hành vi của hook được tạo ra. Điều này cho phép chúng ta tạo ra nhiều biến thể của một hook, mỗi biến thể có một chức năng chuyên biệt, mà vẫn dựa trên một logic tạo dựng chung.

### 2.2.2. Áp dụng trong Custom Hooks

Để minh họa Factory Pattern, chúng ta sẽ sử dụng một factory method dùng để tạo ra các hook xử lý dữ liệu `createDataProcessorHook` (ví dụ như lọc hay sắp xếp dữ liệu). Ý tưởng là thay vì viết riêng từng hook `useFilterByCategory`, `useSortByName`, chúng ta tạo một

factory method có thể tạo ra các hook đó bằng cách truyền vào các hàm xử lý logic tương ứng.

### Factory function `createDataProcessorHook`:

```
export const createDataProcessorHook = (processFunction, initialConfig) => {

  return (data) => {

    const [config, setConfig] = useState(initialConfig);

    const processedData = useMemo(() => {

      if (!Array.isArray(data)) {

        return [];

      }

      return processFunction(data, config);

    }, [data, config]);

    const updateConfig = useCallback((newConfig) => {

      setConfig(prevConfig => ({ ...prevConfig, ...newConfig }));

      }, []);

      return {

        processedData,
        config,
        updateConfig,
      };
    };
  };
};
```

### Các hook được tạo ra:

```
export const useFilteredProductsByCategory =
createDataProcessorHook(filterProductsByCategory, { category: '' });

export const useFilteredProductsByMaxPrice =
createDataProcessorHook(filterProductsByMaxPrice, { maxPrice: Infinity });

export const useSortedProducts = createDataProcessorHook(sortProductsByName,
{ direction: 'asc' });
```

Trong ví dụ này, `createDataProcessorHook` chính là "nhà máy". Nó không tự mình xử lý dữ liệu, mà nhận vào một `processFunction` (hàm xử lý cụ thể như sắp xếp theo tên hoặc lọc theo danh mục) và trả về một Custom Hook đã được cấu hình sẵn (`useSortedProducts`, `useFilteredProductsByCategory`). Các hook này sau đó có thể được sử dụng trực tiếp trong các component để áp dụng các phép biến đổi dữ liệu một cách linh hoạt.

### 2.2.3. Lợi ích và đánh giá

#### Lợi ích:

- **Tính mở rộng:** Factory Pattern giúp dễ dàng thêm các kiểu xử lý dữ liệu mới (ví dụ như lọc theo giá tối thiểu, sắp xếp theo số lượng tồn kho) mà không cần thay đổi `createDataProcessorHook` hoặc các hook đã tạo ra. Chỉ cần viết một `processFunction` mới và dùng factory method trên để sinh ra hook tương ứng.
- **Tính module hóa:** Logic tạo hook được tách biệt khỏi logic xử lý dữ liệu thực tế. Các `processFunction` (`sortProductsByName`, `filterProductsByCategory`) là các hook độc lập, dễ dàng kiểm thử và tái sử dụng ở bất kỳ đâu.
- **Giảm trùng lặp code:** Tránh việc viết lặp lại các đoạn mã cần thiết cho mỗi hook xử lý dữ liệu (như quản lý config và `useMemo` cho `processedData`).
- **Linh hoạt trong cấu hình:** Cho phép cấu hình hành vi của hook được tạo ra thông qua các tham số truyền vào factory.

#### Đánh giá:

Factory Pattern là một mẫu thiết kế mạnh để tạo ra các biến thể của Custom Hooks một cách có hệ thống. Nó đặc biệt hữu ích khi ta có một nhóm các hook có cấu trúc chung nhưng khác nhau về chi tiết thực thi (chẳng hạn như các chiến lược lọc, sắp xếp, định dạng). Mẫu này thúc đẩy việc tổ chức code theo hướng "viết một lần, sử dụng nhiều lần". Tuy nhiên, việc áp dụng Factory Pattern có thể làm tăng nhẹ độ phức tạp ban đầu do phải định nghĩa factory function và các hàm xử lý riêng biệt. Do đó ta vẫn cần cân nhắc khi nào thì Factory Pattern thực sự mang lại lợi ích đáng kể, tránh sử dụng quá mức cho các trường hợp đơn giản.

## 2.3. Adapter Pattern

### 2.3.1. Khái niệm và nguyên lý

Adapter Pattern là một mẫu thiết kế cấu trúc (structural design pattern), giải thích theo nguyên tắc hướng đối tượng, cho phép các đối tượng có giao diện không tương thích có thể làm việc cùng nhau. Nó hoạt động như một "phiên dịch viên", chuyển đổi giao diện của một lớp được phía client gọi thành một giao diện khác mà logic xử lý của phía server có thể hiểu được và xử lý được. Vấn đề thường gặp là khi ta có một lớp hiện có với một giao diện nhất định, nhưng lớp đó không thể được sử dụng trực tiếp bởi một phần khác của hệ thống vốn mong đợi một giao diện hoặc một cấu hình đầu vào khác. Adapter sẽ bọc lớp hiện có và cung cấp một giao diện mới, phù hợp với yêu cầu của client, mà không cần sửa đổi lớp gốc.

Trong bối cảnh của React Custom Hooks, Adapter Pattern thường được áp dụng khi dữ liệu được nhận từ một nguồn bên ngoài (ví dụ như được lấy từ file JSON) có định dạng không hoàn toàn khớp với định dạng mà các component hoặc hook khác trong ứng dụng mong muốn hoặc có thể xử lý hiệu quả. Custom Hook đóng vai trò là adapter sẽ nhận dữ liệu ở định dạng ngoại lai, sau đó biến đổi nó thành một định dạng nội bộ chuẩn hóa, giúp phần còn lại của ứng dụng có thể làm việc dễ dàng mà không cần quan tâm đến sự khác biệt về định dạng ban đầu.

