Optimization & Testing

Trình bày: Bùi Bá Nguyên

Ryomo Vietnam Solutions Co. Ltd

Giới thiệu

- **Mục tiêu:** Tối ưu frontend (loading state, useCallback), kiểm thử backend (JUnit, Mockito).
- **Công cụ:**
  - Frontend: React Hooks, MUI.
  - Backend: JUnit, Mockito, Eclipse.
- **Tầm quan trọng:**
  - Tối ưu: Cải thiện trải nghiệm người dùng.
  - Kiểm thử: Đảm bảo chất lượng mã.

Tối ưu Frontend - Tối ưu với Loading State

- **Loading state** là cách cung cấp phản hồi trực quan cho người dùng khi ứng dụng đang thực hiện các tính toán phức tạp, hoặc chờ đợi phản hồi từ một tác vụ bất đồng bộ (như gọi API). Việc quản lý và hiển thị loading state hiệu quả ảnh hưởng trực tiếp đến **trải nghiệm người dung -** User Experience **(UX)**.

- **Tại sao cần sử dụng Loading State?**
  - Cải thiện trải nghiệm người dùng (UX):
    - Giảm sự khó chịu chờ đợi: người dùng không muốn thấy ứng dụng "treo" hoặc không phản hồi. Loading state cung cấp phản hồi trực quan, giúp họ biết rằng ứng hệ thống vẫn đang hoạt động.
    - Tương tác phản hồi: một ứng dụng phản hồi nhanh và cung cấp thông tin rõ ràng trong thời gian chờ đợi sẽ “giữ chân” người dùng tốt hơn.
    - Dự đoán thời gian chờ đợi (khi có thể): Một số loading state, như thanh tiến trình (progress bar), có thể ước tính và hiển thị mức độ hoàn thành của quá trình tải, giúp người dùng có cái nhìn rõ hơn về thời gian họ cần chờ đợi.

Tối ưu Frontend - Tối ưu với Loading State

- **Tại sao cần sử dụng Loading State?**
  - Hạn chế Lỗi Người dùng:
    - Ngăn chặn các tương tác không mong muốn (ví dụ: người dùng click nút gửi nhiều lần khi request trước đó đang được xử lý).
    - Đảm bảo tính toàn vẹn của dữ liệu
  - Hỗ trợ sửa lỗi và phát triển:
    - Phản hồi trực quan cho nhà phát triển: Trong quá trình phát triển, loading state giúp nhà phát triển dễ dàng nhận biết khi nào một yêu cầu dữ liệu đang diễn ra và khi nào nó đã hoàn thành, hỗ trợ việc gỡ lỗi và kiểm tra luồng dữ liệu.

Tối ưu Frontend - Tối ưu với Loading State

- **Cách Sử dụng Loading State trong Frontend**

  - **Khởi tạo State**: Định nghĩa các biến trạng thái cần thiết.
  - **Quản lý State khi Fetch Data**: Thay đổi trạng thái isLoading trước và sau khi gọi API.
  - **Conditional Rendering**: Hiển thị UI khác nhau dựa trên giá trị của isLoading và dữ liệu.

Tối ưu Frontend - Tối ưu với useCallback

- **useCallback**
  - Là một hook trong React dùng để memorize các hàm callback. Điều này có nghĩa là React sẽ ghi nhớ (tạo lại) một hàm chỉ khi các phần phụ thuộc của nó thay đổi.

- **Tại sao cần sử dụng useCallback?**
  - Trong React, khi một component re-render (render lại), tất cả các hàm được định nghĩa bên trong component đó cũng sẽ được tạo lại. Điều này có thể gây ra hai vấn đề chính:
    - Hiệu suất
    - Vòng lặp vô hạn (Infinite Loop)

Tối ưu Frontend

- **Kết Hợp Loading State và useCallback để Tối Ưu**
  - Sử dụng Loading State: Luôn cung cấp phản hồi trực quan cho người dùng trong suốt quá trình tải dữ liệu hoặc xử lý.
  - Sử dụng useCallback: Memoize các hàm xử lý sự kiện (như onClick, onChange, onSubmit) được truyền xuống các component con, đặc biệt là những component đã được tối ưu bằng React.memo. Điều này đảm bảo rằng các component con không bị re-render chỉ vì một prop hàm thay đổi tham chiếu.

Tối ưu Frontend

- **Thêm loading state và `useCallback`:**
  - Sử dụng MUI CircularProgress khi gọi API.
  - Cập nhật index.tsx:

```tsx
import { CircularProgress, Box } from '@mui/material';
const [isLoading, setIsLoading] = useState(false);
const fetchUsers = useCallback(async () => {
  setIsLoading(true);
  const response = await axios.get('http://localhost:8080/api/users');
  setUsers(response.data);
  setIsLoading(false);
}, []);
// JSX:
{isLoading ? (
  <Box sx={{ display: 'flex', justifyContent: 'center', mt: 4 }}>
    <CircularProgress />
  </Box>
) : (
  <UserList users={users} />
)}
```

Hàm để tải dữ liệu, được memoized bằng useCallback
Chỉ tạo lại khi isLoading thay đổi

Call API

Tối ưu Frontend - Demo

- **Tạo dữ liệu:**
  - **Thêm 1000 người dùng vào DB qua Postman:**

```
[
   {"name": "User 1", "email": "user1@example.com", "active": true},
    {"name": "User 2", "email": "user2@example.com", "active": false},
     ...
]
```

- **Kiểm tra:**
  - Gọi GET /api/users, xem loading state hoạt động.
  - Đảm bảo giao diện không lag khi render danh sách.
- **Mục đích:**
  - Kiểm tra hiệu suất frontend với dữ liệu thực tế.

