Interview 2022

▼ **Rest API questions**

1. **What is rest?**
   - REST stands for Representational State Transfer. REST is an architectural style for web development. REST architecture lays out guidelines for the transfer of resource representations between clients and servers on the web.
2. **What is a REST API?**
   - A *REST API* or *RESTful API* is a web API that conforms to the REST architecture style.
3. **Describe the 5 constraints of the REST architectural style, and their benefits**
   - Uniform interface
   - Client-server
   - Stateless
   - Cacheable
   - Layered system
4. **Explain the HTTP request methods supported by REST, and when they are used.**
   - **GET** method: Request data from server
   - **POST** method: Submit data to create new resource on server-defined URL
   - **PUT** method: Submit data to update a resource at client-defined URL
   - **DELETE** method: Remove resource from server
   - **OPTIONS** method: Return request methods supported by a service
   - **HEAD** method: Return meta information such as response headers
   - **PATCH** method: Modify part of the resource on the server

▼ **REACT & VUE**

▼ **SPA, SSA, PWA**

https://haodev.wordpress.com/2019/03/20/ssr-vs-csr/

- **Single Page Aplication**:
  - Ứng dụng sẽ render HTML, CSS ở phía client, FE sẽ xử lý nhưng logic cơ bản như như get data, validation, navigate và render. Còn BE sẽ xử lý logic để lấy data và trả về cho client thông qua API.
  - Pros:
    - Ít tốn tài nguyên của hệ thống, vì client sẽ chịu trách nhiệm render
    - Vì giao tiếp qua API nên lượng request đến server sẽ được giảm thiểu
    - Nhanh, vì các HTML, css, JS, chỉ được tải 1 lần duy nhất
    - Không cần phải load lại trang, làm tăng trải nghiệm người dùng.

    - Cons:
        - Khó SEO vì nội dung web được render phía client
        - Trình duyệt sẽ xử lý nhiều, nên vấn đề hiệu năng cần được chú ý
        -
- **Server Side Rendering**:
    - Mọi logic về validation, đọc dữ liệu, navigate, hay render đều được xử lý ở phía server
    - Cơ chế hoạt động:
        - Khi user vào trang web, browser gửi GET request tới server
        - Server nhận request, đọc dữ liệu, truy vấn database, xử lí logic, ...
        - Server sẽ render ra HTML và trả về cho client
    - Pros:
        - Hỗ trợ mạnh về SEO vì khi bot google, bing vào web sẽ thấy toàn bộ dữ liệu dưới dạng html
        - Initial load nhanh, dễ optimize vì toàn bộ dữ liệu đều đã được xử lý ở phía server, client chỉ render lại.
        - Sẽ rất thích hợp với những static page, có dữ liệu ít bị thay đổi
        - Chỉ cần code trong 1 project ko cần tách biệt ra FE và BE
    - Cons:
        - Web sẽ xử lý và load lại hoàn toàn nếu có một thay đổi nhỏ xảy ra
        - Khi lượng traffic quá lớn, làm cho server nặng và quá tải vì mọi logic đều xử lý phía server
        - Trải nghiệm người dùng không tốt, vì trang web phải refresh và load lại nhiều lần
- **Progressive web apps**
    - Hiểu đơn giản PWA là cách làm cho web app trở nên ngon hơn, ngon ở đây là khả năng web app chưa làm được.
    - Hiện tại vấn đề lớn nhất mà web app chưa làm được đó là trải nghiệm chưa được mượt mà như native app
    - Để làm được điều này PWA phải đảm bảo được 3 yếu tố:
        - Reliable: app load nhanh và có thể dụng offline
        - Fast: app phải load rất nhanh, nhấn cái chuyển trang liền hoặc animation load vù vù
        - Engaging: có khả năng dụ user sử dụng. Có thể gửi notification, badge
    - Nếu vậy sao không làm native app cho nhanh ?
        - Trên thực tế số lượng người dùng mobile sẽ nhiều hơn web, tuy nhiên thì mỗi user thường chỉ dùng những app top chart và trung bình 1 tháng chỉ cài thêm từ 1 hoặc 2 app.
        - Chi phí để tiếp cận 1 user và dụ user đó dùng app trên web sẽ rẻ hơn trên app (việc chạy quảng cáo trên web sẽ dễ dàng hơn)
        - Về mặt kỹ thuật: đỡ học 2 ngôn ngữ ios và android, có thể tạo bằng RN nhưng PWA sẽ tận dụng src code của web, RN thì không

▼ **Phương pháp SEO**

- Sử dụng thẻ <title />
- Sử dụng thẻ meta description

- Sử dụng những thẻ heading (h1 → h6)
- Broken links: không nên để link của các trang ngừng hoạt động vì ảnh hưởng đến trải nghiệm người dùng cũng như ranking của trang
- Alt attribute image: khi image không thể hiển thị thì alt của image sẽ cung cấp thông tin thay thế
- Sử dung các thẻ HTML5 như header, footer, main, section, nav, ... thay các thẻ div, span truyền thống để trang có ngữ nghĩa hơn cho các search engines
- Loại bỏ inline css trong các thẻ html
- Tối ưu hoá url cho page, url nên chứa keyword liên quan, không nên chứa space hay kí tự đặc biệt

▼ **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.**

▼ **Design pattern common**

https://viblo.asia/p/react-patterns-phan-1-yMnKM1GEK7P

- **React pattern:**
  - https://reactpatterns.js.org/docs/accessing-a-child-component
  - https://blog.openreplay.com/3-react-component-design-patterns-you-should-know-about
  - Proxy component: là component có thể tái sử dụng lại được
  - Call API in **componentDidMount**
  - Stateless function: là cách định nghĩa react component như 1 function thay vì class đồng thời nó không giữ state chỉ nhận props truyền vào
  - Higher-order function: là 1 function return về 1 function khác hoặc nhận tham số là 1 function và return về function đó
  - Higher-order component: tương tự như HOF tuy nhiên nó return hoặc nhận vào tham số là component.
  - ...
- **Vue:** https://learn-vuejs.github.io/vue-patterns/patterns/#component-declaration
  - ...

▼ **Compare react & vue, cons, pros**

-

▼ **Dom (real, virtual)**

https://viblo.asia/p/su-that-thu-vi-ve-react-co-the-ban-chua-biet-L4x5xAawKBM

- https://tonynguyenit.medium.com/how-react-virtual-dom-decide-to-update-browser-dom-91f170718733
- https://reactjs.org/docs/reconciliation.html#the-diffing-algorithm
- Real dom
  -
- Virtual dom
  -

▼ **State management (flux, redux)**

- **FLUX** là một kiến thức quen thuộc được thêm bởi Facebook để sử dụng và làm việc với React.
  - https://kipalog.com/posts/Huong-dan-va-giai-thich-Flux-bang-hinh-ve
  - Flux không được xem là một Framework hay thư viện mà nó chỉ đơn giản là một kiểu kiến trúc hỗ trợ thêm cho React.
  - Đồng thời, nó xây dựng các ý tưởng về luồng dữ liệu một chiều (tên tiếng anh là Unidirectional Data Flow).
  - Cấu trúc Flux bao gồm:
    - Actions: Có nhiệm vụ làm dẫn truyền dữ liệu đến với Dispatcher (nó được xem tương tự như Helper Method).
    - Dispatcher: Nhận những thông tin truyền đạt từ Actions để truyền tải dữ liệu tới các nơi đã thực hiện đăng ký nhận các thông tin.
    - Stores: Là nơi có nhiệm vụ lưu trữ cho trạng thái và các logic của hệ thống, đây là một trong những nơi có nhiệm vụ nhận đăng ký dữ liệu với Dispatcher.
    - Controller Views: Được cho là các React Components có nhiệm vụ nhận các trạng thái từ Stores và truyền dữ liệu cho các thành phần con.
- **REDUX:**
- **FLUX ≠ REDUX**
  - Flux có kiến trúc mang tính tổng quát còn redux thì lại chi tiết hơn vì là một phiên bản được implement từ flux và sử dụng immutable state.
  - Mặc dù phát triển dựa trên flux nhưng redux chỉ có duy nhất 1 store và đã lược bỏ đi dispatcher

▼ **Style design (css, scss, styled), The ways struct css module?**

- Có 4 cách để style css trong react:
  - **CSS stylesheet**: đơn giản là viết 1 file css và import vào component bạn muốn style
  - **Inline styling**: với react thì inline style không được thể hiện bằng 1 string mà là 1 object. Ta có thể tạo một biến để lưu trữ những style object và truyền vào element bất kỳ bằng cú pháp style={name_variable}
  - **CSS module**:
    - Là kiểu viết module hóa stylesheet thành từng file nhỏ, không còn sử dụng một file stylesheet tập trung nữa. Thêm vào đó, tất cả tên class lúc này sẽ được scope lại local.
    - Nói tóm lại, Module CSS sẽ được viết ở cùng folder với Component.
    - Một số lợi ích khi dùng css module:

        - Chỉ tồn tại ở một nơi
        - Chỉ được sử dụng ở component đó mà không sử dụng ở bất kì chỗ nào
        - Không nhất thiết phải dùng scss (muốn dùng vẫn được) vì bản chất css module đã chia nhỏ từng file css theo từng component khác nhau
        - Không sợ bị trùng tên giữa các class vì khi build với webpack tên class của CSS và element đều là duy nhất với hash code đi kèm.

            > `[Tên component]_[Tên value trong file css]__[hash string]`

    - **styled component**:
        - là một lib giúp bạn có thể tổ chức và quản lý code css 1 cách dễ dàng và hiệu quả.
        - Nó được xây dựng với mục tiêu giữ cho các style của component trong react gắn liền với các component đó
        - Không chỉ thay đổi việc implement các component mà còn thay đổi cả tư duy trong việc xây dựng styles cho các component đó.
        - Lợi ích:
            - cho phép ta encapsulate (đóng gói) style vào trong component trong js nhưng vẫn giữ được các tính năng của css như nesting, media query, pseudo-selector, ... Nó giải quyết được vấn đề global scope của css vì ta không cần phải viết selector cho class hay id, bởi styled component sẽ generate class ngẫu nhiên và truyền component thông qua property là className
            - Thay đổi style dựa trên thuộc tính hoặc trạng thái của component dễ dàng hơn. Ta có thể truyền props để thực hiện việc thay đổi style dễ dàng hơn
        - Bất lợi:
            - Tên class được generate ngẫu nhiên nên sẽ gây khó chịu cho người quen debug css bằng tên class. (ta có thể giải quyết bằng việc kết hợp css selector với styled component)
            - Còn khá non trẻ nên chưa được kiểm duyệt tính scale trong các project lớn
            - Nhiều người vẫn không thích css trong js
            - Không được dùng `ref` trên component phải chuyển sang `innerRef` bởi vì ref sẽ được truyền vào wrapper của styled component thay vì component mình muốn.

▼ **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, ...

▼ **Composition in vuejs?**

-

▼ **Proxy & Reflect ?**

-

▼ **Reactivity system in vuejs?**

- là cơ chế cho phép chúng ta phản ứng lại những thay đổi đột biến. Khi bạn thay đổi state thì view sẽ tự động được cập nhật. Điều này làm cho việc quản lý state trở nên dễ dàng và trực quan.

▼ **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

▼ **Immutable và mutable trong react**

- Mutable trạng thái/ dữ liệu của Object có thể thay đổi được.
  - `Object` và `Array` trong JavaScript mặc định đã được *mutate*
  - **Pros**: Mutate Object tạo ra side effect dẫn tới nhiều bugs không mong muốn.
- Immutable trạng thái/ dữ liệu không thể bị thay đổi.
  - Trong JavaScript, tất cả các kiểu dữ liệu nguyên thủy (primitive) đều là immutability.
  - Mỗi khi chúng ta thay đổi dữ liệu, nó sẽ tạo ra một instance mới hoàn toàn và không ảnh hưởng tới instance cũ.
  - **Pros**: Immutability là rất tốt, nó tránh được nhiều bugs nhưng vô hình chung lại làm giảm performance của app. Immutability tạo ra một bản sao giống hoàn toàn so với bản gốc, sau đó edit dữ liệu mà chúng ta muốn thay đổi trên bản sao này. Điều có có nghĩa là nó sẽ tốn rất nhiều memory cho việc copy các `Object` hoặc `Array.` Thử tưởng tượng chúng ta muốn thay đổi 1 giá trị trong một Array bao gồm 1 triệu phần tử thì sẽ tốn nhiều memory như thế nào nhỉ 🤔

  ⇒ Để giải quyết được vấn đề memory lead. Immutable sử dụng 1 cấu trúc dữ liệu có lên là **"trie data structures"**. Cấu trúc dữ liệu này sử dụng một concept là **Structure Sharing** (tối ưu memory bằng cách tái sử dụng).

  Theo cách Immutability thông thường, mỗi khi thay đổi một thuộc tính nào đó, chúng ta phải clone toàn bộ `Object` hoặc `Array` thành một bản sao, sau đó thực hiện modify trên chính bản sao đó.

  Hiện tại có khá nhiều thư viện hỗ trợ chúng ta thực hiện công việc này. Immutablejs và mori là 2 thư viện phổ biến nhất implement immutability sử dụng cấu trúc Structure Sharing.

  https://blog.daovanhung.com/post/ban-da-thuc-su-hieu-mutable-va-immutable

▼ **React hooks**

https://dynonguyen.com/nhung-react-hook-hay-dung-nhat-trong-reactjs/

▼ **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

▼ **HTML / CSS**

▼ **Lazy loading, lazy image**

https://dynonguyen.com/huong-dan-trien-khai-lazy-loading/

▼ **Tại sao sử dụng srcset trong thẻ img?**

- 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

▼ **Sự khác nhau giữa “resetting” và “normalizing” CSS là gì? Bạn sẽ chọn cái nào, và tại sao?**

- Resetting: loại bỏ tất cả kiểu trình duyệt mặc định trên các phần tử.
  - Ví dụ: `margin`, `padding`, `font-size` của tất cả các phần tử được đặt lại như cũ. Bạn sẽ phải khai báo lại kiểu dáng cho các yếu tố kiểu chữ phổ biến.
- Normalizing: Chuẩn hóa bảo tồn các kiểu mặc định hữu ích thay vì “unstyling” mọi thứ. Nó cũng sửa lỗi cho các phụ thuộc trình duyệt phổ biến.

