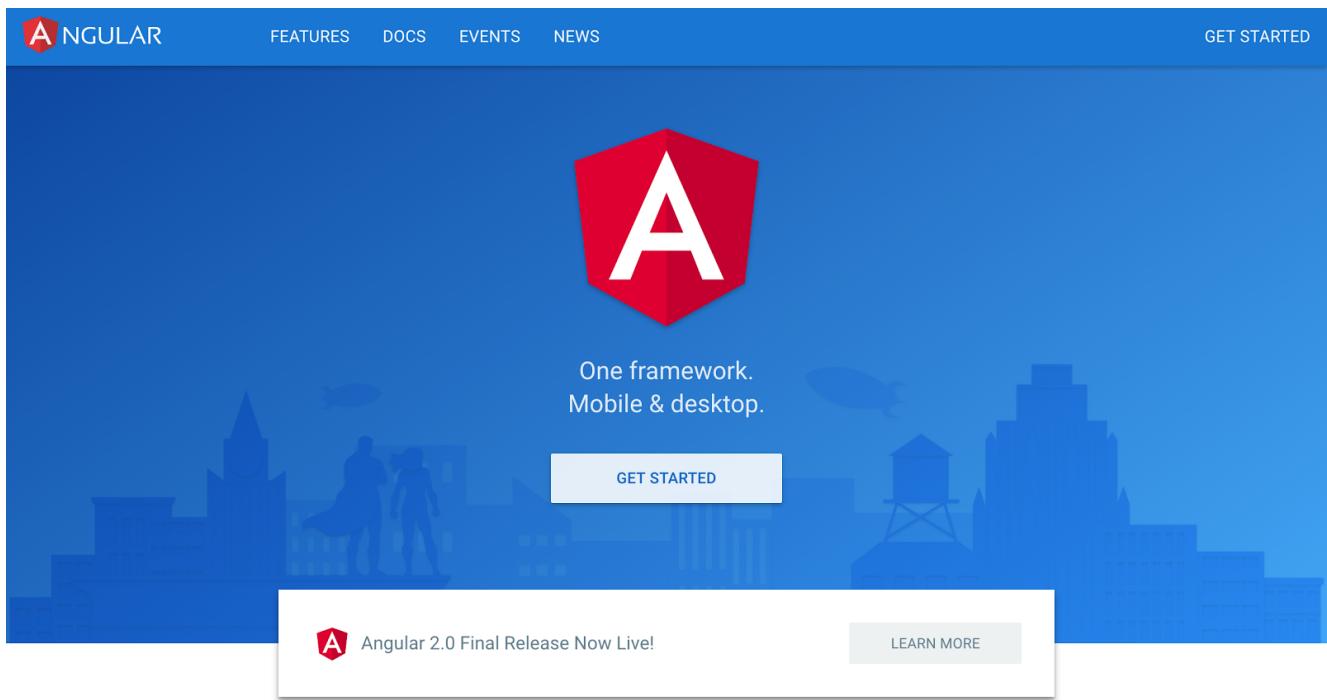


Formation AngularJS 2.



Migration et développement.

AngularJS, devenu la référence des infrastructures JavaScript côté client, propose une refonte du Framework en s'appuyant sur les nouveaux standards du Web. Cette formation vous permettra d'en maîtriser les concepts et d'améliorer les performances de vos applications notamment mobiles.

Objectifs pédagogiques

- Organiser, modulariser et tester les développements JavaScript.
- Maîtriser les fondamentaux du Framework AngularJS 2.
- Créer rapidement des applications Web complexes.
- Savoir intégrer les tests unitaires au développement.
- Connaître les bonnes pratiques de développement et de mise en production.

Présentation

Participants. (Tour de table)

Architectes, développeurs et chefs de projets Web.

Prérequis.

Bonnes connaissances des technologies du Web et des outils modernes de développement Front-End. Connaissances de JavaScript.

Récapitulatif matinal.

Chaque journée commence par un exercice collectif de restitution des concepts abordés la veille. L'objectif étant de renforcer (répétition) les connaissances acquises et les utiliser comme socle pour la journée à venir.

Concertation personnelle.

Votre formateur passera vous assister individuellement aussi souvent que possible.

N'hésitez pas à le solliciter pendant la journée de formation ou pour revenir sur un point particulier en fin de journée.

Votre Formateur.

Renaud Dubuis : [linkedin](#)

Version numérique

Pour votre confort de lecture et de manipulation ce support vous est également distribué en version numérique. (**voir ./documentation**)

Références à l'ouvrage et autres références

La formation est illustrée par la projection d'une **version numérique** (pdf) de votre support et l'utilisation d'autres ressources pertinentes (site internet, démonstration).

Un livre complet (**voir [./documentation](#)**) au format PDF édité par **rangle.io** vous est également délivré sous licence **Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)** [LICENSE](#)

Autres références

- [angular.io](#)
- [learnangular2](#)
- [angular2](#)
- [angular-2-training-book.rangle.io](#)
- [node.js](#)
- [npmjs](#)

Galaxie ng2

- [cli.angular.io](#)
- [universal.angular.io](#)
- [mobile.angular.io](#)
- [augury.angular.io](#)

Introduction

AngularJS 2

[Angular 2 ou ng2](#) se définit comme un [Application Framework Mobile & desktop](#).

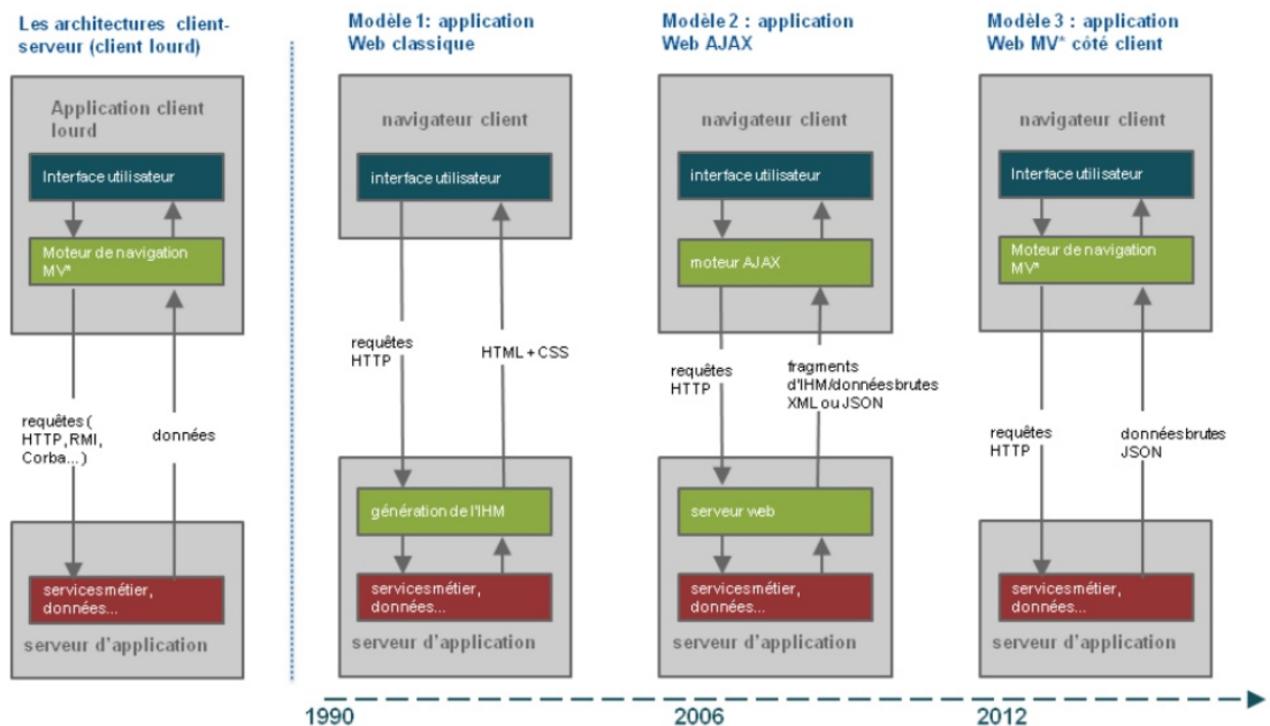
En [Final Release](#) depuis septembre 2016, le framework est bâti sur l'idée d'anticipation des besoins en développement.

- [Platform Agnostic](#) : découpler l'outil de développement d'une plateforme cible spécifique.
- [Faster Rendering](#) : optimiser les cycles de rendus.
- [Better Developping Experience](#) : Tirer parti des nouveaux outils de développement pour simplifier et améliorer la production du code.

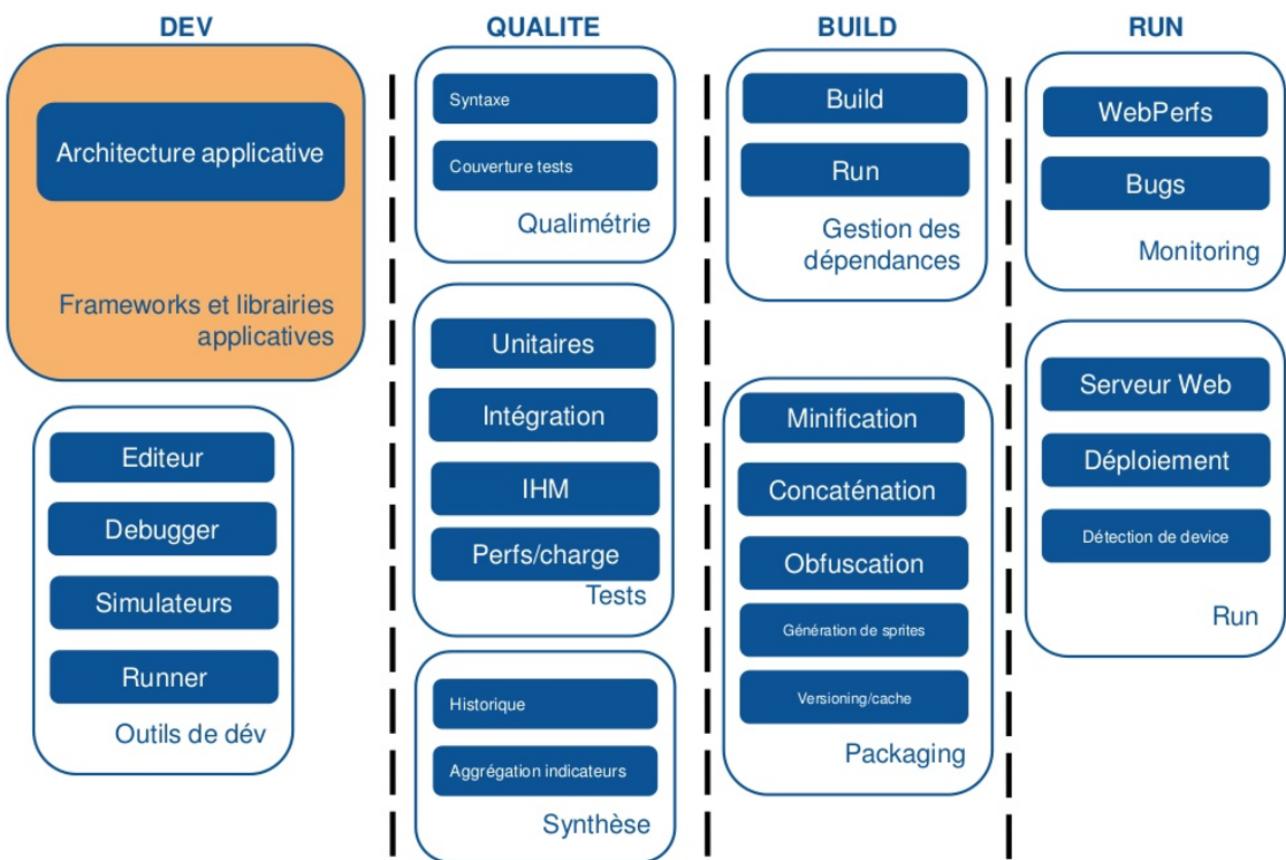
En tant que **framework JavaScript Angular** satisfait au problématiques de développement en utilisant [l'écosystème industrialisé moderne](#)

L'actuel [“marché” des Frameworks UI/MVC JavaScript est très riche](#).

Evolution des développements Front End.

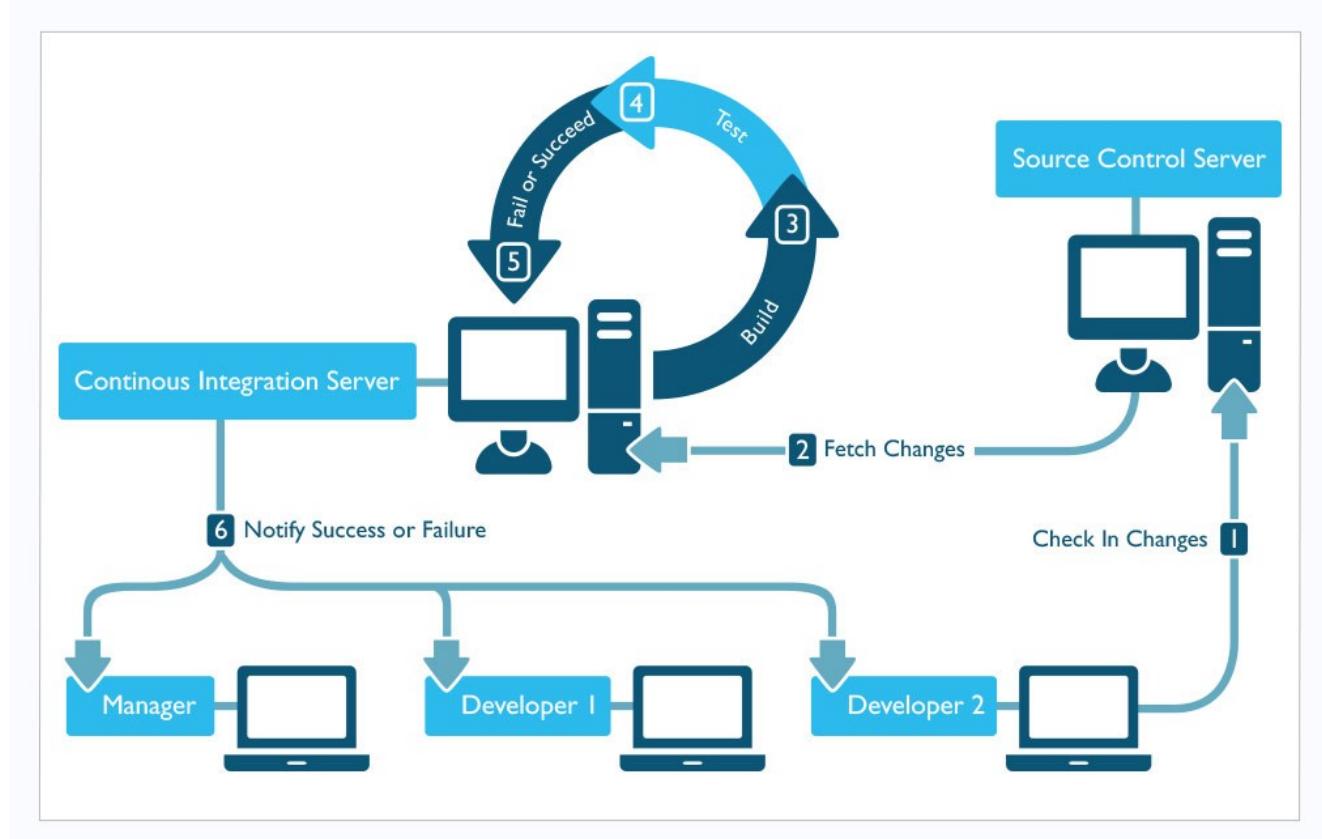


Un grand nombre d'utilitaires permettent d'automatiser les tâches découlant de l'évolution de ces pratiques de développement.



Industrialisation de la production.

Ces différents outils ont rendu possible l'industrialisation des développements par la séparation, la simplification et l'automatisation des différentes tâches.



Outils indispensables pour le développeur Front End.

Il est possible de classer les outils selon trois catégories :

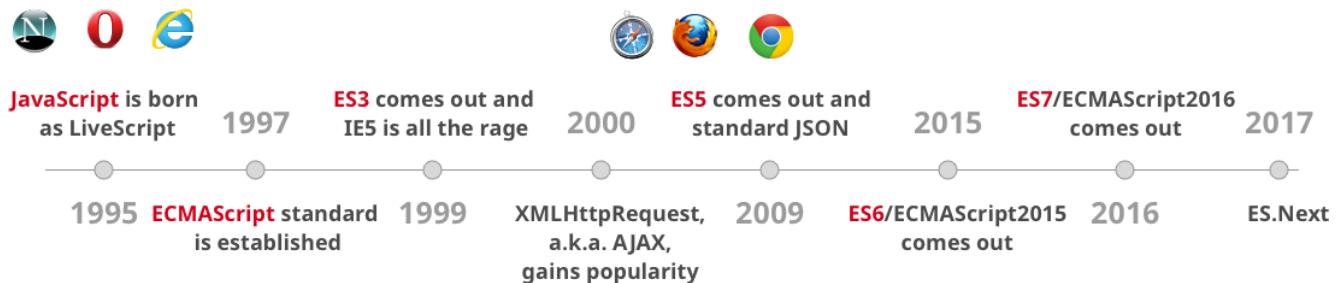
- **Logique** Dépendances tiers , tel que les différents frameworks.
- **Technique** ayant un impact sur la plate-forme de développement pour l'automatisation et le contrôle des tâches par exemple.
- **Productivité** toute autre solution non indispensable permettant d'accélérer le développement, allant du simple plugin à une documentation efficace.

Logique : le développeur front-end maîtrise au moins un [framework CSS](#) dont [bootstrap](#) demeure le choix par défaut.

Technique : en dehors du choix de l'IDE de nombreux utilitaires ayant pour [socle commun NodeJS](#) fluidifient le développement.

Logique : Les points d'entrée vers la documentation sont une des clés de la productivité on citera [http.awesom.re](http://awesom.re) et <http://devdocs.io>.

Prendre en considération l'évolution du langage



Par anticipation angular (et les autres frameworks) se rapprochent de la future pattern de développement qu'apporteront les [Web Components](#)

A propos des Web Components



Composants d'interface graphique réutilisables, qui ont été créés en utilisant des technologies web libres.

