



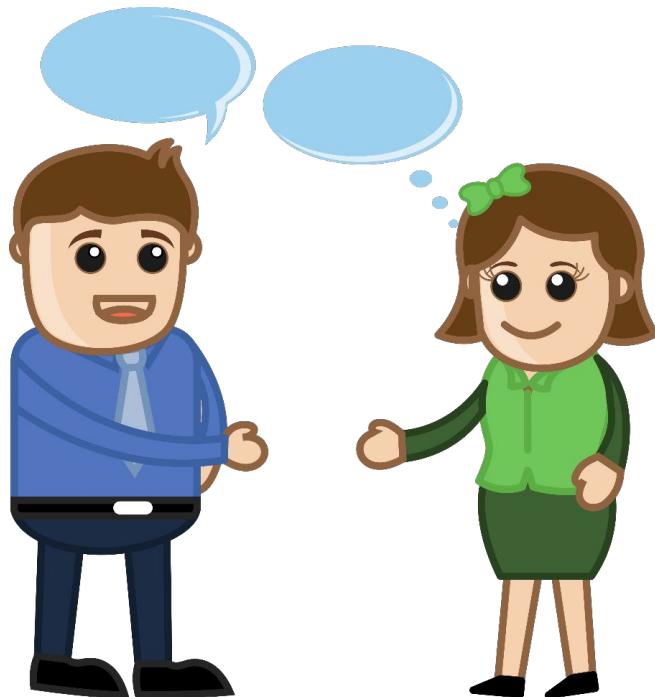
Nicolas Amini-Lamy

**WEBFORCE**  
BE THE CHANGE 

## Présentons-nous

---

- Nom
- Prénom
- Âge
- Objectif personnel
- Passions, détail insolite.. :)



Nicolas AMINI-LAMY

*Ingénieur en architecture des systèmes d'information  
5 années d'expérience en SSII  
Formateur - Entrepreneur depuis 2018*

Pour toute question post-formation :

[pro@naminilamy.fr](mailto:pro@naminilamy.fr)



3

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

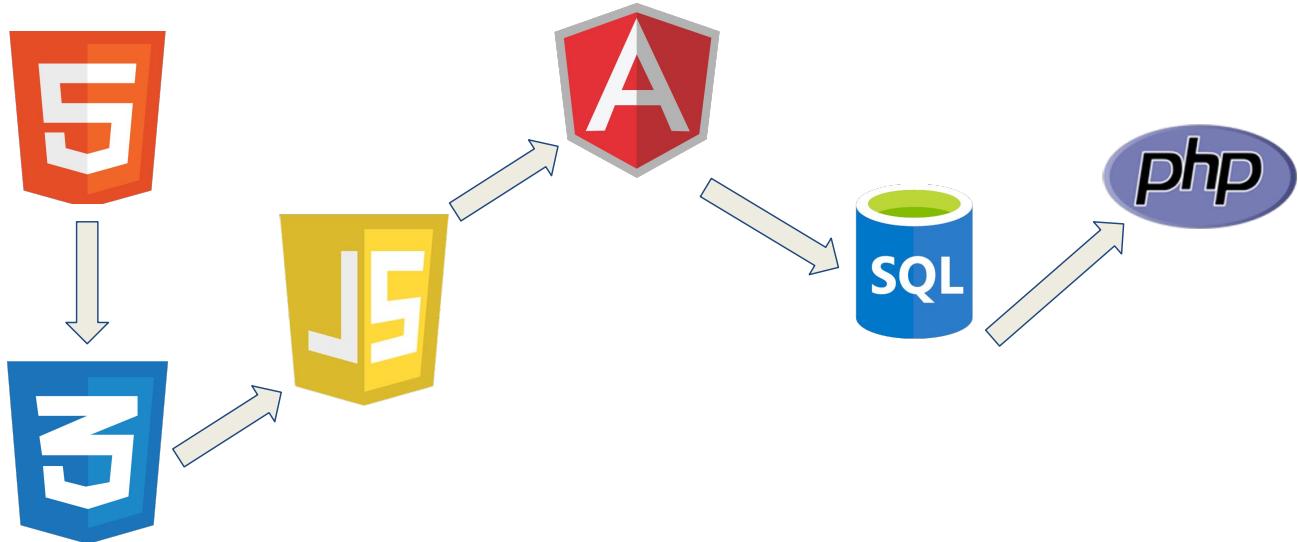
## Programme de la semaine

---

- Point parcours & architectures web
- Compléments JavaScript
- Présentation du langage TypeScript
- Programmation asynchrone
- Présentation Angular & TP Projet

4

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy



5

## Point parcours

---

Votre parcours est constitué de 2 blocs : front & back.

Vous adressez en premier lieu 3 langages (HTML / CSS / JavaScript) qui vous permettent de développer ce que l'on qualifie de ressources statiques.

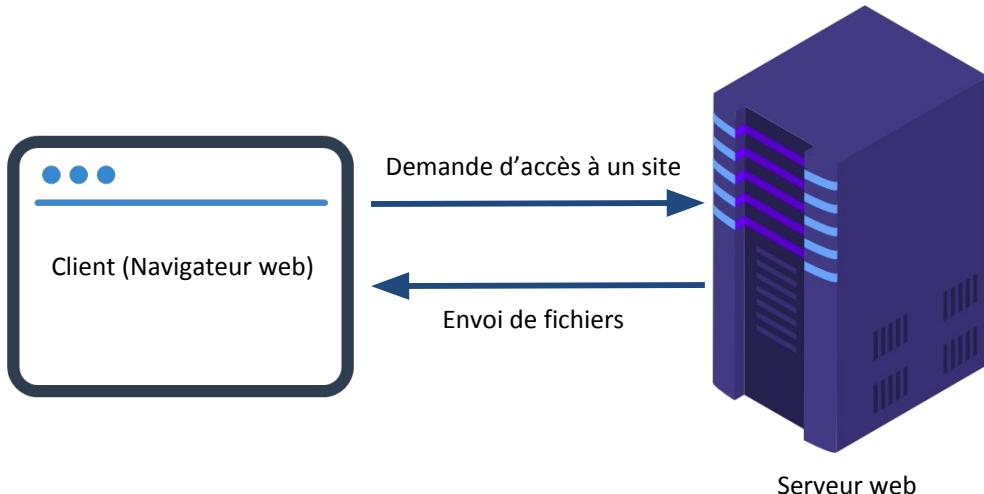
Une ressource statique est une ressource dont le contenu ne changera pas en fonction de la personne qui les récupère. Ce sont des fichiers destinés à être hébergés sur un serveur web.

Un serveur Web est un serveur sur lequel on installe une application dont le rôle est de mettre à disposition des ressources statiques.

Typiquement, un site web vitrine est constitué uniquement de ressources statiques.

Les 3 grandes applications du marché utilisées sont : Microsoft IIS , Apache HTTP Server, Nginx

6



Architecture client / serveur classique.

Un navigateur web interroge un serveur web afin de récupérer des ressources statiques.

7

## Point parcours

---

Dans le cas de figure du “site web”, ou des “ressources statiques”, il n'y a pas :

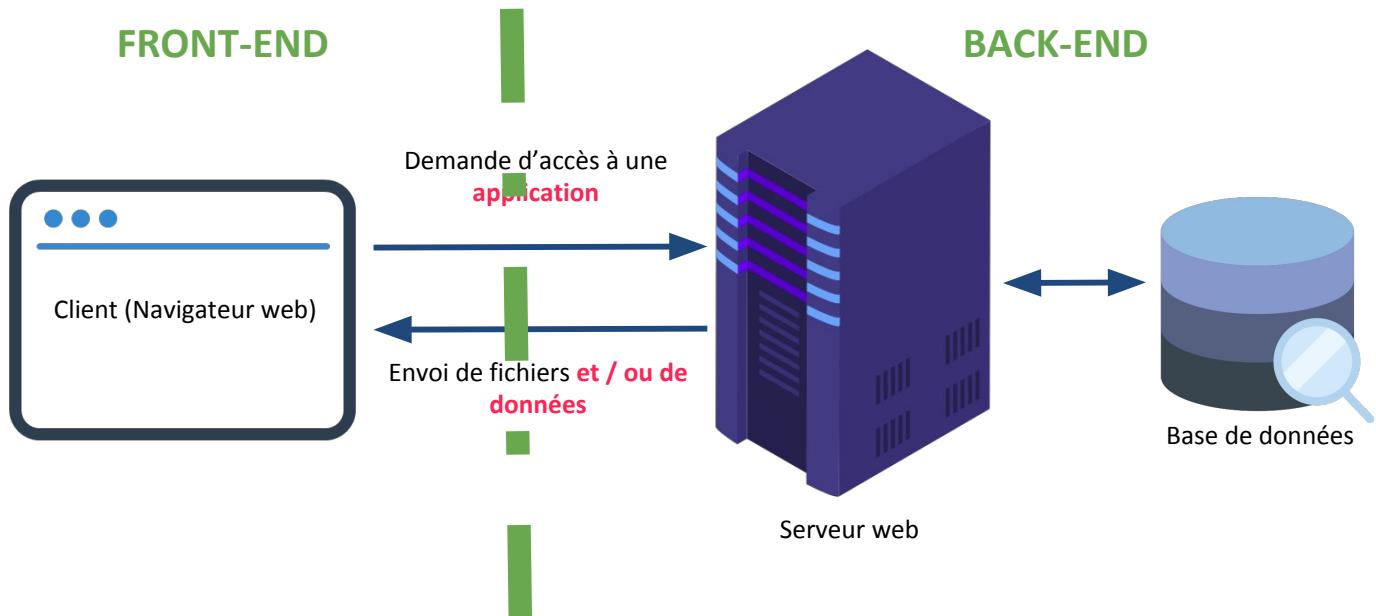
1. De sauvegarde de données en base de données
2. D'affichage différencié en fonction des utilisateurs
3. D'exécution de logiques métier spécifiques, en dehors de logiques d'affichage.

On est dans une logique **informative**.

A partir de maintenant, nous allons développer des **applications web**.

Là où un site web est informatif, une application web sera **interactive**.

Une application web interagit de manière directe ou indirecte avec une base de données.



9

## Point parcours

Dans le cas d'une application web, chaque utilisateur pourra disposer de sa propre interface.