Testing Backend - Test với Junit và Mockito

- **Junit**
  - Là một framework testing mã nguồn mở cho Java, được sử dụng rộng rãi để viết và chạy các bài kiểm thử đơn vị (unit tests). Nó cung cấp các chú thích (annotations) và phương thức (methods) tiện lợi để định nghĩa các trường hợp kiểm thử, thực thi chúng và đánh giá kết quả.

  - **Mục đích chính của JUnit** là giúp các nhà phát triển đảm bảo rằng từng đơn vị mã (ví dụ: một phương thức, một lớp) hoạt động đúng như mong đợi trong quá trình phát triển phần mềm. Điều này giúp phát hiện lỗi sớm, tăng chất lượng mã và giảm thiểu rủi ro khi thay đổi hoặc thêm tính năng mới.

Testing Backend - Test với Junit và Mockito

- **Các tính năng nổi bật của JUnit bao gồm**:
  - **Tạo bài kiểm thử dễ dàng:** Cung cấp các chú thích (annotations) đơn giản như @Test để đánh dấu một phương thức là một bài kiểm thử, @BeforeEach để thiết lập trước mỗi bài kiểm thử, @AfterEach để dọn dẹp sau mỗi bài kiểm thử, v.v.
  - **Các phương thức khẳng định (assertions):** Cung cấp một loạt các phương thức assert (ví dụ: assertEquals(), assertTrue(), assertNull()) để kiểm tra xem kết quả thực tế có khớp với kết quả mong đợi hay không.
  - **Tổ chức bài kiểm thử:** Cho phép nhóm các bài kiểm thử vào các bộ kiểm thử (test suites) và chạy chúng một cách có tổ chức.
  - **Tích hợp tốt:** Có thể dễ dàng tích hợp với các môi trường phát triển tích hợp (IDE) như Eclipse, IntelliJ IDEA, NetBeans và các công cụ xây dựng như Maven, Gradle.

Testing Backend - Test với JUnit và Mockito

- **Mockito**
  - Là một **framework kiểm thử mã nguồn mở phổ biến trong Java** được sử dụng để tạo các **đối tượng giả (mock objects)** trong các bài kiểm thử đơn vị (unit tests). Mục tiêu chính của Mockito là giúp bạn viết các bài kiểm thử đơn vị độc lập, tập trung vào việc kiểm tra logic của một thành phần cụ thể mà không bị phụ thuộc vào các thành phần khác.
- **Vì sao cần Mockito?**
  - Giả lập đối tượng (Mocking objects): Mockito giúp tạo ra các đối tượng giả (mock objects) để thay thế cho các đối tượng thật, từ đó cô lập phần mã cần kiểm thử khỏi các phụ thuộc bên ngoài như cơ sở dữ liệu, API, hoặc hệ thống khác.
  - Kiểm soát hành vi đối tượng: Bạn có thể chỉ định trước hành vi của một đối tượng mock, ví dụ như khi gọi một phương thức nào đó thì nó sẽ trả về kết quả gì. Điều này giúp kiểm tra các tình huống khác nhau một cách linh hoạt.
  - Xác minh tương tác (Verify interactions): Mockito hỗ trợ kiểm tra xem một phương thức cụ thể có được gọi hay không, được gọi bao nhiêu lần, hoặc có được gọi với tham số chính xác không.
  - Giảm độ phức tạp khi kiểm thử: Nhờ vào khả năng cô lập và điều khiển hành vi của các đối tượng phụ thuộc, Mockito giúp các bài kiểm thử trở nên đơn giản, rõ ràng và dễ duy trì hơn.
  - Tăng tốc độ viết test: Với cú pháp đơn giản, dễ hiểu, Mockito giúp tiết kiệm thời gian và công sức khi viết và bảo trì các bài test.

