Tổng hợp câu hỏi PV FE REACT

HTML

▼ **DOCTYPE là gì ?**

- là dòng mã đầu tiên được yêu cầu trong HTML
- hướng dẫn trình duyệt về phiên bản HTML
- đảm bảo trang web được phân tích cú pháp giống nhau bởi trình duyệt khác nhau.

▼ **Thuộc tính DATA_? là gì ?**

- cho phép lưu trữ thêm thông tin dữ liệu trong DOM
- get data → getAttribute trong JS
- set data → setAttribute trong JS

▼ **Cách set nhiều ngôn ngữ trên trang ?**

- dùng thuộc tính **lang** trong thẻ html

▼ **HTML SEMANIC là gì ?**

- sử dụng thẻ thích hợp nhất cho các nhiệm vụ hiện có
- sử dụng những phần tử có ý nghĩa như form, article, table, thay vì sử dụng div hay span

▼ **WEB ACCESSIBILITY?**

- đảm bảo web có thể sử dụng cho người khuyết tất

▼ **Element ≠ attribute ?**

- **element** trong HTML là các thẻ như: div, span, a, img, …
- **attribute** dùng để mô tả đặc điểm của element như: src, class, id

▼ **display: none ≠ visibility: hidden**

- cả 2 đều dùng để ẩn element
- none sẽ ẩn đi hoàn toàn element và không chiếm không gian trong giao diện
- hidden chỉ ẩn đi nhưng vẫn chiếm không gian trên giao diện

▼ **<Script/>, <Script async/>, <Script defer/>**

Khi tải trang HTML sẽ có 2 điều chỉnh được thực hiện:

- Chuyển đổi HTML
- Tải các đoạn script

- <Script/>: thực hiện tuần tự, chặn phân tích HTML → nạp script → thực thi → phân tích lại HTML sau khi thực thi xong
- <Script async/>: thực hiện không đồng bộ → không chặn phân tích cú pháp HTML
- <Script defer/>: phân tích cú pháp HTML xong rồi mới thực thi script

▼ **DOM là gì?**

- Là 1 api cho HTML, XML
- Đại diện các HTML, XML dưới dạng các node và object , có thể sử dụng thông qua JS
- Có cấu trúc dạng tree và mỗi element trên dom là một node

▼ **Bố cục HTML**

- HTML5 giới thiệu 1 số thẻ giúp chúng ta có thể định nghĩa bố cục trang web một cách rõ ràng hơn như:
  - `<header>`: Lưu trữ thông tin bắt đầu về trang web.
  - `<footer>`: Biểu diễn phần cuối cùng của trang.
  - `<nav>`: Menu điều hướng của trang HTML.
  - `<article>`: Nó là một tập hợp thông tin.
  - `<section>`: Nó được sử dụng bên trong `article` để xác định cấu trúc cơ bản của một trang.
  - `<aside>`: Nội dung menu dọc của trang.

▼ **Cách tối ưu hiệu suất tải trang web**

- Lưu trữ CDN → giảm độ trễ
- Nén tập tin → giảm kích thước nội dung → tăng tốc độ truyền dữ liệu
- Nối tập tin → giảm số lượng request được gọi
- Giảm thiểu (minify) tập lệnh: giảm kích thước tệp JS, CSS
- Lazy loading: tải nội dung cần thiết

▼ **Web worker là gì ?**

- JS đơn luồng, do đó những tác vụ lớn đòi hỏi tính toán phức tạp hoặc thời gian phản hồi lâu thì khả năng làm cho giao diện đơ cứng → Web worker ra đời và giải quyết vấn đề nói trên
- Web worker không phải của JS mà là tính năng của browser
- Ta chỉ cần khởi tạo và truyền tham số đầu vào là file JS trong đó chứa các code cần xử lý
- WW giúp thực thi song song với nhưng chạy dưới dạng background
- WW là đa luồng
- Gồm 3 loại:
  - Service worker
  - Shared worker
  - Dedicated worker

▼ **Thẻ <a/> và <link/> khác gì nhau ?**

- Thẻ <a> được dùng để mở liên kết đến trang web khác hay một phần nào đó ở web hiện tại nên nó có thể click vào.
- Thẻ <link> xác định một liên kết đến một nguồn tài nguyên bên ngoài, nó không thể click.

▼ **Khi nào sử dụng script ở header hoặc body**

- Nếu các script chứa các hàm event-trigger hoặc thư viện jquery thì nên đặt ở head.
- Nếu script viết nội dung hoặc không nằm trong hàm thì nên đặt ở cuối body.

- Nói tóm lại có 3 điểm cần nhớ sau:
    - Đặt thư viện hoặc sự kiện script ở head.
    - Đặt script thông thường ở head cho đến khi có vấn đề gì đó về hiệu suất.
    - Đặt script hiển thị nội dung ở cuối body.

▼ **image ≠ figure**

- Thẻ `<figure>` chỉ định nội dung như ảnh, sơ đồ, code snippets,... được dùng để tổ chức các nội dung như ảnh, tiêu đề ảnh
- Thẻ `<img>` dùng để nhúng một ảnh vào HTML5.

▼ **Manifest file là gì ?**

- File manifest được sử dụng để liệt kê các tài nguyên có thể được lưu vào bộ nhớ đệm.
- Trình duyệt sử dụng thông tin này để làm cho trang web tải nhanh hơn lần đầu tiên.
- Có 3 phần trong manifest:
    - CACHE Manifest - File cần lưu vào bộ đệm
    - Network - File không bao giờ lưu vào bộ đệm, cần kết nối mạng.
    - Fallback - File dự phòng trong trường hợp trang không tiếp cận được.

▼ s**rcset trong thẻ img là gì ?**

- srcset cho phép bạn khai báo một tập hợp các hình ảnh sẽ được hiển thị trên các kích thước khung nhìn khác nhau.
- Bạn chỉ cần lưu và hình ảnh ở các độ phân giải khác nhau
- VD: img_small.png 200w, img_medium.png 500w, img_large.png 1000w
- và chúng được ngăn cách bởi dấy phẩy

▼ **lazy loading là gì ? Các cách triển khai ?**

- là kỹ thuật ngăn trình duyệt tải tất cả các resource cùng 1 lúc, thay vào đó chỉ tải những resource cần thiết
- Giúp tăng tốc độ tải trang, trải nghiệm người dùng cũng như tiết kiệm băng thông

- Một số cách triển khai:
  - Bắt sự kiện scroll → tạo data_src → scroll tới target thì copy data_src sang src
  - Dùng Intersection Observer API của browser
  - Dùng thuộc tính loading trong thẻ image
  - Sử dụng một số thư viện có sẵn

CSS

▼ **inline ≠ block ≠ inline-block trong display**

- **inline**: item nằm trên 1 dòng, vượt quá độ dài sẽ tự động xuống dòng. Chỉ điều chỉnh được margin và padding của left, right (span)
- **block**: item luôn được xuống dòng và chiếm toàn bộ width nếu không được set. Ta điều chỉnh được margin, padding của cả 4 hướng (div)
- **inline-block**: các item được sắp xếp trên cùng 1 hàng như inline nhưng sẽ có thuộc tính của block như có thể điều chỉnh 4 hướng của margin và padđing

▼ **BEM là gì ?**

- BEM viết tắt của Blocks, Elements, Modifiers, là một phương pháp đặt tên class cho HTML và CSS.
- Được phát triển tại Yandex giúp lập trình viên hiểu rõ hơn mối quan hệ giữa HTML và CSS trong dự án front end

```
/* Một Block (khối) độc lập */
.btn {}

/* Element (phần tử) con, phụ thuộc vào Block ở trên */
.btn__price {}

/* Modifier (bộ điều chỉnh) thay đổi trạng thái của Block hoặc Element */
.btn--orange {}
.btn--big {}
.btn__price--bold {}
```

▼ **Box-model là gì ?**

- là một hộp chữ nhật bao quanh mọi element trong HTML
- được dùng để xác định chiều cao và rộng cảu
- CSS Box Model giống như là một cái hộp bao quanh element của chúng ta và trong đó có rất nhiều lớp dày mỏng khác nhau, các lớp dày mỏng đó bao gồm: margins, border, padding và cuối cùng là phần nội dung của chúng ta (text và ảnh).
- Nó bao gồm:
  - **Content:** như đã nói ở trên đây là phần mà text và hình ảnh của chúng ta xuất hiện
  - **Padding:** là một khoảng trống kế tiếp bọc xung quanh **content**
  - **Border**: phần khung bao bọc xung quanh **padding** và **content**
  - **Margin**: cuối cùng, **margin** là phần ngoài cùng của Box Model, chỉ là một khoảng trống không màu

▼ **Sự khác biệt giữa các thuộc tính Box Sizing?**

- Thuộc tính CSS box-sizing quy định cách tính tổng chiều rộng và chiều cao của một phần tử.
  - **Context-box:** Giá trị chiều rộng và chiều cao mặc định chỉ áp dụng cho nội dung của phần tử. Padding và border nằm ở bên ngoài hộp.
  - **Padding-box:** Giá trị chiều rộng và chiều cao mặc định chỉ áp dụng cho nội dung của phần tử và padding của nó. Border nằm ở bên ngoài hộp. Hiện tại chỉ có Firefox hỗ trợ padding-box.
  - **Border-box:** Giá trị chiều rộng và chiều cao áp dụng cho nội dung, padding và border.

▼ `* { box-sizing: border-box }` **là gì?**

- Nó điều chỉnh tất cả phần tử có bao gồm padding, border trong không gian phần tử cho tính toàn chiều dài và chiều rộng.
- Trong `box-sizing: border-box`, chiều cao phần tử được tính toán với: height + padding dọc + độ dài border dọc. Còn chiều dài là width + padding ngang + độ dài border ngang.

▼ **CSS selector là gì ?**

- CSS Selector giống như là đường dẫn, chỉ định để cho CSS biết bạn đang muốn điều chỉnh, tạo kiểu cho phần tử HTML nào vậy.

▼ **Một số loại selector:**

- **Universal Selector:** hoạt động như một ký tự đại diện cho tất cả phần tử trong trang.

```
* {
    color: "green";
}
```

- **Element Type Selector:** selector loại này ứng với một hoặc nhiều phần tử HTML cùng tên.

```
ul {
    line-style: none;
}
```

- **ID Selector:** selector này ứng với bất kỳ phần tử HTML nào có thuộc tính ID có cùng giá trị với giá trị của selector.

```
#container {
    width: 960px;
}
```

- **Class Selector:** tương tự như ID Selector nhưng thay vì ứng với ID thì nó ứng với thuộc tính class.

```
.box {
    padding: 10px;
}
```

- **Descendant Combinator:** giúp bạn kết hợp hai hoặc nhiều selector để có thể chỉ định phần tử cụ thể.

```
#container .box {
  float: left;
}
```

- **Child Combinator:** selector sử dụng bộ child combinator tương tự như descendant combinator, ngoại trừ việc nó chỉ nhắm đến các phần tử con.

```
#container> .box {
  float: left;
  padding-bottom: 15px;
}
```

- **General Sibling Combinator:** selector này so với các phần tử có quan hệ anh chị em với phần tử tương ứng.

```
h2 ~ p {
  margin-bottom: 20px;
}
```

- **Adjacent Sibling Combinator:** selector sử dụng ký tự `+` và gần giống với General Sibling Combinator. Sự khác biệt là phần tử được nhắm phải là anh chị ruột thịt chứ không phải anh chị em chung chung.

```
p + p {
  margin-bottom: 0;
}
```

- **Attribute Selector:** nhắm đến các phần tử dựa trên sự xuất hiện và giá trị của thuộc tính HTML. Được khai báo bằng dấu ngoặc vuông.

```
input [type=”text”] {
  width: 200px;
}
```

▼ **Có thể hiện thị 1 trang web trong 1 trang web khác không ?**

- Có

- Sử dụng iframe để nhúng 1 web khác vào
- Có thể nhúng 1 website, video hoặc image, …

▼ **z-index dùng để làm gì ?**

- z-index được sử dụng để chỉ định cách xếp chồng theo chiều sâu của các phần tử chồng lên nhau xảy ra tại thời điểm định vị nó.
- Nó chỉ định thứ tự ngăn xếp theo chiều sâu của các phần tử được định vị giúp xác định cách hiển thị các phần tử diễn ra như thế nào trong trường hợp chồng chéo.

