

Développement d'applications web

Vincent Séguin

Qui suis je?

Ingénieur logiciel

Diplômé de l'Université Laval



Vincent Séguin

Engineering Director

Capdesk from Carta

Capdesk

Pourquoi ce cours?



Passion du **web**



Aucun cours de web à
l'université
(Maintenant oui!)



Web **omniprésent**
Sites web, applications web,
applications desktop,
mobiles, cloud



Navigateurs de plus en
plus **puissants**

Pourquoi ce cours?

But poursuivis



Approfondir les notions
reliées au Web, de la base
jusqu'aux tendances '**on the
edge**'



Travailler sur un projet
concret, vous faire
expérimenter par
vous-mêmes!



Faire de vous des
développeurs **polyvalents**
et prêts à affronter le
marché du travail.

“ *ATTENTION!*

Ce cours explique les notions de base
du développement web, mais
demandera tout de même un peu de
travail!





I Am Developer

@iamdeveloper



Following

Things to try when fixing a bug:

1. Google
2. Stack Overflow
3. Documentation

...

8277. Disturb your co-worker who has headphones in

RETWEETS

3,952

LIKES

1,566



1:36 AM - 1 Feb 2014



4K



1.6K



Format du cours



~3h de cours par
semaine

2 heures de labo
(Les labos **comptent!**)

Labo = Appliquer la
théorie

Pondération



2 Examens

20% chaque



Projet de session

48%



Laboratoires

10%

10 labs

1% chaque

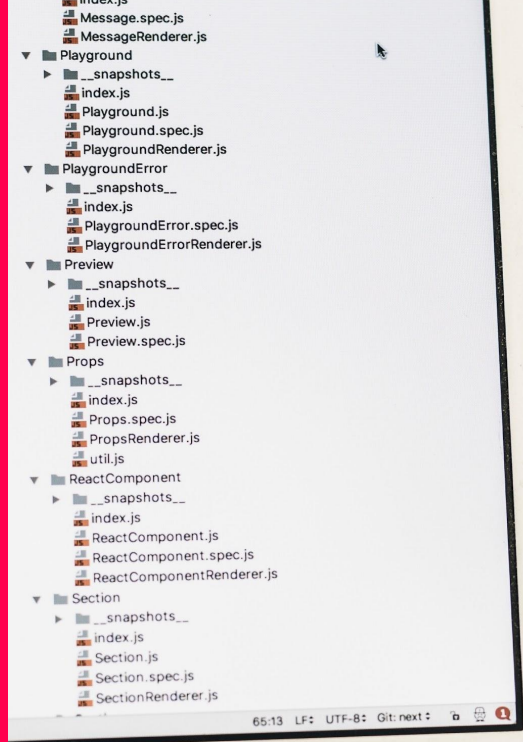


**Évaluation des
pairs**

2%

Possibilité de
perdre jusqu'à **4**
cotes

Projet de session



Outils

Pratique pour développer

- WebStorm (obtenez la [version gratuite](#) en tant qu'étudiant)
- [Visual Studio Code](#)

Pratique pour déboguer

- Console Chrome Dev Tools (voir la [documentation](#) pour plusieurs astuces)
- Mozilla Developer Network (contient toute la [documentation](#) sur JS, CSS, HTML)
- Postman/Insomnia
- [VueJS DevTools](#)

Chapitre 1

Les rudiments

HTML et CSS

Nous ferons un survol rapide du **HTML** et **CSS** car nous considérons ces concepts comme étant simples à apprendre de façon autonome.

Utilisez la documentation sur MDN pour obtenir tous les détails sur ceux-ci.



HTML

À savoir

Bien connaître les balises d'organisation

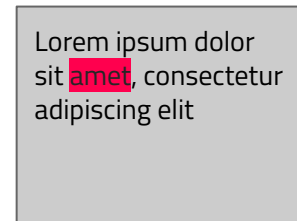
<div/>

Les div sont les éléments les plus courants d'une page web. Ils représentent un simple conteneur.

Un div est **display:block** (implique un saut de ligne) par défaut.



Les span ressemblent aux div, mais sont en **display:inline** (pas de saut de ligne).



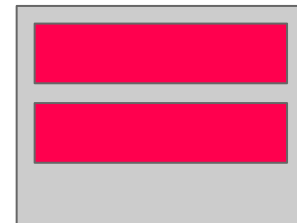
HTML

À savoir

Éléments “block”

`<p> <div> <form> <header> <nav> <h1>`

...



Éléments “inline”

`<a> <i> <cite> <mark> <code>`

...

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit

HTML

À savoir

Bien connaître les balises d'organisation

<p/>

Les **p** sont des balises très semblables aux div, exceptées qu'elles sont faites pour contenir du texte.

HTML valide :

<div>

<p>Hello</p>

</div>

HTML

À savoir

Bien connaître les balises d'organisation

`` et ``

Balises qui devraient exclusivement servir pour des **listes**.
Par défaut, une liste HTML possède des puces.

`<table/>`, `<td/>` et `<tr/>`

Balises qui **devraient** exclusivement servir pour des tableaux.



Les balises **spécialisées** ont des utilisations **spécialisées!**.
Les tableaux/listes sont plus difficiles à utiliser, donc ne pas en abuser...

HTML

À savoir

Bien connaître les balises de mise en forme

``, ``, `<i/>`

Balises de mise en forme de texte. Peut également se faire grâce au CSS.

`<address/>`, `<date/>`, `<phone/>`

Balises très spécifiques de mise en forme possédant déjà du style (et des interactions).

HTML

À savoir

Bien connaître les balises de formulaire

`<form>`

Définit une zone de formulaire. Un form possède le comportement par défaut d'envoyer une requête **POST**...

`<input type="text" name="fname">`

`<input type="date" name="fname">`

`<input type="submit" value="Submit">`

Définit les types de champ d'un formulaire, ainsi que le type d'action possible. Nous y reviendrons...

HTML

À savoir

Autres balises

Balises multimédia : ``, `<audio/>`, `<video/>`

Balises de titre : `<h1/>`, `<h2/>` ... `<h6/>`

Bouton : `<button/>`

Hyperlien : ``

etc... à vous de les découvrir!

helloworld.html

Document de base

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title>My awesome page</title>
    <meta name="description" content="My awesome page">
    <meta name="viewport" content="width=device-width">

    <link rel="stylesheet" href="css/mycss.css">
    <script src="js/myscript.js"></script>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
</html>
```

CSS

À savoir

Le CSS est principalement basé sur des sélecteurs, qui permettent de styliser les balises HTML.

```
<div class="myDiv"></div>
```

se référencera par le sélecteur `.myDiv` en CSS

```
<div id="myDiv"></div>
```

se référencera par le sélecteur `#myDiv` en CSS

CSS

À savoir

Le CSS est principalement basé sur des sélecteurs, qui permettent de styliser les balises HTML.

```
<div class="myDiv"></div>
```

```
<div id="myDiv"></div>
```

Une class devrait être utilisée sur des éléments qui vont se répéter dans une page.

Un id devrait être placé sur une balise unique.

Sélecteurs

Petit guide des sélecteurs

Stylise tous les éléments ayant la classe *foo*

```
.foo {  
    ...  
}
```

Stylise tous les div!

```
div {  
    ...  
}
```

Stylise tous les éléments ayant la classe *bar* dont un parent a la classe *foo*.

```
.foo .bar {  
    ...  
}
```

Sélecteurs

Petit guide des sélecteurs

Stylise tous les éléments ayant la classe *foo* **ET** la classe *bar*

(notez l'absence d'espace entre les 2 classes)

```
.foo.bar {  
    ...  
}
```

Stylise tous les éléments ayant la classe *bar* dont le parent direct possède la classe *foo*

```
.foo > .bar {  
    ...  
}
```

Variations possibles avec classes, id et éléments directs!

```
.foo a  
.foo #bar  
etc...
```


Sélecteurs

Petit guide des pseudo-sélecteurs

Stylise les éléments avec la classe *myClass* lorsque la souris passe dessus

```
.myClass:hover {  
    ...  
}
```

Stylise les liens avec la classe *myClass*, lorsque visités

```
.myClass:visited {  
    ...  
}
```

Stylise le PREMIER élément avec la classe *myClass*.

```
.myClass:first-child(){  
    ...  
}
```

Sélecteurs

Voir : https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Getting_started/Selectors

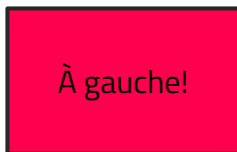
Qu'est-ce qu'on met dans un sélecteur

Tout ce qui est du **STYLE!**

```
.myClass {  
    color : #666666; // Couleur du texte  
    display: inline-block; // Saut de ligne ou  
non  
    float: left; // Gauche, droite  
    background: url('mypicture.jpg')  
no-repeat; // Arrière-plan  
    font-size: 14px; // Grosseur du texte  
    ...  
}
```

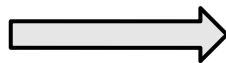
CSS

À savoir - positionnement

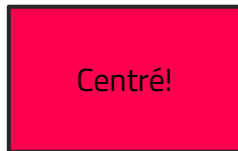


`float : left;`

`float : right;`



`margin-left: auto;`



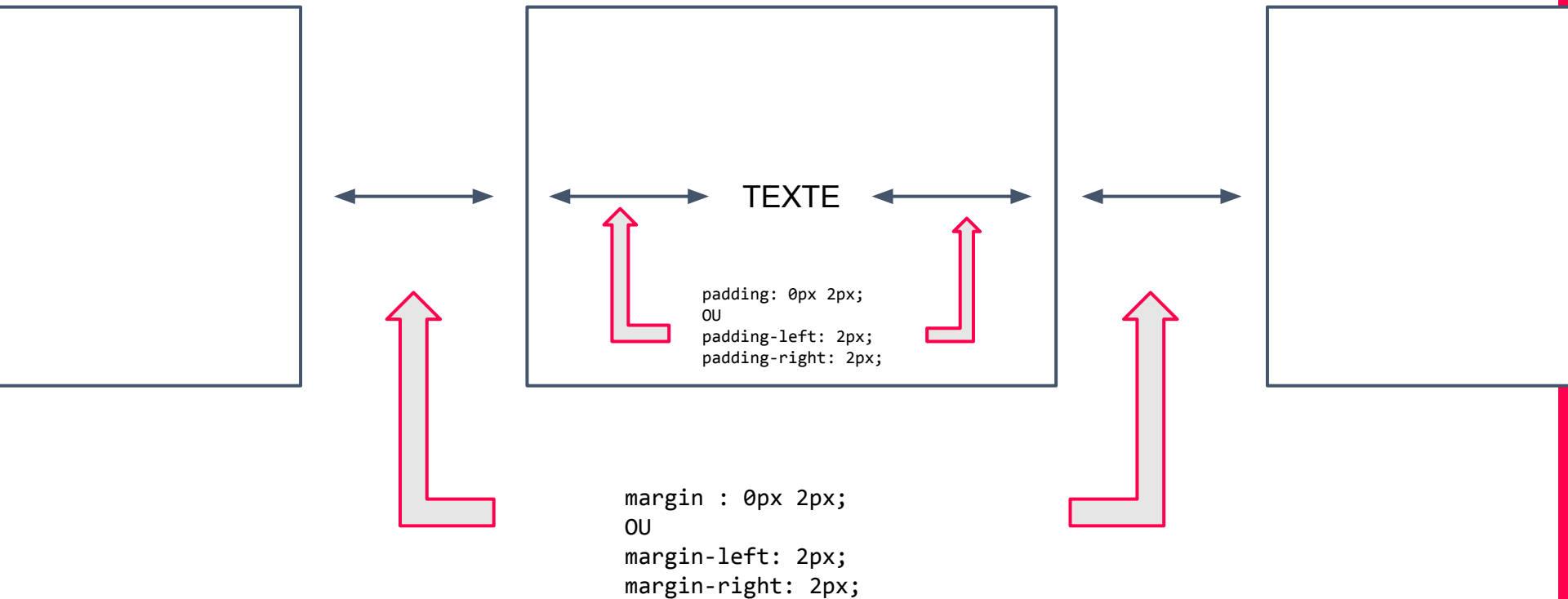
`width:100px;`



`margin-right: auto;`

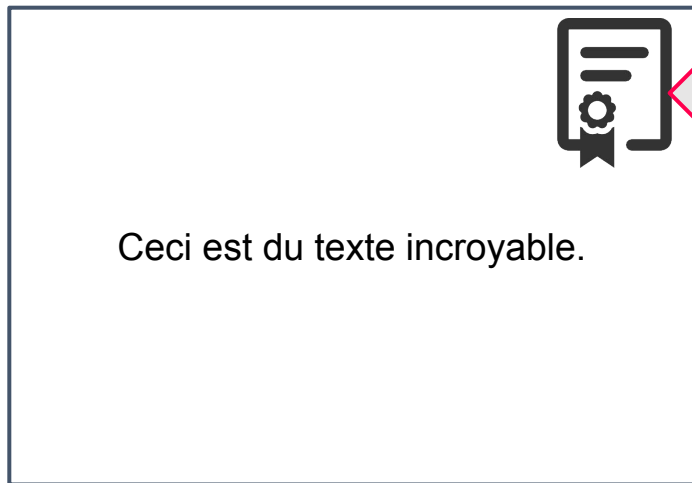
CSS

À savoir - espacement (padding et margin)



CSS

À savoir - positionnement



```
position : absolute;  
right: 0;  
top: 0;
```

```
position : relative;
```

CSS

Héritage et surcharge

Le CSS appliqué sur une seule balise peut être le résultat de plusieurs règles, appliquées en ordre de définition et de précision.

Ainsi, l'élément suivant : `<div class="class1"></div>` pourrait hériter des règles suivantes en ordre :

```
div {           .class1 {           .class1 {           .class1:first-child {  
    ...           ...           ...           ...  
}               }               }               }
```



Le style **inline** est toujours appliqué en dernier.
Possible de forcer l'ordre en utilisant le tag **!important**, mais c'est souvent un mauvais signe...

Voir : <https://developer.mozilla.org/en/docs/Web/CSS/Specificity>

Sites web adaptatifs (Responsive)

Techniques

Le style responsive est de plus en plus important : il s'agit de supporter plusieurs résolutions d'écran avec une même feuille de style!

Tailles en %, rm, rem, vh, vw au lieu de pixels

Les **width**, **padding**, **height**, etc. peuvent s'exprimer en % au lieu de pixels fixes.

Max-width/min-width

Permet de définir des règles de CSS plus complètes qu'une simple largeur fixe.

```
.container {  
  width: 50%;  
  max-width: 1200px;  
}
```

Overflow

hidden, auto, scroll

```
.container {  
  overflow: hidden;  
}
```

Media queries

Permet de surcharger le CSS existant pour des résolutions d'écran données.

Media Queries

Permet de surcharger le CSS existant pour des résolutions d'écran données.

