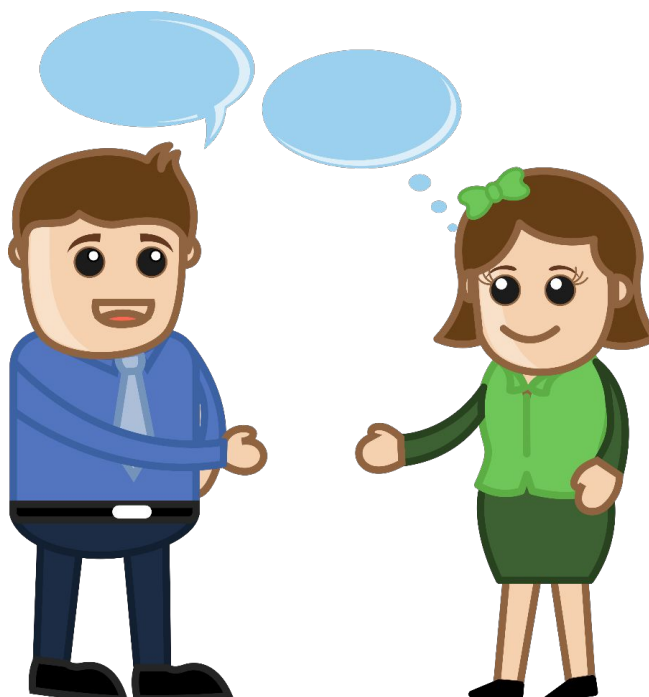


Nicolas Amini-Lamy

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



3

WEBFORCE
BE THE CHANGE 
Nicolas Amini-Lamy

Programme de la semaine



Point parcours & architectures web



Compléments JavaScript



Présentation du langage TypeScript



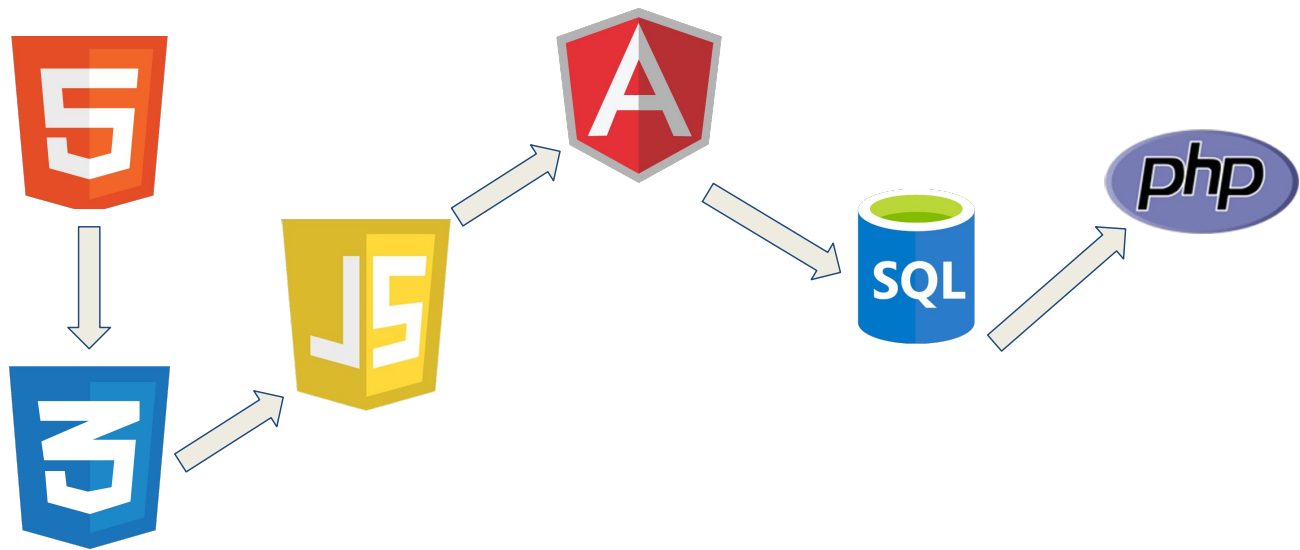
Programmation asynchrone



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.

