

WM

ESIR 2

WEB ENGINEERING : FRONTEND

2022 - 2023

PLAN

Généralités et rappel

Développement Frontend

HTML et CSS

JavaScript

Utilisation de Framework : Angular

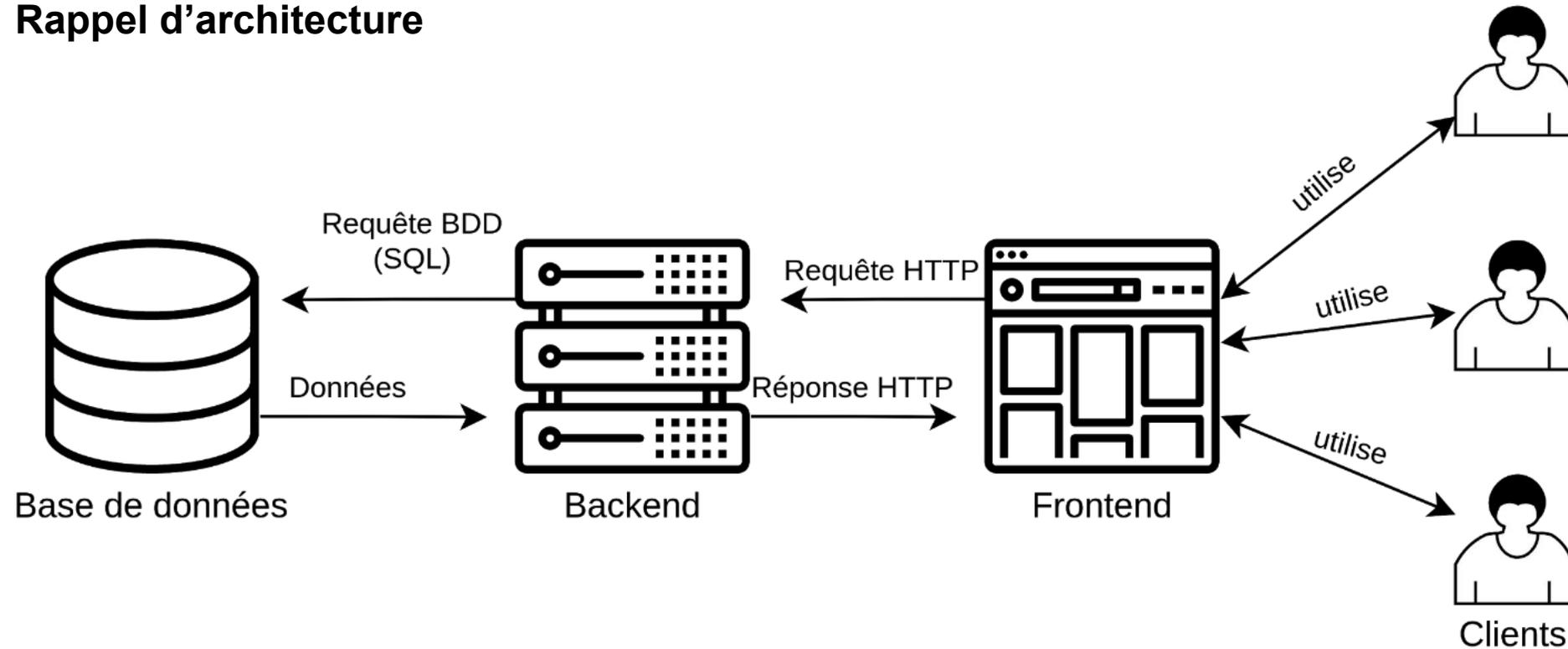
Conclusion

ORGANISATION

- Suite cours backend de Stéphanie Challita (WM – ESIR2)
- 4 séances de CM (8h)
- 2 séances de TP sur Angular (4h)
- 4 séances projet final (8h)
- 1 séance de TP pour l'évaluation du projet (2h)

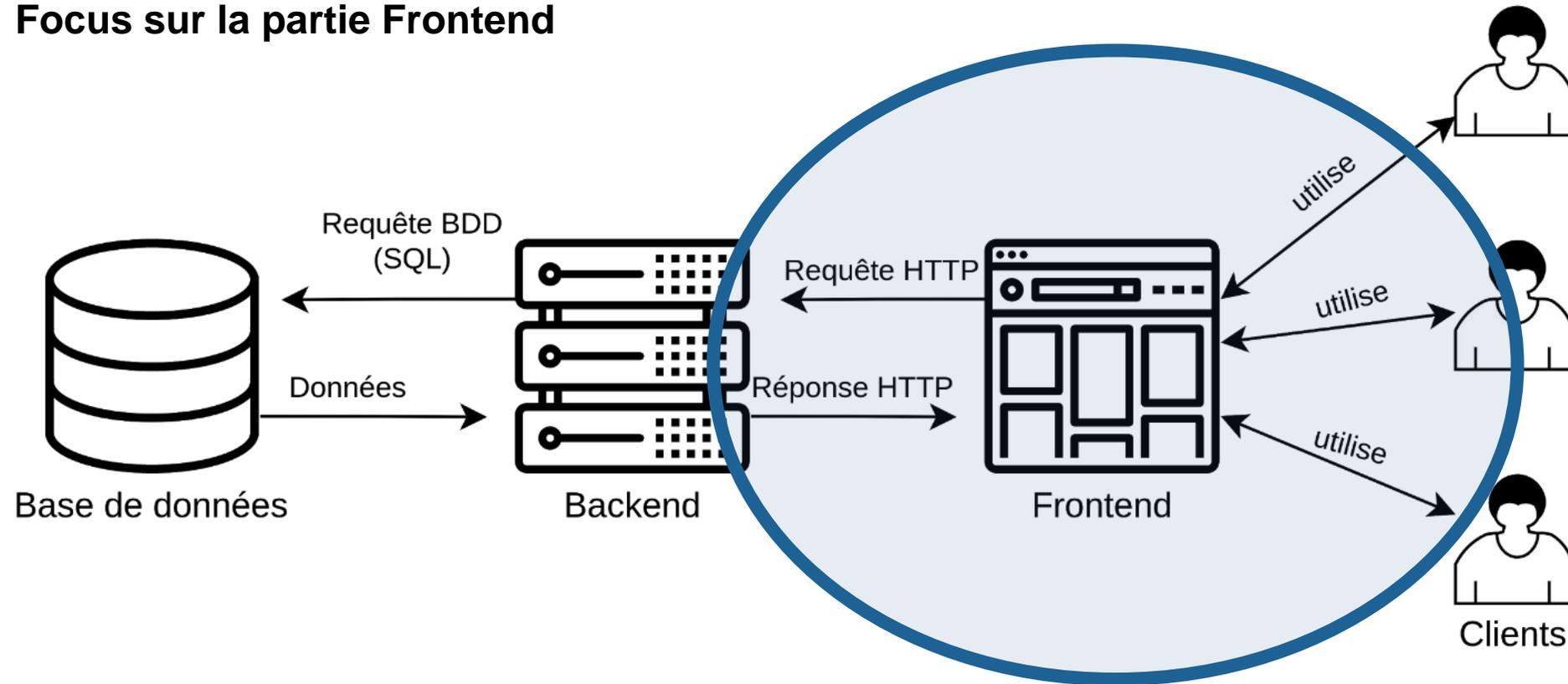
GÉNÉRALITÉS

Rappel d'architecture

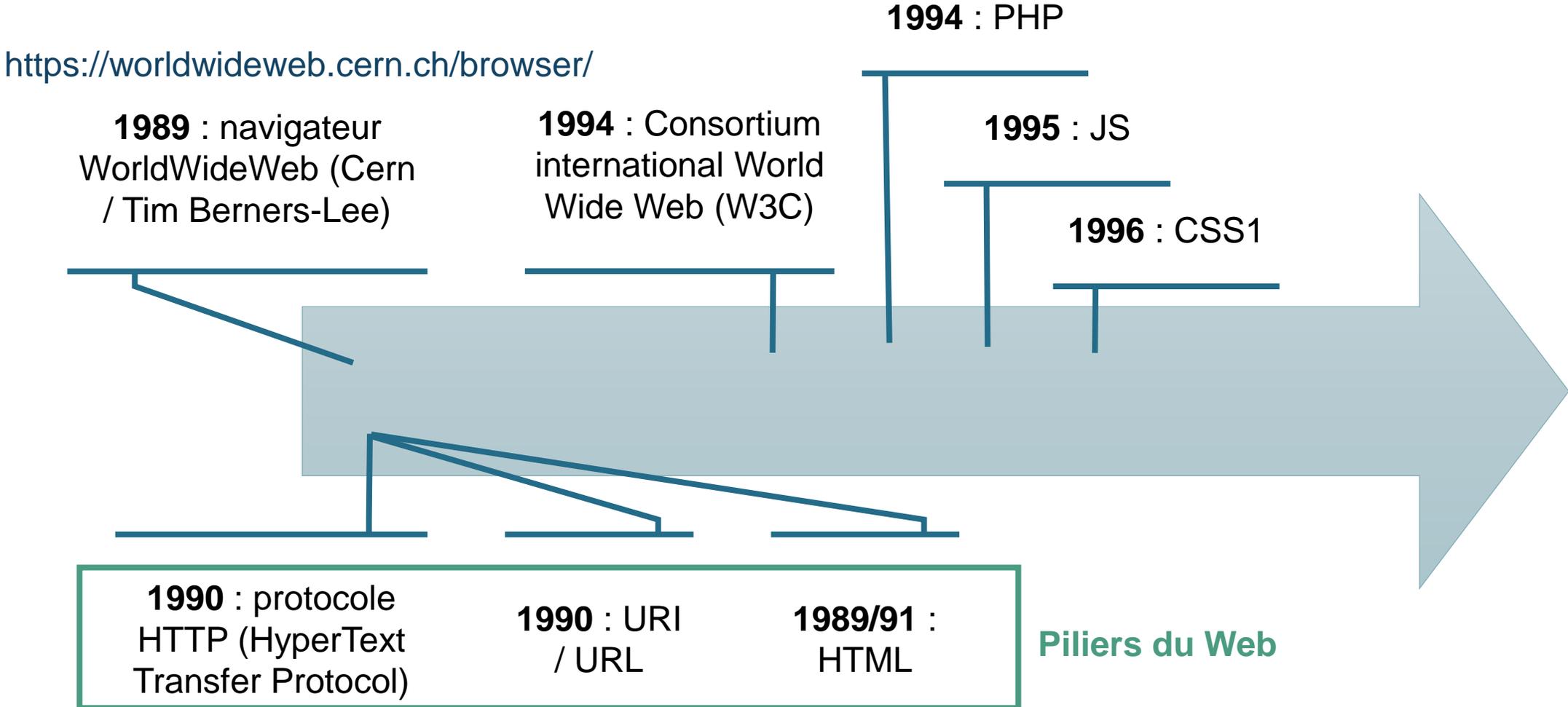


GÉNÉRALITÉS

Focus sur la partie Frontend



GÉNÉRALITÉS



GÉNÉRALITÉS

Evolution du Web

- Séparation mise en forme (CSS)
- Pages statiques vers interactions dynamiques
- Cloud computing, Web sémantique, Web embarqué, etc.
- HTML5 (Conteneur d'applications complexes)

GÉNÉRALITÉS

Evolution du Web

→ **Web 2.0** (web collaboratif, wiki, blog, etc.)

Web 3.0 avec le web sémantique et les objets connectés

→ **Moyens techniques**

Langage de scripts

Côté serveur (PHP, ASP, C#, etc.)

Côté client (Javascript, flash, etc.)

→ **Services Web**

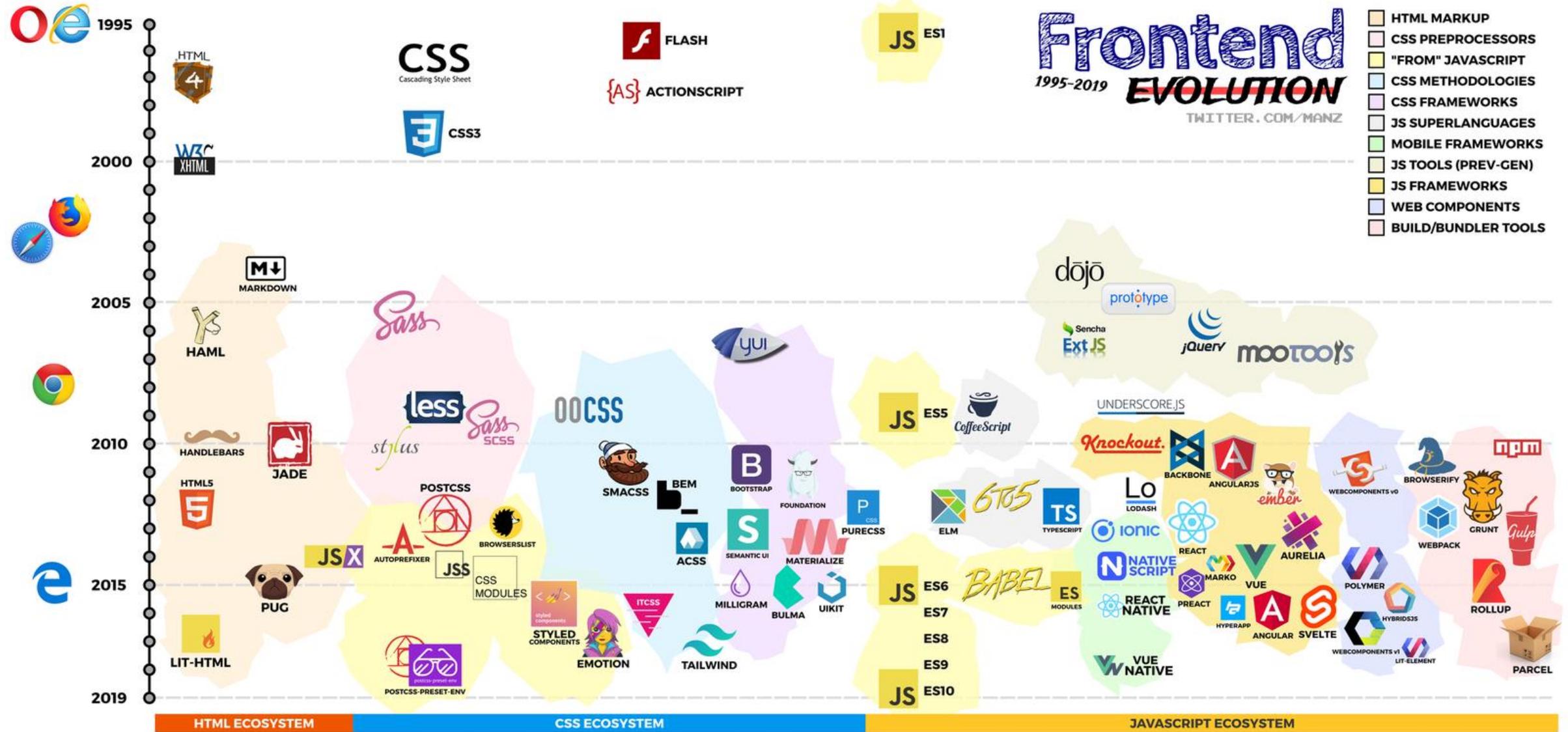
Échange de données entre applications (HTML / XML, JSON)

GÉNÉRALITÉS

Frontend 1995-2019 EVOLUTION

TWITTER.COM/MANZ

- HTML MARKUP
- CSS PREPROCESSORS
- "FROM" JAVASCRIPT
- CSS METHODOLOGIES
- CSS FRAMEWORKS
- JS SUPERLANGUAGES
- MOBILE FRAMEWORKS
- JS TOOLS (PREV-GEN)
- JS FRAMEWORKS
- WEB COMPONENTS
- BUILD/BUNDLER TOOLS



GÉNÉRALITÉS

Pourquoi utiliser des frameworks ?

