**-GIT**

I.Các lệnh git cơ bản

**1) git config**

Tác dụng : Để set user name và email của bạn trong main configuration file.  
Cách xài : Để kiểm tra tên và kiểu email trong cấu hình dùng ***git config -- global user.name* và *git config -- global user.email*. Để set email hoặc tên mới *git config -- global user.name = “Hải Nguyễn”* và *git config -- global user.email =***[***hainguyen@gmail.com***](mailto:hainguyen@gmail.com)

#### ****2) git init****

Tác dụng : Khởi tạo 1 git repository 1 project mới hoặc đã có.

Cách xài: ***git init*** trong thư mục gốc của dự án.

#### ****3) git clone****

Tác dụng: Copy 1 git repository từ remote source.

Cách xài: ***git clone <:clone git url:>***

#### ****4) git status****

Tác dụng: Để check trạng thái của những file bạn đã thay đổi trong thư mục làm việc. VD: Tất cả các thay đổi cuối cùng từ lần commit cuối cùng.

Cách xài: ***git status*** trong thư mục làm việc.

#### ****5) git add****

Tác dụng: Thêm thay đổi đến stage/index trong thư mục làm việc.

Cách xài: ***git add***

#### ****6) git commit****

Tác dụng: commit nghĩa là một action để Git lưu lại một snapshot của các sự thay đổi trong thư mục làm việc. Và các tập tin, thư mục được thay đổi đã phải nằm trong Staging Area. Mỗi lần commit nó sẽ được lưu lại lịch sử chỉnh sửa của code kèm theo tên và địa chỉ email của người commit. Ngoài ra trong Git bạn cũng có thể khôi phục lại tập tin trong lịch sử commit của nó để chia cho một branch khác, vì vậy bạn sẽ dễ dàng khôi phục lại các thay đổi trước đó.

#### ****7) git push/git pull****

Tác dụng: Push hoặc Pull các thay đổi đến remote. Nếu bạn đã added và committed các thay đổi và bạn muốn đẩy nó lên hoặc remote của bạn đã update và bạn apply tất cả thay đổi đó trên code của mình.

Cách dùng: ***git pull <:remote:> <:branch:>*** and ***git push <:remote:> <:branch:>***

#### ****8) git branch****

Tác dụng: liệt kê tất cả các branch (nhánh).

Cách dùng: ***git branch*** hoặc ***git branch -a***

#### ****9) git checkout****

Tác dụng: Chuyển sang branch khác

Cách dùng: ***git checkout <: branch:>*** hoặc ***\*\* \_ git checkout -b <: branch:>*** nếu bạn muốn tạo và chuyển sang một chi nhánh mới.

#### ****10) git stash****

Tác dụng: Lưu thay đổi mà bạn không muốn commit ngay lập tức.

Cách dùng: ***git stash*** trong thư mục làm việc của bạn.

#### ****11) git merge****

Tác dụng: Merge 2 branch lại với nahu.

Cách dùng: Chuyển tới branch bạn muốn merge rồi  dùng ***git merge <:branch\_ban\_muon\_merge:>***

#### ****12) git reset****

Tác dụng: Bạn đã đưa một tập tin nào đó vào Staging Area nhưng bây giờ bạn muốn loại bỏ nó ra khỏi đây để không phải bị commit theo.

Cách dùng: ***git reset HEAD tên\_file***

#### ****13) git remote****

Tác dụng: Để check remote/source bạn có hoặc add thêm remote

Cách dùng: ***git remote*** để kiểm tra và liệt kê. Và ***git remote add <: remote\_url:>*** để thêm.

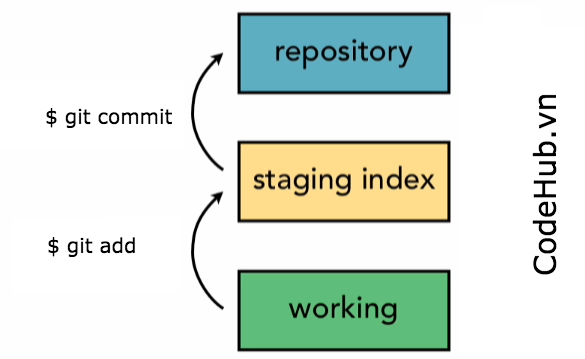
#### ****14) git add****

Tác dụng: Để đưa một tập tin vào Staging Area

Cách dùng: ***git add tên\_file*** hoặc muốn thêm hết file của thư mục thì ***git add all***

II. Cấu trúc của một Git repository trên máy local sẽ gồm có 3 phần:

* Working copy: Chứa các file ở commit hiện tại (hay HEAD).
* Staging index: Chứa các thay đổi đã được thêm vào.
* Repository: Chứa các commit version với các file tương ứng



II. Work Flow

B1 : Tạo Respoitory trên local:

**cd your-project/**

**git init**

Trong trường hợp đã có repository, thì bạn cần clode repository đó về máy:

**git clone url**

B2: Tạo nhánh mới:

**git checkout -b new-branch**

Kiểm tra danh sách các nhánh:

**git status**

B3:Add các file từ Working vào staging index:

**git add file-name**

**//thêm các file trong 1 thư mục**

**git add folder-name/**

**//thêm tất cả các file mới**

**git add \***

B4:Thực hiện commit

**git commit -m”message”**

B5:Pull request từ master về

git **pull origin master**

B6:Có thể gặp các conflict , fix conflict rồi commit lại

B7:Push lên nhánh hiện tại

**git push origin name-branch**

**HTML, CSS**

1. Chuẩn BEM trong đặt tên class CSS

BEM là viết tắt của Block-Element-Modifier, là một tiêu chuẩn quy ước đặt tên cho các tên lớp CSS. BEM giúp cho việc code Frontend dễ đọc và dễ hiểu hơn, dễ làm việc và dễ mở rộng cũng như bảo trì khi làm việc với CSS.