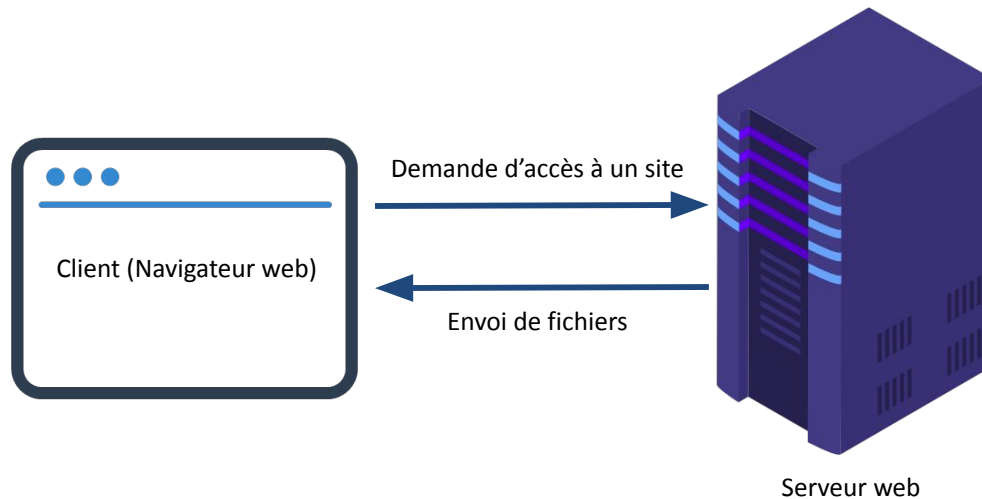
Une ressource statique est une ressource dont le contenu ne changera pas en fonction de la personne qui les récupère. Ce sont des fichiers destinés à être hébergés sur un serveur web.

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.

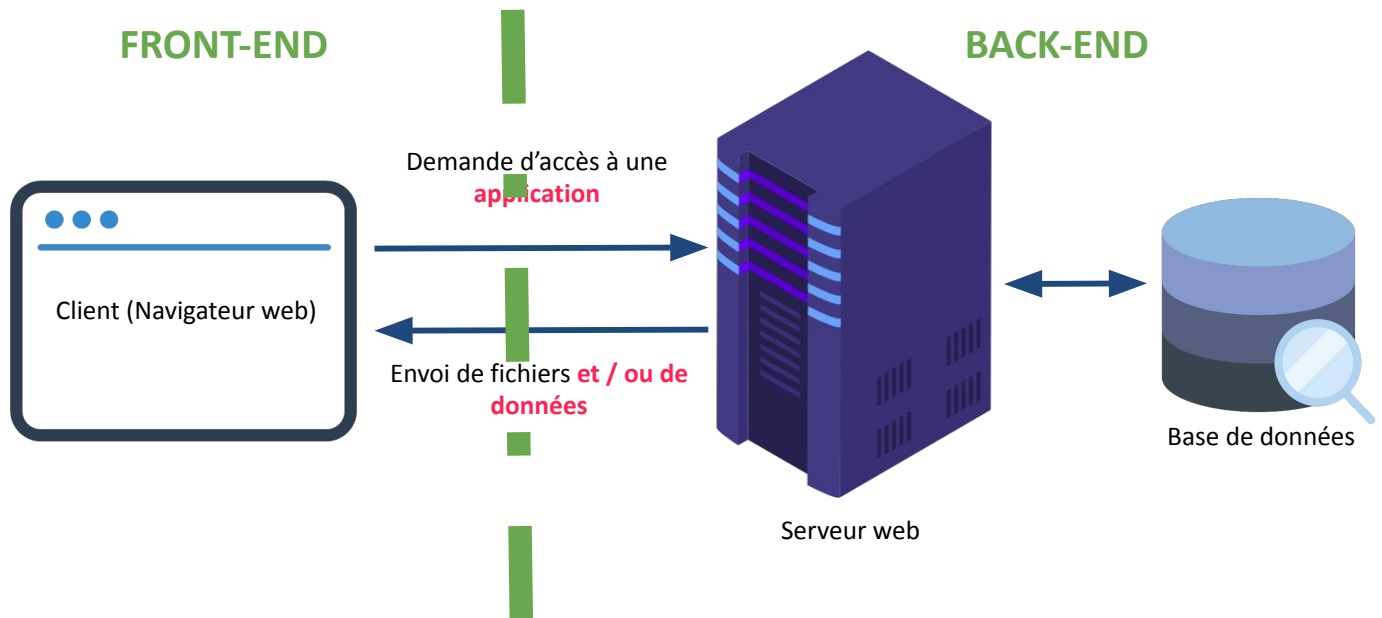
On est dans une logique **informative**.

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.