```
@media screen and (max-width: 640px) {  
  .container {  
    width: 600px;  
  }  
}
```

Lorsque l'écran aura une largeur en bas de 640px, les éléments avec la classe container auront une largeur différente!

CSS - Flexbox / Flex

Flex est maintenant bien supporté par la majorité des navigateurs modernes.

Flex permet de simplifier plusieurs affichages autrement **complexe**.

IE	Edge [*]	Firefox	Chrome	Safari	Opera	iOS Safari [*]	Opera Mini [*]	Android Browser [*]	Chrome for Android
			29					1 4.3	
			49					4.4	
			50					4.4.4	
8	13	47	51			9.2			
4 11	14	48	52	9.1	39	9.3	all	51	51
		49	53	10	40				
		50	54	TP	41				
		51	55						

caniuse.com/#feat=flexbox

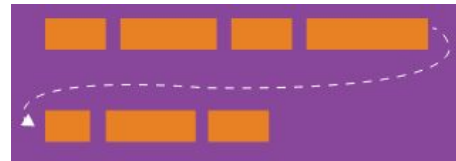
CSS - Élément parent

```
.container {  
  display: flex; /* ou inline-flex */  
}
```



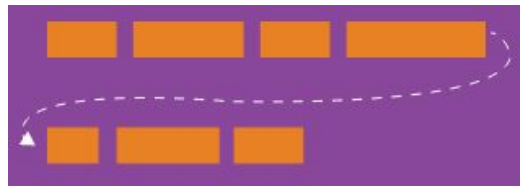
```
.container {  
  flex-direction: row | row-reverse | column |  
                  column-reverse;  
}
```

```
.container {  
  flex-wrap: nowrap | wrap | wrap-reverse;  
}
```



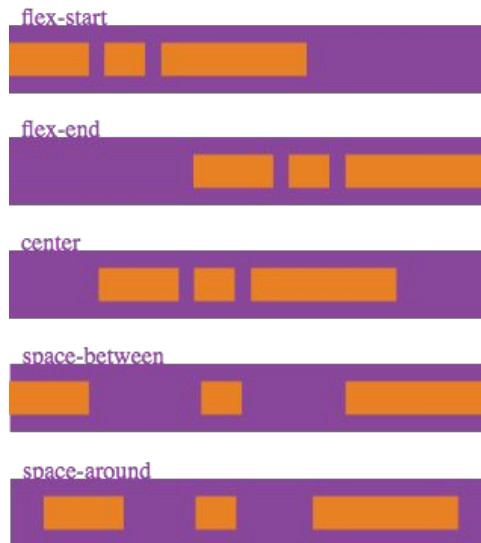
CSS - Élément parent

```
// Combinaison de flex-direction et flex-wrap  
.container {  
  flex-flow: <'flex-direction'> || <'flex-wrap'>  
}
```



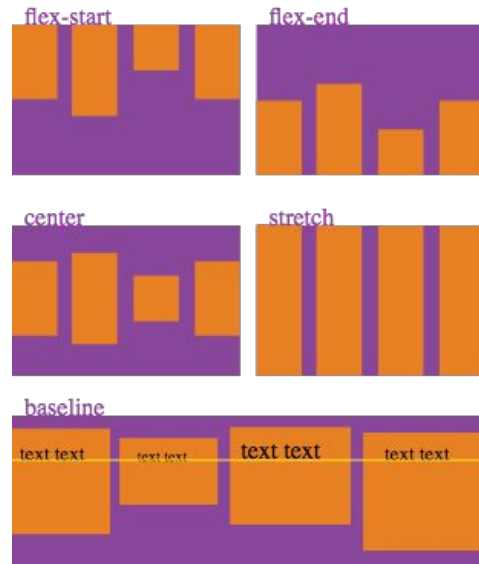
CSS - Alignement horizontal

```
.container {  
  justify-content: flex-start | flex-end |  
                  center | space-between |  
                  space-around;  
}
```



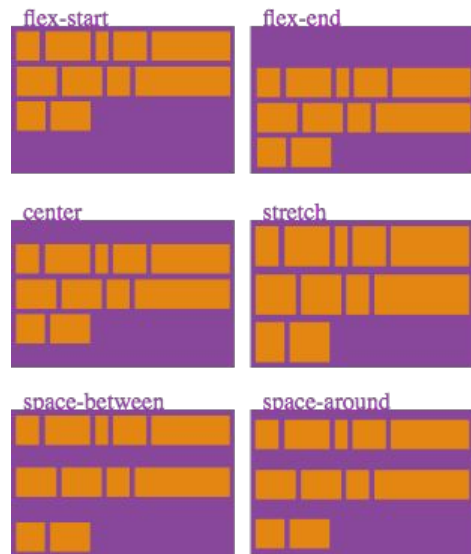
CSS - Alignement vertical

```
.container {  
  align-items: flex-start | flex-end |  
              center | baseline | stretch;  
}
```



CSS - Alignement des enfants

```
.container {  
  align-content: flex-start | flex-end |  
                center | space-between |  
                space-around | stretch;  
}
```



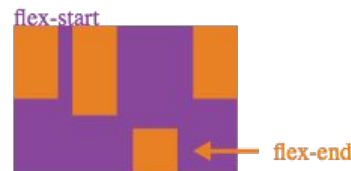
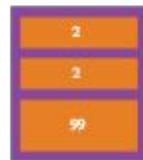
CSS - Éléments enfants

```
.item {  
  order: <integer>;  
}
```

```
.item {  
  flex-grow: <number>; /* default 0 */  
}
```

```
.item {  
  flex-shrink: <number>; /* default 1 */  
}
```

```
.item {  
  align-self: auto | flex-start | flex-end |  
              center | baseline | stretch;  
}
```



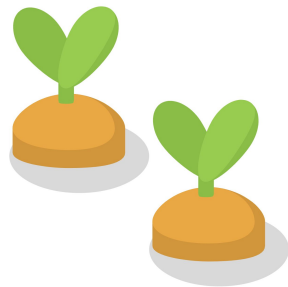
Ressources CSS Flex

- MDN ([Flexible Box Layout](#))
- CSS Tricks ([Complete guide to flexbox](#))
- [Solved by Flexbox](#)
 - Centrer verticalement
 - Footer "sticky"
- [Flexbox froggy](#) (Jeu interactif pour apprendre flex)



Ressources CSS Grid

- MDN ([Grid Layout](#))
- CSS Tricks ([Complete guide to grid](#))
- [Grid garden](#) (Jeu interactif pour apprendre grid)



Autres fonctionnalités utiles...

- **Variables (!)**

```
--main-bg-color: blue;  
  
background-color: var(--main-bg-color);
```

- **Supports queries**

```
@supports (display: flex) { ...
```

- **Nesting**

Frameworks



Bootstrap

<https://getbootstrap.com/>



Bulma

<https://bulma.io/>



Materialize

<https://materializecss.com/>



Semantic

<https://semantic-ui.com/>

À vous de choisir... lisez la documentation avant.

Focusez sur la simplicité!

Chapitre 2

| Architecture client/serveur

| Bases du JavaScript

Plan

- Architecture générale
- Division client/serveur
- JavaScript : types de base et structures de données
- JavaScript : opérations de base
- JavaScript de présentation
- JSON

Distinction importante!

Site web / Page web :

«Ensemble de page(s) pouvant être consulté(es) en suivant des hyperliens.»

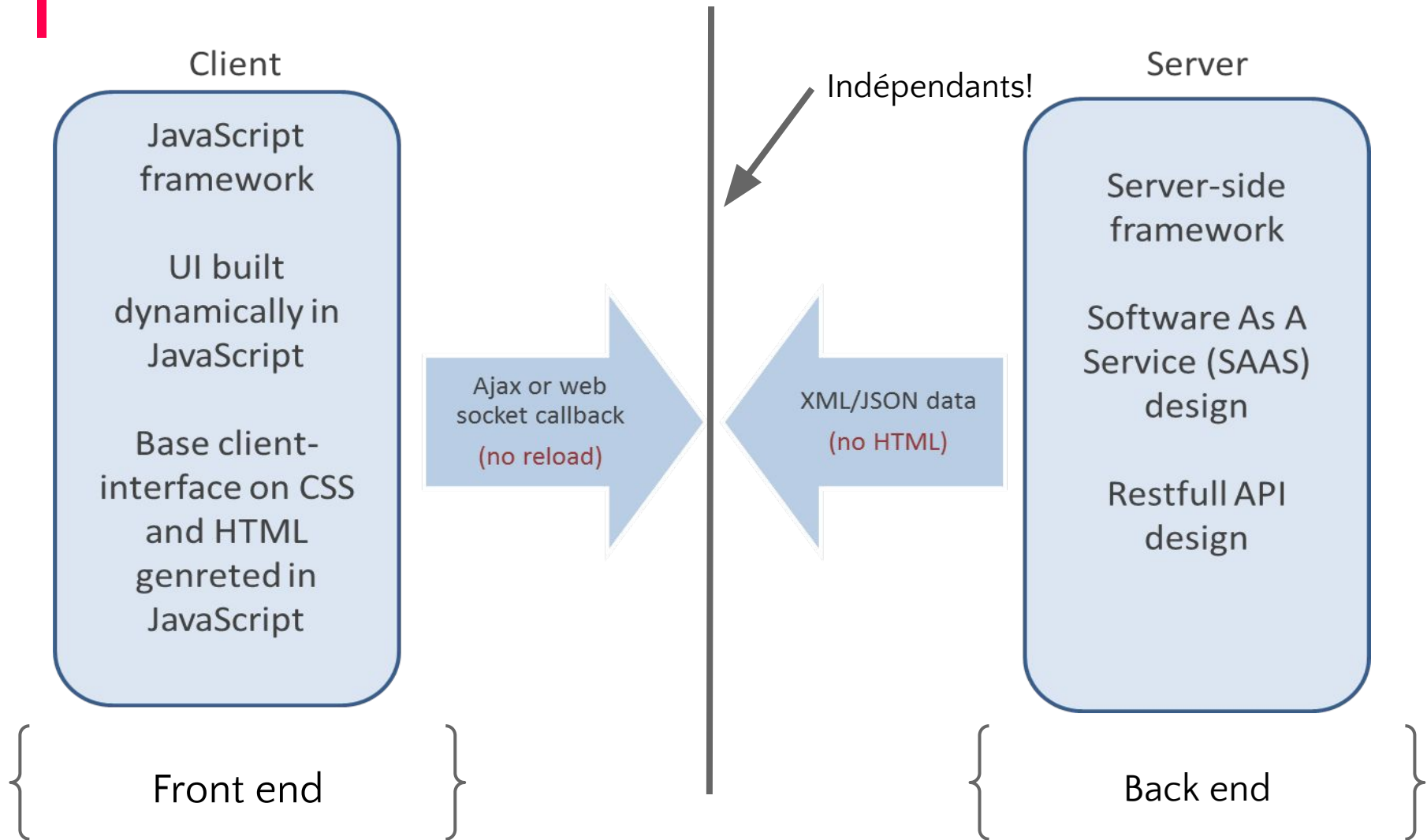
- Construit en HTML, CSS etc.

Application web :

«Logiciel applicatif manipulé grâce à un navigateur Web»

-> C'est ce sur quoi nous allons focuser.

Architecture générale



Architecture générale

Client

JavaScript
framework

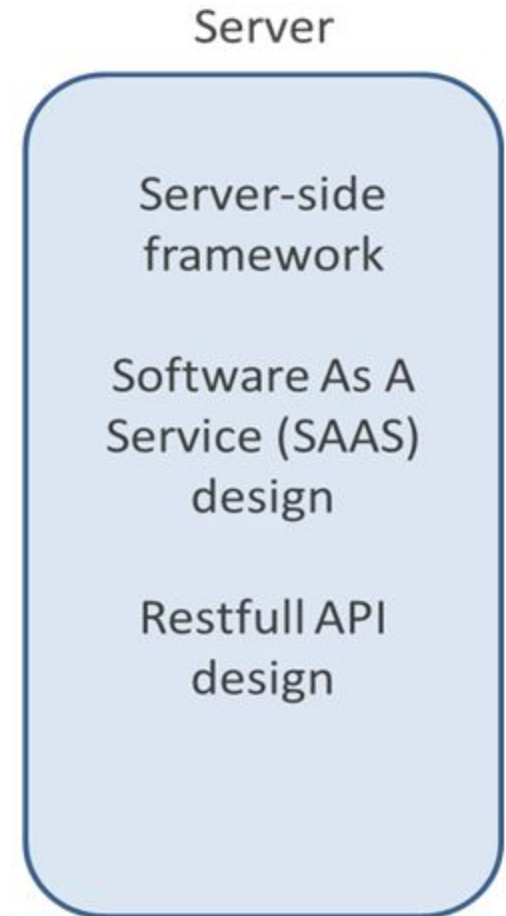
UI built
dynamically in
JavaScript

Base client-
interface on CSS
and HTML
generated in
JavaScript

- Client léger/lourd, selon le type d'application
- Pas de logique DE DOMAINE côté client : logique d'affichage/traitement des données seulement!
- Division **client** : Logique de présentation séparée du traitement des données ou des appels à l'API.

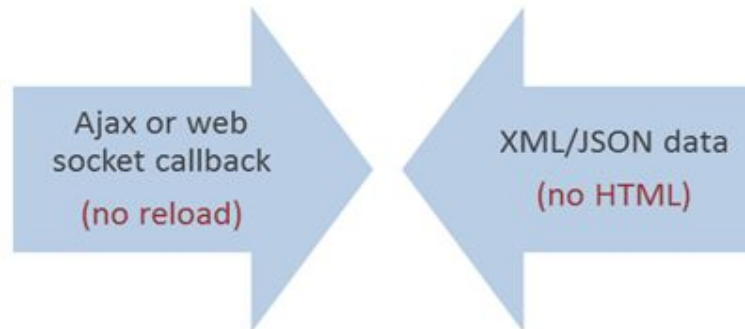
Architecture générale

- Répond aux demandes du client
- A un domaine bien séparé du client, jamais celui-ci ne doit accéder à des objets du serveur!
- Design permettant de supporter n'importe quel client : navigateur ou autre application
- Peut contenir une base de données ou pas : ce n'est pas sensé paraître !



Architecture générale

- Requêtes asynchrones
- Manipule les données fournies par le serveur et lui retourne



- Transforme les objets client en objets du domaine et vice-versa

Front end

Back end

| Client

Voyons le client en
profondeur...

Langages interprétés

Langages qui seront interprétés par le navigateur :

Pas exécuté directement par la machine (comparativement aux langages compilés), mais par un autre programme nommé interprète.

Interprète = **navigateur** dans le cas du Web.

JavaScript: Nous commencerons par les rudiments du langage, pour ensuite exclusivement focuser sur l'**ES6**.

ES6

JavaScript

- Inventé en 1995
- Implémentation de l'ECMAScript (Langage standardisé par l'ECMA, implémenté par le JavaScript, ActionScript et même C++, langage orienté objet)
- Langage destiné à être exécuté par le client
 - (Depuis 2009, langage serveur avec **node.js**, à voir dans le chapitre 7)
 - Alternatives :
 - Google Dart
 - CoffeeScript
 - Microsoft TypeScript
 - Google Web Toolkit (GWT)
 - Autres



JavaScript

Pourquoi voir le **JavaScript**?

- C'est le 'code' de votre application web. Ajoute tout le dynamisme dans vos pages.
- Permet la communication client/serveur, notamment pour l'envoi/réception de données.
- Est devenu un standard incontesté dans le domaine du web. S'assurer que tout le monde est sur la même longueur d'onde...
- Applications isomorphiques en vogue, JavaScript autant client que serveur.

JavaScript

Note importante!

Les exemples suivants sont réalisés en **ES5** - bien qu'étant encore le standard universel, il est de plus en plus *deprecated* au profit de l'**ES6** et de ses successeurs.

L'important ici est de comprendre les concepts et de ne pas s'attarder à la syntaxe.

JavaScript : types

Le JavaScript est un langage très faiblement **typé**.
Toutes les variables utilisent le mot clé '**var**'.