→ Simplifier la vie du développeur et réduire le coup

Différents types d'architectures (push vs pull based architecture) :

→ Basé sur des actions (Django, Ruby on Rails, Symfony)

→ Basé sur des composants (Vue, Angular, React)

OBJECTIFS

- Fondamentaux sur le frontend web (Single Page Application)
- Rappel sur les technologies de base du web (HTML/CSS/JS)
- Notion de composants web
- Panorama des frameworks JavaScript (Angular)

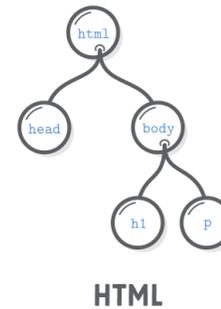
DÉVELOPPEMENT FRONTEND

Les piliers du web

Architecture front-end : MPA - SPA

DÉVELOPPEMENT FRONTEND

→ HTML / CSS / JavaScript



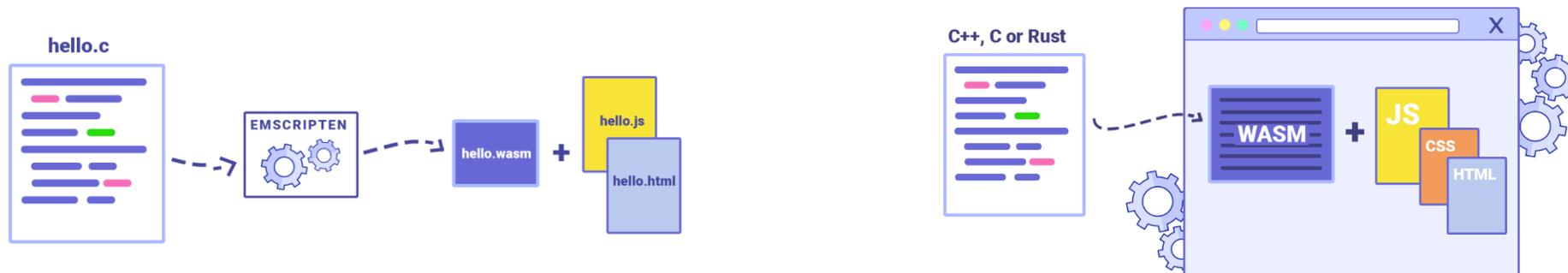
→ De nombreuses variations à JavaScript existent.

DÉVELOPPEMENT FRONTEND



Création du nouveau standard du web en 2019 : **WebAssembly** (wasm)

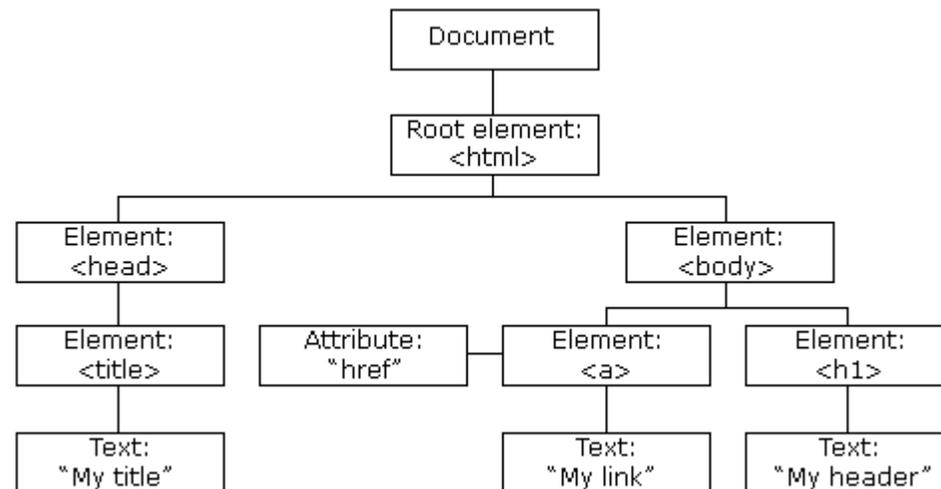
- Ne remplace pas JavaScript mais devient complémentaire.
- Bas niveau (de style assembleur) avec un format binaire (Code C++ de base transformé en binaire *.wasm* via *Emscripten*)
- Permet un gain de performance (format plus léger qu'un fichier JS, déjà compilé en amont, etc.)



DÉVELOPPEMENT FRONTEND

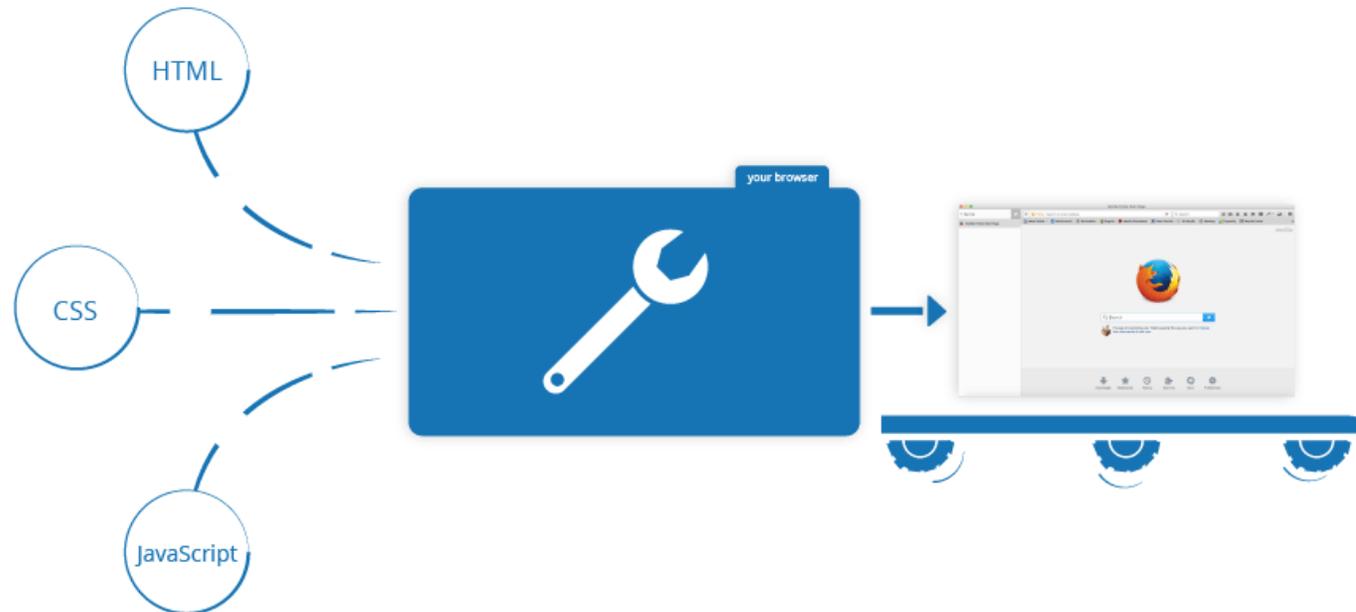
DOM (Document Object Model)

- Représentation objet des données qui composent la structure et le contenu d'un document sur le web.
- Il peut être manipulé à l'aide d'un langage script comme JavaScript.



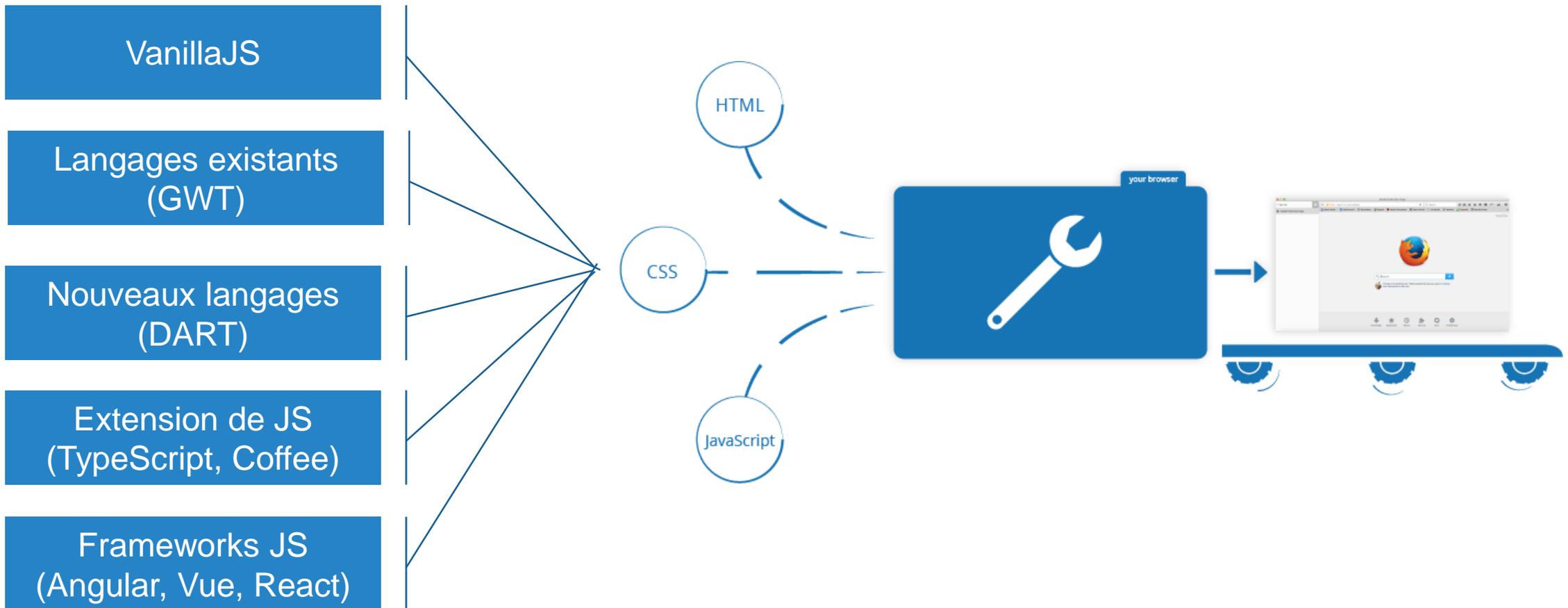
DÉVELOPPEMENT FRONTEND

- Lorsque la page web se charge dans un navigateur, les codes HTML, CSS et JavaScript s'exécutent dans un environnement.
- Le JavaScript est exécuté par le moteur JavaScript du navigateur, après que le HTML et le CSS ont été assemblés et combinés en une page web.



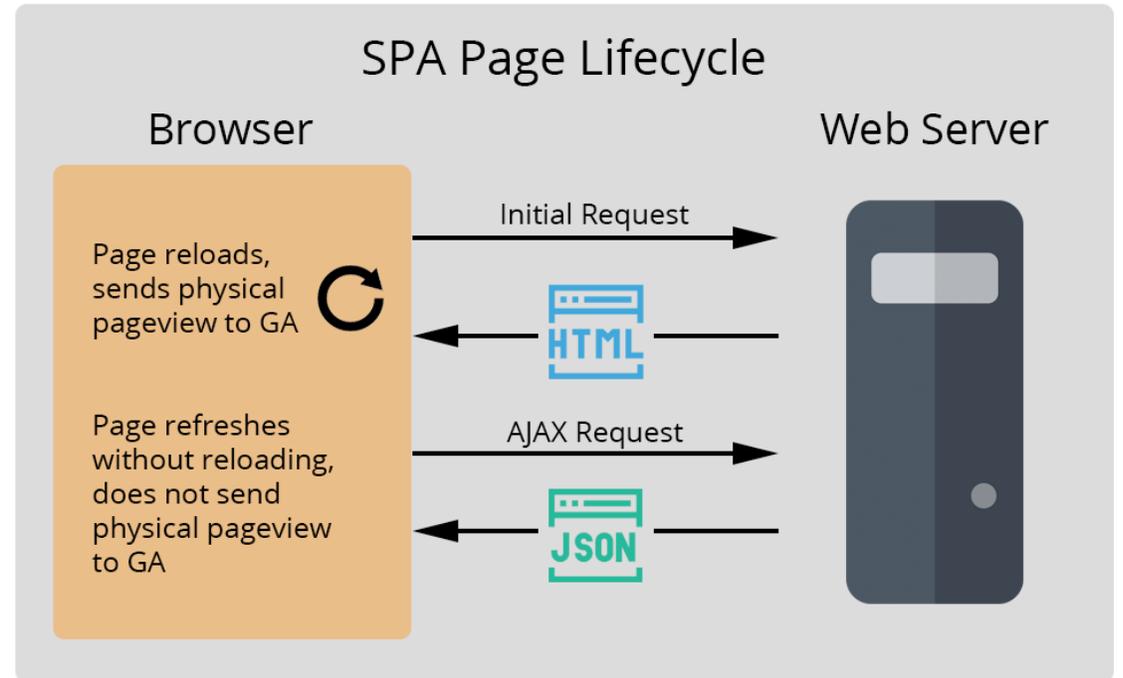
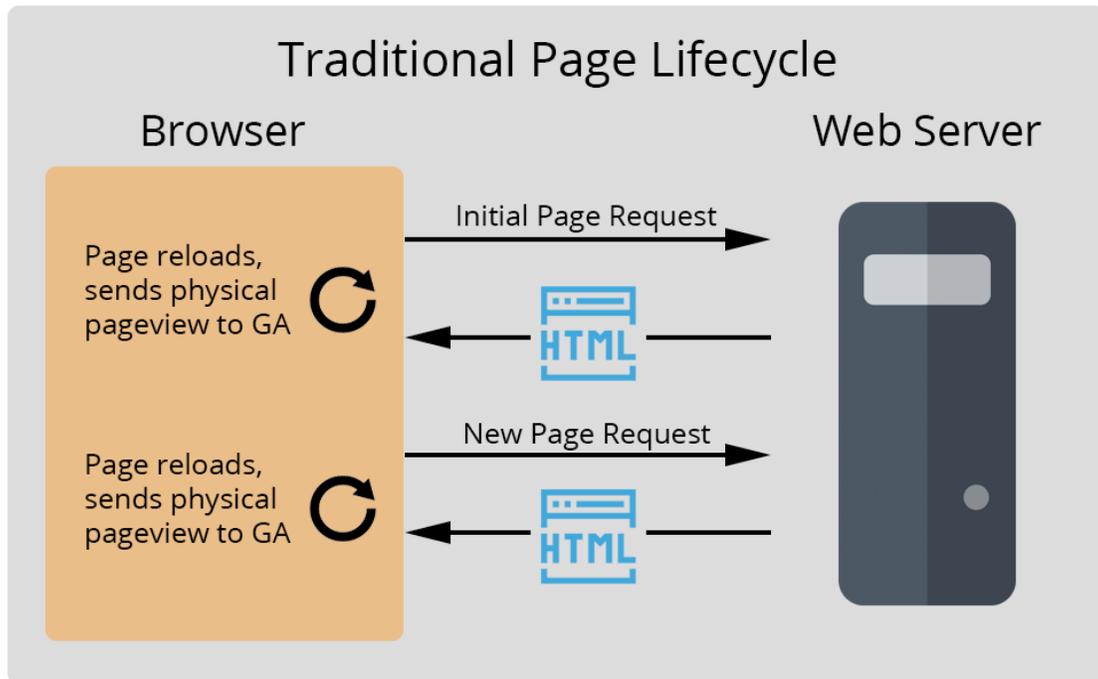
DÉVELOPPEMENT FRONTEND

→ Possibilité d'utiliser différentes technologies pour créer des applications web



MULTIPAGES APPLICATION VS SINGLE PAGE APPLICATION

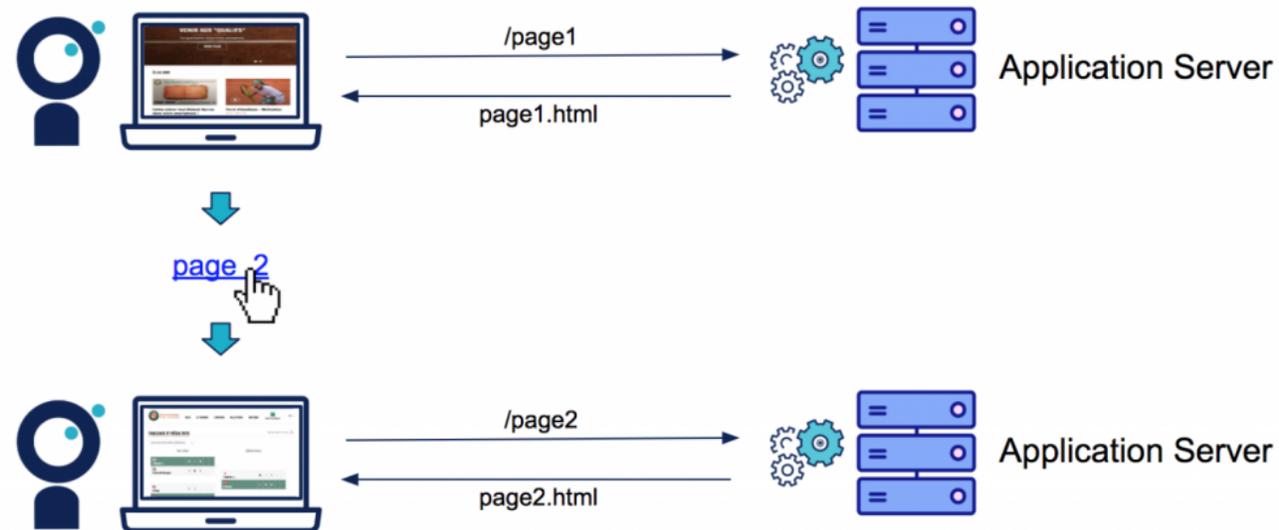
Différentes possibilités d'architecture pour le front



MULTIPAGES APPLICATION VS SINGLE PAGE APPLICATION

Multiple page application (MPA)

- Chaque action de l'utilisateur déclenche une requête HTTP vers le serveur.
- Rechargement complet de la page même si une partie du contenu reste inchangée.