### 2.3.2. Áp dụng trong Custom Hooks

Để minh họa Adapter Pattern, ta sẽ tạo ra một hook có tên là `useProductCalculator`. Custom Hook này không chỉ tính toán tổng số lượng và tổng giá trị của sản phẩm, mà còn đóng vai trò là một adapter bằng cách chuẩn hóa dữ liệu sản phẩm thô nhận được từ file JSON `products.json` sang một định dạng đồng nhất hơn (`ProductEntity`) để ứng dụng dễ dàng thao tác. Sau đây là cấu hình mẫu của file JSON:

```
[  
  {  
    "productId": "PROD001",  
    "productName": "Laptop X1 Carbon",  
    "productDescription": "Ultra-light business laptop",  
    "itemPrice": 1200.00,  
    "quantityInStock": 50,  
    "productCategory": "Electronics",  
    "lastUpdateDate": "2024-05-10T10:00:00Z"  
  },  
  //...  
]
```

Dữ liệu sản phẩm thô từ nguồn bên ngoài có thể có tên trường khác với quy ước của ứng dụng (`productId` thay vì `id`, `itemPrice` thay vì `price`) hoặc kiểu dữ liệu không nhất quán (`itemPrice` hay `quantityInStock` có thể là `string` thay vì `number`). Ta định nghĩa một hàm adapter từ `RawProductApiData` như sau:

```
const adaptProductForCalculation = (rawProduct) => {  
  return {  
    id: rawProduct.productId,  
    name: rawProduct.productName,  
    description: rawProduct.productDescription,  
    price: rawProduct.itemPrice,  
    quantity: rawProduct.quantityInStock,  
    category: rawProduct.productCategory,  
    lastUpdate: rawProduct.lastUpdateDate  
  };  
};
```

```
        id: rawProduct.productId,  
  
        name: rawProduct.productName,  
  
        description: rawProduct.productDescription,  
  
        price: parseFloat(rawProduct.itemPrice),  
  
        stock: parseInt(rawProduct.quantityInStock, 10),  
  
        category: rawProduct.productCategory,  
  
        lastUpdated: new Date(rawProduct.lastUpdateDate).toLocaleString(),  
    };  
};
```

Hàm adaptProductForCalculation được truyền vào useProductCalculator chính là adapter. Nó nhận một đối tượng từ file JSON và trả về một đối tượng ProductEntity đã được chuẩn hóa về tên trường và kiểu dữ liệu.

## ProductEntity:

```
/**  
  
 * @typedef {object} ProductEntity  
  
 * @property {string} id  
  
 * @property {string} name  
  
 * @property {string} description  
  
 * @property {number} price  
  
 * @property {number} stock  
  
 * @property {string} category  
  
 * @property {string} lastUpdated  
  
 */
```

## Hook useProductCalculator:

```
export const useProductCalculator = (products, adapterFunction) => {

  const { totalProducts, totalPrice, adaptedProducts } = useMemo(() => {

    let sumProducts = 0;

    let sumPrice = 0;

    const adaptedList = [];

    if (Array.isArray(products)) {

      products.forEach(rawProduct => {

        const adaptedProduct = adapterFunction(rawProduct);

        adaptedList.push(adaptedProduct);

        sumProducts += adaptedProduct.stock;

        sumPrice += adaptedProduct.price * adaptedProduct.stock;

      });

    }

    return {

      totalProducts: sumProducts,

      totalPrice: sumPrice,

      adaptedProducts: adaptedList,

    };

  }, [products]);

  return { totalProducts, totalPrice, adaptedProducts };
};
```

## Sử dụng hook useProductCalculator:

```
const { totalProducts, totalPrice, adaptedProducts } =  
useProductCalculator(managedProducts, adaptProductForCalculation);
```

Trong ví dụ trên, useProductCalculator nhận vào một mảng products với định dạng JSON và một hàm adapter. Thông qua hàm adapter adaptProductForCalculation, mỗi sản phẩm được chuyển đổi thành ProductEntity, là một định dạng chuẩn hóa, dễ dàng làm việc và dễ dàng được xử lý. Sau đó, hook này trả về danh sách adaptedProducts cùng với các giá trị tổng đã được tính toán từ dữ liệu chuẩn hóa.

### 2.3.3. Lợi ích và đánh giá

#### Lợi ích:

- **Tách biệt mối quan tâm:** Logic chuyển đổi dữ liệu được đóng gói hoàn toàn trong adapter, giữ cho các component sử dụng dữ liệu sạch sẽ và không cần biết về định dạng gốc. Component chỉ việc làm việc với ProductEntity mà nó mong đợi.
- **Khả năng thích ứng:** Nếu API backend thay đổi tên trường hoặc định dạng dữ liệu, chỉ cần thêm một hàm adapter mới và truyền vào hook bằng props mà không cần sửa logic bên trong hook useProductCalculator.
- **Tăng tính tái sử dụng:** ProductEntity trở thành một giao diện chung cho các component hoặc hook khác nhau có liên quan đến product của ứng dụng, đảm bảo tính nhất quán khi làm việc với dữ liệu sản phẩm.
- **Cải thiện khả năng đọc hiểu:** Với kiểu dữ liệu rõ ràng ProductEntity, mã nguồn trở nên dễ hiểu hơn về luồng dữ liệu và các phép biến đổi của nó.

#### Đánh giá:

Adapter Pattern đặc biệt có giá trị trong các ứng dụng React khi tích hợp với các hệ thống backend hoặc thư viện bên thứ ba có định dạng

dữ liệu khác biệt. Nó giúp giảm thiểu sự phụ thuộc chặt chẽ giữa frontend và các nguồn dữ liệu bên ngoài, mang lại sự linh hoạt đáng kể trong việc quản lý và phát triển. Tuy nhiên, ta vẫn cần xác định rõ khi nào sự khác biệt về giao diện là đủ lớn để cân nhắc việc tạo một adapter riêng.

## 2.4. Strategy Pattern

### 2.4.1. Khái niệm và nguyên lý

Strategy Pattern là một mẫu thiết kế hành vi (behavioral design pattern) cho phép ta định nghĩa một nhóm các thuật toán, đóng gói mỗi thuật toán thành một lớp riêng biệt. Mẫu thiết kế này cho phép thuật toán thay đổi độc lập với các client sử dụng nó. Vấn đề thường gặp là khi một đối tượng hoặc một hệ thống cần thực hiện một hành động cụ thể, nhưng cách thực hiện hành động đó có thể khác nhau tùy thuộc vào context hoặc cấu hình. Thay vì sử dụng các câu lệnh điều kiện (if-else, switch-case) phức tạp để chọn thuật toán, Strategy Design Pattern định nghĩa một giao diện chung cho tất cả các thuật toán, sau đó client có thể chọn và sử dụng thuật toán cụ thể tại thời điểm chạy (runtime).

Trong bối cảnh của React Custom Hooks, Strategy Pattern thường được áp dụng khi logic xử lý dữ liệu hoặc hành vi của một hook cần thay đổi linh hoạt dựa trên các tham số đầu vào hoặc cấu hình của ứng dụng. Thay vì viết nhiều if-else trong hook, chúng ta có thể truyền vào các chiến lược (strategy) khác nhau dưới dạng hàm hoặc đối tượng để xử lý dữ liệu theo các cách riêng biệt, giúp hook trở nên linh hoạt và dễ mở rộng hơn.

### 2.4.2. Áp dụng trong Custom Hooks

Để minh họa Strategy Pattern, ta sẽ tạo một Custom Hook `useProductHighlight`. Hook này được thiết kế để đánh highlight các sản phẩm dựa trên các tiêu chí khác nhau (ví dụ như sản phẩm có giá cao, sản phẩm thuộc một danh mục cụ thể, hoặc sản phẩm có số lượng tồn kho thấp). Thay vì chứa tất cả logic xử lý trực tiếp, `useProductHighlight` sẽ nhận một hàm chiến lược (strategy function) làm tham số, cho phép thay đổi hành vi của hook một cách linh hoạt.