8



9

Point parcours

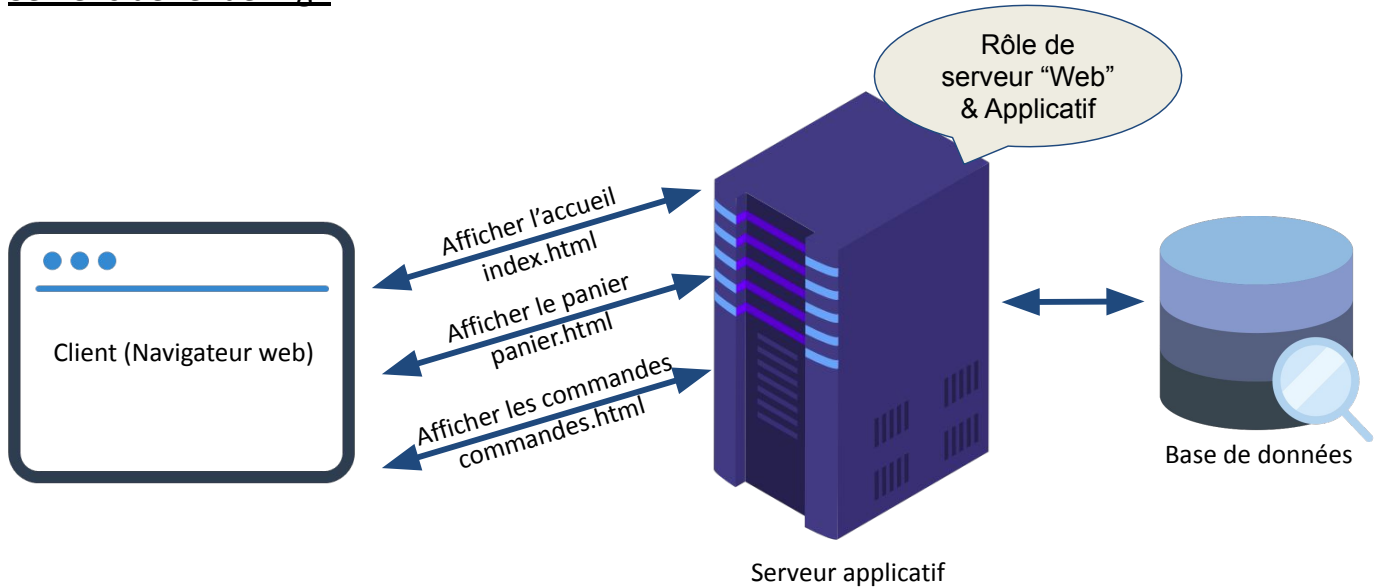
Dans le cas d'une application web, chaque utilisateur pourra disposer de sa propre interface.

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**.

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 fichiers 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

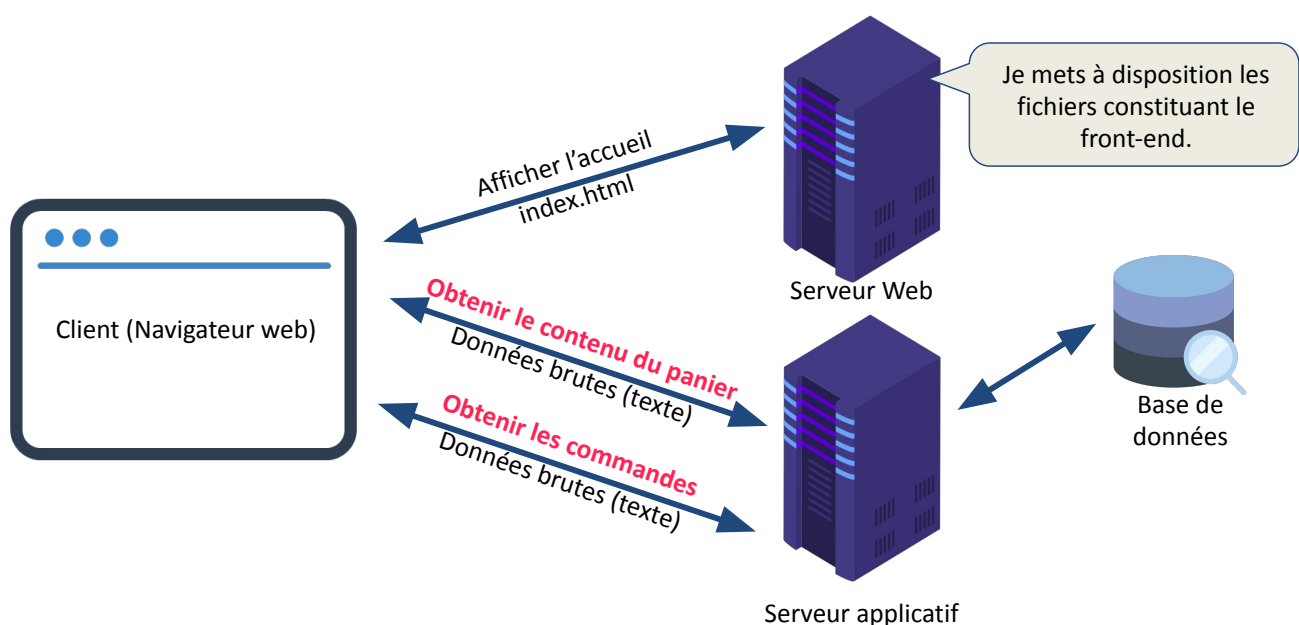
Dans le cas du server side rendering, notre application suit l'architecture "**multi-page application**" (MPA).