BEM là viết tắt của từ Block, Element, Modifier.

Quy tắc đặt tên:

.block {} /\* Block \*/

.block\_\_element {} /\* Element \*/

.block--modifier {} /\* Modifier \*/

1. Các thẻ HTML cơ bản

-mark: đánh dấu bằng cách bôi nền màu

- del: gạch giữa

- ins: gạch chân

-sub: chỉ số dưới

- sup: chỉ số trên

-iframe: sử dụng hiển thị một trang web trên một trang web

EX:

<iframe src="demo\_iframe.htm" name="iframe\_a" height="300px" width="100%" title="Iframe Example"></iframe>

<p><a href="https://www.w3schools.com" target="iframe\_a">W3Schools.com</a></p>

-link : xác định mối quan hệ giữa tài liệu hiện tại và tài nguyên bên ngoài

Sử dụng BEM để viết HTML:  
Với cách đặt tên class này, ta có Block sẽ đại diện cho một component, và trong ví dụ ở đây, là một button .btn. Block cũng sẽ đóng vai trò là một parent mà trong nó sẽ có một hoặc nhiều hơn Element con liên quan. Tên class cho Element và mối quan hệ của nó với Block sẽ được diễn tả bằng tên của Block, tiếp theo là hai gạch dưới, và cuối cùng là tên của Element .btn\_\_price. Thành phần thứ ba của BEM là các Modifier mà chúng sẽ giúp điều chỉnh các trạng thái hoặc phái sinh khác của Block / Element. Tên của Modifier sẽ được nối với tên Block / Element phía trước bởi 02 gạch ngang .btn--orange.

EX:

<a class="btn btn--big btn--orange" href="http://int3ractive.com">

<span class="btn\_\_price">$9.99</span>

<span class="btn\_\_text">Subscribe</span>

</a>

Trước tiên, ký pháp BEM giúp người mới tham gia dự án dễ dàng phát hiện ra các trạng thái và các đối tượng con của một component đã được viết sẵn. Điều này giúp tránh cho họ phải viết lại những kiểu CSS đã có sẵn và hạn chế việc viết thừa code hoặc trùng kiểu CSS, điều mà rất hay xảy ra trong dự án lớn có nhiều người tham gia.

Thứ hai, chỉ cần đọc HTML, bạn vẫn có thể nhanh chóng nắm được các thành phần phụ thuộc lẫn nhau. Trong ví dụ trên, bạn dễ dàng nhìn thấy .btn\_\_price phụ thuộc vào .btn mặc dù bạn chưa biết vai trò cụ thể của nó ngay lập tức.

Thứ ba, với ký pháp BEM, mọi định nghĩa chỉ có một cấp class và không lồng cấp. Điều này giúp cho độ ưu tiên (specificity) chung của hệ thống CSS thấp. Đây là một lợi thế vì sau này bạn không phải “chiến đấu” với specificity của những thuộc tính đã có sẵn (VD: siêu lồng cấp .a .b .c .d .e {...}) cũng như vận dụng những kỹ thuật không hay để thay thế được style (chẳng hạn !important hay inline CSS).

Quy luật thác nước cascading của CSS là con dao hai lưỡi: nó giúp dễ dàng định nghĩa những thuộc tính và kiểu chung trên những selector tổng quát mà không cần phải khai báo lặp lại trên từng phần tử, nhưng nếu không nắm được tầm ảnh hưởng, lập trình viên CSS rất dễ gây ra những tác động phụ đến các đối tượng không liên quan khi chỉnh sửa trên những class có sẵn hoặc thậm chí viết mới. Với ký pháp BEM, lập trình viên sẽ tự tin hơn khi bắt tay chỉnh sửa hoặc viết thêm style vì đã biết rõ tầm ảnh hưởng của selector mà mình đang viết ra.

Tóm lại, ký pháp BEM, nếu áp dụng triệt để, sẽ giúp cải thiện sự phối hợp giữa các thành viên trong nhóm. Ngoài ra, nó buộc người viết CSS phải đầu tư suy nghĩ về việc xây dựng những component độc lập và tái sử dụng được (phù hợp với tiêu chí của

CSS selector và combinator

1. Các CSS selector

- File HTML thì có rất nhiều thẻ giống nhau và thông thường chúng ta sẽ đặt các ID, class cho các thẻ để phân biệt, vậy thì trong CSS sẽ dựa vào các ID và class đó để truy xuất tới và cách truy xuất đó ta gọi là selector.

