Mini Project

Nghiên cứu khả năng tái sử dụng và mở rộng của React Custom Hook

12/6/2025

Giới thiệu Custom Hook

Custom Hook là gì

Hooks được giới thiệu từ React 16.8 để dùng state trong hàm.

Custom Hooks: Hàm bắt đầu bằng use, trích xuất & tái sử dụng logic.

Custom Hook không trả về UI, mà trả về dữ liệu và hàm điều khiển.

Lợi ích chính:

- Tái sử dụng logic trạng thái giữa các component.
- Tách biệt logic nghiệp vụ khỏi UI → đơn giản hóa component.
- Tăng khả năng đọc hiểu mã nguồn (readability).
- Dễ kiểm thử hơn logic gắn liền với UI.

Giới thiệu Custom Hook

Tầm quan trọng của tính tái sử dụng, mở rộng và module hóa

Tái sử dụng: Tiết kiệm thời gian, giảm lỗi, tăng nhất quán.

Mở rộng: Dễ thêm tính năng mới mà không cần viết lại hệ thống.

Module hóa: Tách logic thành đơn vị độc lập → dễ bảo trì & kiểm thử.

Custom Hooks là công cụ trung tâm giúp hiện thực hóa 3 nguyên tắc này.

Các mẫu thiết kế trong Custom Hook

Design Pattern là gì

Design Patterns là các giải pháp tổng quát, có thể tái sử dụng cho các vấn đề phổ biến trong thiết kế phần mềm. Dù React không hoàn toàn theo lập trình hướng đối tượng, nhưng các mẫu thiết kế vẫn có thể được áp dụng linh hoạt vào hệ thống Custom Hooks để:

- Tăng khả năng tái sử dụng logic.
- Thúc đẩy kiến trúc module hóa, dễ bảo trì.
- Hỗ trợ mở rộng và thay đổi yêu cầu nhanh chóng.

Composition Design Pattern

Ý tưởng:

- Kết hợp nhiều hàm/Hook nhỏ chuyên biệt thành một Hook lớn hơn.
- Tập trung vào việc "ghép" các phần nhỏ, mỗi phần đảm nhiệm một chức năng cụ thể.

Composition Design Pattern

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;

            }

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

        }

        setErrors(newErrors);

        return isValid;

    }, [validators]);

    const validate_errors = errors;

    return {validate_errors, validateAll, setErrors};

};
```

Composition Design Pattern

Ví dụ 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;
};
```

Composition Design Pattern

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);
```

Composition Design Pattern

Giải thích ví dụ

Trong ví dụ này, useValidation không chứa trực tiếp logic của các quy tắc xác thực (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.

Lợi ích

- Dễ mở rộng khi thêm trường hoặc quy tắc mới.
- Logic xác thực được chia nhỏ, giúp mã rõ ràng, dễ kiểm thử.

Factory Design Pattern

Ý tưởng:

- Tạo Hook theo cách sản xuất: truyền vào logic xử lý để sinh ra các Hook có hành vi khác nhau.
- Giúp tách biệt logic khởi tạo Hook và logic xử lý dữ liệu.

Factory Design Pattern

Factory function

```
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,

        };

    };

};
```

Factory Design Pattern

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' });
```

Factory Design Pattern

Giải thích ví dụ

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.

Lợi ích

- Mở rộng dễ dàng: thêm logic mới chỉ cần thêm hàm xử lý.
- Tái sử dụng cao: dùng chung 1 factory cho nhiều Hook.
- Giảm lặp code, hỗ trợ cấu hình linh hoạt.

Adapter Design Pattern

Ý tưởng:

- Biến đổi giao diện của dữ liệu từ nguồn bên ngoài thành định dạng mà ứng dụng có thể xử lý được.
- Ứng dụng: Custom Hook đóng vai trò "adapter" giúp chuẩn hóa dữ liệu từ JSON sang dạng ProductEntity.

