WEB GENERAL

WEB GENERAL

▼ Babel

- Babel là một trình biên dịch Javascript (source code =>output code), được dùng với mục đích chuyển đổi mã lệnh JavaScript được viết dựa trên tiêu chuẩn ECMAScript phiên bản mới (Như ES6, ES7,...) về phiên bản cũ hơn.
- Babel chạy trong 3 giai đoạn: parsing, transforming, and printing (Phân tích, chuyển đổi và in).
- **Tại sao lại cần sử dụng Babel?**
  - Ngôn ngữ JavaScript chủ yếu được chạy trên browser, còn browser thì có nhiều loại khác nhau như Chrome, Firefox, Internet Explore, Safari… tất cả đều có những quy định riêng để viết JavaScript. Nên khi code JavaScript của bạn có chạy ngon lành trên Chrome, thì chưa chắc có thể chạy được trên Internet Explore, Safari,...
  - Phiên bản phổ biến của ECMASCript đang được nhiều trình duyệt hỗ trợ hiên nay là ES5. Phiên bản kế tiếp ES6 mặc dù đã được chính thức ra đời tuy nhiên lại mới chỉ được một số trình duyệt hỗ trợ và không hoàn toàn đầy đủ.
  - Dễ hiểu, Babel là công cụ giúp ta viết code trên phiên bản **ECMASCript mới**, nhưng lại compiler ra phiên bản **ECMASCript cũ** để **tất cả browser có thể đều chạy được.**

▼ Build system (webpack, vite)

- **Webpack** là công cụ giúp bạn compile các module Javascript. Nó hay được gọi là “module bundler”.
  - Webpack là công cụ giúp gói gọn toàn bộ file js, jsx, img, css(bao gồm cả scss,sass,..)
  - Việc gói gọn không phải là lộn xộn hết cả lên mà nó được gói theo cấu trúc project, từ phần module này sang phần kia.

- Ngoài ra webpack còn rất nhiều chức năng hữu dụng khác nữa, như optimize hay tuỳ chọn chạy trên môi trường khác nhau(dev hoặc production), …
- Webpack nhận vào các module cùng với các dependencies và generate ra các static assets tương ứng.
- Việc sử dụng Webpack sẽ giúp project của chúng ta được optimize hơn rất nhiều.
- Triết lí cốt lõi:
    - **Mọi thứ đều là module**: khi làm việc với js, chúng ta thường tạo module ứng với 1 hoặc nhiều file js gộp lại. Thì đối với webpack thì những file như (CSS, Images, HTML) đều có thể trở thành module. Nó không khác gì khi chúng ta sử dụng file js cả. Cũng có những câu lệnh import module như **require(“myJSfile.js”)** or **require(“myCSSfile.css”).** Với tính cách module thì chúng ta có thể sử dụng nó ở bất kì ở đâu và có thể re-use khi cho ta muốn.
    - **Load only what you need and when you need**: Thông thường khi làm việc với js, chúng ta sử dụng rất nhiều module khác nhau. Với wepback sẽ gộp tất các cái module đó thành một file **"bundle.js". T**rong các ứng dụng thực tế file "bundle.js" có dung lượng lên đến **"10MB-15MB"**, đều này không tốt khi sử dụng cho website. Khi client request sẽ load rất lâu dẫn đến trải nghiệm người dùng đối với ứng dụng không tốt. Webpack hiểu ra điều đó nên webpack có vài tính năng chia nhỏ file "bundle" thì nhiều file khác nhau ứng với từng mục đích khác nhau. Việc chia nhỏ vậy, sẽ giúp chúng ta cần load những gì và khi nào cần sử dụng nó.
- Ưu điểm
    - Giúp cho cho project dễ dàng phát triển, quản lý, customize
    - Tăng tốc độ cho project
    - Phân chia các module và chỉ load khi cần
    - Đóng gói tất cả file nguồn thành một file duy nhất, nhờ vào loader mà có thể biên dịch các loại file khác nhau
    - Biến các tài nguyên tĩnh (image, css) trở thành 1 module

        - Chuyển đổi các mã nguồn : JSX, less, sass, scss thành js, ... ES6 -> ES5 thông qua babel transpiler ...
    - **Vite** là một tool mới ra mắt cùng vue3 được phát triển bởi evan you. Về chức năng thì cũng na ná như vue-cli tuy nhiên có một số điểm khác biệt như:
        - vite không based trên webpack
        - DevServer sử dụng native ES modules trên trình duyệt.
        - Vite build sử dụng Rollup, thằng này cũng được đánh giá khá nhanh
        - Nhược điểm: kén browser, kén dependencies, còn một số lỗi ở môi trường production, ...