Chọn resetting khi bạn có thiết kế trang web rất tùy chỉnh hoặc độc đáo, và bạn cần phải thực hiện nhiều kiểu của riêng mình và không cần giữ nguyên bất kỳ kiểu mặc định nào.

▼ **Floats?**

- Thuộc tính float trong CSS đặt một phần tử ở bên trái hoặc bên phải vùng chứa của nó, cho phép các phần tử văn bản và nội tuyến quấn quanh nó. Phần tử bị xóa khỏi luồng bình thường của trang, mặc dù vẫn còn lại một phần của luồng (trái ngược với vị trí tuyệt đối).

▼ **Có những cách nào để ẩn 1 element?**

- **opacity**: set về giá trị 0, ta có thể ẩn đi element đó, với cách này ta vẫn có thể đính kèm sự kiện lên nó. Nó không hoàn toàn ẩn đi mà chỉ vô hình mà thôi

```
element {
  opacity: 0;
}
```

- **color Alpha Transparency**: chỉ cần set giá trị Alpha (giá trị thứ 4) về 0 thì ta có thể ẩn đi được phần tử

```
element {
  color: rgba(r, g, b, 0);
  background-color: rgba(r, g, b, 0);
}

/* or */

element {
  color: hsla(h, s, l, 0);
  background-color: hsla(h, s, l, 0);
}
```

- **transform**: ta có thể dùng hàm scale() hoặc translate để biến đổi phần tử. Cụ thể ta có thể dùng 3 cách

```
element {
  transform: scale(0);
}

/* or */

element {
  transform: translateX(-999px);
}

/* or */

element {
  transform: translateY(-999px);
}
```

- **clip-path**

```
element {
  clip-path: circle(0);
}
```

- **visibility:**

```
element {
  visibility: hidden;
}
```

- **display**: nhận giá trị `none`, nó sẽ ẩn hoàn toàn phần tử đi và bạn không thể đính kèm được sự kiện lên phần tử đó

```
element {
  display: none;
}
```

- **hidden:** là một thuộc tính trong HTML mà ở đó nó đã được mặc định default css là `display: none`. Ta sử dụng như sau

```
<p hidden>
  Hidden content
</p>
```

- **Absolute:** Việc sử dụng `absolute` cũng khá giống như `transform`, tức là bạn có thể chỉ định 4 giá trị top/bottom/left/right thành các giá trị âm để di chuyển phần tử ra khỏi khung nhìn

```
element {
  position: absolute;
  left: -999px;
}
```

- **Overlay**: Sử dụng phần tử giả cũng có thể giúp ta ẩn đi phần tử (tuy nhiên lưu ý đây chỉ là ẩn đi "bằng việc set background theo phần tử cha")

```
element {
  position: relative;
}

element:after {
  position: absolute;
  content: '';
  top: 0;
  bottom: 0;
  left: 0;
  right: 0;
  background: #fff;
}
```

- **Giảm thiểu kích thước:** đây là sự kết hợp giữa 3 thuộc tính `height`, `padding`, `overflow`

```
element {
  height: 0;
  padding: 0;
  overflow: hidden;
}
```

▼ **Flex và grid?**

- CSS Grid Layout là hệ thống bố cục mạnh mẽ nhất hiện có trong CSS. Nó là một hệ thống 2 chiều, có nghĩa là nó có thể xử lý cả cột và hàng, không giống như flexbox phần lớn là hệ thống 1 chiều
- Flexbox (hộp linh hoạt) là một hệ thống bố cục một chiều (theo chiều ngang hoặc chiều dọc), nhằm mục đích cung cấp một cách hiệu quả việc bố trí, căn chỉnh và phân phối không gian giữa các mục trong một thùng chứa (container), ngay cả khi kích thước của chúng không xác định hoặc kích thước động.
  - https://www.notion.so/Flexbox-686ee4ef94394e80b355b25425dbfa94

▼ **Ưu điểm của việc sử dụng CSS preprocessors(SASS, SCSS, LESS)?**

- CSS Preprocessors là ngôn ngữ tiền xử lý CSS. Là một ngôn ngữ kịch bản mở rộng của CSS và được biên dịch thành cú pháp CSS giúp bạn viết CSS nhanh hơn và có cấu trúc rõ ràng hơn. CSS Preprocessor có thể giúp bạn tiết kiệm thời gian viết CSS, dễ dàng bảo trì và phát triển CSS.
- Ưu điểm:
  - Có thêm phần mở rộng ngôn ngữ như các biến (variable), nesting, mixins...
  - Nhiều chức năng thao tác với màu sắc và các giá trị khác.
  - Có các đặc tính nâng cao kiểm soát thư viện.
  - Định dạng tốt và dễ tùy chỉnh.

▼ **Độ ưu tiên trong css?**

→ Xem bài tìm hiểu về độ cụ thể trong css

▼ **Hiểu gì về box-model?**

- Một hộp hình chữ nhật được bao quanh mọi phần tử HTML. Box model (mô hình hình hộp) được sử dụng để xác định chiều cao và chiều rộng của hình hộp chữ nhật. Hộp CSS bao gồm Chiều rộng và chiều cao (hoặc trong trường hợp không có, giá trị mặc định và nội dung bên trong), phần đệm (padding), đường viền (border), lề (margin):
  - **Content:** Nội dung thực tế của hộp nơi đặt văn bản hoặc hình ảnh.
  - **Padding**: Khu vực bao quanh nội dung (Khoảng trống giữa đường viền và nội dung).
  - **Border**: Khu vực bao quanh phần đệm.
  - **Margin**: Khu vực bao quanh đường viền.

▼ **Sự khác nhau giữa display: inline, block và inline-block**

- **inline**: các item sẽ nằm trên cùng 1 dòng. Nếu các items vượt quá độ dài của dòng thì item sẽ xuống dòng mới (vd: thẻ). Các inline item sẽ chỉ có thể điều chỉnh **margin** và **padding left and right** (top và bottom thì không thể).
- **block**: khác với inline thì block nó sẽ luôn xuống dòng và chiếm toàn bộ width nếu width không được set. Các item block sẽ set được width, height, margin, padding đầy đủ 4 hướng (top, bottom, left, right)

- **inline-block**: sẽ được sắp xếp giống kiểu inline nghĩa là các item sẽ được xếp trên 1 dòng nhưng sẽ có thuộc tính của block như là set width, height, margin, padding đủ 4 hướng

▼ **Responsive design khác adaptive design như thế nào?**

- RD: thích ứng với nhiều kích thước màn hình khác nhau và chỉ cần thiết kế một phiên bản cho web
- AD: Bố cục riêng biệt cho từng kích thước màn hình và phải thiết kế nhiều phiên bản cho web

▼ **Cách sử dụng position?**

- **Absolute:** Để đặt một phần tử chính xác nơi bạn muốn đặt nó. Absolute position thực sự được đặt so với phần tử gốc. Nếu không có trang gốc nào thì vị trí tương đối với chính trang đó (nó sẽ mặc định tất cả các cách sao lưu vào phần tử).
- **Relative**: Tương đối với chính nó. Vị trí đặt: tương đối; trên một phần tử và không có thuộc tính định vị nào khác, nó sẽ không ảnh hưởng đến định vị của nó. Nó cho phép sử dụng z-index trên phần tử và nó giới hạn phạm vi của các phần tử con được định vị tuyệt đối. Bất kỳ phần tử con nào sẽ được định vị tuyệt đối trong khối đó.
- **Fixed:** Phần tử được định vị liên quan đến chế độ xem hoặc chính cửa sổ trình duyệt. Khung nhìn không thay đổi nếu bạn cuộn và do đó phần tử cố định sẽ ở ngay vị trí cũ.
- **Static:** Mặc định tĩnh cho mọi phần tử trang. Lý do duy nhất bạn muốn đặt một phần tử ở vị trí: static là để xóa một cách một số vị trí đã áp dụng cho một phần tử nằm ngoài tầm kiểm soát của bạn.
- **Sticky**: Sticky positioning là sự kết hợp giữa định vị tương đối và cố định. Phần tử được coi là vị trí tương đối cho đến khi nó vượt qua ngưỡng được chỉ định, tại thời điểm đó, phần tử được coi là vị trí cố định.

▼ **Sự khác nhau của các đơn vị trong css: px, rem, em, vw, vh, ...**

- **Đơn vị tương đối (Relative Units)** là những đơn vị được tính một cách tương đối dựa trên các phần tử khác (có thể là phần tử cha hoặc phần tử root). Các đơn vị loại này khá linh động, rất thích hợp cho việc thích ứng trên các thiết bị khác nhau. Một vài đơn vị tương đối như: **rem, em, %, vw, vh, ex, ch, vmin, vmax**
- **Đơn vị tuyệt đối (Absolute Units)** là những đơn vị mà giá trị của nó không bao giờ thay đổi và không bị ảnh hưởng bởi các thành phần khác. Tức là trong mọi kích thước màn hình thì kích thước của nó vẫn như thế. Với loại đơn vị này chỉ nên dùng cho những thứ có kích thước cố định, hoặc kích thước nhỏ không quá ảnh hưởng như border, … Một vài đơn vị tuyệt đối như: **px, pt, cm, mm, pc, in**.
- **em** là đơn vị mà giá trị của nó được tính dựa trên tỉ lệ so với phần tử cha của nó hoặc chính nó thông qua giá trị của thuộc tính **font-size.** Mức độ ưu tiên sẽ tính theo font-size của nó trước, nếu nó không set thuộc font-size thì lấy của cha trực tiếp. Nếu cha nó không có thì lấy tiếp cha của cha nó Đến khi nào đến root thì thôi.
- **rem (root em)** tương tự như **em**, nhưng đơn giản là nó sẽ tỉ lệ theo thuộc tính font-size của phần tử **root <html>**
- **vw** sẽ tính theo tỉ lệ **chiều rộng khung nhìn** thiết bị của bạn. 1 vw = 1/100 chiều rộng view-port. Ví dụ: màn hình của bạn có chiều rộng 1100px thì 1vw = 11px
- **vh** tương tự vw, vh sẽ tỉ lệ theo chiều cao của khung nhìn thiết bị.
- **%** là đơn vị phần trăm sẽ tỉ lệ theo phần tử cha trực tiếp của nó.
- **Đơn vị vmin và vmax:** 2 đơn vị này tương tự như **vw và vh**. Điểm khác biệt là nó sẽ không tỉ lệ theo 1 hướng mà là cả 2, tuỳ thuộc vào độ lớn của chiều cao và chiều rộng màn hình. Cụ thể:
  - **1 vmin** = 1 vw hoặc 1 vh (Lấy cái nhỏ hơn). VD: màn hình của bạn có kích thước là 840×640 thì 1 vmin = 6.4px, nếu màn hình của bạn là 360×480 thì 1 vmin = 3.6px
  - **1 vmax** = 1 vw hoặc 1 vh (Lấy cái lớn hơn). VD: màn hình của bạn có kích thước là 840×640 thì 1 vmin = 8.4px, nếu màn hình của bạn là 360×480 thì 1 vmin = 4.8px

- **Các đơn vị tuyệt đối px, pt, cm, in, mm:** Với các đơn vị tuyệt đối thì giá trị của nó được cố định và không bị ảnh hưởng bởi bất kỳ thành phần nào khác, các đơn vị này chỉ nên sử dụng với những thành phần đã được xác định chính xác kích thước và không bị biến thiên bởi các thành phần khác. Hoặc các thuộc tính có kích thước nhỏ, không cần quá chính xác như `border: solid 1px;` . Thuộc tính hay được sử dụng nhất là px hoặc pt.

https://dynonguyen.com/tat-tan-tat-don-vi-trong-css/

▼ **JS**

https://completejavascript.com/javascript/

https://iq.js.org/questions/javascript/difference-between-undefined-and-null

▼ **Function programing**

- Functional Programming là phương pháp lập trình lấy function làm đơn vị thao tác cơ bản.
- Các nguyên tắc trong FP:
  - **Immutability**: tính bất biến. Cái nào đã khai báo một lần thì mãi mãi như vậy, không bao giờ thay đổi nữa.
  - **Purity**: tính thuần khiết, thuần tuý không bị pha tạp.
    - Tất cả các hàm đều phải là pure function, không có hiệu ứng phụ (side effect), không được tác động lên bất cứ giá trị nào bên ngoài nó, cũng nói không với chỉnh sửa tham số input.
    - Đặc điểm quan trọng nữa của pure function là với mỗi tập giá trị đầu vào nhất định, luôn có 1 và chỉ 1 kết quả trả về tương ứng. Đây là tính chất của hàm số toán học.
    - Pure function trong Functional Programming thường ngắn gọn, đơn giản và chỉ xử lý duy nhất 1 vấn đề logic.

```
// not pure function
const getDuration = (timestamp) => {
  return Date.now() - timestamp;
};

// pure function
const add = (a, b) => {
  return a + b;
};
```

- **Higher order function**: là một khái niệm đến từ Toán học. Bất cứ hàm nào tiếp nhận 1 function như tham số, hoặc trả về 1 function như kết quả, thì đều được coi là higher-order function.
- **Function Composition:** Đây là khái niệm Toán học mà tiếng Việt ta gọi là "hàm hợp", hay "hàm phức hợp". Mọi thứ trong Functional Programming đều có nguồn gốc Toán học.
  - Function Composition là sự phối hợp, liên kết nhiều hàm lại với nhau, thành một hàm lớn, nhiều chức năng hơn.
  - Hoặc dễ hiểu đây là cách chúng ta kết hợp nhiều function lại với nhau. Kết quả của function này sẽ được sử dụng cho function tiếp theo. Cứ như vậy nó sẽ tạo thành một chuỗi các function thực hiện các nhiệm vụ theo thứ tự.
  - Có 2 kỹ thuật căn bản trong Function Composition là `compose` và `pipe`.
    - **compose**: trong toán học cơ bản với $y = f(g(x))$ ta tính g(x) trước rồi truyền cho f() là ra kết quả, từ phải sang trái. Ý tưởng của `compose`
      là xếp cuốn chiếu các hàm lại với nhau, theo thứ tự từ trái sang phải để tạo ra một hàm mới, mà khi được thực thi, nó sẽ lần lượt gọi các hàm đã truyền vào trước đó theo thứ tự ngược lại, từ phải sang trái (`Right to left`)

        - **Pipe:** Một biến thể của `compose` là `pipe`, vận hành theo chiều ngược lại. Ta có thể implement bằng cách đảo vị trí `f` và `g` (`Left to right`)
    - **Currying function:** là một HOF, biến đổi 1 function n tham số thành n hàm nhận vào duy nhất 1 tham số

▼ **Reactive programing**

- Có thể giới thiệu ngắn ngọn Reactive = Asynchronous + Non-Blocking I/O (NIO), có nghĩa là một chương trình được gọi là Reactive nó sẽ đảm bảo được 2 yếu tố là Asynchronous (xử lý bất đồng bộ) và Non-Blocking I/O.
- Bằng cách viết những đoạn mã asynchronous và non-blocking, chương trình sẽ cho phép switch qua các tách vụ khác mà đang sử dụng cùng một I/O resource, và có thể quay lại sử lý tiếp khi tác vụ đó hoàn thành. Do đó với reactive programing chương trình có thể sử lý nhiều request hơn trên cùng một tài nguyên hệ thống.
- Reactive và non-blocking nhìn chung thì không làm cho ứng dụng chạy nhanh hơn. Lợi ích mà nó được kỳ vọng là ứng dụng chịu tải được tốt hơn mà chỉ yêu cầu ít tài nguyên hơn.