Testing Backend - Test với JUnit và Mockito

- **Thêm dependency trong build.gradle:**

```
dependencies {
            // JUnit 5
            testImplementation 'org.junit.jupiter:junit-jupiter-api:5.10.0’
            testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.10.0’
            // Mockito
            testImplementation 'org.mockito:mockito-junit-jupiter:5.10.0’
}
```

- **Tạo UserServiceTest.java:**

```
public class User {
            private Long id;
            private String name;
            private String email;
            private boolean active;
             // Getters and setters
            public Long getId() { return id; }
            public void setId(Long id) { this.id = id; }
            public String getName() { return name; }
            public void setName(String name) { this.name = name; }
            public String getEmail() { return email; }
            public void setEmail(String email) { this.email = email; }
            public boolean isActive() { return active; }
            public void setActive(boolean active) { this.active = active; }
}
```

Testing Backend - Test với JUnit và Mockito

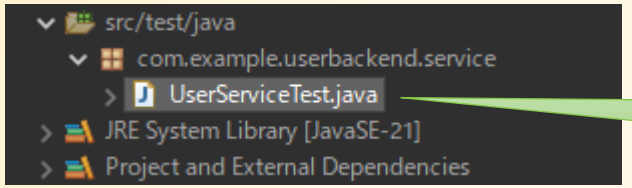
- **Tạo UserServiceTest.java:**

```java
package com.example.userbackend.service;I
mport com.example.userbackend.model.User;
import com.example.userbackend.mapper.UserMapper;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import java.util.Arrays;
import java.util.List;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;@ExtendWith(MockitoExtension.class)
public class UserServiceTest {
    @Mock
    private UserMapper userMapper;
    @InjectMocks
    private UserService userService;
    @Test
    public void testGetAllUsers() {
        User user = new User();
        user.setId(1L);
        user.setName("Test User");
        user.setEmail("test@example.com");
        user.setActive(true);
        List<User> users = Arrays.asList(user);
        when(userMapper.findAll()).thenReturn(users);
        List<User> result = userService.getAllUsers();
        assertEquals(1, result.size());
        assertEquals("Test User", result.get(0).getName());
    }
}
```

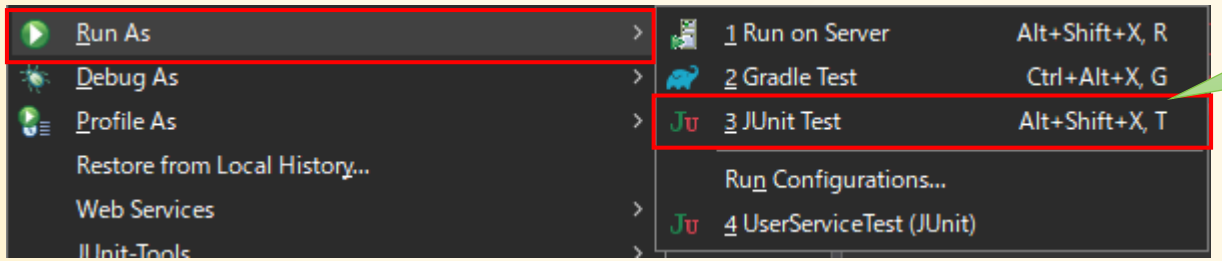
- **@Mock(Mockito):** Tạo ra một đối tượng "giả lập" (mock object) của một lớp hoặc interface.
- **@InjectMocks(Mockito):** "inject" các đối tượng @Mock đã được tạo vào một đối tượng "thực" mà ta muốn kiểm thử.
- **@Test (JUnit):** Đánh dấu một phương thức là một phương thức kiểm thử (test method).

Testing Backend - Test với JUnit và Mockito

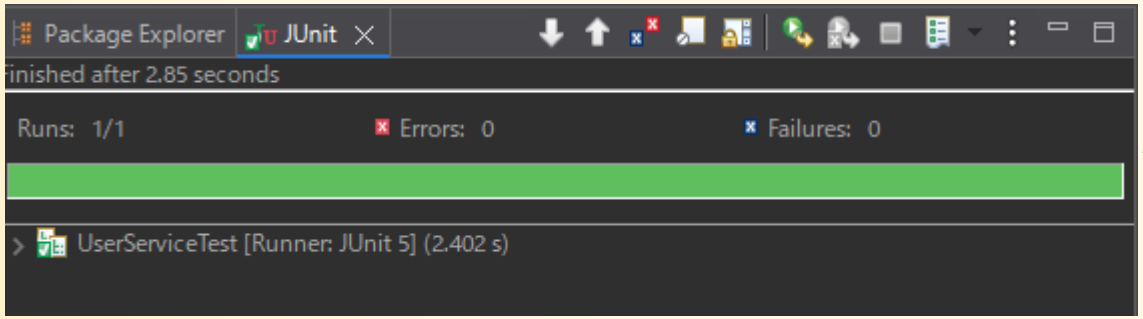
- **Thực thi test bằng Junit với class UserServiceTest.java:**



Click phải class UserServiceTest



Chọn Run As > JUnit Test



Kết quả tra về OK

Thực hành

- **Nhiệm vụ (45 phút):**
  - Thêm loading state vào index.tsx.
  - Viết unit test cho UserService (phương thức createUser).
  - Kiểm tra ứng dụng với 100 người dùng (thêm qua Postman).

- **Debug lỗi:**
  - Test thất bại: Kiểm tra mock (when, thenReturn).
  - Giao diện lag: Đảm bảo useCallback đúng dependency.

Tổng kết

- **Tổng kết:**
  - Tối ưu frontend (loading state, useCallback).
  - Kiểm thử backend với JUnit, Mockito.
- **Bài tập:**
  - Thêm try-catch cho API call trong index.tsx.
  - Viết unit test cho UserService (INSERT, UPDATE, DELETE).

Q&A