▼ **Sự khác biệt giữa reset và normalize CSS?**

- **Reset CSS:** nhằm mục đích xoá tất cả thiết lập style mặc định từ trình duyệt. Ví dụ như margin, padding, font-size của tất cả phần tử đó được reset lại giống nhau.
- **Normalize CSS:** nhằm mục đích làm cho các style mặc định nhất quán trên trình duyệt. Nó cũng sửa các lỗi phổ biến trên trình duyệt.

▼ **Float là gì ?**

- dùng để định vị phần tử theo chiều ngang về bên trái hoặc phải

▼ **Phần tử Pseudo và các lớp Pseudo là gì?**

- **Phần tử pseudo** cho phép ta tạo các mục thường không tồn tại trong DOM.
  - ::before
  - ::after
  - ::first-letter
  - ::first-line
  - ::selection
- **Lớp pseudo** chọn các phần tử thông thường nhưng trong các điều kiện nhất định như khi người dùng di chuột qua liên kết.
  - :link
  - :visited
  - :hover

    - :active
    - :focus

▼ **CSS sprite là gì ?**

- CSS Sprite dùng cho kết hợp nhiều hình ảnh thành một hình ảnh lớn. Nó thường dùng cho biểu diễn icons. Các ưu điểm của nó là:
    - Giảm số lượng yêu cầu HTTP để lấy nhiều ảnh vì nó cho phép chỉ gửi một yêu cầu.
    - Nó giúp tải trước các nội dung giúp hiển thị các icon hoặc hình ảnh khi di chuột và các pseudo-state khác.
    - Khi có nhiều hình ảnh, trình duyệt sẽ thực hiện các lệnh gọi riêng biệt để lấy hình ảnh cho từng hình ảnh đó.

▼ **Tích hợp css vào HTML có bao nhiêu cách ?**

- inline: css trực tiếp trong element bằng thuộc tính style
- external: tạo 1 file css riêng và nhúng vào
- internal: style thông qua thẻ <style> đặc ở <head>

▼ **Lợi thế của dùng translate() thay vì position absolute?**

- Translate() không làm cho trình duyệt kích hoạt repaint layout, mà chỉ thực hiện soạn thảo.
- Còn position: absolute làm trình duyệt phải vẽ lại các luồng DOM.

→ Thế nên `translate()` đem về hiệu suất tốt hơn

▼ **Liệu margin-top hoặc margin-bottom có ảnh hưởng đến các phần tử inline không?**

- Không, nó không ảnh hưởng đến các phần tử inline. Các phần tử inline ở cùng dòng với nội dung của trang.

▼ **Khi nào xảy ra DOM-reflow ?**

- Dom-reflow là quá trình browser tính toán lại vị trí và hình dạng của element trong document, nhằm mục đích hiển thị lại 1 phần hoặc toàn bộ DOM
- Reflow xảy ra khi:

- Chèn, xóa, update element trong DOM
- Sửa đổi nội dung trang
- Thay đổi style css

▼ **Làm thế nào để căn giữa 1 thẻ p trong thẻ div ?**

```
<div>
  <p>Hello</p>
</div>
```

```
// can giua theo chieu ngang
div {
  text-align: center;
}

// can giua theo chieu doc
div {
  position: relative;

  p {
    position: absolute;
    top: 0;
    left: 0;
    right: 0;
    bottom: 0;
    margin: auto;
  }
}
```

▼ **Làm sao để căn giữa một div trong một div khác?**

▼ transform

```
.cn {
  position: relative;
  width: 500px;
  height: 500px;
}

.inner {
  position: absolute;
  top: 50%; left: 50%;
  transform: translate(-50%,-50%);
  width: 200px;
```

```
  height: 200px;
}
```

▼ flex-box

```
.cn {
  display: flex;
  justify-content: center;
  align-items: center;
}
```

▼ grid

```
<div class="wrap_grid">
  <div id="container">vertical aligned text<br />some more text here
  </div>
</div>
```

```
.wrap-grid {
  display: grid;
  place-content: center;
}
```

▼ **Các các ẩn đi 1 phần tử trong css**

- display: none → phần tử sẽ không xuất hiện trong DOM
- visibility: hidden → phần tử có xuất hiện trong DOM nhưng không hiển thị lên màn hình
- position: absolute → set top, right, bottom hoặc left về số âm để di chuyển element ra ngoài màn hình
- transform: translateX(-999px) | translateY(-999px) | scale(0)
- opacity: 0 → ẩn đi, nó chỉ vô hình nhưng ta vẫn có thể add event lên nó

▼ **grid và flex khác nhau gì ?**

- grid là bố cục 2 chiều cả ngang và dọc → tiếp cận theo hướng layout
- flex 1 chiều, theo chiều ngang hoặc dọc → tiếp cận theo hướng nội dung

- Nếu biết rõ nội dung ta cần trình bày thì nên dùng flex ngược lại dùng grid
- Flex phù hợp web có bố cục đơn giản, grid thì phù hợp UI phức tạp

▼ **Grid là gì?**

- hệ thống layout 2 chiều theo trục x và y
- tổ hợp của đường ngang và dọc cắt nhau bao gồm các hàng và cột
- Các phần tử sẽ được đặt trên các hàng và cột này

▼ **Flexbox là gì?**

- là hệ thống bố cục một chiều (ngang hoặc dọc)
- giúp căn chỉnh bố trí những item trong container một cách linh hoạt ngay cả khi kích thước chưa xác định hoặc kích thước động
- Flex bao gồm:
  - flex-container (parent)
    - flex-direction
    - flex-wrap
    - flex-content
    - align-items
    - align-content
  - flex-item (child)
    - order
    - align-self
    - flex-grow
    - flex-shink
    - flex-basis

▼ **Giải thích Position trong css?**

> 💡 Normal flow là cách trình duyệt hiển thị những block element từ trên xuống dưới và mỗi block sẽ chiếm toàn bộ chiều ngang của container (div, p), ngược lại thì là inline(a, span, img)

- **static**: vị trí mặc định, theo dòng chảy thông thường của trang, tuy nhiên ta không thể set **L, T, R, B, z-index** cho nó
- **relative**: tuân theo quy luật của dòng chảy thông thường của trang nhưng có thể set các giá trị **L, T, R, B, z-index**
- **absolute**: element sẽ bị loại bỏ khỏi normal flow và nó sẽ nằm tương đối so với thuộc tính cha gần nhất của nó mà element cha đó phải có thuộc tính position là **relative**, **absolute**, **fixed** hoặc **sticky**.
  - Nếu không có thằng element cha nào mà có các thuộc tính trên thì nó sẽ nằm tương đối với root-element là thẻ html
- **fixed**: tương tự absolute nhưng khác là nó chỉ hiện thị tương đối so với thẻ html (root-element)
- **sticky**: là sự kết hợp giữa **relative** và **fixed**. Tức là nó vẫn nằm trong normal flow của trang như **relative** nhưng nó sẽ trở thành **fixed** nếu ta cuộn xuống đúng vị trí của nó. Và nó chỉ hoạt động trên container chứa nó.

▼ **overflow là gì ?**

- dùng để xử lý khi kích thước nội dung vượt quá kích thước container

▼ **Độ đặc hiệu, độ cụ thể trong css là gì ?**

- Tính đặc hiệu hay độ ưu tiên (specificity) là cách mà trình duyệt quyết định sẽ áp dụng thuộc tính css nào với một phần tử khi có nhiều quy tắc css cùng trỏ đến phần tử đó.
- Inline style sẽ được ưu tiên so với ID rồi đến giá trị lớp (pseudo-class hoặc attribute selector), universal selector **(*)** sẽ không có độ ưu tiên. ID Selector có độ ưu tiên cao hơn attribute selector.

▼ **Đơn vị trong css ?**

- Absolute units: px, pt, cm, mm, …
- Relative units: rem, em, %, vw, vh, vmin, vmax, …

- em: giá trị phụ thuộc vào phần tử cha gần nhất hoặc chính nó, được xác định thông qua thuộc tính font-size
- rem: tương tự như em nhưng phụ thuộc vào root
- vw: tính theo tỉ lệ **chiều rộng khung nhìn** thiết bị. 1 vw = 1/100 width view-port.
  - Ví dụ: màn hình của bạn có chiều rộng 1100px thì 1vw = 11px
- vh: tương tự vw nhưng theo height view-port

▼ **Cách css hoạt động?**

- Ngôn ngữ CSS được thiết kế để sử dụng cùng với ngôn ngữ "đánh dấu" như HTML.
- CSS xác định cách các phần tử HTML được định dạng - kiểm soát bố cục, màu sắc, phông chữ của chúng, ...
- Khi trình duyệt hiển thị một document, nó phải kết hợp nội dung của document với thông tin style của nó.
- Nó xử lý document theo một số giai đoạn, mà chúng ta đã liệt kê bên dưới.

1. Trình duyệt tải HTML (ví dụ: nhận nó từ mạng).
2. Nó chuyển đổi HTML thành DOM.
3. Sau đó, trình duyệt sẽ tìm nạp hầu hết các tài nguyên được liên kết với tài liệu HTML, chẳng hạn như hình ảnh và video được nhúng và CSS được liên kết.
4. Trình duyệt phân tích cú pháp CSS đã nạp và sắp xếp các quy tắc khác nhau theo kiểu selector của chúng thành các "nhóm" khác nhau, ví dụ: phần tử, lớp, ID, ... Dựa trên các selector mà nó tìm thấy, nó sẽ tìm ra các quy tắc nên được áp dụng cho các nút nào trong DOM và đính kèm kiểu cho chúng theo yêu cầu (bước trung gian này được gọi là cây render).
5. Cây render được bố trí trong cấu trúc mà nó sẽ xuất hiện sau khi các quy tắc đã được áp dụng cho nó.
6. Hiển thị trực quan của trang được hiển thị trên màn hình.

JS

▼ **Kỹ thuật debounce, throttling và cách code thực tế**

▼ **Từ khóa “this” trong JS**

- Từ khoá "this" trong javascript tham chiếu đến một đối tượng có thuộc tính là một hàm.
- Giá trị của "this" phụ thuộc vào đối tượng đang gọi hàm.

▼ **Giải thích cách hoạt động của `this` trong JavaScript**

- Không có lời giải thích đơn giản nào cho `this`; nó là một trong những khái niệm khó hiểu nhất trong JavaScript. Giá trị của `this` phụ thuộc vào cách hàm được gọi. Các quy tắc sau được áp dụng:
  - Nếu từ khóa `new` được sử dụng khi gọi hàm, thì bên trong hàm này là một đối tượng hoàn toàn mới.
  - Nếu `apply, call` hoặc `bind` được sử dụng để gọi / tạo một hàm, thì bên trong hàm này là đối tượng được truyền vào dưới dạng đối số.
  - Nếu một hàm được gọi là một phương thức, chẳng hạn như `obj.method ()` – thì `this` là đối tượng mà hàm là thuộc tính của nó.
  - Nếu một hàm được gọi dưới dạng một lệnh gọi hàm miễn phí, nghĩa là nó được gọi mà không có bất kỳ điều kiện nào ở trên, thì đây là đối tượng toàn cục. Trong trình duyệt, nó là đối tượng `window`. Nếu ở chế độ nghiêm ngặt (`'use strict'`), `this` sẽ là `undefined` thay vì đối tượng toàn cục.
  - Nếu áp dụng nhiều quy tắc trên, quy tắc nào cao hơn sẽ được ưu tiên và sẽ đặt giá trị này.
  - Nếu hàm là một arrow function ES2015, nó sẽ bỏ qua tất cả các quy tắc ở trên và nhận `this` của phạm vi xung quanh tại thời điểm nó được tạo.

▼ **Phân biệt call, apply, bind**

- cả 3 đều dùng để thay đổi context của từ khóa this
- thiết đặt và thay đổi giá trị “**this**” theo ý muốn
- cả 3 hàm đều là các prototype của function. Nên chỉ có Function mới gọi được 3 hàm này.