▼ **Hoisting & Javascript hoạt động như thế nào**

- Hoisting là một hành vi mặc định của Javascript di chuyển việc khai báo lên đầu trong scope của nó.
- Tuy nhiên đây là một quan niệm chưa đúng về hoisting, Chúng ta nên xem cách JS hoạt động như thế nào:
    - 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:
        - Giai đoạn 1 (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.**
        - Giai đoạn 2 (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

https://nodejs.vn/user/dũng-vũ-0

▼ **Temporal dead zone**

- 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.

▼ **Scope**

Scope là 1 khái niệm nhằm xác định phạm vi hoạt động của biến.

- Có 2 loại scope:

- **Global Scope,**
  - Biến được định nghĩa bên ngoài function sẽ thuộc về *Global Scope*
  - Biến thuộc Global Scope có thể được sử dụng và thay thế ở bất cứ đâu
  - Nếu biến được định nghĩa mà không sử dụng từ khóa khai báo thì được mặc định thuộc vào Global Scope
- **Local Scope**
  - Biến được định nghĩa bên trong 1 function sẽ thuộc về *Local Scope*
  - Mỗi function khi gọi sẽ tạo ra 1 scope mới.
  - Các function khác nhau sẽ tạo ra các scope khác nhau.
  - Điều này dẫn đến có thể khai báo biến trùng tên ở các function khác nhau.
- **Block Statements**
  - *Block statements* như điều kiện if và switch hay vòng lặp for và while, không giống như function chúng không tạo ra 1 scope mới.
  - Biến được định nghĩa bên trong *Block statements* sẽ vẫn giữ nguyên scope của nó.
  - Javascript ES6 giới thiệu 2 từ khóa khai báo biến là let và const.
  - Trái ngược với var, let và const hỗ trợ tạo ra local scope bên trong Block statements

▼ **spread operator, rest parameters, destructuring**

▼ **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
- Có hai loại Destructuring: **Destructuring Objects** và **Destructuring Arrays**

```
const info = { id: 1, name: 'Tee', gender: 'male'}
// Destructure properties into variables
const { id, name, gender } = info
```

```
const date = ['10', '01', '1998']
// Destructure Array values into variables
const [day, month, year] = date
```

▼ **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.
- Cũng giống như **destructuring** thì Spread cũng làm việc nhiều với Arrays và Objects

```
// Create an Array
const tools = ['hammer', 'screwdriver']
const otherTools = ['wrench', 'saw']

// Concatenate tools and otherTools together
const allTools = tools.concat(otherTools)

// Unpack the tools Array into the allTools Array
const allTools = [...tools, ...otherTools]
```

```
// Create a function to multiply three items
function multiply(a, b, c) {
  return a * b * c
}

multiply(1, 2, 3) ;// 6

// Or

const numbers = [1, 2, 3]
multiply(...numbers);//6
```

▼ **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
- Giống như Spread Syntax (...) nhưng có tác dụng ngược lại

```
function restTest(...args) {
  console.log(args)
}

restTest(1, 2, 3, 4, 5, 6);// [1, 2, 3, 4, 5, 6]
```

▼ **Deep copy ≠ Shallow copy**

link

▼ **Map ≠ SET ≠ WeakMap ≠ WeakSet**

▼ **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`.

https://completejavascript.com/so-sanh-map-voi-object-trong-javascript/

▼ **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`.

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

▼ **map(), filter(), reduce()**

- **map**: để thực hiện một chức năng trên mỗi phần 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

link

▼ **cookie, session, localStorage**

link

▼ **async/await ≠ promise ≠ callback**

https://completejavascript.com/xu-ly-bat-dong-bo-callback-promise-async-await/

https://completejavascript.com/ket-thuc-som-promise-chaining-trong-javascript/

https://completejavascript.com/xu-ly-bat-dong-bo-song-song-tuan-tu/

link

▼ HOF, Currying Fn

link

▼ Closure Fn

- Closure là tạo một phạm vi từ vựng đóng bên trong một phạm vi khác, vì vậy phạm vi bên trong có thể truy cập phạm vi bên ngoài. Closure được tạo khi hàm được tạo. Closure là làm cho các biến và phương thức ở chế độ riêng tư trong phạm vi.
- Currying là một ví dụ của việc closure. Nó thường tự trả về khi tạo ra lexical environment. Môi trường này bao gồm bất kỳ biến cục bộ nào nằm trong phạm vi tại thời điểm closure được tạo. Nó giống như một nhà máy nhỏ để sản xuất một sản phẩm với các chức năng cụ thể từ các thành phần đó.
- Không có cách rõ ràng nào để tạo các phương thức private trong JS, nhưng closure có thể 'private' các phương thức.

https://blog.daovanhung.com/post/scope-closure-this-va-to-chuc-bo-nho-trong-javascript

---

- Closure là một hàm có thể ghi nhớ nơi nó được tạo ra và truy cập được các biến ở bên ngoài phạm vi của nó
- Ứng dụng:
  - Giúp code viết ngắn gọn hơn
  - Biểu diễn, ứng dụng tính private trong OOP
- Lưu ý:
  - Biến được tham chiếu (refer) trong closure sẽ không được xóa khỏi bộ nhớ khi hàm cha thực thi xong
  - Các khái niệm JS nâng cao rất dễ gây nhầm lẫn

link

▼ This, call, apply, bind

link

https://completejavascript.com/phan-biet-call-apply-va-bind-trong-javascript/

- Nhìn chung, hàm `call` và `apply` là gần giống nhau. Chúng đều gọi hàm trực tiếp. Chỉ khác ở cách truyền tham số vào (với **c**all thì đối số phân cách bởi dấu phẩy **c**omma và với **a**pply thì đối số cho bởi mảng **a**rray)
- Hàm `bind` thì hơi khác hơn một chút. Hàm này không gọi hàm trực tiếp mà trả về một hàm mới. Sau đó, bạn có thể sử dụng hàm mới này. Về cách truyền tham số vào thì hàm `bind` giống với hàm `call`.

▼ Prototype

- 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.
- https://completejavascript.com/prototype-la-gi-prototype-trong-javascript/

▼ **Side effect?**

- 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

▼ **Hãy giải thích về event delegation**

- Event delegation là một kỹ thuật liên quan đến việc thêm event listeners vào một phần tử mẹ thay vì thêm chúng vào các phần tử con.
- Listeners sẽ kích hoạt bất cứ khi nào sự kiện được kích hoạt trên các phần tử con do sự kiện làm nổi bong bóng (bubbling up) DOM.
- Lợi ích của kỹ thuật này là:
  - Memory footprint giảm xuống vì chỉ cần một trình xử lý duy nhất trên phần tử mẹ, thay vì phải đính kèm các trình xử lý sự kiện trên mỗi phần tử con.
  - Không cần phải hủy liên kết trình xử lý khỏi các phần tử bị xóa và liên kết sự kiện cho các phần tử mới.

▼ **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.

▼ Sự khác biệt giữa các host objects và native objects là gì?

- Native objects là các đối tượng là một phần của ngôn ngữ JavaScript được xác định bởi đặc tả ECMAScript, chẳng hạn như `String`, `Math`, `RegExp`, `Object`, `Function`, v.v.
- Host objects được cung cấp bởi môi trường thời gian chạy (trình duyệt hoặc Node), chẳng hạn như `window`, `XMLHTTPRequest`, v.v.

▼ Sự khác biệt giữa `let`, `const` và `var` là gì?

Ban đầu, `var` là tùy chọn duy nhất mà JavaScript có để xác định các biến. Trong ES6, chúng ta có **const** và `let` là các tùy chọn bổ sung.

- Các biến được xác định bằng `const` không thể được gán lại.
- Các biến `Const` và `let` là phạm vi khối.
- Biến `Var` là function scoped.
- Các biến được xác định bằng `var` được sử dụng theo cơ chế hoisting

▼ Làm cách nào để bạn kiểm tra xem một biến có phải là một số trong JavaScript hay không?

- Để kiểm tra xem một biến có phải là số hay không, chúng ta có thể sử dụng hàm `isNaN ()` trong JavaScript. Nó xác định xem một giá trị có phải là một số hay không.

▼ 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.

▼ Javascript Proxy

- **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.
- Cú pháp:

```
const p = new Proxy(target, handler);
```

  - Trong đó
    - **target**: là đối tượng sẽ được áp dụng proxy vào
    - **traps**: là những phương thức giúp bạn thay đổi hành vi của đối tượng
    - **handler**: là một object chứa các traps, được đưa vào hàm dựng của lớp Proxy
  - Một số **traps** thông dụng:
    - `handler.get()`
    - `handler.set()`
    - `handler.defineProperty()`
    - `handler.deleteProperty()`
    - `handler.has()`
    - `handler.apply()`
    - `handler.construct()`
- Ví dụ:

```
const u = { name: 'Công Tằng Tôn Nữ Tạ Thị Tòn Ten' }

// Thiết lập proxy cho đối tượng `u`
const p = new Proxy(u, {
  // `get` là một trap, sẽ được gọi khi truy xuất đến thuộc tính
  // của đối tượng
  get(target, prop, receiver) {
    // Thay đổi hành vi khi truy xuất đến một thuộc tính: Nếu là
    // chuỗi, chuyển sang chữ hoa
    if (typeof target[prop] === 'string') return target[prop].toUpperCase()

    return target[prop]
  },
})

console.log(p.name) // CÔNG TẰNG TÔN NỮ TẠ THỊ TÒN TEN
p.email = 'ta.thi@ton.ten'
console.log(p.email) // TA.THI@TON.TEN
```

- https://ehkoo.com/bai-viet/tim-hieu-ve-proxy-trong-es6

▼ Event loop, message queue

- https://viblo.asia/p/cuoi-cung-thi-event-loop-la-gi-LzD5dX705jY

- https://dynonguyen.com/event-loop-bat-dong-bo-trong-javascript/

▼ **factories ≠ class**

- Một factory function là bất kỳ hàm nào mà nó không phải là một class, một constructor trả về một object.
- Trong Javascript bất kỳ hàm nào cũng có thể trả về một một object. Khi thực hiện điều đó mà nó không dùng từ khóa `new` thì đó là **factory function.** Ngược lại thì là **constructor function**
- https://medium.com/@jeffrey.k.vy/factory-function-và-constructor-function-trong-javascript-d80292d7255

▼ **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()**

▼ **web worker**

- JavaScript là một ngôn ngữ chạy đơn luồng. Điều đó có nghĩa là nếu bạn thực hiện một tác vụ quá lớn trên giao diện chính thì khả năng cao là giao diện sẽ bị đơ.
- Để giải quyết vấn đề này, JavaScript đã đưa ra một khái niệm là Worker
- Web Worker không phải của Javascript, mà đây là một tính năng của trình duyệt cho phép chúng ta truy xuất qua Javascript

---

- Web Worker là một đối tượng trong Javascript được tạo ra bởi các hàm constructor như `Worker`, `SharedWorker`... với tham số được truyền vào là một file JS chứa các đoạn code sẽ được thực thi bởi Worker. Các script được viết trong file này sẽ được thực thi ngầm, không ảnh hưởng đến trải nghiệm của người dùng. Vì vậy với các tác vụ tốn nhiều thời gian các bạn có thể dùng Worker để xử lý.
- Web Workers là tiến trình trong trình duyệt nhưng có thể được dùng để thực thi Javascript code mà không cản trở event loop
- Web Workers cho phép developer đặt những công việc có thời gian chạy dài và những công việc nặng về xử lý tính toán trong background mà không gây trở ngại đến UI, làm app của bạn mượt mà hơn. Ngoài ra, không cần phải xài trick với `setTimeout` để đánh lừa event loop nữa.
- Web Workers cho phép bạn làm những việc như thực thi các đoạn code xử lý tốn thời gian để tính toán các phép tính hao tổn nhiều CPU nhưng không làm cản trở UI. Thực ra, nó sẽ chạy song song. Web Workers là đa luồng.
- Có 3 loại Web Workers:
  - Dedicated worker
  - Shared worker
  - Service worker

https://kipalog.com/posts/Duc-khoet-Javascript--Phan-7---Thanh-phan-cua-WebWorker---5-truong-hop-su-dung

https://2kvn.com/web-worker-anh-cong-nhan-tham-lang-trong-ung-dung-cua-ban-p5f33313835

https://completejavascript.com/javascript-web-worker-javascript-o-background/

▼ **TS**

1. Interface ≠ type
2. Generic
3. Class

▼ **Backend**

▼ **Microservice ≠ Monolithic**

▼ **Monolithic**

- là kiến trúc phần mềm dạng nguyên khối, nghĩa là mọi tính năng sẽ nằm trong một project.
- Giả sử mình có một project web bán hàng triển khai theo kiến trúc monolithic, thì các module như khách hàng, đơn hàng, sản phẩm,… sẽ được gói gọn trong project đó.
- **Ưu điểm:**
  - Dễ phát triển vì các stack công nghệ thống nhất ở tất cả các layer.
  - Dễ test do toàn bộ project được đóng gói trong một package nên dễ dàng chạy test integrantion và test end-to-end.
  - Deploy đơn giản và nhanh chóng nếu bạn chỉ có một package để bận tâm.
  - Dễ scale vì chúng ta có thể có nhiều instance cho load banlancer.
  - Yêu cầu team size nhỏ cho việc maintain app.
  - Team member có thể chia sẻ ít nhiều về skill.
  - Tech stack đơn giản và đa số là dễ học.
  - Phát triển ban đầu nhanh hơn do đó có thể đem sale hoặc marketing nhanh hơn.
  - Yêu cầu cơ sở hạ tầng đơn giản. Thậm chí một container đơn giản cũng đủ để chạy ứng dụng.
- **Nhược điểm:**
  - Các component được liên kết chặt chẽ với nhau dẫn đến side effect không mong muốn như khi thay đổi một component ảnh hưởng đến một component khác.
  - Theo thời gian thì project trở nên phức tạp và lớn dần. Các tính năng mới sẽ mất nhiều thời gian hơn để phát triển và tái cấu trúc các tính năng hiện có sẽ nhiều khó khăn hơn.
  - Toàn bộ ứng dụng cần được triển khai lại cho bất kỳ thay đổi nào.
  - 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.
  - Áp dụng công nghệ mới khó khăn vì toàn bộ ứng dụng phải thay đổi. Do đó nhiều ứng dụng một khối thường phụ thuộc một công nghệ cũ và lỗi thời.
  - Các service quan trọng không thể scale riêng dẫn đến lãng phí tài nguyên vì toàn bộ ứng dụng phải scale theo.
  - Các ứng dụng một khối lớn sẽ 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 chia dự án thành nhiều service nhỏ
- Các service trong kiến trúc microservice là độc lập với nhau, chúng có thể có kiến trúc khác nhau, sử dụng công nghệ khác nhau hoặc thậm chí có database riêng

- Chúng trao đổi thông tin với nhau thông qua môi trường mạng (có thể bằng end point Restful API hoặc các 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.