Adapter Design Pattern

Định dạng dữ liệu thô vs dữ liệu được adapted

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

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

Adapter Design Pattern

Hàm adapter

```
const adaptProductForCalculation = (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(),

    };

};
```

Adapter Design Pattern

Hook áp dụng Adapter Design Pattern

```
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 };
```

Adapter Design Pattern

Giải thích ví dụ

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.

Lợi ích

- Giảm phụ thuộc vào định dạng gốc.
- Dễ bảo trì khi API thay đổi.
- Dữ liệu đầu vào rõ ràng, đồng nhất, dễ dùng.
- Dễ mở rộng: chỉ cần thêm hàm adapter mới khi có thay đổi định dạng

Strategy Design Pattern

Ý tưởng:

- Cho phép thay đổi thuật toán xử lý mà không cần sửa hook/component sử dụng.
- Ứng dụng: Hook useProductHighlight() nhận một hàm chiến lược (strategy) để xác định sản phẩm cần highlight.

Strategy Design Pattern

Hook useProductHighlight (đánh dấu các hàng chứa sản phẩm phù hợp với tiêu chí cụ thể)

```
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

    ];

};
```

Strategy Design Pattern

Ví dụ 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());

};
```

Strategy Design Pattern

Sử dụng hook useProductHighlight

```
const [highlightedProducts, highlightConfig, setHighlightConfig

    ] = useProductHighlight(

        displayedProducts,

        null,

        null

    );

setHighlightConfig({

     strategy: highlightHighPrice,

     config: { minPrice: 100.00 }

});
```

Strategy Design Pattern

Giải thích ví dụ

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

Lợi ích

- Dễ mở rộng hành vi xử lý.
- Tách biệt logic thuật toán và logic hiển thị.
- Tránh lồng ghép nhiều if-else, code sạch và linh hoạt.

Chiến lược tổ chức Custom Hook

Theo chức năng

Nhóm các hook theo vai trò như xử lý trạng thái, API, form, hay animation. Phù hợp với dự án nhỏ, dễ tiếp cận và dễ tìm kiếm khi số lượng hook còn ít.

Theo domain nghiệp vụ

Hook được chia theo từng miền nghiệp vụ của ứng dụng (user, product, order...), giúp tăng tính đóng gói và dễ mở rộng khi phát triển theo hướng domain-driven design.

Theo tầng ứng dụng

Hook được tổ chức theo tầng như data-access, business-logic, ui, tuân thủ kiến trúc như Clean Architecture. Phù hợp với ứng dụng lớn, giúp tách biệt rõ ràng trách nhiệm và tăng khả năng mở rộng.

Chiến lược tổ chức Hook

Ví dụ code tổ chức theo chức năng

```
export const useToggle = (initialValue = false) => {

   const [value, setValue] = useState(initialValue);

   const toggle = useCallback(() => setValue(prev => !prev), []);

   return [value, toggle];

};
```

```
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>

    );

}
```

Chiến lược tổ chức Hook

Ví dụ tổ chức theo domain nghiệp vụ

```
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};

};
```

Chiến lược tổ chức Hook

Ví dụ tổ chức theo tầng ứng dụng

```
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};

};
```

Chiến lược tổ chức Custom Hook

Mỗi chiến lược tổ chức Custom Hook đều có ưu nhược điểm riêng và phù hợp với từng loại dự án khác nhau. Cả ba cách (theo chức năng, theo domain, và theo tầng ứng dụng) đều hỗ trợ tốt các tiêu chí như dễ tìm kiếm, module hóa, mở rộng và giảm trùng lặp.

- Theo chức năng: Phù hợp với thư viện tiện ích chung, dễ quản lý các hook tái sử dụng.
- Theo domain: Hiệu quả với ứng dụng có các phần độc lập rõ ràng, phù hợp với kiến trúc Microservices.
- Theo tầng ứng dụng: Thích hợp cho hệ thống lớn, hỗ trợ kiến trúc nhiều tầng như 3-tier (data – logic – UI).

Việc chọn chiến lược nên dựa vào quy mô, độ phức tạp và đặc thù của dự án.

Thank you