- Về cơ bản là giống nhau về cách sử dụng nhưng khác nhau về cách gọi và cách truyền tham số
    - **call**: gọi trực tiếp hàm và cho phép truyền vào 1 object và ds những tham số ngăn cách bởi dấu phẩy
    - **apply**: giống như call nhưng nó truyền tham số là 1 array
    - **bind**: không gọi trực tiếp mà nó sẽ trả về 1 hàm mới và truyền danh sách tham số ngăn cách bởi dấy phẩy

▼ **Giải thích cơ chế hoisting trong JS**

- Hoisting là một hành vi mặc định của Javascript di chuyển việc khai báo lên đầu trong scope của nó.

▼ **Giải thích thêm về cách bộ máy JS hoạt động:**

- Khi bộ máy JS xử lý đoạn code của bạn , nó tạo ra 1 cái gọi là bối cảnh thực thi (execution context). Có 2 quá trình liên quan đến việc tạo cái bối cảnh thực thi này:
    - **GD1** (creation) : Trong giai đoạn này các biến và function được thêm vào bộ nhớ. Bộ máy JS sẽ đi qua một lượt đoạn code của bạn và thêm tất cả các biến vào bộ nhớ máy tính. **Nhưng nó chưa gán giá trị cho các biến này.** Trong khi đó các function thì lại được **thêm toàn bộ vào bộ nhớ bao gồm cả tên và khối code bên trong nó.**
    - **GD2** (execution) : Trong giai đoạn này giá trị sẽ được gán cho các biến và function sẽ được gọi. Nên kể cả bạn khởi tạo 1 biến với giá trị ban đầu thì ở giai đoạn 2 này biến mới được gán giá trị. Ở giai đoạn 1 giá trị không được gán cho biến , nó được thêm vào bộ nhớ với giá trị khởi tạo là undefined.
- Trước khi function được thực thi , nó đã được thêm vào bộ nhớ trong giai đoạn 1 (creation) nên bộ máy Javascript biết function này nằm ở đâu. **Nó không chuyển cái function này lên trên đầu.**
- Đối với biến quá trình cũng giống như vậy nhưng có 1 chút khác biệt. Các biến cũng được thêm vào bộ nhớ trong giai đoạn 1 nhưng không có giá trị nào được gán cho chúng. Trong JS khi một biến được khai báo mà không có giá trị nào bộ máy JS tự động thêm giá trị **undefined**

▼ **Temporal dead zone là gì ?**

- Từ es6 trở lên giới thiệu 2 từ khóa **let** và **const** để khai báo biến.
- Điểm khác biệt so với var là ***ở giai đoạn 1 (creation) chúng không được khởi tạo với giá trị undefined như với var.***
- Thay vào đó: chúng được set 1 chế độ đặc biệt gọi là **Temporal Dead Zone**. Đây là 1 khoảng thời gian giữa việc khai báo và khởi tạo biến.
- Trong giai đoạn này bạn sẽ không thể truy cập vào biến đó được.
- Điều này có nghĩa là chúng vẫn tồn tại ở đó nhưng bạn sẽ không thể truy cập được cho đến khi bạn khởi tạo giá trị cho biến sẽ được thực hiện ở giai đoạn 2.

▼ **Ép kiểu ngầm trong JS là gì**

- là sự chuyển đổi tự động của giá trị từ kiểu dữ liệu này sang kiểu dữ liệu khác
- Ví dụ:
  - số + chuỗi = chuỗi
  - số - chuỗi = số
  - boolean sử dụng toán tử logic
    - Falsy: null, undefined, 0, 0n, -0, NaN, “”
    - Truethy: là những giá trị khác falsy

▼ **JS là static hay dynamic**

- là dynamic, kiểu dữ liệu của biến được kiểm tra tỏng khi đang chạy chương trình
- Đối với static thì nó sẽ kiểm tra sau khi biên dịch xong

▼ **Higher order function là gì (HOC)**

- là 1 hàm nhận vào tham số là hàm khác hoặc return về 1 hàm, hoặc cả 2 vừa nhận vừa return về 1 hàm

▼ **Scope trong JS**

- là khái niệm nhắm xác định phạm vi hoạt động của biến
- Các loại scope
  - **Global scope:**

        - biến được định nghĩa ở bên ngoài function
        - sẽ được sử dụng và thay thế ở bất cứ đâu
        - nếu ta định nghĩa 1 biến mà không dùng từ khóa nào thì mặc định nó thuộc global scope
    - **Local scope:**
        - biến được định nghĩa bên trong 1 function
        - mỗi function khi gọi sẽ tạo ra 1 scope mới
        - mỗi function sẽ tạo ra 1 local scope khác nhau, cho nên ta có thể khai báo các biến trùng tên ở các scope local
    - **Lexical scope:**
        - Ví dụ ta có fn A nằm trong fn B, mà trong A có các biến tham chiếu đến fn B thì ta nói fn A có lexical scope và lexical scope của A chính là scope của nó + với scope của fn B
    - Ngoài ra ta còn có block statements (if/else, switch case, for loop)
        - Không giống như function nó không tạo ra local scope mới mà nó giữ nguyên scope của nó
        - Các biến khai báo bằng var nó sẽ không tạo ra local scope trong block statements
        - let và const giúp tạo local scope bên trong 1 block
        - Biên được khai báo trong block {} thì chỉ truy cập được trong block đó thôi, bên ngoài không truy cập được

▼ **scope chain trong JS ?**

- JS sử dụng scope để tìm kiếm biến
- Nếu không tìm thấy biến bên trong function scope thì nó sẽ tiếp tục tìm biến đó ra phạm vi bên ngoài
- Nếu không có sẽ tiếp tục tìm ở global scope

→ **Nếu không có nữa sẽ có lỗi tham chiếu được đưa ra**

▼ **Closures trong JS là gì ?**

- là 1 hàm mà có khả năng ghi nhớ nơi nó được tạo ra và truy cập được các biến ở bên ngoài phạm vi của nó

→ Vì lexical scope là khái niệm đi với function nên có thể coi closure là tập hợp gồm function và lexical scope của function đó

- Giúp ta viết code ngắn gọn hơn, đồng thời ứng dụng được tính **private** trong OOP

▼ **Currying Function là gì ?**

- Biến đổi 1 hàm nhiều tham sô thành n hàm nhận vào 1 tham số
- Về bản chất ta không thay đổi chức năng của hàm mà chỉ thay đổi cách gọi hàm

▼ **var, let, const khác nhau gì ?**

- về mặt scope
  - let, const giúp tạo local scope bên trong block statements
  - var rất khó xác định scope cụ thể nên bây giờ cũng tránh sử dụng var, ngoài ra các biến khai báo bằng var đều bị hoisting
- Về mặt giá trị
  - const là bất biến ta không thể gán lại giá trị cho biến được khai báo với const
  - var, let thì có thể gán lại giá trị cho biến

▼ **null ≠ undefined như thế nào ?**

- Khi ta khai báo nhưng chưa gán giá trị cho biến đó thì nó là **undefined**
- Khi ta lấy giá trị của một biến mà chưa khai báo thì nó sẽ gây ra lỗi **is not define**
- **Null** có nghĩa là giá trị rỗng hoặc giá trị không tồn tại, nó có thể được sử dụng để gán cho một biến như là một đại diện không có giá trị.
- **Null** typeof là **object** còn **undefined** typeof là **undefined**
- null == undefined = true
- null === undefined = false

▼ **Prototype trong JS là gì ?**

- Prototype là khái niệm cốt lõi trong JavaScript và là cơ chế quan trọng trong việc thực thi mô hình OOP trong JavaScript.

- Tất cả các object trong javascript đều có một prototype, và các object này kế thừa các thuộc tính (properties) cũng như phương thức (methods) từ prototype của mình.
- Trong JavaScript, trừ *undefined*, toàn bộ các kiểu còn lại đều là object.
- Các kiểu string, số, boolean lần lượt là object dạng *String*, *Number, Boolean*. Mảng là object dạng *Array*, hàm là object dạng *Function*
- Trước kia (ES5) trong JavaScript **không có khái niệm class**, do vậy, để **kế thừa các trường/hàm** của một object, ta phải sử dụng prototype.
- Khi ta gọi property hoặc function của một object, JavaScript sẽ tìm trong chính object đó, nếu không có thì tìm lên cha của nó. Do đó, ta có thể gọi các hàm *toUpperCase*, *trim* trong String là do các hàm đó đã tồn tại trong *String.prototype*
- Khi ta thêm function cho prototype, toàn bộ những thằng con của nó cũng học được function tương tự.
- **Giới hạn của prototype trong JavaScript:**
  - Không được phép kế thừa prototype vòng tròn.
  - Giá trị của `__proto__` có thể là `null` hoặc là một object, nhưng các kiểu dữ liệu khác đều bị bỏ qua.
  - Prototype không hỗ trợ thay đổi giá trị thuộc tính

▼ **Tóm lại:**

- Trong JavaScript, tất cả các object đều có thuộc tính ẩn `[[Prototype]]` với giá trị là `null` hoặc kiểu object.
- Bạn có thể sử dụng `obj.__proto__` như là một **getter/setter** để truy cập vào `[[Prototype]]`.
- Object ứng với `[[Prototype]]` được gọi là một prototype.
- Khi truy cập một thuộc tính hay phương thức trong object mà nó không tồn tại thì JavaScript sẽ tự động tìm kiếm trong prototype.
- Prototype chỉ hỗ trợ việc đọc, không hỗ trợ ghi/xóa thuộc tính trực tiếp trên prototype.

- Khi bạn gọi `obj.method()` và `method()` được lấy từ prototype, giá trị của `this` vẫn tham chiếu đến `obj` chứ không phải prototype.
- Vòng lặp `for...in` duyệt tất cả các thuộc tính trong object và thuộc tính của prototype thông qua kế thừa.

▼ **Class trong JS?**

- sử dung oop mà không cần dùng prototype
- không bị hoistring như function, cần khai báo trước khi sử dụng
- tuân thủ use strict
- sử dụng extends để kế thừa

▼ **Constructor trong JS**

- Dùng để tạo ra các thuộc tính mặc định của đối tượng trong JS
- khi chúng ta muốn tạo ra 1 đối tượng với những thuộc tính và phường thức giống nhau nhưng khác giá trị

▼ **Arrow Function là gì**

- Arrow function được giới thiệu từ phiên bản ES6 của javascript.
- Nó cung cấp một cú pháp mới và ngắn hơn cho khai báo hàm.
- Hàm arrow có thể xử dụng như là một biểu thức hàm. Ta sẽ so sánh khai báo hàm thông thường với hàm arrow.
- Sự khác biệt lớn nhất giữa nhất giữa hàm truyền thống với arrow, là ở từ khoá **this**.
  - Như định nghĩa, từ khoá this tham chiếu đến đối tượng chứa hàm được gọi.
  - Còn ở hàm arrow, không có ràng buộc nào của từ khóa this.
  - Từ khoá this trong hàm arrow, không tham chiếu đến đối tượng đang gọi nó. Nó kế thừa giá trị của nó từ phạm vi cha là `window object` trong trường hợp này.

▼ **Destructuring, spread operator và rest parameter ?**

- **Destructuring:** Là một cú pháp cho phép bạn gán các thuộc tính của một Object hoặc một Array. Điều này có thể làm giảm đáng kể các dòng mã cần thiết để thao tác dữ liệu trong các cấu trúc này

- **Spread operator:** Là ba dấu chấm **(...)**, có thể chuyển đổi một mảng thành một chuỗi các tham số được phân tách bằng dấu phẩy. Spread có thể tao ra một cấu trúc dữ liệu shallow copy để tăng tính thao tác dữ liệu.
- **Rest parameter:** Là một cú pháp tạo ra một array từ một số lượng giá trị không xác định Giúp chúng ta có thể định nghĩa một hàm với số lượng tham số có thể thay đổi tuỳ ý. Hay nói theo cách khác khi chúng ta không biết chắc chắn số lượng tham số cần có của một hàm chúng ta có thể sử dụng rest parameter

▼ **Generator trong JS**

- es6
- là function có khả năng hoãn lại quá trình thực thi mà vẫn giữ nguyên context
- Nó có khả năng tạm dừng trước khi hàm kết thúc và có khả năng thực hiện lại tại 1 thời điểm khác, khác với những function bình thường là thực thi hết tất cả câu lệnh trong function

▼ **ES6 có gì nổi bật**

- let, const
- arrow function
- generator function
- Destructuring, spread operator và rest parameter
- class
- async/await
- module
- map, set
- template literal

▼ **const ≠ Object.freeze() gì ?**

- const là 1 key dùng để ràng buộc các biến không thể thay đổi giá trị
- object.freeze() hoạt động với giá trị đối tượng. Làm đối tượng trở nên bất biến. Chỉ read không được write các thuộc tính trong đối tượng

▼ **Proxy trong JS là gì**

- **Proxy** là một class được giới thiệu từ ES6, cho phép bạn can thiệp và thay đổi hành vi của một đối tượng (object).
- Các hành vi này bao gồm: truy xuất/thiết lập (getter/setter) thuộc tính của một đối tượng, thay đổi prototype, gọi hàm, khởi tạo đối tượng bằng từ khóa `new`
- Chúng ta có thể áp dụng Proxy cho bất cứ object nào trong JavaScript, kể cả mảng, hàm hay một proxy khác.