MULTIPAGES APPLICATION VS SINGLE PAGE APPLICATION

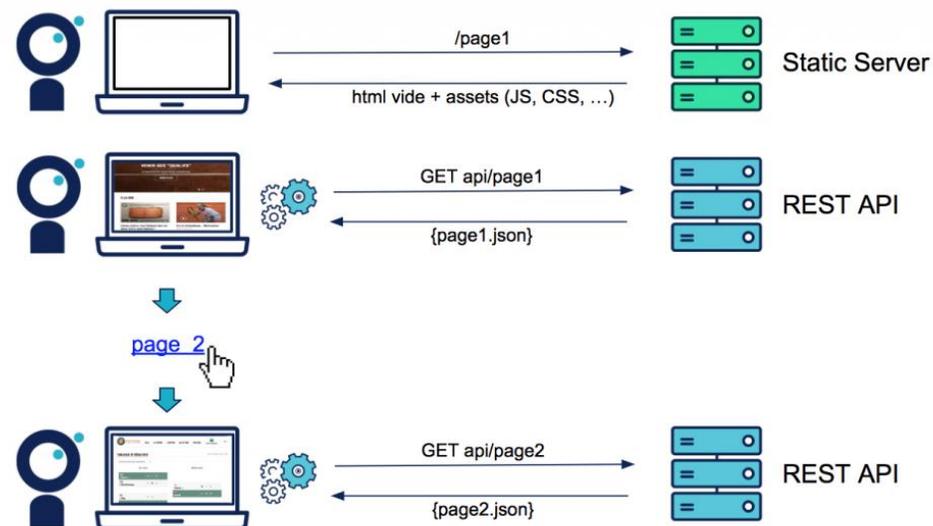
Single page application (SPA)

- Type de pattern apparu avec l'arrivée de nouvelle technologie (AJAX en 2004, jQuery en 2006, Google Chrome avec un nouveau moteur JavaScript V8, etc.).
- Utilisé par les frameworks AngularJS et Backbone.js (2010), Ember (2011), React (2013), Vue (2014) ou encore Angular (2016).

MULTIPAGES APPLICATION VS SINGLE PAGE APPLICATION

Single page application (SPA)

- L'ensemble des éléments de l'application est chargé sur le navigateur. Tous les templates du site sont donc pré-chargés.
- Navigation côté client uniquement (émulation d'une navigation).



HTML

Langage de balises

Anatomie d'un document HTML

Rappel formulaire

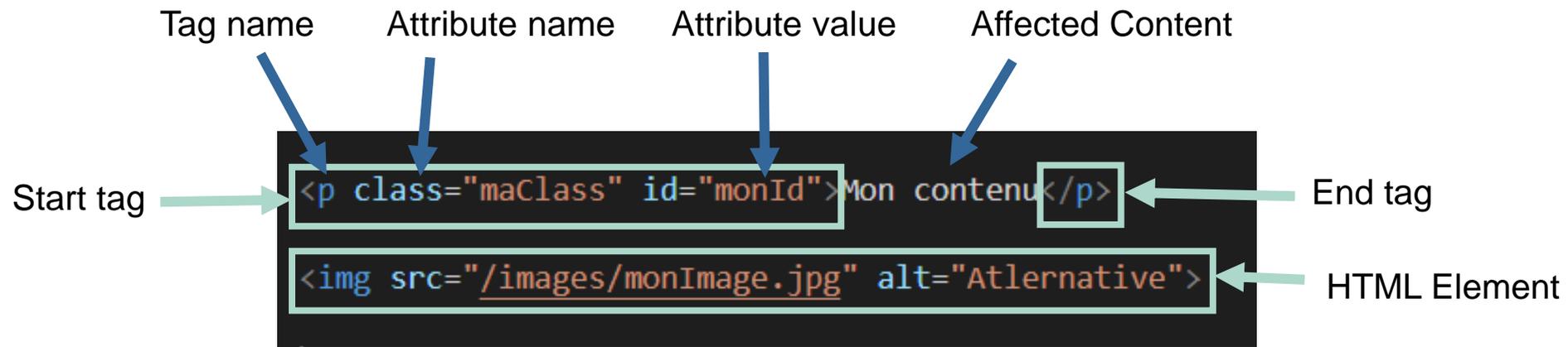
Notion de template



HTML 5

→ **HyperText Markup Language (HTML)** : code utilisé pour structurer une page web et son contenu.

→ **Langage de balises**



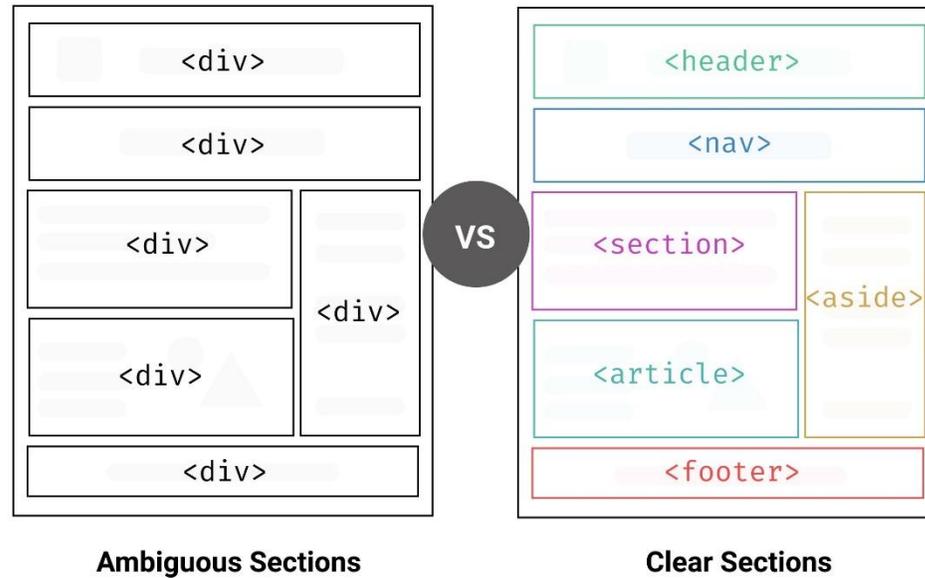
HTML5

→ Structure simple d'une page HTML5 :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <!-- Metadata -->
    <meta charset="utf-8">
  </head>
  <body>
    <!-- Contenu -->
  </body>
</html>
```

HTML5

→ Structure d'une page HTML : balises header, nav, section, article, aside et footer



HTML5

→ Formulaire : balise `<form></form>`

Mon Formulaire

Les champs obligatoires sont suivis de *.

Information du produit

Fruit juice size

Small

Medium

Large

J'aime les cerises

Information paiement

Nom :*

Type de carte :

```
<fieldset>
  <legend>Fruit juice size</legend>
  <p> <input type="radio" name="size" id="size_1" value="small"> <label for="size_1">Small</label> </p>
  <p> <input type="radio" name="size" id="size_2" value="medium"> <label for="size_2">Medium</label> </p>
  <p> <input type="radio" name="size" id="size_3" value="large"> <label for="size_3">Large</label> </p>
</fieldset>
```

```
<label for="taste_1">J'aime les cerises</label>
<input type="checkbox" id="taste_1" name="taste_cherry" value="1">
```

```
<label for="username">Nom :<abbr title="required">*</abbr></label>
<input id="username" type="text" name="username" required>
```

```
<select id="card" name="usercard">
  <option value="visa">Visa</option>
  <option value="mc">Mastercard</option>
  <option value="amex">American Express</option>
</select>
```

```
<button type="submit">Valider</button>
```

HTML5

Formulaire : validation des données côté HTML

Champ *required*

```
<label for="choose">Login ?</label>  
<input id="choose" name="i_like" required>
```

Expression régulière

```
<label for="choose">Mot de passe ? </label>  
<input id="choose" name="mdp" required pattern="[A-F][0-9]{5}">
```

Champ *input*

```
<label for="t2">Adresse électronique ?</label>  
<input type="email" id="t2" name="email">
```

→ API de validation des contraintes disponible en JavaScript

HTML5

Formulaire : envoi du formulaire

- A travers l'attribut action du formulaire.
- En utilisant des requêtes AJAX (e.g. via un objet FormData avec XMLHttpRequest).

HTML5

Balise template :

- Mécanisme utilisé pour stocker côté client du contenu HTML qui ne doit pas être affiché lors du chargement de la page.
- Permet de stocker des modèles de code HTML pouvant être clonés et collés dans un document à l'aide de scripts JS (structure répétitive).

```
<template id="monParagraphe">  
  <h2>JXC</h2>  
</template>
```

```
let template = document.getElementById('monParagraphe');  
let templateContent = template.content.cloneNode(true);  
document.body.appendChild(templateContent);
```

Template

Show hidden content

DESIGN

CSS

Responsive design

Pré et Post-processeurs CSS (SASS, LESS)

Framework CSS (Bootstrap)



DESIGN

- **Le webdesign doit être réfléchi** : UX et UI designer (e.g. expérience utilisateur à travers l'ergonomie et la navigation du site, réalisation de maquettes pour définir la chartre graphique).
- **A éviter** : un design complexe ou chargé, des polices d'écriture trop petites, des couleurs trop marquées, une navigation peu ergonomique, des pop-ups ou bannières trop agressives.



CSS

→ **Cascading Style Sheets** : code utilisé pour mettre en forme une page web.

→ Insertion d'une feuille de style dans un fichier html :

```
<link rel="stylesheet" href="css/monCSS.css" />
```

→ Code CSS inline à éviter :

```
<h1 style="color: blue;background-color: yellow;border: 1px solid black;">Hello World!</h1>
```

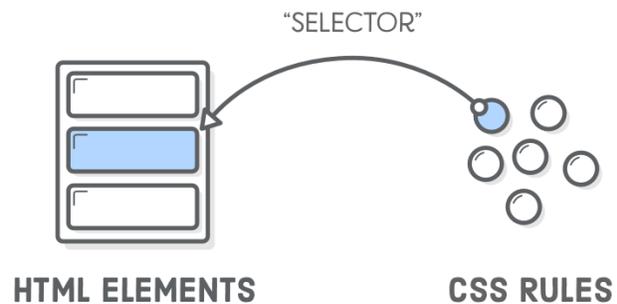
→ Possibilité d'appliquer un style à tous les éléments du même type, plusieurs éléments, un élément particulier.

```
p {  
  color: red;  
  width: 500px;  
  border: 1px solid black;  
}
```

```
p,li,h1 {  
  color: red;  
}
```

CSS : STRUCTURE

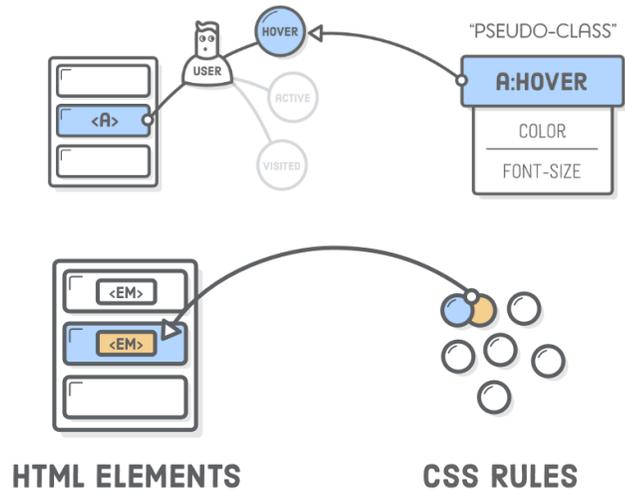
- Sélecteur (élément HTML)
- Déclaration (règle simple qui détermine les propriétés de l'élément mis en forme)
- Propriétés
- Valeur de la propriété



```
sélecteur  
p.important {  
  propriété font-weight: valeur bold;  
  border: 2px solid red;  
}
```

règle

CSS : SÉLECTEUR



```
#maDiv {
  background-color: #15DEA5;
}
.button {
  background-color: #DB464B;
}
p .maClass {
  background-color: steelblue;
}
a:hover{
  color: cornflowerblue;
}
```

Selector	Example
Type selector	h1 { }
Universal selector	* { }
Class selector	.box { }
id selector	#unique { }
Attribute selector	a[title] { }
Pseudo-class selectors	p:first-child { }
Pseudo-element selectors	p::first-line { }
Descendant combinator	article p
Child combinator	article > p
Adjacent sibling combinator	h1 + p
General sibling combinator	h1 ~ p

RESPONSIVE DESIGN

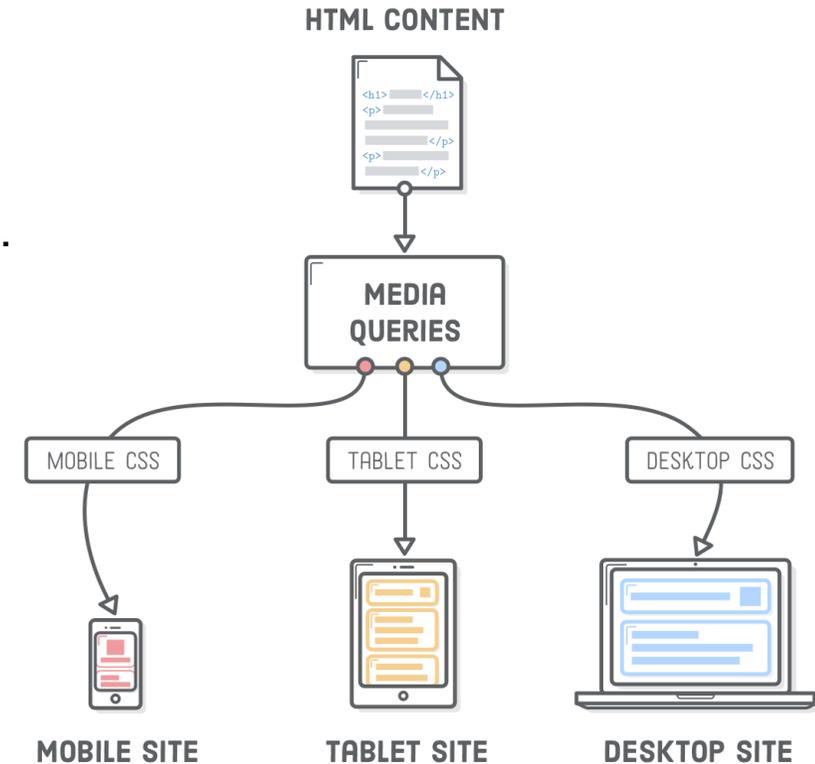
Comment adapter le design aux différents types de supports et résolution d'écran ?

→ Une solution : les **media queries**

Moyen d'appliquer conditionnellement des règles CSS.

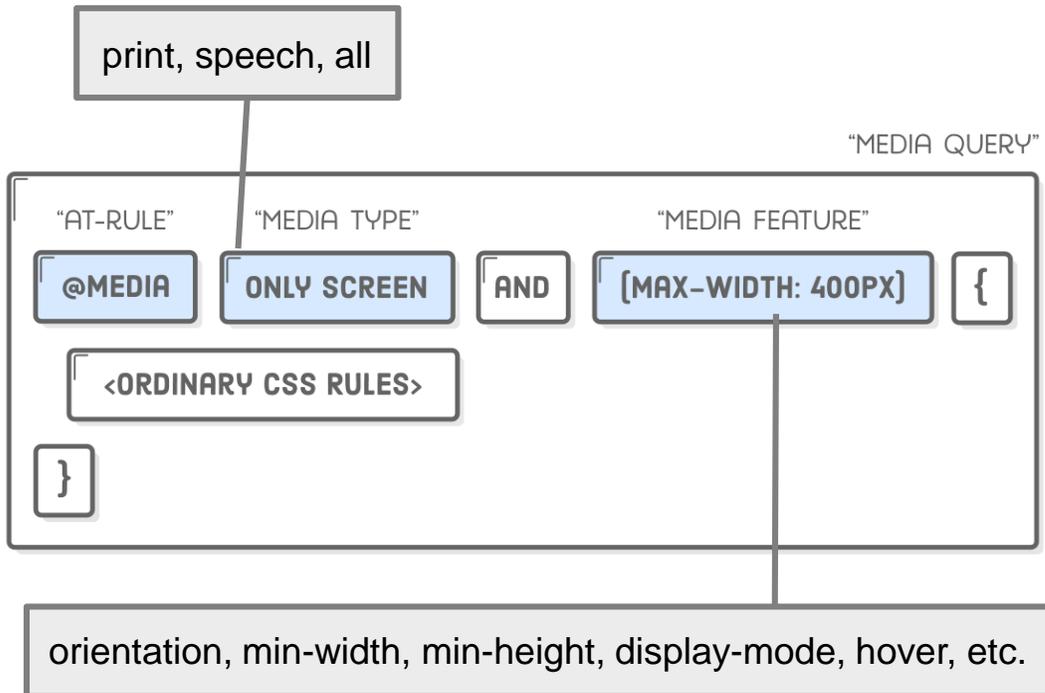
Utilisation des mêmes éléments HTML

→ Généralement débiter par la mise en page mobile.



RESPONSIVE DESIGN

→ Media query