▼ **Rest ≠ graph**

- https://2kvn.com/graphql-vs-rest-apis-p5f31353330
- https://viblo.asia/p/so-sanh-graphql-voi-rest-V3m5WLv8KO7
- https://codelearn.io/sharing/graphql-va-uu-diem-so-voi-rest-api

▼ **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.
- **Cơ chế hoạt động:**
  - Trong trường hợp đơn giản nhất, phía client (tức là cái web app chạy ở browser đó) sẽ tạo request GET, POST, PUT, HEAD, etc để yêu cầu server làm một việc gì đó.

    - Những request này sẽ được đính kèm một header tên là `Origin` để chỉ định origin của client code (giá trị của header này chính là domain của trang web).
    - Server sẽ xem xét `Origin` để biết được nguồn này có phải là nguồn hợp lệ hay không.
    - Nếu hợp lệ, server sẽ trả về response kèm với header `Access-Control-Allow-Origin`.
    - Header này sẽ cho biết xem client có phải là nguồn hợp lệ để browser tiếp tục thực hiện quá trình request.
    - Trong trường hợp thông thường, `Access-Control-Allow-Origin` sẽ có giá trị giống như `Origin`, một số trường hợp giá trị của `Access-Control-Allow-Origin` sẽ nhìn giống giống như Regex hay chỉ đơn giản là `*`
    - Tuy nhiên thì cách dùng `*` thường được coi là không an toàn, ngoại trừ trường hợp API của bạn được public hoàn toàn và ai cũng có thể truy cập được.
    - Nếu không có header `Access-Control-Allow-Origin` hoặc giá trị của nó không hợp lệ thì browser sẽ báo lỗi.
- **Pre-flight request:**
    - Khi thực hiện những request ảnh hưởng đến data như: POST, PUT, DELETE, ... thì browser sẽ tự động thực hiện một request gọi là `preflight request` trước khi thực sự thực hiện request để kiểm tra xem phía server đã thực hiện CORS hay chưa, cũng như để biết được rằng request này có hợp lệ hay không.
    - Ngoài ra thì nếu bạn có thêm những custom header vào trong request thì việc gửi một `preflight request` cũng là cần thiết.
    - Preflight request được gửi lên server với dạng là `OPTIONS` (đây là lý do tại sao khi bạn debug ở client bạn thường thấy có hai request giống nhau nhưng khác request method, một cái là `OPTIONS` một cái là method thật sự bạn muốn gửi).
    - **Ví dụ:** bạn muốn gửi request `DELETE` lên server.
        - Browser sẽ tự tạo một request `OPTIONS` sẽ hỏi xem server có cho phép việc gửi request `DELETE` hay không.
        - Nếu server cho phép, nó sẽ gửi về response đính kèm những header như `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods`, `Access-Control-Max-Age`, etc.
            - Access-Control-Allow-Methods: mô tả những method nào client có thể gửi đi.
            - Access-Control-Max-Age: mô tả thời gian hợp lệ của `preflight request`, nếu quá hạn, browser sẽ tự tạo một `preflight request` mới.
        - Sau đó browser sẽ có thể gửi request `DELETE` và nhận response như bình thường. Và ngược lại, browser sẽ báo lỗi

▼ Middleware

- Middleware đóng vai trò trung gian giữa **request/response** (tương tác với người dùng) và các xử lý logic bên trong web server.
- Middleware sẽ là các hàm được dùng để tiền xử lý, lọc các request trước khi đưa vào xử lý logic hoặc điều chỉnh các response trước khi gửi về cho người dùng.
- Các hàm middleware có thể thực thi ở đầu, giữa, hoặc cuối vòng đời của một request. Trong stack các middleware function luôn được thêm vào theo thứ tự mà chúng ta mong muốn ngay ban đầu.
- Một số middleware phổ biến trong nodejs như: route, cors, auth, logger, helmet, ...

▼ Helmet

- Helmet là một package được viết để giúp bạn bảo vệ ứng dụng của mình khỏi những lỗ hổng đã biết bằng cách thiết lập các Http headers một cách phù hợp.

- Thực tế thì Helmet chỉ là một tập hợp các Middleware nhỏ làm nhiệm vụ thiết lập các Http headers liên quan đến bảo mật.
- Nó giúp ẩn đi những thông tin không cần thiết trong header và thiết lập lại một số thuộc tính giúp cho server an toàn hơn.

▼ **Cookies**

- **Cookie** là những tập tin một trang web gửi đến máy người dùng và được lưu lại thông qua trình duyệt khi người dùng truy cập trang web đó.
- Cookie được dùng để lưu trữ với rất nhiều mục đích như lưu phiên đăng nhập, hoạt động của người dùng khi truy cập trang web.
- **Cookie** có nhiều loại khác nhau và phân chia theo từng mục đích sử dụng. Một số cookies phổ biến như:
  - **Session Cookie:** chỉ tồn tại tạm thời trong bộ nhớ của trình duyệt và sẽ bị trình duyệt tự xóa khi người dùng hết phiên đăng nhập, thông thường loại cookie này không có thời hạn.
  - **Third-party cookie :** thông thường cookie của trang web sẽ trùng với thanh địa chỉ của trình duyệt nhưng có một vài trường hợp sử dụng cookie bên thứ 3 có tên miền khác với url trang web
  - **Secure cookie:** một loại cookie HTTP có bộ thuộc tính secure giới hạn phạm vi của cookie đối với trình duyệt web.

▼ **Sessions**

- Session dùng để lưu trữ dữ liệu trên Server và đồng thời nó sẽ có một đoạn code dữ liệu được lưu trữ ở client (cookie).
- Một số cách lưu trữ sessions ở server:
  - **Cookie**: Chúng ta có thể store sesion trên cookie session nodejs mỗi trình duyệt nhưng chú ý rằng tất cả đều nằm ở Clients.
  - **Memory Cache**: Như chúng ta đã biết, Cache được lưu trữ trong bộ nhớ. Chúng ta cũng có thể sử dụng thêm những cache module như Redis và Memcached.
  - **Database**: (ít được sử dụng)

▼ **WebSocket**

- **WebSocket** là công nghệ hỗ trợ giao tiếp hai chiều giữa client và server bằng cách sử dụng một TCP socket để tạo một kết nối hiệu quả và ít tốn kém.
- Mặc dù được thiết kế để chuyên sử dụng cho các ứng dụng web, lập trình viên vẫn có thể đưa chúng vào bất kì loại ứng dụng nào

▼ **Cache**

- Khi cache client cần truy cập data, việc đầu tiên là check cache. Khi request data tìm thấy dữ liệu cần thiết trong Cache, nó được gọi là ***Cache hit***. Tỉ lệ của kết quả tìm kiếm **cache hit** được biết đến như là *cache hit rate* hay *ratio*.
- Nếu việc tìm kiếm data không thành công, nó gọi là ***Cache miss*** - từ đây dữ liệu sẽ được kéo từ bộ nhớ chính sang bộ nhớ cache. Việc giữ dữ liệu nào cần, hay xoá khỏi bộ nhớ điệm để nhường chỗ cho dữ liệu mới sẽ tùy thuộc vào thuật toán mà system sử dụng.
- **Cache Replacement Policy** nghĩa nôm na là các thuật toán để thay thế giá trị hoặc xóa các giá trị cũ để thêm giá trị mới vào.
  - LRU, LFU, MRU, FIFO, LIFO, ...

- **Memoization** là một kỹ thuật tối ưu hóa, giúp tăng tốc các ứng dụng bằng cách lưu trữ kết quả của các lệnh gọi hàm (mà các hàm này được gọi là `expensive function`) và trả về kết quả được lưu trong bộ nhớ cache khi có cùng một đầu vào yêu cầu (đã được thực thi ít nhất 1 lần trước đó rồi).
- Một số thuật toán điều khiển cache:
  - **Least Frequently Used (LFU)** : theo dõi tần suất truy cập một dữ liệu. Các dữ liệu có số lần truy cập thấp nhất được loại bỏ đầu tiên.
  - **Least Recently Used (LRU)** : lưu trữ các dữ liệu được truy cập gần đây gần đầu bộ đệm. Khi bộ đệm đạt đến giới hạn của nó, các liệu được truy cập gần đây nhất sẽ bị xóa.
  - **Most Recently Used (MRU)** : loại bỏ các dữ liệu truy cập gần đây nhất đầu tiên. Cách tiếp cận này là tốt nhất khi các data cũ có nhiều khả năng được sử dụng.
- Các loại cache hay sử dụng:
  - **cache server** : Một dedicated network server hoặc dịch vụ chuyên dụng hoạt động như một máy chủ lưu các trang web hoặc nội dung internet khác cục bộ. Một cache server đôi khi được gọi là proxy cache.
  - **Cache memory**: Random access memory, hay còn được gọi là **RAM**, Cache memory thường được gắn trực tiếp vào CPU và được sử dụng để cập nhanh các dữ liệu trong CPU.
  - **Flash cache**: Temporary storage of data on NAND flash memory chips -- thường được sử dụng ở **solid-state drives (SSDs) ** , thực hiện các request dữ liệu nhanh hơn có thể nếu bộ đệm nằm trên ổ đĩa cứng truyền thống (HDD)

  https://pymi.vn/blog/memoization/

▼ **Stream, Buffer, Pipe**

- **Stream**: là các đối tượng cho phép bạn đọc dữ liệu từ một nguồn và ghi dự liệu đến một đích (*là một chuỗi dữ liệu sẵn có qua thời gian, hay có thể hình dung stream là một đối tượng chứa dữ liệu sẽ được truyền đi từ nơi này đến nơi khác.*)
  - Có 4 loại Stream trong Nodejs:
    - **Readable**: Là Stream được sử dụng để cho hoạt động đọc
    - **Writable**: Là Stream được sử dụng cho hoạt động ghi
    - **Duplex**: Là Stream được sử dụng cho cả mục đích ghi và đọc
    - **Transform**: Đây là một kiểu Duplex Stream, khác ở chỗ là kết quả đầu ra được tính toán dựa trên dữ liệu bạn đã nhập vào.
  - Mỗi loại Stream là một sự thể hiện của đối tượng **EventEmitter** và ném một vài sự kiện tại các thời điểm khác nhau
  - **Piping Stream:** là một kỹ thuật. Với kỹ thuật này, chúng ta cung cấp kết quả đầu ra của một Stream để làm dữ liệu đầu vào cho một Stream khác. Không có giới hạn nào về hoạt động Piping này, tức là quá trình trên có thể vẫn tiếp tục.
  - **Chaining Stream:** là một kỹ thuật để kết nối kết quả đầu ra của một Stream tới một Stream khác và tạo một chuỗi bao gồm nhiều hoạt động Stream. Thường thì nó được sử dụng với các hoạt động Piping
- **Buffer**: là một vùng dự trữ tạm thời chứa các dữ liệu đang được chuyển từ nơi này đến nơi khác.
  - Buffer có kích thước xác định và giới hạn.
  - Kích thước của buffer được xác định bằng những thuật toán cho từng trường hợp cụ thể.
  - Buffer là một kỹ thuật được phát triển nhằm ngăn chặn sự tắc nghẽn dữ liệu khi truyền từ nơi này đến nơi khác.

- Ví dụ:
  - Trong thực tế, khi chúng ta xem một đoạn phim trên mạng.
  - Nếu đường truyền mạng chúng ta đủ nhanh thì tốc độ stream video sẽ kịp thời làm đầy các buffer (vùng nhớ tạm thời trên RAM) và đoạn dữ liệu này sẽ được gửi đến trình media player để chạy đoạn dữ liệu vừa được làm đầy trong buffer.
  - Trong lúc phát nội dung đó, buffer sẽ trống và lại được làm đầy.
  - Cứ như vậy cho đến khi kết thúc stream.

▼ **Queue**

- **Khái niệm**:
  - Message Queue nôm na là Queue (hàng đợi), chứa Message (Tin nhắn) như hộp thư.
  - Và nó cho phép các thành phần/service trong một hệ thống (hoặc nhiều hệ thống), trao đổi thông tin cho nhau.
  - Ý nghĩa của queue (hàng đợi) là nó thực hiện việc lấy message theo cơ chế vào trước thì ra trước (First In First Out).
- **Thành phần của 1 MQ:**
  - **Message**: Thông tin được gửi đi (có thể là text, binary hoặc JSON)
  - **Message Queue**: Nơi chứa những message này, cho phép producer và consumer có thể trao đổi với nhau
  - **Producer**: Chương trình/service tạo ra thông tin, đưa thông tin vào message queue
  - **Consumer**: Chương trình/service nhận message từ message queue và xử lý
  - Một chương trình/service có thể **vừa là producer, vừa là consumer**
- **MQ giải quyết được các vấn đề:**
  - **Đảm bảo duration/recovery**: Do message đã được lưu trong queue, khi 1 service đang xử lý nhưng bị crash hoặc lỗi, ta không lo bị mất dữ liệu.
    - Vì có thể lấy message từ trong queue ra và chạy lại.
    - Trong 1 hệ thống có nhiều consumer, nếu 1, 2 consume bị crash cũng không làm sụp toàn hệ thống
  - **Phân tách hệ thống**: Giúp phân tách hệ thống thành nhiều service nhỏ hơn, mỗi service chỉ xử lý 1 chức năng nhất định
  - **Hỗ trợ rate limit, batching**: Trong nhiều trường hợp, năng lực xử lý hệ thống có hạn (chỉ có thể xử lý 300 đơn hàng/s).
    - Với message queue, ta có thể dần dần lấy đơn hàng trong queue ra xử lý, không sợ thất lại.
    - Hoặc thay vì mỗi lần gửi email mất thời gian lâu, ta có thể đợi message queue có yêu cầu gửi 200 email rồi gửi luôn 1 lượt.
  - **Dễ scaling hệ thống**: Vào giờ cao điểm, nhiều truy vấn, ta có thể tăng số lượng consumer lên để xử lý được nhiều messege hơn. Khi không cần ta có thể giảm lại.
- **Một số MQ phổ biến:**
  - **Kafka**
  - **RabittMQ**
  - **Bull**

- ...