▼ **OOP là gì? Các thuộc tính ?**

- **OOP** là phương pháp lập trình lấy đối tượng làm nền tảng để xây dựng chương trình (hoặc là phương pháp lập trình dựa trên kiến trúc **lớp** (class) và **đối tượng** (object))
- Trong lập trình hướng đối tượng, **đối tượng** được hiểu như là 1 thực thể: người, vật hoặc 1 bảng dữ liệu, . . .
- Một đối tượng bao gồm 2 thông tin: **thuộc tính** và **phương thức**.
    - **Thuộc tính** chính là những thông tin, đặc điểm của đối tượng. Ví dụ: một người sẽ có họ tên, ngày sinh, màu da, kiểu tóc, . . .
    - **Phương thức** là những thao tác, hành động mà đối tượng đó có thể thực hiện. Ví dụ: một người sẽ có thể thực hiện hành động nói, đi, ăn, uống, .
- **Class**
    - Các đối tượng có các đặc tính tương tự nhau được gom lại thành 1 **lớp đối tượng**.
    - Bên trong lớp cũng có 2 thành phần chính đó là thuộc tính và phương thức.
    - Ngoài ra, lớp còn được dùng để định nghĩa ra kiểu dữ liệu mới.
- **Lớp** là một khuôn mẫu còn **đối tượng** là một thể hiện cụ thể dựa trên khuôn mẫu đó.
- 4 Tính chất của OOP:

    - **Tính đóng gói (encapsulation): c**ác dữ liệu và phương thức có liên quan với nhau được đóng gói thành các lớp để tiện cho việc quản lý và sử dụng. Ngoài ra, đóng gói còn để che giấu một số thông tin và chi tiết cài đặt nội bộ để bên ngoài không thể nhìn thấy.
    - **Tính trừu tượng (abstraction)** : Khi viết chương trình theo phong cách hướng đối tượng, việc thiết kế các đối tượng ta cần rút tỉa ra những đặc trưng chung của chúng rồi trừu tượng thành các interface và thiết kế xem chúng sẽ tương tác với nhau như thế nào.
    - **Tính kế thừa (inheritance):** Lớp cha có thể chia sẽ dữ liệu và phương thức cho các lớp con, các lớp con khỏi phải định nghĩa lại, giúp chương trình ngắn gọn.
    - **Tính đa hình (polymorphism):** Là hiện tượng các đối tượng thuộc các lớp khác nhau có thể hiểu cùng một thông điệp theo các cách khác nhau.
- **Lớp trừu tượng** là lớp được khai báo mà không thể tạo ra đối tượng từ lớp đó. Ta sẽ tạo những lớp con kế thừa lớp trừu tượng.
    - Mục đích lớp trừu tượng là tạo ra lớp chung cho những lớp có liên quan với nhau kế thừa.
    - Ví dụ khi xây dựng phần mềm quản lý nhà trường: Những lớp sinh viên, giảng viên, cán bộ,… có những thuộc tính và phương thức chung như tên, năm sinh, quê quán,… thì ta sẽ tạo một lớp con người là lớp trừu tượng và những đặc điểm chung được để trong lớp con người. Khi phát triển chương trình, ta chỉ có thể tạo các đối tượng từ lớp con kế thừa lớp con người; không thể cho tạo đối tượng từ lớp con người được
- **Các phương thức trừu tượng** là là chỉ định nghĩa mà không có chương trình bên trong, lớp con kế thừa phải bắt buộc override nó lại để sử dụng. Phương thức trừu tượng có ý nghĩa định nghĩa phương thức bắt buộc phải có trong lớp con kế thừa.

▼ **Từ khóa static làm gì?**

- Khi ta khai báo các thuộc tính, phương thức thì nó chỉ được sử dụng khi khởi tạo đối tượng, thông tin cũng thuộc đối tượng đó.
- Có những lúc, ta cần những thông tin chung cho tất cả các đối tượng. Có nghĩa những thông tin đó lưu ở một vùng nhớ duy nhất.

- Từ khóa static sử dụng để quản lý bộ nhớ, khi những thành viên bên trong một lớp có từ khóa **static** thì nó thuộc về lớp, không phải thuộc về riêng một đối tượng nào đó.

▼ **Nguyên lý solid là gì?**

> 💡 ***SOLID** là năm nguyên lý cơ bản trong thiết kế phần mềm hướng đối tượng, giúp code trở nên dễ hiểu, mềm dẻo và dễ bảo trì hơn. Tác giả của SOLID là kỹ sư phần mềm nổi tiếng Robert C. Martin.*

https://kipalog.com/posts/Tim-hieu-nhanh-SOLID-than-thanh

- **S - Single Responsibility Principle (SRP):**

  > 💡 ***A class should have only a single responsibility.***

  - Ý tưởng của nguyên lý này là giúp chúng ta giảm đi sự phức tạp của class: một class chỉ nên phục vụ một mục đích duy nhất

- **O - Open Closed Principle (OCP):**

  > 💡 ***Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.***

  - Ý tưởng của nguyên lý này là khi triển khai các tính năng mới, thay vì sửa đổi code đã tồn tại, chúng ta nên mở rộng/kế thừa.