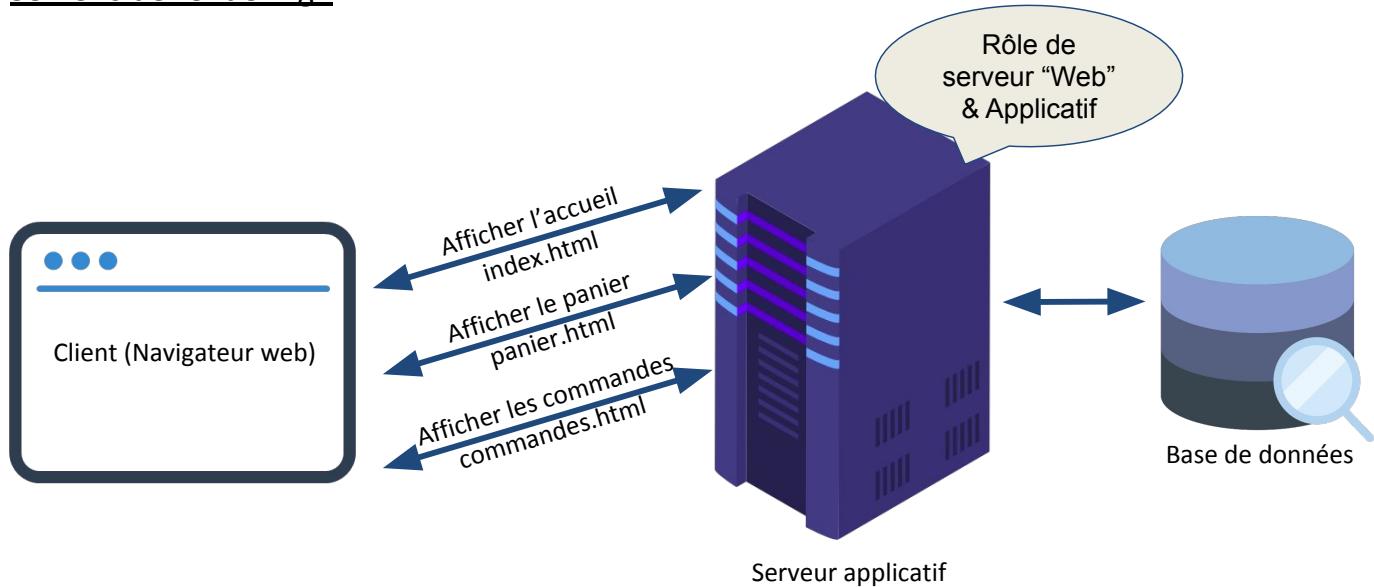
Par exemple, sur un site e-commerce :

1. Nous n'avons pas tous les mêmes suggestions de produits
2. Nous pouvons retrouver l'historique de nos commandes
3. Nous pouvons ajouter des produits dans un panier
4. Nous pouvons passer une commande et payer

Il existe deux grandes manières de développer une application web : le **server side rendering** ou le **client side rendering**.

# Point parcours

## Server side rendering :



11

## Point parcours

Dans le cas du server side rendering, nous allons utiliser un langage (par exemple, le PHP) pour générer des pages HTML au besoin en fonction des demandes des clients.

Nous allons donc générer des fichiers HTML **dynamiquement**, par opposition aux ressources statiques, dont le contenu est toujours le même.

Il existera toujours des ressources statiques, comme des images, du CSS, des polices de caractère... mais plus de fichier HTML "en dur".

Nos fichiers HTML seront générés à l'exécution sur notre serveur.

En terme de sémantique, on introduira la notion de serveur applicatif.

En effet, notre serveur aura désormais 2 rôles :

1. Héberger les ressources statiques (rôle du serveur web)
2. Exécuter le code permettant de générer les fichiers HTML au runtime (ie. à l'exécution) lorsque les navigateurs feront des demandes d'accès à certaines pages. On parle de serveur applicatif.

12

## Point parcours

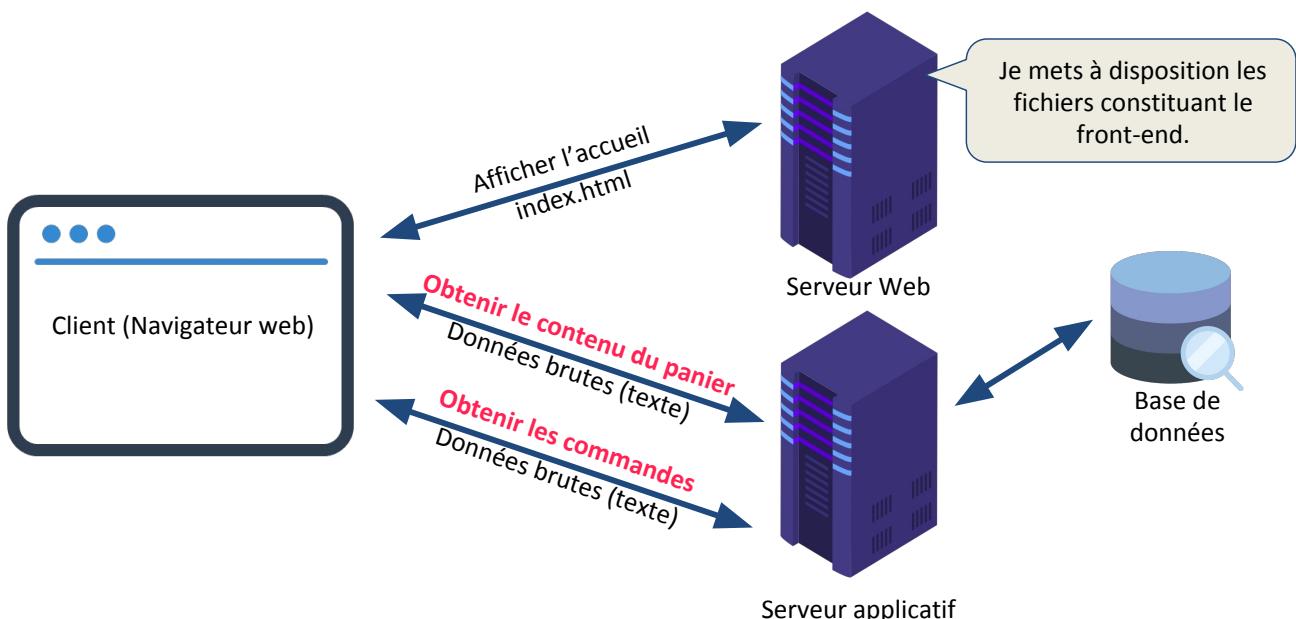
Dans le cas du server side rendering, notre application suit l'architecture “**multi-page application**” (MPA).

Cela signifie que nous avons développé le code nécessaire (en PHP ou dans un autre langage) pour générer dynamiquement des fichiers HTML en fonction des demandes des clients, et que chaque action menée par l'utilisateur conduira à recharger une nouvelle page.

13

## Point parcours

Client side rendering :



14

## Point parcours

---

Dans le cas du client side rendering, notre application suit l'architecture “**single-page application**” (SPA).

En effet le client ne télécharge qu'un seul fichier html : index.html

Ce fichier est autonome, et lorsque l'on navigue dans une SPA, on récupère des données brutes de la part du serveur.

C'est à dire qu'au lieu de nous envoyer de nouveaux fichiers HTML, le serveur web nous envoie uniquement des données non formatées.

C'est à la responsabilité du code JavaScript de récupérer ces données et de les traiter pour les afficher dans le DOM.

Le fichier index.html téléchargé est **autonome**.

15

## Point parcours

---

A l'issue de votre formation, vous serez initiés aux SPA et aux MPA :)

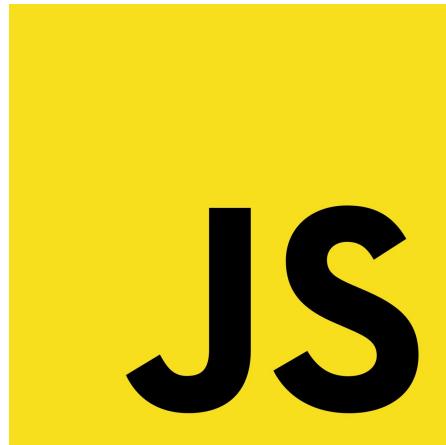
Lorsque l'on développe une SPA, on a une isolation forte entre le front et le back.

C'est moins vrai avec une MPA.

Chaque architecture a ses avantages et inconvénients.

Cette semaine, nous allons mettre en place une SPA.

16



17

## Introduction

---

Le JavaScript (JS) est un langage de script interprété.

Il suit le standard ECMAScript.

Deux grands cas d'utilisation :

1. Dans un navigateur web (client / front-end) : apporte du dynamisme aux pages
2. En dehors d'un navigateur :
  - a. Pour de l'exécution de scripts (comme un script cmd ou shell)
  - b. Côté serveur (back-end)

Ne pas confondre JavaScript avec JAVA !



JAVA est un langage différent de JavaScript à tout point de vue : son usage, sa sémantique...

D'ailleurs, il n'est pas interprétable nativement par un navigateur web !

18

# Introduction - ECMA

ECMA, historiquement “European Computer Manufacturers Association”

Nouveau nom :

“Ecma International - European association for standardizing information and communication systems”

ECMAScript est un standard ayant pour objet de spécifier le comportement de langages de scripts. Nom technique des spécifications : ECMA-262 et ECMA-402

Historiquement implémenté par ActionScript (langage utilisé par Flash), mais surtout par JS

9 versions majeures : ES1 > ES9

ES.Next est un nom de version dynamique qui fait toujours référence à la prochaine version d'ECMAScript.

19

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

# Introduction - ECMA

ECMAScript n'est pas la seule standardisation d'ECMA

ECMA-408 : Spécification du langage DART.

Initialement le DART avait pour vocation d'être une alternative au JS.

Finalement, dans un contexte web il est traduit en JS.



Flutter, le framework montant de Google,  
basé sur Dart

20

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

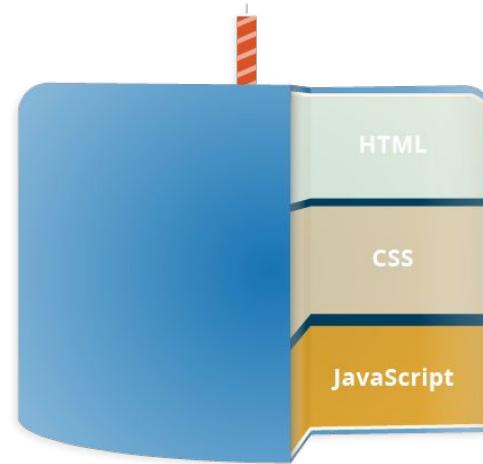
# Introduction - JavaScript dans un navigateur

Premier cas d'usage de JavaScript : dans un navigateur Web.

Permet d'ajouter du dynamisme à nos pages web. Dès lors qu'une page fait plus que simplement afficher du contenu statique (contenu identique / fixe quelque soit la personne qui se connecte ou les actions effectuées) JavaScript a de fortes chances d'être impliqué.

Exemples : un site e-commerce avec la possibilité d'ajouter des éléments dans un panier  
une carte interactive type maps  
des animations 2D/3D

JavaScript est souvent vu comme la 3ème couche après HTML (structure des pages) et CSS (mise en forme).



**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

21

## Introduction - JavaScript dans un navigateur

Les navigateurs principaux (Edge, Firefox, Chrome, Opéra, Safari) supportent tous ES6

Chrome 58	Edge 14
Jan 2017	Aug 2016

Firefox 54	Safari 10	Opera 55
Mar 2017	Jul 2016	Aug 2018

22

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

# Introduction - JavaScript standalone

Autre cas d'usage de JavaScript : en dehors d'un navigateur.

Pour du scripting simple, par exemple l'envoi d'un mail à un instant T.

Pour des réalisations plus complexes, comme un back-end applicatif.