▼ Single sign on

- **SSO** là viết tắt của Single Sign On, có nghĩa là bạn có thể đăng nhập vào một hệ thống trong một nhóm ứng dụng đa hệ thống (google, youtube, console...) và bạn có thể nhận ủy quyền trong tất cả các hệ thống khác mà không cần đăng nhập lại.
- *SSO là cơ chế cho phép người dùng có thể truy cập nhiều trang web, ứng dụng mà chỉ cần đăng nhập một lần. Một khi đã được định danh ở một trang website A, thì cũng sẽ được định danh tương tự ở website B mà không cần lặp lại thao tác đăng nhập*
- Ưu điểm:
  - Giảm số lượng username và pasword mà người dùng cần phải ghi nhớ
  - Giảm số lần phải nhập thông tin username và pasword
  - Rủi ro về việc lộ thông tin người dùng cũng được tiết chế lại
- Nhược điểm:
  - Chi phí phát triển khi thông qua service thứ ba
  - Phụ thuộc vào service bên ngoài
- SSO là một phần của hệ thống nhận dạng liên kết (**Federated Identity Glossary**), có liên quan chặt chẽ với việc xác thực thông tin người dùng. Nó sẽ định danh người dùng, và sau đó chia sẻ thông tin định danh được với các hệ thống con.
  - Hệ thông nhận dang liên kết là nơi tập trung và liên kết thông tin người dùng. Có 4 yếu tố nền tảng cấu thành:
    - Xác thực (Authentication)
    - Phân quyền (Authorization)
    - Trao đổi thông tin người dùng (User attributes exchange)
    - Quản lí người dùng (User management)
- **Cơ chế:**
  - https://www.google.com/search?q=single+sign+on+la+gi&oq=single&aqs=chrome.0.69i59j69i57j69i60l3.1218j0j1&sourceid=chrome&ie=UTF-8

▼ JWT, auth

- link

▼ Single thread, multiple thread

- https://viblo.asia/p/javascript-single-thread-lieu-da-loi-thoi-gAm5yxwkldb
- **Lấy một ví dụ để minh họa JS chạy trên môi trường trình duyệt web chrome là single thread?**

```
let loop = true;

setTimeout(() => {  // callback execution is unreachable because main thread is still busy with below infinite loop
 code.
    console.log('It will never reach here :ohh');
    loop = false;
});

while (loop) {
    console.log('loop', loop);  // Infinite loop
}
```

```
console.log('after loop');  // Unreachable code
```

- *Single thread theo hướng event-driven nó vậy đó fence, các hoạt động i/o như gọi API thì js nó không tự tạo thread để xử lý mà bắn cho thằng khác(trình duyệt) lo còn bản thân js nó ngồi chơi xơi nước đợi kết quả không à. Ngược lại nếu gọi 1 hàm đệ quy tính toán nặng trong main thread(js tự thân vận động) thì web nó treo đến khi hàm đó chạy xong thì thôi. Đấy mới chính là cái khẳng định js trên browser đơn thuần là single thread. Còn việc có thể tạo lập các service worker thì nó là runtime của browser, browser có hỗ trợ thì dùng, không thì thôi nên không thể nói nó là của js đc.*
-
-

▼ **pub & sub system**

- **Publish/subscribe messaging** là một pattern mà đặc trưng bởi việc gửi (publisher) data (message) mà ko chỉ định người nhận rõ ràng .
- Thay vào đó người gửi sẽ phân loại tin nhắn thành các lớp và bằng cách nào đó mà người nhận (subscriber) phải đăng kí vào lớp nhất định để nhận tin nhắn ở lớp đấy.
- Hệ thống pub/sub thường có một broker (nhà phân phối) , là trung tâm nơi mà các message được phân phối.

▼ **ACL, RBAC**

▼ **ACL**: là hình thức phân quyền dựa trên một danh sách các quyền truy cập.

- `Subject` có thể `Action` tới `Object`
- Dựa vào người dùng và nhóm người dùng
- **Ví dụ**:
  - Cho phép Nguyễn Văn A tạo bài viết

```
'Subject' : 'Nguyễn Văn A'
'Action' : 'Tạo'
'Đối tượng' : 'Bài viết'
```

  - Nguyễn Văn A có thể tạo bài viết..
  - Ví dụ : Phân quyền trong `MySql`.

▼ **RBAC**: là hình thức phân quyền dựa vào vai, mỗi Subject sẽ thuộc một hoặc nhiều Role. Mỗi Role lại có một hoặc nhiều Permission thực thi Action tới Object.

- Một `Subject` thuộc một hoặc nhiều `Role`
- Một `Role` có một hoặc nhiều `Permission`.
- **Ví dụ**:
  - Người dùng Nguyễn Văn A có quyền Admin, User.
  - Người dùng Lê Văn B có quyền User.
  - User có quyền `Đọc` bài viết.
  - Admin có quyền `Đọc` , `Thêm` . `Sửa` , `Xóa` bài viết.

  ⇒

  - Người dùng A có quyền `Đọc` , `Thêm` . `Sửa` , `Xóa` bài viết.

- Người dùng Lê Văn B có quyền `Đọc` bài viết.

https://www.phamduytung.com/blog/2021-07-02-mo-hinh-phan-quyen/

https://kipalog.com/posts/Bai-toan-phan-quyen-van-de-muon-thuo-va-rat-kho-hieu

▼ **Một số cách secure trong nodejs**

- Validate user input to limit SQL injections and XSS attacks
- Implement strong authentication
- Avoid errors that reveal too much
- Run automatic vulnerability scanning
- Avoid data leaks
- Set up logging and monitoring
- Use security linters (eslint)
- Avoid secrets in config files
- Implement HTTP response headers
- Don't run Node.js as root
- Protect and observe your Node.js apps in production

https://blog.sqreen.com/nodejs-security-best-practices/

- Chống DOS, DDOS hay brute-force mật khẩu
- Lọc dữ liệu người dùng gửi lên server
- Sử dụng helmet
- Sử dụng `express-validator` để data gửi lên server
- Sử dụng thư viện ORD/ODM để chống SQL/NoSQL Injection
- Sử dụng https
- Sử dụng biến môi trường
- Dùng **bcrypt** hoặc **pbkdf2** để băm mật khẩu
- Giới hạn kích thước payload request gửi lên server

https://hocweb.vn/toi-uu-bao-mat-app-nodejs-tot-hon

▼ **CronJob**

- Cron là chương trình để xử lý các tác vụ lặp đi lặp lại ở lần sau.
- Cron Job đưa ra một lệnh để lên lịch "làm việc" cho một hành động cụ thể, tại một thời điểm cụ thể mà cần lặp đi lặp lại.
  - Giả sử ứng dụng của bạn có chức năng lưu tạm file, vậy mỗi lần người dùng lưu tạm miết vậy và không dùng, đến một lúc nào đó nó sẽ đầy và tốn dùng lượng.
  - Lúc này bạn cần một công việc tự động là 3 ngày nó sẽ dọn các file tạm đó đi.
  - Do đó, đối với các công việc định kì, lặp đi lặp lại thì cron là giải pháp hoàn hảo.

▼ **IoC**

- **Inversion of Control** (*IoC - Đảo ngược điều khiển*) là một nguyên lý thiết kế trong công nghệ phần mềm trong đó các thành phần nó dựa vào để làm việc bị đảo ngược quyền điều khiển khi so sánh với lập trình hướng thủ thục truyền thống.

▼ **Dependency Injection**

- https://2kvn.com/di-la-gi
- **Dependency Injection** (DI) là 1 design pattern được sử dụng để thực hiện Inversion of Control(IoC). DI giúp loại bỏ việc phụ thuộc trực tiếp của class với đối tượng.
- Có thể hiểu Dependency Injection một cách đơn giản như sau:
  - Các module không giao tiếp trực tiếp với nhau, mà thông qua interface. Module cấp thấp sẽ implement interface, module cấp cao sẽ gọi module cấp thấp thông qua interface.
  - Việc khởi tạo các module cấp thấp sẽ do DI Container thực hiện.
  - Việc Module nào gắn với interface nào sẽ được config trong code hoặc trong file XML
  - DI được dùng để làm giảm sự phụ thuộc giữa các module, dễ dàng hơn trong việc thay đổi module, bảo trì code và testing.

https://toidicodedao.com/2015/11/03/dependency-injection-va-inversion-of-control-phan-1-dinh-nghia/

https://kipalog.com/posts/Dependency-injection---Nhung-thu-co-the-ban-bo-qua

▼ **Push notification & real time**

https://kipalog.com/posts/Realtime--Pusher-va-ke-thay-the-Slanger

- Một số cách push notification từ server đến client
  - setInterval gọi 5s hay 10s gì đó thì call api lên server
  - Sử dụng Nodejs với websocket
  - Sử dụng cơ chế pub sub của redis
  - Sử dụng một Pusher (trả phí), Slanger (open source)
  - Sử dụng firebase
  - ...

▼ **Database**

https://techtfq.com/blog/top-20-sql-interview-questions?ref=morioh.com&utm_source=morioh.com

▼ **SQL ≠ NO SQL**

https://viblo.asia/p/nhung-diem-khac-biet-giua-sql-va-nosql-gGJ59b4rKX2

https://viblo.asia/p/sql-vs-nosql-dau-la-lua-chon-phu-hop-cho-du-an-cua-ban-Qbq5QWAzZD8

▼ **Transaction, ACID**

▼ **Transaction** là một tiến trình xử lý có xác định điểm đầu và điểm cuối, được chia nhỏ thành các operation (phép thực thi), tiến trình được thực thi một cách tuần tự và độc lập các operation đó theo nguyên tắc hoặc tất cả đều thành công hoặc một operation thất bại thì toàn bộ tiến trình thất bại. Nếu việc thực thi một operation nào đó bị fail (hỏng) đồng nghĩa với việc dữ liệu phải rollback (trở lại) trạng thái ban đầu.

- Một transaction đòi hỏi phải có 4 tính chất ACID.
- ACID là viết tắt của cụm từ Atomicity (nguyên tử), Consistency (nhất quán), Isolation (Cô lập), và Durability (Lâu bền).

- Đây là các thuộc tính mà mọi transaction đều được đảm bảo bởi SQL Server.
    - **Atomicity**: Mọi thay đổi về mặt dữ liệu phải được thục hiện trọn vẹn khi transaction thực hiện thành công hoặc không có bất kì sự thay đổi nào về mặt dữ liệu nếu có xẩy ra sự cố. ***Giải thích thêm:***
        - Một giao dịch là đơn vị cơ bản của quá trình xử lý. Tất cả các hoạt động của nó đều được thực thi, hoặc không có hoạt động nào trong số chúng. Giả sử rằng hệ thống bị treo sau thao tác Ghi (A) (nhưng trước khi ghi (B).)
        - Cơ sở dữ liệu phải có khả năng khôi phục các giá trị cũ của A và B (hoặc hoàn thành toàn bộ giao dịch)
    - **Consistency**: Sau khi một transaction kết thúc thì tất cả dữ liệu phải được nhất quán dù thành công hay thất bại. ***Giải thích thêm:***
        - Việc thực hiện một giao dịch một mình phải di chuyển cơ sở dữ liệu từ trạng thái nhất quán này sang trạng thái nhất quán khác.
        - Tổng của A và B phải không thay đổi khi thực hiện giao dịch
    - **Isolation**: Các transaction khi đồng thời thực thi trên hệ thống thì không có bất kì ảnh hưởng gì tời nhau.
        - Một giao dịch không được để các giao dịch khác biết đến ảnh hưởng của nó cho đến khi nó được commit.
        - Nếu hai giao dịch thực hiện đồng thời, có vẻ như một giao dịch đã hoàn thành trước khi giao dịch kia bắt đầu. ***Giải thích thêm:***
        - Nếu một giao dịch khác đang thực hiện đồng thời đang đọc (và / hoặc ghi vào) tài khoản A và B, nó sẽ không thể đọc dữ liệu ở trạng thái không nhất quán (sau khi ghi vào A và trước khi ghi vào B)
    - **Durability**: Sau khi một transaction thành công thì tác dụng mà nó tạo ra phải bền vững trong cơ sở dữ liệu cho dù hệ thống có xẩy ra lỗi. ***Giải thích thêm:***
        - Sau khi giao dịch được cam kết, các thay đổi đối với cơ sở dữ liệu không thể bị mất do lỗi trong tương lai.
        - Khi giao dịch hoàn tất, chúng tôi sẽ luôn có các giá trị mới của A và B trong cơ sở dữ liệu

---

▼ **Transaction** là một loạt các câu lệnh SQL hoạt động giống như một đơn vị duy nhất.

- Nói một cách dễ hiểu, giao dịch là một đơn vị mà một chuỗi công việc được thực hiện để hoàn thành toàn bộ hoạt động.
    - Chúng ta có thể lấy ví dụ về Giao dịch ngân hàng để hiểu điều này.
- Khi chúng ta chuyển tiền từ tài khoản “A” sang tài khoản “B”, một giao dịch sẽ diễn ra.
- Mỗi giao dịch đều có bốn đặc điểm, chúng được gọi là thuộc tính ACID.
    - **Atomicity:** Mọi giao dịch đều tuân theo mô hình tính nguyên tử, có nghĩa là nếu một giao dịch được bắt đầu, thì giao dịch đó phải được hoàn tất hoặc quay trở lại.
        - Ví dụ: nếu một người đang chuyển số tiền từ tài khoản “A” sang tài khoản “B”, thì số tiền đó sẽ được ghi có vào tài khoản B sau khi hoàn tất giao dịch.
        - Trong trường hợp nếu có bất kỳ lỗi nào xảy ra, sau khi ghi nợ số tiền từ tài khoản “A”, thay đổi sẽ được hoàn lại.
    - **Consistency:** Tính nhất quán nói rằng sau khi hoàn thành một giao dịch, các thay đổi được thực hiện trong quá trình giao dịch phải nhất quán.