Les [Composants Web](#) sont constitués de plusieurs technologies distinctes. Ils font partie du navigateur, et donc ils ne nécessitent pas de bibliothèques externes comme jQuery ou Dojo.

Un composant Web existant peut être utilisé sans l'écriture de code, en ajoutant simplement une déclaration d'importation à une page HTML. Les Composants Web utilisent les nouvelles capacités standards de navigateur, ou celles en cours de développement.

Les Composants Web sont constitués de ces quatre technologies (bien que chacun peut être utilisé séparément):

- [Custom Elements](#): pour créer et enregistrer de nouveaux éléments HTML et les faire reconnaître par le navigateur,
- [HTML Templates](#): squelettes pour des éléments HTML instanciables,
- [Shadow DOM](#): ce qui sera public ou privé dans vos éléments,
- [HTML Imports](#): pour packager ses composants (CSS, JavaScript, etc.)

Une démonstration minimaliste dans chrome

```
<body>
  <simple-increment data-step="5"></simple-increment>
  <script>
    class SimpleIncrement extends HTMLElement {
      constructor(args) {
        super();
      }
      //Cycle de vie
      createdCallback() {
        this.count = 0;
        this.step = this.dataset.step;
        this.max = 100;
        this.innerHTML = `<button><i>${this.count}</i> of ${this.max}</button>`;
      }

      increment(){
        return (this.count < this.max) ?(this.count += Number(this.step)):this.count;
      }

      attachedCallback() {
        this.querySelector('button').addEventListener('click', (evt) => evt.target
      }
    }
    document.registerElement('simple-increment', SimpleIncrement)
  </script>
</body>
```

En résumé, un **Web Component** est un fragment fonctionnel d'interface encapsulé dans :

Un modèle générique `HTMLElement`.

Un bloc logique `class`.

Un système de rendu `document.registerElement` ici le `DOM`.

Un cycle de vie exposé par le système de renku.



Sommaire

Table des matières.

- [1/ Configuration du poste de développement.](#)
 - [1.1/ Installation de Node.JS :](#)
 - [1.3/ Découverte de l'éditeur VS Code](#)
- [2/ Développement JavaScript : rappels.](#)
 - [2.1/ Bonnes pratiques ECMAScript 5.](#)
 - [2.2/ ES5 “use strict” et méthodes moins connues.](#)
 - [2.3/ API issues de la communauté JavaScript.](#)
 - [Programmation Réactive.](#)
- [3/ Angular2 : Présentation.](#)
 - [Angular-cli : l'outil intégré.](#)
 - [Angular avec TypeScript et ES6/ES2015.](#)
 - [Support courant pour ES6 : compilateurs, polyfills, navigateurs serveurs.](#)
 - [Environnement et outils pour le développeur.](#)
- [4/ ES6/2015 : Evolutions syntaxiques fondamentales.](#)
 - [Aperçu général des apports.](#)
 - [Constantes et variables de bloc. Assignation destructurée.](#)
 - [Fonction, paramètres par défaut, opérateurs “rest / spread”.](#)
 - [Chaînes de caractères : multiligne, template, formatage.](#)
 - [“Arrow Function” : portée lexicale. Usages.](#)
 - [Fonction itératrice : “iterator”.](#)
 - [Objet littéral : évolution.](#)
- [5/ POO, nouveautés pour la conception objet.](#)
 - [Modèles de classe et héritage. Méthodes statiques.](#)
 - [Objets natifs héritables.](#)
- [6/ Nouvelles API JavaScript avec ES6.](#)
 - [Promise : gestion des traitements asynchrones.](#)
 - [Object.API : revisiter les méthodes.](#)
 - [Outils indispensables. Babel, Traceur et Typescript.](#)
 - [Mise en oeuvre de TypeScript.](#)
 - [ES6 approche modulaire.](#)
 - [Système natif de gestion des modules.](#)
 - [“Modules Loaders” : SystemJS, “import/export”.](#)
 - [Asynchronous Module Definition ou CommonJS.](#)
 - [Gestion et résolution des dépendances.](#)
 - [Chargement dynamique.](#)
- [7/ Typescript en détail, configuration.](#)
 - [12.1/ Installation et mise en oeuvre.](#)
 - [12.2/ Option du compilateur.](#)

- [Apports Syntaxiques](#)
 - [Typage des variables.](#)
 - [Inférence de type.](#)
 - [Assertion de type.](#)
 - [Typage des fonction.](#)
 - [Patterns de programmation orientée objet.](#)
 - [Enrichissement des class.](#)
 - [Accesseur de propriétés.](#)
 - [Paramètres de propriétés.](#)
 - [Classe Abstraites.](#)
 - [Modularité.](#)
 - [TypeScript Best Practices.](#)
- [8/ Migrer d'AngularJS 1.x à AngularJS 2](#)
 - [8.1/ Comparaison et “topographie” des concepts.](#)
 - [Tableau comparatif : Angular 1 vs 2](#)
 - [8.2/ Préparer la migration. Structure d'une application AngularJS 2.](#)
 - [8.3/ Les modules AngularJS 2. “core” et principaux modules.](#)
 - [Principaux modules :](#)
 - [8.4/ Principe de l'injection de dépendance.](#)
 - [8.5/ Classification des directives: , Composant, Attribut, services.](#)
 - [8.6/ Les décorateurs : définition des hiérarchies.](#)
 - [Mécanisme](#)
 - [Comparaison des syntaxes](#)
- [9/ Injection de dépendances](#)
 - [9.1/ Utiliser les annotations et décorateurs.](#)
 - [Principaux décorateurs](#)
 - [Autres décorateurs](#)
 - [9.2/ Configuration de l'injecteur.](#)
 - [9.3/ Gestion des modules : bonnes pratiques.](#)
 - [Organisation des dépendances:](#)
 - [9.4/ Création de services injectables. Classification des services.](#)
 - [9.5/ “BootStrap” d'application.](#)
 - [9.6/ Organisation des modules.](#)
- [10/ Définition de composants](#)
 - [10.1/ Cycle de vie dans l'application.](#)
 - [Transmission de données.](#)
 - [Two-Way Data Binding](#)
 - [Accès à un composant enfant.](#)
 - [Cycle de vie.](#)
 - [Manipulation du DOM](#)
 - [10.2/ Syntaxe des templates : interpolation/expression, “Binding” et filtres.](#)

- [Binding de propriété.](#)
- [Binding sur attribut.](#)
- [Binding sur une Classe CSS.](#)
- [Binding sur l'attribut style.](#)
- [Binding événementiel.](#)
- [Utilisation des Pipe \(filtres\)](#)
- [Utilisation depuis le code.](#)
- [10.3/ Directives de transformation.](#)
 - [Directives Natives](#)
 - [Directives Personnalisées](#)
- [10.4/ Définition syntaxique, le symbole \(*\). Variables locales.](#)
- [Créer une directive de structure](#)
- [10.5/ Configuration de directives: selector, provider.](#)
- [10.6/ Directives et évènements utilisateur.](#)
- [11/ Gestion des formulaires, “Routing” et requête HTTP](#)
 - [11.1/ NgForm et FormControl et FormGroup.](#)
 - [Création d'un composant de formulaire.](#)
 - [Afficher / Masquer un message de validation](#)
 - [Soumission du formulaire](#)
 - [11.2/ Validation et gestion d'erreur personnalisée.](#)
 - [Règle de validation personnalisée](#)
 - [11.3/ “FormBuilder”, composants avancés de formulaire.](#)
 - [11.4/ Liaison de données via HTTP.](#)
 - [11.5/ Création de routes. Paramétrage et wildcard.](#)
 - [11.6/ Ciblage, “router-outlet” événements de routage.](#)
- [12/ Tests unitaires. Bonnes pratiques et outils.](#)
 - [12.1/ Configurer l'environnement de test.](#)
 - [12.2/ Ecrire les tests avec Jasmine. Couverture.](#)
 - [12.3/ Cas de test : pipe, composant, application.](#)
 - [Composant](#)
 - [Composant avec dépendance](#)
 - [Composant avec service asynchrone](#)
 - [Composant avec templateUrl](#)
 - [service](#)
 - [Pipe](#)
 - [12.4/ AngularJS2 “Coding guide Style”.](#)
 - [Evolution](#)



Configuration du poste de développement.

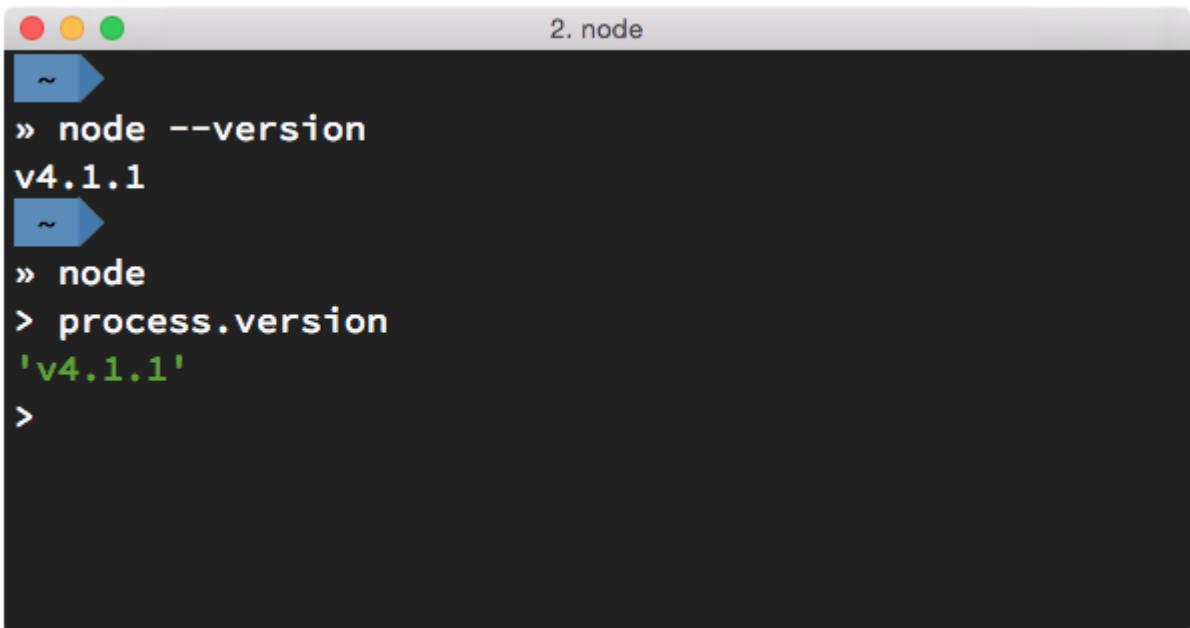
1/ Configuration du poste de développement.

1.1/ Installation de Node.JS :

Il est fortement recommandé d'utiliser un interpréteur de commandes (terminal ou shell). Les systèmes d'exploitation modernes en proposent un, y compris les versions récentes de Windows.

Si vous n'utilisez pas encore de terminal, voici une liste de recommandations non exhaustive pour vous aider :

- OS X : iTerm2, Terminal.app.
- Linux : GNOME Shell, Terminator.
- Windows : PowerShell, Console.



A screenshot of a macOS terminal window titled "2. node". The window has three colored window control buttons (red, yellow, green) at the top left. The title bar shows the number "2. node". The terminal window contains the following text:

```
~
```

```
» node --version
```

```
v4.1.1
```

```
~
```

```
» node
```

```
> process.version
```

```
'v4.1.1'
```

```
>
```

Aisance avec l'invite de commande windows.

cf. pratique

Il est utile de pouvoir ouvrir rapidement une invite de commande pointant directement sur le répertoire voulu.

Manipulation 1

```
MAJ + CLIQUE DROIT > Ouvrir une invite de commande au dossier
```

Manipulation 2

```
A l'aide de l'explorateur windows, se placer dans le dossier 'workshops'
```

```
Saisir 'cmd' + ENTRÉE dans la barre d'adresse
```

Préparer son environnement.

Installer Node n'est pas très compliqué. Il existe cependant plusieurs mécanismes d'installation.

Ces mécanismes vont du téléchargement d'un installateur à une compilation manuelle *via* un terminal.

Les installateurs, les binaires et les sources de Node sont disponibles sur le site officiel
<https://nodejs.org/download/>

Téléchargez l'installateur adapté.

Répertoire de travail

cf. pratique

Créer un répertoire nommé **workshops** sur le bureau.

```
workshops/
|
├── concepts-es5/
|
├── concepts-es6/
|
├── concepts-typescript/
|
├── simple-projects/
|
└── node/
    |
    └── application/
        ├── server/
        └── client/
```

Mais plus simplement utilisons NodeJS

Sauvegardez le fichier (**cf. ressources**) suivant dans votre dossier **workshops** puis ouvrez la console et tapez `node scaffold`

```
//scaffold.js

var fs = require('fs');

[
  './concept-es5',
  './concept-es6',
  './concepts-typescript',
  './simple-projects',
  './node',
  './application/server',
  './application/client'
].forEach( i => fs.writeFileSync( i + '/NOTES.md', '# Fichier de prise de notes'))
```

Outils de développement les autres logiciels :

Pour programmer avec angular en JavaScript, un éditeur de texte suffit, il est intéressant de connaître l'offre en outillage autour de Node.

Nous utiliserons VS Code.

cf. pratique

- Git **cf. ressources** attention à ajouter **git au PATH windows!**
- Node.js **cf. ressources**
- Sublime Text 3 **cf. ressources**

Vérifier des installations :

cf. pratique

Vérifier l'installation de git, node et npm (installation par node), ouvrir une invite de commande et saisir les commandes suivantes :

À noter Pour travailler avec les outils de l'ecosystem Angular2 il faut une version de **Node 4+ et NPM 3+**

```
$> node --version  
x.x.x  
  
$> npm --version  
x.x.x  
  
$> git --version  
x.x.x
```

✓ Installation réussie !

Installer des modules tiers

Il existe deux modes d'installation avec npm :

global (machine)

```
$> npm install --global MODULE_NAME  
$> npm i -g MODULE_NAME
```

local (dossier courant)

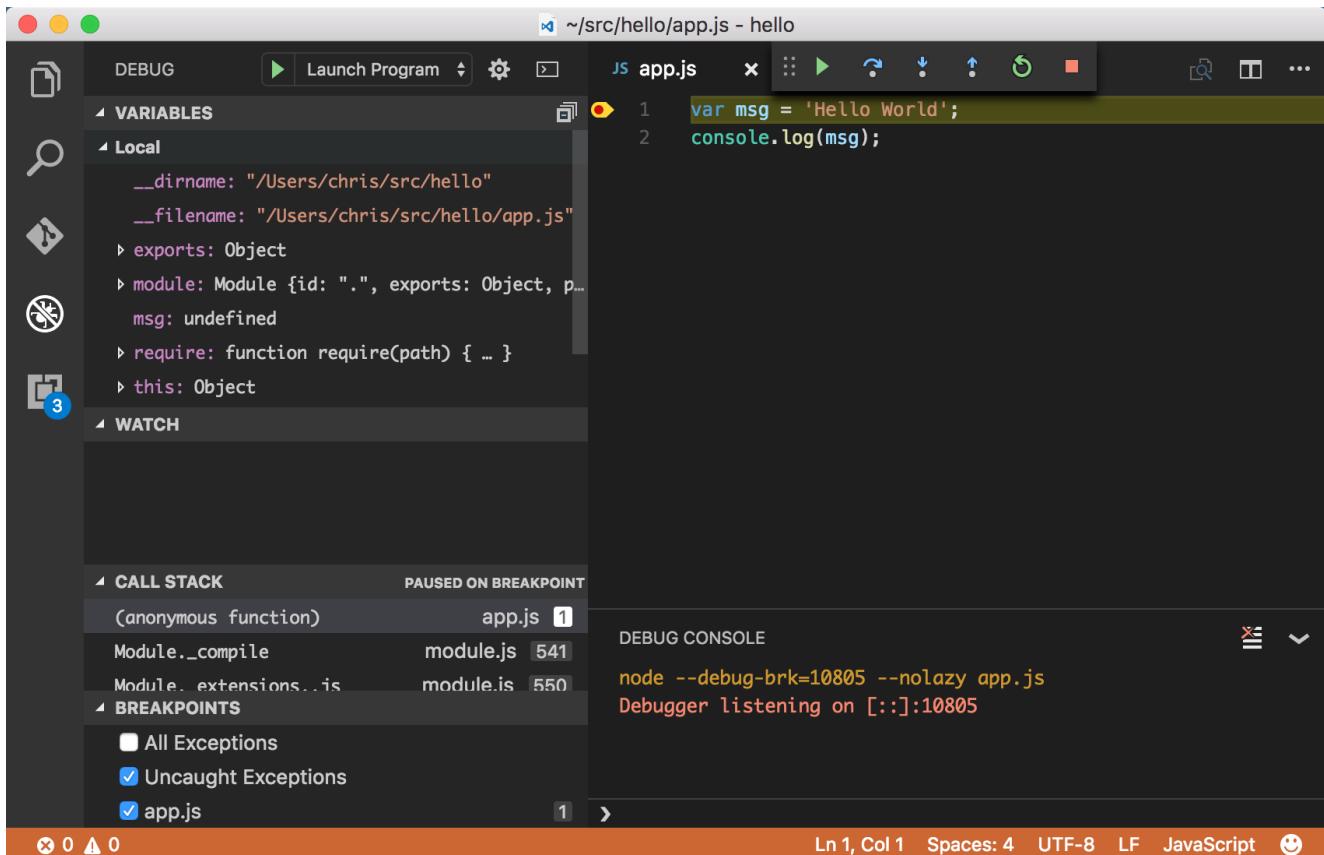
```
$> npm install MODULE_NAME  
$> npm i MODULE_NAME
```

Installation des modules

```
$> npm i -g angular-cli typescript@next json-server lite-server gulp-cli karma-cli
```

1.3/ Découverte de l'éditeur VS Code

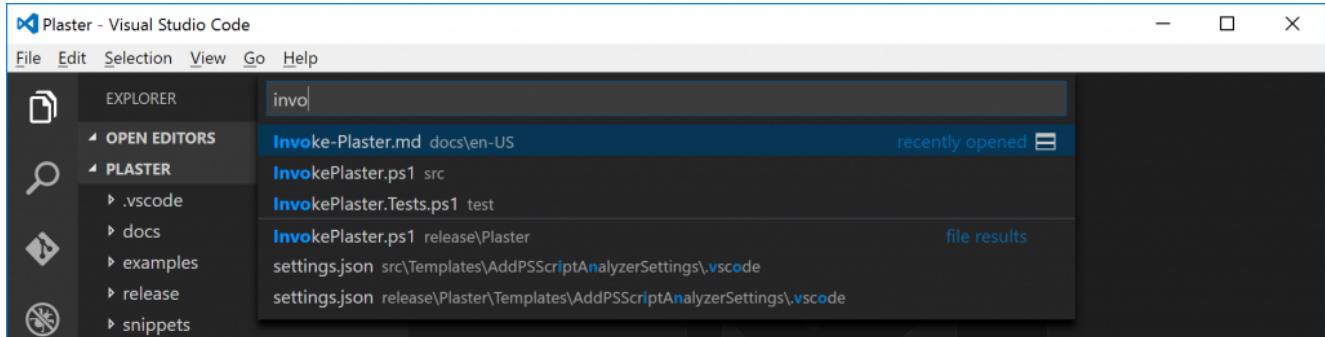
[VS Code](#) apporte toutes les fonctionnalités d'un éditeur moderne, sans nécessiter un investissement de formation.