```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

/* Mobile Styles */
@media only screen and (max-width: 400px) {
  body {
    background-color: #F09A9D; /* Red */
  }
}

/* Tablet Styles */
@media only screen and (min-width: 401px) and (max-width: 960px) {
  body {
    background-color: #F5CF8E; /* Yellow */
  }
}

/* Desktop Styles */
@media only screen and (min-width: 961px) {
  body {
    background-color: #B2D6FF; /* Blue */
  }
}
```

RESPONSIVE DESIGN

- Zoom automatique sur les navigateurs mobiles pour ajuster la page dans la largeur.
- Possibilité de le désactiver si le site est responsive :

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

37

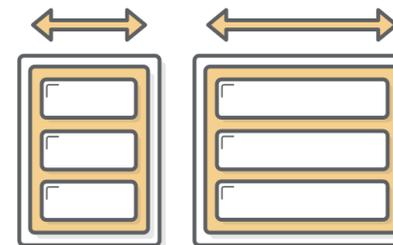
- Attention aux unités de longueur pour un design adaptable (pour la taille du texte par exemple), au type de positionnement des éléments et leur conteneur (Grid, FlexBox, etc.)



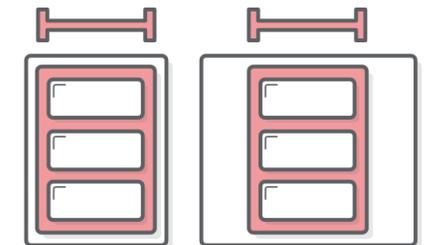
"FLEX CONTAINER"



"FLEX ITEMS"



FLUID LAYOUT



FIXED-WIDTH LAYOUT

DESIGN

CSS

Responsive design

Pré et Post-processeurs CSS (SASS, LESS)

Framework CSS (Bootstrap)



PRE-PROCESSEUR : SASS



→ SASS (**S**yntactically **A**wesome **S**tylesheets) permet d'ajouter à CSS diverse fonctionnalités permettant d'organiser de manière plus maintenable les feuilles de style.

→ Commande pour compiler les fichiers sass en css

```
sass --watch input.scss output.css
```

Outils à disposition pour organiser et gérer les styles :

→ Variables

→ Imbrication des sélecteurs

→ Fonctions

→ Directives @import



PRE-PROCESSEUR : SASS

→ Hiérarchie entre les sélecteurs

```
nav ul {  
  margin: 0;  
  padding: 0;  
  list-style: none;  
}  
nav li {  
  display: inline-block;  
}  
nav a {  
  display: block;  
  padding: 6px 12px;  
  text-decoration: none;  
}
```



```
nav {  
  ul {  
    margin: 0;  
    padding: 0;  
    list-style: none;  
  }  
  
  li { display: inline-block; }  
  
  a {  
    display: block;  
    padding: 6px 12px;  
    text-decoration: none;  
  }  
}
```

PRE-PROCESSEUR : SASS

→ Variables

```
body {  
  font: 100% Helvetica, sans-serif;  
  color: #333;  
}
```



```
$font-stack: Helvetica, sans-serif;  
$primary-color: #333;  
  
body {  
  font: 100% $font-stack;  
  color: $primary-color;  
}
```

PRE-PROCESSEUR : SASS

→ Utilisation de modules

```
body {
  font: 100% Helvetica, sans-serif;
  color: #333;
}

.inverse {
  background-color: #333;
  color: white;
}
```



```
// _base.scss
$font-stack: Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

```
// styles.scss
@use 'base';

.inverse {
  background-color: base.$primary-color;
  color: white;
}
```

PRE-PROCESSEUR : SASS

→ Utilisation des mixins

```
.info {
  background: █DarkGray;
  box-shadow: 0 0 1px █rgba(169, 169, 169, 0.25);
  color: █#fff;
}

.alert {
  background: █DarkRed;
  box-shadow: 0 0 1px █rgba(139, 0, 0, 0.25);
  color: █#fff;
}

.success {
  background: █DarkGreen;
  box-shadow: 0 0 1px █rgba(0, 100, 0, 0.25);
  color: █#fff;
}
```



```
@mixin theme($theme: █DarkGray) {
  background: $theme;
  box-shadow: 0 0 1px rgba($theme, .25);
  color: █#fff;
}

.info {
  @include theme;
}

.alert {
  @include theme($theme: █DarkRed);
}

.success {
  @include theme($theme: █DarkGreen);
}
```

LESS



- LESS (**L**eaner **S**tyle **S**heets) est un préprocesseur CSS, influencé par SASS.
- Permet également une plus grande facilité de gestion des feuilles de styles et supprime les répétitions de code.
- Concept similaires (imbrication, variable, héritages, mixins, etc.)
- Repose sur une syntaxe plus naturelle (ressemblant à CSS) par rapport à SASS qui utilise des mots clés (@use, @include, etc.).

BOOTSTRAP



- Framework open source utilisant Sass, développé au début par une équipe de Twitter.
- Compatible avec les différents navigateurs.
- Prend en charge la conception réactive et propose des modèles de conception prédéfinis.
- Pour inclure Bootstrap : CSS + JS (Bootstrap Bundle avec Popper)

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
```

```
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-ka7Sk0Gln4gmtz2MlQnikT1wXgYs0g+OMhuP+IlRH9sENB00LRn5q+8nbTov4+1p" crossorigin="anonymous"></script>
```

- Développé au début pour mobile (responsive design)

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

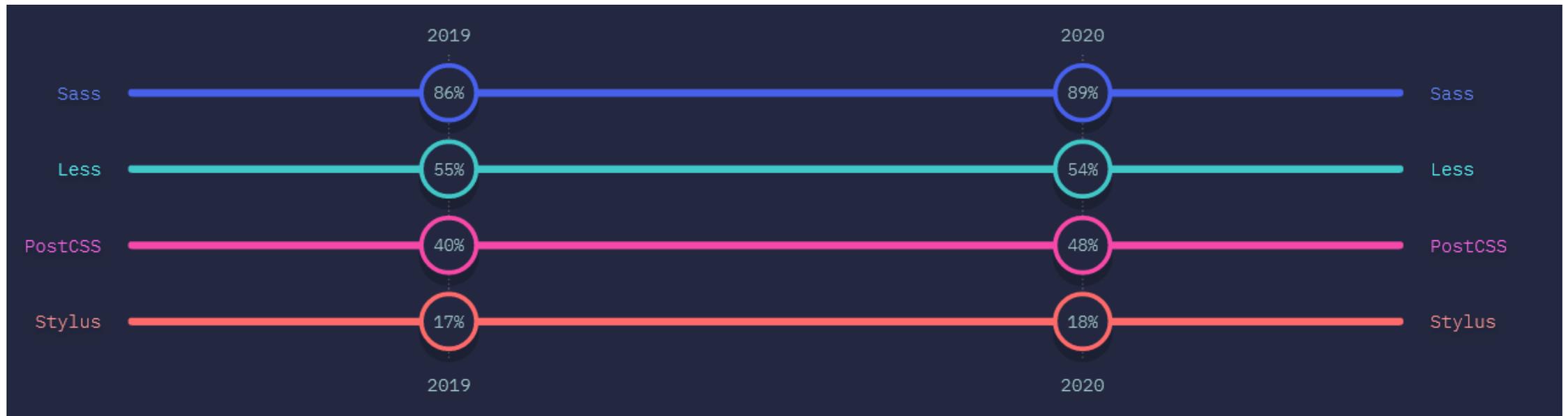
BOOTSTRAP : EXEMPLE

```
<div class="btn-group" role="group" aria-label="Basic mixed styles example">  
  <button type="button" class="btn btn-danger">Left</button>  
  <button type="button" class="btn btn-warning">Middle</button>  
  <button type="button" class="btn btn-success">Right</button>  
</div>
```



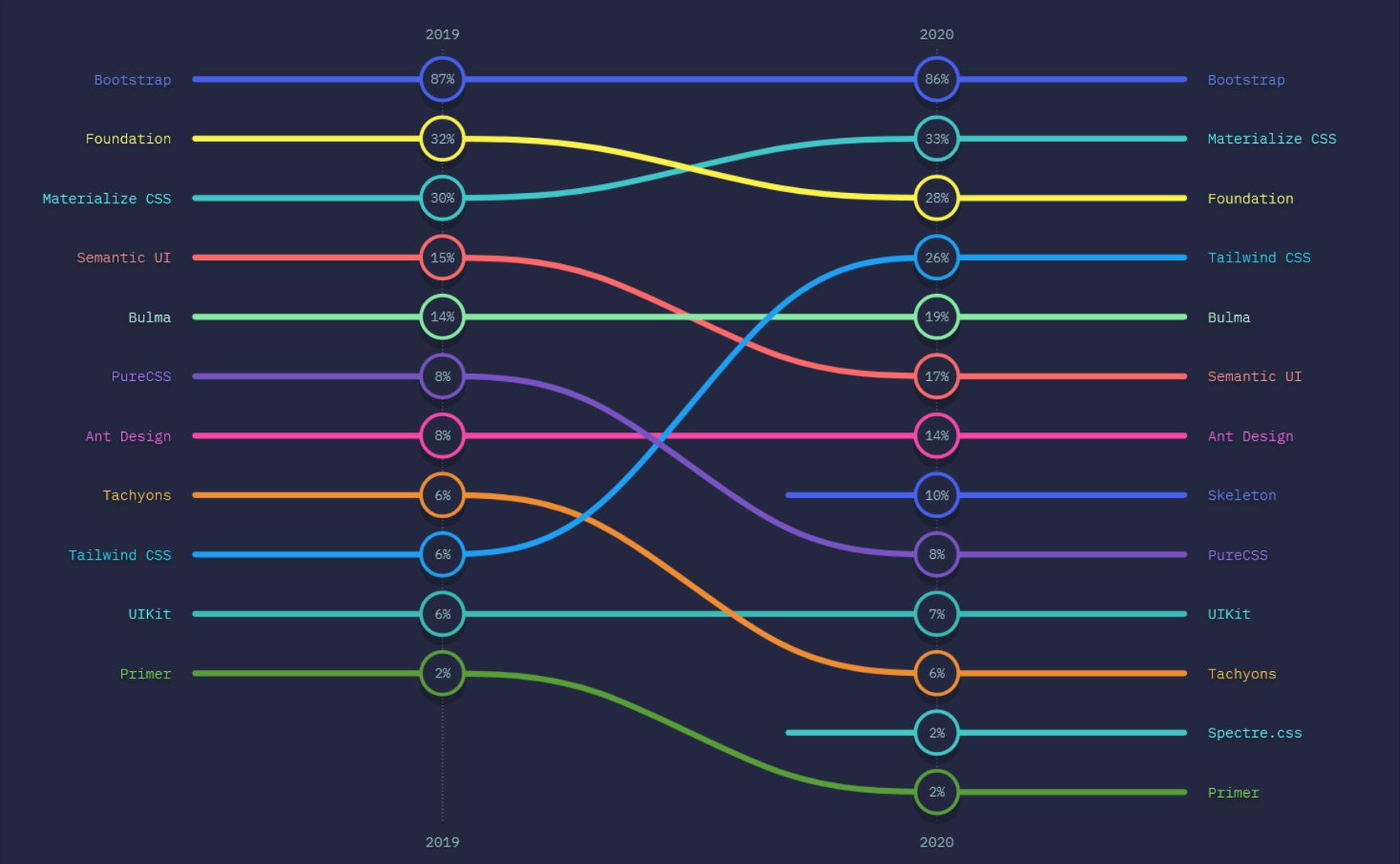
PRE-POSTPROCESSEUR CSS

→ Ratio d'utilisation



FRAMEWORK CSS

→ Ratio d'utilisation



JAVASCRIPT

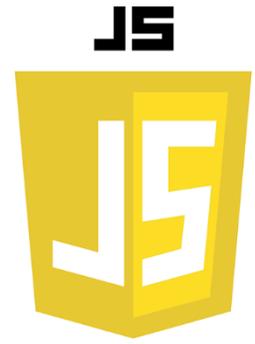
Définition

Syntaxe

Objets

Evènements

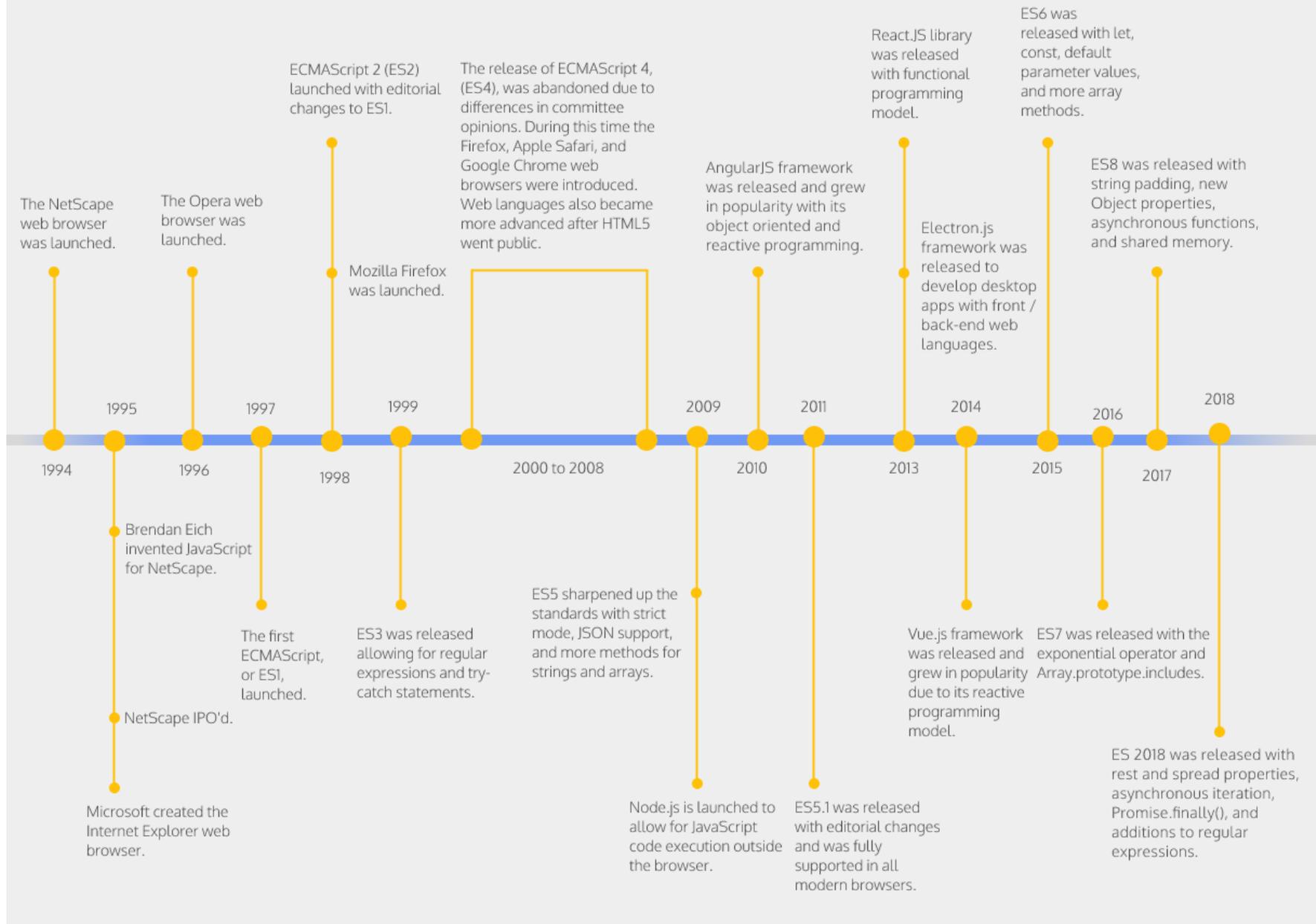
Accès à un éléments quelconque d'une page



JAVASCRIPT

- Développé par Netscape et Sun dans les années 95 (principal développeur : Brendan Eich, co-fondateur du projet Mozilla, de la Mozilla Foundation et Mozilla Corporation)
- Précédentes versions : Mocha (en interne), LiveScript, puis JavaScript
- Conçu à l'origine comme un langage de script complémentaire à Java
- Proposé à la standardisation à ECMA (*European Computer Manufacturers Association*) en 1996 (ECMA-262) :
 - **ES1** en 1997
 - **ES6** en 2015
 - **ES2022** en juin 2022





JAVASCRIPT

Langage populaire :

- Disponible dans tous les navigateurs (compatibilité)
- La plupart des navigateurs modernes supporte le standard ES6 (98% à 100%)
- Modules tiers disponibles grâce à NPM et GitHub : il existe des modules JavaScript pour quasiment chaque besoin.

Utilisation de JavaScript :

- Historiquement exécuté sur le navigateur web
- Node.js et NPM à la base du nouveau succès de JavaScript (serveur, local)

JAVASCRIPT

JavaScript permet :

- De modifier l'apparence de la page
- De communiquer avec le serveur
- D'enregistrer les actions de l'utilisateur
- De réagir aux événements utilisateur
- De sauvegarder des données

JAVASCRIPT

- Langage orienté **prototype** et non orienté objet :
 - Déclaration d'un objet générique (modèle), puis héritage
 - Notion de classe depuis ECMAScript 6

- Dynamique :
 - typage (faiblement typé),
 - fonctions,

- Évènementiel :
 - paradigme de programmation, attente puis réaction aux actions utilisateur

JAVASCRIPT

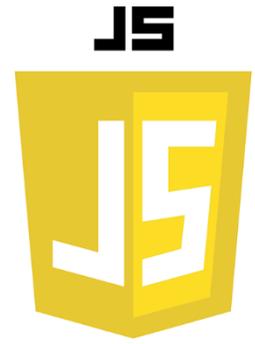
Définition

Syntaxe

Objets

Evènements

Accès à un éléments quelconque d'une page



JAVASCRIPT : SYNTAXE

Intégration dans la page :

→ En **interne** :