        - Ví dụ: nếu tài khoản “A” đã được ghi nợ 200 USD thì sau khi hoàn thành giao dịch, tài khoản “B” sẽ được ghi có 200 USD
        - Nó có nghĩa là các thay đổi phải nhất quán.
    - **Isolation:** nói rằng mọi giao dịch nên được cách ly với nhau, không nên có bất kỳ sự can thiệp nào giữa hai giao dịch.
    - **Durability:** có nghĩa là một khi giao dịch được hoàn thành, tất cả các thay đổi sẽ là vĩnh viễn, có nghĩa là trong trường hợp hệ thống bị lỗi, các thay đổi sẽ không bị mất

▼ **Master & Slave**

> 💡 *Hệ thống sẽ có 2 (hoặc nhiều hơn) database giống hệt nhau, mỗi database được cài trên 1 server khác nhau. Trong số các db đó, có 1 db được gọi là master (ông chủ), các db còn lại được gọi là slave (nô lệ). Khi db master xảy ra một sự kiện làm thay đổi dữ liệu (thêm, sửa, xóa) thì db master sẽ log ra một file, các db slave thì liên tục lắng nghe file log này, và thực hiện việc thay đổi dữ liệu trên db slave như thay đổi trên db master.*

- https://viblo.asia/p/gioi-thieu-ve-mysql-replication-master-slave-bxjvZYwNkJZ

▼ **Các mô hình database phổ biến**

- **Replication:** Hiểu nôm na thì đây là kiến trúc nhân bản. Chúng ta có 1 server Master và 1 hoặc nhiều server Slave. Master dùng chủ yếu để ghi dữ liệu (có thể dùng để đọc trong trường hợp cần thiết), Slave dùng để dọc dữ liệu và không thể ghi dữ liệu.
- **Partitioning:** đây là phương pháp giúp tối ưu truy vấn khi dữ liệu trong một bảng quá lớn hàng tram triệu bản ghi. Để dễ hình dung thì 1 bảng dữ liệu giống như một chiếc hộp nhiều ngăn. Mỗi partition là một ngăn, mỗi một ngăn sẽ chưa một số lượng bản ghi theo một quy luật nào đó. Các cách chia thường theo id của bản ghi, hoặc theo thời gian tạo bản ghi theo ngày tháng
- **Cluster:** là mô hình dư liệu phân tán, kết hợp replication + sharding. Dữ liệu đươc lưu ở các datanode, truy vấn qua các sql node, và có 1 node là manager quản lý toàn bộ các sqlnode và datanode
- **Sharding:** là mô hình tương tự như partition, partition chia dữ liệu theo chiều dọc thì sharding chia dữ liệu theo chiều ngang. Mỗi một partition giờ sẽ là 1 server.

https://kipalog.kaopiz.com/posts/Cac-mo-hinh-database-nen-biet

▼ **Index**

- Link

▼ **Store Procedure**

- **Stored Procedure** là một tập hợp các câu lệnh SQL dùng để thực thi một nhiệm vụ nhất định. Nó hoạt động giống như một *hàm* trong các ngôn ngữ lập trình khác.
- Stored procedure là một khái niệm khá phổ biến và được hầu hết các hệ quản trị cơ sở dữ liệu (DBMS) hỗ trợ, tuy nhiên không phải tất cả đều hỗ trợ Stored Procedure.
- Lợi ích:
    - Module hóa: bạn chỉ cần viết SP 1 lần, sau đó có thể gọi nhiều lần trong ứng dụng
    - Hiệu suất: SP thực thi mã nhanh hơn và giảm tải băng thông.
    - Bảo mật:

https://viblo.asia/p/gioi-thieu-stored-procedure-trong-sql-server-m68Z0VpM5kG

▼ **Trigger**

- Hiểu đơn giản thì Trigger là một stored procedure không có tham số. Trigger thực thi một cách tự động khi một trong ba câu lệnh Insert, Update, Delete làm thay đổi dữ liệu trên bảng có chứa trigger
- Tác dụng:
  - Trigger thường được sử dụng để kiểm tra ràng buộc (check constraints) trên nhiều quan hệ (nhiều bảng/table) hoặc trên nhiều dòng (nhiều record) của bảng.
  - Ngoài ra việc sử dụng Trigger để chương trình có những hàm chạy ngầm nhằm phục vụ nhưng trường hợp hữu hạn và thường không sử dụng cho mục đích kinh doanh hoặc giao dịch

https://viblo.asia/p/su-dung-trigger-trong-sql-qua-vi-du-co-ban-aWj538APK6m

https://www.ibm.com/docs/en/informix-servers/12.10?topic=triggers-when-use

▼ **function ≠ procedure**

- Cả stored procedure và function đều là các đối tượng cơ sở dữ liệu chứa một tập các câu lệnh SQL để hoàn thành một tác vụ.
- Một stored procedure (thủ tục lưu trữ) có thể sử dụng lại nhiều lần. Vì vậy, nếu bạn có một truy vấn SQL mà bạn có ý định sử dụng nhiều lần thì hãy lưu nó dưới dạng một SP, sau đó chỉ cần gọi nó để nó thực thi truy vấn SQL của bạn. Ngoài ra, bạn củng có thể truyền tham số cho một SP
- Một function (hàm) được biên dịch và thực thi mỗi khi hàm đó được gọi. Hàm phải trả về giá trị...
- Khác nhau:
  - Thủ tục lưu trữ có thể trả về giá trị zero, một hoặc nhiều giá trị. Trong khi hàm phải trả về một giá trị duy nhất (có thể là bảng).
  - Các hàm chỉ có thể có các tham số đầu vào cho nó trong khi thủ tục lưu trữ có thể có các tham số đầu vào hoặc đầu ra.
  - Hàm có thể được gọi từ thủ tục lưu trữ trong khi thủ tục lưu trữ không thể được gọi từ hàm.
  - Một ngoại lệ có thể được xử lý bằng **try-catch** trong thủ tục lưu trữ, đối với hàm thì không thể.
  - Có thể sử dụng **transaction** trong thủ tục lưu trữ, với hàm thì không thể.

https://itworld.forumvi.net/t688-topic

▼ **Aggregate**

- Aggregation có thể hiểu là sự tập hợp. Các **Aggregation** operation xử lý các bản ghi dữ liệu và trả về kết quả đã được tính toán. Các phép toán tập hợp nhóm các giá trị từ nhiều Document lại với nhau, và có thể thực hiện nhiều phép toán đa dạng trên dữ liệu đã được nhóm đó để trả về một kết quả duy nhất.
- Trong SQL, count(*) và GROUP BY là tương đương với Aggregation trong MongoDB.

▼ **distinct ≠ group by**

- **DISTINCT** trong SQL được sử dụng kết hợp với câu lệnh SELECT để loại bỏ tất cả các bản ghi trùng lặp và chỉ lấy các bản ghi duy nhất.
- **GROUP BY** trong SQL được sử dụng hợp tác với câu lệnh SELECT để sắp xếp dữ liệu giống nhau thành các nhóm. Mệnh đề GROUP BY này tuân theo mệnh đề WHERE trong câu lệnh SELECT và đứng trước mệnh đề ORDER BY.
- Cả hai mệnh đề đều làm giảm số lượng record trả về bằng cách loại bỏ các bản ghi trùng lặp
- Nhưng khi dùng **group by** ta có thể dùng kết hợp thêm một số aggregate như sum, count, ... còn **distinct** thì không

▼ Inner, outer, left, right join

- **JOIN** trong SQL được sử dụng để kết hợp các bản ghi từ hai hay nhiều bảng trong cơ sở dữ liệu. JOIN là một phương tiện để kết hợp các trường từ hai bảng bằng cách sử dụng các giá trị chung cho mỗi bảng.
- **INNER JOIN**: Trả về các bản ghi có những giá trị phù hợp trong cả hai bảng.
- **LEFT (OUTER) JOIN**: Trả về tất cả bản ghi từ bảng bên trái và bản ghi trùng với bảng bên phải.
- **RIGHT (OUTER) JOIN**: Trả về tất cả bản ghi từ bảng bên phải và bản ghi trùng với bản bên trái.
- **FULL (OUTER) JOI**N: Trả về tất cả bản ghi khi có một kết quả phù hợp trong bảng bên trái hoặc bên phải.

▼ where ≠ having clause

- WHERE - filter kết quả theo dòng
- HAVING - filter kết quả theo GROUP

- Where : Là câu lệnh điều kiện trả kết quả đối chiếu với từng dòng
- Having : Là câu lệnh điều kiện trả kết quả đối chiếu cho nhóm (Sum,AVG,COUNT,…)

⇒ Vì vậy mà sau GROUP BY thì sẽ chỉ dùng được Having còn Where thì KHÔNG dùng được sau GROUP BY

(HAVING có thể thay thế vị trí dùng cho WHERE nhưng ngược lại WHERE thì KHÔNG thể thay thế vị trí cho HAVING)

https://daynhauhoc.com/t/su-khac-nhau-giua-where-va-having/21676/2

▼ delete ≠ truncate ≠ drop

- DELETE : Xóa một hay tất cả dòng trong một bảng theo một điều kiện nhất định, dữ liệu có thể phục hồi lại
- TRUNCATE : Xóa toàn bộ các dòng của bảng, giải phóng bộ nhớ và không thể phục hồi lại
- DROP : Xóa một bảng khỏi database
- Truncate và drop không thể dùng mệnh đề Where

▼ Subqueries?

- Subquery hay còn gọi là truy vấn con, đây là cách viết một câu lệnh SQL mà trong đó có lồng thêm một hoặc nhiều câu truy vấn khác, và thường được sử dụng trong bốn lệnh: SELECT, INSERT, UPDATE hoặc DELETE. Cùng với đó là các toán tử ví dụ như =, <, >, >=, <=, IN, BETWEEN…
- Có một vài quy tắc mà Sub query phải tuân theo:
  - Sub query phải được đặt trong dấu ngoặc đơn.
  - Một sub query có thể chỉ có một cột trong mệnh đề SELECT, trừ khi nhiều cột trong truy vấn chính cho sub query để so sánh các cột đã chọn của nó.
  - Không thể sử dụng lệnh ORDER BY trong sub query, mặc dù truy vấn chính có thể sử dụng ORDER BY. Lệnh GROUP BY có thể được sử dụng để thực hiện chức năng giống như ORDER BY trong một sub query.
  - Sub query trả về nhiều hơn một hàng chỉ có thể được sử dụng với toán tử nhiều giá trị như toán tử IN.
  - Danh sách SELECT không được bao gồm bất kỳ tham chiếu nào đến các giá trị đánh giá BLOB, ARRAY, CLOB hoặc NCLOB.
  - Một sub query không thể được chứa trực tiếp một chức năng set.
  - Toán tử BETWEEN không thể được sử dụng với một sub query. Tuy nhiên, toán tử BETWEEN có thể được sử dụng trong sub query.

▼ Union?

- Toán tử UNION được sử dụng để kết hợp tập hợp kết quả của hai hoặc nhiều câu lệnh SELECT. Mỗi câu lệnh SELECT với UNION phải có cùng số lượng cột, các cột phải có cùng kiểu dữ liệu, các cột trong mỗi câu lệnh SELECT phải có cùng trật tự.

- Mệnh đề **UNION** trong SQL được sử dụng để kết hợp các kết quả của hai hoặc nhiều câu lệnh SELECT mà không cần trả về bất kỳ hàng trùng lặp nào.
- Để sử dụng mệnh đề UNION này, mỗi câu lệnh SELECT cần phải có
  - Cùng một số cột được chọn
  - Cùng một số biểu thức cột
  - Cùng kiểu dữ liệu
  - Có chúng trong cùng một trật tự

- **UNION** kết hợp lại nhưng loại bỏ trùng nhau
- **UNION ALL** kết hợp lại nhưng không loại bỏ trùng nhau

▼ **Connection pool**

- Connection pool (vùng kết nối) : là kỹ thuật cho phép tạo và duy trì 1 tập các kết nối dùng chung nhằm tăng hiệu suất cho các ứng dụng bằng cách sử dụng lại các kết nối khi có yêu cầu thay vì việc tạo kết nối mới.
- Connection Pool Manager (CPM) là trình quản lý vùng kết nối, một khi ứng dụng được chạy thì Connection pool tạo ra một vùng kết nối, trong vùng kết nối đó có các kết nối do chúng ta tạo ra sẵn. Và như vậy, một khi có một request đến thì CPM kiểm tra xem có kết nối nào đang rỗi không? Nếu có nó sẽ dùng kết nối đó còn không thì nó sẽ đợi cho đến khi có kết nối nào đó rỗi hoặc kết nối khác bị timeout. Kết nối sau khi sử dụng sẽ không đóng lại ngay mà sẽ được trả về CPM để dùng lại khi được yêu cầu trong tương lai.

▼ Data warehouse ≠ Data lake

▼ **Design patterns**

1. Strategy

▼ **Singleton**

- https://2kvn.com/design-patterns-singleton-p5f35363236

1. Observer

https://blog.bitsrc.io/3-design-patterns-every-developer-should-learn-71a51568ac9d

▼ **RULE**

1. SOLID
2. DRY

▼ **Systems**

▼ **HTTP ≠ HTTPS, SSL**

- **HTTP**: giao thức không bảo mật, mọi dữ liệu truyền trên internet sẽ không được mã hoá, nghĩa là khi ta submit một form trên trang đó (VD: đăng nhập bằng password, gửi thông tin tài khoản ngân hàng, hay thậm chí các tin nhắn thông thường) thì mọi thông tin bạn nhập và gửi nếu để ai đó trên internet bắt được gói tin, bạn sẽ bị lộ những thông tin đó.
- **HTTPS**: giao thức bảo mật, giúp cho gói tin gửi tới server được mã hoá khi đi trên internet, ai đó có bắt được gói tin họ cũng không thể đọc được thông tin nằm trong gói tin đó.