Fonctionnalités principales:

- Palette de commande
- Gestion des fichiers et des projets
- “Snippets”
- Console
- Débuggeur
- Terminal intégré
- Intégration des plugins

La palette de commande



C'est le centre de contrôle de votre IDE Pour ouvrir la palette de commande, appuyez sur **Ctrl+Maj+P**, tapez le nom d'une commande, les suggestions les plus cohérentes s'affichent dans une liste, validez avec la touche ENTRÉE.

- Utilisation des commandes
- Saisie intuitive

Installation de plugins.

VS Code propose un lien direct vers les extensions disponibles.

Plugins pour Angular2 (ES6 TypeScript)

- Bootstrap 3 Snippets
- Auto Import
- Angular 2 TypeScript Snippets
- Angular 2 TypeScript Emmet
- TypeScript ToolBox
- TypeScript
- AutoFileName

Manipulation

Découverte et utilisation des packages



Développement JavaScript : rappels

2/ Développement JavaScript : rappels.

2.1/ Bonnes pratiques ECMAScript 5.

Il est important d'établir un socle de compétences solides sur JavaScript dans sa norme courante, communément appelée **ES5** avant d'aborder la version **ES6**

- **Vous:** êtes nouveau à la programmation et JavaScript? Alors il vous faut commencer par les base ES5 : **ES6 est un enrichissement de ES5**

Quelques questions simple pour confirmer le socle de compétences.

- **Portée & Closure:** Saviez-vous que JavaScript utilise une portée lexicale basée sur le compilateur ? (pas l'interpréteur!)

Pouvez-vous expliquer :

L'ordre de résolution des variables ?

Leur principe de transmission ?

Pourquoi les *closures* sont le résultat direct de la portée lexicale et fonctions en tant que valeurs?

- **this & Prototypes d'objets:** Pouvez-vous donner les quatre règles simples d'initialisation du mot clé `this` ?
- **Types & Syntaxe:** Connaissez-vous les types **natifs de JS** ? Comment vous sentez-vous avec les nuances syntaxiques en JS ?
- **Async & Performance:** Comment utilisez-vous les `callbacks` pour gérer votre asynchrone? Pouvez-vous expliquer ce qu'est une promesse et pourquoi / comment il résout “callback enfer”?

2.2/ ES5 “use strict” et méthodes moins connues.

Tout vos scripts devraient utiliser la directive ‘**use strict**’;

Le mode strict de ECMAScript 5 permet de choisir une variante restrictive de JavaScript.

Les navigateurs ne supportant pas le mode strict exécuteront le code d'une façon légèrement différente de ceux le supportant.

```
// Tenter d'exécuter ce code
"use strict";
msg = "Allo ! Je suis en mode strict !";
eval('alert(msg)');
```

Le mode strict apporte quelques changements à la sémantique « normale » de JavaScript.

Premièrement, le mode strict élimine quelques erreurs silencieuses de JavaScript en les changeant en erreurs explicites.

Deuxièmement, le mode strict corrige les erreurs limitant l'optimisation du code qui sera exécuté plus rapidement en mode strict.

Troisièmement, le mode strict interdit les mot-clés susceptibles d'être définis dans les futures versions de ECMAScript.

Invoquer le mode strict

Le mode strict s'applique à des scripts entiers ou à des fonctions individuelles.

```
// Script entier en mode strict
"use strict";
var v = "Allo ! Je suis en mode strict !";
```

Attention : il est risqué de concaténer du script en mode strict et du code en mode non-strict. Lors d'une phase de transition, il est donc recommandé de n'activer le mode strict que fonction par fonction.

Le mode strict pour les fonctions

Pour activer le mode strict pour une fonction, on placera l'instruction exacte “use strict”; (ou ‘use strict’;) dans le corps de la fonction avant toute autre déclaration.

```
function strict(){
    // Syntaxe en mode strict au niveau de La fonction
    'use strict';
    function nested() { return "Ho que oui, je le suis !"; }
    return "Allô ! Je suis une fonction en mode strict ! " + nested();
}
function notStrict() { return "Je ne suis pas strict."; }
```

En utilisant “**use strict**”; certaines instructions ou fragments de code lanceront une exception SyntaxError avant l’exécution du script :

- La syntaxe pour la base octale : `var n = 023;`
- L'instruction `with`
- L'instruction `delete` pour un nom de variable `delete maVariable;`
- L'utilisation de `eval()` ou `arguments` comme un nom de variable ou un nom d'argument
- L'utilisation d'un des **nouveaux mots-clés réservés** (en prévision d'ECMAScript 6) :
`implements, interface, let, package, private, protected, public, static, et yield`
- La déclaration de fonctions dans des blocs `if(a<b){ function f(){ } }`
- Déclarer deux fois le nom d'une propriété dans un littéral objet `{a: 1, b: 3, a: 7}`. Ceci n'est plus le cas pour ECMAScript 6 : bug 1041128
- Déclarer deux arguments de fonction avec le même nom `f(a, b, b){}`

Ces erreurs sont bienvenues car elles révèlent de mauvaises pratiques.

Minimisez l'utilisation des éléments dont la sémantique pourrait changer :

- **eval** : n'utilisez cette fonction uniquement si vous êtes certains que c'est l'unique solution
- **arguments** : utilisez les arguments d'une fonction via leur nom ou faites une copie de l'objet en utilisant : `var args = Array.prototype.slice.call(arguments)`
- **this** : n'utilisez this uniquement pour faire référence à un objet que vous avez créé

ES5 méthodes moins connues.

Les fonctionnalités dépréciées

Object.create

La méthode `Object.create()` crée un nouvel objet avec un prototype donné et des propriétés données.

```
var model = {msg:'Hello World'};

var o = Object.create(model, {
    // name est une propriété de donnée
    name: { writable: true, configurable: true, value: 'ES6' },
    // age est une propriété d'accesseur/mutateur
    age: {
        configurable: false,
        get: function() { return 10; },
        set: function(value) { console.log('Définir o.name à', value); }
    }
});

console.log(o.name,o.age,o.msg);
```

Object.keys

La méthode `Object.keys()` renvoie un tableau des propriétés propres à un objet (qui ne sont pas héritées via la chaîne de prototypes) et qui sont énumérables.

```
var arr = ["a", "b", "c"];
console.log(Object.keys(arr)); // affichera ['0', '1', '2']

// un objet semblable à un tableau
var obj = { 0 : "a", 1 : "b", 2 : "c"};
console.log(Object.keys(obj)); // affichera ['0', '1', '2']

// un objet semblable à un tableau avec un ordre de clé aléatoire
var an_obj = { 100: "a", 2: "b", 7: "c"};
console.log(Object.keys(an_obj)); // affichera ['2', '7', '100']

// getName est une propriété non énumérable
var monObjet = Object.create({}, { getName : { value : function () { return this.name } } });
monObjet.name = 1;

console.log(Object.keys(monObjet)); // affichera ['name']
```

Object.seal

La méthode `Object.seal()` scelle un objet afin d'empêcher l'ajout de nouvelles propriétés, en marquant les propriétés existantes comme non-configurables.

Les valeurs des propriétés courantes peuvent toujours être modifiées si elles sont accessibles en écriture.

```
var obj = {
    prop: function () {},
    msg: "Hello"
};

// On peut ajouter de nouvelles propriétés
// Les propriétés existantes peuvent être changées ou retirées
obj.msg = "World";
obj.name = "Bob";
delete obj.name;

var o = Object.seal(obj);
o === obj; // true
Object.isSealed(obj); // true

obj.coincoin = "mon canard"; // La propriété n'est pas ajoutée
delete obj.msg; // La propriété n'est pas supprimée
```

Array.prototype.map

La méthode map() crée un nouveau tableau composé des images des éléments d'un tableau par une fonction donnée en argument.

Array.prototype.filter

La méthode filter() crée et retourne un nouveau tableau contenant tous les éléments du tableau d'origine pour lesquels la fonction callback retourne true.

Array.prototype.reduce

La méthode reduce() applique une fonction qui est un « accumulateur » et qui traite chaque valeur d'une liste (de la gauche vers la droite) afin de la réduire à une seule valeur.

```
var users = [{name:'Bill',age:'10'}, {name:'Bob',age:'19'}, {name:'Marine',age:'25'}]

users.filter(function(e,i,a){return e.age > 18 })
    .map(function(e,i,a){ var n = e; n.name = n.name.toUpperCase(); return n; })
    .reduce(function(e1, e2, i , a) {
        return e1.name.concat(e2.name)
    });
}
```

2.3/ API issues de la communauté JavaScript.

La communauté des développeurs JavaScript apport parfois des convention ou librairie en réponse à une problématique données (DOM, Asynchronicité...) et certaines de ces solutions se voient parfois reprise dans les standards.

Getter/Setter : non standardisé mais très répandu.

```
$(‘li’).html() // getter : retourne la valeur  
$(‘li’).html(‘Hello World’) // setter : affecte la valeur
```

Promise : standardisé.

On doit une première implémentation à [Kris Kowal](#) via la librairie **q**

En substance une **promise** ou traitement déferré doit retoutner un objet contenant une méthode **then()** recevant deux paramètres (**callbackSuccess, callbackError**) afin de représenter les état du traitement (en attente, résolue, rejetée)

```
var defer = new Promise((resolve, reject) => { }); //Standard ES6  
  
//Avec Q  
Q.fcall(promisedFunction)  
.then(function (value) {  
})  
.catch(function (error) {  
})  
.done();
```

Standardisation de la manipulation du DOM basée sur les sélecteur CSS

Héritage de **jQuery**

```
document.querySelector(‘body>.btn’);  
document.querySelectorAll(‘body>.btn’);
```

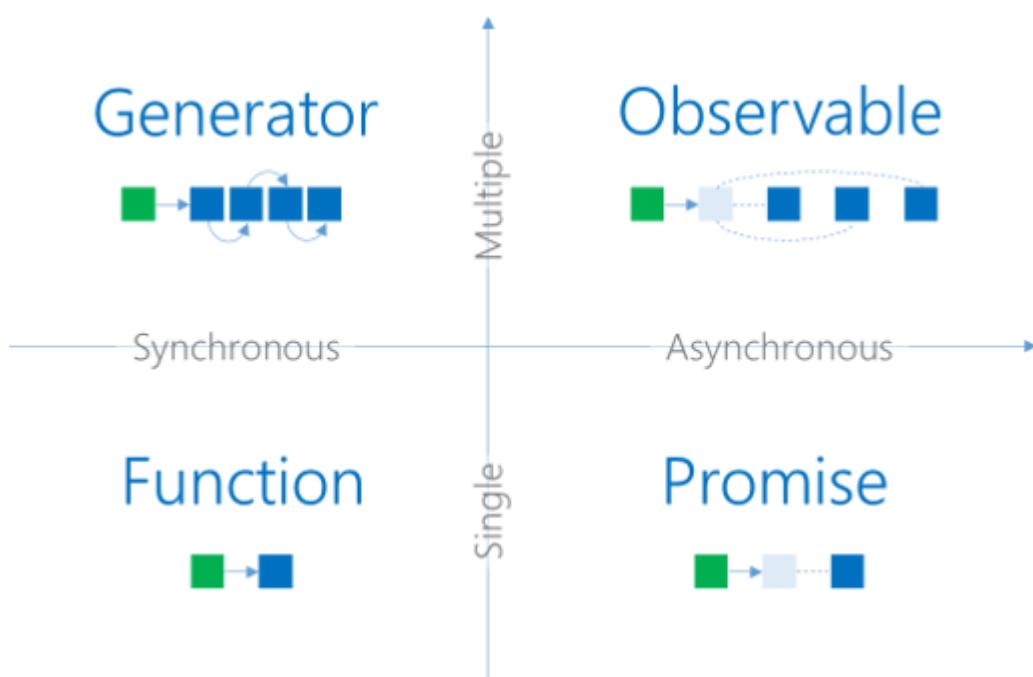
Programmation Réactive.

Définition : Flux Evennementiels. La programmation réactive est un paradigme visant à conserver une cohérence d'ensemble en propageant les modifications d'une source réactive (modification d'une variable, entrée utilisateur, etc.) aux éléments dépendants.

Des librairies telles que **reactivex.io** posent de nouveau paradigmes tels que la représentation d'évennement sous la forme de flux itérables.

Ressources

- <https://www.learnrxjs.io/>
- <http://reactivex.io/rxjs/>

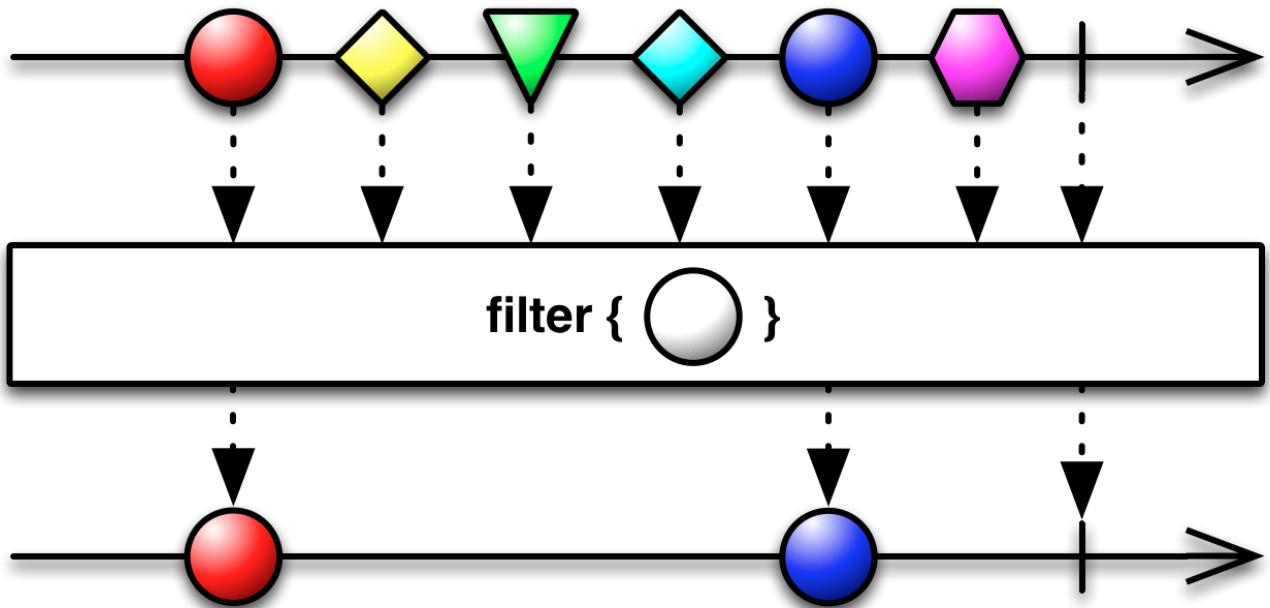


La librairie JavaScript RxJS permet de créer des flux de données qui arrivent au fil du temps, de les filtrer, de les modifier, de les combiner.

Les données d'un flux peuvent être des valeurs (string, number...) mais également des événements du DOM, des tableaux, des promises etc.

Concepts Clés :

- **Observable:** collection de valeurs futures.
- **Observer:** collection de callbacks surveillant un `Observable`.
- **Subscription:** association vers un `Observable`.
- **Operators:** `pure functions` utilisées dans les opérations telles `map`, `filter`, `concat`, `flatMap`, etc.
- **Subject:** permet le `multicasting` vers de multiples `Observers`.
- **Schedulers:** gestion centralisée des mise à jour.



```
//Exemple écouter un click
var button = document.querySelector('button');
Rx.Observable.fromEvent(button, 'click')
  .throttleTime(1000) // Un seul click par secondes
  .scan(count => count + 1, 0) // Création d'un compteur par la méthode scan
  .subscribe(() => console.log('Clicked!'));
```

Créer un Observable

Il existe plusieurs méthodes [permettant la création](#) d'un Observable selon la nature de la source originale

- **create** - Création personnalisée.
- **empty** - Collection terminée.
- **from** - A partir de Array, Iterable, Promise
- **fromEvent** - A partir d' Event
- **fromPromise** - A partir de Promise
- **interval** - A partir d'un interval donné pour la fréquence d'émission de valeur.
- **of** - A partir de valeurs de natures différentes.
- **range** - Pour une plage donnée.
- **throw** - Retourne un erreur.
- **timer** - Selon une fréquence de temps programmée.