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

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

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

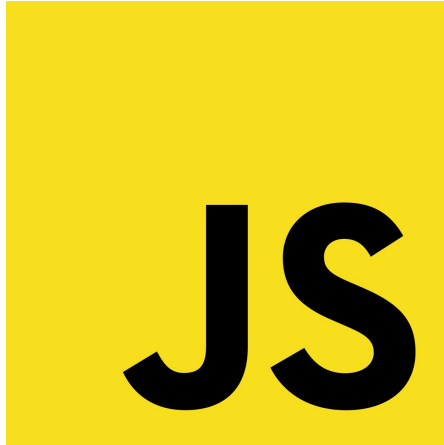
Lorsque l'on développe une SPA, on a une isolation forte entre le front et le back.

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

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

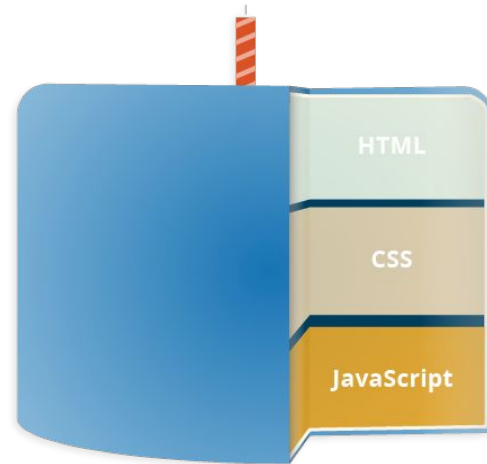
Introduction - JavaScript dans un navigateur

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).






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

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.