- **HTTPS** sử dụng giao thức **SSL (còn gọi là TLS)** để mã hoá dữ liệu của **HTTP**
- Cách giao thức **SSL** hoạt động: https://blog.daovanhung.com/post/cach-hoat-dong-cua-ssl-va-https

▼ **stateless ≠ stateful**

- **stateless**: là design không lưu dữ liệu của client trên server. Có nghĩa là sau khi client gửi dữ liệu lên server, server thực thi xong, trả kết quả thì "quan hệ" giữa client và server bị "cắt đứt" – server không lưu bất cứ dữ liệu gì của client.
- **stateful**: là một design ngược với stateless, server cần lưu dữ liệu của client, điều đó đồng nghĩa với việc ràng buộc giữa client và server vẫn được giữ sau mỗi request (yêu cầu) của client. Data được lưu lại phía server có thể làm đầu vào (input parameters) cho lần kế tiếp, hoặc là dữ kiện dùng trong quá trình xử lý hay phục vụ cho bất cứ nhu cầu nào phụ thuộc vào bussiness (nghiệp vụ) cài đặt.

▼ **Vertical scale ≠ Horizontal scale**

- **scaling vertically**: mở rộng theo chiều dọc, là cách mở rộng server hiện tại bằng cách nâng cấp độ mạnh (power) bằng cách nâng cấp CPU, Ram, Storage, v.V… Vertical-scaling thường bị giới hạn bởi vượt quá khả năng về cấu hình vật lý hiện đại hay độ trễ khi "chẳng may" Server bị downtime để nâng cấp hay deploy hệ thống.
- **scaling horizontally**: mở rộng theo chiều ngang, là cách mở rộng bằng cách thêm nhiều Node/Server vào một mạng lưới đang có, làm tăng khả năng chịu tải có hệ thống. Cách làm này rẻ và dễ làm hơn so với Vertical-scaling, đặc biệt là rất dễ dàng downsize cũng như upsize hệ thống.

▼ **CDN**

- CDN là một nhóm server đặt tại nhiều vị trái khác nhau để hỗ trợ nội dung được trải dài ở nhiều khu vực vị trí địa lý khác nhau.
- CDN cũng được gọi là "distribution networks". Ý tưởng là tạo được nhiều điểm truy cập (Point of Presence – PoPs) ngoài server gốc. Việc này giúp website quản lý tốt traffic hơn bằng cách xử lý nhanh hơn yêu cầu của khách, tăng trải nghiệm người dùng.
- Cách CDN hoạt động:
  - Trong mạng lưới Content Delivery. Mỗi điểm hiện diện (location) được gọi là một **PoPs**.
  - Để tăng thời gian phản hồi giữa client và server (người dùng và trang web), các PoPs (node trong mạng lưới) sẽ **lưu nội dung trang web vào bộ nhớ (cached)** của mình và làm mới nó thường xuyên.
  - Khi người dùng yêu cầu nội dung trang web, người dùng sẽ không trực tiếp truy cập tới trang web (ở bờ Tây nước Mỹ chẳng hạn) mà chỉ truy cập với một điểm CDN **gần mình nhất**.
- Khi nào nên sử dụng?
  - Nếu content của bạn chỉ có **một lượng nhỏ truy cập** ở vị trí địa lí **gần nơi đặt máy chủ**, không cần thiết phải dùng CDN.
  - Ngược lại, nếu nội dung của bạn được truy cập và sử dụng ở khắp nơi trên thế giới. Đăng kí tham gia mạng lưới CDN là cần thiết giúp **tăng trải nghiệm người dùng**.

▼ **DNS**

- DNS là viết tắt của Domain Name System tạm dịch là "Hệ thống phân giải tên miền". Về bản chất cách để máy tính truy cập được một trang web là nhờ địa chỉ IP.
- Ví dụ bạn muốn truy cập vào **google.com** thì tức là browser đang request tới IP máy chủ của google.
- Tuy nhiên có cả triệu website và bạn phải nhớ địa chỉ IP của từng trang web, điều đó là bất khả thi và chưa kể trường hợp địa chỉ IP của trang web đó có thể thay đổi liên tục. Đó cũng chính là nguyên nhân mà DNS được sinh ra.

- DNS sẽ đóng vai trò như một cuốn danh bạ, thay vì phải nhớ 1 dãy IP loằng ngoằng thì bạn sẽ nhớ đến tên miền của trang web đó
- Ví dụ như **google.com** và tất nhiên như vậy sẽ thân thiện với người sử dụng hơn, và DNS sẽ có vai trò là phân giải tên miền thành địa chỉ IP tương ứng nhờ đó browser có thể gửi request tới server.

▼ **Load balancer**

- Load Balancing là quá trình của việc phân phối lưu lượng truy cập một cách hiệu quả thông qua nhiều server hay còn được gọi là `server farm` hay `server pool`
- Việc phân phối đồng một cách đồng đều sẽ cải thiện khả năng đáp ứng và tăng tính khả dụng của các ứng dụng.
- Phương pháp này ngày càng cần thiết vì các ứng dụng ngày nay đã phức tạp hơn, cùng với nhu cầu của người sử dụng tăng và lưu lượng truy cập tăng lên.
- Load balancer đã giải quyết được các vấn đề như:
  - Performance: dễ dàng scale up (vertical scaling) và scale out (horizontal scaling)
  - Availability: có server dự phòng và cơ chế tự động khôi phục. Nếu 1 server bị lỗi sẽ không ảnh hưởng đến toàn bộ hệ thống
  - Economy: triển khai một server có hiệu năng lớn thì đắt hơn so với một cụm server có hiệu năng nhỏ. Chi phí để duy trì một cụm server nhỏ thì rẻ hơn và dễ dàng thêm hoặc nâng cấp server trong cụm này so với việc nâng cấp và thay thế một server lớn
- Load balancer có 3 kiến trúc cơ bản:
  - Dựa trên DNS
  - Dựa trên phần cứng
  - Dựa trên phần mềm
- Các thuật toán cơ bản:
  - Round Robin
  - Weighted Round Robin
  - Dynamic Round Robin
  - Fastest
  - Least Connections

Link: https://anonystick.com/blog-developer/load-balancer-neu-ban-khong-hieu-khong-sao-nhung-neu-ban-la-mot-ky-su-thi-khong-the-khong-hieu-202006243445464

▼ **Nginx**

- là một máy chủ web server, open source, được thiết kế hướng đến mục đích cải thiện tối đa hiệu năng và sự ổn định.
- Ngoài ra nhờ vào khả năng của máy chủ http, nginx còn có thể hoạt động như proxy server cho email, reverse proxy, http caching hay load balancer cho các máy chủ http, tcp, udp
- Nginx được sử dụng kiến trúc đơn luồng và event driven (hướng sự kiện) vì thế nó hiệu quả hơn apache server nếu cấu hình chính xác.
- NGINX được cấu hình theo kiểu bất đồng bộ (asynchronous): nghĩa là 1 NGINX process có thể xử lý nhiều request liên tục, dựa vào số lượng tài nguyên còn lại của hệ thống.

- Nhờ kiểu cấu hình như vậy, NGINX có thể “nhúng” các file lập trình (như .php) vào process riêng của nó. Nghĩa là mọi request yêu cầu data được 1 process riêng của NGINX thực hiện, và trả data lại cho client bằng reverse proxy.
- Bên cạnh đó, đối với những file tĩnh (file .txt, file .css hay các file hình ảnh), NGINX sẽ trả dữ liệu mà không cần sự can thiệp của các module server side.

▼ **TCP ≠ UDP**

- **TCP** là giao thức truyền tải hướng kết nối (connection-oriented), nghĩa là phải thực hiện thiết lập kết nối với đầu xa trước khi thực hiện truyền dữ liệu. Tiến trình thiết lập kết nối ở TCP được gọi là tiến trình ***bắt tay 3 bước*** (*threeway handshake*).
  - Cung cấp cơ chế báo nhận (Acknowledgement) :Khi A gửi dữ liệu cho B, B nhận được thì gửi gói tin cho A xác nhận là đã nhận. Nếu không nhận được tin xác nhận thì A sẽ gửi cho đến khi B báo nhận thì thôi.
- **UDP** là giao thức truyền tải hướng không kết nối (connectionless). Nó sẽ không thực hiện thao tác xây dựng kết nối trước khi truyền dữ liệu mà thực hiện truyền ngay lập tức khi có dữ liệu cần truyền (kiểu truyền best effort) => truyền tải rất nhanh cho dữ liệu của lớp ứng dụng.
  - Không đảm bảo tính tin cậy khi truyền dữ liệu và không có cơ chế phục hồi dữ liệu (nó không quan tâm gói tin có đến đích hay không, không biết gói tin có bị mất mát trên đường đi hay không) => dễ bị lỗi.

https://viblo.asia/p/tim-hieu-giao-thuc-tcp-va-udp-jvEla11xlkw

▼ **Quy tác bắt tay 3 bước**

```
* Gói dữ liệu với cờ SYN (synchronization <=> Sự đồng bộ) dùng để bắt đầu một connection.
* ACK (acknowledgement <=> Xác nhận).
* FIN (finish <=> hoàn thành) dùng để ngắt một connection.
* ...
```

- **B1**:
  - SYN: các chương trình máy con (ví dụ yêu cầu từ browser, ftp client) bắt đầu connection với máy chủ bằng cách gửi một packet với cờ "SYN" đến máy chủ.
  - SYN packet này thường được gửi từ các cổng cao (1024 - 65535) của máy con đến những cổng trong vùng thấp (1 - 1023) của máy chủ.
  - Chương trình trên máy con sẽ hỏi hệ điều hành cung cấp cho một cổng để mở connection với máy chủ.
  - Những cổng trong vùng này được gọi là "cổng máy con" (client port range).
  - Tương tự như vậy, máy chủ sẽ hỏi HĐH để nhận được quyền chờ tín hiệu trong máy chủ, vùng cổng 1 - 1023.
  - Vùng cổng này được gọi là "vùng cổng dịch vụ" (service port).

    ```
    - Ví dụ (mặc định):
        - Web Server sẽ luôn chờ tín hiệu ở cổng 80 và Web browser sẽ connect vào cổng 80 của máy chủ.
        - FTP Server sẽ lắng ở port 21.

     Ngoài ra trong gói dữ liệu còn có thêm địa chỉ IP của cả máy con và máy chủ.
    ```

- **B2**:
  - SYN/ACK: khi yêu cầu mở connection được máy chủ nhận được tại cổng đang mở, server sẽ gửi lại packet chấp nhận với 2 bit cờ là SYN và ACK.
  - SYN/ACK packet được gửi ngược lại bằng cách đổi hai IP của server và client, client IP sẽ thành IP đích và server IP sẽ thành IP bắt đầu. Tương tự như vậy, cổng cũng sẽ thay đổi, server nhận được packet ở cổng

nào thì cũng sẽ dùng cổng đó để gửi lại packet vào cổng mà client đã gửi.

- Server gửi lại packet này để thông báo là server đã nhận được tín hiệu và chấp nhận connection, trong trường hợp server không chấp nhận connection, thay vì SYN/ACK bits được bật, server sẽ bật bit RST/ACK (Reset Acknowledgement) và gởi ngược lại RST/ACK packet.
- Server bắt buộc phải gửi thông báo lại bởi vì TCP là chuẩn tin cậy nên nếu client không nhận được thông báo thì sẽ nghĩ rằng packet đã bị lạc và gửi lại thông báo mới.

- **B3**:
  - ACK: khi client nhận được SYN/ACK packet thì sẽ trả lời bằng ACK packet.
  - Packet này được gởi với mục đích duy báo cho máy chủ biết rằng client đã nhận được SYN/ACK packet và lúc này connection đã được thiết lập và dữ liệu sẽ bắt đầu lưu thông tự do.
  - Đây là tiến trình bắt buộc phải thực hiện khi client muốn trao đổi dữ liệu với server thông qua giao thức TCP.
  - Một số thủ thuật dựa vào đặc điểm này của TCP để tấn công máy chủ (ví dụ DOS).

▼ Reverse Proxy

- Reverse proxy là một loại proxy server trung gian giữa một máy chủ và các client gửi tới các yêu cầu. Nó kiểm soát yêu cầu của các client, nếu hợp lệ, sẽ luân chuyển đến các server thích ứng.
- Trái ngược với một **forward proxy**, là một trung gian cho phép các client liên hệ với nó liên lạc với bất kỳ máy chủ ảo nào, reverse proxy là một trung gian cho các máy chủ liên hệ với nó được liên lạc bởi bất kỳ client nào.
- Ưu điểm lớn nhất của việc sử dụng reverse proxy là khả năng quản lý tập trung. Nó giúp kiểm soát mọi request do client gửi lên các server được bảo vệ.
- Reverse proxy server được dùng để làm gì?
  - Reverse proxy ở giữa client và network service, như là website. Một số tính năng mà nó mang lại như:
    - Bảo mật
    - Load balancing
    - Tăng tốc độ trang web

https://viblo.asia/p/reverse-proxy-server-la-gi-eW65GW4P5DO

▼ Cluster, Node, Container

- **Clustering** chính là 1 kiến trúc được tạo ra với mục đích đảm bảo nâng cao khả năng sẵn sàng cho những hệ thống mạng. Clustering bao gồm những server riêng lẻ được kết nối với nhau đồng thời hoạt động lại cùng nhau trong 1 hệ thống. Những server này giao tiếp với nhau với mục đích trao đổi thông tin và giao tiếp với cả những mạng bên ngoài để thực hiện những yêu cầu. Trong trường hợp có lỗi xảy ra những dịch vụ trong cluster hoạt động tương tác với nhau để duy trì tính ổn định và độ sẵn sàng cao cho hệ thống.
- **Node** là những server con trong cụm cluster
- **Container** là nơi chứa ứng dụng hoặc service của chúng ta

▼ Concurrent, latency, consistency

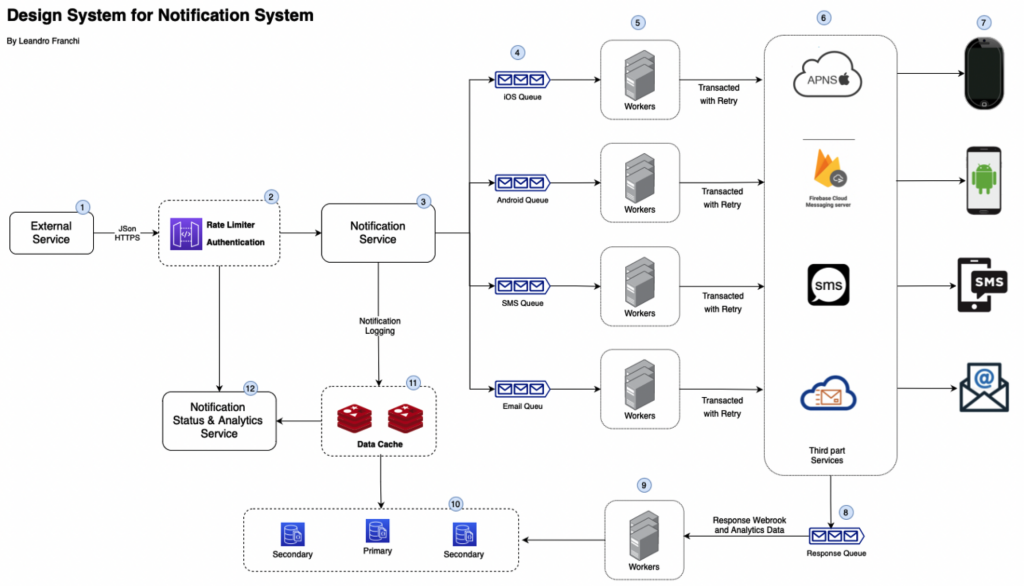
- **Latency** được biết tới như là khoảng thời gian từ lúc chúng ta yêu cầu tải trang web cho tới khi thật sự nhìn thấy nội dung trên trang web đó.