+ selector phân cấp: Dựa trên cha để tìm con: các thẻ lồng nhau

+ selector ID

+ selector Class

Note:  
div#main: thẻ div có ID là main

div #main: thằng con của thẻ div có ID là main

1. Combinator

* Space: select những phần tử con cháu của phần tử đứng trước
* Child combinator(>): select những phần tử con của phần tử đứng trước
* Adjacent sibling combinator (+): Select phần tử nằm liền kề và ngang hàng với phần tử khác (các phần tử "anh em" kề nhau).
* General sibling combinator (~): Select phần tử ngang hàng với phần tử khác (các phần tử "anh em").

1. Pseudo-class

: first-child: phần tử đầu tiên

:not(p): ngoại trừ phần tử p

[attribute^=value]: select các phần tử có attr bắt đầu bằng value

[attribute$=value]: tất cả các phần tử có attr kết thúc bằng value

[attribute\*=value]: tất cả các phần tử có giá trị đặc biệt là value

tag: first-of-type: phần tử tag đầu tiên của cha nó

tag:last-of-type: chọn thành phần tag là con cuối cùng hoặc duy nhất trong các thành phần cha.

tag:only-of-type: chọn thành phần tag là thành phần con duy nhất( là thẻ đấy, có thể có nhiều thành phần cha)

tag: only-child: nó là con duy nhất của cha nó (không chứa các con khác kể cả loại thẻ khác)

tag:nth-child(n): thành phần con thứ n của cha nó, tính từ thẻ đầu

tag:ntl-last-child: giống cái trên nhưng tính từ cuối

":enabled" chọn thành phần được kích hoạt (enabled), thường sử dụng cho các thành phần của form: thành phần k bị disabled

::selection: css cho các phần tử đã được chọn

RESPONSIVE

1. Bố cục linh hoạt: Sử dụng kích thước theo % để tự động resize theo kích thước màn hình
2. Media Queries: Media Queries được xây dựng như một cách mở rộng các đối tượng. Nó giúp chúng ta đặt các kiểu style khác nhau cho đối tượng trên các trình duyệt và tỷ lệ khung hình khác nhau.

Cách sử dụng:  
Cách thức sử dụng Media Queries

Có một vài cách bạn có thể sử dụng Media Queries.

* Sử dụng trực tiếp cú pháp @media ở trong fle style sẵn có.

EX: @media all and (max-width: 1024px) {

.hero { width: 100%; }

}

* Tạo 1 file chứa Media Queries riêng và @import nó vào CSS.

EX: @import url(styles.css) all and (max-width: 1024px) {...}

* Gắn riêng mỗi file style trên các khung hình khác nhau vào qua HTML.

EX: <link href="styles.css" rel="stylesheet" media="all and (max-width: 1024px)">

1. Media linh hoạt

Hình ảnh, video hay audio cần phải có khả năng thay đổi kích thước cũng như tỷ lệ thì mới đáp ứng tiêu chuẩn Responsive Web.

Một cách nhanh nhất là đặt thuộc tính: max-width: 100%. Làm như vậy sẽ đảm bảo bất kỳ media nào sẽ nhanh chóng chuyển xuống bám theo vị trí của khung container.

Sử dụng SASS

file Sass không được hiểu bởi trình duyệt, vì vậy ta cần phải biên dịch sang CSS trước khi đặt vào web. Đó là lý do tại sao bạn cần một số công cụ để giúp bạn biên dịch .scss thành .css

1. Biến

Khai báo: $name: value;

Sử dụng $name

1. MIXINS:có thể hiểu như là xây dựng hàm để tạo ra các CSS

Khai báo: @mixin function\_name($arg1, $arg2…){

}

Sử dụng:

div{

@include function\_name(value1,value2…);

}

1. EXTENDS: kế thừa, sử dụng lại các class đã được định nghĩa và muốn thêm các thuộc tính mới

div{

@extend .name\_class;

new\_attr: value;

}

1. Nesting: cấu trúc lồng nhau chặt chẽ
2. Functions: hỗ trợ các hàm được định nghĩa sẵn, gần giống như mixins nhưng mixin thì k trả về còn function có trả về

Một số attr nổi bật của CSS

* position:static,relative, absolute,fixed,sticky

+ sticky là kết hợp giữa fixed và relative:khi scroll đến vị trí của nó thì sẽ là fixed, khi scroll lên ra khỏi nó thì sẽ là relative

* transform: dùng đễ tùy chỉnh hình dạng và kích thước của phần tử

+ rotate(n deg): quay n độ, 1 turn = 360 deg

+ scale(value): co giãn kích thước \*value. scaleX, scaleY

+ skew(n deg) : kéo nghiêng

+ transform-origin: x y; dịch chuyển hình dạng theo cách chuyển động của transform

* transitions cung cấp một cách để điều khiển tốc độ của hiệu ứng khi thay đổi các thuộc tính của CSS.

+ transition-property: các thuộc tính CSS sẽ được áp dụng nếu nó thay đổi

+ transition-duration: thời gian thay đổi

+ transition-timing-function: hàm thay đổi

+ transition-delay: delay time

1. Animation

1. @keyframes: dùng để thiết lập một chuyển động.

EX:

@keyframes wobble {

0% {

left: 0px;

}

10% {

left: 50px;

}

20% {

left: 25px;

}

30% {

left: 100px;

}

40% {

left: 50px;

}

}

2. animation-duration :thiết lập khoảng thời gian thực thị 1 chuyển động animation

3. animation-timing-function : xác định tốc độ chuyển động của một animation sẽ như thế nào.