## Hook useProductHighlight:

```
export const useProductHighlight = (products, strategy, config) => {

  const [highlightConfig, setHighlightConfig] = useState({
    strategy: strategy || null,
    config: config || null
  });

  const highlightedProducts = useMemo(() => {
    if (!highlightConfig.strategy || !highlightConfig.config) return products;

    return products.map((product) => ({
      ...product,
      isHighlighted: highlightConfig.strategy(product,
        highlightConfig.config),
    }));
  }, [products, highlightConfig]);

  return [
    highlightedProducts, highlightConfig, setHighlightConfig
  ];
};
```

## Ví dụ về các hàm chiến lược

```
export const highlightLowStock = (product, { threshold }) => {
  if(!threshold || isNaN(threshold)) { return false; }

  return product.stock <= threshold;
};
```

```
export const highlightHighPrice = (product, { minPrice }) => {
  if(!minPrice || isNaN(minPrice)) { return false; }
  return product.price >= minPrice;
};

export const highlightCategory = (product, { category }) => {
  if(!category || typeof category !== 'string') { return false; }
  return product.category.toLowerCase().includes(category.toLowerCase());
};
```

## Cách dùng hook useProductHighlight:

```
const [highlightedProducts, highlightConfig, setHighlightConfig
] = useProductHighlight(
  displayedProducts,
  null,
  null
);
```

Ta có thể chọn cách hành vi được cài đặt bằng cách truyền vào hàm chiến lược và cấu hình (tham số) của hàm chiến lược đó:

```
setHighlightConfig({
  strategy: highlightLowStock,
  config: { threshold: 10 }
});

setHighlightConfig({
```

```
        strategy: highlightHighPrice,  
        config: { minPrice: 100.00 }  
    });  
  
setHighlightConfig({  
    strategy: highlightCategory,  
    config: { category: 'Electronics' }  
});
```

Trong ví dụ trên,

- **highlightLowStock**: Highlight sản phẩm có số lượng tồn kho thấp hơn ngưỡng cho trước.
- **highlightHighPrice**: Highlight sản phẩm có giá cao hơn mức tối thiểu.
- **highlightCategory**: Highlight sản phẩm thuộc danh mục chứa chuỗi con là category.

Các chiến lược trên được định nghĩa dưới dạng các hàm độc lập, mỗi hàm xử lý một logic dựa trên tiêu chí riêng. Các hàm này có điểm chung là nhận một (hoặc một tập) giá trị cho trước, rồi thực thi và trả về giá trị Boolean. Vì vậy chúng có thể được đóng gói bằng một hook, ở đây là **useProductHighlight**, trong đó có các trạng thái (bao gồm “chiến lược” và “tham số” (optional)) được quản lý bởi hook này. Hook **useProductHighlight** sẽ thực thi hàm chiến lược được truyền vào, tách biệt phần sử dụng thuật toán khỏi phần cài đặt thuật toán.

### 2.4.3. Lợi ích và đánh giá

**Lợi ích:**

- **Tính linh hoạt cao**: useProductHighlight cho phép thay đổi tiêu chí highlight sản phẩm một cách linh hoạt bằng cách đơn giản là truyền vào một hàm chiến lược khác thông qua setHighlightConfig. Điều này loại bỏ sự cần thiết của các câu

lệnh điều kiện if-else phức tạp bên trong hook chính để chọn loại highlight.

- **Dễ mở rộng:** Khi cần thêm một tiêu chí highlight mới (ví dụ: highlight sản phẩm mới về, sản phẩm bán chạy nhất), ta chỉ cần tạo một hàm chiến lược mới (ví dụ: highlightNewArrivals) và đưa nó vào thư mục chứa các hàm chiến lược. Không cần sửa đổi logic cốt lõi của useProductHighlight, tuân thủ nguyên tắc “Open/Closed Principle” (hạn chế sửa đổi và ưu tiên mở rộng).
- **Tính module hóa và tách biệt vai trò:** Mỗi thuật toán highlight sản phẩm được đóng gói thành một hàm riêng biệt, có trách nhiệm duy nhất là xác định xem một sản phẩm có nên được highlight hay không. useProductHighlight chỉ tập trung vào việc áp dụng chiến lược được chọn và quản lý trạng thái highlight, giúp mã nguồn rõ ràng, dễ đọc và dễ bảo trì.
- **Dễ kiểm thử:** Các hàm chiến lược (ví dụ: highlightLowStock, highlightHighPrice, highlightCategory) là các hàm thuần túy. Chúng có thể được kiểm thử độc lập mà không cần môi trường React, giảm đáng kể độ phức tạp của việc kiểm thử tổng thể.

## Đánh giá:

Strategy Pattern là một mẫu thiết kế cực kỳ hữu ích trong việc xây dựng các Custom Hooks có khả năng thích ứng và dễ mở rộng, đặc biệt là khi hook đó cần thực hiện các hành vi khác nhau dựa trên các yếu tố đầu vào hoặc cấu hình linh hoạt. Với useProductHighlight, mẫu này cho phép tách biệt việc sử dụng chiến lược (trong ví dụ này là chiến lược highlight) khỏi logic hay cách cài đặt thuật toán highlight cụ thể. Bằng cách ủy quyền thuật toán highlight cho các hàm riêng biệt, useProductHighlight duy trì sự đơn giản và tập trung, trong khi các chiến lược có thể phát triển độc lập. Điều này giúp tránh việc phải thay đổi code của hook chính mỗi khi có một tiêu chí highlight mới, tăng cường tính bền vững và khả năng bảo trì của ứng dụng. Tuy nhiên, việc áp dụng mẫu này cần được cân nhắc để tránh tạo ra quá nhiều chiến lược cho các trường hợp rất đơn giản, điều này có thể làm tăng số lượng file và độ phức tạp không cần thiết. Nó phát huy hiệu quả nhất khi ta có một tập hợp, hay một lớp các thuật toán có cùng đặc

điểm (giải thích theo nguyên tắc hướng đối tượng thì có thể cùng được cài đặt từ một interface).

### 3. Chiến lược tổ chức và phân loại Custom Hooks

Việc thiết kế các Custom Hooks tuân thủ theo cách tổ chức và phân loại chúng trong cấu trúc thư mục của dự án đóng vai trò then chốt trong việc phát triển ứng dụng. Một chiến lược tổ chức hợp lý sẽ giúp nhóm phát triển dễ dàng tìm kiếm, đọc hiểu và quản lý các hook, từ đó nâng cao hiệu quả làm việc và giảm thiểu lỗi. Phần này sẽ bàn về các phương pháp tổ chức phổ biến cho các Custom hooks trong kiến trúc ứng dụng ReactJS.