Manipuler les séquences

RxJS permet la manipulation fine des traitements asynchrones à [travers de très nombreuses méthodes](#) :

Manipuler chaque valeur		map/select
Accéder à une propriété de chacune des valeurs		pluck
Notification de l'évolution des valeurs sans modification		do/tap doOnNext/tapOnNext doOnError/tapOnError doOnCompleted/tapOnCompleted
Inclure des valeurs additionnelles	Logique personnalisée	filter/where
	Depuis le début de la séquence	take
	Logique personnalisée	takeWhile
	A partir de la fin de la séquence	takeLast
	Jusqu'à la complétion d'une séquence différente	takeUntil
Toutes	Toutes	ignoreElements
	Depuis le	skip

Ignorer des valeurs	début de la séquence	Logique personnalisée	skipWhile
	A partir de la fin de la séquence		skipLast
	Jusqu'à la complétion d'une séquence différente		skipUntil
	de valeur identique		distinctUntilChanged
	trop fréquentes		throttle
Calculer	Somme	des valeurs	sum
	Moyenne		average
	Logique personnalisée	Retourner seulement le résultat final	aggregate reduce
		Retourner toutes les valeurs calculées	scan
	Compte du nombre de valeurs		count
Ajouter des 'meta-data'	descriptifs		materialize
	contenant le temps écoulé depuis le dernier message		timeInterval
	avec timestamp		timestamp
Gérer l'inactivité	Lever une Erreur		timeout
	Changer de séquence		timeout
Vérifier l'unicité de valeur	Et lever une erreur		single
	et utiliser une valeur par défaut		singleOrDefault
Accéder seulement à	Et lever une erreur à défaut		first
	ou utiliser une valeur par défaut		firstOrDefault

la première valeur	Dans une période de temps données	sample
Accéder seulement à la dernière valeur	Et lever une erreur à défaut	last
	et utiliser une valeur par défaut	lastOrDefault
I want to know if a condition is satisfied	by any of its values	some
	by all of its values	all/every
Retarder les messages		delay
	Logique personnalisée	delayWithSelector
Grouper les valeurs	Jusqu'à la complétion	
	Logique personnalisée	toArray
		toMap
		toSet
	Par taille	buffer
		window
	Selon le temps	bufferWithCount
		windowWithCount
	Alternance Temps / Taille	bufferWithTime
		windowWithTime
	Selon une clé	bufferWithTimeOrCount
		windowWithTimeOrCount
	jus'qu'à la completion	groupBy
		groupByUntil



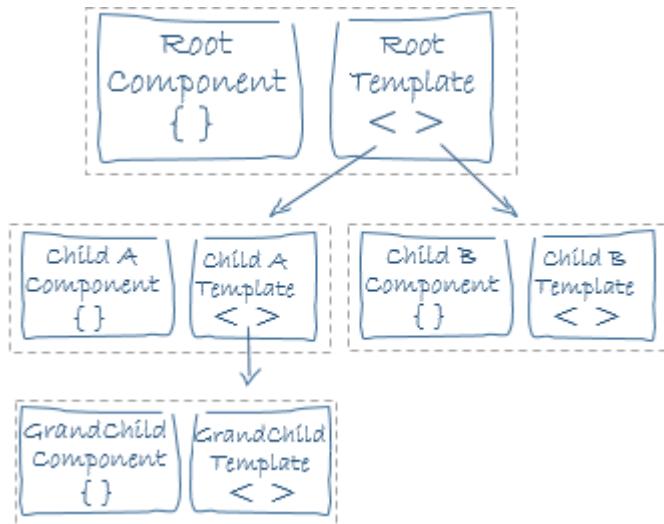
Angular2 : Présentation

3/ Angular2 : Présentation.

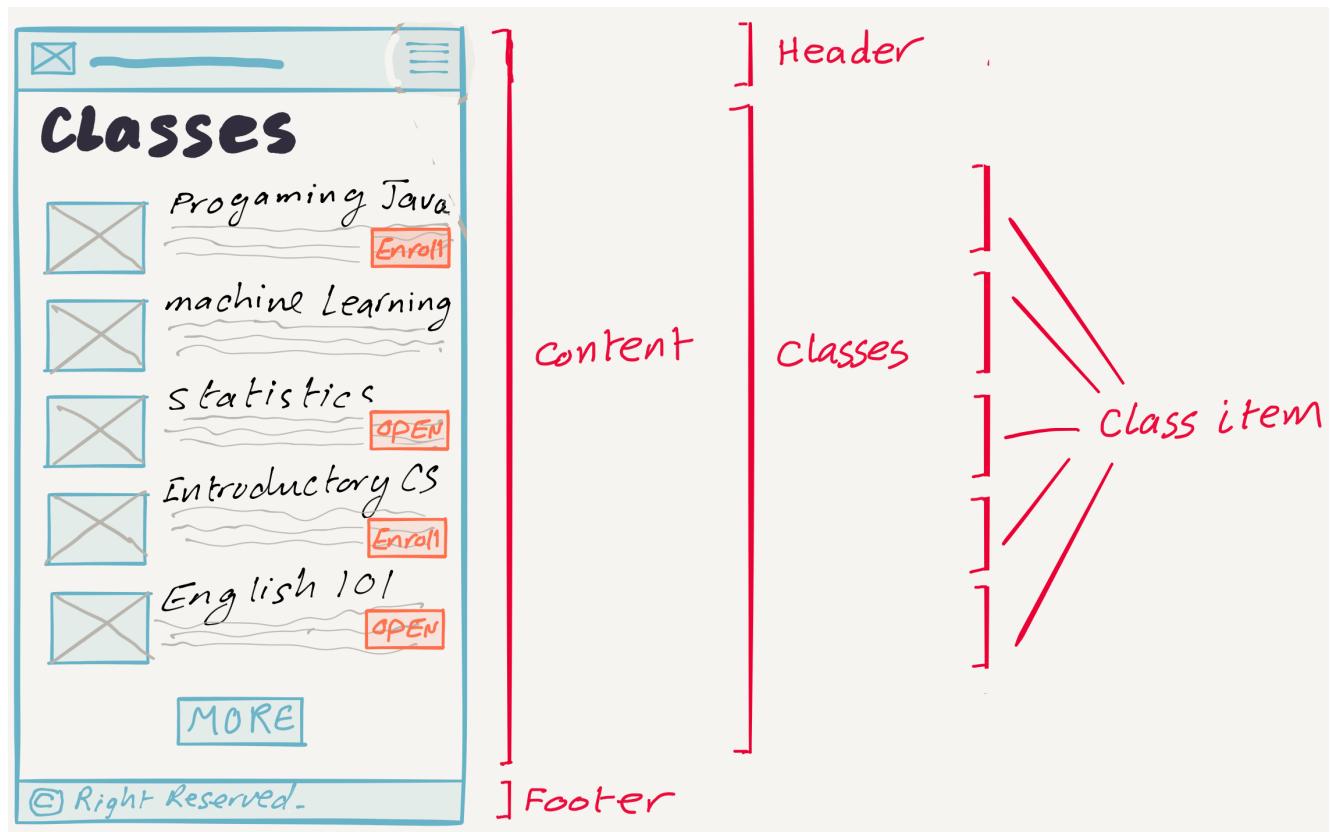
La refonte en version 2 du framework est basée sur **4 principes fondateurs**.

- **Augmenter les performances** : L'un des plus grands reproches qui ait été fait à Angular sont ses lacunes en termes de performance.
- **Améliorer la productivité** : la syntaxe est expressive, basée sur la syntaxe de ES2015/TypeScript (annotations, import, types, ...), plutôt que sur des surcouches.
- **S'adapter au mobile** : la conception modulaire du framework permet de réduire considérablement son empreinte mémoire sur les terminaux mobiles.
- **Embrasser les nouveaux standards du Web** en se basant sur des technologies telles que le ShadowDom (WebComponents), Observables et autres nouveautés apportées par ES2015.

Le framework angular permet de concevoir une application comme une arborescence de composants.



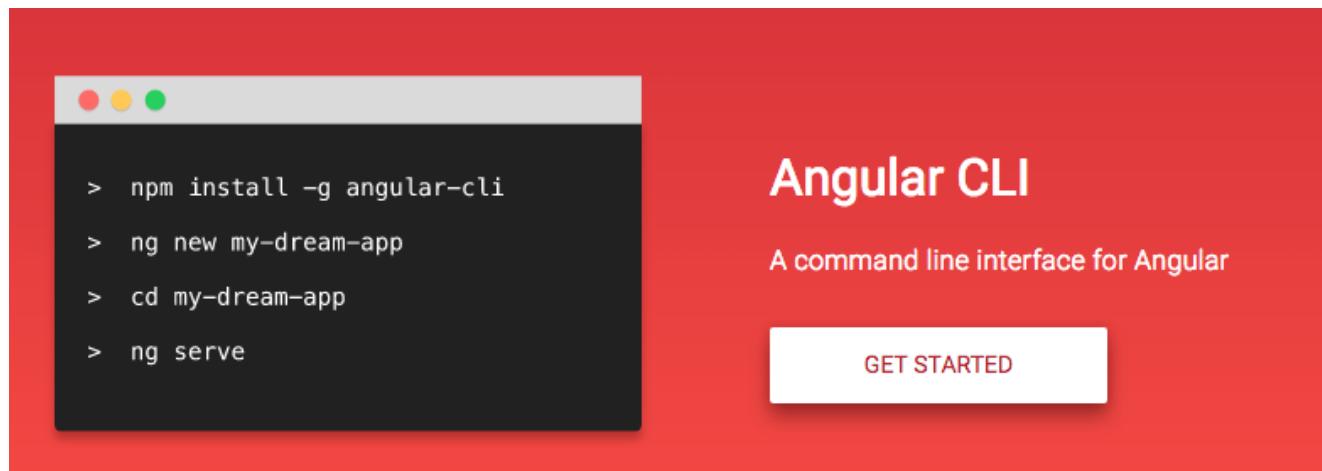
Les composants définissent des unités fonctionnelles plus ou moins réutilisables au sein de l'interface utilisateur.



Angular-cli : l'outil intégré.

Angular2 propose un [outil de génération](#) pour assister la mise en oeuvre du workflow .

Les nombreuses dépendances et l'ensemble des tâches nécessaires à leur mise en oeuvre rendent l'utilisation du `scaffolder` indispensable.



Initialiser le projet

La commande `ng new APP_NAME` initialise la structure globale du projet. Tandis que `ng help` permet de voir toutes les commandes disponibles.

Développer avec le génératrices

En plus des commandes illustrées sur l'image cli propose des micro-générateurs:

Scaffold	Commande
Component	<code>ng g component my-new-component</code>
Directive	<code>ng g directive my-new-directive</code>
Pipe	<code>ng g pipe my-new-pipe</code>
Service	<code>ng g service my-new-service</code>
Class	<code>ng g class my-new-class</code>
Interface	<code>ng g interface my-new-interface</code>
Enum	<code>ng g enum my-new-enum</code>
Module	<code>ng g module my-module</code>

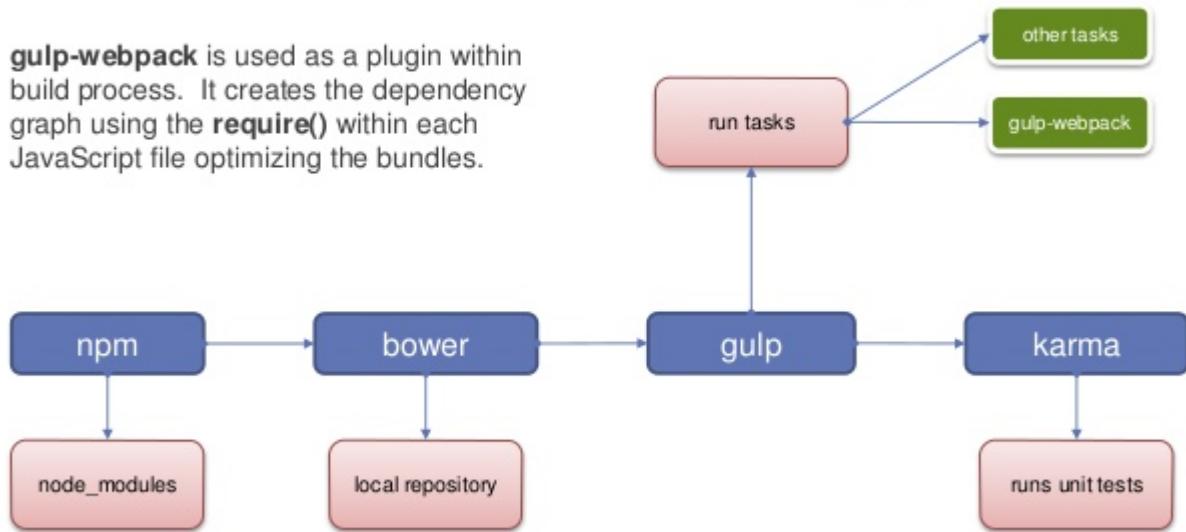
Toutes ses commandes ne sont pas égales. Seules component, directive, pipe, service, module génèrent des définitions propres à Angular les autres commandes sont des assistances.

Les autres commandes disponibles (sujettes à variation) sont énumérées sur la [page du projet](#)

Build et Workflow

Webpack workflow

gulp-webpack is used as a plugin within build process. It creates the dependency graph using the **require()** within each JavaScript file optimizing the bundles.



[WebPack](#) . est un **module bundler** , un utilitaire manipulant les ressources afin de les préparer pour leur distribution.

L'ensemble des transformations sont ainsi réalisées en tâche de fond et immédiatement répercutées dans le navigateur du développeur.

Build et Workflow

L'outil prévoit le **build** en se basant sur l'utilitaire [WebPack](#) .

```
ng build --bh /myUrl/  
ng build --prod
```

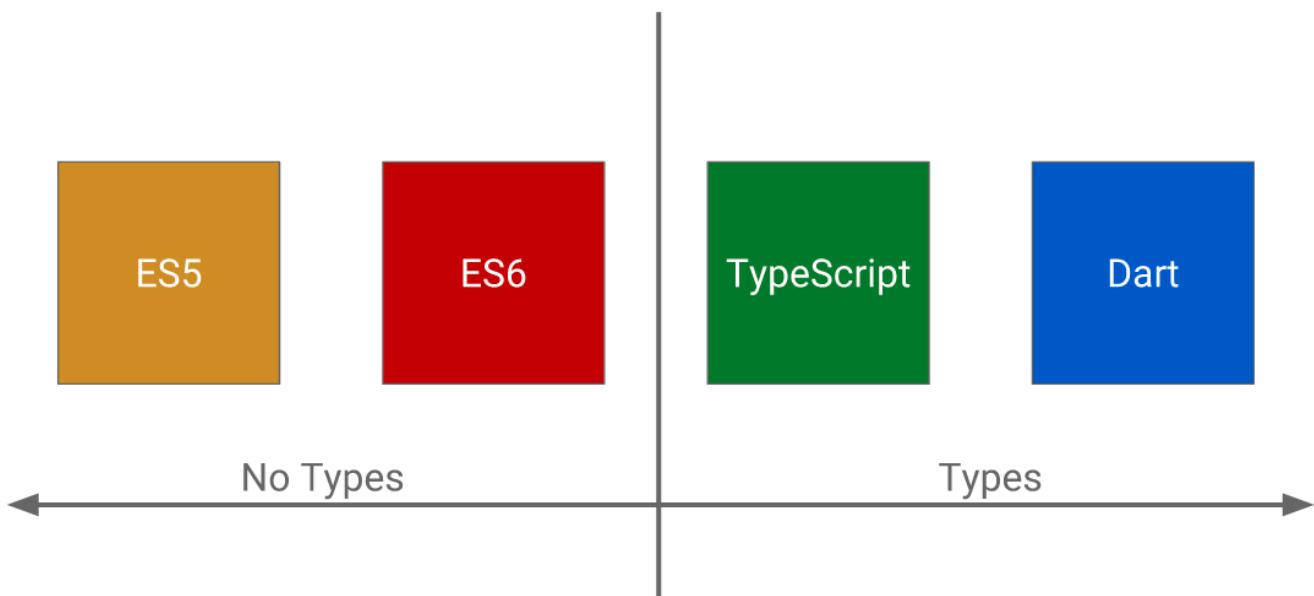
Angular avec TypeScript et ES6/ES2015.

Angular2 est écrit en **TypeScript** si il est recommandé d'adopter l'outil `TypeScript`, Angular2 supporte aussi bien ES5.

TypeScript est un `superset` d'Ecma Script 6/7. Utilisé conjointement avec des [polyfills](#) il permet de tirer parti des évolutions du langages.

L'apprentissage d'Angular passe par celui de `TypeScript` étant lui même un superset ES6/2015

ES5 > ES6 > TypeScript > Angular



Développées par les [membres](#) du groupe de travail [TC39](#), ECMAScript 6 proposé en 2015 est une évolution significative du langage depuis ES5 (2009).



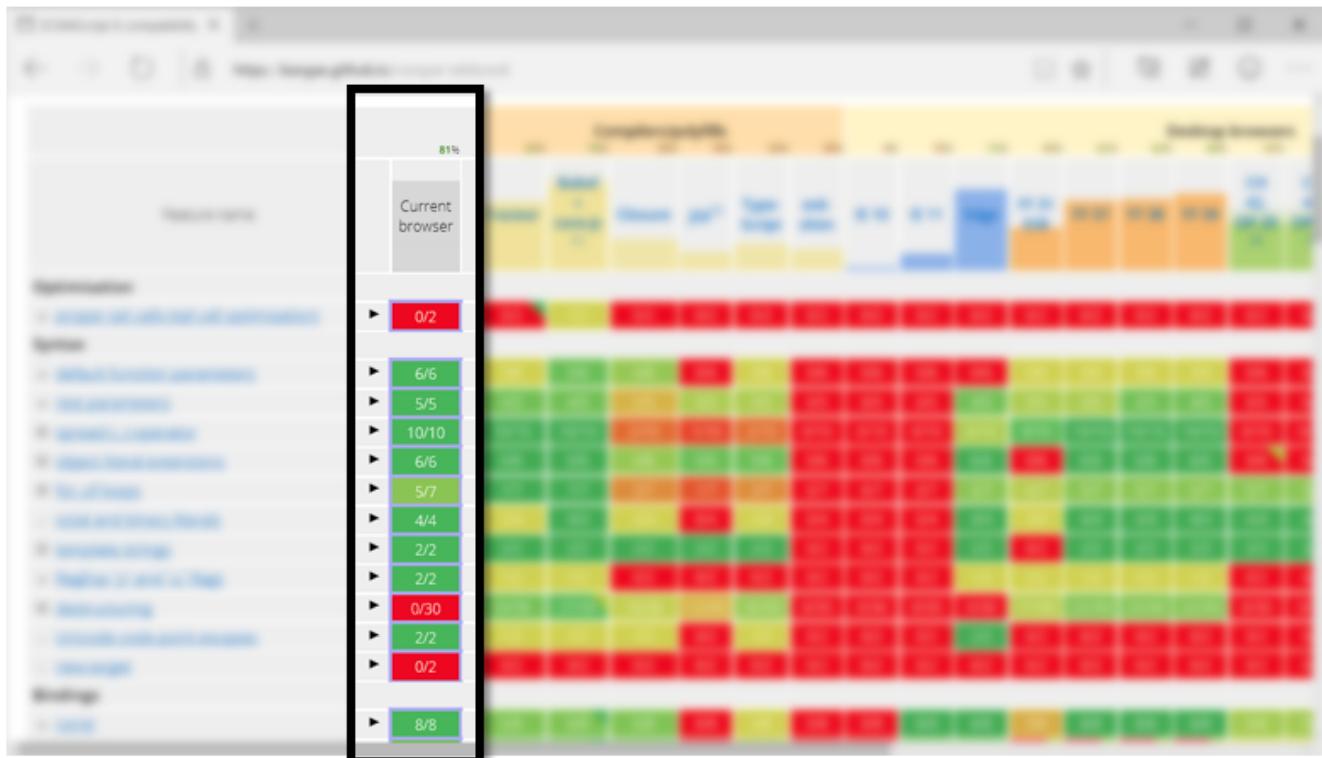
Les moteurs JavaScript mettent en œuvre des caractéristiques individuelles graduellement.