4 . animation-delay : xác định độ trễ của mỗi lượt chuyển động.

5. animation-iteration-count: thiết lập số lần thực hiện một animation.

6. animation-direction : xác định xem chiều chạy của animation sẽ như thế nào

7. animation-fill-mode: xác định trạng thái của một animation, khi mà animation không được chạy (có thể là animation này đã chạy xong hoặc đang bị delay): sau khi thực hiện xong có muốn trả về ban đầu k

8. animation-play-state: xác định trạng thái của animation. (ví dụ: pause)

II. 2 pseudo-element ::after và ::before

* ::before sẽ tạo ra một phần tử giả luôn luôn là element con đầu tiên của element được sử dụng ::before.
* Tương tự đối với ::after nhưng phần tử giả lúc này luôn luôn là element con cuối cùng.

Biến trong CSS và sử dụng các hàm tính toán

1. Biến trong CSS

Khai báo: sử dụng - - var\_name:value.

Muốn biến là toàn cục thì khai báo trong :root

Sử dụng biến: var(var\_name, default\_value)

1. Các đơn vị tương đôis trong CSS

-em : tính theo font-size của cha: 1em=font\_size of parent

-rem : tính theo font-size của thẻ gốc html

- vh và vw: là phần trăm của màn hình

- vmin và vmax: 1% của khung nhìn nhỏ nhất và lớn nhất

-ex: 1ex = với chiều cao của 1 chữ x (in thường) của font hiện tại

-ch: 1ch = chiều rộng của số 0

III. Các hàm tính toán trong CSS

calc: EX cacl(100% - 30px);

Dàn trang với CSS Flexbox và CSS GRID

1. Flexbox

Flexbox là một kiểu dàn trang (layout mode) mà nó sẽ tự cân đối kích thước của các phần tử bên trong để hiển thị trên mọi thiết bị.

Các attr hỗ trợ flexbox:

* display: flex;
* Flex-direction:row, column: hiển thị theo hang hoặc cột
* flex-wrap: wrap : tự căn chỉnh để bao bọc đủ container( có thể xuống dòng)
* order: 1,2,3,… điều chỉnh thứ tự của các item
* flex-grow: lấy các phần dư rồi lấy kích thước theo tỷ lệ
* justify-content: để điều chỉnh vị trí bắt đầu

+ flex-start, flex-end

+center: bắt đầu sao cho các phần tử ở chính giữa

+ space-between: sao cho các phần tử cách đều nhau

+ space-arround: các phần tử cách đều nhau và có thêm cách ở đầu và cuối

1. CSS GRID

* display:gird , inline-gird
* grid-template-rows: 50px 50px 100px; hàng 1,2 có cao 50px , hàng 3 cao 100px;
* grid-template-columns: 90px 90px 120px; cột 1,2 rộng 90px, cột 3 rộng 120px;

\* Có thể dùng đơn vị fr để đặt tỉ lệ cho kích thước so với kích thước của toàn container:1 ft 2fr 1 fr item 1,3 rộng 25% item 2 rộng 50% kích thước của container

VUEJS

1. Cài đặt và khởi tạo project

#Cài đặt vue-cli

npm i -g vue-cli

# Tạo project

vue init webpack(webpack-simple) my-project;

# Di chuyển vào project

cd my\_project

# Run project

npm run dev

1. Các kiến thức cơ bản về VueJS
   1. Attribute của VueJS

* v-bind: binding dữ liệu 1 chiều
* v-model: binding dữ liệu 2 chiều
* v-on: lắng nghe sự kiện
* v-if & v-else-if & v-else : điều kiện
* v-show: tương tự v-if nhưng render ra nhưng ẩn đi
* v-for: chạy vòng lặp
* v-html: hiển thị html
* v-once: sẽ không thay đổi khi thuộc tính thay đổi
  1. Computed property và watcher
* computed: các hàm bên trong sẽ chạy lại khi các thuộc tính có trong hàm thay đổi
* watch: theo dõi sự thay đôi của một thuộc tính
  1. Dynamic Component: cho phép lựa chọn component muốn hiển thị

Cú pháp: <component is=”currentComponent” />

Sử dụng thuộc tính(binding) v-bind:is

\* Thêm <keep-alive> bao quanh thẻ <component /> để giữ trạng thái của các component khi switch

* 1. Async Component: render component theo đồng bộ

EX:

const AsyncComponent = () => ({

// The component to load (should be a Promise)

component: import('./MyComponent.vue'),

// A component to use while the async component is loading

loading: LoadingComponent,

// A component to use if the load fails

error: ErrorComponent,

// Delay before showing the loading component. Default: 200ms.

delay: 200,

// The error component will be displayed if a timeout is

// provided and exceeded. Default: Infinity.

timeout: 3000

})

* 1. Provide và inject: cách truyền dữ liệu từ các component cha xuống các component con cháu mà không phải theo từng bậc
* provide: định nghĩa ở component cha, nơi cung cấp dữ liệu.