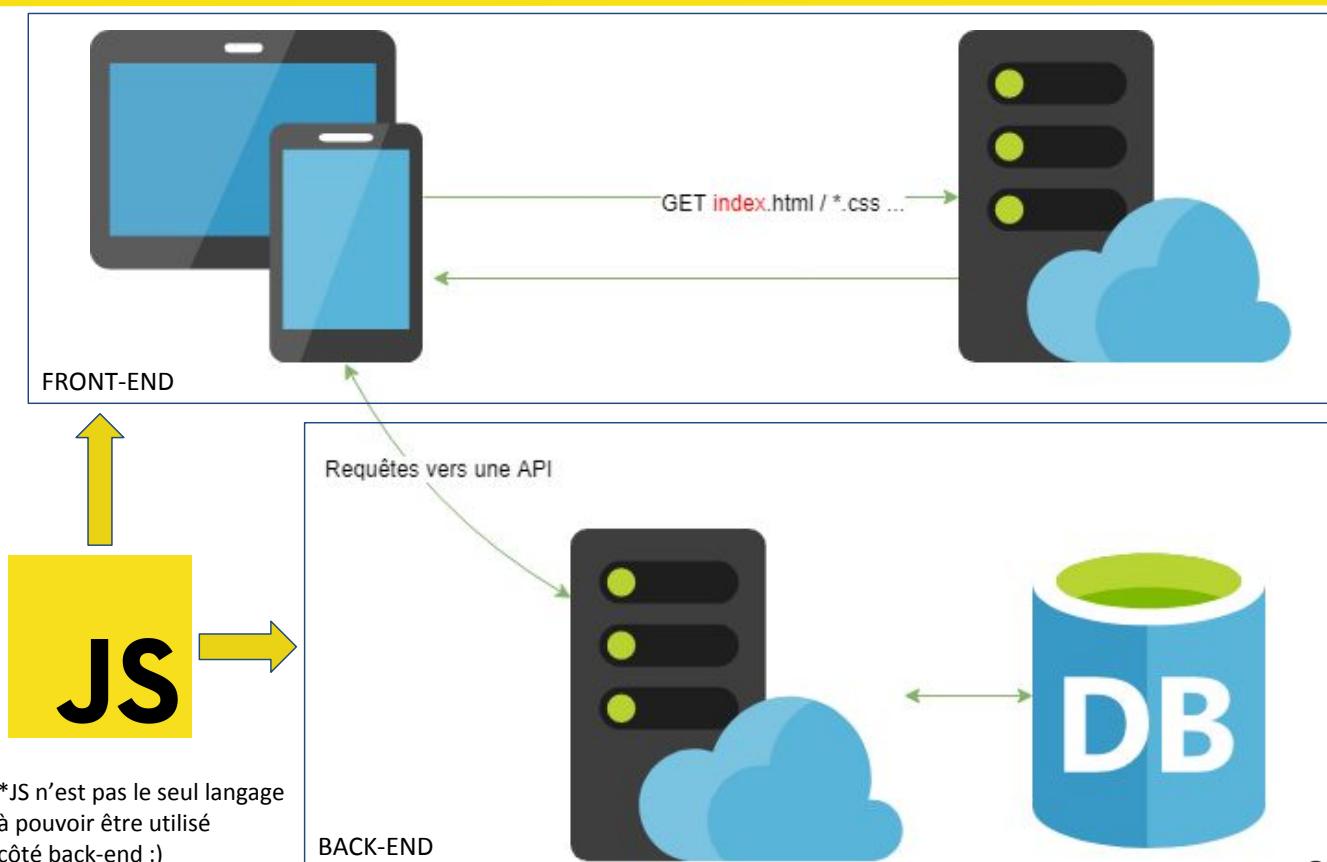


Node.JS, un programme permettant d'interpréter du JavaScript

23

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

## Introduction - Exemple d'architecture (SPA)



\*JS n'est pas le seul langage à pouvoir être utilisé côté back-end ;)

24

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

## Introduction - Installation Node.JS

---

Installez node.js : <https://nodejs.org/en/>

Créez un programme simple permettant d'afficher une liste d'étudiants d'une promotion dans la console, à partir d'un tableau.

Exécutez le via Node.JS avec la commande suivante : node <mon\_fichier.js>

25



## Les nouveautés ES6

---

{ ES6 JS }

26



## Nouveautés ES6 - Template Strings

En fonction des cas, utiliser l'opérateur + pour concaténer des chaînes peut être fastidieux...

Exemple :

```
1 let prenom = "Pierre";
2 let nom = "Dupont";
3 let nbMessages = 5;
4
5 let message = "Bonjour " + prenom + " " + nom + ". Vous avez " + nbMessages + " messages non lus.";
6
7 console.log(message);
```

Run >

> "Bonjour Pierre Dupont. Vous avez 5 messages non lus."

Reset

27



Nicolas Amini-Lamy

## Nouveautés ES6 - Template Strings

Depuis ES6, la notion de template string nous permet d'en finir avec l'enfer des + (surtout quand on doit juste ajouter un espace !) :

```
1 let prenom = "Pierre";
2 let nom = "Dupont";
3 let nbMessages = 5;
4
5 let message = `Bonjour ${prenom} ${nom}. Vous avez ${nbMessages} messages non lus.`;
6
7 console.log(message);
```

Run >

> "Bonjour Pierre Dupont. Vous avez 5 messages non lus."

Reset

28



Nicolas Amini-Lamy

## Nouveautés ES6 - Template Strings

Les template string permettent aussi de passer à la ligne :

```
1 let prenom = "Pierre";
2 let nom = "Dupont";
3 let nbMessages = 5;
4
5 let message = `Bonjour ${prenom} ${nom}.
6 Vous avez ${nbMessages} messages non lus.`;
7
8 console.log(message);
```

Run >

> "Bonjour Pierre Dupont.  
Vous avez 5 messages non lus."

Reset

29

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

## Nouveautés ES6 - Template Strings

Au delà de la simple substitution de variable, on peut également injecter des instructions JavaScript simples :

```
1 let prenom = "Pierre";
2 let nom = "Dupont";
3 let nbMessages = 5;
4
5 let message = `Bonjour ${prenom} ${nom}.
6 Vous avez ${++nbMessages} messages non lus.`;
7
8 console.log(message);
```

Run >

> "Bonjour Pierre Dupont.  
Vous avez 6 messages non lus."

Reset

30

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

## Nouveautés ES6 - Template Strings

On peut même utiliser une ternaire et une template string dans une template string (évitez cependant de créer trop de complexité) :

```
1 let prenom = "Pierre";
2 let nom = "Dupont";
3 let nbMessages = 5;
4
5 let message = `Bonjour ${prenom} ${nom}.
6 ${nbMessages != 0 ? `Vous avez ${nbMessages} messages non lus.` : "Vous n'avez pas de nouveaux messages."}`;
7
8 console.log(message);
```

Run >

Reset

> "Bonjour Pierre Dupont.  
Vous avez 5 messages non lus."

31



## Nouveautés ES6 - Classes

JavaScript est un langage qui permet de développer selon le paradigme de la programmation orientée objet.

Un paradigme est en quelque sorte une “manière” de programmer.

Vous avez déjà rencontré d’autres paradigmes :

1. Le paradigme impératif
2. Le paradigme déclaratif

32



# Paradigme impératif

Le paradigme de programmation impératif est l'un des paradigmes principaux que l'on rencontre en informatique.

Le principe est d'écrire des instructions les unes à la suite des autres.

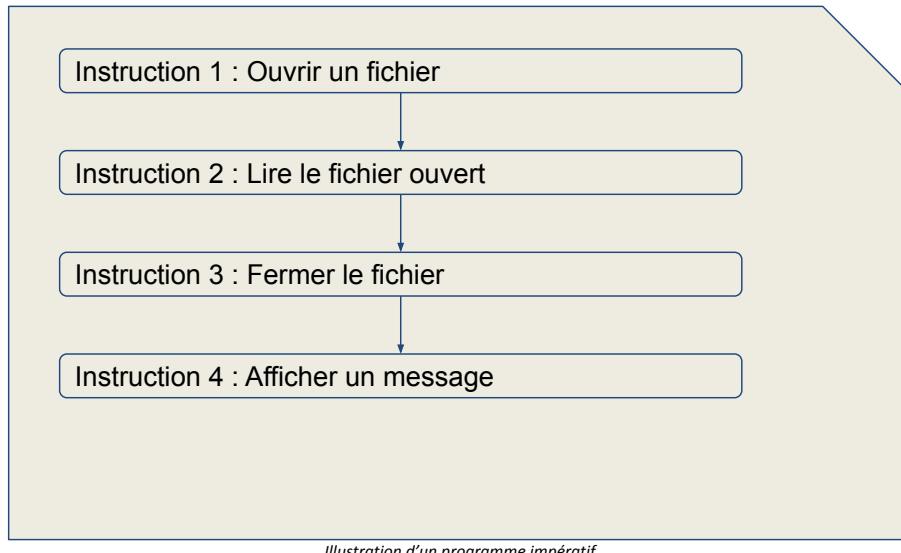


Illustration d'un programme impératif

33

## Paradigme déclaratif

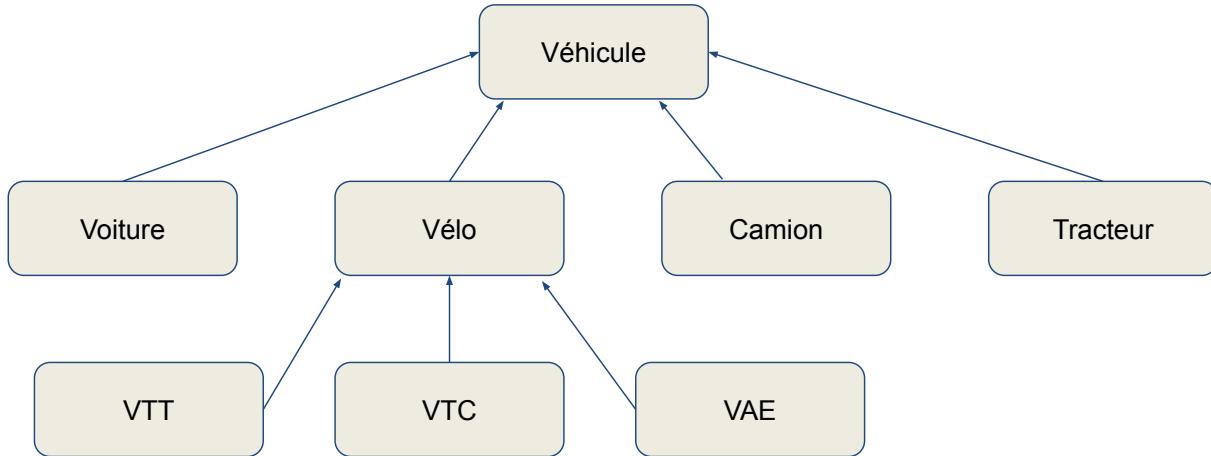
The screenshot shows a LaTeX editor window titled "LaTeXiT-1". Inside, there is a display area showing a mathematical integral: 
$$\int_1^{\infty} \frac{1}{x^2} dx = \left[ -\frac{1}{x} \right]_1^{\infty} = 1$$
. Below the display area, the LaTeX code for this integral is visible:  $\int_1^{\infty} \frac{1}{x^2} dx = \left[ -\frac{1}{x} \right]_1^{\infty} = 1$ . At the bottom of the editor, there are buttons for "Auto", "Align", "Display", and "Inline", and a color picker set to dark red. To the right of the editor, a portion of a larger HTML file is shown, containing code related to patient details and a search bar.

```
<div *ngIf="!isUserInPatientDetailsView" class="containerFiltres d-flex align-items-center">
<div class="btn-group filteringBlock" role="group">
<button #displayUnlockedPatientsButton type="button" class="btn btn-secondary">Afficher les patients déverrouillés</button>
<button #displayLockedPatientsButton type="button" class="btn btn-secondary">Afficher les patients verrouillés</button>
<button #displayAllPatientsButton type="button" class="btn btn-secondary">Afficher tous les patients</button>
</div>
<div class="input-group pl-2 filteringBlock">
<div class="input-group-prepend">
<span class="input-group-text" id="inputRecherche"><i class="fa fa-search"></i></span>
</div>
<input type="text" class="form-control" placeholder="Rechercher..." aria-label="Rechercher..." aria-describedby="inputRecherche"/>
</div>
</div>
<div *ngIf="!isUserInPatientDetailsView" [ngClass]="{disabledContent : isPatientDetailsView}" class="card patientsList">
<ngx-datatable class="material" [reorderable]="reorderable" [rowHeight]="getRowHeight()" [headerHeight]=>
```

HTML et LaTeX ; deux langages déclaratifs.