```
var toto1 = 1;  
var toto2 = "thisIsAString";  
var toto3 = new Array();  
var toto4 = {};
```

⇒ Ne jamais oublier que le langage est **interprété**!
À vous de savoir ce que vous manipulez!

JavaScript : types

Le JavaScript est un langage très faiblement **typé**.

Comment connaître le type? Utilisez l'opérateur **typeof**.



```
var toto1 = 1;  
  
alert(typeof toto1); // Affiche "number".
```

Il existe les types ***null*** et ***undefined*** en JavaScript.

Utilisation :

```
if (typeof toto1 !== 'undefined') {  
    // Pourquoi le faire ainsi?  
}
```

JavaScript : opérateurs

Les **opérateurs** sont pratiquement les mêmes que les autres langages.

Petites particularités :

`==` VS `===`

`!=` VS `!==`

```
1 == '1' // Retourne true
```

```
1 === '1' // Retourne false
```

La triple égalité prend en compte le **type** des variables

⇒ Utilisée dans la grande majorité des cas!

JavaScript : types

Les **types de base** offrent plusieurs méthodes. Nous verrons les plus courants, à vous de lire pour découvrir les autres!

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux

1. String

```
var text = "Bonjour!";  
  
text.charAt(1); // Renvoie 'o'.  
  
text.toLowerCase(); // Renvoie la chaîne "bonjour!".  
  
text.substring(1,4); // Renvoie la chaîne "onj".  
  
text.trim(); // Enlèverait les espaces au début/à la fin.
```

etc... voir

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/String

JavaScript : types

2. Date

```
var date = new Date(); // Équivalent à Date.now().  
  
var anniversary = new Date(1995, 11, 17);  
  
anniversary.setYear(1996); // Changeons l'année...  
  
anniversary.getDate(); // Renvoie le jour du mois (1-31), donc 17.  
  
anniversary.getDay(); // Renvoie le jour de la semaine (0-6), donc 0.
```

Plusieurs méthodes de disponibles, à vous de les expérimenter...

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Date

JavaScript : types

3. Array

⇒ Seule structure de données de base en JavaScript, le tableau!

```
var array = []; // Peut également s'écrire new Array();

var years = [1950, 1960, 1970];

years.pop(); // Renvoie 1970 (dernière valeur) et l'enlève du tableau.

years.push(1970); // Ajoute la valeur 1970 à la fin du tableau.

var test = years[0]; // Renvoie la valeur à l'indice 0, donc 1950.

var test2 = years.length; // Renvoie le # d'éléments, donc 3.

years.push("une string"); // Acceptable? Pourquoi?
```

JavaScript : types

3. Array : plusieurs méthodes

```
var years = [1950, 1960, 1970];

years.reverse(); // Donne [1970, 1960, 1950].

years.sort(); // Donne [1950, 1960, 1970].

years.splice(0,1); // Donne [1950].

years.forEach(function(year) {
    console.log(year);
});
```

Plusieurs méthodes de disponibles, à vous de les expérimenter...

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array

JavaScript : fonctions

Le JavaScript permet évidemment de déclarer des fonctions.



```
function toto1() {  
    return "hello";  
}  
  
var toto1 = function() {  
    ...  
}
```

Remarquez qu'il n'y pas de **type de retour**. Il est donc impossible de savoir si une fonction retourne quelque chose ou non, il faut regarder le code plus attentivement...

JavaScript : fonctions

Exemple 2 : Remarquez qu'il est possible d'obtenir une variable qui consiste en **un pointeur** sur **la fonction**.



⇒ Encore plus important de connaître ce que vous manipulez !

En effet, la seule variable pourrait être une string, un array etc. ou une fonction qui s'appelle ainsi :

```
var toto1;
```

```
toto1();
```

JavaScript : *debugging*



TRÈS TRÈS IMPORTANT!

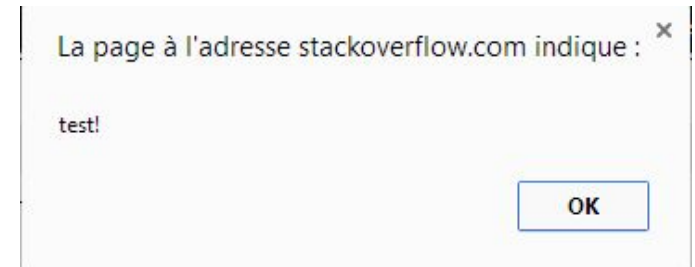
Apprenez à utiliser les fonctionnalités de ***debugging*** offertes par le JavaScript!

ENCORE PLUS IMPORTANT :



La touche **F12** ouvre les outils de développement de votre navigateur. Utilisez-les! Vous pourrez facilement expérimenter et débbugger!

```
alert("test!");
```



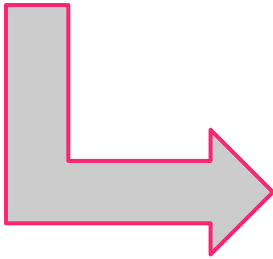
JavaScript : *debugging*

```
console.log('test!');
```



```
console.log("test!");  
test!
```

```
debugger;
```



Lorsque la console de votre navigateur est ouverte, permet de lancer un ***breakpoint***.

Autrement dit, le navigateur va pauser l'exécution du code JavaScript sur cette ligne précise, vous permettant de tester ou de voir les valeurs des variables locales par exemple.

JavaScript : opérations de base

Le JavaScript permet évidemment d'utiliser des opérations telles que les **for** et **while**.



```
for (var i = 0; i < myArray.length; i++) {  
    console.log(myArray[i]);  
}
```

Qu'en est-il du **for in**?

"Cette instruction effectue, dans un ordre arbitraire, une boucle sur les propriétés énumérables d'un objet".

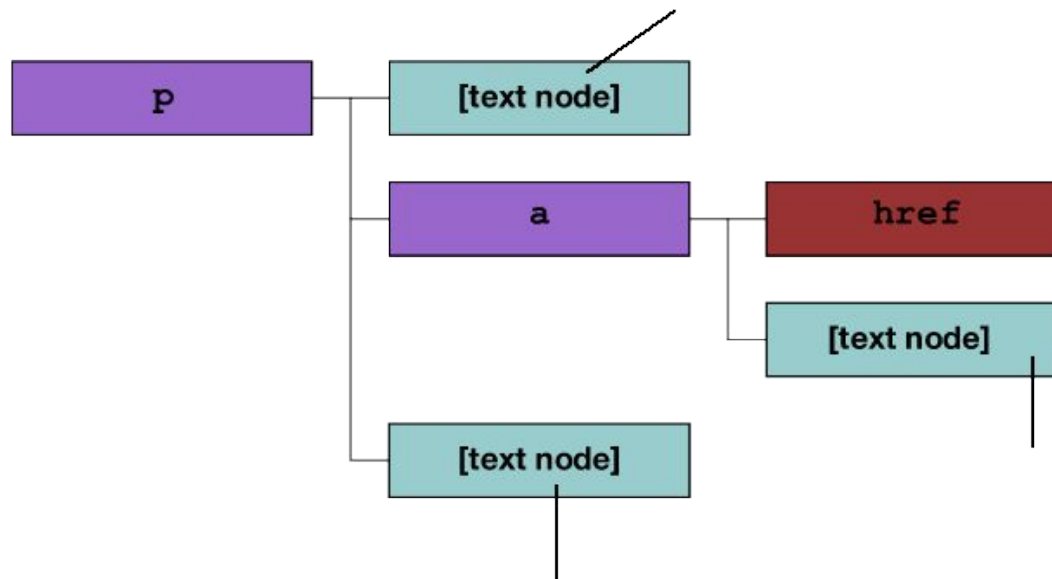
Nous verrons les objets très bientôt... et également une autre sorte de *loop* ES6 beaucoup plus pratique...

JavaScript de présentation

Utilise le DOM : Document Object Model

- Représentation objet du document.

Ex : <p>Hi! <a href="<http://example.com>">click </p>



JavaScript de présentation

Le DOM permet de :

- Accéder à n'importe quel élément dans la page et manipuler son style, son contenu et ses attributs
- Créer des nouveaux éléments, leur donner du contenu et les insérer dans la page seulement lorsqu'ils sont nécessaires.



Le JavaScript de présentation devrait être séparé du reste, il n'a pas le même but!

JavaScript de présentation

Les deux principaux éléments du DOM à savoir manipuler sont :

window : Représente la fenêtre du navigateur et tout son contenu. Tout agit sur cet élément. Par exemple, **alert()** est en réalité **window.alert()**;

document (ou window.document) : Le document représente le contenu HTML de la page courante. Il permet de le lire ainsi que de le modifier.

JavaScript de présentation



Méthodes intéressantes de **document** :

```
// Retourne l'élément html identifié par myId, ou undefined.
```

```
document.getElementById('myId');
```

```
// Retourne un array des éléments ayant l'attribut name défini
```

```
// à myName.
```

```
document.getElementsByName('myName');
```

```
// Retourne un array des éléments ayant la classe myClassName.
```

```
document.getElementsByClassName('myClassName');
```

JavaScript de présentation

Les éléments HTML sont de type **HTMLElement**.

<https://developer.mozilla.org/fr/docs/Web/API/HTMLElement>

⇒ Un élément HTML possède énormément de méthodes et de propriétés, permettant notamment de :

- Lire/Modifier ses **attributs**.
- Lire/Modifier son id/ses classes CSS.
- Lire/Modifier le texte ou son *inner* HTML.
- Lui attacher des **événements**.

JavaScript de présentation

Exemple très simple :

```
> myElement.textContent = "Accepter ce message"
< "Accepter ce message"
> myElement
< <button id="dismiss-shortcuts-btn" type="button" tabindex="-1">Accepter ce message</button>
```

Exemple un peu plus complexe :

```
> myElement.onclick = function() {
    alert("You clicked me!");
}
< function () {
    alert("You clicked me!");
}
```

Le Javascript possède une logique **événementielle**. Ici par exemple, lorsque l'événement *onclick* est lancé par le navigateur, la fonction définie ci-haut sera alors **appelée**.

JavaScript de présentation

```
element.click();
```

L'événement est lancé, par exemple lors d'une action de l'utilisateur.

```
element.onclick = function() {  
    // Do something  
};
```

L'événement est attrapé par le *event handler*, et la fonction y correspondant est exécutée.

JavaScript : divers

Outre les opérations de base, le langage est fait pour être utilisé dans une logique majoritairement asynchrone/événementielle.

"On doit embrasser l'asynchrone, et non pas lutter contre"

- Les appels au serveur ne seront jamais bloquants...
- La majorité de votre logique réagira à des événements, le flot est très loin d'être séquentiel...

JavaScript : divers

Il est important de savoir que les requêtes Web ne répondent pas immédiatement : on doit donc prévoir un mécanisme de callbacks!

Callback : Fonction de retour pouvant être appelée lorsqu'une opération est terminée.

Change donc drastiquement le fil d'exécution du code.

Votre application ne doit jamais ***geler***.

JavaScript : divers

Exemple simple:

```
function doSomething(callback) {  
    // ...  
    // Call the callback  
    callback('hello!');  
}  
  
function foo(message) {  
    // I'm the callback  
    alert (message);  
}  
  
doSomething(foo);
```

JavaScript : divers

Portée des variables

Attention aux variables **globales** : TOUT est partagé entre les différents fichiers, seulement l'ordre importe!

Toute variable déclarée dans une fonction, peu importe où, est accessible partout dans la fonction.

JavaScript : divers

Portée des variables : exemple

```
function test() {  
  
    var a;  
  
    for (var i = 0; i < 10; i++) {  
        var b = i * 10;  
        c = 12;  
    }  
  
    return [a, b, c];  
}
```

Est-ce que ça fonctionne?

JavaScript : divers

Portée des variables : exemple

```
function test() {  
  
    var a;  
  
    for (var i = 0; i < 10; i++) {  
        var b = i * 10;  
        c = 12;  
    }  
  
    return [a, b, c];  
}
```

Sans déclaration, on réfère au 'global object', soit (window) ou **this**.

JavaScript : divers

Closures

Fonctionnalité méconnue mais puissante : possibilité, pour une fonction, d'accéder à des variables qui ne sont plus à sa portée :

```
function adder(number) {  
    function add(value) {  
        return number + value;  
    }  
    return add;  
}  
  
var add10 = adder(10);  
add10(1); // ---> retourne quoi?
```

JavaScript : divers

Pratique

setInterval/setTimeout : Timer intégré dans JavaScript pour répéter des événements.

```
setInterval(function() { interval() }, 1000 );

function interval() {
    var date = new Date();
    var dateString = date.toLocaleTimeString();
    document.getElementById("time").value = dateString;
}
```

JSON: JavaScript Object Notation

Le JSON est une façon simple de représenter des objets en JavaScript. Les objets JSON sont simplement des paires de clés/valeurs, utilisées autant du côté client que sur le fil.

Un objet JSON peut se déclarer ainsi :

```
var myObject = {}; // Objet vide. Correspond à new Object();

// Déclare un objet ayant une clé name dont la valeur est John.
// Remarquez les doubles guillemets.
var myObject2 = { "name" : "John" };

// Pas exactement la même syntaxe qu'un objet JavaScript pur!
var myJavascriptObject = { name : 'John' }
```

JSON: JavaScript Object Notation

JSON un peu plus complexe :

```
var contact = {  
  "firstName": "John",  
  "lastName": "Smith",  
  "address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
    "state": "NY",  
    "postalCode": 10021  
  },  
  "phoneNumbers": [  
    "212 555-1234",  
    "646 555-4567"  
  ]  
}
```

JSON: JavaScript Object Notation

```
myObject2.name; // Que renvoie ce code?
```

```
myObject2["name"]; // Et celui-ci?
```

L'accès aux propriétés des éléments JSON peut se faire directement, soit en utilisant leur représentation en chaîne de caractères.

Si la propriété n'existe pas, elle est automatiquement créée.