#### 3.1. Các phương pháp tổ chức phổ biến

Trong quá trình phát triển ứng dụng React, có ba chiến lược chính để tổ chức các Custom Hooks:

##### 3.1.1. Theo chức năng

Phương pháp này nhóm các hooks dựa trên loại chức năng chung mà chúng cung cấp. Đây là cách tiếp cận phổ biến nhất cho các thư viện hook hiện nay hoặc các dự án có nhiều hooks tiện ích nhỏ.

- **Đặc điểm:** Các hooks được đặt trong các thư mục như hooks/state, hooks/ui, hooks/effect, hooks/utils
  - hooks/state: Chứa các hook quản lý trạng thái (ví dụ: useToggle, useCounter, useLocalStorage).
  - hooks/ui: Chứa các hook liên quan đến tương tác UI (ví dụ: useClickOutside, useMediaQuery).
  - hooks/utils: Chứa các hook tiện ích chung (ví dụ: useDebounce, useThrottle).
- **Ưu điểm:**
  - **Dễ tìm kiếm:** Khi ta biết chức năng mong muốn (ví dụ: cần một hook quản lý trạng thái), việc tìm kiếm trở nên dễ dàng hơn.
  - **Tính tái sử dụng cao:** Thúc đẩy việc tạo ra các hook nhỏ, chuyên biệt, có thể sử dụng ở bất kỳ đâu mà không phụ thuộc vào context nghiệp vụ cụ thể.

- **Nhược điểm:**

- **Thư mục lớn:** Có thể dẫn đến các thư mục chứa quá nhiều file nếu số lượng hook tiện ích tăng lên.
- **Khó hình dung context:** Đôi khi khó hiểu mục đích nghiệp vụ lớn hơn của một hook chỉ dựa vào tên chức năng của nó.

### Ví dụ về tương tác giữa các hooks trong chiến lược tổ chức theo chức năng:

Trong phương pháp này, các components hoặc các hooks khác từ bất kỳ domain nào đều có thể gọi tới và sử dụng các hooks chức năng chung. Ta xét một ví dụ useToggle như sau:

```
export const useToggle = (initialValue = false) => {  
  const [value, setValue] = useState(initialValue);  
  
  const toggle = useCallback(() => setValue(prev => !prev), []);  
  
  return [value, toggle];  
};
```

Một component productFilter sử dụng useToggle:

```
function ProductFilter() {  
  
  const [showAdvancedFilters, toggleAdvancedFilters] = useToggle(false);  
  
  return (  
    <div>  
      <button onClick={toggleAdvancedFilters}>  
        {showAdvancedFilters ? 'Hide' : 'Show'} Advanced Filters  
      </button>  
      {showAdvancedFilters && (  
        <div className="advanced-filters">
```

```
    /* Logic advanced filter */  
  
    </div>  
  
  )}  
  
</div>  
  
);  
}
```

Ở đây, `ProductFilter` là một component thuộc domain sản phẩm, sử dụng `useToggle`, là một hook chức năng quản lý trạng thái boolean để điều khiển việc hiển thị bộ lọc nâng cao (advanced filters). Hook `useToggle` hoàn toàn không biết gì về "sản phẩm" hay "bộ lọc", nó chỉ cung cấp một chức năng chung là quản lý biến trạng thái và cung cấp một hàm `toggle` để thay đổi trạng thái Boolean đó.

### 3.1.2. Theo domain nghiệp vụ

Phương pháp này tổ chức hooks dựa trên các domain hoặc thực thể nghiệp vụ của ứng dụng. Đây là cách tiếp cận lý tưởng cho các ứng dụng có nhiều domain riêng biệt và phức tạp.

- **Đặc điểm:** Các hook được đặt trong các thư mục tương ứng với các miền nghiệp vụ (auth, users, products). Ví dụ:
  - `hooks/products`: Chứa `useProductDetails`, `useProductsList`, `useAddProduct`.
  - `hooks/users`: Chứa `useUserSession`, `useUserPermissions`.
- **Ưu điểm:**
  - **Rõ ràng về context:** Dễ dàng tìm thấy tất cả logic liên quan đến một miền nghiệp vụ cụ thể.
  - **Tách biệt domain:** Giúp duy trì sự độc lập giữa các miền, giảm thiểu sự phụ thuộc chéo.
  - **Phù hợp cho dự án lớn:** Hỗ trợ tốt cho các nhóm phát triển lớn, nơi mỗi nhóm có thể chịu trách nhiệm cho một miền nhất định.
- **Nhược điểm:**

- **Trùng lặp logic:** Các hooks chức năng chung (như `useLocalStorage` hoặc `useFetch`) có thể bị trùng lặp nếu nhiều miền cần chúng, hoặc chúng phải được đặt ở một vị trí chung không thuộc miền nào.
- **Khó phân loại:** Một số hooks có thể liên quan đến nhiều hơn một miền, gây khó khăn trong việc quyết định vị trí đặt.

### Ví dụ về tương tác giữa các hooks trong chiến lược tổ chức theo domain:

Trong nội bộ một domain nghiệp vụ, ta xét ví dụ domain nghiệp vụ liên quan đến products. Ta có các hook: `useFetchProducts`, `useProductListLogic`, `useProductFilterUI`. Giả sử đã có sẵn các hàm dịch vụ như `fetchAllProducts` hay `applyLogic`, các hook trên có thể được triển khai như sau:

```
export const useFetchProducts = () => {

  const [products, setProducts] = useState([]);

  const [loading, setLoading] = useState(true);

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

  useEffect(() => {

    const getProducts = async () => {

      setLoading(true);

      try {

        const data = await fetchAllProducts();

        setProducts(data);

      } catch (err) {

        setError(err);

      } finally { setLoading(false); }

    };

  });

};
```

```
        getProducts();

    }, []);

    return { products, loading, error };

};
```

```
export const useProductListLogic = (logic) => {

    const { products, loading, error } = useFetchProducts();

    const processedProducts = useMemo(() => {

        let result = [...products];

        result = applyLogic(result, logic);

        return result;

    }, [products, logic]);

    return { processedProducts, loading, error };

};
```

```
export const useProductFilterUI = () => {

    const [filters, setFilters] = useState({ category: '', minPrice: '' });

    const { processedProducts, loading, error } = useProductListLogic(filters);

    const handleFilterChange = useCallback((e) => {

        const { name, value } = e.target;

        setFilters(prev => ({ ...prev, [name]: value }));

    }, []);

    return {processedProducts, loading, error, filters, handleFilterChange};

};
```

Các hook trên phụ thuộc lẫn nhau, hook `useProductFilterUI` sử dụng hook `useProductListLogic` để lấy dữ liệu đã qua xử lý, trong khi hook `useProductListLogic` sử dụng hook

`useFetchProducts` để lấy dữ liệu thô từ nguồn. Có thể thấy các hook trong cùng miền nghiệp vụ có sự tương tác chặt chẽ.