- **L - Liskov Substitution Principle(LSP):**

  > 💡 ***Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.***

  - Nguyên lý này có thể hiểu là các đối tượng của class cha có thể được thay thế bởi các đối tượng của các class con mà không làm thay đổi tính đúng

đắn của chương trình.

- **I - Interface Segregation Principle (ISP):**

> 💡 ***Many client-specific interfaces are better than one general-purpose interface.***

  - Nguyên lý này có thể hiểu là thay vì viết một interface cho một mục đích chung chung, chúng ta nên tách thành nhiều interface nhỏ cho các mục đích riêng.
  - Chúng ta không nên bắt buộc client phải implement các method mà client không cần đến.

- **D - Dependency Inversion Principle:**

> 💡 ***Depend on abstractions, not on concretions.***

  - Ý tưởng của nguyên lý này là các module cấp cao không nên phụ thuộc vào các module cấp thấp, cả hai nên phụ thuộc vào abstraction.

Microservice - Micro-frontend

▼ **Sự khác nhau giữa monolithic và microservice**

- **Monolithic** là kiến trúc cũ trước đây thường sử dụng, nó là kiến trúc dạng nguyên khối, nghĩa là mọi tính năng đều sẽ nằm trong 1 project

  ▼ **Ưu điểm:**

  - Dễ phát triển vì các stack công nghệ thống nhất với nhau
  - Testing, deploy cũng tương đối đơn giản

- Dễ scale vì có thể tạo nhiều instance cho load balancer
- Về mặt cơ sở hạ tầng (infrastructure) đơn giản. Chỉ cần 1 container cũng có thể chạy ứng dụng
- Team size nhỏ

▼ **Nhược điểm:**

- Các component nó liên kết chặt chẽ với nhau nên khi có 1 thay đổi trong 1 component nào đó cũng có thể ảnh hưởng tới component khác
- Khi project trở nên lớn dần. Các tính năng mới sẽ mất nhiều thời gian để phát triển cũng như maintain những tính năng hiện có cũng sẽ gặp nhiều khó khăn
- Áp dụng công nghệ mới sẽ rất khó khăn vì toàn bộ ứng dụng phải thay đổi.
- Không hề dễ để hiểu project do các module liên quan chặt chẽ lẫn nhau. Một issue nhỏ cũng có thể làm chết toàn bộ ứng dụng.
- Có thời gian khởi động lâu và tốn tài nguyên CPU cũng như bộ nhớ.
- Các team tham gia vào dự án phải phụ thuộc lẫn nhau và tất khó để mở rộng quy mô team.

- **Microservice** là kiến trúc mới, chia dự án thành nhiều service nhỏ. Mỗi service sẽ độc lập với nhau. Có thể có kiến trúc khác nhau, hoặc sử dụng công nghệ khác nhau, hoặc dùng cả database khác nhau. Chúng giao tiếp với nhau thông qua môi trường mạng như restful API hoặc message queue

▼ **Ưu điểm:**

- Các component có kết nối lỏng lẻo dẫn đến dễ cách ly, dễ test và khởi động nhanh.
- Vòng đời phát triển nhanh hơn. Tính năng mới được phát triển nhanh hơn và tính năng cũ được cấu trúc lại dễ hơn.
- Các service có thể deploy độc lập nên ứng dụng dễ đọc, dễ tạo các bản vá hơn.
- Những issue, ví dụ liên quan đến memory leak một trong các service, bị cô lập và có thể không làm sập ứng dụng.

- Việc áp dụng các công nghệ mới dễ hơn. Các component có thể được nâng cấp độc lập với nhau.
- Các mô hình scale phức tạp và hiệu quả hơn có thể được thiết lập. Các service quan trọng có thể scale hiệu quả hơn. Các component riêng sẽ khởi động nhanh hơn và cải thiện thời gian khởi động của cả hệ thống.
- Các team tham gia sẽ ít phụ thuộc lẫn nhau. Kiến trúc này rất thích hợp cho các đội Agile.