```
var myObject = {  
    name : "John"  
}  
myObject.lastName; // Retourne undefined  
myObject.lastName = "Smith" // Ajoute la propriété lastName
```

JSON: JavaScript Object Notation

JSON : Méthodes utiles

JSON.parse()

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/JSON/parse

⇒ Transforme une **string** JSON en un objet. Très utile lors du **retour** d'une **requête** vers le serveur.

JSON.stringify()

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/JSON/stringify

⇒ Transforme un objet JSON en sa représentation **string**. Très utile lors de l'**envoi** d'une **requête** vers le serveur.

Chapitre 3

JavaScript moderne

Plan

- JavaScript orienté-objet
- EcmaScript 6
- Processus de développement automatisé
 - NPM Scripts

JavaScript orienté-objet

Le JavaScript ne devrait pas être qu'un langage de **script...**

⇒ Demande une architecture rigoureuse!

Par exemple, vous créez un composant de UI quelconque...

⇒ Comment le rendre réutilisable?

⇒ Pensez que le JavaScript est très prôné à l'**open-source...**

⇒ Est-ce qu'une tierce personne pourrait facilement **réutiliser** votre code?

JavaScript orienté-objet

Objet constructeur

```
// vehicle.js  
function Vehicle(make) {  
    this.make = make;  
    this.getMake = function() {  
        alert('Make: ' + this.make);  
    };  
};
```



La ***function*** agit comme définition du **constructeur** de l'objet.

Il est ensuite possible de définir les **méthodes** de l'objet.

```
// main.js  
var myCar= new Vehicle('Mercedes');  
myCar.getMake();  
  
// Alert "Make: Mercedes"
```



Remarquez la division en **fichiers JavaScript** différents.

JavaScript orienté-objet

Objet « literal »

```
// main.js
var myCar = {
  make : 'Mercedes',
  getMake : function() {
    alert('Make: ' + this.make);
  };
};

myCar.getMake();

// Alert "Make: Mercedes"
```



Définition *on-the-fly* d'un objet. Plus difficile à réutiliser.

Il est ensuite possible de définir les **méthodes** de l'objet.



Remarquez ici qu'on ne découpe pas en plusieurs fichiers, puisqu'il s'agit d'un objet unique.

JavaScript orienté-objet

Notation *prototype*

Comme nous avons vu, il est possible de **redéfinir** un objet en ajoutant/modifiant ses propriétés/méthodes.

⇒ Notation souvent utilisée : prototype.

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Object/prototype

```
var Dog = function () {}; // Déclare un objet sans aucun comportement.
```

```
var teckel = new Dog();
```

```
teckel.sleep(); // Que se passe-t-il? Est-ce valide?
```

JavaScript orienté-objet

Notation *prototype*

```
Dog.prototype.sleep = function() {  
  console.log("Zzz zzz");  
}
```

```
teckel.sleep(); // Et maintenant, que se passe-t-il?
```

```
var pitbull = new Dog();
```

```
pitbull.sleep(); // Est-ce valide?
```

JavaScript orienté-objet

Comment savoir quel comportement sera exécuté?

1. Est-ce que le comportement est directement défini sur l'instance?

```
teckel.sleep = function() {  
  console.log("zzz");  
}
```

2. Sinon, est-ce que le comportement a été défini par un prototype sur toutes les instances?

```
Dog.prototype.sleep = function() {  
  console.log('zzz');  
}
```

3. Sinon, est-ce que le comportement a été défini dans la déclaration?

```
var Dog = function() {  
  this.sleep = function() {  
    console.log('zzz');  
  }  
}
```


JavaScript orienté-objet

Est-ce possible de faire de l'héritage en JavaScript?

⇒ La réponse est **oui**! Il existe plusieurs façons de faire...

Note : Il ne s'agit pas d'héritage à proprement dit tel qu'en C# ou Java, mais bien une façon de le simuler.

Soit la déclaration suivante :

```
function Vehicle (hasEngine, hasWheels) {  
    this.hasEngine = hasEngine || false;  
    this.hasWheels = hasWheels || false;  
}
```

JavaScript orienté-objet

Soit cette autre déclaration :

```
function Car (make, model, hp) {  
  this.hp = hp;  
  this.make = make;  
  this.model = model;  
}
```

Comment faire pour que **Car** dérive de **Vehicle**?

Grâce à l'héritage dynamique!

```
Car.prototype = new Vehicle(true, true); // Indique à l'interpréteur d'appeler le constructeur de  
Vehicle lorsque le constructeur de Car est appelé.
```

JavaScript orienté-objet

Il est ensuite possible de travailler sur **Car** seulement :

```
Car.prototype.displaySpecs = function() {  
    console.log(this.make + ", " + this.model + ", " + this.hp + ", " +  
    this.hasEngine + ", " + this.hasWheels);  
}
```

```
var myCar = new Car("Toyota", "Corolla", 170)
```

```
// Car {hp: 170, make: "Toyota", model: "Corolla", hasEngine: true, hasWheels: true,  
// __proto__: Vehicle}  
// ⇒ remarquez bien le __proto__, qui expose la valeur de l'objet prototype interne (qui  
// correspond à Vehicle dans ce cas).
```

```
Vehicle.prototype.hasTrunk = true;
```

```
myCar.hasTrunk; // Est-ce valide?
```

JavaScript orienté-objet

Autre façon de faire, un peu plus simple :

```
function Car(make, model, hp, hasEngine, hasWheels) {  
  this.base = Vehicle;  
  this.base(hasEngine, hasWheels);  
  this.hp = hp;  
  this.make = make;  
  this.model = model;  
}
```

```
// Ne pas oublier d'activer l'héritage dynamique  
Car.prototype = new Vehicle;
```



L'héritage est intéressant, mais il ne faut pas en **abuser**... il amène une complexité supérieure qui n'est peut-être pas nécessaire...

JavaScript orienté-objet

Est-ce possible de faire de l'encapsulation en JavaScript?
Autrement dit, existe-il une notion de **visibilité** ?

⇒ La réponse est **oui**! Mais encore une fois, il s'agit de pseudo-privé/public... (grâce aux Closures!)

Reprenons notre déclaration précédente...

```
function Car (make, model, hp) {  
  this.hp = hp;  
  this.make = make;  
  this.model = model;  
}
```

JavaScript orienté-objet

Et bonifions notre classe Car quelque peu...

```
function Car(make, model, hp) {  
  var insurancePrice = 100;  
  this.hp = hp;  
  this.make = make;  
  this.model = model;  
  this.insurance = calculateInsurance(this.hp);  
  
  function calculateInsurance(hp) { // Remarquez l'absence de this. au début  
    return hp * insurancePrice;  
  }  
  
  this.getCarDescription = function () {  
    return make + ' ' + model;  
  }  
}
```

JavaScript orienté-objet

Et maintenant, utilisons notre nouvelle classe!

```
var myCar = new Car('Tesla', 'Model S', 288); // +1 pour Tesla.  
myCar.model; // Renvoie la chaîne Model S.  
myCar.calculateInsurance();
```

`TypeError: Object #<Car> has no method 'calculateInsurancePrice'`

⇒ Évidemment! `calculateInsurance` est une méthode **privée** de notre classe, elle n'est donc pas accessible à l'extérieur.

⇒ La même chose peut s'appliquer aux **propriétés** d'une classe!

Réflexion ...

Est-ce que le JavaScript **pur** tel que nous venons d'approfondir est encore d'actualité?

⇒ **Évidemment!** voir <http://vanilla-js.com/> (parodie)

À lire: <https://snipcart.com/blog/learn-vanilla-javascript-before-using-js-frameworks>

Le JavaScript **pur** est toujours beaucoup plus performant que n'importe quel framework l'utilisant.

Retrieve DOM element by ID

	Code	ops / sec
<i>Vanilla JS</i>	<code>document.getElementById('test-table');</code>	12,137,211
Dojo	<code>dojo.byId('test-table');</code>	5,443,343
Prototype JS	<code>\$('test-table')</code>	2,940,734
Ext JS	<code>delete Ext.elCache['test-table']; Ext.get('test-table');</code>	997,562
jQuery	<code>\$jq('#test-table');</code>	350,557
YUI	<code>YAHOO.util.Dom.get('test-table');</code>	326,534
MooTools	<code>document.id('test-table');</code>	78,802

Réflexion ...

Tous les navigateurs n'ont pas le même **interpréteur** JavaScript... Ce ne sont donc pas toutes les fonctions qui marchent universellement.

(Valide aussi pour le CSS)

```
myString = myString.trim(); // Ne fonctionne pas sous IE8/IE9, \@?@#%&$%#*(%#
```

- Utilisation de **polyfills**

<https://stackoverflow.com/questions/7087331/what-is-the-meaning-of-polyfills-in-html5>

| jQuery?

jQuery est un framework qui fut extrêmement populaire au début des années 201X. Nous ne le verrons pas.

- De moins en moins utilisé, souvent trop lourd
- Anciens navigateurs de moins en moins supportés
- Fonctionnalités ES6

| EcmaScript 6

EcmaScript 6

EcmaScript 6 est l'itération courante du standard JavaScript. Il regroupe plusieurs nouvelles fonctionnalités et une syntaxe allégée pour:

- Variables
- Classes, Héritage
- Template strings
- Destructuring
- Arrow functions
- Modules

ES6 vs ES5

Exemples

Variables

ES6 introduit de nouveaux mots clés pour identifier des variables.

const

Permet d'assigner une constante qui ne peut pas être écrasée par la suite

```
const foo = 'bar';
```

```
foo = 'baz';
```

Variables

let

Permet d'assigner une variable qui respecte la portée (enfin!)

```
for (let i = 0; i < a.length; i++) {  
    let x = a[i]  
    ...  
}  
for (let i = 0; i < b.length; i++) {  
    let y = b[i]  
    ...  
}
```

`i`, `x`, `y` ne sont pas disponibles à l'extérieur de leur boucles respectives.

Priorisez **toujours** `let` au lieu de `var`.

Il sera très rare qu'on utilisera `var` à partir de maintenant.

<https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Statements/let>

<https://stackoverflow.com/questions/762011/whats-the-difference-between-using-let-and-var-to-declare-a-variable>

Let

Voyons ce que le **transpileur** fait pour convertir notre code ES6 en ES5

ES6

```
for (let i = 0; i < a.length; i++) {  
  let x = a[i]  
}  
console.log(x);
```

ES5

```
'use strict';  
for (var i = 0; i < a.length; i++) {  
  var _x = a[i];  
}  
console.log(x);
```

Tester dans le babel repl :

babeljs.io/repl/

Destructuring (objets)

Voyons ce que le **transpileur** fait pour convertir notre code ES6 en ES5

ES6

```
const obj = {  
  a: 'a',  
  b: 'b',  
  c: 'c'  
}
```

```
const {a, b, c} = obj;  
console.log(a, b, c);
```

Tester dans le babel repl :

babeljs.io/repl/

ES5

```
'use strict';
```

```
var obj = {  
  a: 'a',  
  b: 'b',  
  c: 'c'  
};
```

```
var a = obj.a,  
    b = obj.b,  
    c = obj.c;
```

```
console.log(a, b, c);
```

Destructuring (arrays)

Voyons ce que le **transpileur** fait pour convertir notre code ES6 en ES5

ES6

```
const array = ['one', 'two', 'three'];
```

```
const [one, two] = array;
```

```
console.log(one, two);
```

ES5

```
'use strict';
```

```
var array = ['one', 'two', 'three'];
```

```
var one = array[0],  
    two = array[1];
```

```
console.log(one, two);
```

Tester dans le babel repl :

babeljs.io/repl/

Classes

```
class Rectangle {  
  constructor(height, width) {  
    this.height = height;  
    this.width = width;  
  }  
  
  get area() {  
    return this.calcArea();  
  }  
  
  calcArea() {  
    return this.height * this.width;  
  }  
}
```

```
const rectangle = new Rectangle(2, 4);  
rectangle.area; // 8  
rectangle.calcArea(); // 8
```

```
class Square extends Rectangle {  
  constructor(size) {  
    super(size, size);  
  }  
}
```

```
const square = new Square(2);  
square.area; // 4  
square.calcArea(); // 4
```

Tester dans le [babel repl](#)

Tester dans le [typescript playground](#)

Arrow functions

=>

Le JavaScript ES5 a un problème fondamental où le `this` ne représente pas toujours le contexte de la classe. Prenons l'exemple suivant:

```
function Prefixer(prefix) {  
  this.prefix = prefix;  
}  
Prefixer.prototype.prefixArray = function (arr) {  
  return arr.map(function (x) {  
    return this.prefix + x;  
  });  
};
```

<https://jsbin.com/modohadita/edit?js,console>

Arrow functions

Solutions avant ES6:

1) that = this

Le JavaScript ES5 a un problème fondamental où le `this` ne représente pas toujours le contexte de la classe. Prenons l'exemple suivant:

```
function Prefixer(prefix) {  
    this.prefix = prefix;  
}  
Prefixer.prototype.prefixArray = function (arr) {  
    var that = this;  
    return arr.map(function (x) {  
        return that.prefix + x;  
    });  
};
```

<https://jsbin.com/zopudehojo/1/edit?js,console>


Arrow functions

Solutions avant ES6:

2) Spécifier le contexte

Certaines méthodes permettent de passer le contexte en paramètre pour palier à ce problème.

```
function Prefixer(prefix) {  
  this.prefix = prefix;  
}  
Prefixer.prototype.prefixArray = function (arr) {  
  return arr.map(function (x) {  
    return this.prefix + x;  
  }, this);  
};
```



<https://jsbin.com/xeritijacu/1/edit?js,console>


Arrow functions

Solutions avant ES6:

3) bind

Il est possible d'utiliser la méthode `bind()` pour passer le contexte.

```
function Prefixer(prefix) {  
  this.prefix = prefix;  
}  
Prefixer.prototype.prefixArray = function (arr) {  
  return arr.map(function (x) {  
    return this.prefix + x;  
  }).bind(this);  
};
```



<https://jsbin.com/jidosezuna/edit?js,console>

Arrow functions

Arrow function à la rescousse

Les “arrow functions” conservent le contexte du parent.