EX:

provide: function () {

return {

name: object or function

}

}

* inject: định nghĩa ở component con, sử dụng dữ liệu được cung cấp từ component cha

EX:

inject: ['name']🡪 sử dụng như props

* 1. Các trình lắng nghe sự kiện
* Simple : $emit thì lắng nghe bằng v-on
* Beside :
* Lắng nghe sự kiện : $on(eventName, eventHandler)
* Lắng nghe sự kiện một lần: $once(eventName, eventHandler)
* Dừng lắng nghe sự kiện : $off(eventName, eventHandler)
  1. Transition cho enter/leave: áp dụng các hiệu ứng cho các phần tử được thêm vào, thay đổi, gỡ bỏ
* Sử dụng: bao bọc phần tử cần có hiệu ứng bằng thẻ <transition name=”name” > 🡪 name đặt để tý css cho transiton
* Các class transition:
* v-enter: Trạng thái bắt đầu của enter. Được áp dụng trước khi phần tử được thêm vào DOM và gỡ bỏ đi một frame sau đó.
* v-enter-active: Trạng thái active của enter. Được áp dụng trong suốt quá trình enter, từ ngay sau khi phần tử được thêm vào DOM cho đến khi transition/animation kết thúc. Class này có thể được dùng để định nghĩa duration, delay, và hàm easing cho transition enter.
* v-enter-to: 2.1.8+. Trạng thái kết thúc của enter. Áp dụng một frame sau khi element được thêm vào DOM (cùng lúc với việc v-enter được gỡ bỏ), gỡ bỏ đi khi transition/animation kết thúc.
* v-leave: Trạng thái bắt đầu của leave. Được áp dụng ngay khi một leave transition được kích hoạt và gỡ bỏ đi một frame sau đó.
* v-leave-active: Trạng thái active của leave. Được áp dụng trong suốt quá trình leave, từ khi transition được kích hoạt cho đến khi transition/animation kết thúc. Class này có thể được dùng để định nghĩa duration, delay, và hàm easing cho leave transition
* v-leave-to: 2.1.8+. Trạng thái kết thúc của leave. Áp dụng một frame sau khi leave transition được kích hoạt (cùng lúc với việc v-leave được gỡ bỏ), gỡ bỏ đi khi transition/animation kết thúc.
  1. Mixin: giống như kế thừa, sử dụng để tái sử dụng lại các component
* Sử dụng :mixins:[name]
* Nếu data bị trùng nhau sẽ sử dụng data của component
* Những hàm hook trùng tên nhau thì sẽ được thêm vào mảng để tất cả các hàm đều được gọi, hàm tromg mixin sẽ được gọi trước
* Những tùy chọn có giá trị là object như methods, components và directives sẽ được merge vào chung một object. Nếu khóa (key) của các tùy chọn trùng nhau thì tùy chọn của component sẽ được ưu tiên
  1. Directive tùy biến: tạo ra các tùy biến của riêng mình( v-my\_directive)
* Đăng kí directive toàn cục:

EX: Đăng kí v-focus để auto focus

// Đăng kí một directive tùy biến cấp toàn cục với tên là `v-focus`

Vue.directive('focus', {

// Khi phần tử liên quan được thêm vào DOM...

inserted: function (el) {

// Ta gán focus vào phần tử đó

el.focus()

}

})

* Đăng kí directive cục bộ:

EX:

directives: {

focus: {

// định nghĩa cho directive

inserted: function (el) {

el.focus()