```
<script>alert("Hello World");</script>
```

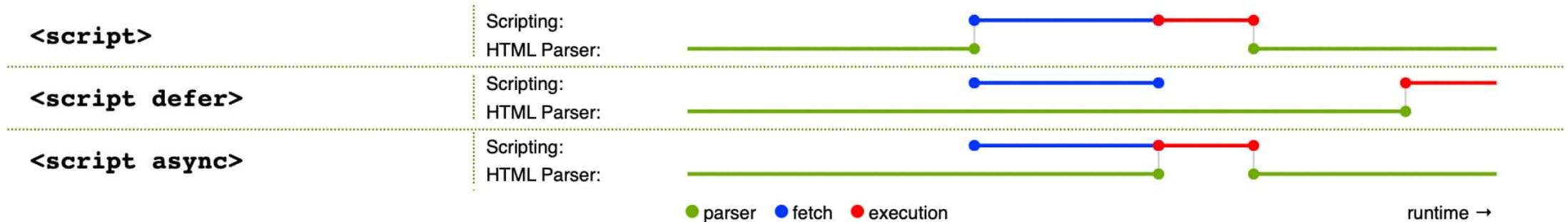
→ En **externe** :

```
<script src="js/fichierJS.js"></script>
```

→ Attention au **positionnement des balises** dans le fichier html.

Quand le navigateur rencontre un bloc JavaScript, il l'exécute dans l'ordre. Le code est **interprété**, le résultat du code exécuté est envoyé directement.

JAVASCRIPT : SYNTAXE



58

Depuis **ES6** : mot-clé **async** et **defer**

```
<script src="js/monScript.js" defer></script>
```

→ **async** : le navigateur continue de traiter l'HTML et le fichier JS est téléchargé en parallèle. Une fois chargé, le contenu est exécuté.

→ **defer** : diffère l'exécution du JavaScript jusqu'à ce que la page soit complètement chargée.

JAVASCRIPT : SYNTAXE

Bonne pratique :

- fichier *.js* externe
- lecture par évènement ***onload***,
- *separation of concerns* (Dijkstra):
 - pour le script principale
 - puis utilisation de fonction

JAVASCRIPT : SYNTAXE

→ **Syntaxe et opérateur semblable à C / C# / Java**

Opérateurs (+, *, !+, etc.)

Variables

Chaînes de caractères et Array

Structures conditionnelles

Structures itératives

Fonctions

JAVASCRIPT : SYNTAXE

Portées des variables

- **var** : permet de déclarer une variable dont la portée est le **contexte d'exécution courant** (**la fonction** qui contient la déclaration ou **le contexte global** si la variable est déclarée en dehors de toute fonction).
- **let** : permet de déclarer une variable dont la portée est **le bloc courant**. *let* crée une variable globale alors que *var* ajoute une propriété à l'objet global au niveau le plus haut.
- **const** : variable accessible qu'en lecture.

JAVASCRIPT : SYNTAXE

```
function varTest() {
  var x = 31;
  if (true) {
    var x = 71;
    console.log(x);
  }
  console.log(x);
}

function letTest() {
  let x = 31;
  if (true) {
    let x = 71;
    console.log(x);
  }
  console.log(x);
}
```

```
var x = 'global';
let y = 'global2';
console.log(this.x);
console.log(this.y);
console.log(y);
```

```
var a = 5;
var b = 10;

if (a === 5) {
  let a = 4;
  var b = 1;

  console.log(a);
  console.log(b);
}

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

JAVASCRIPT : SYNTAXE

Types de données :

- type **booléen** (true et false)
- Type **nul** (null)
- Type **indéfinie** (undefined)
- Type pour les **nombres** entiers ou réels (number)
- Type pour les **chaînes de caractères** (string)
- Type pour les **symboles** (depuis ES6 : type pour des données immuables et uniques)
- Type pour les **objets** (Object, avec par exemple Array)

```
let x;           //undefined
x = 5;          //number
x = "Toto";     //string
let myArray = [1, 3.3, "toto"];
let myHashTable = {a: 5, c:9.5, c:"toto"};
```

JAVASCRIPT : SYNTAXE

Objet :

- Chaque variable est considéré comme un objet.
- Méthodes existantes pour les objets (string et array par exemple)

String :

```
let x = "toto tata titi tutu tyty tete";  
console.log(x[3]);  
console.log(x.charAt(8));  
console.log("test".substring(1, 3));  
console.log("test".toUpperCase());
```

```
let y = 16 + 4 + "Volvo";  
let z = "Volvo" + 16 + 4;
```

```
parseInt("234");  
parseInt("2.99abc");
```

```
console.log("toto"=="tata");
```

JAVASCRIPT : SYNTAXE

Liste

Pour appliquer un traitement à chaque élément d'une liste :

→ *forEach(<fonction de callback>)*

→ *map(<fonction de callback>)*

```
let arr = [1, 2, 3, 4, 5];
console.log(arr.length);
arr.push(6);
arr.splice(3, 0, 7);
console.log(arr.indexOf(7));
arr.sort();
arr.forEach(element => {
  console.log(element);
});
console.log(arr.map(element => element*2 ));
```

JAVASCRIPT : SYNTAXE

Structures conditionnelles

→ structure **if ... else**

```
var a = 0;
var b = true;
if (typeof(a)=="undefined" || typeof(b)=="undefined") {
  document.write("Variable a or b is undefined.");
}
else if (!a && b) {
  document.write("a==0; b==true;");
} else {
  document.write("a==" + a + "; b==" + b + ";");
}
```

```
if (x > 50){
  // faire quelque chose
} else if (x > 5) {
  // faire autre chose
} else {
  // faire encore autre chose
}

/*
Greater than: >
Less than: <
Greater than or equal to: >=
Less than or equal to: <=
Equal: ==
Not equal: !=
*/
```

JAVASCRIPT : SYNTAXE

Structures conditionnelles

→ structure **switch ... case ...**

```
switch (variable) {  
  case 1:  
    // do something  
    break;  
  case 'a':  
    // do something else  
    break;  
  case 3.14:  
    // another code  
    break;  
  default:  
    // something completely different  
}
```

```
var toto = 1;  
var output = 'Résultat : '  
switch (toto) {  
  case 0:  
    output += 'Donc '  
  case 1:  
    output += 'quel '  
    output += 'est '  
  case 2:  
    output += 'votre '  
  case 3:  
    output += 'nom '  
  case 4:  
    output += '?'  
    console.log(output);  
    break;  
  case 5:  
    output += '!'  
    console.log(output);  
    break;  
  default:  
    console.log('Veuillez choisir un nombre entre 0 et 5 !');  
}
```

JAVASCRIPT : SYNTAXE

Structures itératives

→ boucles classiques

```
for(let counter=0 ; counter < 5 ; counter++){  
  console.log(counter);  
}  
var i=0;  
while(i<5){  
  console.log(++i);  
}  
do{  
  --i;  
} while(i>0)  
console.log(i);
```

JAVASCRIPT : SYNTAXE

Structures itératives

→ boucles **for ... in ...** et **for ... of ...**

```
let phones = {"type":"phone", "brand":"tomato", "name":"tomatoPhone"};
for(let key in phones){
  console.log(key);
  console.log(phones[key]);
}
let brandPhone = ["tomato", "pear"];
for(let i in brandPhone){
  console.log(i);
}
for(let elem of brandPhone){
  console.log(elem);
}
```

type
phone
brand
tomato
name
tomatoPhone
0
1
tomato
pear

JAVASCRIPT : SYNTAXE

Fonctions

Définition d'une fonction :

```
function nomFonction(arg1, arg2)
```

→ Contrairement au langage C, on ne donne pas le type des arguments ni celui de la valeur de retour éventuelle.

```
function average(a, b, c) {  
    return ( a + b + c ) /3;  
}
```

→ Possibilité d'écrire une fonction **anonyme** :

```
var result = function() { /* instructions */ }
```

JAVASCRIPT : SYNTAXE

Il n'est pas obligatoire :

- de définir une valeur de retour
- de spécifier tous les arguments lors de l'appel d'une fonction

Les fonctions ont accès à tous les paramètres d'entrée via le tableau **arguments** :

```
function sum() {  
    var sum = 0;  
    for(let i = 0 ; i<arguments.length ; i++)  
        sum += parseInt(arguments[i]);  
    return sum;  
}  
alert(sum(1, 2, 4));
```

JAVASCRIPT : SYNTAXE

→ Imbrication et fermeture de fonction

La fonction interne à accès aux variables et paramètres de la fonction parente (mais pas l'inverse).

Création d'une fermeture lorsque la fonction interne est disponible en dehors de la fonction parente.

72

Fermeture de fonction

Renvoie la fonction interne pour la rendre disponible en dehors de la portée de la fonction parente.

```
var animal = function(nom) {  
  var getNom = function () {  
    return nom;  
  }  
  return getNom;  
}  
  
monAnimal = animal("Licorne");  
console.dir(monAnimal);  
console.log(monAnimal());
```

```
▼ f getNom() ⓘ  
  arguments: null  
  caller: null  
  length: 0  
  name: "getNom"  
  ▶ prototype: {constructor: f}  
    [[FunctionLocation]]: monScript.js:256  
    ▶ [[Prototype]]: f ()  
    ▼ [[Scopes]]: Scopes[2]  
      ▶ 0: Closure (animal) {nom: 'Licorne'}  
      ▶ 1: Global {window: Window, self: Window,  
Licorne
```

JAVASCRIPT

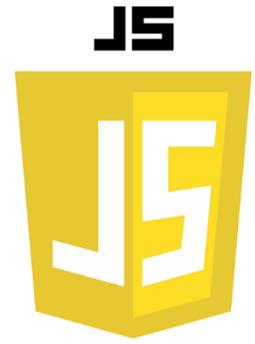
Définition

Syntaxe

Objets

Evènements

Accès à un éléments quelconque d'une page



JAVASCRIPT : OBJET

→ Notion d'objet est important en JavaScript : quasiment **tout est objet**.

Avant la normalisation ES6, il n'existait pas de notion de **class**.

Il est possible d'interagir à deux niveaux :

→ Au niveau du navigateur internet

→ Au niveau de la page affichée dans le navigateur

Tous les éléments HTML du DOM peuvent être manipulés en tant qu'objet.

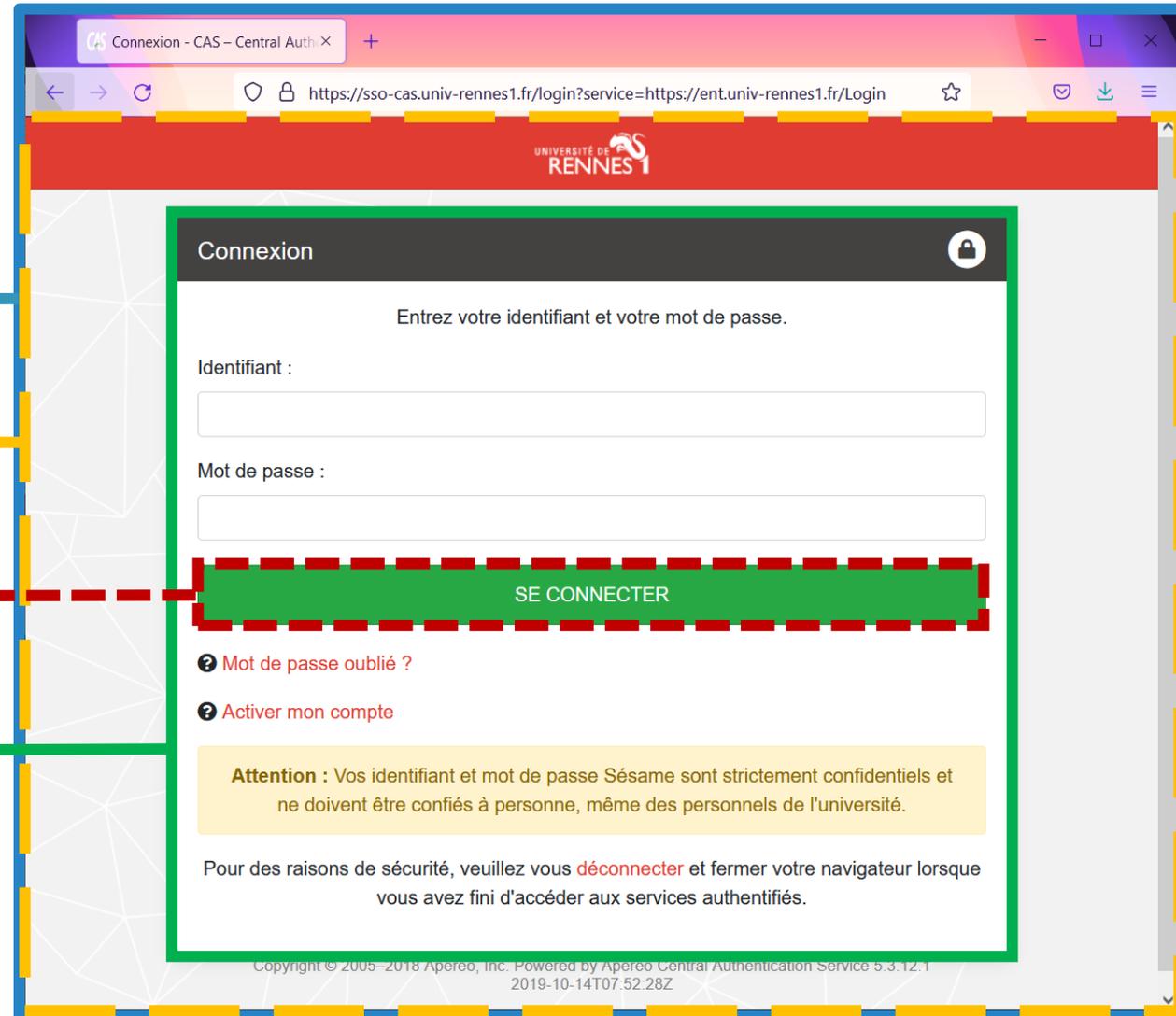
JAVASCRIPT : OBJET

Objet fenêtre

Objet document

Objet bouton

Objet formulaire



JAVASCRIPT : OBJET

→ L'accès se fait de façon hiérarchique.

```
window.document.forms["nomFormulaire"].nomElement
```