```
class Prefixer {  
  constructor(prefix) {  
    this.prefix = prefix;  
  }  
  
  prefixArray (arr) {  
    return arr.map((x) => {  
      return this.prefix + x; // `this` représente l'instance de `Prefixer`  
    });  
  }  
}
```

<https://jsbin.com/teqifuyuqa/1/edit?js,console>

L'arrow function a aussi l'avantage d'être plus **compact** et plus lisible que le mot clé **function**.

| Modules

Modules

ES6 permet d'importer des ressources **spécifiques** de d'autres fichiers (variables, fonctions, classes, etc) contrairement à l'ES5 ou tout est accessible partout (ce qui menait à plusieurs conflits/bugs).

Afin de supporter les modules, on doit spécifier quels modules seront disponibles à l'extérieur du fichier via la primitive **export**.

Modules

```
// moduleA.js
```

```
export function foo() {  
}
```

foo est exporté, donc
disponible à l'extérieur de
moduleA.js

```
function bar() {  
}
```

bar est seulement
disponible à l'intérieur du
fichier moduleA.js

```
// moduleB.js
```

```
import {foo} from './moduleA.js';  
  
foo();
```

on importe la fonction foo
en spécifiant le "path" vers
le fichier

<https://www.webpackbin.com/bins/-Kr8YvxmfTGkUS8AsRHq>

Modules

default export, vs named export

```
// moduleA.js
export function foo() {

}
```

```
export default foo;
```

```
export function bar() {

}
```

```
// moduleB.js
import foo, {bar} from './moduleA';
import renamedFoo, {bar} from './moduleA';
import foo, {bar as renamedBar} from './moduleA';
```

```
// moduleC.js
import $ from 'jquery';
import _ from 'underscore';
```

`foo` est exporté par défaut. On recommande d'utiliser ceci pour les fichiers qui ont une seule classe/fonction. Il ne peut y avoir qu'un seul "default export" par fichier.

`bar` est un "named export".

les "default exports" peuvent être nommé à la valeur de notre choix au moment de l'importation

les "named exports" peuvent être renommé via la primitive "`as`"

les bibliothèques utilisent souvent les "default exports" comme JQuery, Underscore, React, etc.

Processus de développement automatisé

Gestion des dépendances

npm

Votre projet fera appel souvent à des librairies ou des outils externes (jQuery, babel, webpack, vue, angular, react, etc)

La méthode optimale pour télécharger ces modules est de faire appel à npm, (node package manager).

npm est équivalent à maven pour le Java.



Gestion des dépendances

npm

npm permet de recueillir toutes les dépendances de votre projet dans un fichier nommé `package.json`

- On génère ce fichier à l'aide de la commande `npm init`
- Une fois le `package.json` généré, on peut ajouter des dépendances à l'aide de la commande `npm install myPackage`

docs.npmjs.com





package.json

Exemple

<https://github.com/GL03102/UFood/blob/master/package.json>



Build process

Pourquoi avoir un build process en front-end?

Au fil du temps vous aurez besoin d'utiliser plusieurs commandes pour:

- rouler la suite de tests
- compiler le projet
- lancer un serveur
- déployer
- formater le code
- etc.

“

Difficile à retenir toutes ces commandes !



npm scripts

npm à la rescousse

Le **package.json** permet d'ajouter des alias vers des commandes plus complexes dans la section **scripts**.

```
{  
  "name": "MyPackage",  
  "version": "2.3.7",  
  "scripts": {  
    "build": "babel src --out-dir build",  
    "build:watch": "babel src --out-dir build --watch"  
  }  
}
```

Beaucoup plus facile à retenir, et documente les commandes disponibles pour votre projet.



Notez bien:

`npm`, `babel` et `webpack` sont des outils extrêmement personnalisables qui offrent une configuration plutôt complexe. Ils seront **préconfigurés** pour vos projets de session afin de vous donner une longueur d'avance.

À vous de lire leur documentations respectives pour approfondir le sujet.

Un usage plus avancé est vu dans la suite de ce cours.

webpack.github.io/docs
babeljs.io/docs/setup
docs.npmjs.com



Structure de fichiers

Une application Web comporte généralement un dossier '**resources**', dans lequel on retrouve les fichiers déjà mentionnés :

- **css** : stylesheets
- **img / assets** : images
- **node_modules** librairies installés via npm
- **src** : tous les fichiers JS en ES6
- **build / dist** fichiers prêt pour le navigateur (transpilés, concaténés, minifiés)

Presque toutes vos librairies seront installées grâce à npm, même le css.

Chapitre 4

Interactions serveur

Plan

- Protocole HTTP
- REST
- Restful API
- AJAX

Protocole HTTP

- Hyper Text Transfer Protocol : protocole de la couche Application , créé en 1990
- Standard(s) actuel(s) : **HTTP 1.1, 2.0, 3.0**
- Variante : HTTPS : Utilise le SSL ou TLS pour la communication sécurisée (nous en reparlerons après la relâche)
- Par défaut, utilise le port **80**, ou **443** pour HTTPS
- Les clients HTTP se connectent à des serveurs HTTP tels que Apache ou IIS.



Protocole HTTP

Pourquoi étudier le protocole **HTTP**?

⇒ Protocole **par défaut** utilisé sur les internet.

⇒ Toute communication impliquant un client/serveur passe par le HTTP.

Donc important de connaître et comprendre ce que ça fait!

HTTP **3.0** en cours de développement



Requête HTTP

Ligne de commande (Commande, URL, Version de protocole)

En-tête de requête

[Ligne vide]

Corps de requête

Exemple typique :

Adresse voulue

Protocole avec version

Primitive

GET /page.html HTTP/1.0

Host: example.com

Referer: http://example.com/

User-Agent: Mozilla 4.0

Requête HTTP

Host : Site web concerné par la requête.

Referrer : Indique l'adresse du document qui a donné lien vers la ressource demandée.

User-Agent : Logiciel utilisé pour la requête. Peut être un navigateur ou un robot.

Autres entêtes: Connection, Accept, Accept-Charset, Accept-Language, Transfer-Encoding, Trailer....

Requête HTTP

Un header très important : le **Content-Type**.

Sert à spécifier le **MIME type** du contenu de votre requête.

Exemple :

Content-Type : application/json

Le client doit indiquer quel genre de contenu il envoie au serveur afin que les données se rendent sous un format acceptable.

Réponse HTTP

Ligne de statut (Version, Code-réponse, Texte-réponse)
En-tête de réponse
[Ligne vide]
Corps de réponse

Exemple typique :

```
Date: Fri, 31 Dec 1999 23:59:59 GMT
Server: Apache/0.8.4
Content-Type: text/html
Content-Length: 59
Expires: Sat, 01 Jan 2000 00:59:59 GMT
Last-modified: Fri, 09 Aug 1996 14:21:40 GMT
```

Réponse HTTP

Date : Moment auquel le message est généré.

Server : Modèle du serveur répondant à la requête.

Content-Type : Type MIME de la ressource obtenue

Ex : text/html, application/xml, application/json, etc.

Content-Length : Taille en octets de la ressource

Expires : Moment où la ressource devient obsolète : pratique pour la mémoire cache

Last-Modified : Date de dernière modification

et autres...

Primitives HTTP

- **GET** :
Méthode permettant d'**obtenir** une ressource. Requête sans effet sur la ressource : doit pouvoir être répétée, **idempotente**.
- **HEAD** :
Semblable à GET, mais demande des **informations** sur la ressource au lieu de demander la ressource elle-même.
- **POST** :
Permet typiquement de **soumettre** une nouvelle ressource. Sert également de primitive pour tout ce qui *action* sur une ressource.

Primitives HTTP

- **OPTIONS** :
Obtient les **options** de communication d'une ressource ou du serveur
- **CONNECT** :
Permet d'utiliser un **proxy** ou tunnel.
- **TRACE** :
Demande au serveur de retourner ce qu'il a reçu, à des fins de **débogage**.
- **PUT** :
Permet de **remplacer** ou **d'ajouter** une ressource sur le serveur.

Primitives HTTP

- **PATCH** :
Permet la **modification partielle** d'une ressource.
- **DELETE** :
Permet de **supprimer** une ressource du serveur.



Primitives HTTP

POST - PUT - PATCH

POST: Création de ressource

PUT: Modification complète d'une ressource existante ou création

PATCH: Modification partielle d'une ressource existante

Un bon API utilise les primitives HTTP de manière appropriée: GET, PUT, POST, DELETE etc.

En pratique

Onglet Network de la console Chrome

Elements Network Sources Timeline Profiles Resources Audits Console									
Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	Timeline	100 ms	150 ms
me?access_token=CAACEdEose0cBADzUUB7cLKyaI8Wy... graph.facebook.com/v1.0	GET	200 OK	application/json	qLhkcdpa6Ki.js:15 Script	667 B 79 B	188 ms 186 ms			
?id=721503217860715&ev=PixelInitialized&dl=https%... www.facebook.com/tr	GET	200 OK	image/gif	fbds.js:10 Script	1.5 KB 43 B	89 ms 88 ms			
?id=675141479195042&ev=PixelInitialized&dl=https%... www.facebook.com/tr	GET	200 OK	image/gif	fbds.js:10 Script	1.5 KB 43 B	96 ms 94 ms			
*me?access_token=CAACEdEose0cBADzUUB7cLKyaI8Wy... graph.facebook.com/v1.0/schema	GET	200 OK	application/json	6s5kuGF1xtc.js:154 Script	3.2 KB 20.0 KB	97 ms 96 ms			

En pratique

Postman:

<https://www.getpostman.com/>



Outil de référence pour développement web. Importance de tester vos appels HTTP (REST) *avant* de coder quoique ce soit...

Status HTTP

- Les **codes d'erreur** sont très pratiques pour diagnostiquer le bobo lors d'une requête qui n'aboutit pas ou qui retourne le mauvais résultat.
- Les codes d'erreur sont répartis dans des **intervalles**, qui elles sont divisées en catégories

1xx : Information



2xx : Succès



3xx : Redirection



4xx : Erreur Client



5xx : Erreur Serveur



Une bonne API doit se conformer au standard et retourner des statuts cohérents.

Status HTTP

200 OK : La requête s'est effectuée avec succès

GET : La page désirée a bien été retournée

POST : La requête contient le résultat de l'action postée, peut-être du contenu HTML, une ressource, ou rien du tout (204 dans ce cas)

302 FOUND : La ressource a été trouvée, mais après une redirection.

Dans la réponse, l'entête '**Location**' doit normalement indiquer le nouvel URL de la ressource.

400 BAD REQUEST :

Requête mal formée, rejetée par le serveur.

Status HTTP

401 : UNAUTHORIZED

L'accès à la ressource nécessite une **authentification** qui n'a pas réussi. Par exemple, l'utilisateur n'est pas valide ou a fourni un mauvais mot de passe.

403 : FORBIDDEN

Le serveur a bien reçu la requête, mais refuse de l'interpréter car l'utilisateur n'a pas les permissions nécessaires.

404 : NOT FOUND

Le serveur ne trouve rien correspondant à l'URL de la requête.

Status HTTP

500 : INTERNAL SERVER ERROR

Le serveur a fait une erreur en essayant de remplir la requête (typiquement une exception)

503 : SERVER UNAVAILABLE

Le serveur ne peut répondre à la requête, par exemple suite à un volume trop lourd ou une maintenance planifiée...

Pour les autres erreurs, il y a la documentation...

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

418 !!

REST (Representational State Transfer)

Le **REST** est devenu le style architectural de référence pour toute application web moderne.

- Ce n'est pas un **protocole**, mais bien **un style architectural**.
- Il s'agit simplement d'une façon de structurer son API en suivant un certain standard.
- Utilise toutes les notions d'HTTP vues précédemment.
- Très, très, très flexible...

REST

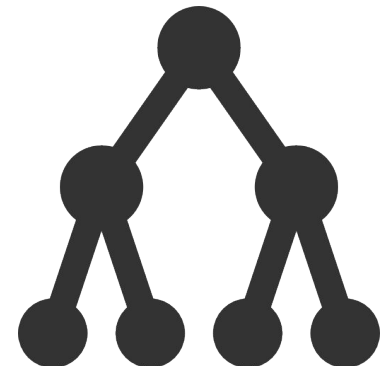
CONCEPT :

Le REST repose sur l'image qu'on se fait d'une application Web : l'utilisateur progresse à travers des **liens** qui pointent sur des **ressources**. L'accès à un lien se traduit à une **requête** HTTP:

GET, POST, DELETE, etc .

→ Le REST est donc plutôt **invisible**!

→ Le REST est **hiérarchique**!



REST

CONCEPT :

REST repose sur l'existence de ***ressources***.

→ **Chaque ressource est référencée par un identifiant unique, soit son URI.**

Une ressource est souvent un simple bout de code s'exécutant lors de l'appel de l'URI en question.

Les données échangées peuvent prendre plusieurs formes: JSON, HTML, images etc.

REST

Exemple :

L'accès à <http://ufood.herokuapp.com/unsecure/users>

Exécute du **code** et retourne une **ressource** reliée à cet URL.

Il est important de comprendre que le code peut être littéralement n'importe quoi, et retourner un format complètement arbitraire.

Ceci dit, UFood est un **JSON** REST API - donc retourne strictement ce format (plus certains fichiers).

RESTful API

Lorsqu'on parle de RESTful API, on parle donc du style architectural REST, appliqué à une API.

- Respecte les primitives HTTP vues.
- Les données échangées seront traditionnellement sous la forme de JSON. Puisqu'il s'agit d'une API, pas de HTML échangé...
- Destinée à être consommée par d'autres applications ou par un navigateur.

RESTful API

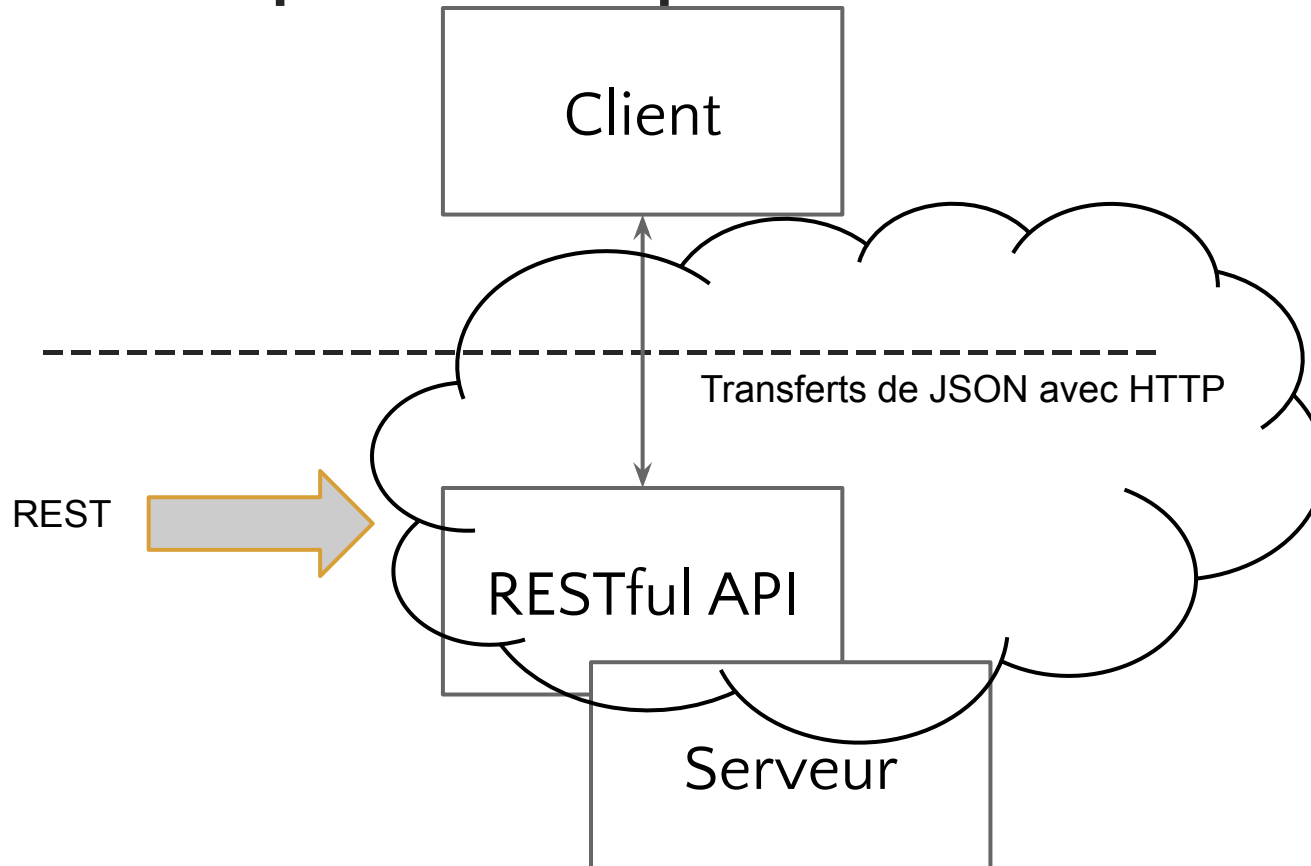
IMPORTANT

- **Destinée à être consommée par d'autres applications ou par un navigateur.**

Le design d'un API RESTful est aussi important que le code. Les méthodes doivent être claires, respecter les standards, exposer des statuts HTTP cohérents etc. Puisqu'il ne s'agit que de JSON, vous ne pouvez prévoir qui sera votre utilisateur!

RESTful API

! Très important à comprendre !



RESTful API



URL	GET	POST
Collection : http://www.google.com/resources/	Liste les id des différents éléments dans la collection.	Crée une nouvelle entrée dans la collection. Le nouvel ID est automatiquement assigné.
Élément: http://www.google.com/resources/item1	Retourne une certaine représentation de l'élément.	Généralement non utilisé .

RESTful API



URL	PUT	DELETE
Collection : http://www.google.com/resources/	Remplace la collection par une nouvelle collection.	Supprime la collection.
Élément: http://www.google.com/resources/item1	Remplace un élément de la collection ou le crée s'il n'existe pas.	Supprime l'élément de la collection.

RESTful API

Exemple d'une **bonne** ressource:

<http://ubeat.herokuapp.com/unsecure/users/userId>

Exemple d'une **mauvaise** ressource:

<http://ubeat.herokuapp.com/unsecure/users?id=userId>

Les identifiants de ressource doivent former un ensemble **hiérarchique cohérent**. Les **query params** ne devraient servir qu'à filtrer, et non pas impacter quel appel est fait sur le serveur.

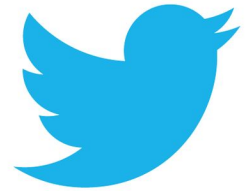
Exemple typique:

http://ubeat.herokuapp.com/unsecure/users?name=Vincent

Examples

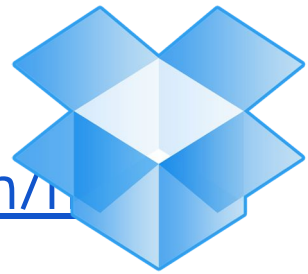
Twitter REST API:

<https://developer.twitter.com/en/docs/api-reference->



Dropbox Core API:

<https://www.dropbox.com/developers/documentation/html/documentation>



Slack API:

<https://api.slack.com/web>



Exemples

Any-API

<https://any-api.com/>

Public API

<https://github.com/public-apis/public-apis>

Pourrait être pratique pour votre projet de session...

CORS (Cross-Origin Resource Sharing)

Le **CORS** est un mécanisme permettant d'obtenir/envoyer des ressources entre des **domaines différents**.

Pourquoi s'en soucier?

⇒ Par défaut, **vous ne pouvez pas envoyer de requêtes à partir de votre navigateur sur un serveur sur un différent port et/ou url.**

Votre serveur, qui roule par exemple sur **localhost:8080/**, n'a pas la même origine que votre client!

Ce mécanisme est imposé par votre **navigateur afin** d'éviter des **failles de sécurité**.

CORS (Cross-Origin Resource Sharing)

Comment s'en sortir?

⇒ Ajouter **un header** sur les réponses HTTP afin de modifier les restrictions d'origines!