Để minh họa về tương tác giữa các hook khác miền nghiệp vụ với nhau, ta xét thêm một domain nghiệp vụ nữa là cart (giỏ hàng). Trong domain này, ta xét hook `useCartTotalPrice` dùng để lấy dữ liệu sản phẩm trong giỏ hàng và tính tổng giá của các sản phẩm trong giỏ hàng:

```
export const useCartTotalPrice = (initialCartData) => {
  const [cartItems, setCartItems] = useState([]);
  const totalPrice = useMemo(() => {
    setCartItems(initialCartData);
    return calculateTotal(cartItems);
  }, [initialCartData]);
  return { cartItems, totalPrice };
};
```

Hook `useCartTotalPrice` này và các hook trong domain nghiệp vụ product không trực tiếp tương tác với nhau, mà thông qua một UI component như ví dụ dưới đây (giả sử đã có hàm lấy sản phẩm trong giỏ hàng từ mảng product - `getProductsInCart`):

```
function ProductListingPage() {
  const [products, loading, error] = useProductListLogic({filter: "identity"})
  // Logic để lấy sản phẩm trong giỏ hàng
  const itemsInCart = getProductsInCart(products);
  const {cartItems, totalPrice} = useCartTotalPrice(itemsInCart);
  return (
    <div>
      {/* Display Cart items, totalPrice */}
    </div>
  )
}
```

Điều này thể hiện cách các hook từ các domain nghiệp vụ khác nhau có thể được sử dụng cùng nhau trong một component, duy trì sự tách biệt về mặt trách nhiệm của các hook, và không tương tác trực tiếp với nhau.

### 3.1.3. Theo tầng ứng dụng

Phương pháp này phân loại hooks dựa trên tầng trong kiến trúc ứng dụng mà chúng hoạt động. Đây là cách tiếp cận thường thấy trong các kiến trúc phức tạp hơn, nơi có sự phân tách rõ ràng giữa lớp dữ liệu, lớp nghiệp vụ và lớp trình bày.

- **Đặc điểm:** Các hooks được nhóm vào các thư mục như hooks/data-access (tương tác API/database), hooks/business-logic (chứa các quy tắc nghiệp vụ), hooks/ui-interaction (tương tác trực tiếp với UI). Ví dụ:
  - hooks/data-access: Chứa useFetchUser, useSortProduct.
  - hooks/business-logic: Chứa useShoppingCart, useOrderProcessing.
  - hooks/ui-interaction: Chứa useFormInput, useNavigation.
- **Ưu điểm:**
  - **Tách biệt mối quan tâm rõ ràng:** Thúc đẩy nguyên tắc Clean Architecture (các tầng bên trong không nên biết gì về các tầng bên ngoài, quan hệ phụ thuộc giữa các tầng nên hướng vào bên trong), giúp mỗi tầng có trách nhiệm duy nhất.
  - **Bảo trì dễ dàng:** Thay đổi ở tầng ngoài ít có khả năng ảnh hưởng đến các tầng trong.
- **Nhược điểm:**
  - **Phức tạp cho ứng dụng nhỏ:** Độ phức tạp có thể quá mức cần thiết đối với các dự án đơn giản.
  - **Khó phân loại một số hook:** Một số hook có thể giao thoa giữa các tầng, gây khó khăn trong việc quyết định vị trí chính xác.

#### Ví dụ về tương tác giữa các hooks trong chiến lược tổ chức theo tầng ứng dụng:

Các component thường được sử dụng ở tầng trình bày hoặc tầng ứng dụng (chứa quy tắc nghiệp vụ). Các hook ở tầng ứng dụng thường sử dụng hook ở tầng dữ liệu, tuân theo quy tắc phụ thuộc hướng vào trong của kiến trúc ứng dụng 3 tầng. Các hook sau đây được cài đặt giống như trong cách tổ chức hook theo domain nghiệp vụ, nhưng khác về cách tổ chức. Giả sử có một hàm dịch vụ fetchAllProducts được

sử dụng ở hook `useFetchProducts` được đặt trong thư mục `data-access` (chứa các hook xử lý ở tầng dữ liệu). Hook `useFetchProducts` được triển khai như sau:

```
export const useFetchProducts = () => {

  const [products, setProducts] = useState([]);

  const [loading, setLoading] = useState(true);

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

  useEffect(() => {

    const getProducts = async () => {

      setLoading(true);

      try {

        const data = await fetchAllProducts();

        setProducts(data);

      } catch (err) {

        setError(err);

      } finally {

        setLoading(false);

      }

    };

    getProducts();
  }, []);

  return { products, loading, error };
};
```

Ở tầng ứng dụng, ta có hook `useProductListLogic`, dùng để xử lý dữ liệu và trả ra mảng dữ liệu đã được xử lý đó. Giả sử có một hàm xử lý logic `applyLogic`, hook `useProductListLogic` được triển khai như sau:

```
export const useProductListLogic = (logic) => {

  const { products, loading, error } = useFetchProducts();

  const processedProducts = useMemo(() => {

    let result = [...products];

    result = applyLogic(result, logic);

    return result;
  }, [products, logic]);

  return { processedProducts, loading, error };
};
```

Ở tầng trình bày, một hook `useProductFilterUI` sử dụng hook `useProductListLogic` ở tầng ứng dụng để lấy mảng dữ liệu đã được xử lý dựa vào logic truyền vào, sau đó trả về mảng dữ liệu và cung cấp một hàm handler cho sự kiện thay đổi tiêu chí của bộ lọc filters, trong đó filters là một component UI chịu trách nhiệm hiển thị tiêu chí filter và người dùng có thể thay đổi các tiêu chí đó. Hook `useProductFilterUI` được triển khai như sau:

```
export const useProductFilterUI = () => {

  const [filters, setFilters] = useState({ category: '', minPrice: '' });

  const { processedProducts, loading, error } = useProductListLogic(filters);

  const handleFilterChange = useCallback((e) => {
    const { name, value } = e.target;
    setFilters(prev => ({ ...prev, [name]: value }));
  }, [setFilters]);
};
```

```
}, []);  
  
return {processedProducts, loading, error, filters, handleFilterChange};  
};
```

Trong ví dụ này, hook `useProductFilterUI` ở tầng trình bày chỉ quan tâm tới việc lấy được dữ liệu đã qua xử lý bằng việc sử dụng hook `useProductListLogic` ở tầng ứng dụng. Hook `useProductListLogic` lại gọi hook `useFetchProducts` ở tầng dữ liệu để lấy dữ liệu thô từ nguồn và xếp chúng thành một mảng. Điều này thể hiện sự phân tách rõ ràng giữa các tầng, mỗi hook chịu trách nhiệm cho một loại logic cụ thể ở tầng mà nó được xếp vào và chỉ tương tác với các tầng nằm bên trong và liền kề với nó (phụ thuộc hướng vào trong: tầng trình bày -> tầng ứng dụng -> tầng dữ liệu).