▼ **Memoization là gì ?**

- là một kỹ thuật tối ưu, nhằm tăng tốc chương trình bằng cách lưu trữ kết quả của các câu gọi function và trả về các kết quả này khi function được gọi với cùng input đã gọi.

▼ **Có thể dùng những vòng lặp nào trong JS (for-loop) ?**

- **map**: để thực hiện một chức năng trên mỗi phàn tử của một mảng. Sử dụng **map** nếu chúng ta muốn thực hiện cùng một thao tác hoặc chuyển đổi trên từng phần tử của mảng và lấy lại một array mới có cùng **length** với các **value** được chuyển đổi.
- **filter**: khi chúng ta muốn xoá các mục không thoã mãn điều kiện. Mỗi phần tử của mãng được truyền cho hàm callback. Trên mỗi lần lặp nếu callback trả về true, thì phần tử đó sẽ được thêm vào mảng mới và ngược lại
- **reduce**: sử dụng để lặp qua các phần tử của mảng sau đó tính toán và trả về 1 kết quả cuối cùng. Thường được sử dụng để giải quyết các bài toán như lặp qua một mảng → xử lý gì đó → trả về một kết quả cuối cùng
- **for...in**: dùng để lặp lại trong một object. Số lượng vòng lặp sẽ tương ứng với số lượng thuộc tính trong object. Array cũng là object nên ta có thể sử dụng cho cả array, nhưng key sẽ là giá trị index của phần tử
- **for...of**: dùng để lặp các đối tượng từ Array, String, Map, WeakMap, Set, ... Đặc biệt có thể dùng bất đồng bộ trong vòng lặp
- for, do while, forEach, while

▼ **CORS**

- (Cross-Origin Resource Sharing) là một kĩ thuật được sinh ra để làm cho việc tương tác giữa client và server được dễ dàng hơn, nó cho phép JavaScript ở

một trang web có thể tạo request lên một REST API được host ở một domain khác.

- Hiểu sâu hơn đó chính là chia sẻ tài nguyên có nhiều nguồn gốc khác nhau. Chính sách nguồn gốc giống nhau của trình duyệt là một cơ chế bảo mật quan trọng. Khách hàng từ các nguồn khác nhau không thể truy cập tài nguyên của nhau nếu không được phép. Định nghĩa của tương đồng là protocol, domain và port của liên kết truy cập là giống nhau.

▼ **Xử lý bất đồng bộ là gì ?**

Giả sử bạn có một nhiệm vụ bao gồm 2 công việc tốn thời gian, tạm gọi là A và B.

- **Xử lý đồng bộ:**
  - bạn sẽ thực hiện công việc A
  - đợi A hoàn thành xong thì sẽ thực hiện B
  - rồi lại đợi B hoàn thành thì nhiệm vụ cuối cùng mới coi như xong.
  - Nghĩa là thời gian để hoàn thành nhiệm vụ là tổng của thời gian hoàn thành A và B. Hơn nữa, trong khoảng thời gian này bạn sẽ không thể thực hiện thêm 1 hành động nào khác
  - Điều này rõ ràng làm giảm hiệu năng và trải nghiệm người dùng đối với chương trình.
- **Xử lý đa luồng:**
  - Để khắc phục tình trạng này, các ngôn ngữ lập trình như C/C++, Java,... sẽ sử dụng **cơ chế đa luồng (multi-thread)**.
  - Nghĩa là mỗi công việc tốn thời gian sẽ được thực hiện trên một thread riêng biệt mà không can thiệp vào thread chính.
  - Bạn vẫn có thể thực hiện các công việc tốn thời gian mà vẫn có thể bắt các sự kiện ở thread chính.
  - Với ví dụ trên, thời gian để hoàn thành nhiệm vụ sẽ chỉ bằng thời gian hoàn thành của A hoặc B. Cái nào thực hiện xong trước sẽ đợi cái còn lại hoàn thành thì nhiệm vụ sẽ kết thúc.
- **Xử lý bất đồng bộ:**

- Tuy nhiên, JavaScript lại là một câu chuyện khác. Hai nền tảng quan trọng với JavaScript đều là **single-thread.**
- Chính vì vậy, bạn không thể xử lý đa luồng với JavaScript được mà phải sử dụng cơ chế **xử lý bất đồng bộ**
- Với cách xử lý bất đồng bộ, khi A bắt đầu thực hiện, chương trình tiếp tục thực hiện B mà không đợi A kết thúc.
- Việc mà bạn cần làm ở đây là cung cấp một phương thức để chương trình thực hiện khi A hoặc B kết thúc.

▼ **Callback function là gì, Promise là gì, async await là gì? so sánh?**

▼ **callback function**

- Là một hàm sẽ được thực hiện sau khi một khác đã thực hiện xong
- Trong JS, hàm là các Object dạng function. Do đó các hàm có thể lấy các hàm làm đối số và có thể được trả về bởi các hàm khác. Các hàm thực hiện điều này gọi là **HOC**.
- Bất kỳ hàm nào được truyền dưới dạng đối số thì đều được gọi là hàm **callback**

▼ **Promise**

- Là một đối tượng sẽ trả về một giá trị trong tương lai.
- Rõ hơn thì một object **Promise** đại diện cho một giá trị ở thời điểm hiện tại có thể chưa tồn tại, nhưng sẽ được xử lý và có giá trị vào một thời gian nào đó trong tương lai.
- Ta thường hay gặp các trường hợp sử dụng quá nhiều **callback** dãn đến tình trạng gọi **callback hell**, nên **promise** là cách để giải quyết vấn đề trên
- Việc sử dụng promise sẽ giúp cho việc cử lý không đồng bộ sẽ trở nên đồng bộ và gọn gàng hơn.
- **Promise có 3 trạng thái**
  - Pending (đang xử lý)
  - Fulfilled (đã hoàn thành)
  - Rejected (đã bị từ chối)

▼ **Async await**

- Trước đây, để làm việc với code bất đồng bộ, chúng ta sử dụng callback và promise, **Async/Await** là một cách mới để viết code bất đồng bộ.
- Nó được xây dựng trên Promise và tương thích với tất cả Promise dựa trên API
- **Async** được dùng để khai báo một hàm bất đồng bộ
  - Tự động biến đổi một hàm thông thường thành một Promise.
  - Khi gọi tới hàm async nó sẽ xử lý mọi thứ và được trả về kết quả trong hàm của nó.
  - Async cho phép sử dụng Await
- **Await** được sử dụng để chờ một Promise. Nó chỉ có thể được sử dụng bên trong một khối Async.
  - Khi được đặt trước một Promise, nó sẽ đợi cho đến khi Promise kết thúc và trả về kết quả.
  - Await chỉ làm việc với Promises, nó không hoạt động với callbacks.
  - Await chỉ có thể được sử dụng bên trong các function async.
- **Dùng Async Await vì**:
  - Ngắn gọn, sạch sẽ: viết code tuần tự, làm cho số lượng code viết giảm đáng kể.
  - Xử lý lỗi: giúp ta xử lý cả error đồng bộ lẫn bất đồng bộ theo cùng 1 cấu trúc
  - Câu lệnh điều kiện: giúp ta xử lý những câu điều kiện dựa vào kết quả trả về một cách dễ hiểu và nhanh hơn
  - Debug: việc debug trở nên dễ dàng hơn
- **Async / Await làm cho promise lỗi thời ?**
  - Khi làm việc với Async / Await, thật ra chúng ta vẫn đang sử dụng ngầm Promises.
  - Vì thế, kể cả khi đang sử dụng Async / Await cần một sự hiểu biết tốt về Promises sẽ rất tốt cho chúng ta.

    - Ngoài ra, có những trường hợp mà Async / Await không sử dụng được và chúng ta phải sử dụng Promises.
    - Ví dụ khi ta sử dụng để gọi song song các tác vụ bất đồng bộ và không phụ thuộc vào nhau thì async await không giải quyết được, nhưng **Promise.all()** có thể làm được

▼ **Shallow copy và deep copy trong js**

- **shallow copy**: nhiệm vụ của nó chỉ copy những giá trị nông, nghĩa là nó sao chép những giá trị đối tượng bình thường nhưng các giá trị lồng nhau vẫn sử dụng để tham chiếu (reference) đến đối tượng ban đầu.
- **deep copy**: đơn giản cũng giống với shallow copy nhưng các giá trị của reference trong object gốc không thay đổi trong object đã clone

▼ **localStorage & sessionStorage & cookies**

▼ **sessionStorage:** Dữ liệu chỉ tồn tại cho đến khi người dùng đóng tab hoặc đóng trình duyệt, còn lại mọi thứ đều giống localStorage

▼ **cookie:** Thường có khoảng thời gian sống nhất định và tuỳ thuộc vào mục đích sử dụng sẽ có khoảng thời gian sử dụng khác nhau, ngoài hạn sử dụng cookie còn có thời gian sống (max-age)

- Khả năng lưu trữ thông thường khoảng 4 KB
- Thông tin được gửi lên server
- Có thể đọc ở phía máy chủ khác với Local/Session Storage chỉ đọc được ở phía máy khách.

▼ **localStorage:** Dữ liệu được lưu trữ vô thời hạn, dữ liệu sẽ không mất đi trừ khi sử dụng chức năng xoá của trình duyệt hoặc dùng localStorage API để xoá.

- Khả năng lưu trữ khoảng 5MB.
- Dữ liệu không được gửi đi đến server thông qua các request header.
- Được lưu trữ trên trình duyệt của người dùng nên việc sử dụng sẽ không ảnh hưởng đến hiệu suất của trang web nhưng sẽ làm nặng trình duyệt (rất nhỏ gần như không đáng kể).

▼ **Map, Set, WeakMap, WeakSet là gì?**

▼ **Map** trong JavaScript là một **cấu trúc dữ liệu** cho phép lưu trữ dữ liệu theo kiểu **key-value**, tương tự như object. Tuy nhiên, chúng khác nhau ở chỗ là:

- **Object** chỉ cho phép sử dụng String hay Symbol làm key
- **Map** cho phép mọi kiểu dữ liệu (String, Number, Boolean, NaN, Object,...) có thể làm key và Map có thuộc tính **size** và một số phương thức đặc trưng
- Một số phương thức và thuộc tính của Map:
  - `new Map([iterable])`: khởi tạo Map với tham số là một iterable object (không bắt buộc) với mỗi phần tử có dạng `[key, value]`.
  - `map.set(key, value)`: lưu `value` bởi `key` và trả về `map`.
  - `map.get(key)`: trả về `value` bởi `key`, nếu `key` không tồn tại thì trả về `undefined`.
  - `map.has(key)`: trả về `true` nếu `key` tồn tại, ngược lại thì trả về `false`.
  - `map.delete(key)`: xóa giá trị ứng với `key` và trả về `true` nếu `key` tồn tại, ngược lại thì trả về `false`.
  - `map.clear()`: xóa tất cả các phần tử trong `map`.
  - `map.size`: trả về số phần tử hiện tại có trong `map`.

▼ **Set trong Javascript** là một loại object cho phép bạn lưu trữ dữ liệu một cách duy nhất, không trùng lặp.

- Set là một loại object
- Dữ liệu trong set là duy nhất và không trùng lặp. Không trùng lặp ở đây được hiểu là các phần tử không được giống nhau.
- Có thể lưu `NaN` và `undefined` vào Set trong JavaScript.
- Các phương thức của Set là:
  - `new Set(iterable)`: khởi tạo Set bằng cách truyền vào một iterable object (không bắt buộc), trường hợp không truyền vào tham số nào thì Set sẽ rỗng.
  - `set.add(value)`: thêm phần tử `value` vào Set và trả về chính Set đó.
  - `set.delete(value)`: xóa một phần tử trong Set và trả về `true` nếu giá trị `value` tồn tại trong Set, ngược lại trả về `false`.