▼ **Nhược điểm:**

- Phức tạp hơn về mặt tổng thể vì các component khác nhau có các stack công nghệ khác nhau nên buộc team phải tập trung đầu tư thời gian để theo kịp công nghệ.
- Khó thực hiện test end-to-end và integration test vì có nhiều stack công nghệ khác nhau.
- Deploy toàn bộ ứng dụng phức tạp hơn vì có nhiều container và nền tảng ảo hóa liên quan.
- Ứng dụng được scale hiệu quả hơn nhưng thiết lập nâng cấp sẽ phức tạp hơn vì nó sẽ yêu cầu nâng cao nhiều tính năng như truy tìm dịch vụ (service discovery), định tuyến DNS,…
- Yêu cầu một team-size lớn để maintain ứng dụng vì có nhiều component và công nghệ khác nhau.
- Các thành viên trong team chia sẻ các skill khác nhau dựa trên component họ làm nên sẽ tạo ra sự khó khăn khi thay thế và chia sẻ kiến thức.
- Stack công nghệ phức tạp và khó để học hơn.
- Thời gian phát triển ban đầu là chậm nên thời gian để có thể làm marketing lâu hơn.
- Yêu cầu cơ sở hạ tầng phức tạp. Thông thường sẽ yêu cầu nhiều container (Docker) và nhiều máy JVM để chạy.

▼ **Vai trò của Docker trong Microservices?**

- Docker thường cung cấp một môi trường container, trong đó bất kỳ ứng dụng nào cũng có thể được host.

- Điều này được thực hiện bằng cách đóng gói chặt chẽ cả ứng dụng và các phụ thuộc cần thiết để hỗ trợ nó.
- Các sản phẩm đóng gói này được gọi là Container và vì Docker đã quen với việc đó nên chúng được gọi là Docker container.
- Về bản chất, Docker cho phép bạn chứa các microservice của mình và quản lý các microservices này dễ dàng hơn.

▼ **Giải thích về OAuth và OAuth2?**

- **OAuth** là một phương thức xác thực giúp một ứng dụng bên thứ 3 có thể được ủy quyền bởi người dùng để truy cập đến tài nguyên người dùng nằm trên một dịch vụ khác. OAuth là từ ghép của O(Open) và Auth tượng trưng cho:
  - *Authentication*: xác thực người dùng.
  - *Authorization*: cấp quyền truy cập đến tài nguyên mà người dùng hiện đang nắm giữ.
- **OAuth2** là bản nâng cấp của **OAuth1.0**, là một giao thức chứng thực cho phép các ứng dụng chia sẻ một phần tài nguyên với nhau mà không cần xác thực qua username và password như cách truyền thống từ đó giúp hạn chế được những phiền toái khi phải nhập username, password ở quá nhiều nơi hoặc đăng ký quá nhiều tài khoản mật khẩu mà chúng ta chẳng thể nào nhớ hết.

▼ **Giải thích cách microservice giao tiếp với các phần khác?**

- Giao tiếp giữa các microservice có thể thực hiện:
  - HTTP/REST với JSON hoặc giao thức nhị phân cho request/response.
  - Websocket cho streaming
  - Một broker hoặc server dùng cho các thuật toán routing.

→ RabbitMQ, Kafka,... có thể dùng như một message broker, mỗi cái được xây dựng để xử lý message cụ thể.

▼ **Các thành phần chính trong Microservices?**

- Containers, Clustering, và Orchestration.
- IaC [Infrastructure as Code Conception]
- Cloud Infrastructure

- API Gateway
- Enterprise Service Bus
- Service Delivery

▼ **Micro frontend là gì?**

- Ý tưởng của Micro Frontends cũng giống như microservice ở phía BE đó là sẽ phân tách các ứng dụng này thành các phần kết hợp của các tính năng, mỗi tính năng có thể được phát triển bởi một team độc lập.
- Trước đó có các mô hình phát triển phần mềm trước khi có micro fe:
  - **Monolithic**: một team phát triển toàn bộ các thành phần của sản phẩm từ Database, Backend, Frontend
  - **Front & Back:** chia team phát triển thành 2 team FE và BE
  - **Microservices:** chúng ta chia nhỏ các chức năng thành các dịch vụ riêng để thuận tiện cho quá trình phát triển. Tuy nhiên việc phần chia các dịch vụ này chỉ ở phần backend cho nên phía frontend vẫn phải phát triển chung các chức năng với nhau ở một bộ source code.

  →**Mô hình Micro frontends**: mỗi team sẽ phát triển các sản phẩm độc lập (từ Database, Backend đến Frontend). Sau đó tích hợp các sản phẩm độc lập này lại với nhau thành một sản phẩm chung.

▼ **Khi nào nên dùng Micro FE?**

- Một sản phẩm có nhiều module chức năng và bạn muốn nhiều team có thể phát triển cùng lúc
- Có thể bạn sẽ muốn phát triển một progressive hoặc responsive web application nhưng bạn gặp khó khăn trong việc tích hợp vào source code hiện tại của mình
- Có thể bạn muốn sử dụng một thư viện mới để tăng tốc quá trình phát triển sản phẩm của mình (vd: trước đó sử dụng Angularjs (1.x) để phát triển và hiện tại muốn sử dụng ReactJS để phát triển)
- Bạn muốn sử dụng một thư viện mới để hỗ trợ cho các chức năng sản phẩm, như sử dụng Webpack 5.x nhưng project hiện tại đang sử dụng Webpack 3.x và khó có thể nâng cấp lên Webpack 5.x được vì có khá nhiều dependence bị ảnh hưởng.