On décrit le “quoi” et non le “comment”.

34



35

## Nouveautés ES6 - Classes

Le principe de la POO est de modéliser notre application en “classes”.

Ces classes représentent des objets (pas forcément au sens physique) que l’on souhaite représenter dans notre application.

Lorsque l’on définit une classe, on détermine la structure et le comportement de tous les objets de cette classe.

Par exemple, un chien a 4 pattes. C’est une de ses caractéristiques.  
=> ses 4 pattes font donc partie de sa **structure**.

Un chien peut aboyer : on lui ajoutera une méthode permettant de le faire.  
=> cette méthode fera partie de son **comportement**.

36

# Nouveautés ES6 - Classes

## Vocabulaire élémentaire en POO :

Classe : Créer une classe, c'est créer un nouveau type de données. C'est une entité regroupant des variables (attributs) et des fonctions (comportements) que l'on souhaite associer car ils ont du sens entre eux.

Par exemple, tous les rectangles ont une longueur, une largeur. On peut également calculer leur aire. On peut donc créer une classe Rectangle.

Instance / Objet : une instance d'une classe est un objet construit à l'aide de la classe.

On peut avoir 10 rectangles, ils auront tous une longueur / largeur et on peut calculer l'aire. On va donc utiliser notre classe Rectangle.

Propriété / Attribut : Element de structure. Exemple : une personne peut avoir un nom, prénom, un sexe, une couleur de cheveux...

Les propriétés d'un rectangle sont : sa longueur, sa largeur.

37



# Nouveautés ES6 - Classes

Méthode : action applicable à un objet. Une méthode permet d'implémenter un comportement.

On peut créer une méthode permettant de calculer l'aire d'un rectangle.  
Le calcul sera identique quelque soit le rectangle.

Héritage : Une classe peut hériter d'une autre classe. On dit qu'elle la **spécialise**. Par exemple, une classe Chat peut hériter de la classe Animal.

Lorsqu'une classe hérite d'une autre classe, elle hérite des caractéristiques et du comportement de la classe héritée. On parle de classes mères / filles.

Parfois, on utilise le verbe "étendre" en lieu et place d'"hériter".

Constructeur : Le constructeur est une méthode particulière, obligatoire dans toute classe, qui permet d'initialiser les attributs d'une instance d'une classe.

38



## Nouveautés ES6 - Classes

```
1 class Animal {
2   constructor(categorie) {
3     this.categorie = categorie;
4   }
5   direBonjour() {
6     console.log("Bonjour, je suis un animal et un " + this.categorie);
7   }
8 }
9
10 class Chat extends Animal {
11   constructor(nom) {
12     super("Vertébré");
13     this.nom = nom;
14   }
15 }
16
17 let monChat = new Chat("Felix");
18 console.log(monChat.nom);
19 console.log(monChat.categorie);
20 monChat.direBonjour();
```

39



## Nouveautés ES6 - Classes

La notion de Classe existe en JavaScript via ES6.

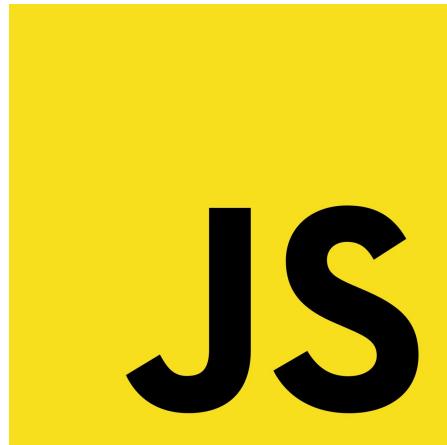
Il s'agit d'un syntaxic sugar pour écrire de la POO en JavaScript sans passer par son fonctionnement natif à base de prototypes.

Pour instancier une classe, il faut utiliser le mot clef **new**.

Ce mot clef new est un type d'appel particulier sur les fonctions.

40





Retour sur les fonctions

41

## Les fonctions - Déclaration

---

Le principe d'une fonction est le même quelque soit le langage : regrouper des instructions qui ont du sens ensemble et qui sont réutilisables, afin de gagner en lisibilité et en maintenabilité.

En JavaScript, une fonction peut se créer de différentes manières.

La syntaxe “classique” est la suivante :

```
1 function additionnerDeuxNombres(nombre1, nombre2) {  
2     return nombre1 + nombre2;  
3 }  
4  
5 console.log(additionnerDeuxNombres(10, 20));
```

42

# Les fonctions - Portées de variables

Dans une fonction il est possible d'appeler des variables déclarées précédemment.

```
1 var nbDecimales = 2;
2
3 function additionnerDeuxNombres(nombre1, nombre2) {
4   return (nombre1 + nombre2).toFixed(nbDecimales);
5 }
6
7 console.log(additionnerDeuxNombres(10.34, 20.45643));
```

43



## Les fonctions - Portées de variables

L'accès aux variables “parentes” est opérationnel même si la fonction parente a terminé son exécution :

```
1 function uneFonction() {
2   let variableFonctionParente = "Démonstration";
3
4   function fonctionEnfant() {
5     console.log(variableFonctionParente + " des closures");
6   }
7
8   setTimeout(fonctionEnfant, 3000); →
9 }
10
11 uneFonction();
```

Run >

Reset

> "Démonstration des closures"

fonctionEnfant sera exécutée “dans” 3 secondes, alors que l'exécution de “uneFonction” sera terminée depuis plusieurs secondes.

Pour autant, le scope de variable est conservé et la fonction enfant peut toujours accéder à la variable du scope parent.

C'est le principe des closures (“fermetures”).

44



# Les fonctions - Invocations

Il existe plusieurs manières d'invoquer une fonction en JavaScript :

- ◆ comme une fonction (oui... )
- ◆ comme une méthode
- ◆ comme un constructeur
- ◆ avec apply() ou call()

Les fonctions sont un concept clef du langage et certaines spécificités sont souvent mal comprises lorsque l'on vient d'autres langages (comme JAVA !).

Lorsque l'on appelle une fonction (quelque soit la méthode) ; le moteur JavaScript ajoute automatiquement (et implicitement) 2 paramètres : **this** et **arguments**.

45

## Les fonctions - Invocations

Appeler une fonction... “comme une fonction”.

C'est le premier type d'appels, le plus simple.

The screenshot shows a code editor on the left with the following JavaScript code:

```
1 function exemple(param) {  
2   console.log(this, arguments);  
3 }  
4  
5 exemple("test");
```

Two yellow arrows point from the text "Contexte" de la fonction and "Paramètres de la fonction" to the "this" and "arguments" parameters in the code respectively. To the right of the code is a browser developer tools console window with the following output:

Run > > [object Window] Object { 0: "test" }  
Reset

Below the arrows, the text "Contexte" de la fonction and "Paramètres de la fonction" are written.

Lorsque l'on appelle une fonction de cette manière, la variable **this** correspond à l'objet **window** (dans un contexte web) ou **global** (contexte node).

46

# Les fonctions - Invocations

Seconde manière d'appeler une fonction : comme une méthode.

L'idée est d'attacher une fonction à un objet ; et d'appeler cette fonction sur l'objet.

C'est de cette manière que l'on peut faire de la programmation orientée objet en JavaScript.

```
1 var robot = {  
2   prenom: "Henri"  
3 };  
4  
5 robot.direBonjour = function() {  
6   console.log(this);  
7   console.log("Bonjour");  
8 }  
9  
10 robot.direBonjour();
```

Run >

Reset

```
> Object { prenom: "Henri", direBonjour: function() {  
    console.log(this);  
    console.log("Bonjour");  
} }  
> "Bonjour"
```

Dans ce cas, la variable `this` correspond à l'objet "robot".

On retrouve le sens classique de la variable "this" tel qu'on peut le rencontrer dans des langages comme JAVA.

47

# Les fonctions - Invocations

Troisième manière d'appeler une fonction : comme un constructeur avec le mot clef `new`.

```
1 function exemple(param) {  
2   console.log(this, arguments);  
3 }  
4  
5 exemple("test");  
6  
7 new exemple("test");
```

Run >

Reset

```
> [object Window] Object { 0: "test" }  
> [object Object] Object { 0: "test" }
```

Lorsque l'on utilise `new` :

- ◆ un nouvel objet (vide) est créé
- ◆ cet objet est passé au constructeur et correspond au `this`
- ◆ en l'absence d'un return dans le constructeur, ce nouvel objet est renvoyé implicitement.

48

# Les fonctions - Invocations

L'invocation par constructeur est pratique pour définir des objets en POO.

Sans utiliser cette méthode, pour créer deux objets de la même classe, on devrait écrire :

```
1 function direBonjour() {  
2   console.log("Bonjour");  
3 };  
4  
5 var robot = {  
6   prenom: "Henri",  
7   direBonjour : direBonjour  
8 };  
9  
10 var robot2 = {  
11   prenom: "Jacques",  
12   direBonjour : direBonjour  
13 };
```

```
1 function Robot(prenom) {  
2   this.prenom = prenom;  
3   this.direBonjour = function() {  
4     console.log("Bonjour");  
5   };  
6 }  
7  
8 var henri = new Robot("Henri");  
9 var jacques = new Robot("Jacques");  
10  
11 henri.direBonjour();  
12 jacques.direBonjour();
```

Run >

Reset

> "Bonjour"

> "Bonjour"

49

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

## Les fonctions - Invocations

Une fonction dispose de ses propres méthodes. En effet, toute fonction en JavaScript est un objet Function. Cet objet permet d'utiliser certaines méthodes dont apply() et call().

La dernière manière d'invoquer une fonction est de faire appel à apply() et call().