- `set.has(value)`: trả về `true` nếu giá trị `value` tồn tại trong Set, ngược lại thì trả về `false`.
- `set.clear()`: xóa tất cả các phần tử trong Set.
- `set.size`: trả về số lượng phần tử trong Set.

https://completejavascript.com/ban-biet-gi-ve-set-trong-javascript/

▼ WeakMap

- WeakMap trong JavaScript tương tự như Map, cho phép lưu trữ dữ liệu theo kiểu `key-value`. Tuy nhiên, WeakMap chỉ chấp nhận **object** làm **key** còn map thì cho phép tất cả mọi kiểu dữ liệu
- Khi object bị hủy, object tương ứng trong WeakMap sẽ bị giải phóng vì không còn cách nào để truy cập vào object đó nữa.

```
let alex = { name: "Alex" };

// khai báo map và sử dụng biến alex làm key cho map
let map = new Map();
map.set(alex, "1");

// ghi đè giá trị của biến alex
alex = null;

// mặc dù biến alex bị gán bằng null, nhưng object alex vẫn tồn tại trong map
console.log(map.size); // 1
for (let item of map) {
  console.log(item);
  /**
   * [{name: 'Alex'}, '1']
   */
}
// Để hủy ta phải dùng method delete trong map
```

```
let alex = { name: "Alex" };

// khai báo map và sử dụng biến alex làm key cho weakMap
let weakMap = new WeakMap();
weakMap.set(alex, "1");

// ghi đè giá trị của biến alex
alex = null;

// Sau khi biến alex bị gán bằng null,
```

```
// không còn cách nào có thể truy cập vào phần tử với key là alex trước đó.
// Vì vậy, object với alex sẽ bị hủy.
```

- WeakMap không phải iterable object nên không có cách nào để duyệt hết các phần tử trong WeakMap (như cách dùng `for...of` với Map).
- Ứng dụng của weakMap:
  - Lưu dữ liệu ví dụ lưu lại số lần truy cập của một đối tượng user
  - Caching dữ liệu
- Các phương thức của WeakMap là:
  - `weakMap.set(key, value)`: lưu giá trị `value` vào thuộc tính `key` và trả về chính WeakMap.
  - `weakMap.get(key)`: trả về giá trị của thuộc tính `key`, nếu `key` không tồn tại thì trả về `undefined`.
  - `weakMap.delete(key)`: xóa thuộc tính `key` trong WeakMap, nếu `key` tồn tại thì trả về `true`, ngược lại thì trả về `false`.
  - `weakMap.has(key)`: trả về `true` nếu `key` tồn tại trong `weakMap`, ngược lại thì trả về `false`.

https://completejavascript.com/weakmap-trong-javascript/

▼ WeakSet

- WeakSet trong JavaScript tương tự như Set, cho phép **lưu trữ dữ liệu một cách duy nhất**, không trùng lặp. Tuy nhiên, WeakSet chỉ chấp nhận phần tử kiểu object.
- Khi object bị hủy, object tương ứng trong WeakSet sẽ bị giải phóng vì không còn cách nào để truy cập vào object đó nữa.
- WeakSet không phải iterable object nên không có cách nào để duyệt hết các phần tử trong WeakSet (như cách dùng `for...of` với Set).
- Các phương thức của WeakSet là:
  - `weakSet.add(value)`: lưu giá trị `value` vào WeakSet và trả về chính WeakSet.

    - `weakSet.delete(value)`: xóa phần tử `value` trong WeakSet, nếu `value` tồn tại thì trả về `true`, ngược lại thì trả về `false`.
    - `weakSet.has(value)`: trả về `true` nếu `value` tồn tại trong `weakSet`, ngược lại thì trả về `false`.

▼ **Sự khác biệt giữa `Set`, `WeakSet`, `Map` và `WeakMap` trong JavaScript là gì?**

- `WeakSet` và `Set` đều là tập hợp các giá trị duy nhất. Sự khác biệt chính là `WeakSet` chỉ lưu trữ đối tượng và không thể chứa các giá trị tùy ý thuộc bất kỳ loại nào, nhưng các `Set` thì có thể.
- Sets hữu ích khi bạn cần nối từng dữ liệu một vào cấu trúc dữ liệu nhưng cũng muốn loại bỏ các phần trùng lặp. Các hoạt động tập hợp có giá trị trung bình là `O(1)`, điều này làm cho chúng tiết kiệm thời gian.
- `WeakMap` và `Map` là tập hợp các cặp khóa / giá trị. Sự khác biệt chính là trong `WeakMap`, các khóa phải là các đối tượng. Trong `Map`, các khóa có thể thuộc bất kỳ loại nào.
- Cũng cần biết rằng các giá trị `WeakMap` không thể được lặp lại và chúng giữ một tham chiếu yếu (weak reference) đến khóa. Ví dụ: nếu bạn xóa thủ công một khóa được tham chiếu trong `WeakMap`, khóa đó sẽ được thu gom.

▼ **setTimeout, setInterval**

- Là những hàm cho phép bạn thực hiện một đoạn mã Javascript tại một thời điểm nào đó trong tương lai. Nó được gọi là "lập lịch một cuộc gọi" (scheduling a call)
    - **setTimeout**: sử dụng để thực thi một hàm hoặc đoạn mã được chỉ định chỉ một lần sau một khoảng thời gian nhất định.
    - **setInterval**: sử dụng để thực thi một hàm hoặc đoạn mã được chỉ định lặp đi lặp lại vào những khoảng thời gian cố định.
- Dừng thực thi bộ đếm thời gian sử dụng: **clearTimeout()** và **clearInterval()**

▼ **Event loop trong JS**

- **Event Loop** là cơ chế giúp Javascript có thể thực hiện nhiều thao tác cùng một lúc (concurrent model)
- Cuối cùng thì event loop là gì ?
- Event loop trong JS

- Do JS đơn luồng nên nó chỉ xử lý được duy nhất một tác vụ trong cùng một thời điểm, điều này rất khác với các ngôn ngữ đa luồng (multi-thread) khác như C#, Java, PHP với mỗi tác vụ thì nó sẽ chia ra một luồng để xử lý.
- Nếu như thế thì JS rất dễ rơi vào tình trạng **Blocking** nếu gặp phải các tác vụ mất nhiều thời gian như xử lý nhiều request, call APIs, …
- Event Loop ra đời để giải quyết vấn đề này, khiến JS chỉ đơn luồng như cân tất nhiều tác vụ cùng lúc
- Về Event Loop thì cần biết thêm 2 khái niệm đó chính là:
    - **Web APIs**: bản chất Runtime của Javascript chỉ có 1 luồng và không thể chạy multi-thread, vì thế browser đã viết thêm một Web APIs để bọc runtime này lại (tương tự dưới NodeJS sẽ dùng C++ để bọc V8 lại). Web APIs này sẽ giúp cho JS có thể hoạt động một cách bất đồng bộ như multi-thread.
    - **Callback Queue**: Như tên của nó là hàng đợi các callback do thằng Web APIs ở trên trả về.
- **Cách hoạt động:**
    - Khi hàm main được chạy thì các đoạn code trong main sẽ được thực thi. Nó sẽ lần lượt đẩy các hàm vào bên trong call stack theo nguyên tắc LIFO
    - Các hàm hay tác vụ liên quan đến Events (click, change, listener, …), AJAX (Call APIs), Timing (setTimeout, setInterval) sẽ được đẩy từ call stack sang Web APIs. Còn lại thì sẽ được thực thi trong call stack đến khi nào xong thì pop nó ra cho hàm bên dưới được thực thi.
    - Ở Web APIs sẽ tận dụng các nhân của thiết bị để xử lý riêng biệt các tác vụ này. Sau khi hoàn tất thì Web APIs sẽ trả về một callback và đẩy nó vào trong Callback Queue.
    - Callack Queue hoạt động theo nguyên tắc của queue là FIFO (vào trước ra trước) không như stack.
    - Event loop hiểu nôm na là một vòng lặp vô tận, nó luôn trực chờ ở đó để quan sát Callback Queue và bé Call stack.
    - Bất kể khi nào mà call stack trống (tất cả các hàm được pop ra) thì nó sẽ nắm các thằng callback ở trong Callback Queue và ném vào trong Call Stack để tiếp tục thực thi.

TS

▼ **Các kiểu dữ liệu trong TS?**

- **union**: cho phép sử dụng nhiều kiểu dữ liệu trong 1 biến
- **any**: khi không biết trước kiểu dữ liệu là gì
- **void**: dùng để thông báo function không có giá trị trả về
- **never**: giá trị đó sẽ không xảy ra. Được sử dụng khi chắc chắn việc gì đó không xảy ra
- **unknown:** giống như any nhưng ta không thể thực hiện bất kỳ thao tác nào khi mà chưa xác định type cụ thể của biến đó.
- **tuple**: có thể hiểu là kiểu dữ liệu mở rộng của array. Giúp chúng ta kiểm soát được thứ tự kiểu dữ liệu của các phần tử trong array.
- **intersection:** cho phép bạn kết hợp các thành viên của hai hoặc nhiều kiểu bằng cách sử dụng toán tử `&`. Điều này cho phép bạn kết hợp các kiểu hiện có để có được một kiểu duy nhất với tất cả các tính năng bạn cần.

▼ **null và undefined trong TS?**

- **null**: giá trị null cho biết không có giá trị. Một biến null không trỏ đến bất kỳ đối tượng nào. Do đó, bạn không thể truy cập bất kỳ thuộc tính nào trên biến hoặc gọi một phương thức trên đó.
- **undefined:** khi một biến được khai báo mà không tạo giá trị, nó sẽ được gán giá trị undefined.

▼ **Enum trong TS?**

- Enum là từ viết tắt của Enumeration (sự liệt kê), Enum dùng để định nghĩa kiểu dữ liệu với số lượng giá trị hữu hạn.

▼ **Tự suy trong TS là gì ?**

- TypeScript có thể tự suy kiểu của biến nếu bạn không cung cấp kiểu cụ thể.

- Diều này gọi là tự suy kiểu. Nó thường dùng khi các biến hoặc tham số được khởi tạo khi khai báo.

▼ **Interface và type khác nhau gì ?**

- Cả **Interface** và **type** trong ts cho phép bạn định nghĩ thuộc tính và phương thức là gì mà đối tượng cần để được triển khai (implement).
- Nếu đối tượng tuân thủ đúng theo khuôn mẫu thì sẽ được thực thi đúng ngược lại sẽ báo lỗi
- **Một số điểm khác nhau:**
  - **Type aliases có thể sử dụng computed properties**
    - Từ khóa `in` có thể được sử dụng để iterate tất cả các item bên trong một tập hợp keys. Chúng ta có thể sử dụng tính năng này để tạo mapped types.
  - **interface** có thể kế thừa từ 1 interface khác được còn **type** thì không
  - **Interface** có thể **merge**, **type** thì không. Nhiều khai báo có cùng tên chỉ hợp lệ khi sử dụng `interface`. Làm như vậy sẽ không ghi đè trước đó mà tạo ra kết quả hợp nhất chứa từ tất cả các khai báo

▼ **Generic trong TS?**

- Hiểu nôm na: kiểu dữ liệu mà có nhận tham số và trả về kiểu dữ liệu tương ứng.
- Hoặc hiểu đơn giản thì Generic type là việc cho phép truyền type vào components(function, class, interface) như là 1 tham số.Điều này sẽ giúp các components mềm dẻo hơn. Tái sử dụng tốt hơn.

REACTJS

▼ **React là gì, ưu điểm, hạn chế?**

- React là một lib, open source đùng để phát triển giao diện người dùng. Dùng cho các app SPA. Nó hữu ích trong việc tạo các UI phức tạp và có thể tái sử dụng

tuân theo mô hình component.