}

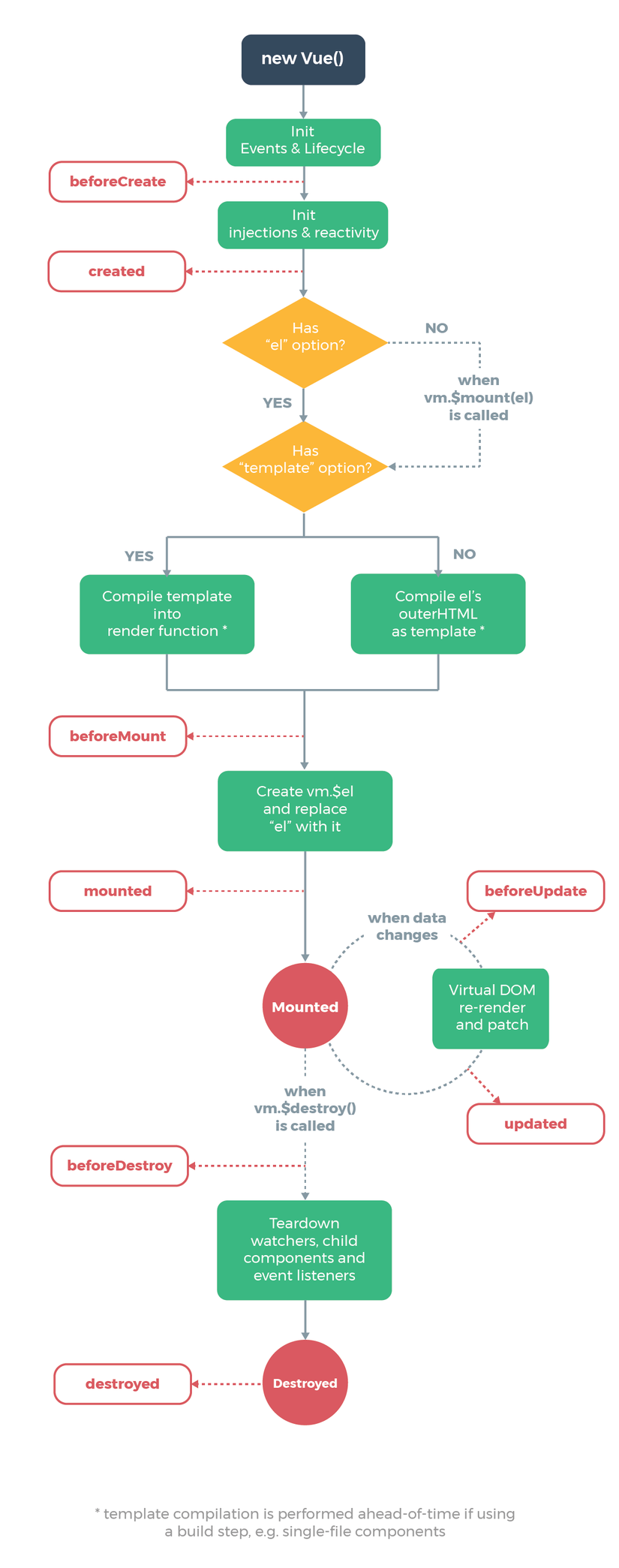
}

}

* 1. Lifecycle
* Creating:
* beforeCreate: chạy trước khi component được khởi tạo
* created: chạy khi component đã được tạo (thường dùng để fetch data từ server) = (componentWillMount)
* Mouting:
* beforeMount: gọi khi component đã được compile và trước lần render đầu tiên
* mounted: gọi khi component đã được render( hiển thị trên màn hình)
* Updating:
* beforeUpdate: được gọi sau khi dữ liệu trên component bị thay đổi và trước khi re-render
* updated: khi component đã được re-render , DOM đã được cập nhật
* Destroy:
* beforeDestroy: được gọi trước khi component bị huỷ đi (ví dụ như lúc chúng ta chuyển từ component này sang component khác, hay như lúc ta chuyển route,...). Tại đây component vẫn còn đầy đủ những yếu tố như data, events,... Thường dùng hàm này để xoá đi các sự kiện không cần thiết sau khi component bị huỷ (= componentWillUnmount)
* destroyed: component đã bị hủy

**##Note:**

* created: thường dùng để gọi API lấy dữ liệu từ server, khởi tạo websocket, lắng nghe event Laravel Echo,... miễn là ta không động gì vào DOM thật là được
* mounted: thường dùng khi ta muốn sử dụng JQuery hoặc truy vấn tới 1 phần tử HTML cụ thể, ví dụ: document.getElementById('id'),...
* beforeDestroy: thường dùng khi ta muốn huỷ lắng nghe các sự kiện: như sự kiện onscroll, hay các sự kiện lắng nghe socket.io, larave-echo,...



1. Component && Communicating Event
2. Props

* Props cho phép truyền data từ parent component xuống children component.\*\*\*,
* Lưu ý: Khi dữ liệu thay đổi ở component cha, các props mà ta khai báo ở các component con cũng tự động update theo.
* Template syntax:
* **Tại component parent:**

**<my-component v-bind:prop="parentValue"></my-component>**

* Tại component con:

props:[‘prop’]

* + - * + Props validation

props:{

propName:{

type: typeName,

default: value  
 }  
}

or Hoặc cũng có thể tự custom validate cho riêng mình bằng cách sử dụng **validator** key trong object của prop bạn cần ràng buộc.

EX:

Vue.component('ex',{

props: {

point: {

**validator: function (value) {**

**return value > 10;**

**}**

}

}

});

1. Custom Events

* Component cha truyền dữ liệu xuống cho thằng con thông qua props
* Component con thông báo cho component cha thông qua các event
* Component con thông báo cho thằng cha thông qua :$emit

EX: vm.$emit(‘eventName’, data}

* Component cha lắng nghe sư kiện mà component con phát ra thông qua v-on or $on

EX: <child v-on:counter="countPlus"></child>

1. Slot

* Thẻ <slot /> sẽ đại diện cho phần dữ liệu nằm giữa hai thẻ con trên component cha để component con có thể sử dụng lại

1. Vue Routing & Navigation
2. Vue Router

* Sử dụng thư viện :vue-router
* Các bước tạo router:
* B1: Tại file src/router/index.js: Định nghĩa và khai báo các router

Vue.use(Router)

export default new Router({

routes: [

{

path: '/loi',

name: 'HelloWorld',

component: HelloWorld

}

]

})

* B2: Thêm router vừa định nghĩa vào phần khai báo trong thư mục **main.js**

Import router from “./router”

new Vue({

el: '#app',

**router,**

components: { App },

template: '<App/>'

})

* B3: Hiển thị các router Component dựa trên các router sử dụng <router-view />
* B4: Sử dụng <router-link> để chuyển qua lại giữa các navigation

<div id="app">

<nav>

<router-link to='/'>Home</router-link>

<router-link to='/register'>Register</router-link>

<router-link to='/login'>Login</router-link>

</nav>

<router-view />

</div>

1. Router động

* Path: { path: '/student/:id', component: Student },🡪 giống nodeJS
* Lấy tham số truyền: $router.params.paramName

1. Redirecting trong ứng dụng

* vm.$router.push(‘path’)

**## Note:**

* Có thể thêm name của router để dễ ghi nhớ
* <router-link to=”{name=’routerName’}”
* Bật tắt chế độ hash router:
* Thêm mode=”history” trong lúc khai báo Router

1. Axios và API call

* Hoạt động tương tự như fetch có một số cải tiến so với fetch được thể hiện qua từng API : GET, POST,..