Apply comme call permettent d'appeler une fonction tout en surchargeant son contexte, c'est à dire la valeur de this.

La différence résulte dans la signature de la fonction au niveau du passage des arguments (via un tableau ou non).

```
1 function direBonjour(nomInterlocuteur) {  
2   console.log(this);  
3   console.log("Bonjour " + nomInterlocuteur);  
4 }  
5  
6 direBonjour.apply({}, ["Jean"]);  
7 direBonjour.call({}, "Jean");
```

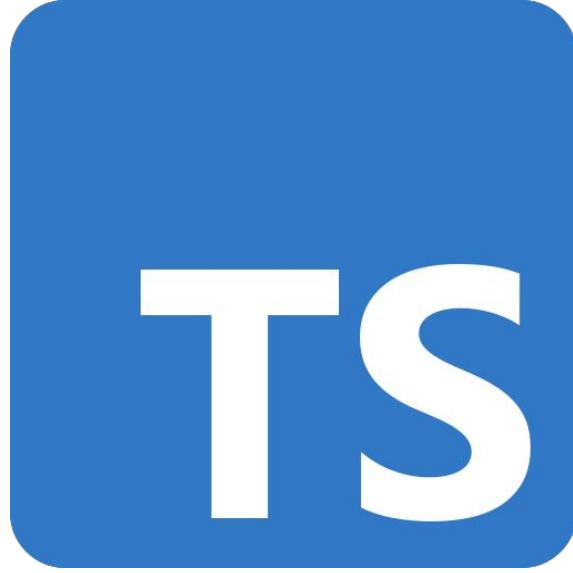
Run >

Reset

> Object { }  
> "Bonjour Jean"  
> Object { }  
> "Bonjour Jean"

50

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy



51

## Introduction

---

JavaScript est un langage :

- ◆ Interprété : il n'y a aucune phase de compilation, le code n'est pas vérifié avant son exécution.
- ◆ Faiblement typé : une variable n'est pas conditionnée à un seul type de données : il est possible de changer son type lors de l'exécution du programme.

Ces caractéristiques ont des avantages mais aussi des inconvénients :

- ◆ Nécessité de tester très rigoureusement son code
- ◆ Pas d'aides lors du développement
- ◆ Apparition de potentiels bugs au runtime

Pour pallier à ces inconvénients, il est possible :

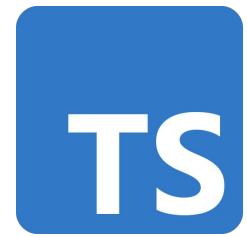
- ◆ d'utiliser des librairies type flow (un vérificateur de types statique)
- ◆ d'utiliser un *linter* sur son projet (le plus connu : ESLint -voir bibliographie-) afin d'avoir une remontée d'alertes sous forme de warning sur son projet.

Une solution plus complète consiste à développer en TypeScript.

52

# Introduction

TypeScript est une surcouche du langage JavaScript développée par Microsoft.



Le principe global du TypeScript est de typer toutes les variables.

Néanmoins, utiliser du TypeScript ne se résume pas simplement à cela.

Le langage vise à apporter un certain nombre d'améliorations vis à vis du JavaScript, de la programmation orientée objet à la programmation générique par exemple.

Lorsque l'on développe en TypeScript (extension de fichier conventionnelle : .ts) on doit utiliser un compilateur qui traduira finalement notre code en JavaScript.

On parle d'ailleurs plutôt de transcompilation, et non de compilation, puisque l'on traduit un code écrit dans un langage en un code d'un langage du même niveau (à l'inverse de la compilation où l'on passe d'un langage lisible humainement à un langage bas niveau).

53

## TP JavaScript

1) Créez une fonction JavaScript permettant d'additionner deux nombres

2) Créer une page web permettant de saisir deux nombres et d'afficher le résultat de l'addition

54

# Installation TypeScript

---

Installez typeScript via la ligne de commande :

**npm install -g typescript**

55



## TP TypeScript

---

Créez un nouveau projet en reprenant le TP précédent.

Cette fois-ci, écrivez votre code dans un fichier .ts et ajoutez à vos nombres le type number :

```
function calculerSomme(operande1: number, operande2: number)
```

Utilisez la commande tsc <mon\_fichier.ts> pour générer votre fichier JS.

56



# TP TypeScript - Observations

TypeScript nous alerte sur les deux points suivants :

- ◆ Lorsque l'on récupère les éléments du DOM, il peut s'agir de n'importe quel type de noeud.  
La propriété value n'est donc pas forcément accessible. TypeScript nous invite explicitement à vérifier que l'on travaille bien avec une balise de type input.
  - ◆ Notre fonction d'addition prend en paramètre 2 nombres, et la valeur récupérée de notre input est un string. Il nous force donc à faire la conversion, ce qui nous prévient d'un bug à l'exécution si l'on n'avait pas fait attention.
- C'est un exemple simple, mais dans une application plus complexe, le fait de typer explicitement nos variables nous aidera grandement à ne pas écrire d'erreurs (par exemple, accéder à une propriété d'un objet en l'écrivant mal).

57

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

## Npm



58

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

# Npm - Généralités

Npm est un **gestionnaire de dépendances**.

L'acronyme signifie “Node Package Manager”.

Historiquement utilisé sur des projets Node (ie. JS hors Web), il l'est désormais également sur presque tous les projets de développement Web, y compris Front.

Lorsque l'on parle de NPM, on peut évoquer :

- ◆ Son site web
- ◆ Son registry
- ◆ Sa CLI (command line interface)

59

# Npm - Généralités

Le principe général de NPM est de mettre à disposition une gigantesque base de données (le **registry** npm) contenant des librairies JS publiques ou privées.

Chaque librairie doit suivre le versionning **semver**, dont le principe général est le suivant :

- ◆ Un numéro de version est sous la forme MAJOR.MINOR.PATCH
- ◆ On incrémente le numéro :
  - Majeur lorsque l'on a un changement d'API qui introduit des incompatibilités avec le code utilisant la librairie dans sa version précédente ;
  - Mineur lorsque l'on ajoute de nouvelles fonctionnalités rétro-compatibles avec la version précédente ;
  - Patch lorsque l'on corrige des bugs qui n'introduisent pas d'incompatibilités.

60

Les intérêts de NPM sont :

- ◆ De ne plus avoir besoin d'aller télécharger soi même ses dépendances
- ◆ De bien isoler ses sources de ses dépendances (on ne versionne **jamais** une librairie)
- ◆ De gérer les dépendances et les incohérences de versions
- ◆ D'être certain de récupérer les librairies dans les bonnes versions
- ◆ D'être certain d'avoir téléchargé correctement nos librairies (pas de fichiers corrompus)

En bref... l'époque du téléchargement manuel de nos librairies est révolu :)

61

## Npm - En pratique

Dans un projet utilisant NPM, on retrouve deux fichiers principaux :

- ◆ package.json
- ◆ package-lock.json

Le fichier package.json contient :

- ◆ Des informations générales sur notre projet
- ◆ Des scripts (alias de commandes)
- ◆ La liste des dépendances de notre projet (on peut indiquer des dépendances de production ou de développement)

Le fichier package-lock.json contient des métadonnées issues de l'installation des packages et permet de figer les versions téléchargées entre développeurs.

Il existe également un répertoire **node\_modules** qui contiendra l'ensemble des librairies téléchargées via npm sur un projet.

62

Les commandes principales de Npm sont :

- ◆ npm init
- ◆ npm install
- ◆ npm remove
- ◆ npm publish
- ◆ npx <nom d'un package npm>

Exemples d'installation de dépendances :

- ◆ npm install @angular/core
- ◆ npm install @angular/core@latest => équivalent à la commande précédente
- ◆ npm install @angular/core@9.1.0 => permet d'obtenir une version spécifique
- ◆ npm install -g @angular/cli => Installation d'un package au niveau OS
- ◆ npx create-react-app helloworld => Exécution d'un package (ici init. d'une app react) sans l'installer
- ◆ npm install typescript --save-dev => Installation d'un package qui servira uniquement côté développement (ne sera pas inclus lors d'un packaging de production)

63

## Npm - En pratique

Lorsque l'on ajoute une dépendance à notre projet, on peut contrôler la plage de versions autorisées lors de l'installation des dépendances sur un autre poste / répertoire (indispensable lorsque l'on travaille en équipe).

Voici les différentes options :

- ◆ Fixer une version :

```
"@angular.core": "9.1.0"
```

- ◆ Fixer le numéro de version majeur :

```
"@angular.core": "^9.1.0"
```

*Autorise les versions 9.1.0 à 9.X.X*

- ◆ Fixer les numéros de version majeur & mineur :

```
"@angular.core": "~9.1.0"
```

*Autorise les versions 9.1.0 à 9.1.X*

Il existe des variantes (“<x.x.x”,”x.x.x - x.x.x”).

Cela fonctionne dans le meilleur des mondes si les développeurs des packages que l'on utilise respectent bien la norme **semver**.

64

# Npm - En pratique

Lorsque l'on installe un package, npm récupère la dernière version autorisée de celui-ci en fonction des règles établies dans le fichier package.json.

Problème : deux développeurs peuvent se retrouver avec des versions différentes des librairies sur un même projet.

Cas simple : une dépendance dont vous avez spécifié une latitude sur le numéro de patch ou mineur.

Plus vicieux : vous avez spécifié des numéros de version précis, mais une dépendance de vos dépendances a été mise à jour sans que vous ne le sachiez.

Encore plus vicieux : le meilleur des mondes n'existe pas et le développeur d'un package se contrefiche de la norme semver (c'est vilain).

65



## Npm - En pratique

As an example, consider package A:

```
{  
  "name": "A",  
  "version": "0.1.0",  
  "dependencies": {  
    "B": "<0.1.0"  
  }  
}
```

Après un npm i :

```
A@0.1.0  
`-- B@0.0.1  
     '-- C@0.0.1
```

package B:

```
{  
  "name": "B",  
  "version": "0.0.1",  
  "dependencies": {  
    "C": "<0.1.0"  
  }  
}
```

Si B est mis à jour en 0.0.2, une nouvelle installation donnera l'arbre de dépendances suivant :

```
A@0.1.0  
`-- B@0.0.2  
     '-- C@0.0.1
```

and package C:

```
{  
  "name": "C",  
  "version": "0.0.1"  
}
```

Pour répondre à ces problématiques, npm génère un fichier package-lock.json

Ce fichier a pour but de figer (d'où "lock") l'arbre de dépendances téléchargées lors d'un npm install.

De cette manière lorsqu'un développeur clone un repository contenant ce fichier package-lock on est certains de télécharger exactement le même arbre de dépendances.

Ce fichier doit être versionné et mis à jour régulièrement (à voir en fonction de votre politique projet).

67

## Npm - son successeur arrive

Il existe un projet : YARN ("Yet another package manager").

Ce projet vise à améliorer NPM principalement en terme de rapidité de téléchargement des dépendances puisqu'il s'effectue en parallèle là où npm fonctionne de manière séquentielle.

Pour l'utiliser, il faut l'installer globalement : `npm install -g yarn`

Puis pour installer les dépendances d'un projet : `yarn install`

La différence de perf n'est pas toujours flagrante.