- Một số tính năng của react như:
  - Tăng hiệu suất của app bằng việc sử dụng Virtual DOM
  - JSX giúp code dễ đọc và dễ viết
  - Có thể render cả 2 phía client và server
  - Dễ testing hoặc kết hợp với những framework khác
  - Cộng đồng lớn và được hậu thuẫn bởi facebook
- **Ưu điểm**
  - Viết code dễ dàng hơn khi sử dụng JSX có thể nhúng mã HTML CSS và JS
  - Sử dụng component, giúp chia nhỏ ứng dụng thành phần những thành phần nhỏ hơn và có thể tái sử dụng được
  - Hệ sinh thái đa dạng từ app CSR, SSR, app native, hay app desktop với electron, …
- **Nhược điểm:**
  - React không phải là 1 framework hoàn chỉnh mà chỉ là thư viện, phụ thuộc vào cộng đồng và LIB bên ngoài nhiều

▼ **Vòng đời life cycle của REACT**

Có 3 giai đoạn trong vòng đời component React.

- **Mounting**: đề cập đến việc đưa phần tử vào DOM của trình duyệt. Vì React dùng virtual DOM, toàn bộ DOM của trình duyệt đã render sẽ không được làm mới. Bao gồm các phương thức trong giai đoạn này như: `constructor` và `componentDidMount`.
- **Updating**: Trong giai đoạn này, component sẽ được cập nhật khi có thay đổi state hoặc props của component. Các phương thức trong giai đoạn này: `getDerivedStateFromProps`, `shouldComponentUpdate`, `render`, và `componentDidUpdate`.
- **Unmounting**: Ở giai đoạn cuối, component sẽ bị xoá khỏi DOM. Giai đoạn này sẽ có phương thức là `componentWillUnmount`

Ngoài ra còn có nhiều method khác nữa nhưng đây là những method hay dùng nhất trong vòng đời react

---

Các phương thức trong vòng đời:

- `constructor()`: phương thức được gọi khi component được tạo trước khi thực hiện bất kỳ hành động gì. Nó giúp tạo state và props.
- `getDerivedStateFromProps()`: nó sẽ gọi trước khi phần tử được render vào DOM. Nó giúp thiết lập đối tượng state dựa trên props khởi tạo. Phương thức `getDerivedStateFromProps` sẽ có một state như đối số và trả về một đối tượng để thay đổi state. Nó sẽ là phương thức đầu tiên được gọi khi thực hiện cập nhật.
- `render()`: phương thức này sẽ render HTML từ DOM với thay đổi mới nhất. Phương thức `render` sẽ được gọi mỗi khi có thay đổi đến component.
- `componentDidMount()`: phương thức sẽ được gọi sau khi render component. Ta có thể chạy lệnh cần component đã được lưu trong DOM.
- `shouldComponentUpdate()`: trả về giá trị boolean để quyết định xem có render hay không. Mặc định sẽ là True.
- `getSnapshotBeforeUpdate()`: cung cáp truy cập cho props cung như state trước khi cập nhật. Nó dùng cho kiểm tra giá trị trước khi cập nhật.
- `componentDidUpdate()`: được gọi sau khi cập nhật component trong DOM.
- `componentWillUnmount()`: phương thức được gọi khi component bị xoá khỏi DOM.

▼ **Virtual DOM và Real DOM**

- `DOM` là viết tắt của Document Object Model (Mô hình Đối tượng Tài liệu) dùng để truy xuất các tài liệu dạng HTML và XML. DOM đại diện cho một tài liệu như là một cây cấu trúc dữ liệu. Còn node thì đại diện cho một phần tử trong DOM.
- `Virtual DOM` (VDOM hay DOM ảo) , là cách thể hiện DOM thật của một trang web dưới dạng các Javascript object. Khi thay đổi state của app thì VDOM sẽ được cập nhật lại và so sánh với VDOM cũ (VDOM cũ được đồng bộ hóa với DOM thật trước đó) bằng thuật toán gọi là diffing hay change detection để tìm ra những node cần thay đổi. Cuối cùng nó sẽ cập nhật những node đó trên DOM thật.

▼ **Tại sao cần VirtualDom ?**

- Thao tác DOM là 1 phần không thể thiếu của bất ký app nào. Tuy nhiên thao tác DOM khá chậm so với các thao tác trong JS
- Dẫn đến hiệu năng của app bị ảnh hưởng khi thực hiện thao tác trực tiếp trên DOM
- Trước đó thì những Framework JS sẽ cập nhật toàn bộ lại DOM dù cho ta chỉ có thay đổi 1 hoặc 1 vài thành phần
- React đưa ra khái niệm VD để giải quyết vấn đề đó
- Đối với mỗi đối tượng DOM sẽ có 1 VD tương ứng, nó có các tính giống nhau.
- Sự khác nhau cơ bản là khi có sự thay đổi trên VD nó sẽ không phản ánh trực tiếp lên màn hình.
- React sử dụng 2 VD để hiển thị. Một cái dùng để lưu trữ trạng thái hiện tại và 1 cái là trước đó.
- Khi có sự cập nhật trên VD nó sẽ so sánh 2 bản VD đó để tìm ra node cần thay đổi.
- Và chỉ cập nhật những node đó trên DOM thật, thay vì toàn bộ.

▼ N**guyên tắc *single source of truth* ở trong React là gì ?**

- Thông thường với việc sử dụng HTML + JS thì state hoặc value của thẻ `<input />` được điều khiển bằng **browser** chứ không phải là do **JS**.
- Nếu bạn cũng giữ giá trị của đầu vào như vậy trong JS thì nó có nghĩa rằng có ít nhất "two sources of truth - 2 nguồn của sự thật"
- Với controlled component trong React thì **state** và **value** luôn luôn khớp với nhau.
- Bởi vì, React luôn đảm bảo rằng giá trị của element input trong browser bằng với giá trị bạn cung cấp từ JS.

→ Nó chính là "single source of truth".

▼ **Controlled component khác gì uncontrolled component?**

- **Controlled component:**
  - giá trị của phần tử **input** được điều khiển bởi **React**

    - Ta lưu trữ trạng thái của phần tử input trong code, và sử dụng **callback,** với bất kỳ thay đổi nào đến input sẽ được phản ánh tương tự trong code.
    - Khi người dùng nhập dữ liệu vào phần tử input trong controlled component, hàm `onChange` kích hoạt
    - và ta kiểm tra giá trị nhập vào là hợp lệ hay không. Nếu hợp lệ, ta thay đổi trạng thái và re-render phần tử input với giá trị mới.
- **Uncontrolled component:**
    - gái trị của phần tử input được xử lý bởi DOM
    - các phần tử input này hoạt động giống như phần tử input HTML.
    - trạng thái của phần tử input được xử lý bởi DOM.
    - Nên khi giá trị input thay đổi, callback sẽ không được gọi.
    - Hoặc có thể nói là React không thực hiện bất cứ hành động nào khi xảy ra thay đổi.
    - Khi người dùng nhập dữ liệu vào trường input, dữ liệu cập nhật được hiển thị trực tiếp.
    - Để truy cập giá trị phần tử input, ta có thể dùng **ref**.

▼ **state và props trong react là gì ?**

- **State** là đối tượng bên trong 1 component dùng để chứa thông tin hoặc dữ liệu về component. Bất cứ sự thay đổi nào về state trong component cũng dẫn đến việc re-render. Chỉ được khởi tạo và chỉnh sửa chỉ chính bản thân component chứa nó
- **Props** là đối tượng nhận vào của 1 component, cho phép giao tiếp những component với nhau bằng cách truyền tham số qua lại giữa các component

→ Điểm khác nhau lớn nhất giữa props và state đó là props không thể thay đổi, còn state có thể thay đổi do đó hiệu năng của props tốt hơn state.

▼ **stateless và stateful component là gì ?**

- Stateless component là các component chỉ chứa props, các component loại này chỉ dùng để render() thì sẽ hiệu quả hơn.

- Stateful Component là các component chứa cả props và state, các component này được dùng xử lý data, phản hồi yêu cầu người dùng, phù hợp cho mô hình client server…

▼ **JSX là gì ?**

- JSX là viết tắt của JavaScript XML.
- Nó cho phép ta viết HTML trong JS và đặt nó vào DOM mà không cần dùng `appendChild()` hay `createElement()`.

▼ **Keys trong react dùng để làm gì**

- Key là một thuộc tính đặc biệt trong element được dùng khi render ra danh sách các phần tử
- Key giúp chúng ta định danh các phần tử trong 1 danh sách, mỗi phần tử là unique trong danh sách
- Nếu không dùng key thì react nó sẽ không hiểu được thứ tự của từng phần từ trong danh sách.

▼ **Sự khác nhau giữa class component và function component?**

- Trước đây, các function component được gọi là stateless component và ít dùng trong react.
- Sau khi hooks ra đời ở những phiên bản sau thì việc sử dụng function component ngày càng nhiều
- Dù function component đang là trend hiện tại, nhưng class component vẫn còn rất quan trọng.
- Một số điểm khác biệt đến từ
  - Cách khai báo: function thay vì class
  - Cách xử lý props, state
  - Cú pháp

▼ **React hook là gì và tại sao cần dùng nó?**

- RH là một tính năng mới của react được giới thiệu ở phiên bản 16.
- Giúp chúng ta viết component bằng function thay vì sử dụng class như các phiên bản trước.

- Nó giúp chúng ta viết code linh hoạt và ngắn gọn hơn
- Nó không thay thế hoàn toàn class, nó chỉ là cách viết component mới, những tính năng của class đều có trên hooks tuy nhiên cú pháp khác nhau mà thôi.

▼ **Tại sao dùng hooks thay vì class ?**

- Trước đây, các function component được gọi là stateless component.
- Chỉ các class component mới được sử dụng cho các phương thức quản lý trạng thái và vòng đời. Nhưng vì class component quá nặng nếu như chỉ cần thay đổi một vài state hay phương thức trong lifecycle.
- Điều đó dẫn đến sự ra đời của React Hooks.

▼ **Hiệu suất của hooks so với class ?**

- React Hooks sẽ tránh được rất nhiều chi phí như tạo thực thể, liên kết các sự kiện, .., có trong các lớp.
- Các hook trong React sẽ dẫn đến các cây component nhỏ hơn vì chúng sẽ tránh được việc lồng nhau tồn tại trong HOC và sẽ render props dẫn đến việc React phải thực hiện ít công việc hơn

▼ **Các quy tắc sử dụng hooks?**

- Chỉ có thể gọi hooks trong function component (không thể dùng trong class).
- Chỉ được gọi hooks ở top level, không được gọi trong 1 loop, condition hay trong 1 nested function

▼ **Giới thiệu một số hooks cơ bản của react**

Trong react hooks nó sẽ gồm 2 loại:

- **Hooks được cung cấp sẵn từ react:**
  - **useState()**: dùng để thiết lập và chỉnh sửa state trong component
  - **useEffect()**: dùng để thực hiện những side effects trên function component
  - **useContext()**: dùng để tạo dữ liệu chung có thể truy cập trong hệ thống phân cấp component mà không cần truyền dữ liệu từ trên xuống dưới.
  - **useReducer()**: dùng khi các logic state của component trở nên phức tạp thì dùng nó sẽ giúp quản lý trở nên dễ dàng hơn. Nó có thể được xem là phiên bản nâng cấp của useState()

- useMemo(): được sử dụng để tính toán lại giá trị đã ghi nhớ khi có sự thay đổi trong một trong các dependencies, giúp tránh các tính toán tốn kém trên mỗi lần render.
- useCallback(): giúp tránh một số trường hợp useEffect từ các component con thực thi lại khi nhận callback là một dependencies từ phía component cha. Nó mất 1 vùng nhớ nhất định để ghi nhớ được function mà chúng ta bọc ở trong useCallback.
- **useRef():** nó sẽ cho phép tạo một tham chiếu đến phần tử DOM trực tiếp trong function component. Ngoài ra nó còn là 1 function trả về object với thuộc tính current được khởi tạo thông qua tham số truyền vào. Object được trả về này có thể mutable và sẽ tồn tại xuyên suốt vòng đời của component.
- **useLayoutEffect()**: dùng cho đọc bố cục từ DOM và re-render bất đồng bộ