- Có thể bạn muốn tăng tốc quá trình phát triển sản phẩm bằng cách nhiều team khác nhau tham gia vào phát triển một sản phẩm cùng lúc bằng việc tách ra nhiều module và phát triển độc lập.

▼ **Một số ưu điểm và nhược điểm của Micro Frontends là gì ?**

- **Ưu điểm:**
  - Tách biệt các module chức năng thành nhiều phần source code riêng biệt. Từ đó giảm các dependencies ở mỗi project, lượng code sẽ ít hơn, giúp cho quá trình build deploy nhanh hơn và các file js bundle cũng sẽ nhẹ hơn
  - Có khả năng mở rộng một cách dễ dàng bằng cách nhiều team cùng tham gia.
  - Có thể sử dụng các thư viện, framework khác nhau (React, Angular) để phát triển các module khác nhau của một dự án.
  - Có khả năng cập nhật, nâng cấp thư viện hoặc phát triển lại một phần nào đó của dự án.
  - Dễ dàng kiểm thử (testing) các chức năng một cách độc lập.
- **Nhược điểm:**
  - Chia nhỏ các dự án sẽ dẫn tới trùng lập các dependencies hoặc source code
  - Nhiều team phát triển nên khó trong việc quản lý source code nếu không có quy định chung rõ ràng từ ban đầu.

▼ **Một số phương pháp triển khai Micro Frontends**

https://micro-frontends.tuando.net/

▼ **Build-time integration**

- là việc coi các ứng dụng như một package và ứng dụng chính sẽ thêm các ứng dụng con như một thư viện như sau:

```
{
  "name": "@micro-frontends/container",
  "version": "1.0.0",
  "description": "Micro frontends demo",
  "dependencies": {
    "@micro-frontends/products": "^1.2.3",
    "@micro-frontends/checkout": "^4.5.6",
    "@micro-frontends/user-profile": "^7.8.9"
```

```
  }
}
```

- Cách tiếp cận này có một số hạn chế như:
  - Chúng ta sẽ phải re-compile (bundle) các ứng dụng chính và release lại mỗi khi các ứng dụng con có thay đổi (release version mới từ 0.0.1 ⇒ 0.02)
  - Không có sự đồng bộ chức năng giữa các ứng dụng chính nếu chúng ta bỏ xót quá trình đồng bộ version của ứng dụng con (Cũng có thể là một điểm lợi nếu chúng ta không muốn nâng cấp chức năng ở một trang nào đó)
  - Phụ thuộc các dependencies với nhau
    - Nếu project `@micro-frontends/container` sử dụng React và `@micro-frontends/products` cũng sử dụng React thì sẽ bị trùng lập thư viện và tăng dung lượng khi tải trang web
    - Nếu project `@micro-frontends/container` sử dụng React và `@micro-frontends/products` sử dụng chung React với project chính thì sẽ bị phụ thuộc vào version của project chính.

▼ **Run-time integration via iframes**

```
<html>
  <head>
    <title>Micro frontends</title>
  </head>
  <body>
    <h1>Welcome to Micro frontends</h1>

    <iframe id="micro-frontend-container"></iframe>

    <script type="text/javascript">
      const microFrontendsByRoute = {
        '/': 'https://micro-frontends.tuando.net/demo/react-example',
        '/products': 'https://micro-frontends.tuando.net/demo/react-example/products'
      };

      const iframe = document.getElementById('micro-frontend-container');
      iframe.src = microFrontendsByRoute[window.location.pathname];
    </script>
  </body>
</html>
```

- Mỗi lần thay đổi url từ `/` sang `/products` phần nội dụng của trang sẽ được tải lại bởi một nội dung từ domain khác, trong ví dụ là `/demo/react-example/products`
- Ưu điểm:
  - Không bị ảnh hưởng bởi styles (CSS) giữa các trang chính và trang trong iframe
- Hạn chế:
  - Phải tải lại toàn bộ trang khi thay đổi đường dẫn
  - Khó khăn trong việc giao tiếp giữa các chức năng

▼ **Run-time integration via JavaScript**

- Các tiếp cận này là việc chúng ta khai báo các global function hỗ trợ render các chức năng ở dự án con. Sau đó ở dự án chính ta sẽ gắn các script bundle file của các dự án con, tiếp theo cần hiện thị chức năng nào thì chỉ việc gọi chức năng đó thôi.