68



Nous allons désormais aborder les bases du TypeScript :

- ◆ Types primitifs
- ◆ Tuples
- ◆ Énumérés
- ◆ Any
- ◆ Union
- ◆ Type littéral
- ◆ Alias
- ◆ Retours de fonctions
- ◆ Never

69

## Types primitifs et généralités

En TypeScript, le type d'une variable est déclaré ou inféré :

```
let unNombre: number = 10; //Type déclaré
let unAutreNombre = 10; //Type inféré
unAutreNombre = "test"; //Le type de la variable est bien number
```

Généralement on utilise l'inférence de types lorsque l'initialisation est immédiatement consécutive à la déclaration de la variable. On prendra pour bonne habitude de typer explicitement toutes les variables non initialisées immédiatement (exemple : paramètres d'une fonction).

TypeScript et JavaScript fonctionnent de la même manière pour les types :

- ◆ number
- ◆ string
- ◆ boolean

70

# Types primitifs et généralités

Lorsque l'on manipulera des objets, TypeScript va inférer ses propriétés :

```
let etudiant = {
    prenom : "Pierre",
    nom : "Dupont"
};  
any  
Property 'nomComplet' does not exist on type '{ prenom: string; nom: string; }'. ts(2339)  
Peek Problem (Alt+F8) No quick fixes available  
etudiant.nomComplet = "Pierre Dupont";
```

L'objectif est de s'assurer de l'existence des propriétés / méthodes auxquelles on accède.

71



Nicolas Amini-Lamy

## Types primitifs et généralités

Il est possible de “surcharger” l’inférence de types pour un objet :

```
let etudiant: {prenom: string, nom: string, nomComplet: string};  
  
let etudiant: {  
    prenom: string;  
    nom: string;  
    nomComplet: string;  
}  
Property 'nomComplet' is missing in type '{ prenom: string; nom: string; }' but required in  
type '{ prenom: string; nom: string; nomComplet: string; }'. ts(2741)  
index.ts(17, 45): 'nomComplet' is declared here.  
Peek Problem (Alt+F8) No quick fixes available  
etudiant = {  
    prenom : "Pierre",  
    nom : "Dupont"  
};
```

Cela présente néanmoins assez peu d'intérêt, souvent on passera plutôt par un type / une classe personnalisé(e).

72



Nicolas Amini-Lamy

# Types primitifs et généralités

Lorsque l'on manipule des tableaux, on déclare un type de variable associé :

```
let tabNotes: number[];  
let tabNotesInfere = [10, 12, 14];  
tabNotesInfere.push("test");
```

Dans une boucle sur un tableau, le type de la variable de boucle est automatiquement déduit :

```
let tabNotes: number[];  
let tabNotesInfere = [10, 12, 14];  
  
let note: number  
for (let note of tabNotesInfere) {  
    console.log(note);  
}
```

73

## Tuples

En TypeScript il est possible de créer des **tuples**.

Un tuple est un ensemble de 2 valeurs. En quelque sorte, c'est un tableau qui ne contiendra que 2 éléments.

Exemples :

```
let role: [number, string];  
  
role = [0, "User"];  
  
role = ["User", 0]; //incohérence de types  
  
role[0].toFixed(2); //le type est connu  
role[0].substring(1); //substring est applicable sur un string, pas un number  
  
role[2] = "test"; //impossible  
  
role.push("test"); //Par contre, on reste sur le prototype array !
```

74

# Énumérés

Un énuméré TypeScript est en quelque sorte une liste d'alias.

```
enum Color {  
    RED,  
    GREEN,  
    BLUE  
}  
  
let rouge = Color.RED;  
  
console.log(rouge); //0  
console.log(Color.GREEN); //1  
console.log(Color[2]); // "BLUE"
```

Par défaut, chaque valeur de l'enum est traduite en un nombre (sa position dans l'enum), en partant de 0.

Il est possible de démarrer à 1 :

```
enum Color {  
    RED = 1,  
    GREEN,  
    BLUE  
}
```

75

# Énumérés

Il est possible d'utiliser un enum avec pour valeur des strings et non des numbers :

Attention par contre dans ce cas, ce n'est pas un dictionnaire où l'on peut passer facilement de la clef à la valeur et vice versa...

```
enum Roles {  
    ADMIN = "Administrateur",  
    USER = "Utilisateur"  
}  
  
console.log(Roles.ADMIN); //Administrateur  
console.log(Roles["Administrateur"]); //undefined
```

76

## Any

Il est possible d'utiliser le mot clef any si l'on ne souhaite pas préciser de type. Cela revient à développer en JS natif donc l'intérêt reste limité.

```
let uneVariableSansType: any;  
  
uneVariableSansType = "test";  
uneVariableSansType = 34;
```

Any est surtout utilisé lorsque les déclarations de type ne sont pas connues (exemple une librairie qui ne les fournit pas).

77

## Union

Il est possible d'utiliser une union de type, c'est à dire de définir le fait qu'une variable puisse être d'un type **ou** d'un autre.

On peut changer de type comme on le souhaite.

```
let chaineOuNombre: string | number;  
  
chaineOuNombre = 13;  
console.log(typeof chaineOuNombre); //number  
chaineOuNombre = "test";  
console.log(typeof chaineOuNombre); //string  
  
chaineOuNombre = false; //interdit !
```

On peut indiquer autant de types possibles qu'on le souhaite :

```
let chaineOuNombre: string | number | boolean ;
```

78

Le type littéral permet de définir des valeurs exactes.

Cela peut être pratique en combinaison avec l'union de types. Par exemple, si l'on souhaite que la valeur d'une variable soit comprise dans un ensemble de valeurs possibles :

```
let maVariable : "valeur_possible_1" | "valeur_possible_2" | 3;  
  
maVariable = "autre valeur"; //erreur  
maVariable = 3;  
maVariable = "valeur_possible_1";
```

79

## Alias de type

Afin d'éviter de répéter une définition de type, on peut créer un alias.

```
type MonAliasDeTypes = string | number;  
type MonAutreAliasDeTypes = "valeur_1" | "valeur_2";
```

Fonctionne également pour des objets :

```
type Produit = { titre: string; prix: number };  
  
const p1: Produit = { titre: "Chaussures", prix: 45};  
  
const p2: Produit = { title: "test", price: 20, stock: 40 } //génère une erreur
```

80

# Retours de fonctions

Le type de retour d'une fonction peut être inféré ou défini.

Préférence personnelle : définir pour chaque fonction son type de retour. Cela aide à la lisibilité.

```
function uneFonction(): string {  
    return "Hello function";  
}
```

```
function uneFonction(): string  
uneFonction();
```

```
function uneFonction() {  
    return "Hello function";  
}
```

```
function uneFonction(): string  
uneFonction();
```

Lorsqu'une fonction ne retourne rien, le type inféré par TypeScript est **void**. Void signifie “rien” et est différent de undefined.

```
function uneFonction() { }  
  
function uneFonction(): void  
uneFonction();
```

81

## Function

Il est possible d'utiliser le type **Function**.

Exemple :

```
function uneFonction() { }  
  
uneFonction();  
  
let monPointeurDeFonction: Function;  
monPointeurDeFonction = uneFonction;  
monPointeurDeFonction = false; //erreur
```

On peut préciser la signature attendue de la fonction.

Exemple :

```
function uneFonction() { }  
function uneAutreFonction(p1:number) { return p1 };  
  
let monPointeurDeFonction: (p1: number) => number;  
  
monPointeurDeFonction = uneFonction; //signature incorrecte  
monPointeurDeFonction = uneAutreFonction;
```

82

# Function

Dans le cadre de la programmation asynchrone, on utilise souvent cette notation.

Exemple :

```
function addAndHandle(n1: number, n2: number, callback: (num: number) => void) {  
    callback(n1 + n2);  
}
```

Ecrire void ici signifie que l'on ne s'intéresse pas à l'éventuel type de retour du callback s'il en existe un.

```
function addAndHandle(n1: number, n2: number, callback: (num: number) => number) {  
    let resultatCallback = callback(n1 + n2);  
    let resultatCallback: number  
    console.log(resultatCallback);  
}
```

83

# Unknown

Le type **unknown** est à utiliser lorsque l'on ne connaît pas le type d'une variable à l'avance.

Il est différent de **any** dans le sens où il ne désactive pas la vérification de types et force le transstypage / la vérification de type (nous en parlerons plus tard) :

```
let variableDontJeNeConnaisPasLeType: unknown;  
let unNombre: number;  
  
variableDontJeNeConnaisPasLeType = 42;  
  
//type unknown non assignable au type number  
unNombre = variableDontJeNeConnaisPasLeType;  
  
//force la vérification  
if (typeof variableDontJeNeConnaisPasLeType === "number") {  
    unNombre = variableDontJeNeConnaisPasLeType; //OK  
}
```

84

Le type **never** est utilisé pour définir le fait qu'une fonction ne se terminera jamais.

Cas typique : fonction utilitaire pour le déclenchement d'exceptions :

```
function fonctionSansRetour(): never {
    throw new Error("Ma fonction ne se termine pas");
    console.log("Ceci ne sera jamais affiché");
}

fonctionSansRetour();
```

Autre exemple : boucle infinie d'écoute d'événements.

```
function fonctionSansRetour(): never {
    while(true) {
        //traitement métier
    }
}

fonctionSansRetour();
```

85

## Le compilateur TypeScript

---

La commande tsc fait appel au compilateur typescript.

tsc traduit du code typescript en code javascript.

Il est possible d'utiliser le **watch mode**, ce qui permet d'être en écoute permanente des changements côté TypeScript (à chaque sauvegarde du fichier) et régénérer automatiquement le code JavaScript.

Pour lancer le watch mode : `tsc <mon_fichier.ts> -w`

On peut modifier le comportement de la compilation :)

86

# Compilateur - configuration

La compilation TypeScript > JavaScript peut être paramétrée à l'aide d'un fichier **tsconfig.json**

Ce fichier est généré en lançant la commande : `tsc --init`

On peut désormais compiler directement tous les fichiers TS de notre répertoire : `tsc` ou `tsc -w`

Dans ce cas, tous nos fichiers JS et TS sont dans le même répertoire... c'est peu pratique.

Première configuration : on définit un répertoire pour les sources TS, un autre pour les fichiers JS générés (la structure du répertoire source sera conservée) :

```
"outDir": "./dist",
"rootDir": "./src",
à modifier dans tsconfig.json
```



87

# Compilateur - configuration

Il est également possible d'inclure / exclure certains fichiers / répertoires de la compilation via la configuration exclude (ou à l'inverse, include -les deux sont à définir-) :

```
"exclude": [
  "*.*model.ts",
  "**/*.*model.ts"
],
"include": [
  "**/*.*ts"
]
```

Par défaut, le répertoire node\_modules est exclu (on ne compile pas les libs). Pensez à l'indiquer si vous redéfinissez exclude.

88

Le code TypeScript est traduit en JavaScript.

Vous avez la possibilité de choisir la version du standard EcmaScript utilisée !