```
Access-Control-Allow-Origin = "*"
```

Peut-être que mettre * n'est pas la technique la plus **sécuritaire**...

Autres headers pertinents :

```
Access-Control-Allow-Methods = GET, POST
```

```
Access-Control-Allow-Credentials = true
```

Voir : https://developer.mozilla.org/fr/docs/HTTP/Access_control_CORS

AJAX



Qu'est-ce que l'**AJAX**?

- Simplement, il s'agit du nom donné à la façon d'interagir avec le serveur en JavaScript.
- X stands for XML, ce qui n'est évidemment plus d'actualité... mais le nom demeure.

AJAX



L'**AJAX** représente l'intégration pratique des différents concepts vus jusqu'à présent.

- Une façon simple d'interagir avec un API en JavaScript de manière asynchrone.
- Toute application web moderne est basée sur l'AJAX. C'est ce qui permet de rafraîchir différentes parties d'une page de manière indépendante...

AJAX



Traditionnellement, l'**AJAX** était basé sur l'objet **XMLHttpRequest** en JavaScript. Malgré son nom, il s'agit encore de la référence utilisée en ES5.

C'est ce qui permet de faire des requêtes au serveur et d'handler la réponse de manière asynchrone.

<https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest>

Promise

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Promise

- Un objet dont la valeur n'est pas connue `_encore_` et sera déterminée de manière asynchrone.
- On dit une Promise comme **resolved** lorsque l'appel serveur est exécuté avec succès, ou **rejected** si l'appel échoue.
- Peu importe la syntaxe, il est toujours important de traiter les 2 cas possibles
 - Toujours assumer que vos requêtes serveur peuvent **échouer!**

Promise

```
const promise = new Promise((resolve, reject) => {
  const request = new XMLHttpRequest();

  request.open('GET', 'https://api.icndb.com/jokes/random');
  request.onload = () => {
    if (request.status === 200) {
      resolve(request.response);
    } else {
      reject(Error(request.statusText));
    }
  };

  request.onerror = () => {
    reject(Error('Error fetching data.'));
  };

  request.send();
});

promise.then((data) => {
  document.body.textContent = JSON.parse(data).value.joke;
}, (error) => {
  console.log(error.message);
});
```

Async/Await

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/async_function

- La syntaxe **async/await** permet de faire des promesses de manière plus élégante.
- Introduite dans la même veine que les changements ES6.
- Permet de ne pas tomber dans ce qui est souvent référencé comme le *callback hell*.

Async/Await

```
async function doRequest(options) {  
  return new Promise ((resolve, reject) => {  
    const req = http.request(options);  
    req.on('response', res => { resolve(res); });  
    req.on('error', err => { reject(err); });  
  });  
}  
  
try {  
  const res = await doRequest(options);  
} catch (err) {  
  console.log('some error occurred...');  
}
```

AJAX

Fetch API

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Enfin, XMLHttpRequest remplacé par **Request** et **Response**, en plus d'ajouter quelques nouvelles fonctionnalités.

N'est évidemment compatible qu'avec les plus récents navigateurs.

AJAX

Fetch API

```
const myHeaders = new Headers();

const myInit = { method: 'GET',
                  headers: myHeaders,
                  mode: 'cors',
                  cache: 'default' };

fetch('flowers.jpg', myInit).then(function(response) {
  return response.blob();
}).then(function(myBlob) {
  const objectURL = URL.createObjectURL(myBlob);
  myImage.src = objectURL;
});
```

Exemples complets:

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

<https://scotch.io/tutorials/how-to-use-the-javascript-fetch-api-to-get-data>

AJAX

Axios

<https://github.com/axios/axios>

Librairie largement utilisée pour toute ce qui est requête HTTP - souvent reconnue comme plus simple et avec une interface plus pratique que fetch.

Fonctionne aussi bien en Promise qu'avec async await.

Chapitre 5

Architecture web

Plan

- Architecture de haut niveau
- Patrons de conception d'interface
 - MVC
 - MVP
 - MVVM
- *Templating*
- Application: VueJS

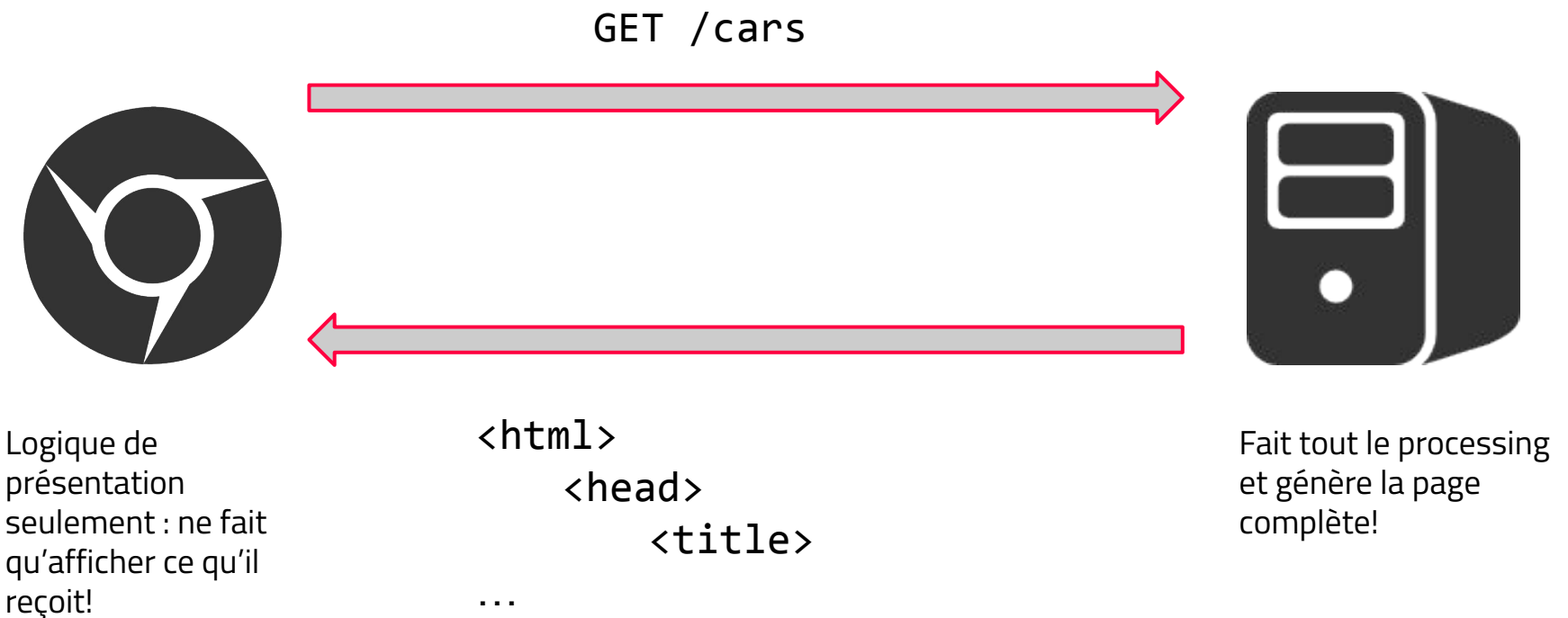
2 écoles de pensée...

Nous commençons à discuter d'**application** web...
et de haut niveau, il existe typiquement 2 façons de voir les choses.

- La manière 'classique' : server-side
- La manière 'moderne' : client-side

Server-side

Le client fait des requêtes REST vers le serveur, qui renvoie les pages HTML complètes!



Server-side

Existe encore beaucoup aujourd'hui

- ASP.NET
- Spring MVC
- PHP
- Ruby on Rails
- Django

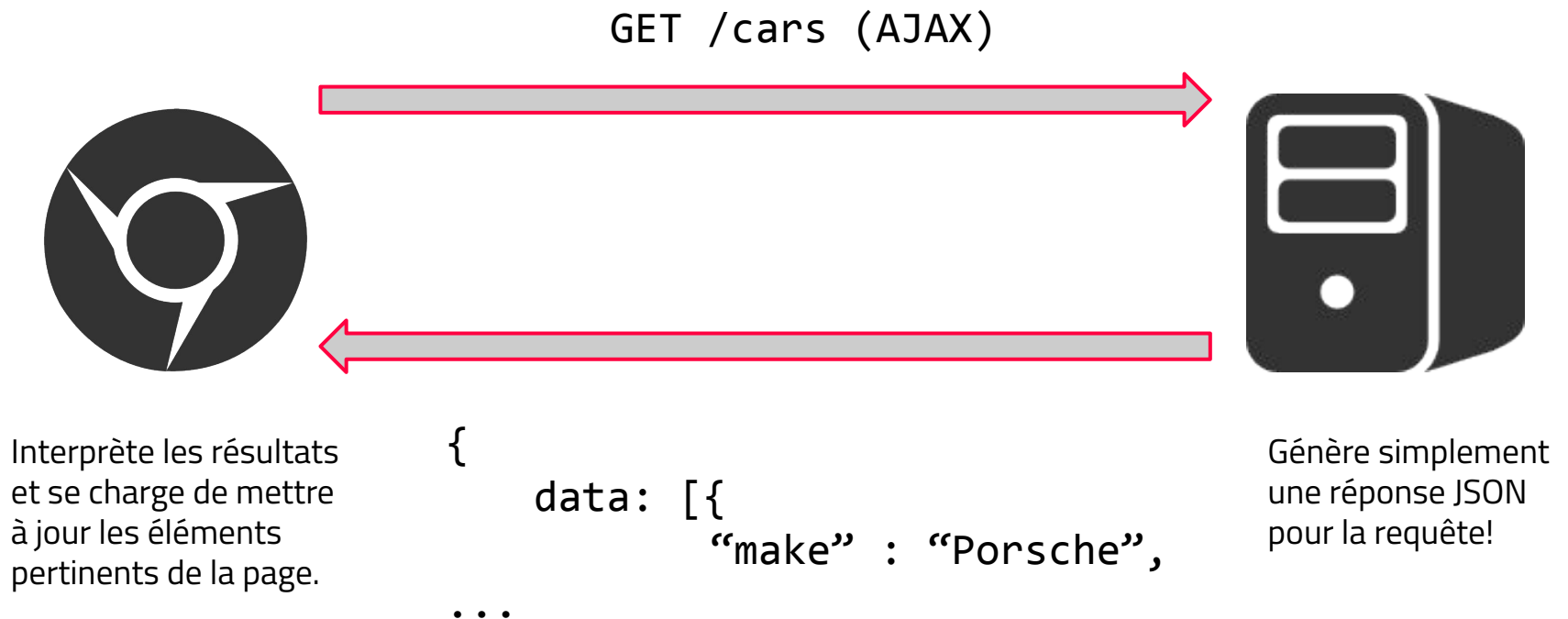
⇒ Très **performant** (évidemment, le navigateur n'a pas grand chose à faire...

⇒ Couple le **déploiement** du serveur et du client.

⇒ Demande un redéploiement complet pour une modification de style...

Client-side

Le client fait des requêtes AJAX vers le serveur qui n'expose qu'une API et renvoie du JSON.



Client-side

La manière *server-side* est toujours d'actualité... dans certains cas.

La majorité des applications sont développées comme client-side... avec une twist hybride pour éviter les problèmes de CORS.

- Déploiements indépendants, équipes indépendantes!
- Interchangeabilité des deux parties

Client-side

Côté client, le but est d'arriver à une **single page application**.

- Pourquoi recharger les header/footer de ce monde alors qu'ils ne changent pas?
- Chaque particule de l'application peut utiliser l'API... ou non.
- Optimiser la performance au maximum, les changements au DOM sont extrêmement coûteux.

| Patrons de conception

M V C

M V P

MVVM

Model View Controller

Model View Presenter

Model View ViewModel

MVC, MVP, MVVM

- Les 3 patrons d'apporter des solutions pour :
 - Découpler la logique des données et de l'interface
 - Séparation des responsabilités
- **Vue** (Interface Utilisateur)
- **Modèle** (Données affichées dans le UI)
- « **Colle** » (Gestion d'évènements, logique, etc)
- Vue et Modèle ont des définitions semblables dans les 3 patrons
- La « Colle » est la partie qui diffère.
 - C : Controller
 - P : Presenter
 - VM : ViewModel

Il s'agit de **design patterns d'interface!**

MVC, MVP, MVVM

Vue