1. GET :lấy dữ liệu từ server về

axios.get("url")

.then(response => console.log("response", response.data))

* Fetch sẽ cần tới 2 promise mới trả ra được response
* Axios cần 1 lần promise

1. POST : đẩy dữ liệu nên server

axios.post("url", {

a:”b”,

c:1

})

.then(response => console.log(response.data))

.catch(error => console.log(error));

* Fetch ghi post dữ liệu nên thì cần convert data sang JSON thông qua phương thức JSON.stringify() còn Axios thì không cần thiết, có thể truyền trực tiếp lên.

1. PUT : sửa dữ liệu trên server

axios.put(‘url’);

1. DELETE : xóa dữ liệu trên server

Axios.delete(‘url’)

1. Tương tác vơi database

* Tạo tài khoản firebase
* Tạo project trên firebase rồi lấy config
* Sử dụng vuefire: $ npm install firebase vuefire --save
* Khai báo sử dụng vuefire trong file main.js
* Import Vuefire from ‘vuefire’
* Vue.use(Vuefire)
* Kết nối với project trên firebase thông qua các tham số vừa lấy(config):
* let config = {
* //do thing
* };
* let app = Firebase.initializeApp(config)
* let db = app.firestore()
* db.collection(‘collectionName').get()
* .then(querySnapshot=>{
* querySnapshot.forEach(doc=>{
* tmp.push(doc.data());
* })
* })
* Set:

/ Add a new document in collection "cities"

**db.collection("cities").doc("LA").set({**

**name: "Los Angeles",**

**state: "CA",**

**country: "USA"**

**})**

**db.collection('cities').doc('BJ').set({**

**capital: true**

**}, { merge: true });**

* Get:

**db.collection("cities").doc("SF")**

**.get()**

**.then(function(doc) {**

**if (doc.exists) {**

**console.log("Document data:", doc.data());**

**} else {**

**// doc.data() will be undefined in this case**

**console.log("No such document!");**

**}**

**}).catch(function(error) {**

**console.log("Error getting document:", error);**

**});**

* Add:

**// Add a new document with a generated id.**

**db.collection("cities").add({**

**name: "Tokyo",**

**country: "Japan"**

**});**

* Update:

**// Create an initial document to update.**

**var frankDocRef = db.collection("users").doc("frank");**

**frankDocRef.set({**

**name: "Frank",**

**favorites: { food: "Pizza", color: "Blue", subject: "recess" },**

**age: 12**

**});**

**// To update age and favorite color:**

**db.collection("users").doc("frank").update({**

**"age": 13,**

**"favorites.color": "Red"**

**});**

1. Authen, Author

**Có 2 cơ chế chính để thực thi việc xác thực 1 client trong REST API:**

* Bearer Token
* Authentication cookie

1. Bearer Token

* Bearer token là giá trị được truyền vào Authentication header của 1 HTTP Request. Nó không được lưu tự động, không hết hạn và không gắn liền với domain. Nó chỉ là 1 giá trị.
* Để có 1 ứng dụng phi trạng thái, chúng ta thường sử dụng JWT cho việc tạo ra token. Hiểu đơn giản, JWT gồm 3 phần:
* Header
* Payload và expiration time (optional)
* Signature
* JWT là một mật mã an toàn, trong quá trình trao đổi thông tin khiến việc authentication trở nên stateless. Signature sẽ chứng thực cho payload không bị chỉnh sửa bằng các thuật toán symmetric hoặc asymmetric (RSA). Header chứa thông tin public key để xác minh Signature. Ứng dụng phía Client, lần đầu request lên server, sẽ lấy về một JWT token thông qua việc đăng nhập username, password. Sau đó, thông qua Javascript để đính kèm JWT token bên trong mỗi HTTP header. Server sẽ xác thực signature tương ứng với payload, nếu chúng trùng nhau, chúng ta có thể tin tưởng nội dung của Payload.

1. Authentication cookie

* Cookie là một cặp name-value, được lưu trên web browser và chúng có thời gian hết hạn, gặp liền với một domain. Cookie được tạo bởi client browser bằng Javascript.

EX: **document.cookie = 'my\_cookie\_name=my\_cookie\_value'**

hoặc từ Server thông quan HTTP Response header:

**Set-Cookie: my\_cookie\_name=my\_cookie\_value**

* Web browser tự động gửi cookie kèm mỗi request tớicookie domain
* Thông thường, Cookie được sử dụng để lưu một Session ID. Session ID được quản lý bởi server. Ở đây, mình đang nhắc đến một stateful app nơi mà server cần quản lý trạng thái trên server trong khi JWT token là stateless.