Autres références

- Site : (anglais) [outils pour ES6](#)
- Site : (playground) [Tester ES6](#)
- Site : (playground) [Learn Harmony](#)
- Site : (playground) [ES6 ES5 Comparaison](#)
- Livre : (anglais) [JS.next: A Manager's Guide](#)
- Livre : (anglais) [You Don't Know JS: ES6 & Beyond](#)

Support courant pour ES6 : compilateurs, polyfills, navigateurs serveurs.

La meilleure façon de suivre l'évolution du support ES6 sont les [tableaux de compatibilité ECAMScript](#) maintenu par **Juriy Zaytsev** (@kangax)



Environnement et outils pour le développeur.

Transpilers

Avec l'évolution rapide des caractéristiques, un problème se pose pour les développeurs JavaScript qui désirent utiliser les nouvelles fonctionnalités tout en maintenant ses applications pour les anciens navigateurs.

Avant ES5 il était d'usage d'attendre que la plupart, sinon tous, que les environnements javascript supporte ES5. En conséquence beaucoup d'applications n'utilisent pas encore des options comme le `strict`, qui a pourtant plus de cinq ans.

Il est considéré néfaste pour l'avenir de l'écosystème JS d'attendre avant d'utiliser les fonctionnalités qui représentent de bonnes pratiques.

Comment anticiper le support natif ?

La réponse est l'outillage, en particulier une technique appelée **transpiling** (transformation + compilation).

L'idée est d'utiliser un outil pour transformer votre code ES6 à un équivalent (ou presque!) ES5.

Les *transpilers* assure cette transformation comme part du **workflow** de la même manière que les *linter* ou les opérations de minification.

```
var foo = [1,2,3];

var obj = {
  foo // means `foo: foo`
};

obj.foo; // [1,2,3]
```

Exemple de *transpilation*:

```
var foo = [1,2,3];

var obj = {
  foo: foo
};

obj.foo; // [1,2,3]
```

Shims/Polyfills

Polyfill implémentation d'une fonctionnalité attendue mais non disponible.

```
if (!Object.is) {
  Object.is = function(v1, v2) {
    // test for `~-0`
    if (v1 === 0 && v2 === 0) {
      return 1 / v1 === 1 / v2;
    }
    // test for `NaN`
    if (v1 !== v1) {
      return v2 !== v2;
    }
    // everything else
    return v1 === v2;
  };
}
```

[“ES6 Shim”](#) est une collection de polyfills que vous pouvez utiliser.

{js}

ES6 Est utilisable en production au moyen de transpiler.

Les nouvelles librairies et frameworks se baseront sur ES6.

ES6-Shim Est disponible.

Utilisez le mode strict dans votre code.

JavaScript est un language dynamique suivant des règles simples (code hoisting, scope...).

Object et Array possède des méthodes ES5 qui allégeront votre code.



ES6/2015 : Evolutions syntaxiques.

4/ ES6/2015 : Evolutions syntaxiques fondamentales.

Aperçu général des apports.

Promise easy basics creation chaining `then()` the API	Array Array.from() Array.of() [].fill() [].find() [].findIndex() [].entries() [].keys() [].values()	Class easy creation accessors easy static extends more extends super in method super in constructor	Destructuring easy array easy string easy object easy defaults parameters assign	Generator creation iterator yield expressions send value to a generator send function to a generator `return` inside a generator function	Map easy Basics map.get() map.set() initialize easy map.has()
Reflect easy Basics Reflect.apply() Reflect.getPrototypeOf() Reflect.construct() Reflect.defineProperty()	Set basics set.add() set.delete() easy the API easy set.clear()	Iterator array string protocol usage	Object literal basics computed properties getter setter	String easy string.includes() easy string.repeat(count) easy string.startsWith() easy string.endsWith()	
Template strings easy basics easy multiline tagged template strings `raw` property	Symbol basics Symbol.for() Symbol.keyFor()	Arrow functions easy basics easy function binding	Block scope easy `let` declaration easy `const` declaration	Rest operator as parameter with destructuring	Spread operator with arrays with strings
Default parameters easy Basics	Modules easy `import` statement	Number easy Number.isInteger()	Object easy Object.is()	Unicode in strings	

Constantes et variables de bloc. Assignton destructurée.

Variables de bloc.

L'instruction `let` permet de déclarer des variables dont la portée est limitée à celle du bloc dans lequel elles sont déclarées.

Au niveau le plus haut (la portée globale), `let` crée une variable globale alors que `var` ajoute une propriété à l'objet.

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

Rappel : Le mot-clé `var` permet de définir une variable globale ou locale à une fonction (sans distinction des blocs utilisés dans la fonction).

```
for (let i = 1; i <= 5; i++) {
    setTimeout(function(){console.info(i)},1000)
}

for (var i = 1; i <= 5; i++) {
    setTimeout(function(){console.warn(i)},1000)
}
```

Redéclarer une même variable au sein d'une même portée de bloc entraîne une exception

TypeError Contrairement au corps de fonction.

```
if (x) {  
    let myVar;  
    let myVar; // TypeError  
}  
  
function foo() {  
    let myVar;  
    let myVar; // Cela fonctionne.  
}
```

Faire référence à une variable dans un bloc avant la déclaration de celle-ci avec `let` entraînera une exception `ReferenceError`.

La variable est placée dans une « zone morte temporaire » entre le début du bloc et le moment où la déclaration est traitée.

Zone morte temporaire (temporal dead zone / TDZ) concerne les erreurs liées à `let`

```
function foo() {  
    console.log(myVar); // ReferenceError  
    let myVar = true;  
}
```

Constantes

La déclaration `const` permet de déclarer un identifiant constant.

Attention: Cela ne signifie pas que la valeur contenue est immuable, uniquement que l'identifiant ne peut pas être réaffecté.

Les constantes font partie de la portée du bloc (comme les variables définies avec `let`).

```
const myVar = true;
myVar = false;

const myObj = {};
myObj.property = false;
myObj = {};
```

- Il est nécessaire d'initialiser une constante lors de sa déclaration.
- Il est impossible d'avoir une constante qui partage le même nom qu'une variable ou qu'une fonction.(Au sein d'une même portée)

Usage:

On préférera `let` pour les variables et `const` pour les fonctions.

```
const foo = function foo() {
  let myVar = true;
  console.log(myVar);
};
```

Attribution déstructurée.

L'affectation par décomposition (destructuring en anglais) est une expression JavaScript permettant d'extraire des données d'un tableau ou d'un objet d'après une comparaison de forme syntaxique.

```
[a, b] = [1, 2]
[a, b, ...c] = [1, 2, 3, 4, 5]
{a, b} = {a:1, b:2}
```

L'intérêt de l'attribution par décomposition est de pouvoir lire une structure entière en une seule instruction.

```
var myVar= ["un", "deux", "trois"];

// sans utiliser la décomposition
var un    = myVar[0];
var deux  = myVar[1];
var trois = myVar[2];

// en utilisant la décomposition
var [un, deux, trois] = myVar;
```

Échange de variables :

```
var a = 1;
var b = 3;

[a, b] = [b, a];
```

Renvoyer plusieurs valeurs:

Il était déjà possible de renvoyer un tableau/objet mais cela ajoute un nouveau degré de flexibilité.

```
function f() {
    return [1, 2];
}
var [a, b] = f();
var x = f();
console.log("A vaut " + a + " B vaut " + b, x);

var url = "http://www.orsys.fr/formation-Ecmascript-6.asp/";

var parsedURL = /^(\\w+)\\:\\\\\\/([\\\\/]+)\\\\/(.*$)/.exec(url);
var [, protocol, fullhost, fullpath] = parsedURL;

console.log(protocol); // enregistre "http"
```

Ignorer certaines valeurs:

```
function f() {
    return [1, 2, 3];
}

var [a, , b] = f();
console.log("A vaut " + a + " B vaut " + b);
```

Décomposition d'objet:

```
var o = {a: 111, b: true};  
var {a, b} = o;  
  
console.log(a); // 111  
console.log(b); // true  
  
// Réasignation  
var {a: newA, b: newB} = o;  
  
console.log(newA); // 111  
console.log(newB); // true
```

Propriétés calculées:

Il est possible d'utiliser des noms de propriétés calculés, comme avec les littéraux objets, avec la décomposition.

```
let key = "a";  
let { [key]: myVar } = { a: "Hello World" };  
  
console.log(myVar); // "truc"
```

Fonction, paramètres par défaut, opérateurs “rest / spread”.

Paramètres par défaut.

En JavaScript, les paramètres de fonction non renseignés valent `undefined`. EN ES6 il est possible de définir une valeur par défaut différente.

```
//ES5
unction multiplier(a, b) {
  b = typeof b !== 'undefined' ? b : 1;

  return a*b;
}

multiplier(5); // 5
```

Code allégé avec ES6

```
function multiplier(a, b = 1) {
  return a*b;
}

multiplier(5); // 5
```

Les paramètres déjà rencontrés dans la définition peuvent être utilisés comme paramètres par défaut dans la suite de la définition :

```
function singulierAutoPluriel(singulier, pluriel = singulier+s", message = pluriel +
  return [singulier, pluriel, rallyingCry ];
}

function go() {
  return ":P"
}

function avecDéfaut(a, b = 5, c = b, d = go(), e = this, f = arguments, g = this.value
  return [a,b,c,d,e,f,g];
}
```

Auparavant, pour définir une valeur par défaut pour un paramètre, il fallait tester s'il valait `undefined` et lui affecter une valeur choisie le cas échéant.

L'argument par défaut est évalué à l'instant de l'appel. Un nouvel objet est créé à chaque appel de la fonction.

Paramètres “rest”.

La syntaxe des paramètres du reste permet de représenter un nombre indéfini d'arguments contenus dans un tableau.

```
function multiplier(facteur, ...lesArgs) {  
    return lesArgs.map(x => facteur * x);  
}  
  
var arr = multiplier(2, 1, 2, 3);  
console.log(arr); // [2, 4, 6]
```

Opérateur “spread”.

L'opérateur `spread` permet de développer une expression lorsque plusieurs arguments ou plusieurs éléments sont nécessaires (respectivement pour les appels de fonctions et les littéraux de tableaux).

Syntaxe

Pour l'utilisation de l'opérateur dans les appels de fonction :

```
foo(...objetIterable);
```

Pour les littéraux de tableaux :

```
[...objetIterable, 4, 5, 6]
```

Pour la décomposition :

```
[a, b, ...objetIterable] = [1, 2, 3, 4, 5];
```

Utilisation

Tout argument passé à une fonction peut être décomposé grâce à l'opérateur et l'opérateur peut être utilisé pour plusieurs arguments.

```
function foo(v, w, x, y, z) { console.log([v, w, x, y, z])}
var args = [0, 1];
foo(-1, ...args, 2, ...[3]);

//Ignorer des paramètres

foo(...[, ,4]);
```

Pour créer un nouveau tableau composé du premier, on peut utiliser un littéral de tableau avec la syntaxe de décomposition, cela devient plus succinct :

```
var articulations = ['épaules', 'genoux'];
var corps = ['têtes', ...articulations, 'bras', 'pieds'];
// ["têtes", "épaules", "genoux", "bras", "pieds"]
```

Pour la création d'objet :

```
var champsDate = lireChampsDate(maBaseDeDonnées);
var d = new Date(...champsDate);
```

Pour optimiser les méthodes :

```
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
arr1.push(...arr2);
```

Chaînes de caractères : multiligne, template, formatage.

Les **template string** (template literals) sont des gabarits de chaînes de caractères qui intègrent des expressions.

Il est possible d'utiliser des chaînes de caractères sur plusieurs lignes ainsi que les fonctionnalités d'interpolation de chaînes.

Syntaxe : le délimiteur est le symbole `

Les **template string** délimités par des accents graves seuls (backticks) et non avec des doubles ou simples quotes.

```
`chaîne de texte`  
  
`chaîne ligne 1  
chaîne ligne 2`  
  
`texte ${expression} texte chaîne`  
  
function tag(){console.log(arguments)}  
tag `texte ${expression} texte chaîne`
```

Ils peuvent contenir des éléments de substitution (placeholders) indiqués par le signe dollar (\$) et des accolades : **\${expression}**.

Les expressions contenues dans les éléments de substitution et le texte sont ensuite passés à une fonction.

Si une expression précède le gabarit (par exemple : tag ci-dessus), le gabarit est appelé “étiqueté”.

Dans ce cas, l’expression qui étiquette (une fonction) le gabarit est appelée pour traiter le gabarit.

Interpolation d'expression

- Le code desubstitution peut être toute expression JavaScript, appels de fonction, logique et arithmétique sont autorisés.
- Si l'une des valeurs n'est pas une chaîne, il sera converti en une chaîne en utilisant les règles habituelles. Par exemple, si l'action est un objet, sa méthode `.toString()` sera appelée.
- Si vous devez écrire un backtick dans une chaîne de modèle, vous devez échapper avec une barre oblique inverse: `\``.
- Pour les deux caractères `$ {` vous pouvez échapper le caractère avec une barre oblique inverse: `\$` ou `\{`.

```
var a = 5;
var b = 10;
console.log(`Quinze est ${a + b} et n'est pas ${2 * a + b}.`);
// "Quinze est 15 et n'est pas 20."
```

Il est possible d'utiliser les gabarits de manière plus avancée grâce à l'étiquetage de gabarits.

“Arrow Function” : portée lexicale. Usages.

Les **Arrows Function** sont un raccourci syntaxique pour la création de fonction utilisant le symbole `=>`.

Les fonctions ainsi créées sont syntaxiquement similaire à la fonction en C#, Java 8 et CoffeeScript.

Elles définissent le corps (bloc de code) et la valeur de retour. Contrairement aux fonctions classiques, les **Arrows function** partagent la même valeur lexicale pour le mot clé `this` que leur code environnant.

```
var odds = evens.map(v => v + 1);
var nums = evens.map((v, i) => v + i);
var pairs = evens.map(v => ({even: v, odd: v + 1}));  
  
nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
});  
  
var bob = {
  _name: "Bob",
  _friends: [],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " knows " + f));
  }
}
```

Une expression de fonction fléchée permet d'utiliser une syntaxe plus concise que les expressions de fonctions classiques. La valeur `this` est alors **liée lexicalement**. Les fonctions fléchées sont nécessairement anonymes.

Les **Arrows** sont un raccourci pour la création de fonction en utilisant le `=>`. Elles définissent le corps (bloc de code) et la valeur de retour.

Fonction itératrice : “iterator”.

Un des éléments ajoutés par ECMAScript 2015 (ES6) n'est ni une nouvelle syntaxe ni un nouvel objet natif mais un protocole.

Ce protocole peut être implémenté par n'importe quel objet qui respecte certaines conventions.

Deux composantes : les itérables, les itérateurs.

Le protocole « itérable » permet aux objets JavaScript de définir ou de personnaliser leur comportement lors d'une itération, par exemple la façon dont les valeurs seront parcourues avec une boucle `for..of`.

Certains types natifs tels que `Array` ou `Map` possèdent un comportement itératif par défaut, d'autres types, comme `Object` n'ont pas ce type de comportement.

itérable

Pour itérable, un objet doit implémenter la méthode `@@iterator`, l'objet doit avoir une propriété avec une clé `Symbol.iterator` :

[`Symbol.iterator`] Une fonction sans argument qui renvoie un objet conforme au protocole itérateur.

itérateur

Un objet est considéré comme un itérateur lorsqu'il implémente une méthode `next()`:

Une fonction sans argument qui renvoie un objet qui possède deux propriétés :

- **done (booléen)** true lorsque l'itérateur a fini la suite. **false** lorsque l'itérateur a pu produire la prochaine valeur de la suite.
Si on ne définit pas la propriété `done`, on aura ce comportement par défaut.
- **value (any)**, renvoyée par l'itérateur.
Cette propriété peut être absente lorsque `done` vaut `true`.

Exemple : Une String est un exemple d'objet natif itérable :

```
var msg = "Hello";
typeof msg[Symbol.iterator];// "function"

var iterator = msg[Symbol.iterator]();
console.log(iterator); // "[object String Iterator]"

iterator.next(); // { value: "H", done: false }
iterator.next(); // { value: "e", done: false }
iterator.next(); // { value: "l", done: false }
iterator.next(); // { value: "l", done: false }
iterator.next(); // { value: "o", done: false }
iterator.next(); // { value: undefined, done: true }
```

Les itérables natifs

String, Array, TypedArray, Map et Set sont des itérables natifs car leurs prototypes possèdent une méthode @@iterator.

Eléments de syntaxique utilisant des itérables

```
// Itération avec for of
for(let value of ["a", "b", "c"]){
  console.log(value);
}

// rest, spread
[..."abc"]; // ["a", "b", "c"]

// Fonction génératrice
function* gen(){
  yield* ["a", "b", "c"];
}

gen().next(); // { value:"a", done:false }

// Décomposition
[a, b, c] = new Set(["a", "b", "c"]);
a; // "a"
```

Objet littéral : évolution.