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";

window.renderProducts = (containerId, history) => {
    ReactDOM.render(
        <App history={history} />,
        document.getElementById(containerId),
    );
};
```

▼ **Run-time integration via Web Components**

- Cách tiếp cận này cho phép chúng ta khai báo một HTML Custom Element, ví dụ như ta khai báo một HTML Custom Element `<web-components-products></web-components-products>` thì chỗ nào muốn sử dụng ta chỉ cần chèn đoạn mã `<web-components-products></web-components-products>` là có thể sử dụng được rồi.
- Ưu điểm:

    - Không bị phụ thuộc dependencies giữa các dự án với nhau (ví dụ: khác version React giữa các dự án)
    - Vì cho phép tạo một HTML Custom Element nên ta có thể gắn thẻ HTML Custom này vào bất cứ đoạn mã HTML nào, không quan trọng dự án đó đang sử dụng frontend framework nào
    - Hỗ trợ Shadow DOM: cho phép style css độc lập, không ảnh hưởng css giữa các dự án với nhau
    - Có thể phát triển theo hướng package (publish lên một registry) mà không cần phải có domain host cho dự án vì vậy đơn giản trong việc quản lý các version release.
- Hạn chế:
    - Không thể chia sẻ tài nguyên giữa các dự án với nhau (ví dụ: sử dụng chung thư viện React)

▼ **Module Federation Webpack 5**

- Module Federation là một tính năng mới của Webpack 5. Nó cho phép chúng ta cấu hình để một ứng dụng có thể dynamic load code từ một ứng dụng khác.
- Hiểu đơn giản là chúng ta có 2 ứng dụng được phát triển độc lập A và B, ứng dụng B là một phần nhỏ chức năng của ứng dụng A. Module Federation sẽ cho phép ta nhúng ứng dụng B và ứng dụng A và chia sẻ tài nguyên giữa chúng.
- Ưu điểm:
    - Có thể chia sẻ tài nguyên giữa các dự án. Ví dụ dự án A sử dụng React 16.x và dự án B cũng sử dụng React 16.x thì khi tải module B sẽ không cần phải tải thêm React một lần nữa, nếu 2 version khác nhau thì nó sẽ tự động tải thêm version React còn thiếu.
    - Giao tiếp giữa các dự án một cách đơn giản, có thể sử dụng chung một Redux store giữa các dự án với nhau
- Hạn chế:
    - Các dự án phải sử dụng Module Federation của Webpack 5.x

- Buộc phải các dự án phải có các static domain để tải các bundle file tương ứng. Vì các chức năng Module Federation chỉ hỗ trợ cấu hình tải các file từ một remote url

GIT

▼ **Tìm hiểu một số câu lệnh git**

http://thaunguyen.com/blog/software/giai-thich-chi-tiet-nhung-cau-lenh-thuong-dung-trong-git#git_revert

▼ **Git fork là gì ? Sự khác nhau giữa git fork, branch và clone?**

- **Git fork:** là một bản copy của một repository (Kho chứa source code của bạn trên Github). Việc fork một repository cho phép bạn dễ dàng chỉnh sửa, thay đổi source code mà không ảnh hưởng tới source gốc.
- **Git clone**: khác với fork; nó là một bản remote local copy của một số kho lưu trữ. Khi bạn sao chép, bạn đang sao chép toàn bộ repo, bao gồm tất cả lịch sử và các nhánh.
- **Git branch**: là một cơ chế để xử lý các thay đổi trong một kho lưu trữ duy nhất để cuối cùng merger chúng với phần còn lại của code. Branch là cái gì đó nằm trong một repo. Về mặt khái niệm, nó đại diện cho một luồng phát triển.

▼ **Sự khác nhau giữa "git pull" and "git fetch"?**

- Nhìn chung, git pull thực hiện git fetch theo sau là git merge
- Khi bạn sử dụng **pull:**
  - git sẽ cố gắng tự động thực hiện công việc của bạn cho bạn.
  - Vì vậy Git sẽ merger bất kỳ commit ở trong nhánh bạn đang làm việc.
  - Tự động merger các commit mà không cho phép bạn xem chúng trước.
  - Nếu bạn không quản lý chặt chẽ các branch của mình, bạn có thể gặp phải conflicts thường xuyên.
- Khi bạn **fetch:**

    - git tập hợp bất kỳ commit nào từ target branch không tồn tại trong nhánh hiện tại của bạn và lưu trữ chúng trong local repository của bạn.
    - Tuy nhiên, nó không merger chúng với nhánh hiện tại của bạn.