1. VUEX

* Vuex như là một ‘kho lưu trữ’ các “state”.
* Khi các state của store thay đổi, thì các state được các component sử dụng sẽ tự cập nhật
* Cài đặt:

**npm i vuex –save**

* Tạo store:

**Vue.use(Vuex) // sử dụng Vuex cho project**

**const store=new Vuex.Store({**

**//do thing**

**})**

* Khai báo sử dụng store vừa tạo cho project trong file **main.js:**
* **Import store from “…”**

**New Vue({**

**el:#app,**

**store,**

**…,**

**Render:h=>(App)**

**})**

1. State

* Các biến để các component có thể dùng chung
* Khai báo:

State:{

stateName:value,

…

}

* Sử dụng tại các component: ***this.$store.stateName***

1. Getter

* Biến đổi state theo ý mình rồi trả kết quả

**Getters:{**

**getName:state=>{**

**return function(state);**

**}**

**}**

* Các component sử dụng: **this.$store.getters.getName**
* mapGetters: tự động map đến các getter có trong store theo tên:

import {mapGetters} from ‘vuex’

mapGetters([getterName,…] ) 🡪 hiển thị : {{getterName}}

* Thường hay sử dụng trong computed:

Computed:{

… mapGetters([getterName,…] ),

computedName:function(){

//do thing }

}

* Cho phép các getter có thể truyền tham số:

**EX: getTodoById: (state) => (id) => {**

**return state.todos.find(todo => todo.id === id)**

**}**

1. Mutations

* Các hàm được khai báo trong mutation được sử dụng để thay đổi state trong store

**mutations: {**

**increment (state) {**

**// mutate state**

**state.count++**

**}**

**}**

* Cách để gọi các mutation**: this.$store.commit('muName', data)**
* mapMutationns: tự động map đến các hàm trong mutations theo tên( tương tự getter)
* Các hàm trong mutations chạy đồng bộ

1. Actions

* Action tương tự như Mutations, nhưng có 2 điểm khác biệt như sau:
* Thay vì trực tiếp thay đổi state,thì action sẽ gọi Mutations để thay đổi state
* Các Actions có thể chứa hoạt động không đồng bộ

**EX:**

**const store = new Vuex.Store({**

**state: {**

**count: 0**

**},**

**mutations: {**

**increment (state) {**

**state.count++**

**}**

**},**

**actions: {**

**increment (context) {**

**context.commit('increment')**

**}**

**}**

**})**

* Các Actions gọi các mutation thông qua: context.commit(‘mutationName’)
* Tương tự, có thể truy cập state và getter thông qua context: context.state và context.getter
* Chúng cũng cho phép chúng ta gọi các action khác trong một action: context.dispatch(‘actionName’)
* Gọi một action từ bên ngoài: store.dispatch(‘actionName’)

1. Module

* Sử dụng một state tree duy nhất, tất cả các state lưu trữ trong một ứng dụng lớn 🡪 khi dự án lớn, sẽ khiến cho store bị phình to và khó quản lý
* Chia thành các module để dễ quản lý, mỗi module gồm state, muations, actions, getters
* Sử dụng:

**const moduleA = {**

**state: () => ({ ... }),**

**mutations: { ... },**

**actions: { ... },**

**getters: { ... }**

**}**

**const moduleB = {**

**state: () => ({ ... }),**

**mutations: { ... },**

**actions: { ... }**

**}**

**const store = new Vuex.Store({**

**modules: {**

**a: moduleA,**

**b: moduleB**

**}**

**})**

**store.state.a // -> `moduleA`'s state**

**store.state.b // -> `moduleB`'s state**

* mapState với module (state hay được lấy ra bằng computed)

**computed:mapState({**

**name: state=>state.moduleName.stateName**

**})**

* mapGetters:

**mapGetters(‘moduleName’,{**

**name1:’nameGetterOfModule’,**

**name2:’nameGetterOfModule’,..**

**}**

* mapAction với module : tương tự như getter
* Gọi action : this.$store.dispatch(‘moduleName/actionName’)

1. Strict Mode

* Khi bất kì một state nào thay đổi bên ngoài các hàm mutations thì sẽ throw ra 1 lỗi
* Sử dung: strict: true

## Note: Khi sử dụng Strict Mode thì không thể thay đổi trực tiếp state từ các handler bên ngoài mutations. Vì vậy v-model sẽ không sử dụng được như bình thường, có 2 cách để xử lý:

* C1: Sử dụng v-bind:value và event change or input
* **C2:** set biến gồm 2 hàm set và get:

**EX:** **message: {**

**get () {**

**return this.$store.state.obj.message**

**},**

**set (value) {**

**this.$store.commit('updateMessage', value)**

**}**

**} 🡪 hàm get lấy value**

* + - * + **Hàm set sẽ gọi mutation để thay đổi state**

1. Đa ngôn ngữ

* Sử dụng thư viện **vue-i18n**
* Cài đặt : npm i –save vue-i18n
* Config để sử dụng:

B1: Tạo các file ngôn ngữ

EX:

**# vn.json**

**{**

**"common": {**

**"select\_lang": "Ngôn ngữ"**

**},**

**"login": {**

**"title": "Đăng nhập vào dibochit",**

**"buttons": {**

**"login": "Đăng nhập"**

**}**

**}**

**}**

B2: Khởi tạo vue-i18n 🡪 tạo instance

**Vue.use(VueI18n) 🡪 sử dụng vue-i18n**

**const messages = {**

**vn: vnMessage, 🡪 link đến các file lang đã định nghĩa trước**

**en: enMessage,**

**}**

**const i18n = new VueI18n({**

**locale: 'vn', // set ngôn ngữ mặc định**

**messages,**

**fallbackLocale: 'vn',**

**})**

* Hiển thị: **$t("Định danh theo kiểu object")**