Les objets littéraux sont étendus pour :

- supporter la définition du prototype à la construction.
- simplifier les affectations de propriété/méthodes.
- permettre avec `super` des appels au model prototypal.
- calculer des noms de propriété avec des expressions.

```
const handler = () => true;

var obj = {
    //Définition du prototype : __proto__
    __proto__: {toString:()=>'Hello World !'},
    // Raccourci pour 'handler: handler'
    handler,
    // Simplification syntaxique
    desc() {
        // Appel avec super
        return "Got " + super.toString();
    },
    // propriété calculée
    [ 'code' + (() => 'FR')(): 'France',
      [ 'code' + 'UK' ]: 'England',
    ];
}

console.log(obj);
obj.handler();
obj.desc();
```

Définir un accesseur avec l'opérateur get

La syntaxe `get` permet de lier une propriété d'un objet à une fonction qui sera appelée lorsqu'on accédera à la propriété.

À noter:

- peut être identifié par un nombre ou une chaîne de caractères
- ne doit pas posséder de paramètre
- * l'identifiant ne doit pas être utilisé avec autre propriété

Un accesseur peut être supprimé grâce à l'opérateur `delete`.

```
var o = {
    get dernier() {
        if (this.journal.length > 0) {
            return this.journal[this.journal.length - 1];
        }
        else {
            return null;
        }
    },
    journal: []
}

o.dernier()

delete o.dernier;
```

Utiliser un nom de propriété calculé

```
var expr = "hello";

var obj = {
    get [expr]() { return "Hello World"; }
};

console.log(obj.hello); // "Hello World"
```

Définir un mutateur avec l'opérateur set

La syntaxe `set` permet de lier une propriété d'un objet à une fonction qui sera appelée à chaque tentative de modification de cette propriété.

Il n'est pas possible d'avoir à la fois un mutateur et une valeur donnée pour une même propriété.

À noter:

peut être identifié par un nombre ou une chaîne de caractères

doit avoir exactement un paramètre

* l'identifiant ne doit pas être utilisé avec autre propriété

Pour retirer un mutateur, on peut utiliser l'opérateur delete :

```
var o = {
  set courant (str) {
    this.log= str + ' World';
  },
};

o.courant = 'Hello';
console.log(o);

delete o.courant;
console.log(o);
```

{js}

ES6 apporte une syntaxe fondamentalement nouvelle.

Préservez la lisibilité du code

Les nouveaux types **Set et Map** ne sont pas convertibles au moyen de **JSON.parse**.

Les symbols est un méta-programmation.

L'itération est un méta-comportement sur les objets.

Les "const" assure juste la non réattribution de l'identifiant.

Définissez un chemin de migration.



POO, nouveautés.

5/ POO, nouveautés pour la conception objet.

Modèles de classe et héritage. Méthodes statiques.

Les classes ont été introduites dans JavaScript avec ECMAScript 6 et sont un **sucré syntaxique** de l'héritage prototypal. Le mot clé `class` fournit une syntaxe plus claire pour utiliser un modèle et gérer l'héritage.

Cette syntaxe n'introduit pas un nouveau modèle d'héritage dans JavaScript !

JavaScript est un langage à paradigme multilpe : Orienté Objet (prototypal), impératif et fonctionnel.

ES6 n'introduit pas de paradigme Orienté Objet basé sur les classes

Les classes sont des *fonctions spéciales* de la même façon qu'il y a des expressions de fonctions et des déclarations de fonctions, on aura deux syntaxes :

Déclarations de classes

```
class Polygone {  
    constructor(hauteur, largeur) {  
        this.hauteur = hauteur;  
        this.largeur = largeur;  
    }  
}
```

Expressions de classes

Si on utilise un nom dans l'expression, ce nom ne sera accessible que depuis le corps de la classe.

```
// anonyme
var Polygone = class {
    constructor(hauteur, largeur) {
        this.hauteur = hauteur;
        this.largeur = largeur;
    }
};

// nommée
var Polygone = class Polygone {
    constructor(hauteur, largeur) {
        this.hauteur = hauteur;
        this.largeur = largeur;
    }
};
```

Remontée des déclarations (hoisting)

Les déclarations de classes ne sont pas remontées dans le code, il est nécessaire de d'abord **déclarer la classe avant de l'utiliser**.

```
var p = new Polygone(); // ReferenceError  
  
class Polygone {}
```

Corps d'une classe et définition des méthodes

Le corps d'une classe, partie contenue entre les accolades, définit les propriétés d'une classe comme ses méthodes ou **son constructeur**.

Si la classe contient plusieurs occurrences d'une méthode constructor, cela provoquera une exception SyntaxError.

```
class Polygone {  
    constructor(hauteur, largeur) {  
        this.hauteur = hauteur;  
        this.largeur = largeur;  
    }  
  
    get area() {  
        return this.calcArea();  
    }  
  
    calcArea() {  
        return this.largeur * this.hauteur;  
    }  
}  
  
const carré = new Polygone(10, 10);  
  
console.log(carré.area);
```

À noter Il n'y pas de séparateur entre les membres d'une classe.

get/set Permet l'utilisation de méthodes sous la forme de propriétés.

Méthodes statiques

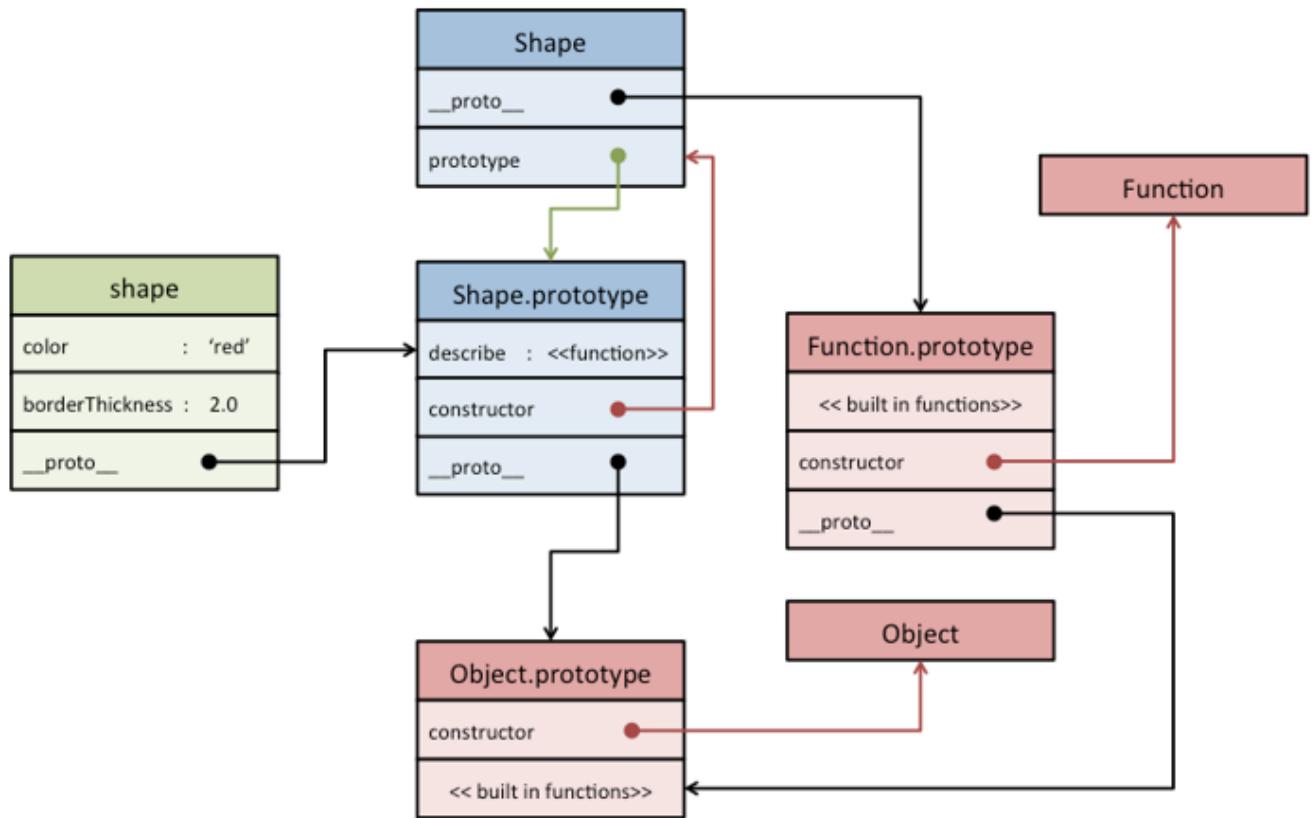
Le mot-clé `static` permet de définir une méthode statique pour une classe.

Les méthodes statiques sont appelées par rapport à la classe entière et non par rapport à une instance donnée.

Ces méthodes sont généralement utilisées sous formes d'utilitaires.

```
class Point {  
    constructor(x, y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    static distance(a, b) {  
        const dx = a.x - b.x;  
        const dy = a.y - b.y;  
  
        return Math.sqrt(dx*dx + dy*dy);  
    }  
}  
  
const p1 = new Point(5, 5);  
const p2 = new Point(10, 10);  
  
console.log(Point.distance(p1, p2));
```

Modèle prototypal



```

class Person {
    constructor(name) {
        this.name = name;
    }
    toString() {
        return `Person named ${this.name}`;
    }
    static logNames(persons) {
        for (const person of persons) {
            console.log(person.name);
        }
    }
}

class Employee extends Person {
    constructor(name, title) {
        super(name);
        this.title = title;
    }
    toString() {
        return `${super.toString()} (${this.title})`;
    }
}

const jane = new Employee('Jane', 'CTO');
console.log(jane.toString()); // Person named Jane (CTO)

```

Héritage de classes

Le mot-clé `extends`, utilisé dans les déclarations ou les expressions de classes, permet de créer une classe qui hérite d'une autre classe (on parle aussi de « sous-classe » ou de « classe-fille »).

Le le mot-clé `super` fait référence de la classe parente.

```
class Animal {
  constructor(nom) {
    this.nom = nom;
  }

  parle() {
    console.log(this.nom + ' fait du bruit.');
  }
}

class Chien extends Animal {
  parle() {
    super.parle();
    console.log(this.nom + ' aboie.');
  }
}
```

Une classe ECMAScript ne peut avoir qu'une seule classe parente.

Créer une classe statique

```
class Manager {  
    constructor() {  
        return Manager;  
    }  
  
    static create(value) {  
        Manager.collection.push(value);  
    }  
  
    static remove(value) {  
        Manager.collection.splice(Manager.collection.indexOf(value),1);  
    }  
}  
Manager.collection = []  
  
Manager.create();
```

Objets natifs héritables.

Avec ES6, les objets natifs tels que Array, Date et DOMElements peuvent être étendus.

La construction d'objet se décompose en deux phases :

1. Appel du constructeur pour allouer l'objet, l'initialisation d'un comportement spécial
2. Invocation du constructeur sur une nouvelle instance pour initialiser

Les types natifs exposent leur constructeur explicitement.

```
// Pseudo-code de Array
class Array {
    constructor(...args) { /* ... */ }
    static [Symbol.create]() {
        // Initialisation[[DefineOwnProperty]]
    }
}

// Héritage de Array
class MyArray extends Array {
    constructor(...args) { super(...args); }
}

// Deux phase 'new':
// 1) Appel @@create pour allouer l'objet
// 2) Création de l'instance
var arr = new MyArray();
arr[1] = 12;
arr.length == 2
```

{js}

class est juste un mot clé donnant plus de lisibilité.

N'attendez pas et n'essayez pas de reproduire le comportement d'un language tel que JAVA.

L'Héritage est simplifié, est-il nécessaire ?.

Set , Map et Symbol permettront une optimisation de la mémoire.



Nouvelles API JavaScript avec ES6.

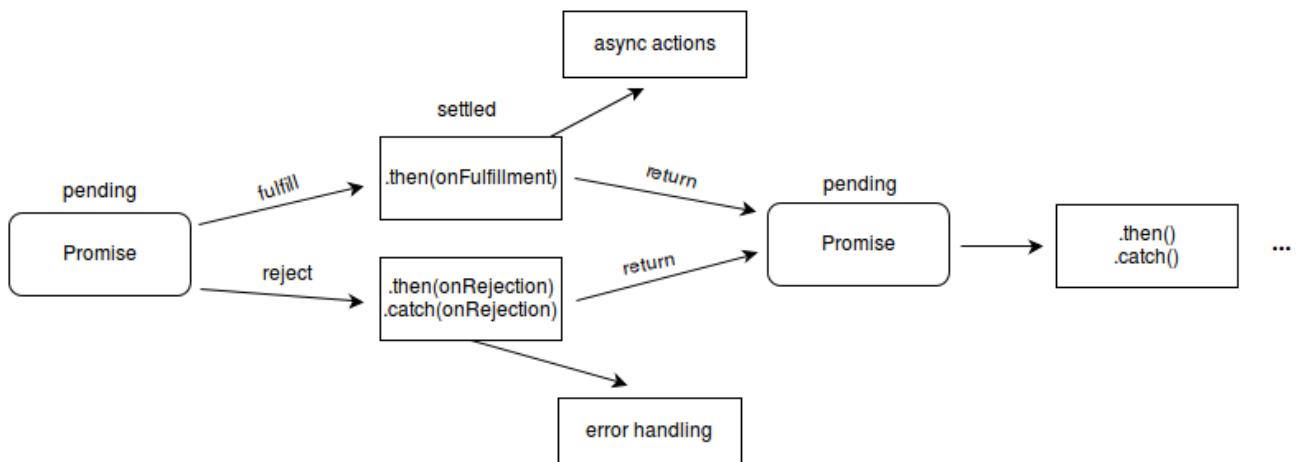
6/ Nouvelles API JavaScript avec ES6.

Promise : gestion des traitements asynchrones.

L'objet Promise (pour « promesse ») est utilisé pour réaliser des opérations de façon asynchrone. Une promesse est dans un de ces états :

- en attente : état initial, la promesse n'est ni remplie, ni rompue
- tenue : l'opération a réussi
- rompue : l'opération a échoué
- acquittée : la promesse est tenue ou rompue mais elle n'est plus en attente.

```
new Promise(function(resolve, reject) { ... });
```



Promise : méthodes.

Promise.all(itérable)

Renvoie une promesse qui est tenue lorsque toutes les promesses de l'argument itérables sont tenues. Si la promesse est tenue, elle est résolue avec un tableau contenant les valeurs de résolution des différentes promesses contenues dans l'itérable.

Promise.race(itérable)

Renvoie une promesse qui est tenue ou rompue dès que l'une des promesses de l'itérable est tenue ou rompue avec la valeur ou la raison correspondante.

Promise.reject(raison)

Renvoie un objet Promise qui est rompu avec la raison donnée.

Promise.resolve(valeur)

Renvoie un objet Promise qui est tenue (résolue) avec la valeur donnée.

Gérer les appels asynchrones

```
const get = (url) => {
  return new Promise((resolve, reject) => {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', url);
    xhr.onload = () => resolve(xhr.responseText);
    xhr.send();
  });
};

/* getTweets (Generator) */

const getTweets = (function* () {
  // 1er
  yield get('https://api.myjson.com/bins/2qjdn');
  // 2nd
  yield get('https://api.myjson.com/bins/3zjqz');
  // 3rd
  yield get('https://api.myjson.com/bins/29e3f');

})();

// Initialisation et différentes consommations

Promise.all([...getTweets]).then((valeur)=> console.log(valeur), (raison) => console.
```

Object API : revisiter les méthodes.

Les Constructeur natifs connus sont enrichis de nouvelle méthodes **static** ou **prototypal**

Object : méthodes statiques

Object.assign

La méthode Object.assign() est utilisée afin de copier les valeurs de toutes les propriétés directes (non héritées) d'un objet qui sont énumérables sur un autre objet cible. Cette méthode renvoie l'objet cible.

```
var o1 = { a: 1 };
var o2 = { [Symbol('Hello')]: 2 };
var o3 = { c: 3 };

var obj = Object.assign(o1, o2, o3);
console.log(obj); // { a: 1, c: 3, [Symbol("Hello")]: 2 }
console.log(o1);
```

Object.is

La méthode Object.is() permet de déterminer si deux valeurs sont les mêmes.

```
Object.is("Hello", "Hello");      // true
Object.is(window, window);       // true

Object.is("Hello", "World");     // false
Object.is([], []);              // false

var test = {a: 1};
Object.is(test, test);          // true

Object.is(null, null);          // true
```

String : méthodes sur le prototype

String.prototype.repeat

La méthode repeat() construit et renvoie une nouvelle chaîne de caractères qui contient le nombre de copie demandée de la chaîne de caractères sur laquelle la méthode a été appelée, concaténées les unes aux autres.

```
"abc".repeat(-1)      // RangeError
"abc".repeat(0)        // ""
"abc".repeat(1)        // "abc"
"abc".repeat(2)        // "abcabc"
"abc".repeat(3.5)      // "abcabcabc" (Le compteur est converti en un nombre entier)
"abc".repeat(1/0)      // RangeError
```

String.prototype.startsWith

La méthode `startsWith()` renvoie un booléen indiquant si la chaîne de caractères commence par la deuxième chaîne de caractères fournie en argument.

```
var str = "Être, ou ne pas être : telle est la question.";  
  
console.log(str.startsWith("Être"));           // true  
console.log(str.startsWith("pas être"));        // false  
console.log(str.startsWith("pas être", 12));    // true
```

String.prototype.endsWith

La méthode `endsWith()` renvoie un booléen indiquant si la chaîne de caractères se termine par la deuxième chaîne de caractères fournie en argument.

```
var str = "Être, ou ne pas être : telle est la question.";  
  
console.log(str.endsWith("question."));         // true  
console.log(str.endsWith("pas être"));          // false  
console.log(str.endsWith("pas être", 20));       // true
```

String.prototype.includes

La méthode `includes()` détermine si une chaîne de caractères est contenue dans une autre et renvoie `true` ou `false` selon le cas de figure.

```
var str = "Être ou ne pas être, telle est la question.";  
  
console.log(str.includes("Être"));           // true  
console.log(str.includes("question"));        // true  
console.log(str.includes("pléonasme"));       // false  
console.log(str.includes("Être", 1));          // false  
console.log(str.includes("ÊTRE"));            // false
```

Array : méthodes statiques

Array.from

La méthode **Array.from()** permet de créer une nouvelle instance d'Array à partir d'un objet itérable ou semblable à un tableau.

Avec ES6, la syntaxe de classe permet d'avoir des sous-classes pour les objets natifs comme pour les objets définis par l'utilisateur. Ainsi, les méthodes statiques de classe comme `Array.from` sont héritées par les sous-classes d'Array et créent de nouvelles instances de la sous-classe d'Array.