### 3.2. So sánh và đánh giá các chiến lược

Mỗi chiến lược tổ chức đều có những ưu và nhược điểm riêng, và không có một giải pháp nào là phù hợp cho tất cả. Việc lựa chọn phụ thuộc vào quy mô, độ phức tạp và yêu cầu cụ thể của dự án.

Nhìn chung, đối với các tiêu chí dễ tìm kiếm, module hóa, dễ mở rộng và giảm trùng lặp logic, cả ba cách tổ chức hook nêu trên đều có thể đáp ứng tốt. Đối với trường hợp xây dựng thư viện hook chứa các tiện ích chung, cách tổ chức theo chức năng là phù hợp nhất. Cách tổ chức theo hướng domain phát huy hiệu quả khi trong ứng dụng có nhiều phần độc lập rõ ràng, đặc biệt khi xây dựng ứng dụng theo kiến trúc Microservices. Trong khi đó, cách tổ chức theo tầng ứng dụng cung cấp sự tách biệt giữa logic các tầng, đặc biệt là các hệ thống lớn với cấu trúc phức tạp, đòi hỏi đáp ứng kiến trúc ứng dụng nhiều tầng (ví dụ như kiến trúc 3 tầng client-server, gồm 3 tầng là tầng dữ liệu, tầng logic và tầng trình bày).

## 4. Testing Custom Hooks

Việc kiểm thử (testing) là một giai đoạn không thể thiếu trong quy trình phát triển phần mềm, và Custom Hooks trong React cũng không phải là

ngoại lệ. Với bản chất là các hàm JavaScript mang trạng thái và logic tái sử dụng, Custom Hooks cần được kiểm thử một cách kỹ lưỡng để đảm bảo chúng hoạt động đúng như mong đợi và không gây ra lỗi tiềm ẩn trong ứng dụng.

#### 4.1. Lý do cần kiểm thử Custom Hooks

Kiểm thử Custom Hooks mang lại nhiều lợi ích quan trọng, góp phần nâng cao chất lượng, độ ổn định và khả năng bảo trì của ứng dụng. Việc kiểm thử Custom Hook đóng vai trò:

- **Đảm bảo tính đúng đắn của logic:** Custom Hooks thường đóng gói các logic phức tạp, bao gồm quản lý trạng thái, xử lý hiệu ứng phụ (ví dụ như gọi API, tương tác với DOM), và các quy tắc nghiệp vụ. Kiểm thử giúp xác minh rằng các logic này hoạt động chính xác trong mọi trường hợp, từ các kịch bản sử dụng bình thường đến các corner cases hoặc lỗi. Ví dụ: Một hook quản lý form (useForm) cần đảm bảo rằng việc xác thực dữ liệu, xử lý input, và submit form đều hoạt động đúng.
- **Đảm bảo tính mở rộng và module hóa:** Custom Hooks được thiết kế để tái sử dụng trên nhiều component và thậm chí nhiều dự án. Khi một hook được sử dụng rộng rãi, bất kỳ lỗi nào trong đó có thể ảnh hưởng đến nhiều phần của ứng dụng. Việc có một bộ kiểm thử mạnh mẽ giúp tăng độ tin cậy khi thực hiện các thay đổi, tái cấu trúc, hoặc thêm tính năng mới. Điều này giúp giảm thiểu rủi ro và chi phí bảo trì về lâu dài.
- **Xác minh tính nhất quán của hành vi:** Hooks có thể phụ thuộc vào lifecycle của React (như useEffect, useState, useCallback). Kiểm thử giúp đảm bảo rằng hook phản ứng đúng với các thay đổi của props, trạng thái nội bộ, hoặc các hiệu ứng theo thời gian. Ví dụ như useEffect trong một hook gọi API chỉ nên chạy khi các dependencies thay đổi đúng cách, và không gây ra vòng lặp vô hạn.
- **Dễ dàng phát hiện và sửa lỗi sớm:** Việc kiểm thử tự động được chạy thường xuyên trong quá trình phát triển (trên môi trường CI/CD). Điều này giúp phát hiện lỗi ngay khi chúng được đưa vào, giảm đáng kể thời gian và công sức để sửa chữa so với việc phát hiện lỗi ở giai đoạn sau (ví dụ như trên môi trường staging hoặc production).

## 4.2. Các công cụ kiểm thử phổ biến

Để kiểm thử Custom Hooks một cách hiệu quả, đã xuất hiện nhiều công cụ giúp mô phỏng môi trường React và tương tác với hooks một cách linh hoạt

### 4.2.1. Jest

Jest là một framework kiểm thử JavaScript phổ biến được phát triển bởi Facebook, thường được sử dụng rộng rãi trong các dự án React. Jest cung cấp một môi trường kiểm thử hoàn chỉnh bao gồm một test runner, một thư viện assertion (mong đợi), và các tính năng như mocking, snapshot testing. Với Custom Hooks, Jest được sử dụng để chạy các bài kiểm thử, quản lý test suite, và cung cấp các hàm expect để kiểm tra kết quả.

Ưu điểm của Jest:

- Cung cấp công cụ CLI với interactive mode để dễ dàng kiểm soát việc testing.
- Zero config: Jest được thiết kế để chạy ngay lập tức trên các dự án JavaScript nói chung và React nói riêng mà không cần cấu hình phức tạp.
- Hỗ trợ mocking và kiểm thử cô lập : Jest có khả năng cô lập các thành phần cần kiểm thử bằng cách giả lập (mocking) các phụ thuộc của chúng.
- Cung cấp tính năng snapshot testing mạnh mẽ, cho phép xác minh tính toàn vẹn của các đối tượng lớn hoặc cấu trúc UI mà không cần viết nhiều assertion chi tiết, giúp quá trình kiểm thử front-end trở nên hiệu quả hơn.

Cách sử dụng cho việc test Custom Hook:

Bản thân Jest không tự cung cấp môi trường cho React lifecycle, nên cần kết hợp sử dụng `@testing-library/react-hooks` hoặc `@testing-library/react` cho UI component để chạy các unit test. Jest ở đây đóng vai trò như một Test Runner, chỉ thực thi các bài unit test và cung cấp các hàm assertion.