- **Custom hooks**: là một hook đặc biệt do mình tự định nghĩa ra, giúp ta tách biệt logic ra khỏi UI và có thể chia sẻ logic giữa các component.
  - Trong custom hook ta có thể sử dụng lại các hook có sẵn hoặc kết hợp với những custom hook khác.
  - Đặt tên custom hook với prefix là use.
  - Custom hooks return data thay vì JSX như component.
  - Khi nào cần dùng:
    - Khi một đoạn code (logic) được tái sử dụng nhiều nơi (dễ thấy khi bạn copy cả 1 đoạn code mà không cần sửa gì, trừ tham số truyền vào. Tách như cách mà bạn tách một function).
    - Khi logic quá dài và phức tạp. Bạn muốn viết nó ở 1 file khác, để component của bạn ngắn hơn và dễ đọc hơn vì không cần quan tâm đến logic của hook đó nữa.
  - Một số custom hooks như: useAuthentication, useAuthorization, useNotification, useScroll, useFetch, …

▼ **useCallback khác gì useMemo ?**

- useCallback: ghi nhớ 1 function, thường được sử dụng để tránh việc function của component cha gây ra tình trạng re-render của 1 component con

- useMemo: ghi nhớ 1 giá trị, thường được sử dụng để tránh việc thực hiện lại các tính toán phức tạp khi dữ liệu đầu vào không hề thay đổi

▼ **Trường hợp sử dụng useEffect và useLayoutEffect như thế nào?**

- Sự khác nhau giữa **useEffect** và **useLayoutEffect** là thời điểm chúng được gọi. Để hiểu được khi nào chúng được gọi, chúng ta theo dõi các render của DOM.
- Giả sử chúng ta triển khai một hook **useEffect** sau:
  - User tương tác với App. VD: Click vào một button
  - `State` của component sẽ thay đổi
  - DOM sẽ thay đổi
  - UI được thay đổi trên màn hình
  - Hàm `cleanup` sẽ được gọi để `clean` những `effect` đã render trước đó nếu đối số thứ 2 của `useEffect` thay đổi.
  - `useEffect` hook sẽ được gọi
- Đối với **useLayoutEffect:**
  - User tương tác với App. VD: Click vào một button
  - `State` của component sẽ thay đổi
  - DOM sẽ thay đổi
  - Hàm `cleanup` sẽ được gọi để `clean` những `effect` đã render trước đó nếu đối số thứ 2 của `useEffect` thay đổi.
  - `useLayoutEffect` hook sẽ được gọi
  - UI được thay đổi trên màn hình

▼ **Tại sao setState không trả về async**

- **setState**: KHÔNG TRẢ VỀ ASYNC mà nó trả về 1 dispatch function. Vì:
  - Khi gọi useState, kết quả trả về là 1 mảng gồm: 1 giá trị + 1 dispatch function
  - **dispatch function**: nhận vào 1 giá trị và trả về void (lưu ý, là trả về `void`, không phải `promise`) nên `setState` không phải `async`
- **Nếu không phải async thì tại sao nó không thể update giá trị ngay?**

    - Theo như reactjs có nói: Sau khi giá trị được truyền vào, thì nó sẽ đi vào 1 hàng đợi, và chờ được xử lý
    - Đến khi component **re-render** thì giá trị mới sẽ được cập nhật

▼ **Redux, Context API, Hooks có thực sự giống nhau ?**

Thực sự 3 thứ này là khác hoàn toàn, chúng có thể bổ trợ cho nhau nhưng về bản chất thì khác.

- Redux là thư viện để quản lý state và chia sẻ state giữa các component
- Bản thân redux cũng có dựa trên context API
- Về mặt nào đó thì Context API cũng có thể làm phần việc của redux nhưng không phải là tất cả, bởi phải xử lý nhiều mới tối ưu được như thằng redux nếu dùng Context API thô
- Còn đối với hooks thì đó là cách implement mới của react giúp việc functional component thuận tiện hơn.
- Cơ bản thì vẫn có những hook để xử lý local state như useReducer, useState, useRef.
- Ngoài ra còn có những hook để xử lý context như useContext. Vậy nếu muốn dùng context bạn vẫn phải qua Context API và hook chỉ là phương tiện hỗ trợ cho dễ dàng hơn thôi

▼ **Context trong React**

- Context cung cấp phương pháp truyền data xuyên suốt component tree mà không cần phải truyền props một cách thủ công qua từng level. link
- Khi nào nên dùng:
    - Context được thiết kế để chia sẽ data khi chúng được xem là “global data” của toàn bộ ứng dụng React, chẳng hạn như thông tin về user hiện tại đang đăng nhập, theme, hoặc ngôn ngữ mà người dùng đã chọn
    - Sử dụng context, chúng ta có thể tránh được việc truyền props qua các elements trung gian

▼ **React hook có làm việc với static typing?**

- Static typing đề cập đến quá trình kiểm tra code trong suốt thời gian biên dịch để đảm bảo mọi biến đề sẽ được nhập.

- React Hook là hàm được thiết kế để đảm bảo mọi thuộc tính sẽ được nhập tĩnh. Để thực thi nhập tĩnh chặt chẽ hơn trong code, ta có thể sử dụng API React với các Hook tùy chỉnh.

▼ **Làm thế nào để giữ được state trước đó với hooks ?**

- Nếu dùng state khi thay đổi nó sẽ trigger re-render, còn nếu dùng local variable thì nó sẽ bị reset sau mỗi lần re-render
- Ta có thể dùng global variable để giữ state trước đó tuy nhiên thì cách này không khuyến khích dùng.

→ Sử dụng **refs**, hoặc **useRef** của hooks

- Khi giá trị của ref thay đổi, nó không trigger re-render

▼ **So sánh component và Pure component? Pure componet và React.Memo có giống nhau?**

- `React.Component` *cho phép dev override lại* `shouldComponentUpdate` *hook, mặc định hook này reference compare để quyết định re-render lại hay không.*
- `React.PureComponent` *không cho phép dev override lại* `shouldComponentUpdate` *hook, nếu bạn cố tình override thì bạn sẽ ăn ngay warning. Hook này shallow compare để quyết định re-render lại hay không.*
- **PureComponent** giúp chúng ta kiểm tra props và state xem có sự thay đổi về giá trị không để cho phép render lại UI cần thiết. Bản chất PureComponent đã override lại hàm shouldComponentUpdate và kiểm tra giá trị ở props và state để trả về true/false cho việc render UI này.
- **React.memo():** là một HOC, chứ không phải là hooks, tương tự như là PureComponent, chỉ render lại component nếu props có sự thay đổi, sử dụng cơ chế shallow comparison.
- Shallow comparison là chỉ so sánh những giá trị của các thuộc tính ngoài cùng của đối tượng, những thuộc tính lồng nhau và tham chiếu đến đối tượng khác sẽ không so sánh được
- Deep comparison cũng giống như shallow nhưng nó so sánh luôn những giá trị đối tương lồng nhau trong các thuộc tính
- Reference compare thì nó so sánh địa chỉ của đối tượng trong bộ nhớ thay vì so sánh giá trị của đối tượng.

▼ **Các kiểu side effects trong component là gì ?**

- Side effects dùng để:
  - Gọi API lấy dữ liệu
  - Tương tác với DOM
  - Subscriptions
  - setTimeOut, setInterval
- Có 2 loại side effects:
  - Effects **không cần clean up**: gọi API, tương tác với DOM
  - Effects **cần clean up**: subscriptions, setTimeOut, setInterval. Để dọn dẹp bộ nhớ khi unmouting tránh sự cố rò rĩ bộ nhớ hoặc những lỗi không rõ nguyên nhân.

▼ **Prop drilling trong react là gì ?**

- Đôi khi trong react ta cần phải truyền dữ liệu từ component cao hơn xuống sâu component bên dưới. Để truyền được như vậy ta phải truyền qua rất nhiều component trung gian cho đến khi đến component cần nhận props. Đó là prop drilling.
- Tuy nhiên khi app càng lớn prop drilling làm cho việc truy cập dữ liệu hết sức phức tạp

▼ **Strict mode trong react là gì ?**

- StrictMode là công cụ được thêm vào ở React v16.3 để highlight các vấn đề tiềm ẩn trong React. Nó thực hiện kiểm tra bổ sung lên ứng dụng.
- StrictMode giúp giải quyết các vấn đề sau:
  - Khi chúng ta gọi hàm bất đồng bộ tại 1 lifecycle không an toàn. Strictmode sẽ cung cấp cho ta cái cảnh báo về việc sử dụng đó.
  - Cảnh báo khi chúng ta sử dụng findDom() để tìm cây của node trong DOM. Vì phương thức này react không còn hỗ trợ cho nên sẽ đưa ra cảnh báo.
- Nói chung thì nó giúp chúng ta tránh những lỗi tiềm ẩn có thể xảy ra trong quá trình chạy.

▼ **Higher order component trong react là gì ? (HOC)**

- **Higher order function** là một function mà nó nhận vào tham số là function hoặc return về một function.
- **Higher order component** là 1 function và nó nhận vào tham số là 1 component nó sẽ return về một component.

⇒ Khi sử dụng **HOC** thì có 3 điểm bạn cần lưu ý khi sử dụng là:

- Không sử dụng **HOC** trong phương thức render()
- Các phương thức static cần phải được copy lại
- Refs không được truyền qua **HOC**

⇒ Có thể gặp HOC ở:

- **withRouter** của React Route
- hàm **connect** của React-redux

▼ **Các cách khác nhau để chỉnh style component trong react?**

- **Inline Styling:** ta có thể chỉnh style trực tiếp lên phần tử bằng cách dùng thuộc tính style. Nhớ giá trị của style luôn là đối tượng JavaScript
- **Javascript Object**: ta có thể tạo đối tượng JavaScript và tập mô tả thuộc tính style. Các đối tượng có thể dùng như giá trị của thuộc tính style.
- **CSS Stylesheet:** Ta sẽ tạo một file CSS riêng và viết tất cả style cho component trong file đó. Sau đó import nó vào file React.
- **CSS Module:** Tương tự như file CSS, nhưng ta sửa thành `.module.css`, với cách này tên lớp sẽ được mã hoá, đồng thời nó hỗ trợ kiểu viết tương tự **sass**.

▼ **Error boundary là gì ?**

- Được giới thiệu ở React v16, error boundary cung cấp một cách để xử lý lỗi xảy ra trong giai đoạn render.
- Bất kỳ component nào sử dụng các phương thức lifecycle cũng được xem là một error boundary. Các vị trí mà error boundary có thể được phát hiện:
  - Giai đoạn render
  - Trong một phương thức lifecycle
  - Trong constructor

- Khi không dùng error boundary khi có error xảy ra như ở trên ta sẽ thấy một trang trống thay vì lỗi.
- Bất cứ lỗi nào trong phương thức render đều dẫn đến unmounting component.
- Để hiển thị lỗi khi đó, ta sử dụng error boundary. Là một component bọc ngoài các component.

▼ **Làm thế nào để ngăn chặn re-render trong react ?**

- Nguyên nhân của việc gây ra re-render là có sự thay đổi của 1 state hoặc props trên component
- Ta có thể overide lại hook **shouldComponentUpdate()** để ngăn chặn việc re-render
- Hoặc sử dụng một số kỹ thuật như useMemo, useCallBack, …

▼ **Các kỹ thuật tối ưu hiệu suất (optimize performance) trong react là gì ?**

- **useMemo()**
  - Là hook dùng cho caching CPU.
  - Đôi khi trong các ứng dụng web, các hàm đắt (tính toán nhiều, tốn bộ nhớ) được gọi liên túc do re-render dẫn đến tốc độ render chậm, hiệu suất kém.
  - useMemo() có thể sử dụng cho cache cám hàm như vậy. Bằng cách dùng useMemo() các hàm đó chỉ được gọi khi cần thiết.
- **React.PureComponent**
  - Là class component cơ sở để kiểm tra state và props của một component để biết khi nào nó nên được cập nhật.
  - Thay vì dùng React.Component, ta có sử dụng React.PureComponent để giảm việc re-render không cần thiết.
- **Duy trì vị trí state**
  - Đây là quá trình chuyển state đến nơi bạn nhất có thể.
  - Thỉnh thoảng ta có các state không cần thiết nằm trong component cha để gây khó đọc và bảo trì hơn, thậm chí là dẫn đến re-render không cần thiết.
  - Để tốt hơn, ta chuyển các state vô nghĩa ở component cha sang một component riêng biệt.

- **Lazy Loading**
  - Đây là kỹ thuật dùng để giảm thời gian tải của ứng dụng React. Lazy loading giúp tối ưu hiệu suất ứng dụng web bằng cách chỉ tải khi cần thiết.