Paramètres

arrayLike Un objet semblable à un tableau ou bien un objet itérable dont on souhaite créer un tableau, instance d'Array.

fonctionMap Argument optionnel, une fonction à appliquer à chacun des éléments du tableau.

thisArg Argument optionnel. La valeur à utiliser pour `this` lors de l'exécution de la fonction `fonctionMap`.

```
// créer une instance d'Array à partir de l'objet arguments qui est semblable à un tableau
function foo() {
  return Array.from(arguments);
}

foo(1, 2, 3); // [1, 2, 3]

// Ça fonctionne avec tous les objets itérables...
var s = new Set(["toto", window]);
Array.from(s); // ["toto", window]

var m = new Map([[1, 2], [2, 4], [4, 8]]);
Array.from(m); // [[1, 2], [2, 4], [4, 8]]

Array.from("toto"); // ["t", "o", "t", "o"]

// En utilisant une fonction fléchée pour remplacer map
Array.from([1, 2, 3], x => x + x); // [2, 4, 6]

// Pour générer une séquence de nombres
Array.from({length: 5}, (v, k) => k); // [0, 1, 2, 3, 4]
```

Array.of

La méthode `Array.of()` permet de créer une nouvelle instance d'objet `Array` avec un nombre variable d'argument, quels que soient leur nombre ou leur type.

La différence avec la méthode `Array.of()` et le constructeur `Array` se situe dans la façon de gérer les arguments entiers. Ainsi `Array.of(42)` créera un tableau avec un seul élément (42) alors que `Array(42)` créera un tableau contenant 42 éléments, chacun valant

```
Array.of(1);          // [1]
Array.of(1, 2, 3);   // [1, 2, 3]
Array.of(undefined); // [undefined]
```

Array : méthodes sur le prototype

Array.prototype.copyWithin

La méthode `copyWithin()` permet de copier une suite d'éléments à l'intérieur d'un tableau à partir d'un indice cible. L'argument fin est facultatif et sa valeur par défaut correspond à la taille du tableau.

`copyWithin` est une méthode qui modifie l'objet courant. En effet, elle modifiera l'objet `this` avant de le renvoyer et ne créera pas un tableau à part pour le résultat

```
[1, 2, 3, 4, 5].copyWithin(0, 3); // [4, 5, 3, 4, 5]
[1, 2, 3, 4, 5].copyWithin(0, 3, 4); // [4, 2, 3, 4, 5]
[1, 2, 3, 4, 5].copyWithin(0, -2, -1); // [4, 2, 3, 4, 5]
[].copyWithin.call({length: 5, 3: 1}, 0, 3); // {0: 1, 3: 1, length: 5}
```

Array.prototype.find

La méthode `find()` renvoie une valeur contenue dans le tableau si un élément du tableau respecte une condition donnée par la fonction de test passée en argument. Sinon, la valeur `undefined` est renvoyée.

La méthode `find` exécute la fonction `callback` une fois pour chaque élément présent dans le tableau jusqu'à ce qu'elle retourne une valeur vraie (qui peut être convertie en `true`). Si un élément est trouvé, `find` retourne immédiatement la valeur de l'élément.

find ne modifie pas le tableau à partir duquel elle est appelée.

Array.prototype.findIndex

La méthode `findIndex()` renvoie l'indice d'un élément du tableau qui satisfait une condition donnée par une fonction. Si la fonction renvoie faux pour tous les éléments du tableau, le résultat vaut `-1`.

```
var inventaire = [
  {nom: 'pommes', quantité: 2},
  {nom: 'bananes', quantité: 0},
  {nom: 'cerises', quantité: 5},
];

function trouveCerises(fruit) {
  return fruit.nom === 'cerises';
}

console.log(inventaire.find(trouveCerises));
console.log(inventaire.findIndex(trouveCerises));
```

Array.prototype.fill

La méthode `fill()` remplit tout les éléments d'un tableau entre deux index avec une valeur statique.

```
arr.fill(valeur[, début = 0[, fin = this.length]])
```

```
[1, 2, 3].fill(4);           // [4, 4, 4]
[1, 2, 3].fill(4, 1);       // [1, 4, 4]
[1, 2, 3].fill(4, 1, 2);    // [1, 4, 3]
[1, 2, 3].fill(4, 1, 1);    // [1, 2, 3]
[1, 2, 3].fill(4, -3, -2);  // [4, 2, 3]
[1, 2, 3].fill(4, NaN, NaN); // [1, 2, 3]
Array(3).fill(4);           // [4, 4, 4]
[].fill.call({length: 3}, 4); // {0: 4, 1: 4, 2: 4, length: 3}
```

Array.prototype.keys

La méthode **keys()** renvoie un nouveau **Array Iterator** qui contient les clefs pour chaque indice du tableau.

```
var arr = ["a", "b", "c"];
var itérateur = arr.keys();

console.log(itérateur.next()); // { value: 0, done: false }
console.log(itérateur.next()); // { value: 1, done: false }
console.log(itérateur.next()); // { value: 2, done: false }
console.log(itérateur.next()); // { value: undefined, done: true }

var arr = ["a", , "c"];
var clésCreuses = Object.keys(arr);
var clésDenses = [...arr.keys()];
console.log(clésCreuses); // ["0", "2"]
console.log(clésDenses); // [0, 1, 2]
```

Array.prototype.values

La méthode **values()** renvoie un nouvel objet **Array Iterator** qui contient les valeurs pour chaque indice du tableau.

```
var arr = ['w', 'y', 'k', 'o', 'p'];
var eArr = arr.values();
// votre navigateur doit supporter les boucles for..of
// et les variables définies avec let
for (let lettre of eArr) {
  console.log(lettre);
}

var arr = ['w', 'y', 'k', 'o', 'p'];
var eArr = arr.values();
console.log(eArr.next().value); // w
console.log(eArr.next().value); // y
console.log(eArr.next().value); // k
console.log(eArr.next().value); // o
console.log(eArr.next().value); // p
```

Array.prototype.entries

La méthode **entries()** renvoie un nouvel objet de type **Array Iterator** qui contient le couple clef/valeur pour chaque élément du tableau.

```
var arr = ["a", "b", "c"];
var eArr = arr.entries();

console.log(eArr.next().value); // [0, "a"]
console.log(eArr.next().value); // [1, "b"]
console.log(eArr.next().value); // [2, "c"]
```

{js}

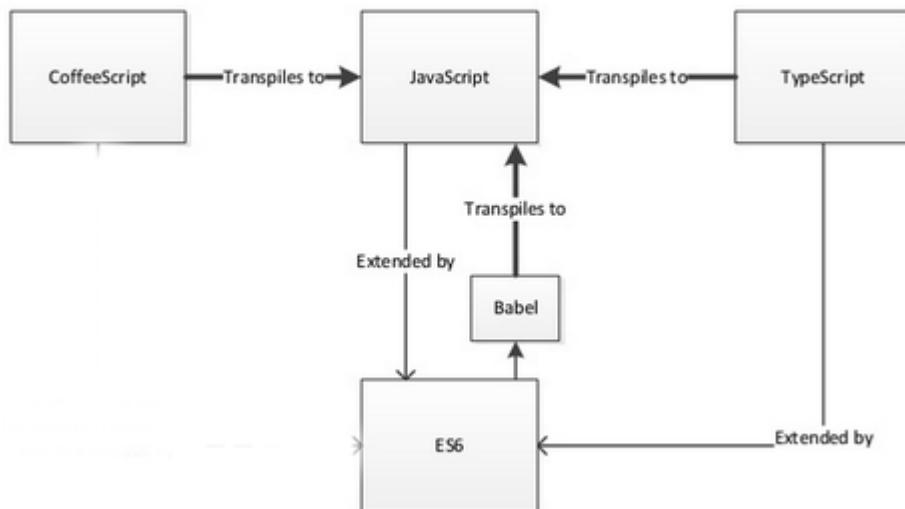
Les Promises apportent une solution à l'organisation de code asynchrone.

Les nouvelles Méthodes de Array et Object sont optimisées.

Outils indispensables. Babel, Traceur et Typescript.

À date (2016) le système natif de chargement n'est pas implémenté dans les navigateurs.

De plus les navigateurs ne supportent pas uniformément la syntaxe ES6.



Pour anticiper la migration de code et bénéficier de la syntaxe ES6 il faut choisir un "transpiler" et un système de chargement de module.

[es6-module-loader](#) est un polyfill utilisable en production.

Choix du “transpiler” : présentation des solutions.

Il existe beaucoup de “transpiler” capables de convertir du code ES6 en ES5

Références

[Babel](#)

[Traceur compiler](#)

* [TypeScript](#)

À noter Ces trois références principales existent sous la forme de tâches pour **gulp** et **grunt**

Avec browserify

- [es6ify](#)
- [babelify](#)
- [es6-transpiler](#)

Modules

- Square’s [es6-module-transpiler](#)

Autres

Facebook’s [regenerator](#)

Facebook’s [jstransform](#)

[Et encore bien d’autres](#)

Mise en oeuvre de TypeScript.

Installation

```
$> npm init  
$> npm install --global typescript traceur jspm yo gulp-cli  
$> npm i -g generator-traceur-gulp generator-babel generator-jspm-es6-quick generator
```

TypeScript.

Niveau de support 60%



Mise en oeuvre

```
tsc file.ts  
tsc --watch
```

TypeScript : offre en plus de la syntaxe ES6, le support de type, d'interface...
[référence](#)

Configuration tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "out": "../../built/local/tsc.js",
    "sourceMap": true
  },
  "exclude": [
    "node_modules",
    "wwwroot"
  ]
}
```

ES6 approche modulaire.

Système natif de gestion des modules.

Avec le système natif le code du module est traité différemment pour gérer les exportations et les importations.

```
<script type = "module">
```

est introduit pour distinguer le code de script définissant des modules.

Les conventions de nommagent autorise les noms de modules à utiliser des url.

```
import $ from 'jquery'  
//équivaut à  
import $ from 'https://code.jquery.com/jquery.js'
```

Export :

L'instruction export est utilisée pour permettre d'exporter des fonctions et objets ou des valeurs primitives à partir d'un fichier (ou module) donné.

La syntaxe ES6 est similaire à celle des module CJS

```
export function someMethod() {  
}  
  
export var another = {};  
  
class Module {  
  
    constructor(args='cool') {  
        console.warn(args);  
        this.name = args;  
    }  
    done(){  
        console.info(this.name);  
    }  
}  
  
//alias  
export {Module as User}
```

À noter le nom de la variable ou fonction est réutilisé comme nom d'export.

Mais il est possible de redéfinir des alias

Consommation (Importation) :

L'instruction import est utilisée pour importer des fonctions, des objets ou des valeurs primitives exportées depuis un module externe ou un autre script.

L'import peut se faire par nom et/ou en redéfinissant des alias

```
import { someMethod, another as newName } from './exporter';

someMethod();
typeof newName == 'object';
```

Le chemin de fichier ne nécessite pas l'extension

Import/Export par défaut :

Il est possible de définir par défaut un export simplifié

```
//export-default.js:
export default function foo() {
  console.log('foo');
}
```

```
//import-default.js:
import customName from './export-default';

customName(); // -> 'foo'
```

Variation de la syntaxe :

```
import 'jquery';                                // importe un module
import $ from 'jquery';                         // importe l'export par défaut
import { $ } from 'jquery';                      // importe un export nommé $
import { $ as jQuery } from 'jquery';            // importe un export nommé $ dans l'alias jQuery

export var x = 42;                               // exporte une variable
export function foo() {};                        // export une fonction

export default 42;                              // exporte par défaut de variable
export default function foo() {};                // exporte par défaut de function

export { encrypt };                            // exporte la variable existante encrypt
export { decrypt as dec };                    // exporte la variable existante encrypt sous le nom dec
export { encrypt as en } from 'crypto';         // exporte la variable existante encrypt du module crypto
export * from 'crypto';                        // exporte tous les exports nommés du module crypto
                                                // (sauf l'export par défaut)
import * as crypto from 'crypto';              // importe tous les exports nommés du module crypto
```

“Modules Loaders” : SystemJS, “import/export”.

Un module est simplement un fichier JavaScript exportant des valeurs, qui peuvent ensuite être importés par d'autres modules.

Un module loader permet de charger dynamiquement des modules, et assure également le suivi de tous les modules chargés dans un registre de modules.

Asynchronous Module Definition ou CommonJS.

Le besoin de modularisation et d'encapsulation logique à trouver plusieurs réponses techniques, bien avant la proposition standardisée en ES6.

On distingue

Global Module global.

CJS Common JavaScript Module : implémenté par node.js

AMD Asynchronous Module Définition JavaScript Module : implémenté par node.js

System La proposition ES6

UMD Universal Module Définition

Module global

Exposition d'une API dans l'espace global.

```
(function IIFE(context){
  context.API = {
    value:true
    logic:logic
  }
  function logic(){
    }
})(this);
```

Common JavaScript Module

Implémenté par node.js. Synchrone. Portable côté navigateur avec **browserify**
Export nommés.

CommonJS

S'appuie sur l'implémentation de l'objet **module.exports** et de la méthode **require()**

Export :

```
module.exports.foo = function(){
  console.log('foo')
}
```

Consommation :

```
var foo = require('foo')

foo();
```

Asynchronous Module Définition

Reprend la logique de modules CJS en ajoutant la dimension asynchrone (réseau)
Implémentation connue: requirejs

AMD

S'appuie sur méthode de définition **define()** et de résolution **require()** de dépendances exécuter par un **loader**

Export :

```
define('myModule', ['dep1', 'dep2'], function (dep1, dep2) {  
  
    //Export logic  
    return function () {};  
});
```

Consommation :

```
// main.js  
requirejs(['jquery', 'myModule'],  
    function ($, myModule) {  
        //jQuery, canvas and the app/sub module are all  
        //loaded and can be used here now.  
    }  
);
```

La librairie require.js (ou le loader choisi) résout les chargement.

Après son propre chargement **require.js** charge le point d'entrée défini par son attribut **data-main**

```
<script data-main="scripts/main.js" src="scripts/require.js"></script>
```

Malheureusement cette solution demande beaucoup de configuration.

Universal Module Définition

JavaScript Pattern vise à permettre la compatibilité entre les différentes définitions de modules. (AMD,CJS...)

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD
        define(['jquery'], factory);
    } else if (typeof exports === 'object') {
        // Node, CommonJS-Like
        module.exports = factory(require('jquery'));
    } else {
        // Browser globals (root is window)
        root.returnExports = factory(root.jQuery);
    }
})(this, function ($) {
    // methods
    function myFunc(){}
    // exposed public method
    return myFunc;
}));
```

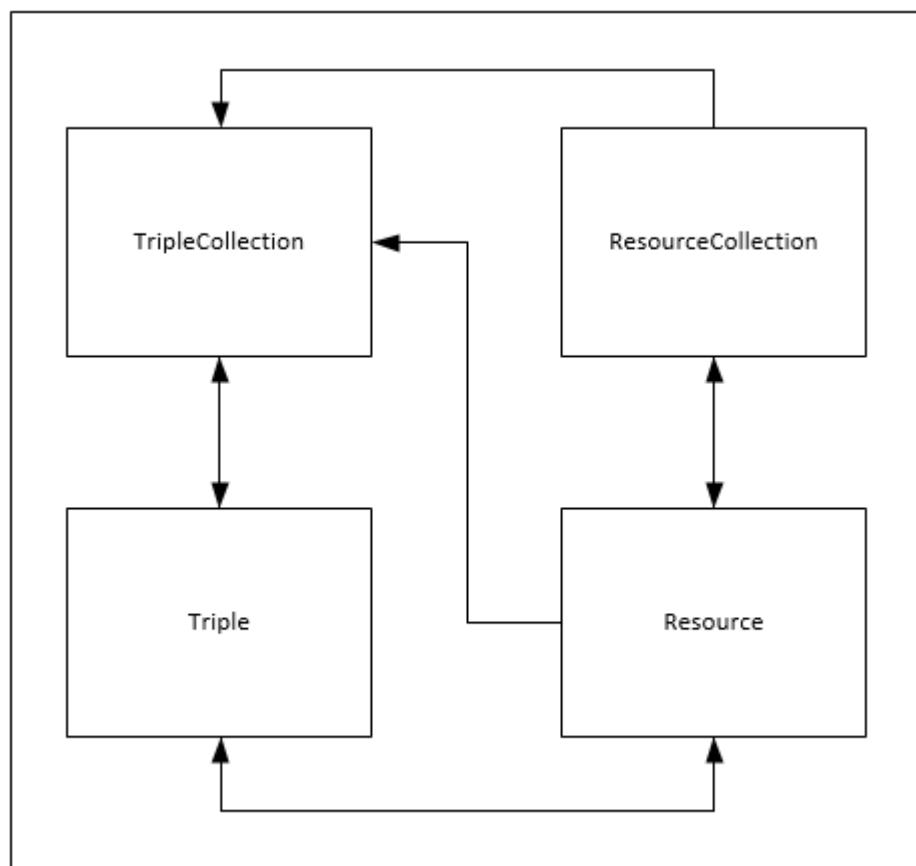
Inconvénient qui ajoute beaucoup de **overhead**

[SystemJS](#) est une élégante solution de chargement supportant les Modules ES6.

Gestion et résolution des dépendances.

AMD, CommonJS et ES6 traitent les dépendances circulaires différemment. Cela implique l'analyse de l'arbre de dépendance et des couches alternées de modules ES6 / non-ES6 avec des références circulaires dans chaque couche pour lier. Les couches sont ensuite reliées individuellement, avec le traitement de la référence circulaire appropriée qui se fait dans chaque couche.

Ceci permet à des références CommonJS circulaires d'interagir avec des références circulaires ES6.



Chargement dynamique.

Le Module Loader ES6 va chercher la source, déterminer les dépendances, et attendre jusqu'à ce que ces dépendances soient chargées avant d'exécuter le module.

Le chargement CommonJS via un module de ES6, s'appuie sur l'analyse statique des déclarations, et est seulement exécuté une fois les dépendances chargées.

Cette cohabitation exclue les imports dynamiques CJS

```
if (condition) require('some' + 'name') ;
var mod = (condition)? require('moduleA') : require('moduleB') ;
```

ES6 imports are declarative and meant for static analysis. They cannot be dynamic

{js}

Choisissez un système de module cohérent : CJS/ES6.