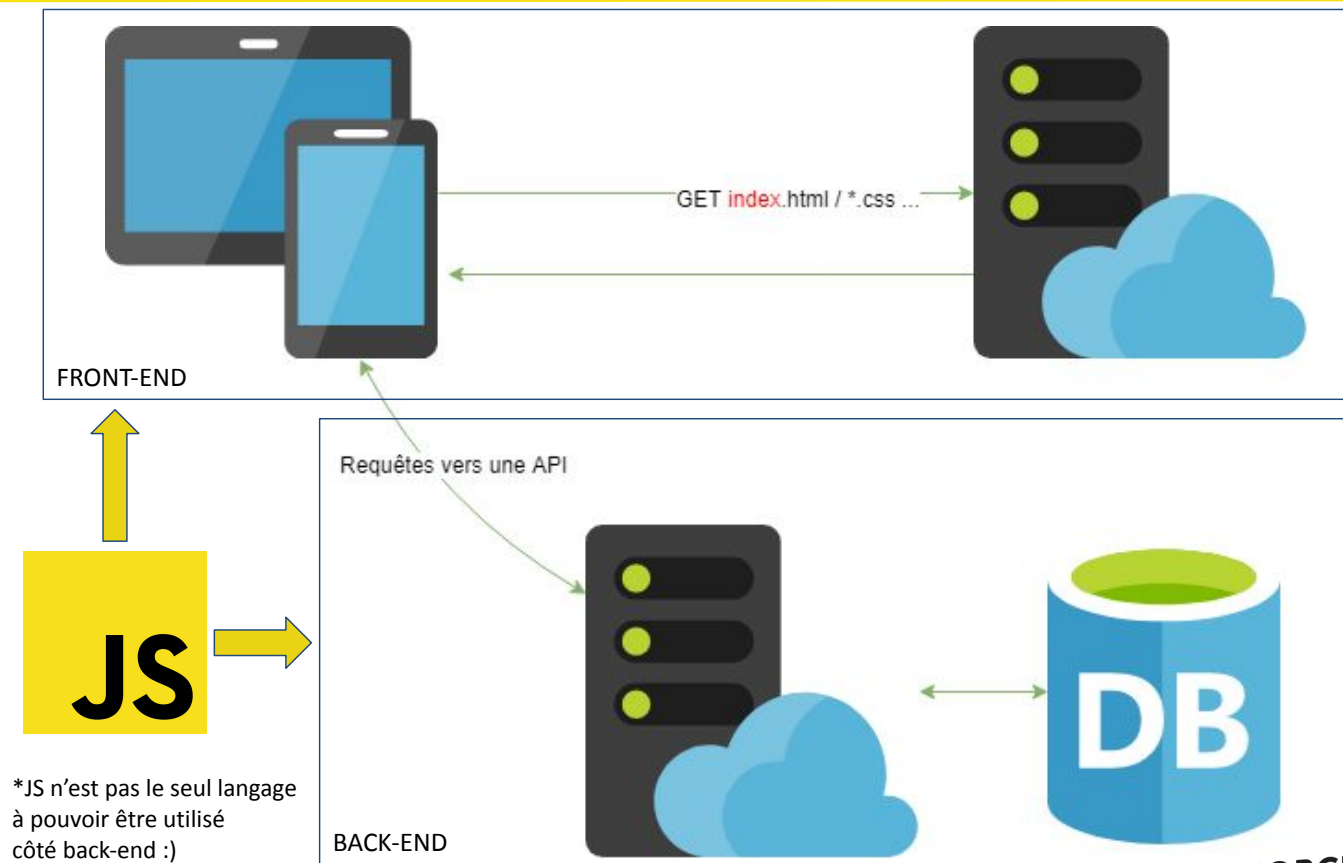
Pour des réalisations plus complexes, comme un back-end applicatif.



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

23

Introduction - Exemple d'architecture (SPA)



24

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>`

Les nouveautés ES6



{ ES6 JS }

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

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

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 >

Reset

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

29

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 >

Reset

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

30

Nouveautés ES6 - Template Strings

On peut même utiliser une ternaire et une template string dans une template string (évitons 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.