```
<html>
  <head>
    <title>
      MyPage
    </title>
  </head>
  ...

```

Modèle

```
{
  data: [{
    "make" : "Porsche",
    ...
  ]
}
```

Contrôleur

Fait l'association entre les **vues** et les **modèles**... est donc responsable de la logique!

MVC, MVP, MVVM

- Modèle
 - Représentation des données/domaine
 - Ne connaît **PAS** les notions de vue, contrôleur, etc.
- Vue
 - Représentation (visuelle) des données
 - *Templates*
 - Ne connaît **PAS** les notions de modèle, contrôleur, etc.
- Controller, Presenter, ViewModel
 - Communique avec la vue et le modèle
 - Logique d'affaire

MVC

Model View Controller
(pas Model View Colle...)

MVC

- Inventé par des développeurs *Smalltalk* (1979)
- Une grande majorité de frameworks reposent sur ce patron
 - Django, Ruby on Rails, CakePHP, SpringMVC, ASP.NET MVC, Play!, etc.

* Dans les débuts de AngularJS (maintenant plus près de MVVM)

MVC

- **Contrôleur**

- Lien entre l'utilisateur et l'application
- Les événements dans la vue lancent des actions que le contrôleur utilise pour **modifier le modèle** et déterminer **quelle vue doit-être affichée** ou mise-à-jour.
- Il peut y avoir plusieurs vues pour un contrôleur et vice-versa.

- En server-side

- Le contrôleur détermine quelle vue doit être affichée
- La vue envoie des événements au contrôleur via une requête HTTP (URL) qui est acheminée (« routed ») au contrôleur approprié et à la méthode adéquate.

- En client-side

- Les événements sont des événements JavaScript lancés par les interactions de l'utilisateur.

MVC

MVC utilise activement les concepts déjà présentés auparavant (AJAX, REST etc.)

⇒ Il s'agit vraiment d'une couche architecturale **au-dessus** !

SpringMVC fonctionne selon la manière **classique**, c'est-à-dire que tout le traitement est fait server-side.

⇒ MVC peut être implémenté autant **server-side** que **client-side**!
(Nous allons faire du client-side, ça s'en vient...)

MVP

Model View Presenter

MVP

- Dérivé du MVC
- Popularisé par Microsoft **ASP.NET**
- Utilisé dans GWT, GWTP, Vaadin, Swing
- Communication avec la vue
 - Communication bidirectionnelle avec la vue
 - La vue communique avec le Presenter en appelant **directement** des fonctions du Presenter.
 - Le Presenter communique avec la vue en appelant une **interface implémentée par** la vue
 - Il y a (habituellement) **un** Presenter pour **une** vue.

M V VM

Model View ViewModel

MVVM

- Popularisé par Microsoft avec Windows Presentation Foundation (**WPF**) et **Silverlight**
- Utilisé dans KnockoutJS, KendoUI, KnockBack.JS, **VueJS**, AngularJS*, etc.
- Communication avec la vue
 - Communication bidirectionnelle avec la vue
 - Le ViewModel représente la vue (Les attributs du ViewModel correspondent plus à la vue qu'au modèle)
 - La vue est attaché (« bound ») au modèle. Dès qu'un changement au ViewModel est effectué, la vue est instantanément mise-à-jour (vue **active** au lieu de passive)
- La plupart des **frameworks JavaScript modernes** s'inspirent de ce patron (mais ne l'appliquent pas à 100%)

MVVM

Pas de contrôleur ou de présentateur?

⇒ Le ViewModel observe les changements du modèle et s'occupe de mettre à jour la vue en conséquence.

⇒ Typiquement moins de logique qu'un **contrôleur** traditionnel.

⇒ Ressemble au pattern **Observer**.

MVVM

Model

```
// Model
var data = {
    "Id": 1001,
    "SalePrice": 1649.01,
    "ListPrice": 2199.00,
    "ShortDesc": "Taylor 314CE",
    "Description": "Taylor 314-CE Left-Handed Grand Auditorium Acoustic-Electric
Guitar"
};
```

View

```
<h1 data-bind="text: shortDesc"></h1>
<div data-bind="text: description" class="descArea"></div>
<span class="label label-success" data-bind="text: formatCurrency(salePrice)"></span>
```

ViewModel

```
// ViewModel
var viewModel = {
    id: ko.observable(data.Id),
    salePrice: ko.observable(data.SalePrice),
    listPrice: ko.observable(data.ListPrice),
    shortDesc: ko.observable(data.ShortDesc),
    description: ko.observable(data.Description),
    formatCurrency: function(value) {
        return "$" + value().toFixed(2);
    }
};
// Bind the ViewModel to the View using Knockout
ko.applyBindings(viewModel);
```

MVVM

Comment est-ce que les mises à jour du modèle sont effectuées?

```
<div>
  You've clicked <span data-bind="text: numberOfClicks"></span> times
  <button data-bind="click: incrementClickCounter">Click me</button>
</div>

<script type="text/javascript">
  var viewModel = {
    numberOfClicks : ko.observable(0),
    incrementClickCounter : function() {
      var previousCount = this.numberOfClicks();
      this.numberOfClicks(previousCount + 1);
    }
  };
</script>
```

Les annotations **data-bind** permettent de définir les actions auxquelles le ViewModel réagissent! (exemple avec KnockoutJS)

MV*

- Ne pas se perdre dans les patterns
- Le but principal est la **séparation des responsabilités!**
- **Google a même déclaré AngularJS comme étant framework MVW (Model View Whatever)**
- Toujours de la magie -> Comprenez ce que vous faites!!



AngularJS

Shared publicly - Jul 19, 2012

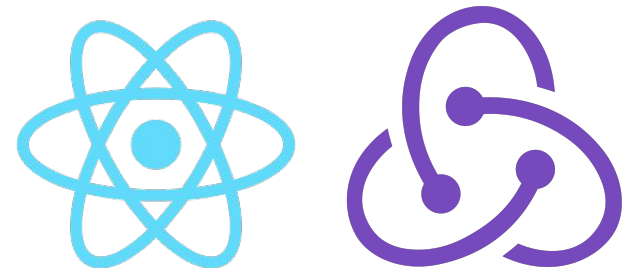
Having said, I'd rather see developers build kick-ass apps that are well-designed and follow separation of concerns, than see them waste time arguing about MV* nonsense. And for this reason, I hereby declare AngularJS to be MVW framework - Model-View-Whatever. Where Whatever stands for "whatever works for you".

Igor Minar, Employé de Google à propos d'AngularJS

React/Redux?

Certains d'entre vous ont peut-être déjà fait du **React/Redux**. Nous ne verrons pas ces *frameworks* dans le cours, mais notez bien que...

- React n'est **pas** un framework MVC.
- Redux est un framework utilisant le patron **Flux**. Il s'agit d'un paradigme un peu plus avancé que nous voyons dans la suite du cours.



| *Templating*

Templating

Le **templating** est un mécanisme permettant de mettre à jour facilement une page HTML avec de nouvelles informations.

L'engin de templating (il en existe de multiples) possède son propre interpréteur, et sert à remplacer des **annotations** spécifiques par des paramètres fournis.

⇒ Le résultat est du HTML, mais qui a été en quelque sorte "**compilé**".

Templating

En JavaScript

Framework permettant de transformer du "HTML" avec balises spécifiques en du HTML compréhensible par le navigateur...

- **Facilité de réutilisation**
- **Découpage**
- **Plus performant pour le navigateur -> ne pas recompiler de templates inutilement**

Templating

De multiples engins de templating existent :

- JSP (avec Spring)
- MVC Razor
- UnderscoreJS
- Handlebars : handlebarsjs.com
- Mustache

Templating

Exemple classique en JSP :

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ page session="false"%>
<html>
<body>
<c:import url="navbar.jsp" />
  <div class="container">
    <c:import url="header.jsp" />
    <div class="row-fluid">
      <h1 class="whitetext span6 well centered title">${title}</h1>
      <c:if test="${searchresults != null}">
        <span class="span12">
          ${searchresults} result<c:if test="${searchresults > 1}">s</c:if> found.
        </span>
      </c:if>
      <c:import url="carlisttemplate.jsp"></c:import>
    </div>
  </div>
  <c:import url="footer.jsp" />
</body>
</html>
```

⇒ Ressemble à du HTML, mais avec des balises spécifiques à l'engin courant.

Templating

Soit le cas d'utilisation suivant :

Vous avez ce HTML :

```
<div id="cars">  
  <div>Toyota Corolla</div>  
  <div>Toyota Yaris</div>  
  <div>Toyota 4Runner</div>  
</div>
```

Et vous désirez le **mettre à jour** suivant une requête au serveur... qui vous a renvoyé les informations suivantes :

["Toyota Corolla", "Toyota Yaris"]

Que feriez-vous avec vos connaissances actuelles?

Templating

Solution basée sur ce que nous avons vu :

1. Supprimer les éléments du div courant.
2. Créer des nouveaux éléments pour chaque résultat.

Pas très performant, et **beaucoup** de code pour peu de valeur.

Templating

Solution utilisant le templating (Handlebars)

1. Définition de la template

```
<script id="cars-template" type="text/x-handlebars-template">
  {{#each cars}}
    <div>{{this}}</div>
  {{/each}}
</script>
```

2. Définition de l'élément HTML

```
<div id="cars"></div>
```



Templating

Solution utilisant le templating (Handlebars)

3. Appel de la template

```
<div id="cars"></div>
<script>
  // Une seule fois au début. Compile la template.
  const source  = document.getElementById("cars-template").innerHTML;
  const template = Handlebars.compile(source);

  // Appel de la template avec le nouveau data. 2 seules lignes de code!
  const data = {cars : ["Toyota Corolla", "Toyota Yaris"]};
  document.getElementById("cars").innerHTML = template(data);
</script>
```

Templating

Solution utilisant le templating (Handlebars)

code complet :

```
<!DOCTYPE html>
<html>
<head>
  <title>Handlebars.js Demo</title>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.7.7/handlebars.min.js"></script>
  <script id="cars-template" type="text/x-handlebars-template"> <!-- Ne pas oublier le tag script -->
    {{#each cars}}
      <div>{{this}}</div>
    {{/each}}
  </script>
</head>
<body>
<div id="cars"></div> <!-- Remarquez comment le HTML devient très très simple -->
</body>
<script>
  // Une seule fois au début. Compile la template.
  const source  = document.getElementById("cars-template").innerHTML;
  const template = Handlebars.compile(source);

  // Appel de la template avec le nouveau data. 2 seules lignes de code!
  const data = {cars : ["Toyota Corolla", "Toyota Yaris"]};
  document.getElementById("cars").innerHTML = template(data);
</script>
</html>
```

| VueJS



Pourquoi Vue ?

vs Angular, React, Svelte

Vue.js s'avère un bon framework pour débiter, avec un peu moins de complexité initiale que Angular, React et cie. Il est également beaucoup utilisé dans le marché (même si moindre que React).

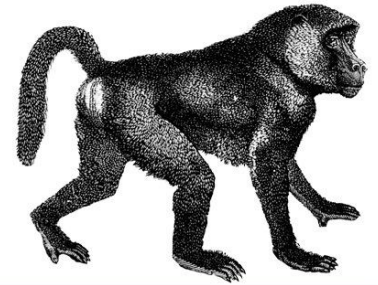
Les concepts amenés dans Vue sont **simples** et **universels**. Apprendre Vue vous permettra d'appliquer des concepts semblables dans des frameworks ou librairie plus complexes comme Angular, React ou Svelte.

“

Il est encore une fois important de noter que la compréhension du JavaScript pur est fondamentale avant l'utilisation de ce framework.

snipcart.com/blog/learn-vanilla-javascript-before-using-js-frameworks

Everything You Need To Know About How To Pretend It



Pretending To
Know JavaScript

The Definitive Guide

O RLY?

@IamDeveloper

Attention!

**Nous utiliserons Vue
version 3 dans le
cadre du cours**



Hello World

JS

```
import { createApp } from 'vue'

createApp({
  data() {
    return {
      count: 0
    }
  }
}).mount('#app')
```

HTML

```
<div id="app">
  <button @click="count++">
    Count is: {{ count }}
  </button>
</div>
```

“

Although not strictly associated with the MVVM pattern, Vue's design was partly inspired by it. As a convention, we often use the variable `vm` (short for `ViewModel`) to refer to our Vue instance.