Pour cela il faut modifier la propriété `target`

Les valeurs possibles sont :

- ▶ "ES3" (default)
- ▶ "ES5"
- ▶ "ES6"/"ES2015"
- ▶ "ES2016"
- ▶ "ES2017"
- ▶ "ES2018"
- ▶ "ES2019"
- ▶ "ES2020"
- ▶ "ESNext"

89

## Compilateur - configuration (librairies)

Il est possible de préciser quelles fonctions / librairies sont utilisables dans notre projet.

Par exemple, pour du code TS associé à une application web, on dispose de l'API DOM.

Pour cela il faut paramétrier l'option `lib`. Cette option indique également quelle version d'ES on peut utiliser. Pour pouvoir appeler les fonctions apportées par ES6 par exemple, il faut l'ajouter dans le tableau des libs. Configuration par défaut :

- ▶ For `--target ES5: DOM,ES5,ScriptHost`
- ▶ For `--target ES6: DOM,ES6,DOM.Iterable,ScriptHost`

Consulter la doc : <https://www.typescriptlang.org/docs/handbook/compiler-options.html>

90

Il existe une fonctionnalité dite de “source mapping” permettant de faire le lien entre le code TypeScript écrit et le code JavaScript généré.

Ces fichiers de correspondance suivent l'extension .js.map

Ils sont compris par les outils de debug modernes et permettent de déboguer directement le code TypeScript !

Pour activer la génération de ces fichiers, activez l'option sourceMap (=true)

91

## Compilateur - configuration

Par défaut les fichiers JavaScript sont générés même si le code TypeScript comporte des erreurs.

On peut désactiver ce comportement en modifiant l'option **noEmitOnError**.

Il est possible d'affiner le niveau de vérification des types :

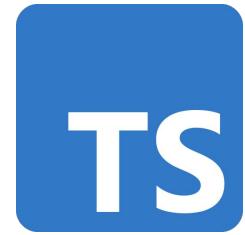
- ◆ noImplicitAny : permet de s'assurer que chaque variable a un type déclaré ou inféré
- ◆ noImplicitReturns : permet de s'assurer que toutes les branches d'une fonction retournent quelque chose (dans le cas où la signature déclare un type de retour)
- ◆ strictNullChecks : activer ou non les alertes typeScript indiquant la possibilité qu'une variable soit null (plus besoin d'utiliser l'opérateur !)
- ◆ strict : active ou non un jeu de paramètres préconfigurés

Il existe d'autres options qui visent à améliorer la qualité du code : warning si variables inutilisées (autres que dans le scope global) par exemple.

Pour connaître la liste des options disponibles, consulter la doc :

<https://www.typescriptlang.org/docs/handbook/compiler-options.html>

92



Nous allons désormais aborder les spécificités de la POO en TypeScript :

- ◆ Visibilité
- ◆ Constructeur rapide
- ◆ Héritage
- ◆ Getters & Setters
- ◆ Méthodes et champs statiques
- ◆ Classes abstraites
- ◆ Pattern singleton
- ◆ Interfaces

93

## Visibilité

Chaque attribut ou méthode d'une classe est créé selon une visibilité.

Les trois valeurs possibles sont :

- ◆ public
- ◆ private
- ◆ protected

- public signifie que l'attribut / la méthode est accessible en dehors de la classe
- private signifie que seul un accès interne est autorisé (on dit qu'on "encapsule" le champ ou la méthode)
- protected signifie que l'attribut / la méthode n'est pas accessible directement en dehors de la classe, sauf pour les classes filles.

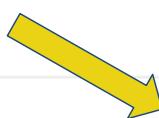
Nb : La notion de champ privé est au stade de la proposition expérimentale du standard EcmaScript, et devrait apparaître dans une prochaine version ES (utilisation du # devant le nom d'un champ ou d'une méthode).

94

# Constructeur rapide

Pour gagner du temps lors de l'écriture d'une classe, TypeScript nous propose un "constructeur rapide" qui crée et initialise les champs de nos instances automatiquement :

```
class MaClasseMetier {  
    private id : number;  
    public myField: string;  
  
    constructor(id: number, myField: string) {  
        this.id = id;  
        this.myField = myField;  
    }  
}
```



```
class MaClasseMetier {  
    constructor(private id: number,  
              public myField: string) { }  
}
```

95

## Lecture seule

Un champ d'une classe peut être en **readonly**.

Il n'est pas possible d'avoir un champ d'une instance déclaré comme une constante, c'est donc une alternative. A retenir : on utilise const pour une variable, readonly pour un attribut de classe.

Un champ readonly doit être affecté dans un constructeur. Il peut l'être à plusieurs reprises, mais uniquement dans le constructeur.

```
class MaClasseMetier {  
    private readonly myField: string;  
  
    constructor(private id: number) {  
        this.myField = "test";  
        //....  
        this.myField = "une autre valeur";  
    }  
  
    uneMethode() {  
        this.myField = "test";  
    }  
}
```

96

TypeScript apporte peu de spécificités au niveau de l'héritage.

Une classe fille qui ne définit pas de constructeur hérite de son constructeur parent :

```
class MaClasseMetier {
    constructor() {
        console.log("Constructeur parent");
    }
}

class MaClasseFille extends MaClasseMetier {}

new MaClasseFille(); //console log parent affiché
```

Il est obligatoire d'appeler le constructeur parent si l'on définit le constructeur enfant :

```
53  class MaClasseFille extends MaClasseMetier {
54  |     constructor() {
✖ index.ts 1 of 1 problem
      Constructors for derived classes must contain a 'super' call. ts(2377)
}
55  | }
56 }
```

97

## Getters & Setters

TypeScript propose une syntaxe particulière pour les getters / setters, qui masque en quelque sorte l'encapsulation :

```
class MaClasseMetier {
    constructor(private _id: string) { }

    get id() {
        return this._id;
    }

    set id(newId: string) {
        this._id = newId;
    }
}

new MaClasseMetier("abcd124345").id = "un autre id";
```

Ajouter un underscore devant le nom du champ (c'est une convention)

On manipule id comme s'il était public !

98

# Méthodes et champs statiques

Comme en POO classique (existe aussi en ES6), il est possible de définir un attribut ou une méthode statique.

Cela signifie que cet attribut / méthode n'est pas directement rattachée à l'instance mais à la classe.

Il n'existe pas de constructeur statique comme en Java, on peut passer par une fonction d'initialisation :

```
class MaClasseMetier {  
    private static COUNTER : number;  
  
    constructor() {  
        MaClasseMetier.COUNTER++;  
    }  
  
    static initialize() {  
        MaClasseMetier.COUNTER = 0;  
    }  
  
    MaClasseMetier.initialize();
```

99



Nicolas Amini-Lamy

## Classes abstraites

Une méthode d'une classe peut être abstraite, c'est à dire non implémentée.

Toute classe possédant une ou plusieurs méthodes abstraites et elle-même abstraite et ne peut être instanciée.

```
46 abstract class Animal {  
47  
48     constructor() {}  
49  
50     abstract manger() : void;  
51 }  
52  
53 class Chat extends Animal {  
54     manger() {  
55         console.log("Je mange des souris");  
56     }  
57 }  
58  
59 new Animal();
```

index.ts 1 of 1 problem

Cannot create an instance of an abstract class. ts(2511)

```
60 let unChat : Animal;   
61 unChat = new Chat();
```

Peut aider à la programmation générique !



Nicolas Amini-Lamy

100

Pour implémenter le pattern singleton (ie. une classe ne peut être instanciée qu'une seule fois), on peut utiliser l'astuce suivante :

```
class MonService {
    private static singleton: MonService;

    private constructor() { }

    static getInstance(): MonService {
        if (!this.singleton) {
            this.singleton = new MonService();
        }
        return this.singleton;
    }

    //impossible d'instancier la classe
    let uneInstance = new MonService();

    //mais on peut récupérer son singleton
    let unSingleton = MonService.getInstance();
}
```

101

## Interfaces

TypeScript apporte la notion d'interface qui n'existe pas en JavaScript.

Le rôle d'une interface est de décrire un comportement applicable à des objets

C'est un outil complémentaire à la notion de classe, mais que l'on utilise à des fins différentes.

En effet, une interface est comparable à une classe complètement abstraite, c'est à dire sans implémentation.

Lorsque l'on utilise les classes et l'héritage, on a un lien de filiation entre la classe fille et la classe mère. Ce lien doit avoir un sens sémantique.

Par exemple, un loup est un animal. On peut donc écrire une classe Loup qui étend une classe Animal.

On va utiliser les interfaces lorsque ce lien de filiation n'a pas de sens sémantique.

Par exemple, un vélo et un loup peuvent tous deux marcher. Pour autant, cela aurait-il du sens de créer un lien de filiation entre loups et vélos ?

102

# Interfaces

Une interface contient uniquement des déclarations de propriétés ou de méthodes publiques, on ne retrouve aucune valeur ou implémentation.

Une classe peut **implémenter** une interface :

```
interface Animal {  
    race: string;  
    parler(): void;  
}  
  
class Chat implements Animal {  
    race: string;  
  
    constructor() {  
        this.race = "Chat";  
    }  
  
    parler() {  
        console.log("Miaou");  
    }  
}
```

103

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

# Interfaces

Tout ce qui est défini dans une interface doit être public dans la classe qui l'implémente. Il n'y a pas de notion de visibilité dans une interface.

Il est possible de définir un champ readonly dans une interface.

Une interface peut étendre une autre interface :

```
interface Animal {  
    race: string;  
    parler(): void;  
}  
  
interface Vertebre extends Animal {  
    nombreDeVertebres: number;  
}
```

104

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

# Interfaces

Une interface peut définir un champ optionnel à l'aide de l'opérateur ? :

```
interface Animal {  
    race?: string;  
    parler() : void;  
}  
  
class Chat implements Animal {  
    parler() {  
        console.log("Miaou");  
    }  
}
```

Cet opérateur peut également être utilisé dans la signature d'une fonction. C'est une alternative si l'on ne souhaite pas définir de valeur par défaut.

Note : l'opérateur instanceof ne fonctionne pas pour une interface.

105



Nicolas Amini-Lamy

# Interfaces

À votre avis, quel sera le code JavaScript généré ?

```
interface Animal {  
    race?: string;  
    parler() : void;  
}
```

...absolument aucun code n'est généré =)

Une interface est simplement une aide au développement !

106



Nicolas Amini-Lamy



107

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

## Framework

---

Qu'est-ce qu'un framework ?