▼ 10 mẹo cải thiện hiệu suất máy chủ

- Sử dụng máy chủ proxy reverse để tăng tốc và bảo vệ các ứng dụng
- Load balancing

- Cache và content
- Nén dữ liệu
- Tối ưu hóa SSL / TLS
- Triển khai HTTP / 2 hoặc SPDY
- Cập nhật phiên bản phần mềm liên tục
- Tối ưu hóa hiệu suất Linux
- Tối ưu hóa hiệu suất của máy chủ Web
- Giám sát các hoạt động thời gian thực để giải quyết các vấn đề và tắc nghẽn
- https://anonystick.com/blog-developer/10-meo-de-cai-thien-hieu-suat-cua-cac-ung-dung-web-len-10-lan-2020051556310698#t-2

▼ **Notification system**



- **External software** sẽ send một JSON message qua https với message data giống như address, type, message, ...
- **Rate limiter** validate những rules để bảo vệ system khỏi bị overload và những vấn đề về bảo mật
- Notification service sẽ nhận message và chuyển message đó đến với những message queue tương ứng và sau đó sẽ writes một số logs về notification xuống **Data cache cluster**
- Các **workers** sẽ lấy message trong queue và connect đến với third part software để send message đến các thiết bị tương ứng
- **Third part software** sẽ call back bằng cách sử dụng web hook để nhận status và một số thông tin của message
- Sau đó có những **workers** sẽ chạy và lấy những thông tin response và lưu trữ vào **data store**
- Thông tin về status và analytics data lúc này sẽ có sẵn trong **Notification status & analytics service** và ta có thể truy cập service này từ bên ngoài với **external service**

⇒ **Lợi ích:**

- **Reliability**: giảm thiểu lỗi
- **Security**: chạy trên https với appKey và appSecret để đảm bảo user đã authorized mới có thể send message
- **Tracking and monitoring**: logs, status và analytics data sẽ được lưu trữ trên db có thể truy vấn cách dễ dàng
- **Rate limiting**: bảo vệ hệ thống khỏi overload, và những vấn đề bảo mật không mong muốn

▼ **HTTP status codes**

- 1xx Informational
- 2xx Succesful
- 3xx Redirection
- 4xx Client Error
- 5xx Server Error

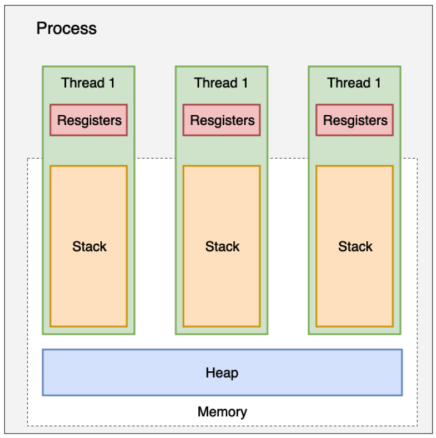
▼ **Xử lý đồng thời và xử lý song song**

https://zalopay-oss.github.io/go-advanced/ch1-basic/ch1-05-concurrency-parallelism.html

- **Xử lý đồng thời** là khả năng phân chia và điều phối nhiều tác vụ khác nhau trong cùng một khoảng thời gian và tại một thời điểm chỉ có thể xử lý một tác vụ. Khái niệm này trái ngược với **xử lý tuần tự** (sequential processing).
  - **Xử lý tuần tự** là khả năng xử lý chỉ một tác vụ trong một khoảng thời gian, các tác vụ sẽ được thực thi theo thứ tự hết tác vụ này sẽ thực thi tiếp tác vụ khác.
- **Xử lý song song** là khả năng xử lý nhiều tác vụ khác nhau trong cùng một thời điểm, các tác vụ này hoàn toàn độc lập với nhau.
  - Xử lý song song chỉ có thể thực hiện trên máy tính có số nhân lớn hơn 1.
  - Thay vì một nhân CPU chúng ta chỉ có thể xử lý một tác vụ nhỏ tại một thời điểm thì khi số nhân CPU có nhiều hơn chúng ta có thể xử lý các tác vụ song song với nhau cùng lúc trên các nhân CPU.

▼ **Processes & Threads**

- **Process**
  - Tiến trình có thể hiểu đơn giản là một chương trình đang chạy trong máy tính.
  - Khi chúng ta mở trình duyệt và truy cập một trang web thì đây được xem là một tiến trình.
  - Khi chúng ta viết 1 chương trình máy tính bằng ngôn ngữ lập trình như C, Java, hay Go, sau khi tiến hành biên dịch và chạy chương trình thì hệ điều hành sẽ cấp cho chương trình một không gian bộ nhớ nhất định, PID (process ID),...
  - Mỗi tiến trình có ít nhất một luồng chính (main thread) để chạy chương trình, nó như là xương sống của chương trình vậy. Khi luồng chính này ngừng hoạt động tương ứng với việc chương trình bị tắt.
- **Thread**
  - Thread hay được gọi là tiểu trình nó là một luồng trong tiến trình đang chạy.
  - Các luồng được chạy song song trong mỗi tiến trình và có thể truy cập đến vùng nhớ được cung cấp bởi tiến trình, các tài nguyên của hệ điều hành,...



Mô hình xử lý song song

- Các thread trong process sẽ được cấp phát riêng một vùng nhớ `stack` để lưu các biến riêng của thread đó.
- Stack được cấp phát cố định khoảng `1MB-2MB`. Ngoài ra các thread chia sẻ chung vùng nhớ `heap` của process.
- Khi process tạo quá nhiều thread sẽ dẫn đến tình trạng stack overflow. Khi các thread sử dụng chung vùng nhớ sẽ dễ gây ra hiện tượng race condition.

https://zalopay-oss.github.io/go-advanced/ch1-basic/ch1-05-concurrency-parallelism.html

https://stackoverflow.com/questions/200469/what-is-the-difference-between-a-process-and-a-thread/200543#200543

▼ CI/CD

- **CI** là Continuous Integration. Nó là phương pháp phát triển phần mềm yêu cầu các thành viên của team tích hợp công việc của họ thường xuyên, mỗi ngày ít nhất một lần.
  - Mỗi tích hợp được "build" tự động (bao gồm cả test) nhằm phát hiện lỗi nhanh nhất có thể.
  - Cả team nhận thấy rằng cách tiếp cận này giảm thiểu vấn đề tích hợp và cho phép phát triển phần mềm nhanh hơn.
- Nếu CI đảm nhận nhiệm vụ xây dựng và kiểm tra một cách tự động thì CD lại có nhiệm vụ cao hơn một chút. CD được viết tắt bởi Continuous Delivery - chuyển giao liên tục.
  - Đây là quá trình nâng cao hơn chút đó là kiểm tra tất cả những thay đổi về code đã được build và code trong môi trường kiểm thử.
  - CD cho phép các lập trình viên tự động hóa phần mềm testing, kiểm tra phần mềm qua nhiều thước đo trước khi triển khai.
  - Những bài test này có thể bao gồm UI testing, integration testing, API testing,... CD sử dụng Deployment Pipeline giúp chia quy trình chuyển giao thành các giai đoạn.
  - Mỗi giai đoạn có những mục tiêu riêng để xác minh chất lượng của các tính năng từ một góc độ vô cùng khác để có thể kiểm định được chức năng và tránh những lỗi phát sinh ảnh hưởng đến người dùng.
- Trên thực tế thì **CI/CD** là một quy trình làm việc, code của bạn sẽ được build test và sau đó deploy trên server hoặc cloud một cách tự động luôn.

▼ Docker

- **Khái niệm**:

- Docker là một nền tảng để cung cấp cách để building, deploying và running ứng dụng dễ dàng hơn bằng cách sử dụng các containers (trên nền tảng ảo hóa) để đóng gói ứng dụng.
- Docker sử dụng công nghệ ảo hóa containerization để triển khai các ứng dụng vào trong container ảo hóa.
- Docker sử dụng nhân kernel linux để chạy các container, trên hệ điều hành Linux, Docker có thể sử dụng trực tiếp nhân của máy host; còn với các hệ điều hành Windows, MacOS – có thể vì lý do bảo mật nên docker không thể trực tiếp xài chung kernel với các hệ điều hành này nên trên các hệ điều hành này docker sẽ tạo ra một máy ảo virtual guest với nhân linux để chạy các container.
- Container là đơn vị phần mềm cung cấp cơ chế đóng gói ứng dụng, mã nguồn, thiết lập, thư viện... vào một đối tượng duy nhất. Ứng dụng sau khi được đóng gói có thể hoạt động một cách nhanh chóng và hiệu quả trên các môi trường điện toán khác nhau. Từ đó nó có thể tạo ra một môi trường hoàn hảo nơi mà có mọi thứ để chương trình có thể hoạt động được, không chịu sự tác động từ môi trường của hệ thống cũng như không làm ảnh hưởng ngược lại về phía hệ thống chứa nó.

- **Docker gồm có 3 thành phần chính:**
  - **Docker file: l**à một file dạng text không có phần đuôi mở rộng, chứa các đặc tả về một trường thực thi phần mềm, cấu trúc cho Docker image. Docker image có thể được tạo ra tự động bằng cách đọc các chỉ dẫn trong Dockerfile. Từ những câu lệnh đó, Docker sẽ build ra Docker image
  - **Image:** là 1 đơn vị đóng gói chứa mọi thứ cần thiết để 1 ứng dụng chạy. Image được tạo thành từ nhiều layer xếp chồng lên nhau, bên trong image là 1 hệ điều hành bị cắt giảm và tất cả các phụ thuộc (dependencies) cần thiết để chạy 1 ứng dụng.
  - **Container:** là đơn vị phần mềm cung cấp cơ chế đóng gói ứng dụng, mã nguồn, thiết lập, thư viện... vào một đối tượng duy nhất. Ứng dụng sau khi được đóng gói có thể hoạt động một cách nhanh chóng và hiệu quả trên các môi trường điện toán khác nhau. Từ đó nó có thể tạo ra một môi trường hoàn hảo nơi mà có mọi thứ để chương trình có thể hoạt động được, không chịu sự tác động từ môi trường của hệ thống cũng như không làm ảnh hưởng ngược lại về phía hệ thống chứa nó.

Mỗi container bao gồm mọi thứ cần thiết để chạy được nó: code, runtime, system tools, system libraries, setting. Mỗi container như 1 hệ điều hành thực sự, bên trong mỗi container sẽ chạy 1 ứng dụng

▼ **How work when clicking to url on browser?**

Làm thế nào một trang web khi truy cập bằng địa chỉ url (Ví dụ: https://google.com/) lại có thể hiển thị được. Khi anh em bật trình duyệt lên, nhập địa chỉ một trang web (Ví dụ: https://google.com/) và bấm Enter, vài giây sau nội dung của trang web sẽ được hiện ra trên trình duyệt. Quá trình đó được tóm tắt như sau:

1. Từ tên miền (Ví dụ: https://google.com/) máy của anh em sẽ sử dụng DNS để tìm ra địa chỉ IP thực sự của web server tương ứng chứa website có tên miền đó (Ví dụ: https://google.com/ tương ứng với IP: 142.251.10.99).
2. Sau khi đã tìm được địa chỉ IP, trình duyệt sẽ gửi gói tin yêu cầu – HTTP request đến địa chỉ của web server, yêu cầu trả về nội dung trang web. Gói tin yêu cầu đó cũng như tất cả các gói tin, dữ liệu khác trao đổi giữa máy chủ với máy chúng ta (gọi là máy khách) được thực hiện qua một bộ giao thức TCP/IP.
3. Khi nhận được yêu cầu từ máy khách, máy chủ sẽ tiến hành trả về các tập tin HTML, CSS, JS,... để hiển thị trên trình duyệt. Các tập tin này có thể được chia thành nhiều gói tin (packets) nhỏ và gửi về cho trình duyệt của người dùng đang ở máy khách.
4. Khi nhận được, trình duyệt sẽ ghép những gói tin nhỏ nhận được thành những tập tin hoàn chỉnh và hiển thị lên màn hình. Như thế là chúng ta có một trang web hoàn chỉnh.

▼ **GIT**

▼ **Difference between git fetch and git pull?**

- The Git fetch command only downloads new data from a remote repository.
- Git pull updates the current branch with the latest changes from the remote server.

▼ **rebase ≠ revert**

https://blog.it-club.ge/interview-questions-junior-developer

VNG interview

- kiểm tra số chính phương
- kiểm tra có 2 kí tự liên tiếp trong 1 string ko, có thì swap 2 kí tự đó Data structure hỏi LL
- quick sort vs merge sort, complexity, binary search tree
- khi nào dùng LL/array
- OOP hỏi có biết design pattern nào ko?, mình quất ngay singleton 4 characteristic của OOP, polimorphisim, static var, phần cuối cùng là statistic
- phân biệt mean, mode, median, khi nào sài 3 cái đó
- cách phân biệt dữ liệu phân tán, tập trung, linear regression vs logistic regression, standard deviation

- Cách thiết kế LRU cache
- Cách thiết kế bit.ly, rút gọn link