→ Par chaque objet il existe des méthodes et des attributs.

```
<form name="nomForm">
  <label for="login">Votre login :</label>
  <input type="text" name="nomLogin" id="login"/>
</form>
```

Votre login :

→ Par exemple, pour obtenir la valeur du champ login du formulaire :

```
<script>
  console.dir(window.document.forms["nomForm"].nomLogin);
  let userLogin = window.document.forms["nomForm"].nomLogin.value;
</script>
```

```
▼ input#login ⓘ
  accept: ""
  accessKey: ""
  align: ""
  alt: ""
  ariaAtomic: null
  ariaAutoComplete: null
  ariaBrailleLabel: null
  ariaBrailleRoleDescription: null
  ariaBusy: null
  ariaChecked: null
  ariaColCount: null
  ariaColIndex: null
```

JAVASCRIPT : OBJET

Possibilité de créer ses propres objets

→ Objets littéraux (JSON)

```
let myPhone = {"type": "phone", "brand": "Tomato"};
```

```
console.log(typeof myPhone);  
console.dir(myPhone);
```

```
object  
▼ Object ⓘ  
  brand: "Tomato"  
  type: "phone"  
  ► [[Prototype]]: Object
```

Une propriété d'un objet peut avoir n'importe quelle valeur :

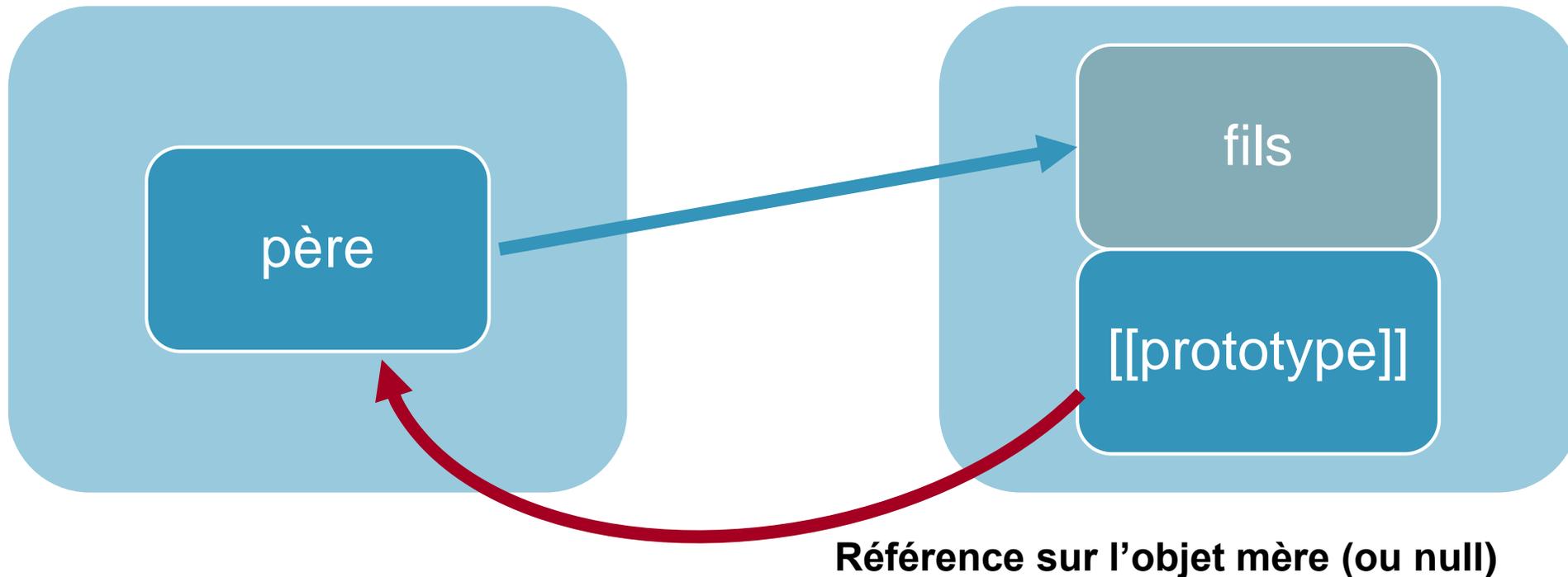
- Une valeur booléenne,
- Une valeur scalaire,
- Une liste,
- Un objet,
- Un code implémentant une fonction ou une classe

JAVASCRIPT : OBJET

→ Héritage par chaînage de **prototype**

Création d'un nouveau objet héritant des propriétés d'un autre objet via **[[prototype]]**

Propriété cachée



JAVASCRIPT : OBJET

→ Héritage par chaînage de **prototype**

```
let tomatoPhone = { brand: "Tomato" };  
console.dir(tomatoPhone);
```

```
let myPhone = {  
  name: "myPhone",  
  __proto__: tomatoPhone  
};  
console.dir(myPhone);
```

Attention avec l'utilisation de `__proto__`

Obsolète : `obj.__proto__`

Object.prototype

```
▼ Object i  
  brand: "Tomato"  
  ▼ [[Prototype]]: Object  
    ▶ constructor: f Object()  
    ▶ hasOwnProperty: f hasOwnProperty()  
    ▶ isPrototypeOf: f isPrototypeOf()  
    ▶ propertyIsEnumerable: f propertyIsEnumerable()  
    ▶ toLocaleString: f toLocaleString()  
    ▶ toString: f toString()  
    ▶ valueOf: f valueOf()  
    ▶ __defineGetter__: f __defineGetter__()  
    ▶ __defineSetter__: f __defineSetter__()  
    ▶ __lookupGetter__: f __lookupGetter__()  
    ▶ __lookupSetter__: f __lookupSetter__()  
    __proto__: (...)  
    ▶ get __proto__: f __proto__()  
    ▶ set __proto__: f __proto__()
```

```
▼ Object i  
  name: "myPhone"  
  ▼ [[Prototype]]: Object  
    brand: "Tomato"  
    ▶ [[Prototype]]: Object
```

`__proto__` ~~≠~~ `[[prototype]]`

getter/setter de `[[prototype]]`

JAVASCRIPT : OBJET

→ Création d'un objet par une **fonction constructrice**

```
function Phone(name, brand) {  
  this.name = name;  
  this.brand = brand;  
}  
  
let tomatoPhone = new Phone("tomatoPhone", "Tomato");  
console.dir(tomatoPhone);
```

```
Phone {  
  brand: "Tomato"  
  name: "tomatoPhone"  
  [[Prototype]]: Object  
    ▶ constructor: f Phone(name, brand)  
    ▶ [[Prototype]]: Object
```

→ Notion de **.prototype** : propriété que toutes les fonctions possèdent et qui est utilisée quand la fonction est utilisée comme fonction constructrice.

F.prototype contient une référence d'objet

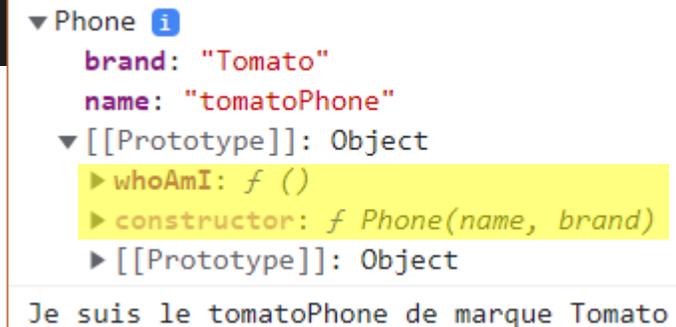
.prototype est défini comme le prototype du nouvel objet créé via la fonction constructrice

JAVASCRIPT : OBJET

- Création d'un objet par une **fonction constructrice**
- Possibilité d'ajouter/supprimer des propriétés au **prototype** sans modifier la propriété **constructor**.

```
Phone.prototype.whoAmI = function() {  
  console.log(`Je suis le ${this.name} de marque ${this.brand}`);  
}
```

```
tomatoPhone.whoAmI();
```



▼ Phone ⓘ

- brand: "Tomato"
- name: "tomatoPhone"
- ▼ [[Prototype]]: Object
 - ▶ whoAmI: f ()
 - ▶ constructor: f Phone(name, brand)
 - ▶ [[Prototype]]: Object

Je suis le tomatoPhone de marque Tomato

JAVASCRIPT : OBJET

→ L'héritage est implémenté par le **chaînage de prototypes**.

```
function Phone(name) {
  this.name = name;
}
Phone.prototype.whoAmI = function() {
  console.log("Je suis le " + this.name + " de marque " + this.brand);
}

function TomatoPhone(name) {
  this.brand = "Tomato";
}

var tomatoPhone = new TomatoPhone("tomatoPhone");
tomatoPhone.whoAmI();
```

▼ TomatoPhone ⓘ
brand: "Tomato"
▼ [[Prototype]]: Object
▶ constructor: f TomatoPhone(name)
▶ [[Prototype]]: Object

✖ ▶ Uncaught TypeError: tomatoPhone.whoAmI is not a function
at prototypage.js:33:13

▼ TomatoPhone ⓘ
brand: "Tomato"
▼ [[Prototype]]: Object
▶ whoAmI: f ()
▶ constructor: f Phone(name, brand)
▶ [[Prototype]]: Object

Je suis le undefined de marque Tomato

▼ TomatoPhone ⓘ
brand: "Tomato"
name: "tomatoPhone"
▼ [[Prototype]]: Object
▶ whoAmI: f ()
▶ constructor: f Phone(name, brand)
▶ [[Prototype]]: Object

Je suis le tomatoPhone de marque Tomato

JAVASCRIPT : OBJET

→ Création de **classe** (ES6)

```
class Phone {
  constructor(name, brand){
    this.name = name;
    this.brand = brand;
  }
  whoAmI() {
    console.log("Je suis le " + this.name + " de marque " + this.brand);
  }
}

let tomatoPhone = new Phone("tomatoPhone", "Tomato");
tomatoPhone.whoAmI();
```

```
console.log(typeof Phone);
console.dir(tomatoPhone);
```

Identique à l'héritage par prototype, mais syntaxe plus simple pour créer des objets et manipuler l'héritage

Je suis le tomatoPhone de marque Tomato

function

▼ Phone ⓘ

brand: "Tomato"

name: "tomatoPhone"

▼ [[Prototype]]: Object

▶ constructor: class Phone

▶ whoAmI: f whoAmI()

▶ [[Prototype]]: Object

JAVASCRIPT : OBJET

→ Création de **classe** (ES6) :

Champ de classe : syntaxe permettant d'ajouter des propriétés uniquement à l'objet

```
class Phone {  
  version = "test";  
  constructor(name, brand){  
    this.name = name;  
    this.brand = brand;  
  }  
}
```

```
let tomatoPhone = new Phone("tomatoPhone", "Tomato");  
console.log(tomatoPhone.version);  
console.log(Phone.prototype.version);
```

test
undefined

JAVASCRIPT : OBJET

→ **Héritage** entre classe (ES6)

Utilisation du mot clé **extends**

Appel du constructeur de la classe mère avec **super()**

```
Je suis le tomatoPhone de marque Tomato  
Je suis le super pearPhone de marque Pear
```

```
class Phone {  
  constructor(name, brand){  
    this.name = name;  
    this.brand = brand;  
  }  
  whoAmI() {  
    console.log("Je suis le " + this.name + " de marque " + this.brand);  
  }  
}  
class SuperPhone extends Phone{  
  constructor(name, brand){  
    super(name, brand);  
  }  
  whoAmI() {  
    console.log(`Je suis le super ${this.name} de marque ${this.brand}`);  
  }  
}  
let tomatoPhone = new Phone("tomatoPhone", "Tomato");  
tomatoPhone.whoAmI();  
let smartPhone = new SuperPhone("pearPhone", "Pear");  
smartPhone.whoAmI();
```

JAVASCRIPT : CLASS

→ Attention à la compatibilité des navigateurs avec les fonctionnalités récentes proposé par ECMASript.

	📱					📱					☰		
	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android	Deno	Node.js
<code>classes</code>	✓ 49 ...	✓ 13	✓ 45	✓ 36 ...	✓ 9	✓ 49 ...	✓ 45	✓ 36 ...	✓ 9	✓ 5.0 ...	✓ 49 ...	✓ 1.0	✓ 6.0.0 ...
<code>constructor</code>	✓ 49 ...	✓ 13	✓ 45	✓ 36 ...	✓ 9	✓ 49 ...	✓ 45	✓ 36 ...	✓ 9	✓ 5.0 ...	✓ 49 ...	✓ 1.0	✓ 6.0.0 ...
<code>extends</code>	✓ 49 ...	✓ 13	✓ 45	✓ 36 ...	✓ 9	✓ 49 ...	✓ 45	✓ 36 ...	✓ 9	✓ 5.0 ...	✓ 49 ...	✓ 1.0	✓ 6.0.0 ...
<code>Private class fields</code>	✓ 74	✓ 79	✓ 90	✓ 62	✓ 14.1	✓ 74	✓ 90	✓ 53	✓ 14.5	✓ 11.0	✓ 74	✓ 1.0	✓ 12.0.0
<code>Private class fields 'in'</code>	✓ 91	✓ 91	✓ 90	✓ 77	✓ 15	✓ 91	✓ 90	✓ 64	✓ 15	✓ 16.0	✓ 91	✓ 1.9	✓ 16.4.0
<code>Private class methods</code>	✓ 84	✓ 84	✓ 90	✓ 70	✓ 15	✓ 84	✓ 90	✓ 60	✓ 15	✓ 14.0	✓ 84	✓ 1.0	✓ 14.6.0
<code>Public class fields</code>	✓ 72	✓ 79	✓ 69	✓ 60 ...	✓ 14.1	✓ 72	✓ 79	✓ 51	✓ 14.5 ...	✓ 11.0	✓ 72	✓ 1.0	✓ 12.0.0
<code>static</code>	✓ 49 ...	✓ 13	✓ 45	✓ 36 ...	✓ 9	✓ 49 ...	✓ 45	✓ 36 ...	✓ 9	✓ 5.0 ...	✓ 49 ...	✓ 1.0	✓ 6.0.0 ...
<code>Static class fields</code>	✓ 72	✓ 79	✓ 75	✓ 60	✓ 14.1	✓ 72	✓ 79	✓ 51	✓ 14.5	✓ 11.0	✓ 72	✓ 1.0	✓ 12.0.0
<code>Class static initialization blocks</code>	✓ 94	✓ 94	✓ 93	✓ 80	✗ No	✓ 94	✓ 93	✓ 66	✗ No	✓ 17.0	✓ 94	✓ 1.14	✓ 16.11.0

JAVASCRIPT : OBJET

Mot clé `this`

Il se comporte légèrement différemment des autres langages de programmation.

→ Dans le contexte global :

`this` fait référence à l'objet global.

Dans le cas d'un navigateur

`this = window`

```
console.log(this === window);

this.a = 37;
console.log(window.a);

this.b = "JXC";
console.log(window.b);
console.log(b);

var c = 1;
console.log(this.c);
console.log(window.c);

let d = true;
console.log(window.d);
```

JAVASCRIPT : OBJET

Mot clé `this`

→ Dans le contexte d'une fonction :

la valeur de `this` dépend de la façon dont la fonction est appelée.