### 4.2.2. React Testing Library

React Testing Library là một tập hợp các tiện ích giúp kiểm thử UI component và hooks theo cách tập trung vào trải nghiệm người dùng (blackbox testing)

- `@testing-library/react-hooks`: Là một tiện ích mở rộng cụ thể cho việc kiểm thử Custom Hooks độc lập với một component thực tế. Nó cung cấp hàm `renderHook` để tạo ra một môi trường React tối thiểu và quản lý lifecycle của hook.
- `@testing-library/react`: Được sử dụng để kiểm thử các React components, bao gồm cả các hooks được sử dụng bên trong component đó.

Ưu điểm:

- Tập trung vào hành vi người dùng: Giúp viết các bài kiểm thử mô phỏng cách người dùng tương tác với ứng dụng, thay vì kiểm thử chi tiết cách cài đặt.
- Đảm bảo tính khả dụng: Khuyến khích kiểm thử dựa trên các thuộc tính DOM mà người dùng (và công cụ hỗ trợ tiếp cận) có thể nhìn thấy, thay vì kiểm thử state nội bộ.
- Cung cấp môi trường cho lifecycle của Hooks: `renderHook` cho phép render lại hook và chờ đợi các side effects, giúp kiểm thử các hook phức tạp.

React Testing Library hỗ trợ kiểm thử Custom Hooks thông qua các hàm `renderHook` và `act`, cho phép gọi trực tiếp hook và đảm bảo các thay đổi trạng thái được xử lý chính xác. Kết quả trả về từ hook có thể được kiểm tra bằng các assertion qua việc sử dụng hàm `expect` để xác minh hành vi mong muốn. Do React Testing Library là một tập hợp các tiện ích, có thể cung cấp các hàm `renderHook` hay `act`, hỗ trợ việc kiểm thử các Custom Hook trong môi trường React, nên nó thường được sử dụng cùng với Jest để thực hiện kiểm thử.

### **4.3. Ví dụ kiểm thử cho các Custom Hook**

#### **4.3.1. Kiểm thử hành vi của hook khi xử lý logic hoặc dữ liệu**

Cách cài đặt cho hook `useProductCalculator` dưới đây giống như ở phần Adapter Pattern. Trước khi bắt đầu kiểm thử, ta mock data của các sản phẩm và mock hàm adapter cho dữ liệu đầu vào:

```
const mockProducts = [  
  {  
    productId: '1',  
    productName: 'Product A',  
    productDescription: 'Description A',  
    itemPrice: '10.00',  
    quantityInStock: '5',  
    productCategory: 'Category A',  
    lastUpdateDate: '2023-06-01T12:00:00Z',  
  },  
  {  
    productId: '2',  
    productName: 'Product B',  
    productDescription: 'Description B',  
    itemPrice: '20.00',  
    quantityInStock: '3',  
    productCategory: 'Category B',  
    lastUpdateDate: '2023-06-02T12:00:00Z',  
  },  
];  
  
const adaptedProducts = [  
  {  
    id: '1',  
    name: 'Product A',  
    description: 'Description A',  
    price: 10.0,  
    stock: 5,  
    category: 'Category A',  
    lastUpdate: '2023-06-01T12:00:00Z',  
  },  
  {  
    id: '2',  
    name: 'Product B',  
    description: 'Description B',  
    price: 20.0,  
    stock: 3,  
    category: 'Category B',  
    lastUpdate: '2023-06-02T12:00:00Z',  
  },  
];
```

```
{
  id: '1',
  name: 'Product A',
  description: 'Description A',
  price: 10.0,
  stock: 5,
  category: 'Category A',
  lastUpdated: '6/1/2023, 12:00:00 PM',
},
{
  id: '2',
  name: 'Product B',
  description: 'Description B',
  price: 20.0,
  stock: 3,
  category: 'Category B',
  lastUpdated: '6/2/2023, 12:00:00 PM',
},
];

const mockAdapterFunction = jest.fn();
mockAdapterFunction.mockImplementation((rawProduct) => {
  return {
    id: rawProduct.productId,
```

```

        name: rawProduct.productName,
        description: rawProduct.productDescription,
        price: parseFloat(rawProduct.itemPrice),
        stock: parseInt(rawProduct.quantityInStock, 10),
        category: rawProduct.productCategory,
        lastUpdated: new Date(rawProduct.lastUpdateDate).toLocaleString('en-US', {
          hour12: true,
          timeZone: 'UTC'
        }),
      };
    );
  );
}

```

## **Test Case 1: Tính toán và chuyển đổi chính xác khi có dữ liệu đầu vào hợp lệ**

**Input:** 2 sản phẩm hợp lệ và một adapter function.

**Expected output:**

- totalProducts = 8 ( $5 + 3$ )
- totalPrice = 110.0 ( $10 * 5 + 20 * 3$ )
- adaptedProducts phải khớp với mảng mẫu đã định nghĩa  
(result.current.adaptedProducts khớp với adaptedProducts đã được định nghĩa)

**Ví dụ code:**

```

it('should calculate total products, total price, and adapt products correctly', () => {
  const { result } = renderHook(() =>
    useProductCalculator(mockProducts, mockAdapterFunction)
  );
  expect(result.current.products.length).toBe(2);
  expect(result.current.products[0].name).toBe('Laptop');
  expect(result.current.products[0].price).toBe(1000);
  expect(result.current.products[0].stock).toBe(5);
  expect(result.current.products[1].name).toBe('Smartphone');
  expect(result.current.products[1].price).toBe(500);
  expect(result.current.products[1].stock).toBe(3);
  expect(result.current.totalProducts).toBe(8);
  expect(result.current.totalPrice).toBe(1100);
  expect(result.current.adaptedProducts).toEqual([
    {
      id: 1,
      name: 'Laptop',
      price: 1000,
      stock: 5
    },
    {
      id: 2,
      name: 'Smartphone',
      price: 500,
      stock: 3
    }
  ]);
});

```

```
);

expect(result.current.totalProducts).toBe(8);

expect(result.current.totalPrice).toBe(110.0);

expect(result.current.adaptedProducts).toEqual(adaptedProducts);

expect(mockAdapterFunction).toHaveBeenCalledTimes(2);

expect(mockAdapterFunction).toHaveBeenCalledWith(mockProducts[0]);

expect(mockAdapterFunction).toHaveBeenCalledWith(mockProducts[1]);

});
```

## Test Case 2: Xử lý danh sách sản phẩm rỗng

**Input:** Mảng rỗng [ ]

**Expected output:**

- totalProducts = 0
- totalPrice = 0
- adaptedProducts = [ ]

**Hành vi mong đợi khác:** Adapter function không được gọi.

**Ví dụ code:**

```
it('should return zero totals and empty array when products is empty', () =>
{

  const { result } = renderHook(() =>

    useProductCalculator([], mockAdapterFunction)

  );

  expect(result.current.totalProducts).toBe(0);

  expect(result.current.totalPrice).toBe(0);
```

```
    expect(result.current.adaptedProducts).toEqual([]);

    expect(mockAdapterFunction).not.toHaveBeenCalled();

});
```

### Test Case 3: Xử lý khi đầu vào là null hoặc undefined

**Input:** null (hoặc undefined)

**Expected output:**

- totalProducts = 0
- totalPrice = 0
- adaptedProducts = [ ]

**Hành vi mong đợi khác:** Adapter function không được gọi.

**Ví dụ code:**

```
it('should handle null or undefined products gracefully', () => {

  const { result } = renderHook(() =>

    useProductCalculator(null, mockAdapterFunction)

  );

  expect(result.current.totalProducts).toBe(0);

  expect(result.current.totalPrice).toBe(0);

  expect(result.current.adaptedProducts).toEqual([]);

  expect(mockAdapterFunction).not.toHaveBeenCalled();

});
```

### Test Case 4: Re-render khi dữ liệu thay đổi

## **Input:**

- Lần đầu: danh sách có 2 sản phẩm.
- Sau đó: thay đổi danh sách sang một danh sách sản phẩm mới (newProducts).