▼ **Những phương pháp giúp tối ưu performance?**

- **Code splitting**: chỉ load những page hoặc component cần thiết lúc render, không nên load hết tất cả lên, vd: khi vào homepage ta chỉ cần load page home và component liên quan đến page đó thôi
- Lazy load image: thay vì load hết tất cả img thì ta nên load những img hiển thị trên viewport, khi scroll thì tiếp tục load những hình ảnh còn lại
- Lazy size image: với mỗi screen device sẽ có những size ảnh khác nhau thay vì chỉ load 1 size ảnh cho all device
- Server side rendering
- Sử dụng CDN
- Tối ưu CSS
- Minified HTML, CSS, JS with webpack
- Tránh việc re-render nhiều lần trong app
- Thêm loading hoặc skeleton để tăng trải nghiệm người dùng

▼ **React Router là gì ?**

- Là một thư viện dùng để routing trong react. Cho phép điều hướng các trang trong app mà không cần làm mới (reload) lại toàn bộ trang.
- Nó cho phép ta thay đổi URL của trình duyệt nhưng vẫn giữ UI đồng bộ với URL

▼ **withRouter trong react-router-dom là gì?**

- `withRouter()` là một HOC cho phép truy cập thuộc tính đối tượng `history` ứng với `<Route>` gần nhất. Nó sẽ truyền `match`, `location` và `history` như props đến component được bọc bất cứ khi nào nó render.

▼ **Link và NavLink khác nhau gì ?**

- `<Link>` dùng cho điều hướng sang các trang khác nhau trong ứng dụng web, tương tự thẻ a

- `<NavLink>` khá giông link về cách sử dụng nhưng nó hỗ trợ thêm các thuộc tính như **activeClassName** và **activeStyle ,** 2 thuộc tính này giúp cho khi mà nó trùng khớp thì nó sẽ được active lên và chúng ta có thể style cho nó.

▼ **Nested routing là gì ? Khi nào cần dùng ?**

- Sử dụng **nested** route trong React Router giúp dễ dàng tạo các **nested** route trong trang web của chúng ta, giúp dễ dàng hiển thị và quản lý theo **component**.
- Ví dụ: khi cần làm một **sub menu** chúng ta có nhiều menu đa cấp thì ta sử dụng nested route giúp ta phân chia các code thành những component nhỏ bên trong, giúp ta dễ dàng quản lý và phân chia code

▼ **Setup routing cho mấy trang login như thế nào ?**

- Tạo ra một **custom route extend** từ route thông thường, trong đó ta check xem nếu chưa login thì ta redirect sang trang login, còn login rồi thì thôi
- Hoặc tạo một middleware check, nếu chưa login thì redirect sang trang login

▼ **Handle phần authentication trong app như thế nào ? Cách lưu các token ?**

- **B1**: Check cookies nếu có JWT payload thì vào các trang member nếu không redirect ra trang login
- **B2**: Ở trang login khi user hoàn tất nhập username, pass, ta gửi lên server để thực hiện việc login, nếu thành công thì lấy mã token và lưu vào cookie sau đó redirect về trang home
- **B3**: Nếu trang /login dùng chuẩn xác thực bằng OpenID thông qua một cơ chế OAuth. Theo authorization code grant flow, trang /login sẽ redirect browser về /backend/auth/<provider>. Sau đó nếu flow OAuth xong và hợp lệ (user grant đăng nhập với Facebook), server response sẽ set authentication cookie với JWT bên trong. Sau đó browser sẽ redirect về trang của SPA. SPA sẽ quay lại check như bước 1.

▼ **Bạn thường dùng thư viện nào để quản lý form ?**

- **Formik**
- **Redux-Form**
- **React-Hook-Form**

▼ **Render có điều kiện (condition) trong react ?**

- Giúp ta hiển thị kết quả dynamic dựa vào điều kiện state, hay dữ liệu chúng ta truyền vào.
- Một số cách:
  - Sử dụng if else
  - Toán tử 3 ngôi
  - Sử dụng một biến phần tử

▼ **Cách hiển thị dữ liệu API với axios?**

- Axios là một promise dựa trên HTTP để tạo yêu cầu HTTP đến trình duyệt hay web server.
- **Tính năng**
  - **Interceptors**: Truy cập cấu hình yêu cầu hoặc phản hồi (header, dữ liệu, v.v.) khi chúng gửi đến hoặc đi. Các hàm này có thể hoạt động như các cổng để kiểm tra cấu hình hoặc thêm dữ liệu.
  - **Instances**: Tạo thực thể có thể tái sử dụng như baseUrl, headers, và cấu hình khác đã thiết lập.
  - **Defaults**: Thiết lập giá trị chung cho header chung (như Authorization) với các yêu cầu. Nó hữu ích khi bạn cần xác thực đến server trên mọi yêu cầu.

▼ **Caching trong react?**

- Ta có thể caching dữ liệu trong React bằng nhiều cách như:
  - Local Storage
  - Redux Store
  - Giữ dữ liệu giữa mounting và unmounting

▼ **Có thể dùng đc component mà không extends không ?**

- Có thể được, miễn là không sử dụng **JSX**
- Tuy nhiên sẽ không sử dụng được những lifecycle methods, cũng như các props, state và render

▼ **window.reloaded vs dom.reloaded khác gì nhau ?**

- **window** là gọi khi cả html, js đc load xong

- còn **dom** là khi mới chỉ có html chưa có gì

▼ **Redux là gì ? Thành phần trong redux ? Cách hoạt động ? Nguyên tắc?**

> 💡 *Redux là 1 thư viện giúp chúng ta quản lý các state 1 cách tốt hơn. Thay vì phải truyền state qua từng Component thì Redux sẽ tạo ra 1 store duy nhất dùng để thay đổi dữ liệu.*

- **Đặt điểm:**
  - State trong redux là có thể dự đoán được
  - Redux sử dụng kiến trúc 1 chiều: uni-directional data flow
  - Redux state là READ-ONLY. Muốn update phải dispatch một action (js object)
  - Những thay đổi của redux state được thực hiện bởi Pure functions (reducer)
  - Redux có thể dùng cho các javascript apps, không chỉ riêng gì ReactJS.
- **Thành phần:**
  - `Store` gồm có:
    - `State`: là dữ liệu hiện tại được lưu trên state
    - `Reducer`: là hàm biến đổi state cũ thành state mới
    - Dispatcher: quản lý `middlewares` và chuyển dữ liệu xuống reducer.
  - `Action`: tạo ra các action dùng để mô tả event do người dùng tạo ra
  - `View`: hiển thị dữ liệu được cung cấp bởi Store.
- **Nguyên lý hoạt động của Redux:**
  - **B1**: Khi có 1 sự kiện (event) như là GET, POST, UPDATE, DELETE... thì thằng `action` creators sẽ sinh ra 1 action mô tả những gì đang xảy ra.
  - **B2**: `Action` sẽ thực hiện điều phối `Reducer` xử lý event thông qua hàm `dispatch(action)`.
  - **B3**: `Reducer` dựa vào những mô tả của `Action` để biết cần thực hiện thay đổi gì trên `State` và thực hiện update.

- **B4**: Khi `State` được update thì các trigger đang theo dõi state đó sẽ nhận được thông tin update và tiến hành render lại phần `view` để hiển thị ra cho người dùng

- **3 Nguyên tắc trong redux:**
  - **Store** luôn là nguồn dữ liệu đúng và tin cậy duy nhất.
  - **State** chỉ được phép đọc, cách duy nhất để thay đổi **State** là phát sinh một Action, và để Reducer thay đổi State.
  - Các function Reducer phải là **Pure function** (với cùng 1 đầu vào chỉ cho ra 1 đầu ra duy nhất)
- **Khi nào cần sử dụng Redux:**
  - Dữ liệu được sử dụng ở nhiều nơi
  - Có hỗ trợ chức năng `undo` / `redo`
  - Cần `cache` dữ liệu để tái sử dụng cho những lần sau.

▼ **Một số middleware trong redux ?**

> 💡 *`middleware` là một lớp nằm giữa `Reducers` và `Dispatch Action`, nó sẽ modify và được gọi trước khi action được dispatch. Thường được dùng trong việc logging, reporting, async api, routing, ...*

- **Logging, Reporting, Redux-saga, Redux-thunk, redux-persist**

▼ **Tại sao phải sử dụng middleware như redux-saga hay redux-thunk?**

Khi sử dụng Redux ta gặp một số ràng buộc như:

- Các xử lý trong `Reducers` phải là các hàm đồng bộ và pure, trả về state mới
- `Reducers` sẽ không được sử dụng các hàm `async` vì không được thay đổi `global state`

⇒ Để giải quyết các side effects cần phải thực hiện ở `middleware`

▼ **Redux saga là gì?**

`Redux-Saga` là một thư viện redux middleware, giúp quản lý những side effect trong ứng dụng redux trở nên đơn giản hơn. Bằng việc sử dụng tối đa tính năng

Generators (function*) của ES6, nó cho phép ta viết async code nhìn giống như là synchronous.

> 💡 `Generator function` *là function có khả năng hoãn lại quá trình thực thi mà vẫn giữ nguyên được context. (Nói một cách đơn giản thì generator function là 1 function có khả năng tạm ngưng trước khi hàm kết thúc và có thể tiếp tục chạy tại một thời điểm khác, khác với pure function khi được gọi sẽ thực thi hết các câu lệnh trong hàm)*

▼ **Một số helper trong redux-sage ?**

- `takeEvery()` : thực thi và trả lại kết quả của mọi actions được gọi.
- `takeLastest()` : có nghĩa là nếu chúng ta thực hiện một loạt các actions, nó sẽ chỉ thực thi và trả lại kết quả của của actions cuối cùng.
- `take()` : tạm dừng cho đến khi nhận được action
- `put()` : dispatch một action.
- `call()`: gọi function. Nếu nó return về một promise, tạm dừng saga cho đến khi promise được giải quyết.
- `race()` : chạy nhiều effect đồng thời, sau đó hủy tất cả nếu một trong số đó kết thúc

▼ **Redux thunk là gì?**

- Redux Thunk là một Middleware cho phép bạn viết các Action trả về một function thay vì một plain javascript object bằng cách trì hoãn việc đưa action đến reducer.
- Redux Thunk được sử dụng để xử lý các logic bất đồng bộ phức tạp cần truy cập đến Store hoặc đơn giản là việc lấy dữ liệu như từ server

▼ **Tại sao lại cần dùng redux toolkit ?**

Redux tool kit là một thư viện giúp chúng ta viết redux tốt hơn, dễ hơn và đơn giản hơn (tiêu chuẩn để viết redux).

Ba vấn đề làm nền tảng để RTK ra đời:

- Việc Configure một store trong redux rất phức tạp.

- Phải cài thêm nhiều package để làm việc với redux tốt hơn.
- Redux yêu cầu quá nhiều boilerplate code

▼ **Ngoài redux ra còn có thư viện nào hỗ trợ quản lý state?**

- CÓ
- Như: mobx, zustand, Context API của react, recoil

▼ **Làm thế nào để tạo menu đa cấp bằng đệ quy trong react ?**

▼ **Tree shaking là gì ?**

- “Tree shaking" là một tối ưu hóa hiệu suất bắt buộc phải có khi đóng gói JavaScript.
- Nói một cách đơn giản, Tree shaking có nghĩa là xóa code mà không sử dụng đến, hay gọi là code thừa.

▼ **Webpack là gì ?**

- Webpack được biết đến là một công cụ phần mềm được sử dụng để quản lý các module JavaScript. Nó sẽ đóng gói tất cả các mã nguồn của chương trình cũng như CSS, font, image,... khi nó hoạt động. Assets chính là tên để gọi những thứ được đóng gói này và chúng sẽ được Webpack đóng gói thành 1 file hoặc một vài file.
- **Tác dụng:** mặc dù đóng gói rất nhiều dữ liệu nhưng chúng được đóng gói một cách rất cẩn thận, bài bản và ngăn nắp, nó được sắp xếp với cấu trúc tương tự như viết mã code. Những dữ liệu này được lập trình sẵn xem cái nào chạy trước, cái nào chạy sau và phần nào sẽ phụ thuộc vào nhau.

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**