Quand une fonction est appelée comme **méthode d'un objet**, `this` correspond à l'objet possédant la méthode qu'on appelle.

```
var o = {
  prop: 37,
  f: function() {
    return this.prop;
  }
};

console.log(o.f()); // 37
```

Quand une fonction est utilisée comme un **constructeur** (avec le mot clef `new`), `this` sera lié au nouvel objet.

```
function Phone(){
  this.brand = "tomato";
}
var myPhone = new Phone();
console.log(myPhone.brand); // tomato
```

JAVASCRIPT : OBJET

Mot clé `this`

→ Dans le contexte d'une fonction :

La même fonction est assignée à deux objets différents et a un `this` différent dans les appels.

La valeur de `this` est évaluée pendant l'exécution en fonction du contexte.

```
let user = { name: "John" };
let admin = { name: "Admin" };

function sayHi() {
  console.log( this.name );
}

// utiliser la même fonction dans deux objets
user.f = sayHi;
admin.f = sayHi;

// ces appels ont un this différent
// "this" à l'intérieur de la fonction
//se trouve l'objet "avant le point"
user.f(); // John (this == user)
admin.f(); // Admin (this == admin)
```

JAVASCRIPT : OBJET

Mot clé `this`

→ Dans le contexte d'une fonction :

Une fonction peut être appelée sans être liée à un objet :

```
function f1(){  
  return this;  
}  
console.log(f1() === window); // true (objet global)
```

```
"use strict";  
  
function f1(){  
  return this;  
}  
  
console.log(f1()); //undefined
```



Mode use strict non activé

Cet appel peut être vu comme une erreur de programmation : si il y a un `this` dans une fonction, il s'attend à être appelé dans **un contexte d'objet**.

JAVASCRIPT : OBJET

Mot clé `this`

En utilisant les **fonctions fléchées**, `this` correspond à la valeur de `this` utilisée dans le contexte englobant (ces fonctions ne possèdent pas de `this`).

```
let objet = {
  i: [10, 20, 30],
  j: 100,
  b: () => console.log(this.j, this),
  c: function() {
    console.log(this.j, this);
  },
  showList() {
    this.i.forEach(
      elem => console.log(this.j + ': ' + elem)
    );
  }
}
```

```
objet.b();
objet.c();
objet.showList()
```

JAVASCRIPT : OBJET

Mot clé `this`

Pour passer `this` d'un **contexte** à un **autre**, les fonctions suivantes peuvent être utilisées : `call()`, `apply()` et `bind()`.

```
"use strict";

function fonction(name) { this.name = name; }
fonction.prototype.methode = function(callback) {
  console.log('fonction ', this);
  callback();
}

let objet = new fonction("objet1");
objet.methode(function() {
  console.log('objet ', this);
})
```

```
fonction ▶ fonction {name: 'objet1'}
objet undefined
```

JAVASCRIPT : OBJET

Mot clé **this**

Méthode **call()** :

Réalise un appel à une fonction avec une valeur **this** donnée et des possibles arguments.

93

```
fonction.prototype.methodeCall = function(callback){
  console.log('fonction :', this);
  callback.call(this, "un param");
}
let objet2 = new fonction("objet2");
objet2.methodeCall(function(param) {
  console.log('call ', param, ', ', ', this);
})
```

```
fonction : ▶ fonction {name: 'objet2'}
call un param , ▶ fonction {name: 'objet2'}
```

JAVASCRIPT : OBJET

Mot clé **this**

Méthode **apply()** :

Appelle une fonction en lui passant une valeur **this** et des arguments sous forme d'un tableau ou d'une liste.

94

```
function.prototype.methodeApply = function(callback){
  console.log('fonction :', this);
  callback.call(this, ["un param", "deux param"]);
}
let objet3 = new fonction("objet3");
objet3.methodeApply(function(param) {
  console.log('apply ', param, ', ', ', this);
})
```

```
fonction : ► fonction {name: 'objet3'}
apply ► (2) ['un param', 'deux param'] ,
  ► fonction {name: 'objet3'}
```

JAVASCRIPT : OBJET

Mot clé **this**

Méthode **bind()** :

Création d'une nouvelle fonction qui possède le même corps et la même portée mais où le **this** sera lié au premier argument passé à **bind** (de façon permanente).

```
function.prototype.methodeBind = function(callback){
  console.log('fonction :', this);
  let newfonction = callback.bind(this, "un param");
  newfonction();
}
let objet4 = new function("objet4");
objet4.methodeBind(function(param) {
  console.log('bind ', param, ', ', this);
})
```

95

```
fonction : ▶ fonction {name: 'objet4'}
bind un param , ▶ fonction {name: 'objet4'}
```

JAVASCRIPT : OBJET

Mot clé **this**

Méthode **bind()** :

```
let module = {
  x: 11,
  getX(){
    return this.x;
  }
};

module.getX(); //11

let getx = module.getX;
getx(); //error : this undefined

let bindGetX = getx.bind(module);
bindGetX(); //11 : création d'une nouvelle fonction
//liée à module en tant que this
```

JAVASCRIPT

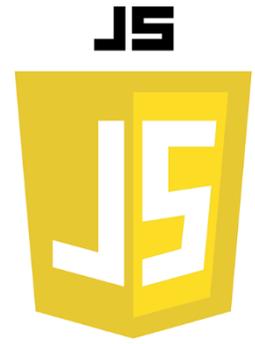
Définition

Syntaxe

Objets

Evènements

Accès à un éléments quelconque d'une page



JAVASCRIPT : EVENEMENT

L'action sur un élément de la page HTML se fait lors d'un évènement particulier :

- clic sur un bouton,
- champ input d'un formulaire qui change,
- Redimensionnement de la fenêtre,
- Formulaire en cours de soumission,
- fin de chargement de la page HTML, etc.

Les **gestionnaires d'évènements** peuvent être utilisés pour gérer et vérifier les entrées utilisateur, les actions utilisateur et les actions du navigateur.

JAVASCRIPT : EVENEMENT

Exemple d'évènement possible pour une page Web :

→ click (onClick)

→ load (onLoad)

→ unload (onUnload)

→ mouseOver (onMouseOver)

→ mouseOut (onMouseOut)

→ focus (onFocus)

→ change (onChange)

→ submit (onSubmit)

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

JAVASCRIPT : EVENEMENT

→ Exemple d'utilisation de **onChange** avec un champ **input** d'un formulaire :

Fichier HTML

```
<body>
  <form name="monForm">
    <label for="login">Votre login :</label>
    <input type="text" name="login" id="login" onchange="afficherLogin();" />
    <input type="text" name="loginBis" id="loginBis" />
  </form>
</body>
```

Fichier .js

```
function afficherLogin(){
  window.document.forms["monForm"].loginBis.value =
  window.document.forms["monForm"].login.value;
}
```

Résultat

Votre login :

JAVASCRIPT : EVENEMENT

→ Exemple d'utilisation de **onChange** avec un champ **input** d'un formulaire :

Fichier HTML

```
<body>
  <form name="monForm">
    <label for="login">Votre login :</label>
    <input type="text" name="login" id="login" />
    <input type="text" name="loginBis" id="loginBis" />
  </form>
</body>
```

Fichier .js

```
var loginEvent = document.forms["monForm"].login.onChange = function() {
  window.document.forms["monForm"].loginBis.value =
    window.document.forms["monForm"].login.value;
}
```

Résultat

Votre login :

Méthode à privilégier

JAVASCRIPT : EVENEMENT

→ Utilisation de la méthode `addEventListener()` :

Enregistre un écouteur d'évènement sur un élément DOM.

Permet également d'enregistrer plusieurs gestionnaires pour le même écouteur.

Possibilité de supprimer un écouteur ajouté précédemment (*`removeEventListener()`*).

102

```
<body>
  <form name="monForm">
    <label for="login">Votre login :</label>
    <input type="text" name="login" id="login" />
    <input type="text" name="loginBis" id="loginBis" />
  </form>
</body>
```

Fichier HTML

```
var inputLogin = document.forms["monForm"].login;
inputLogin.addEventListener('change', function(){
  window.document.forms["monForm"].loginBis.value =
    window.document.forms["monForm"].login.value;
});
```

Fichier .js

JAVASCRIPT

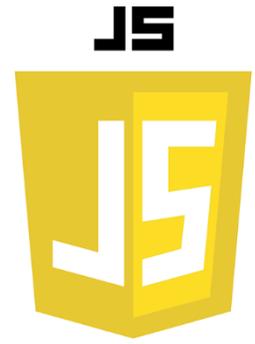
Définition

Syntaxe

Objets

Evènements

Accès à un éléments quelconque d'une page



JAVASCRIPT : ACCES ELEMENT

La manière la plus simple pour avoir **accès à un élément d'une page** est d'utiliser son **identifiant (Id)**.

→ Méthode **getElementById('id de l'élément')**

```
var inputLogin = document.forms["monForm"].login;  
var inputLogin = document.getElementById("login");
```

Il est possible ensuite d'avoir accès à des informations et d'agir sur l'élément.

→ **innerHTML** : récupération/modification du contenu HTML de l'élément.

→ **textContent** : récupération/modification du contenu de l'élément.

→ **nodeName** : récupération du nom de l'élément.

JAVASCRIPT : EXEMPLE

Fichier HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello world</title>
    <link rel="stylesheet" href="css/monCSS.css" />
    <script src="js/mesFonctions.js" defer></script>
  </head>
  <body>
    <div id="maDiv"></div>
    <button type="button" id="btnvert">Vert</button>
    <button type="button" id="btnrouge">Rouge</button>
  </body>
</html>
```

JAVASCRIPT : EXEMPLE

Fichier CSS

```
#maDiv{  
  background-color: #635ead;  
  width:200px;  
  height:200px;  
}
```

JAVASCRIPT : EXEMPLE

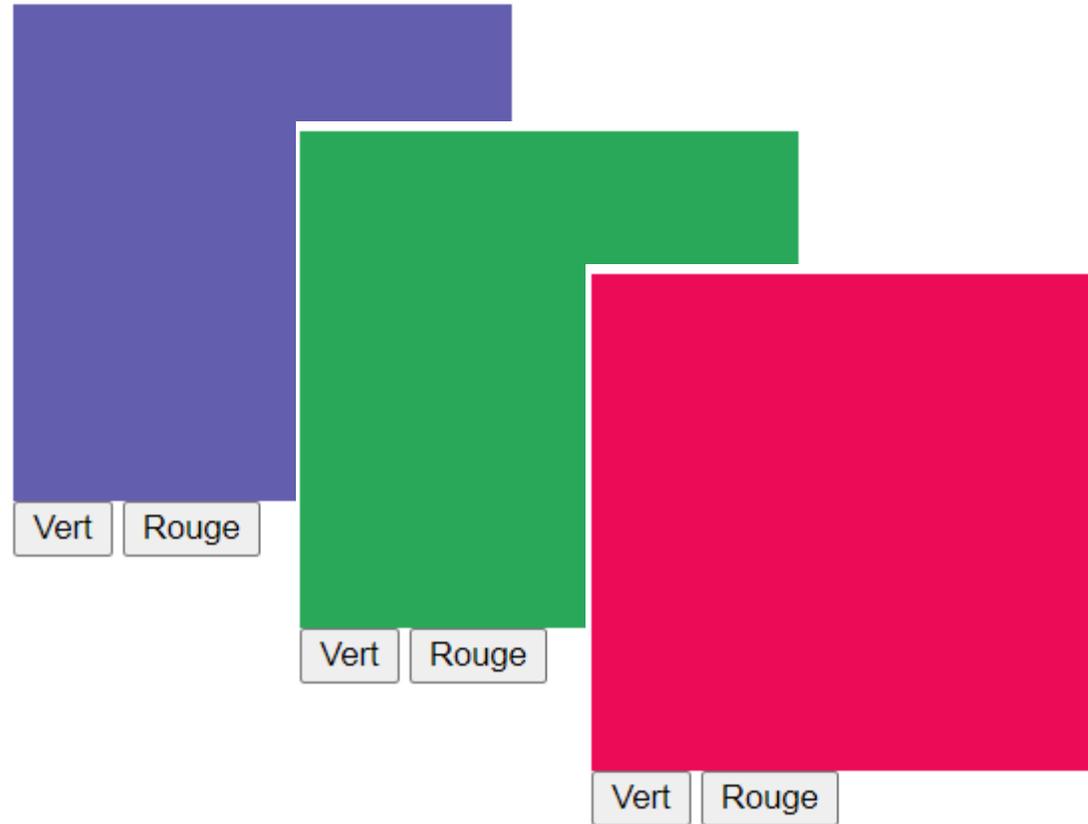
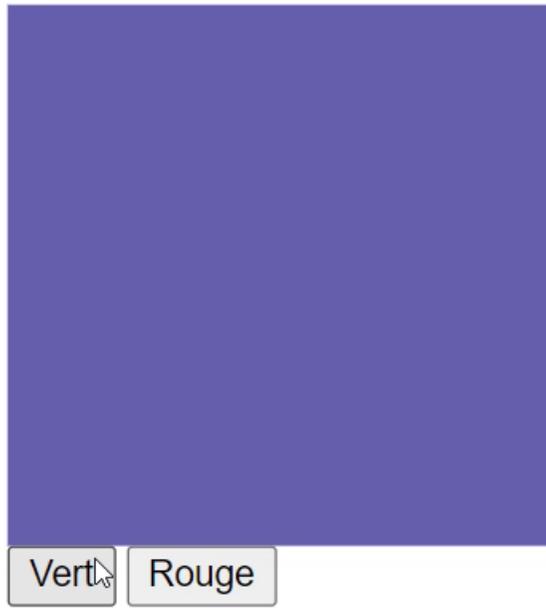
Fichier JS

```
function vert(){
    document.getElementById('maDiv').style.backgroundColor='#29a85a';
}
function rouge(){
    document.getElementById('maDiv').style.backgroundColor='#ec0b56';
}

document.getElementById('btnvert').addEventListener('click', vert);
document.getElementById('btnrouge').addEventListener('click', rouge);
```

JAVASCRIPT : EXEMPLE

Navigateur



JAVASCRIPT : ECMASCRIPT

→ **Lien vers les nouvelles spécifications ECMAScript :**

<https://www.ecma-international.org/technical-committees/tc39/?tab=published-standards>

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources

→ **Table de compatibilité ECMAScript :**

<http://kangax.github.io/compat-table/es6/>

<http://kangax.github.io/compat-table/es2016plus/>

FRAMEWORK

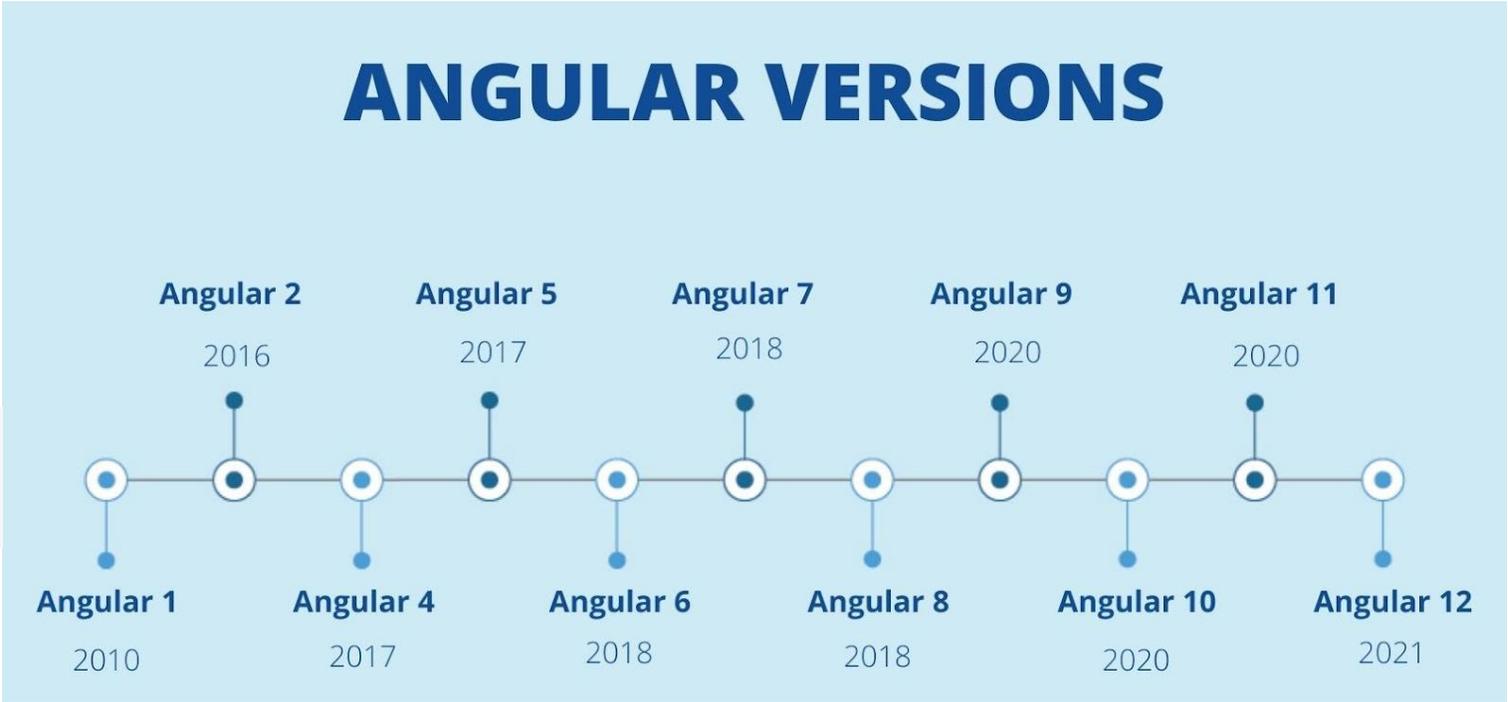
Utilisation de Framework

Notion de composant web

Concept d'Angular

FRAMEWORK ANGULAR

ANGULAR VERSIONS



FRAMEWORK ANGULAR

Impérative vs déclarative

JQuery

→ Lien sur les évènements pour déclencher des actions