    - Điều này đặc biệt hữu ích nếu bạn cần cập nhật kho lưu trữ của bạn, nhưng đang làm việc trên project rất dễ có thể bị ảnh hưởng nếu bạn cập nhật các file của mình.
    - Để merger các commit vào nhánh chính của bạn, bạn sử dụng merger.

▼ **Làm thế nào để revert previous commit trong git?**

- sử dụng git reset

▼ **"git cherry-pick" là gì?**

- Lệnh git cherry-pick thường được sử dụng để xem các commit cụ thể từ một nhánh trong một repo trên một nhánh khác.
- Việc sử dụng phổ biến là commit chuyển tiếp hoặc back-port commits từ maintenance đến branch phát triển.
- Điều này trái ngược với các cách khác như merger và rebase mà thường áp dụng nhiều commit vào một nhánh khác.

▼ **Khi nào nên sử dụng "git stash"?**

- Lệnh git stash thực hiện uncommitted changes của bạn (both staged and unstaged), lưu chúng lại để sử dụng sau này và sau đó chuyển đổi chúng từ from your working copy.
- Ta có thể sử dụng stashing là nếu ta phát hiện ra đã quên một gì đó trong lần commit cuối cùng và đã bắt đầu làm việc trên nhánh tiếp theo trong cùng một nhánh

▼ **Khi nào bạn sử dụng "git rebase" thay vì "git merge"?**

- Cả hai lệnh này được thiết kế để tích hợp các thay đổi từ một nhánh này sang một nhánh khác.
- Khi nào dùng:
    - Nếu bạn có bất kỳ nghi ngờ, sử dụng merge.

- Sự lựa chọn cho rebase or merge dựa trên những gì bạn muốn lịch sử của bạn trông như thế nào.

▼ **Nhiều yếu tố cần xem xét:**

- Nhánh bạn có đang nhận được những thay đổi từ việc chia sẻ với các developers khác bên ngoài nhóm của bạn (ví dụ: nguồn mở, công khai) không?

  → Nếu vậy, đừng rebase. Rebase phá hủy nhánh và repo của các developers đó sẽ bị ảnh hưởng / không nhất quán trừ khi họ sử dụng lệnh git pull --rebase.

- Development team có kỹ năng như thế nào? Rebase là một hoạt động phá hoại.

  → Điều đó có nghĩa, nếu bạn không áp dụng nó một cách chính xác, bạn có thể mất commit và / hoặc phá vỡ sự thống nhất repo của developers khác.

- Bản thân nhánh có đại diện cho thông tin hữu ích không?

  → Một số nhóm sử dụng mô hình nhánh cho mỗi nhánh, trong đó mỗi nhánh đại diện cho một feature (hoặc fixbug, hoặc tính năng phụ, vv) Trong mô hình này nhánh giúp xác định các tập hợp các commit liên quan. Trong trường hợp mô hình nhánh cho mỗi developer, chính nhánh không truyền tải bất kỳ thông tin bổ sung nào. Sẽ không có hại gì khi rebasing.

- Bạn có muốn revert những pull đã merger vì bất kỳ lý do nào không?

  → Reverting a rebase sẽ hơi khó khăn và / hoặc không thể (nếu rebase có conflict) so với reverting a merge. Nếu bạn nghĩ rằng có thể bạn sẽ muốn revert sau đó sử dụng merge.

TESTING

▼ **Có những loại test nào ?**

- **Unit test:** chủ yếu test `function`, chỉ test riêng lẻ một module trong code của bạn. Đối với *React* thì có thể xem đây là test một component
- **Integration test:** test sự liên kết giữa các component. Đôi khi chạy một mình không sao nhưng ghép lại thì ra cả một bầu trời đầy sao ????. Khi viết *integration test* thì chúng ta sẽ sử dụng dữ liệu giả để dễ dàng kiểm soát được đầu ra cuối cùng
- **End-to-end test (E2E):** cũng giống như *integration test* nhưng chúng ta sẽ test như môi trường *production*. Loại test này giống y hệt cách các bạn hay test bằng tay, điểm khác biệt là các bạn sẽ tự động hóa quá trình này

▼ **Cách phân bổ các loại test khi viết test trong react**

- **Unit test**: đảm bảo component được render mà không gây ra lỗi. Phần này cần có nhưng *không cần tập trung quá nhiều* vào nó
- **Integration test**: phần chúng ta *nên tập trung vào nhiều nhất* vì nó sẽ gần sát với thực tế nhất
- **E2E test**: bài test phản ánh thực tế khi sử dụng sản phẩm. Tuy tính chính xác cao nhưng lại phụ thuộc vào các thành phần khác trong hệ thống như *database*, *API*, ... nên khá khó để kiểm soát đầu ra. Theo minh thì phần này chiếm tỉ lệ ít nhất trong tổng số các bài test

▼ **Các bước test trong react**

- **Arrange**: chuẩn bị input (dữ liệu đầu vào) cho bài test