## **Expected output sau khi render lại:**

- totalProducts = 4 (Product C có 4 đơn vị)
- totalPrice = 60.0 ( $15 * 4$ )

**Hành vi mong đợi khác:** Adapter function được gọi tổng cộng 3 lần (2 lần đầu, 1 lần sau khi re-render).

## **Ví dụ code:**

```
it('should recalculate when products change', () => {

  const { result, rerender } = renderHook(
    ({ products }) => useProductCalculator(products, mockAdapterFunction),
    {
      initialProps: { products: mockProducts },
    }
  );

  expect(result.current.totalProducts).toBe(8);
  expect(result.current.totalPrice).toBe(110.0);

  const newProducts = [
    {
      productId: '3',
      productName: 'Product C',
      productDescription: 'Description C',
      itemPrice: '15.00',
    }
  ];

  rerender({ products: newProducts });

  expect(result.current.totalProducts).toBe(10);
  expect(result.current.totalPrice).toBe(150.0);
});
```

```

        quantityInStock: '4',
        productCategory: 'Category C',
        lastUpdateDate: '2023-06-03T12:00:00Z',
    },
];
rerender({ products: newProducts });

expect(result.current.totalProducts).toBe(4);
expect(result.current.totalPrice).toBe(60.0);
expect(mockAdapterFunction).toHaveBeenCalledTimes(3);
});

```

Kết quả kiểm thử của 4 test case:

```

PASS  src/hooks/adapters/__tests__/useProductCalculator.test.js
useProductCalculator
  ✓ should calculate total products, total price, and adapt products correctly (9 ms)
  ✓ should return zero totals and an empty array when products is empty (7 ms)
  ✓ should handle null or undefined products gracefully (4 ms)
  ✓ should recalculate when products change (11 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        0.647 s, estimated 1 s

```

### 4.3.2. Kiểm thử hành vi của hook trong tương tác với UI

Ở ví dụ này ta sẽ xét hook useProductHighlight (highlight các dòng trong bảng chứa sản phẩm khớp với tiêu chí). Ta có component như sau:

```

export default function ProductTable({ products }) {
  const [selected, setSelected] = useState("");
  const [highlightedProducts, setHighlightConfig] = useProductHighlight(
    products,

```

```
        strategies[selected].fn,  
  
        strategies[selected].config  
    );  
  
    const handleChange = e => {  
  
        // ...  
    };  
  
    return (  
        <div>  
  
            <select aria-label="highlight" value={selected}  
            onChange={handleChange}>  
  
                <option value="lowStock">Kho còn ít sản phẩm</option>  
  
                <option value="highPrice">Giá cao</option>  
  
                <option value="category">Theo loại sản phẩm</option>  
            </select>  
  
            <table>  
  
                <tbody>  
  
                    {highlightedProducts.map((p) => (  
  
                        <tr key={p.id} style={{ backgroundColor: p.isHighlighted ?  
                            '#ffeb3b' : 'white' }}>  
  
                            <td>{p.name}</td>  
  
                            <td>{p.price}</td>  
  
                            <td>{p.stock}</td>  
  
                            <td>{p.category}</td>  
                        </tr>  
                    ))}  
    
```

```
</tbody>

</table>

</div>

);

}
```

## Strategies:

```
const strategies = {

  lowStock: { fn: highlightLowStock, config: { threshold: 10 } },
  highPrice: { fn: highlightHighPrice, config: { minPrice: 100 } },
  category: { fn: highlightCategory, config: { category: 'Electronics' } },
};
```

Mảng products được mock như sau:

```
const products = [
  { id: '1', name: 'A', price: 50, stock: 2, category: 'Electronics' },
  { id: '2', name: 'B', price: 200, stock: 20, category: 'Books' },
];
```

**Test case 1: Highlight dòng chứa sản phẩm có số lượng trong kho ít hơn threshold (default là 10)**

**Input:** products; strategy lowStock

**Hành vi mong đợi:**

- Dòng đầu tiên (sản phẩm A) được highlight

- Dòng thứ hai (sản phẩm B) không được highlight

### Ví dụ code:

```
it('should highlight products with stock quantity lower han lowStock', () =>
{
  render(<ProductTable products={products} />);

  const rows = screen.getAllByRole('row');

  expect(rows[0]).toHaveStyle('background-color: #ffeb3b');

  expect(rows[1]).not.toHaveStyle('background-color: #ffeb3b');

});
```

### Test case 2: Highlight dòng chứa sản phẩm giá cao hơn minPrice (default là 100.00)

**Input:** products; strategy highPrice

### Hành vi mong đợi:

- Dòng đầu tiên (sản phẩm A, giá 50) không được highlight
- Dòng thứ hai (sản phẩm B, giá 200) được highlight

### Ví dụ code:

```
it('should highlight products with price higher than highPrice', () => {
  render(<ProductTable products={products} />);

  fireEvent.change(screen.getByLabelText('highlight'), { target: { value: 'highPrice' } });

  const rows = screen.getAllByRole('row');

  expect(rows[1]).toHaveStyle('background-color: #ffeb3b');

  expect(rows[0]).not.toHaveStyle('background-color: #ffeb3b');

});
```

### Test case 3: Highlight dòng chứa sản phẩm khớp với thẻ loại

Input: products; strategy category

Hành vi mong đợi:

- Dòng đầu tiên (sản phẩm A, loại electronics) được highlight
- Dòng thứ hai (sản phẩm B, loại Books) không được highlight

Ví dụ code:

```
it('should highlight based on category chosen', () => {  
  render(<ProductTable products={products} />);  
  
  fireEvent.change(screen.getByLabelText('highlight'), { target: { value: 'category' } });  
  
  const rows = screen.getAllByRole('row');  
  
  expect(rows[0]).toHaveStyle('background-color: #ffeb3b');  
  expect(rows[1]).not.toHaveStyle('background-color: #ffeb3b');  
});
```