```
<input type="text" id="yourName">
<h1 id="helloName"></h1>
<script type="text/javascript">
  $(function() {
    $("#yourName").keyup(function () {
      | $("#helloName").text("Hello " + this.value + "!");
    });
  });
</script>
```

Angular

→ Déclarer des liens entre les éléments de la vue et du modèle (maj automatique)

```
<input type="text" [(ngModel)]="yourName">
<h1>Hello {{yourName}}!</h1>
```

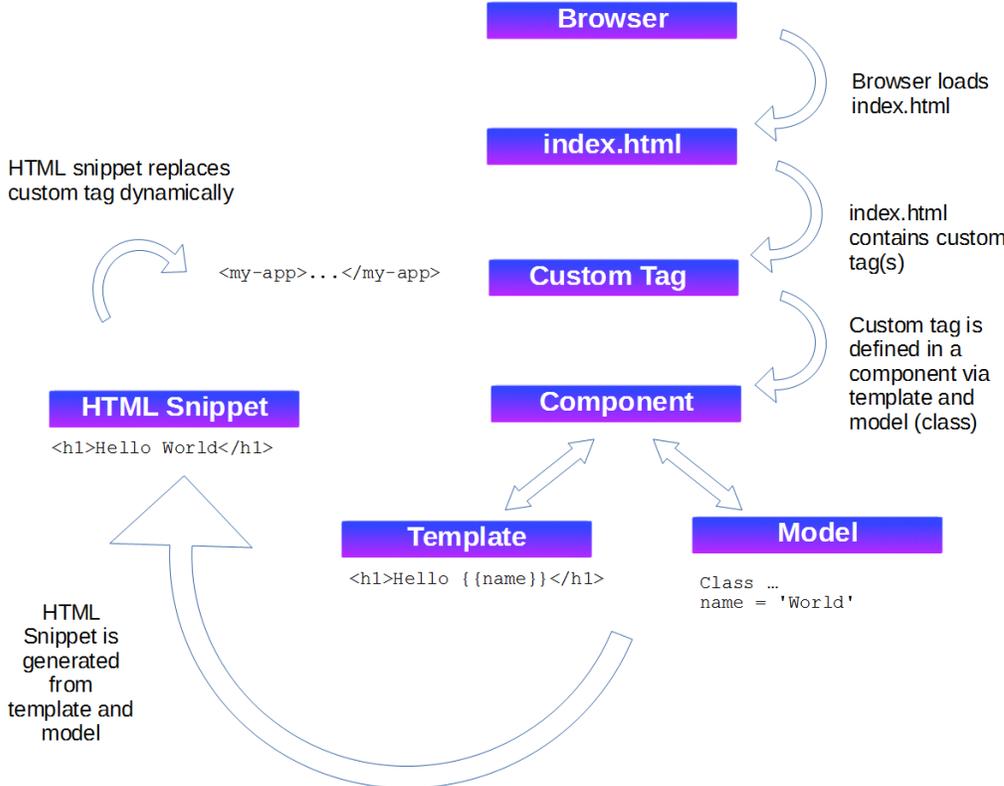
FRAMEWORK ANGULAR

Abstraction :

- jQuery a déjà une abstraction par rapport au fonctionnement du navigateur (traversé du DOM, event binding, etc.)
- Angular rend les relations abstraites (entre le modèle et la vue, différents éléments de la vue, etc.)

FRAMEWORK ANGULAR

→ Angular 'compile' HTML : utilisation de custom tag



FRAMEWORK : COMPOSANT WEB

- Faciliter la réutilisation d'éléments dans la monde du web (création d'élément réutilisables, encapsulés et versatiles sans risquer une collision avec d'autres morceaux de code).

Standard du web :

- **Custom Elements** (permet de créer et enregistrer des nouveaux éléments HTML)
- **HTML Templates** (squelette pour créer des éléments HTML instanciables)
- **Shadow DOM** (permet d'encapsuler le JavaScript et le CSS des éléments)
- **HTML Imports** (abandonnées au profit des imports JavaScript)

FRAMEWORK : COMPOSANT WEB

→ Enrichi le web avec des nouveaux tag.

Principe :

Créer des nouveaux tags

Encapsuler le code pour masquer et isoler sa complexité

Etre capable d'importer et de déclarer les tags dans d'autres pages ou projets.

FRAMEWORK : COMPOSANT WEB

- Standardisation des composants web en 2012 par W3C.
- Google via son projet **Polymer** à aider au développement des web composants.

Browser support	 CHROME	 OPERA	 SAFARI	 FIREFOX	 EDGE
 HTML TEMPLATES	 STABLE	 STABLE	 STABLE	 STABLE	 STABLE
 CUSTOM ELEMENTS	 STABLE	 STABLE	 STABLE	 STABLE	 STABLE
 SHADOW DOM	 STABLE	 STABLE	 STABLE	 STABLE	 STABLE
 ES MODULES	 STABLE	 STABLE	 STABLE	 STABLE	 STABLE

COMPOSANT WEB : CUSTOM ELEMENT

→ 4 grandes étapes :

Etape 1 : Créer une classe (syntaxe de classe ES6) dans laquelle est spécifiée la **fonctionnalité du composant web**.

Etape 2 : Enregistrer le nouvel élément personnalisé avec la commande ***define***.

Etape 3 : Connecter un shadow DOM à l'élément personnalisé, puis ajouter des éléments fils, des écouteurs d'évènements, etc.

Etape 4 : Possibilité de définir un **template HTML**. Utilisation des méthodes DOM pour cloner le template et le connecter au shadow DOM.

COMPOSANT WEB : CUSTOM ELEMENT

Etape 1 : Créer une classe (syntaxe de classe ES6) dans laquelle est spécifiée la **fonctionnalité du composant web**.

- Hérite de *HTMLElement*
- Possibilité de définir des rappels se déclenchant à différents points du cycle de vie de l'élément.
- Extension de l'API JavaScript *CustomElements* pour créer des nouveaux éléments.

```
class HelloWorld extends HTMLElement {
  constructor(){
    super();
    console.log("constructor");
    //Fonctionnalité de l'élément
  }
  //Rappels du cycle de vie
  connectedCallback(){
    this.innerHTML = '<p>Composant connecté pour la 1ère fois au DOM : '+
      'Ajout du contenu initial / fetch data</p>'
  }
  disconnectedCallback(){
    //l'élément personnalisé est déconnecté du DOM du document
  }
  adoptedCallback(){
    //l'élément personnalisé est déplacé vers un nouveau document
  }
  attributeChangedCallback(){
    //un des attributs de l'élément personnalisé est ajouté, supprimé ou modifié
  }
}
```

COMPOSANT WEB : CUSTOM ELEMENT

Etape 2 : Enregistrer le nouvel élément personnalisé avec la commande *define*.

→ Le nom de l'élément doit contenir un « - »

```
//customElements.define(name, constructor, options);  
customElements.define('hello-world', HelloWorld);
```

→ Possibilité d'ajouté des options

```
customElements.define('word-count', WordCount, { extends: 'p' });
```

```
class WordCount extends HTMLParagraphElement { /*...*/ }
```

COMPOSANT WEB : CUSTOM ELEMENT

Deux types d'élément customisé :

→ Éléments customisés autonomes (ne dépend pas d'un autre élément HTML)

```
<hello-world></hello-world>
```

```
document.createElement('hello-world');
```

→ Éléments intégrés personnalisés (héritant d'un élément HTML de base)

```
<p is="word-count"></p>
```

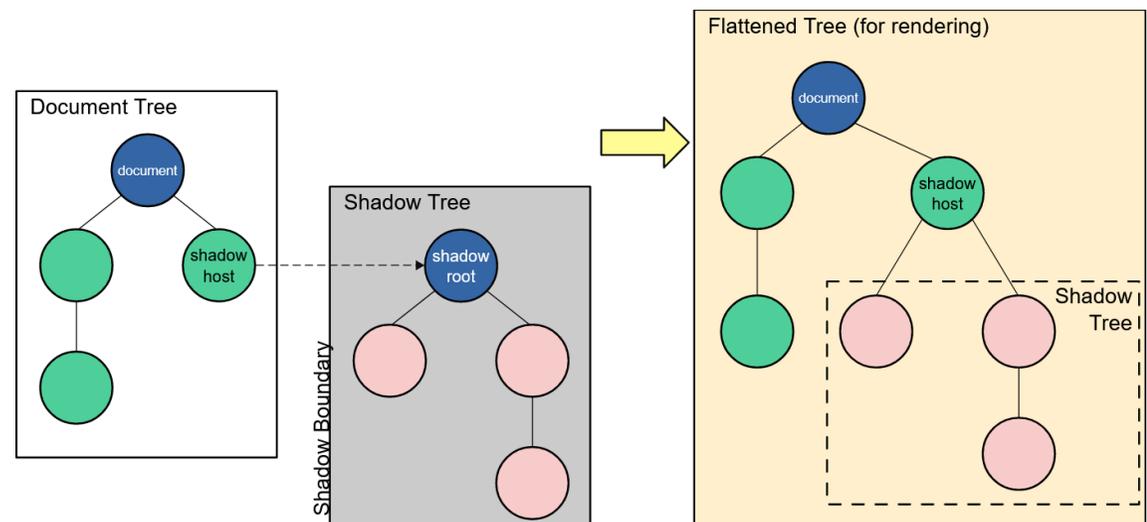
```
document.createElement("p", {is: "word-count"})
```

COMPOSANT WEB : SHADOW DOM

Étape 3 : Connecter un **shadow DOM** à l'élément personnalisé, puis ajouter des éléments fils, des écouteurs d'évènements, etc.

→ **Encapsulation** possible grâce au **Shadow DOM** : possible d'utiliser JavaScript et du CSS dans un élément customisé sans affecter les autres composants.

Lors de la création du **shadow DOM**, un sous arbre est rattaché à un élément du DOM.



COMPOSANT WEB : SHADOW DOM

Etape 3 : Connecter un **shadow DOM** à l'élément personnalisé, puis ajouter des éléments fils, des écouteurs d'évènements, etc.

→ Associer une racine fantôme à un élément :

```
let fantome = element.attachShadow({mode: 'open'});  
let fantome = element.attachShadow({mode: 'closed'});
```

→ Ajout d'élément au Shadow DOM :

```
let paragraphe = document.createElement('p');  
fantome.appendChild(paragraphe);
```

COMPOSANT WEB : EXEMPLE 1

→ Création d'un Shadow DOM

Web Component

Nom de la balise : Hello !

Hello word du shadow DOM !

```
class HelloWorld extends HTMLElement {
  constructor(){
    super();
    console.log("constructor");
    //Fonctionnalité de l'élément
    this.nom = "Hello !"
    //Shadow DOM
    let shadowRoot = this.attachShadow({mode: 'open'});
    let newParagraph = document.createElement('p');
    newParagraph.setAttribute('class', 'paragraphe');
    newParagraph.innerHTML = 'Hello word du shadow DOM !';
    let otherParagraph = document.createElement('p');
    otherParagraph.innerHTML = this.afficherNom();
  }
}
```

```
let newStyle = document.createElement('style');
newStyle.textContent = `
  .paragraphe {
    border: 1px solid black;
    border-radius: 5px;
    background-color: yellow;
  }`;
shadowRoot.appendChild(otherParagraph);
shadowRoot.appendChild(newStyle);
shadowRoot.appendChild(newParagraph);
}
afficherNom(){
  return 'Nom de la balise : ' + this.nom;
}
}
```

COMPOSANT WEB : SHADOW DOM

→ Possibilité d'ajouter une feuille de style externe au lieu de la balise style :

```
const linkElem = document.createElement('link');  
linkElem.setAttribute('rel', 'stylesheet');  
linkElem.setAttribute('href', 'style.css');  
  
shadow.appendChild(linkElem);
```

COMPOSANT WEB : HTML TEMPLATE

- Portion de code réutilisable
- Le moteur ne vérifie que la validé du contenu, le contenu n'est pas affiché
- Les scripts et les images ne sont pas chargés et le contenu n'est pas attaché au DOM (*document.getElementById()* et *querySelector()* ne fonctionneront pas)

Rappel d'utilisation :

```
<template id="hello">  
  <p>Autre Hello World !</p>  
</template>
```

```
let template = document.getElementById('hello');  
let templateContent = template.content.cloneNode(true);  
document.body.appendChild(templateContent);
```

Code JavaScript pour ajouter le template au DOM et l'afficher

COMPOSANT WEB : HTML TEMPLATE

Etape 4 : Possibilité de définir un **template HTML**. Utilisation des méthodes DOM pour cloner le template et le connecter au shadow DOM.

→ Constructeur de la classe *HelloWorld* :

```
constructor() {  
  super();  
  let template = document.getElementById('hello');  
  let templateContent = template.content;  
  
  const shadowRoot = this.attachShadow({mode: 'open'});  
  shadowRoot.appendChild(templateContent.cloneNode(true));  
}
```

COMPOSANT WEB : EXEMPLE

→ Ajout d'un template

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello world</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
  </head>
  <body>
    <p>Web Component</p>
    <hello-world></hello-world>
    <template id="hello">
      <p>Autre Hello World avec template !</p>
    </template>
    <script src='js/composantWeb.js'></script>
  </body>
</html>
```

```
//Shadow DOM
let shadowRoot = this.attachShadow({mode: 'open'});

let template = document.getElementById('hello');
let templateContent = template.content;

shadowRoot.appendChild(templateContent.cloneNode(true));
```

135

Web Component

Autre Hello World avec template !

Nom de la balise : Hello !

Hello word du shadow DOM !

COMPOSANT WEB : EXEMPLE 2

→ Page HTML :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello world</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
  </head>
  <body>
    <template id="counter">
      <style>
        button {
          background-color: ■ red;
          color: ■ white;
          padding: 4px;
        }
      </style>
      <button>Click me</button>
      <span id="times">0</span>
    </template>
    <my-counter></my-counter>
    <button>Pas dans le même composant</button>
    <script src='js/composantWeb2.js'></script>
  </body>
</html>
```

COMPOSANT WEB : EXEMPLE 2

→ Page JavaScript :

```
class MyCounter extends HTMLElement {
  times = 0;
  constructor() {
    super();
    const template = document.getElementById('counter');
    const shadowRoot = this.attachShadow({mode: 'open'});
    shadowRoot.appendChild(template.content.cloneNode(true));
    this.onClick = this.onClick.bind(this);
    this.shadowRoot.querySelector('button')
      .addEventListener('click', this.onClick);
  }
  onClick() {
    this.times += 1;
    this.shadowRoot.querySelector('#times').textContent = this.times;
  }
}
customElements.define("my-counter", MyCounter);
```

COMPOSANT WEB : EXEMPLE 2

→ Résultat :