108

**WEBFORCE**  
BE THE CHANGE  
Nicolas Amini-Lamy

# Introduction

---

- Un cadre de travail
- Une méthodologie permettant d'organiser son développement
  - Guide l'architecture d'une application
- Des conventions, outils intégrés
- Des fonctions courantes déjà développées
  - Gestion du routage d'une application
  - Déploiement sur un environnement de production
  - Gestion des formulaires



Symfony



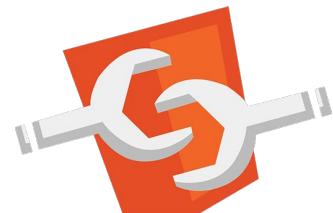
109

## WebComponents

---

Angular basé sur la philosophie des « WebComponents »

- Standardisation en cours par le W3C
- Créer des balises HTML personnalisées , réutilisables et ensapsulées
- Exemple : affichage d'un produit
  - La « structure » d'affichage est indépendante des données
- Compatibilité pas encore parfaite avec l'ensemble des navigateurs
  - Utilisation de « polyfills »



110



- Standard impliquant 4 technologies :

- Custom Elements (création d'une balise personnalisée)
- HTML Templates (génération d'un squelette HTML dynamique au runtime par JS)
- Shadow DOM (encapsulation JS / CSS)
- HTML Imports (permet le packaging / l'import des fichiers d'un composant)

Voir <https://developer.mozilla.org/en-US/docs/Web/API/CustomElementRegistry>

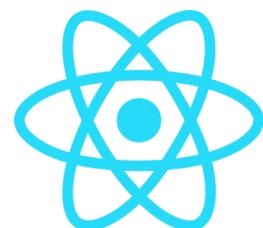


111

WEBFORCE  
BE THE CHANGE  
Nicolas Amini-Lamy

## WebComponents - Frameworks

- Programmation de nos IHM en « composants web » : principe fondamental des 3 grands frameworks concurrents du moment



- NB : Très bon article sur le choix d'un framework parmi ces 3 :  
<https://blog.dyma.fr/quel-framework-choisir-en-2020-angular-vue-js-ou-react/>

112

WEBFORCE  
BE THE CHANGE  
Nicolas Amini-Lamy

# Quelques mots sur Angular

---



- Simplifier le développement d'un front-end
- Historiquement nommé Angular.JS
- Structurer le développement
- Projet initialisé avec plusieurs outils (Webpack, Npm entre-autres)
- Interface en ligne de commande (CLI)
  - ng new
  - ng generate
  - ng serve

113

**WEBFORCE**  
BE THE CHANGE  
*Nicolas Amini-Lamy*

C'est parti !

---

Initialisation d'un premier projet Angular



<https://cli.angular.io/>

114

**WEBFORCE**  
BE THE CHANGE  
*Nicolas Amini-Lamy*

# Points d'attention

- Fichier index.html
  - Composant « app-root » : nous sommes bien sur une SPA
- Tests e2e
  - Commande « ng e2e »
- Tests unitaires
  - Commande « ng test »
- Fichier package.json

115

## Compatibilité avec les navigateurs

- Depuis Angular 8 ; notion de « Differential loading »
- Objectifs :
  - Supporter les « anciens navigateurs » tels IE11
  - Ne pas pénaliser les utilisateurs des navigateurs récents
- ES6 (ES2015) n'est pas compatible avec tous les navigateurs
- Angular propose de builder les polyfills nécessaires uniquement sur les navigateurs qui en ont besoin

116

# Compatibilité avec les navigateurs

## Exemple :

```
PS C:\Users\comme\Documents\repo\formationAngular\paper-dashboard-angular-master> ng serve
Your global Angular CLI version (8.3.20) is greater than your local
version (8.0.4). The local Angular CLI version is used.

To disable this warning use "ng config -g cli.warnings.versionMismatch false".

Date: 2019-12-16T19:51:11.243Z
Hash: b1e0f62ac48ab574f47b
Time: 27696ms
chunk {layouts-admin-layout-admin-layout-module} layouts-admin-layout-admin-layout-module.js, layouts-admin-layout-
yout-admin-layout-module) 1.06 MB [rendered]
chunk {main} main.js, main.js.map (main) 41.5 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 274 kB [initial] [rendered]
chunk {polyfills-es5} polyfills-es5.js, polyfills-es5.js.map (polyfills-es5) 462 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 8.82 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 1.41 MB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 4.94 MB [initial] [rendered]
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
i [wdm]: Compiled successfully.
```

117



## Comment cela fonctionne ?

### Ouvrons le fichier index.html :

```
<body>
  <app-root>
    <div class="loader-container">
      <div class="loader"></div>
    </div>
  </app-root>
  <script src="runtime-es2015.16e3380b2e28525e12cd.js" type="module"></script>
  <script src="polyfills-es2015.c0b536e90b442f3a04ac.js" type="module"></script>
  <script src="runtime-es5.8a55f9c4693719a855f8.js" nomodule></script>
  <script src="polyfills-es5.34612e117dfc68ab5042.js" nomodule></script>
  <script src="main-es2015.54da2fd8d7db24978728.js" type="module"></script>
  <script src="main-es5.0a62eaed968ec6eee6aa.js" nomodule></script>
</body>
```

118



## Comment cela fonctionne ?

- Ouvrons le fichier index.html :

```
<body>
  <app-root>
    <div class="loader-container">
      <div class="loader"></div>
    </div>
  </app-root>
  <script src="runtime-es2015.16e3380b2e28525e12cd.js" type="module"></script>
  <script src="polyfills-es2015.c0b536e90b442f3a04ac.js" type="module"></script>
  <script src="runtime-es5.8a55f9c4693719a855f8.js" nomodule></script>
  <script src="polyfills-es5.34612e117dfc68ab5042.js" nomodule></script>
  <script src="main-es2015.54da2fd8d7db24978728.js" type="module"></script>
  <script src="main-es5.0a62eaed968ec6eee6aa.js" nomodule></script>
</body>
```

Norme ECMA Script

119

## Comment cela fonctionne ?

- Ouvrons le fichier index.html :

```
<body>
  <app-root>
    <div class="loader-container">
      <div class="loader"></div>
    </div>
  </app-root>
  <script src="runtime-es2015.16e3380b2e28525e12cd.js" type="module"></script>
  <script src="polyfills-es2015.c0b536e90b442f3a04ac.js" type="module"></script>
  <script src="runtime-es5.8a55f9c4693719a855f8.js" nomodule></script>
  <script src="polyfills-es5.34612e117dfc68ab5042.js" nomodule></script>
  <script src="main-es2015.54da2fd8d7db24978728.js" type="module"></script>
  <script src="main-es5.0a62eaed968ec6eee6aa.js" nomodule></script>
</body>
```

120

- Le type de script « module » est lié aux notions d'import / export de modules JavaScript
  - Pris en charge par les navigateurs récents implémentant ES6
- Les « anciens » navigateurs ne reconnaissent pas ce type et ignorent donc la balise
- A l'inverse, ils vont charger les scripts flagués « nomodule »
- Les navigateurs récents comprendront qu'il ne faut pas charger ce module de rétrocompatibilité

121

## Intérêts

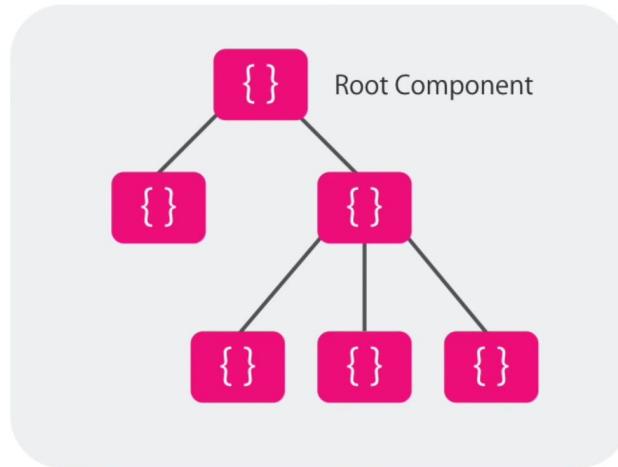
---

- Pouvoir développer à l'aide des dernières évolutions JS
- Cibler tout de même des navigateurs qui ne les comprennent pas
- Ne pas pénaliser les clients de navigateurs récents
- Pour forcer une version ES5, modifiez le fichier tsconfig (target = es5)
  - Permet d'utiliser « ng serve » même avec un navigateur récent, sous es5
- Pour plus d'informations :
  - <https://angular.io/guide/deployment#differential-loading>
  - <https://angular.io/guide/browser-support>
  - <https://auth0.com/blog/angular-8-differential-loading/>

122

# Composants

- Une balise HTML personnalisée est un “composant”
- Un composant en contient souvent d’autres
- Un composant fils ne peut modifier son composant père !
  - Principe de l’encapsulation, comme l’héritage en POO



123

# Composants

- Créer un nouveau composant :
  - `ng generate component myComponent`



124

## Composants : le décorateur @Component

- Chaque classe associée à un composant utilise le décorateur @Component
- Ce décorateur permet d'indiquer à Angular que l'on se situe dans la classe d'un composant.
  - On lui indique des paramètres de configuration :

The diagram shows a code snippet for the `AppComponent` class. It includes two annotations with arrows pointing to specific parts of the code:

- An arrow points to the `selector: 'app-root'` property, labeled "Nom de la balise HTML".
- An arrow points to the `templateUrl: './app.component.html'` and `styleUrls: ['./app.component.css']` properties, labeled "Template et feuille de style associée au composant".

```
3  @Component({  
4      selector: 'app-root', ← Nom de la balise HTML  
5      templateUrl: './app.component.html', ← Template et feuille de style associée au composant  
6      styleUrls: ['./app.component.css']  
7  })
```

125

## Composants

- Chaque composant a son propre cycle de vie
- Angular s'occupe du rendering :
  - Création du composant
  - Affichage
  - Création et affichage des composants fils
  - Destruction (ie = retrait du DOM)
- Un composant peut effectuer un traitement particulier à chaque étape de son cycle de vie
- Implémentation d'interfaces spécifiques
- <https://angular.io/guide/lifecycle-hooks>

126

# Composants

ngOnChanges()	Respond when Angular (re)sets data-bound input properties. The method receives a <a href="#">SimpleChanges</a> object of current and previous property values. Called before ngOnInit() and whenever one or more data-bound input properties change.
ngOnInit()	Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties. Called <i>once</i> , after the <i>first</i> ngOnChanges().
ngDoCheck()	Detect and act upon changes that Angular can't or won't detect on its own. Called during every change detection run, immediately after ngOnChanges() and ngOnInit().
<a href="#">ngAfterContentInit()</a>	Respond after Angular projects external content into the component's view / the view that a directive is in. Called <i>once</i> after the first ngDoCheck().
ngAfterContentChecked()	Respond after Angular checks the content projected into the directive/component. Called after the <a href="#">ngAfterContentInit()</a> and every subsequent ngDoCheck().
<a href="#">ngAfterViewInit()</a>	Respond after Angular initializes the component's views and child views / the view that a directive is in. Called <i>once</i> after the first ngAfterContentChecked().
ngAfterViewChecked()	Respond after Angular checks the component's views and child views / the view that a directive is in. Called after the <a href="#">ngAfterViewInit()</a> and every subsequent ngAfterContentChecked().
ngOnDestroy()	Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called <i>just before</i> Angular destroys the directive/component.

Nicolas Amini-Lamy

# Composants

Invoqué lorsqu'un composant est instancié (new)

Invoqué lorsqu'un composant a été initialisé.  
Appelé une seule fois après le premier ngOnChanges

Invoqué avant qu'un composant soit détruit,  
pour permettre par exemple la libération de ressources

Invoqué à chaque mise à jour d'un paramètre d'entrée du composant

Cycle de vie spécifique pour la détection de changements