- **Act**: thực hiện test (invoke function, trigger event `click`/`change`, ...)
- **Assert**: kiểm tra output (kết quả)

▼ **Một số tool testing trong react**

- React-testing-library
- Jest
- **Mocha**
- Chai
- …

DESIGN PATTERNS

6 design patterns JS cần biết

pattern.dev

▼ **Các loại design pattern ?**

> 💡 ***Design pattern là các mẫu lập trình phổ biến được xây dựng bởi các lập trình viên nhiều kinh nghiệm, nó giúp source code của chúng ta dễ đọc, dễ hiểu và dễ mở rộng về sau này. Có thể nói design pattern định nghĩa ra các tiêu chuẩn code và các mẫu lập trình mà từ đó chúng ta mới có hàng trăm nghìn thư viện lập trình như ngày hôm nay.***

- Được chia làm 3 nhóm:

  ▼ **Creational Patterns:** Nhóm này được dùng để phục vụ cho việc khởi tạo đối tượng.

  1. **Singleton**: Tạo ra đối tượng sử dụng cho toàn chương trình và chỉ khởi tạo 1 lần
  2. **Factory**: Tạo ra các đối tượng theo một quy tắc nhất định
  3. **Factory Method**: Tạo ra các đối tượng theo một quy tắc nhất định, nhưng cho phép lớp thừa kế quy định đối tượng sẽ được tạo ra
  4. **Abstract Factory**: Tạo ra các đối tượng theo một quy tắc nhất định mà không cần biết kiểu của đối tượng
  5. **Builder**: Thường dùng để tạo các đối tượng readonly hay immutable
  6. **Prototype**: Thường dùng để clone 1 đối tượng từ 1 đối tượng có sẵn
  7. **Object Pool:** Dùng để tạo ra các đối tượng có thể dùng lại nhiều lần để tránh khởi tạo không cần thiết

  ▼ **Behavioral Patterns:** Nhóm này phục vụ cho việc xử lý các hành động (action, request, event)

1. **Chain of Responsibility**: Là một chuỗi xử lý cho một hành động
2. **Command**: Xử lý hành động theo kiểu tương ứng
3. **Interpreter**: Chuyên từ đối tượng này sang đối tượng khác
4. **Iterator**: Dùng để duyệt qua một collection như bạn vẫn hay dùng vòng for với List và Set
5. **Mediator**: Định nghĩ ra một đối tượng sử dụng chung cho các lớp, ví dụ đối tượng Graphic sẽ được sử dụng bởi các lớp Button, TextView, ...
6. **Memento**: Dùng để quản lý trạng thái và trở lại trạng thái trước khi cần thiết, giống như undo và redo
7. **Observer**: Lắng nghe một sự kiện sẽ xảy đến và xử lý sự kiện đó
8. **Strategy**: Đưa ra các xử lý tương ứng với hành động xảy đến
9. **Template Method**: Quy định trình tự gọi hàm để đảm bảo lập trình viên không bị mắc sai lầm, nếu bạn dùng Android bạn sẽ thấy các hàm onCreate, onStart, ... đây là 1 trong những design pattern quan trọng nhất của Android
10. **Visitor**: "Thăm quan" các đối tượng trong một mảng hay 1 collection
11. **Null Object**: Có thể hiểu đơn giản là if (value == null) do something else do something

▼ **Structural Patterns:** Nhóm này phục vụ cho việc kết nối các đối tượng và mở rộng hệ thống

1. **Adapter**: Chuyển một interface của một class sang 1 interface của 1 class khác, cảm giác nó cũng hơi giống giống Command pattern
2. **Bridge**: Giữ lại các phần giống nhau, và tách các phần khác nhau ra thành các lớp riêng biệt
3. **Composite**: Kết hợp các lớp lại với nhau để tránh phải extends hay implements quá nhiều lớp và interface, ví dụ chúng ta hay có lớp service sử dụng rất nhiều lớp repo và các service khác
4. **Decorator**: Là lớp để "trang hoàng" thêm cho đối tượng của chúng ta trước khi đưa đối tượng này vào sử dụng hoặc lưu trữ, phản hồi

5. **Flyweight**: Dùng để chia sẻ một lượng lớn các đối tượng được khởi tạo một lần
6. **Proxy**: Dùng để wrap lại đối tượng thực tế, thông thường chúng ta sử dụng wrap lại các đối tượng của thư viện bằng đối tượng của chúng ta, để dễ dàng thay đổi thư viện sau này

▼ **Các design pattern trong react?**

- HOC pattern
- Render props pattern
- Hooks pattern
- Compound pattern

SECURITY

- Các vấn đề bảo mật cần nắm trong react
- Cải thiện bảo mật trong React

ĐỌC THÊM

- **Câu hỏi hooks thường gặp**
- **Trick interview JS**
- **Một số bài viết hay với react**