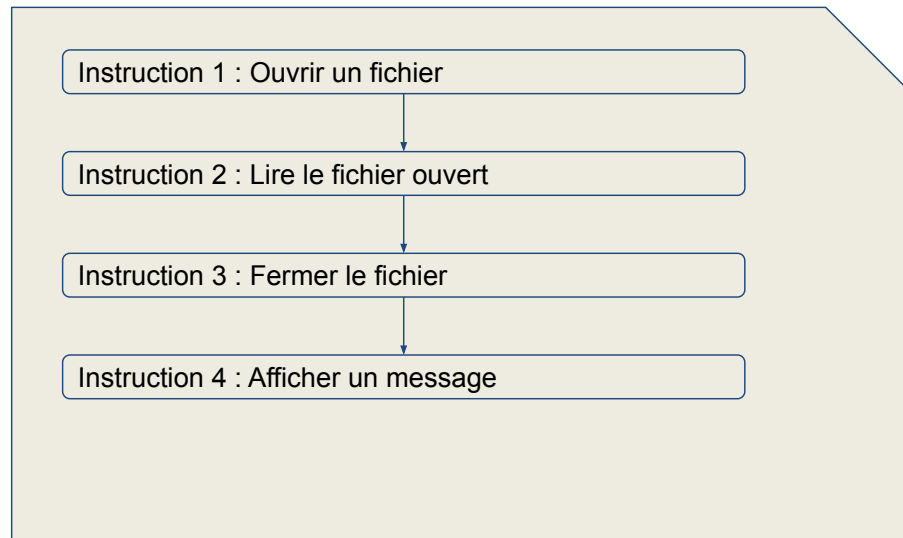


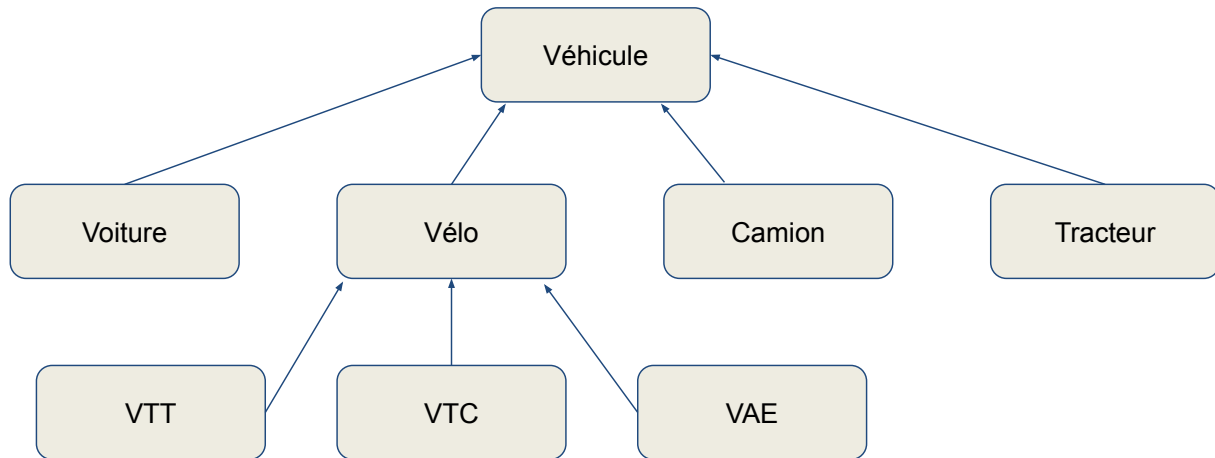
Illustration d'un programme impératif

33

Paradigme déclaratif

```
58 <div *ngIf="!isUserInPatientDetailsView" class="containerFiltres d-flex al
59 <div class="btn-group filteringBlock" role="group">
60 <button #displayUnlockedPatientsButton type="button" class="btn btn-se
61 <button #displayLockedPatientsButton type="button" class="btn btn-seco
62 <button #displayAllPatientsButton type="button" class="btn btn-seconda
63 </div>
64
65 <div class="input-group pl-2 filteringBlock">
66 <div class="input-group-prepend">
67 <span class="input-group-text" id="inputRecherche"><i class="fa fa-s
68 </div>
69 <input type="text"
70 class="form-control"
71 placeholder="Rechercher..."
72 aria-label="Rechercher..."
73 aria-describedby="inputRecherche"/>
74 </div>
75 </div>
76 <div *ngIf="!isUserInPatientDetailsView" [ngClass]="{disabledContent : isP
77 <div class="card patientsList">
78 <ngx-datatable
79 class='material'
80 [reorderable]="reorderable"
81 [rowHeight]="getRowHeight()"
82 [headerHeight]="50"
```

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

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.

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.

```
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

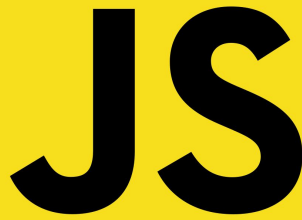
La notion de Classe existe en JavaScript via ES6.

Il s'agit d'un syntactic 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 >

> "Démonstration des closures"

Reset

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 with the following code:

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

Two yellow arrows point from the code to the console output:

- An arrow points from the `this` parameter in the `console.log` statement to the console output.
- An arrow points from the `arguments` parameter in the `console.log` statement to the console output.

The console output is:

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

Below the code editor, there are two buttons: "Run >" and "Reset".

Labels below the arrows:

- "Contexte" de la fonction
- Paramètres de la fonction

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

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



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 bibliothèques 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.

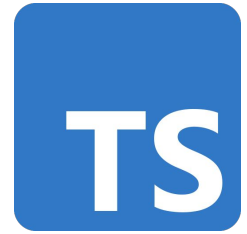
Une solution plus complète consiste à développer en TypeScript.

52

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).

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

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

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 prémunit 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).

Npm



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)

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.

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 :)

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.

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

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"
}
```

66

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).

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.



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"  
};  
etudiant.nomComplet = "Pierre Dupont";
```

any
Property 'nomComplet' does not exist on type '{ prenom: string; nom: string; }'. ts(2339)
Peek Problem (Alt+F8) No quick fixes available

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

71

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;  
}  
etudiant = {  
  prenom : "Pierre",  
  nom : "Dupont"  
};
```

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

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

72

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);
}
```

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 !
```

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

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";
```

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
```


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);  
    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 transtypage / 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();
```

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 :)

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"

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 paramétrer 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>