Vue 2 vs Vue 3

Le principal changement de Vue 3 est l'introduction du **Composition API** (vs **Option API**). Nous verrons des exemples dans les deux styles.

Comment choisir?

Voir la documentation de Vue ->

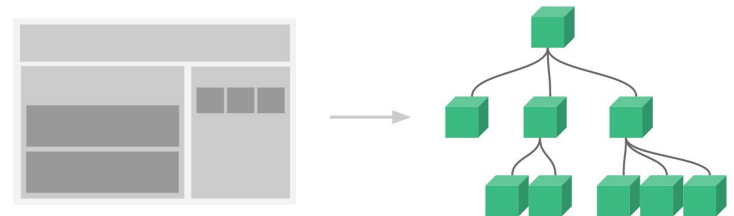
<https://vuejs.org/guide/introduction.html#does-composition-api-cover-all-use-cases>

<https://vuejs.org/guide/extras/composition-api-faq.html#what-is-composition-api>

Les deux sont **équivalents** - à vous de choisir celui qui vous rejoint le plus (OOP vs Functional).

Components

Pour développer une application complète de style **Single Page App**, on devra combiner plusieurs petites applications MVVM appelés **Components**.



Single File Component

Combiner CSS, HTML et JavaScript dans un fichier .vue

<https://vuejs.org/guide/scaling-up/sfc.html>

Conditions

Les opérations dans les templates Vue sont toujours préfixés par **v-**

<https://vuejs.org/guide/essentials/conditional.html>

v-if

```
<div id="app-3">
  <p v-if="seen">Now you see me</p>
</div>
```

```
let app3 = new Vue({
  el: '#app-3',
  data: {
    seen: true
  }
})
```

<https://jsfiddle.net/2u438meg/>

Boucles

Les opérations dans les templates Vue sont toujours préfixés par **v-**

<https://vuejs.org/guide/essentials/list.html#v-for>

v-for

```
<div id="app-4">
  <ol>
    <li v-for="todo in todos">
      {{ todo.text }}
    </li>
  </ol>
</div>
```

```
let app4 = new Vue({
  el: '#app-4',
  data: {
    todos: [
      {text: 'Learn JavaScript'},
      {text: 'Learn Vue'},
      {text: 'Build something awesome'}
    ]
  }
})
```

<https://jsfiddle.net/1kOrscah/>

Évènements

Clicks, etc.

Les opérations dans les templates Vue sont toujours préfixés par **v-** ou la notation **@**

<https://vuejs.org/guide/essentials/event-handling.html>

```
<div id="app-5">
  <p>{{ message }}</p>
  <button v-on:click="reverseMessage">Reverse Message</button>
</div>
```

```
let app5 = new Vue({
  el: '#app-5',
  data: {
    message: 'Hello Vue.js!'
  },
  methods: {
    reverseMessage: function () {
      this.message = this.message.split('').reverse().join('')
    }
  }
})
```

<https://jsfiddle.net/62bcyxno/>

Two-way binding

Entrée utilisateur

Les opérations dans les templates Vue sont toujours préfixés par **v-**

<https://vuejs.org/guide/essentials/forms.html>

```
<div id="app-6">
  <p>{{ message }}</p>
  <input v-model="message">
</div>
```

```
let app6 = new Vue({
  el: '#app-6',
  data: {
    message: 'Hello GLO-3102!'
  }
})
```

Imaginez ce simple composant en JavaScript pur! Vue nous simplifie grandement la vie.

<https://jsfiddle.net/xn9uph1u/>

Components

```
<ol>
  <!-- Create an instance of the todo-item component -->
  <todo-item></todo-item>
</ol>
```

```
Vue.component('todo-item', {
  template: '<li>This is a todo</li>'
})
```

Évidemment, ceci donne un composant “statique”, donc le contenu sera toujours le même.

Components (Options API)

Passer des paramètres aux composants

```
Vue.component('todo-item', {
  props: ['todo'],
  template: '<li>{{ todo.text }}</li>'
})

let app7 = new Vue({
  el: '#app-7',
  data: {
    groceryList: [
      { id: 0, text: 'Vegetables' },
      { id: 1, text: 'Cheese' },
      { id: 2, text: 'Whatever else humans are
supposed to eat' }
    ]
  }
})
```

```
<div id="app-7">
  <ol>
    <todo-item
      v-for="item in groceryList"
      v-bind:todo="item"
      v-bind:key="item.id">
    </todo-item>
  </ol>
</div>
```

<https://jsfiddle.net/0h975o0g/>

Components (Options API)

Passer des paramètres aux composants

- **data**
 - Propriété interne au component - non exposée à l'externe
- **props**
 - Propriétés exposées à l'externe du component - permet aux composants parent d'injecter des données
- **computed**
 - Propriétés internes composées à partir de d'autres propriétés
- **watch**
 - Fonctions internes qui réagissent à partir de changements au composant.

<https://vuejs.org/guide/components/props.html>

<https://vuejs.org/guide/essentials/computed.html>

<https://vuejs.org/guide/essentials/watchers.html>

Communication

Entre composants

Utilise la syntaxe `$.emit`. Ici un composant enfant (le composant **blog-post**):

<https://vuejs.org/guide/components/events.html#emitting-and-listening-to-events>

```
<button v-on:click="$emit('enlarge-text', 0.1)">
  Enlarge text
</button>
```

Et son parent:

```
<blog-post
  ...
  v-on:enlarge-text="onEnlargeText"
></blog-post>
...

methods: {
  onEnlargeText: function (enlargeAmount) {
    this.postFontSize += enlargeAmount
  }
}
```

Components (Composition API)

Le **Composition API** vise à remplacer l'approche orientée-objet vers une approche fonctionnelle, un peu à la manière de React. Il s'agit d'une manière un peu différente de faire des composants.

Composition over Inheritance

Remplace les data/computed par la simple méthode **setup**.

```
export default {  
  setup() {  
    const query = ref('')  
  
    return {  
      query,  
    }  
  },  
}
```

Components (Composition API)

L'utilisation de méthodes devient ainsi

```
export default {  
  setup() {  
    const query = ref('')  
  
    const reset = (evt) => {  
      query.value = '' // clears the query  
    }  
  
    return {  
      reset,  
      query,  
    }  
  },  
}
```

Components (Composition API)

Avec **props** et **context**

```
import { toRefs, toRef } from 'vue'

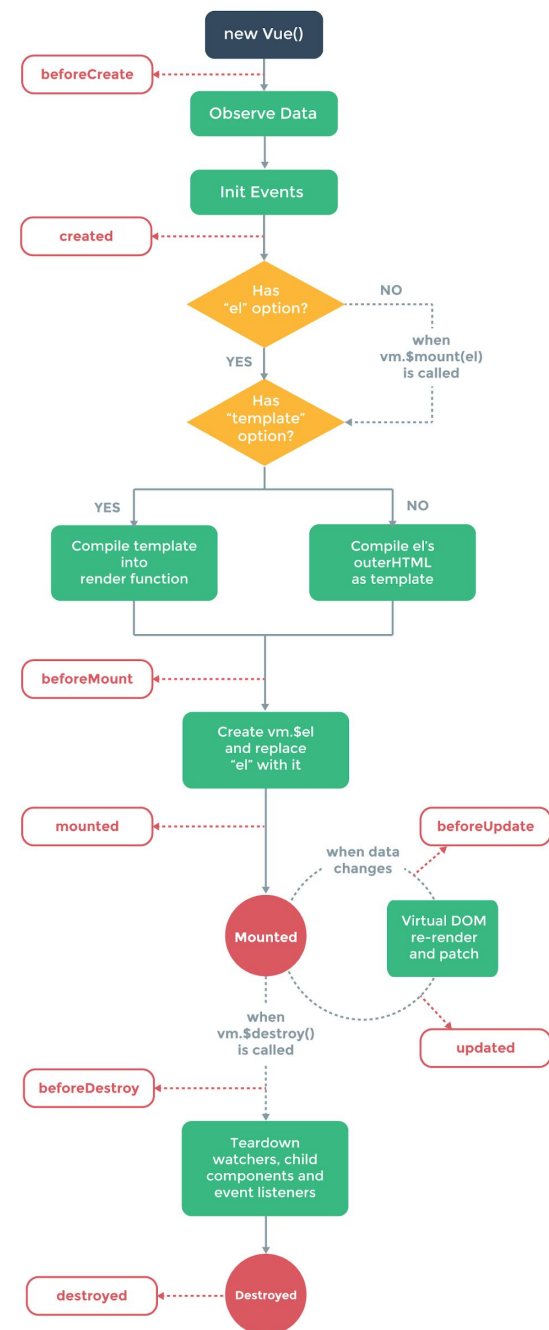
export default {
  setup(props) {
    const { title } = toRefs(props)

    console.log(title.value)

    // Alternative
    const title = toRef(props, 'title')
  }
}
```


Cycle de vie d'un component (Options API)

beforeCreate
created
beforeMount
mounted
beforeUpdate
updated
beforeDestroy
destroyed



Lifecycle (Options API)

Exemple

```
const vm = new Vue({
  data: {
    a: 1
  },
  created: function () {
    // `this` points to the vm instance
    console.log('a is: ' + this.a)
  }
})

// -> "a is: 1"
```

<https://vuejs.org/guide/essentials/lifecycle.html>

1

□ □ □



Lifecycle (Composition API)

Example

```
import { onMounted } from 'vue'
export default {
  setup() {
    onMounted(() => {
      console.log('mounted')
    })
  },
}
```

<https://vuejs.org/api/composition-api-lifecycle.html>

Composition API: Réutilisation

Le **Composition API** permet également de faire des *composables* - très similaires aux fameux *hooks* de React. Les composables permettent de réutiliser certains comportements entre les composants.

Voir exemple ici -

<https://vuejs.org/guide/reusability/composables.html#mouse-tracker-example>

Ce qui a amené à des librairies de *composables* réutilisables

<https://vueuse.org/guide/>

Routeur

Ajouter le support des URLs à votre application

Dans un contexte d'une **single-page application**, il n'y a qu'une seule *page* html, donc un seul URL pour accéder à l'application.

Il est fort probable que vous voudriez avoir des URLs uniques pour les différentes sections de votre application afin de permettre:

- Signets (bookmarks)
- Liens à partir d'autres pages/sites
- etc

Routeur

Ajouter le support des URLs à votre application

Il est possible de simuler des URL uniques avec un routeur. Nous allons utiliser le # (*hash*) de l'URL pour permettre de toujours charger le document index.html et interpréter le hash avec le javascript pour afficher la page correspondante.

`window.location.hash` permet de lire la valeur du hash

`window.onhashchange` permet d'écouter sur les changement du hash

Il existe plusieurs librairies qui offrent ces fonctionnalités (et beaucoup plus).

Ex: vue-router, react-router, backbone-router, etc.

vue-router

Ajouter le support des URLs à votre application Vue JS

router.vuejs.org

vue-router

```
import Vue from 'vue';
import Router from 'vue-router';
import Home from '@/components/Home';
import Album from '@/components/Album';
import Artist from '@/components/Artist';
```

```
Vue.use(Router);
```

```
export default new Router({
  routes: [
    {
      path: '/',
      name: 'Home',
      component: Home,
    }, {
      path: '/artist',
      name: 'Artist',
      component: Artist
    }, {
      path: '/album',
      name: 'Album',
      component: Album
    }
  ],
});
```

vue-router

```
const router = new VueRouter({
  routes: [
    // dynamic segments start with a colon
    { path: '/user/:id', component: User }
  ]
})

const User = {
  template: '<div>User {{ $route.params.id }}</div>'
}
```

<https://router.vuejs.org/guide/essentials/dynamic-matching.html>

Afficher des données du serveurs

Ajax avec "Promises"

```
<template>
  <ul v-if="posts && posts.length">
    <li v-for="post of posts">
      <p><strong>{{post.title}}</strong></p>
      <p>{{post.body}}</p>
    </li>
  </ul>
  <ul v-if="errors && errors.length">
    <li v-for="error of errors">
      {{error.message}}
    </li>
  </ul>
</template>

<script>
import axios from 'axios';

export default {
  data: () => ({
    posts: [],
    errors: []
  }),

  created() {
    axios.get(`http://jsonplaceholder.typicode.com/posts`)
      .then(response => {
        // JSON responses are automatically parsed.
        this.posts = response.data
      })
      .catch(e => {
        this.errors.push(e)
      })
  }
}
</script>
```

Afficher des données du serveurs

Ajax avec "async/await"

```
<template>
  <ul v-if="posts && posts.length">
    <li v-for="post of posts">
      <p><strong>{{post.title}}</strong></p>
      <p>{{post.body}}</p>
    </li>
  </ul>

  <ul v-if="errors && errors.length">
    <li v-for="error of errors">
      {{error.message}}
    </li>
  </ul>
</template>

<script>
import axios from 'axios';

export default {
  data: () => ({
    posts: [],
    errors: []
  }),

  async created () {
    try {
      const response = await axios.get(`http://jsonplaceholder.typicode.com/posts`)
      this.posts = response.data
    } catch (e) {
      this.errors.push(e)
    }
  }
}
</script>
```

Afficher des données du serveurs

Conseils

1. Remarquez que l'interaction avec le serveur est typiquement faite dans la méthode **created()**.

-> À cette étape, aucun DOM n'est encore créé, permet donc d'optimiser le rendering si la requête ne fonctionne pas, par exemple.
2. Assurez vous que vos appels serveurs sont fait dans des composants qui ne servent pratiquement qu'à ça.
-> Divisez l'affichage des données!

Stores

State management

Un des problèmes classiques de Vue (ou tout autre framework de composants) est que plus la complexité de votre app augmente, plus vos composants deviennent **pollués**.

=> Les composants en haut de la chaîne multiplient les **props inutiles**

=> Nécessité de partager du *state* entre les composants

Le pattern **flux** vient résoudre ce problème, mais il n'est vu que dans le cours 2. Ceci dit, sachez que les *stores* sont relativement faciles à intégrer dans votre application.

<https://vuejs.org/guide/scaling-up/state-management.html#simple-state-management-with-reactivity-api>

Pinia

State management

Pour des besoins plus complexes, Vue propose également Pinia, qui est le successeur de Vuex.

<https://pinia.vuejs.org/>

If your app is simple, you will most likely be fine without Pinia.



Examples

- TodoMVC
 - todomvc.com/examples/vue
 - tastejs/todomvc/tree/gh-pages/examples/vue
- Hacker news clone
 - <https://github.com/vuejs/vue-hackernews-2.0>
- [vuejs/awesome-vue](https://github.com/vuejs/awesome-vue)

| RTFM

<https://vuejs.org/guide/introduction.html>

**Excellents tutoriels vidéos
complets sur laracasts**

<https://laracasts.com/series/learn-vue-3-step-by-step>