Utilisez un wrapper (browserify) plutôt qu'une implémentation UMD manuelle.

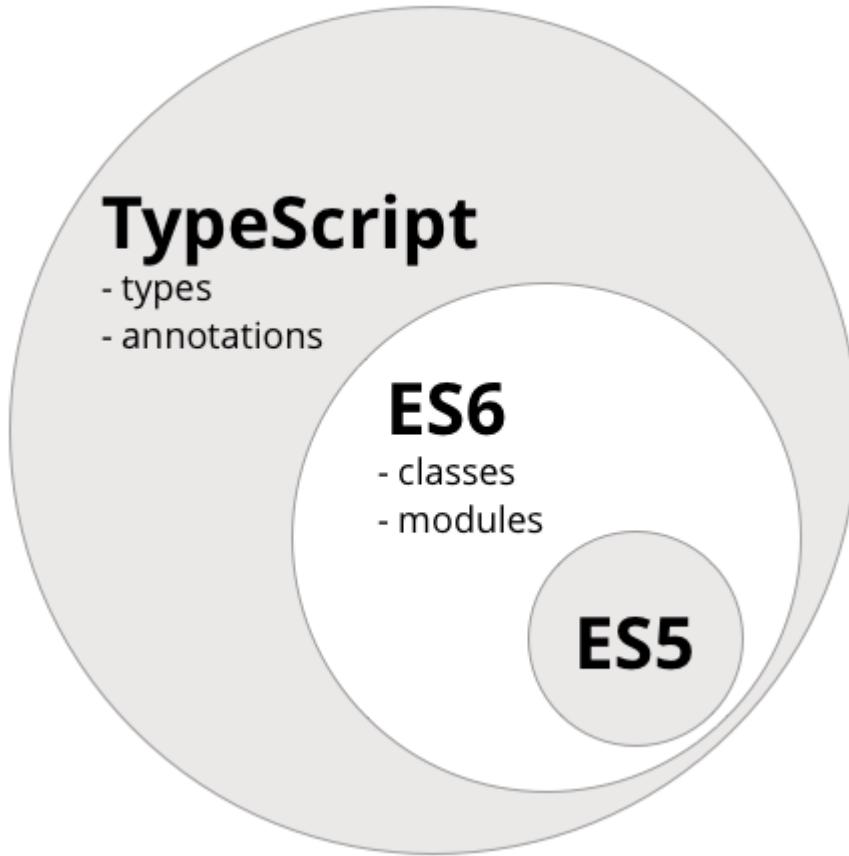
JSPM est un très bon “Module Loader” (à date 03/2016).

Définissez une stratégie : bundle, modulaire ou mixte.

7/ Typescript en détail, configuration.

TypeScript est un langage facilitant le développement d'applications larges et "scalables", écrites en JavaScript.

TypeScript apporte un [support supérieur à 58%](#) des nouveautés syntaxique ES6/2015



Le **transpiler** TypeScript permet l'ajout de concepts *propre* à la programmation orientée objet par **classe** tels que :

- Les interfaces
- Les génériques
- Le typage statique
- Décorateurs

C'est une surcouche de JavaScript : tout le code JavaScript est valide en TypeScript ce qui permet de l'ajouter de façon transparente à n'importe quel projet.

Parceque le code TypeScript est *transcomplié* en JavaScript en tant que language cible on le qualifie de [Superset JavaScript](#).

Le [Playground](#) permet de tester la syntaxe TypeScript en illustrant le mécanisme de transpilation.

12.1/ Installation et mise en oeuvre.

Les deux principaux mode d'installation de TypeScript :

1. En autonomie sur le socle NodeJS via npm
2. Par [plugin selon votre éditeur](#)

Installation avec NPM

```
npm install -g typescript
```

Premier Script

A noter les fichier typescript utilisent l'extension .ts

```
// user.ts

class User {
    userName:string;

    constructor(name:string){
        this.userName = name;
    }
}

let currentUser = new User('Linus');
```

Compilation

The screenshot shows the TypeScript playground interface. On the left, the TypeScript code for `user.ts` is displayed:

```
1 // user.ts
2
3 class User {
4     userName:string;
5
6     constructor(name:string){
7         this.userName = name;
8     }
9
10 }
11
12 let currentUser = new User('Linus');
```

On the right, the generated `user.js` code is shown:

```
1 // user.ts
2 var User = (function () {
3     function User(name) {
4         this.userName = name;
5     }
6     return User;
7 })();
8 var currentUser = new User('Linus');
9
```

At the top, there are buttons for "Select...", "TypeScript", "Share", "Run" (which is highlighted in blue), and "JavaScript".

```
tsc user.ts
```

12.2/ Option du compilateur.

```
tsc -h
```

La commande **tsc -help** donne accès à la liste des [options possibles](#)

Principales options de compilation:

Option	Type	Default	Description
--declaration -d	boolean	false	Génère le fichier de définition correspondant ‘.d.ts’.
-- declarationDir	string		Répertoire de génération des fichiers de définition ‘.d.ts’.
--help -h			Print Affiche l'aide.
--init			Initialise un projet TypeScript en créant un fichier tsconfig.json .
--lib	string[]	<ul style="list-style-type: none"> ► --target ES5 : dom,es5,scripthost ► --target ES6 : dom, es6, dom.iterable, scripthost 	<p>Liste des librairies à inclure. Valeurs possible:</p> <ul style="list-style-type: none"> ► es5 ► es6 ► es2015 ► es7 ► es2016 ► es2017 dom webworker scripthost ► es2015.core ► es2015.collection ► es2015.generator ► es2015 iterable ► es2015.promise ► es2015.proxy ► es2015.reflect ► es2015.symbol ► es2015.symbol.wellknown ► es2016.array.include ► es2017.object ► es2017.sharedmemory
-- listEmittedFiles	boolean	false	Affiche les noms des fichiers générés.
--listFiles	boolean	false	Affiche les noms des fichiers sources.

Option	Type	Default	Description
--module -m	string	target === 'ES6' ? 'ES6' : 'commonjs'	Mode d'encapsulation des modules: 'none' , 'commonjs' , 'amd' , 'system' , 'umd' , 'es6' , or 'es2015' .
-- moduleResolution	string	module === 'amd' 'system' 'ES6' ? 'classic' : 'node'	Mode de résolution des modules.
--outFile	string		Concaténation vers un fichier unique.
--project -p	string		Compile selon la définition de projet d'un fichier tsconfig.json file. Voir tsconfig.json .
--version -v			Affiche la version de TypeScript
--watch -w			Compilation automatique à la détection de changement dans les fichiers.

Créer un fichier de configuration

```
tsc --init
```

La commande d'initialisation permet de générer simplement le fichier `tsconfig.json`

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false
  }
}
```

A noter : Pour un projet JavaScipt on préférera les options suivante.

```
rm tsconfig.json & tsc --init --moduleResolution node --target ES5 --sourceMap --modu
```

Stratégie de résolution des module

```
tsc --init --moduleResolution node
```

Dans la commande précédente l'option `--moduleResolution node` stipule le [mode de résolution](#) pour les importations **non relatives**

Une importation relative se réfère au chemin du répertoire courant

```
import Entry from "./components/Entry";
import { DefaultHeaders } from "../constants/http";
import "/mod";
```

Autrement l'importation doit être résolue selon une règle globale.

```
import * as $ from "jQuery";
import { Component } from "angular2/core";
```

La valeur **node** utilise la résolution de [module selon NodeJS](#)

Cette option permet l'adoption de npm comme gestionnaire de dépendances.

Il est possible de “mapper” des chemins spécifique dans le fichier de configuration.

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false,
    "paths": {
      "jquery": ["node_modules/jquery/dist/jquery.d.ts"]
    }
  }
}
```

Apports Syntaxiques

L'usage de TypeScript renforce la syntaxe JavaScript :

- Support (partiel) de la syntaxe ES6/2015
- Typage fort (variable, fonction, types génériques)
- Enrichissement des “patterns OO” (class, interface, héritage)
- Modularité (namespace)

Grâce à ces enrichissement et sa phase de compilation TypeScript offre une plus grande prédictibilité du code.

Typage des variables.

Declaration Type	Namespace	Type	Value
Namespace	X		X
Class		X	X
Enum		X	X
Interface		X	
Type Alias		X	
Function			X
Variable			X

TypeScript propose des types basiques :

- Boolean
- Number
- String
- Array
- Tuple
- Enum
- Any
- Void
- Null et Undefined
- Never

Les 3 premiers sont des représentations standards JavaScript.

```
//boolean
let complete: boolean = false;
//number
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
//string
let color: string = "blue";
```

Array

Les tableaux peuvent être annoté de deux façons :

- Type des éléments suivis de []
- Notation générique **Array<elemType>**

```
//array
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];
```

Tuple

Les **tuple** sont l'expression d'un **array** dont le type des éléments est connu bien potentiellement différents.

```
// Declare un tuple
let x: [string, number];
// Initialisation
x = ["hello", 10]; // OK
// Error d'initialisation
x = [10, "hello"]; // Error
```

Enum

Les **enum** ou un type énuméré est un type de données qui consiste en un ensemble de constantes indiquées.

```
enum Color {Red, Green, Blue};  
let c: Color = Color.Green;
```

Any et Void, Null et Undefined et Never

any tous les types acceptés.

```
let notSure: any;  
notSure = "maybe a string instead";  
notSure = false; // okay, definitely a boolean  
  
let list: any[] = [1, true, "free"];  
  
list[1] = 100;
```

Les types **any** et **void** apportent un comportement opposé.

void aucun type accepté.

```
function warnUser(): void {  
    alert("This is my warning message");  
}
```

Null et **Undefined** représentation peu utiles.

never particulièrement utilisé pour les fonctions propageant une erreur ou sans valeur de retour.

```
// Function returning never must have unreachable end point
function error(message: string): never {
    throw new Error(message);
}

// Inferred return type is never
function fail() {
    return error("Something failed");
}
```

Inférence de type.

L'[inférence de types](#) est un mécanisme qui permet à un compilateur de rechercher automatiquement les types associés à des expressions, sans qu'ils soient indiqués explicitement dans le code source.

Il s'agit pour le compilateur ou l'interpréteur de trouver le type le plus général que puisse prendre l'expression.

```
let x = 3; // Type Inferred to number
x = ''; // Error : Type 'string' is not assignable to type 'number'.
```

Best common type lorsque l'inférence repose sur plusieurs types la représentation la plus commune est recherchée.

```
let x = [0, 1, null];
x.push('');// Error : Argument of type 'string' is not assignable to parameter of type 'number'

let zoo = [new Rhino(), new Elephant(), new Snake()];
let zoo: Animal[] = [new Rhino(), new Elephant(), new Snake()];
```

Si aucun type ne peut être déterminé l'inférence sera résolue sur `object **{}**`

Erreur de contexte :

```
window.onmousedown = function(mouseEvent) {
    console.log(mouseEvent.button); //<- Error
};

//versus

window.onmousedown = function(mouseEvent: any) {
    console.log(mouseEvent.button); //<- Now, no error is given
};
```

Assertion de type.

L'assertion de type ou `Casting de variable` permet de renseigner le compilateur selon deux syntaxes.

```
let someValue: any = "this is a string";  
  
let strLength: number = (<string>someValue).length;
```

Ou préférablement :

```
let someValue: any = "this is a string";  
  
let strLength: number = (someValue as string).length;
```

Typage des fonction.

Les [définitions de fonctions](#) reçoivent également des information de types sur :

- Les arguments.
- Les arguments optionnels.
- Les valeurs de paramètres par défaut.
- Les valeurs de retour.

```
function add(x: number, y?: number = 1 ): number {  
    return x + y;  
}
```

Surcharge de fonction.

Puisqu 'il est possible d'exprimer un typage fort dans les signature de fonction, on peut utiliser cette différentiation pour **surcharger la définition de fonction**

```
let suits = ["hearts", "spades", "clubs", "diamonds"];  
  
function pickCard(x: {suit: string; card: number;}[]): number;  
function pickCard(x: number): {suit: string; card: number;};  
function pickCard(x): any {  
  
    if (typeof x == "object") return Math.floor(Math.random() * x.length);  
  
    if (typeof x == "number") return { suit: suits[Math.floor(x / 13)], card: x % 13 }  
}  
  
let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { suit: "he  
let pickedCard1 = myDeck[pickCard(myDeck)];  
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);  
  
let pickedCard2 = pickCard(15);  
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

Patterns de programmation orientée objet.

Enrichissement des `class`.

TypeScript enrichie la syntaxe **ES6/2015**

- Interface
- Déclaration de propriétés

Interface.

En programmation orientée objet, une interface décrit à minima les méthodes accessibles depuis l'extérieur de la classe, par lesquelles on peut modifier l'objet.

Interfaces Typecript en détail

il est commun à toutes les Interfaces de déclarer chacune de ses méthodes sous la forme d'une signature :

```
nomDeFonction (argument1:Type, ...):typeRetour;
```

Pour rappel la différenciation **publique/privée** ou portée de variable permet :

- d'éviter de manipuler l'objet de façon indue, en limitant ses modifications à celles autorisées comme publiques par le concepteur de la classe
- à ce concepteur, de modifier l'implémentation interne de ces méthodes de manière transparente.

De part le typage structurel dynamique ou "["Duck Typing"](#)" de TypeScript, les interfaces peuvent également **décrire la structure d'un objet en incluant les propriétés.**

```
interface User {  
    connect(credential: string): boolean;  
    name: string;  
    email: string;  
    score: number;  
}
```

Bien qu'il soit parfois d'usage de préfixer une interface d'un I majuscule cet usage n'est pas recommandé. (voir Best Practices)

Une fonction pourra par exemple utiliser l'interface pour vérifiée la structure de l'objet à manipuler :

```
function increaseUserScore(user:User):boolean{  
    if(!user.connect('12345678')) return false;  
    user.score += 1;  
}
```

"Duck Typing" Le type dans le contexte où il est utilisé, est déterminée par l'ensemble de ses méthodes et de ses attributs.

Alternativement, il est possible de décrire la structure attendue dans la signature de fonction :

```
function increaseUserScore(user:{  
    connect(credential: string): boolean;  
    name: string;  
    email: string;  
    score:number;  
} ):boolean{  
    if(!user.connect('12345678')) return false;  
    user.score += 1;  
}
```

Avec la même notation que les paramètres de fonction certains membres peuvent être rendus optionnels

```
interface User {  
    connect(credential: string): boolean;  
    disconnect?(): boolean; //Optionnel  
    name: string;  
    email?: string; //Optionnel  
    score:number;  
}
```

readonly identifie les propriétés modifiables uniquement par le constructeur.

TypeScript implémente les **ReadonlyArray<T>** (Array sans mutateurs)

```
interface User {  
    connect(credential: string): boolean;  
    readonly name: string;  
    readonly email?: string; //Optionnel  
    score:number;  
}
```

Utiliser **const** pour les variables et **readonly** pour les propriétés.s

Les sont **interface** utilisables pour les `casting` :

```
let me = {connect(credential){return true;},name:'Me',score:0} as User;
```

Les interfaces peuvent également décrire des type de fonction ou de collections.

```
interface User {
  readonly name: string;
  readonly email?: string; //Optionnel
  score:number;
}

class Player implements User{}
```

Class.

Les class peuvent implémenter une ou plusieurs interface(s).

A noter une class peut également servir d'interface.

```
interface User {
    name: string;
    email?: string; //Optionnel
    score:number;
}

interface AppUser {
    ...
}

class AdminUser implements User, AppUser {
    ...
}
```

Héritage.

Une classe ou une interface peuvent en étendre d'autre(s).

```
interface Shape {  
    color: string;  
}  
  
interface PenStroke {  
    penWidth: number;  
}  
  
interface Square extends Shape, PenStroke {  
    sideLength: number;  
}
```

On notera qu'une interface peut également étendre une classe.

```
class Control {  
    private state: any;  
}  
  
interface SelectableControl extends Control {  
    select(): void;  
}  
  
class Button extends Control {  
    select() { }  
}
```

Accesseur de propriétés.

TypeScript offre un espace déclaratif pour les propriétés d'une classe.

Par défaut toutes les propriétés et méthodes sont **public** mais il est possible de préciser leur visibilité avec les mots clés **public** et **private** ou **protected**.

Un membre **private** n'est qu'accessible à l'intérieur de la classe qui le contient.

Un membre **public** est visible partout.

```
class Animal {  
    //Espace déclaratif  
    private name: string;  
    public constructor(theName: string) { this.name = theName; }  
}  
  
new Animal("Cat").name; // Error: 'name' is private;
```

Les membres **protected** ont un accès similaire à **private** à la différence qu'ils sont également accessibles depuis les classes dérivées.

```
class Person {  
    protected name: string;  
    constructor(name: string) { this.name = name; }  
}  
  
class Employee extends Person {  
    private department: string;  
  
    constructor(name: string, department: string) {  
        super(name);  
        this.department = department;  
    }  
  
    public getElevatorPitch() {  
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;  
    }  
}  
  
let howard = new Employee("Howard", "Sales");  
console.log(howard.getElevatorPitch());  
console.log(howard.name); // error
```

Paramètres de propriétés.

Ils offrent une simplification du code par l'initialisation des propriétés.

```
class Octopus {
    readonly number_of_legs: number = 8;
    constructor(readonly name: string) {
    }
}
```

Classe Abstraites.

Une classe abstraite est une classe dont l'implémentation n'est pas complète et qui n'est pas instanciable. Elle sert de base à d'autres classes dérivées (héritées).

Les méthodes marquées **abstract** ont la même signature que dans un interface.

```
abstract class Animal {
    abstract makeSound(): void;
    move(): void {
        console.log("roaming the earth...");
    }
}
```

Modularité.

Les [namespace\(s\) TypeScript](#) sont un excellent moyen de réutiliser dans le code source d'une application du code indépendant.

- Inclure des fichier utilitaire (validation...)
- Inclure des fichier de **définition strucurale** de code javascript **.d.ts**

Inclusion de fichiers utilitaire*

Les modules ES6/2015 ne devrait pas utiliser de référence.

```
//validation.ts
namespace Validation {

    const lettersRegexp = /^[A-Za-z]+$/;
    export class LettersOnlyValidator {
        static isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }

    const numbersRegexp = /^[0-9]+$/;
    export class NumbersOnlyValidator {
        static isAcceptable(s: string) {
            return numbersRegexp.test(s);
        }
    }
}
```

Utilisation

```
//main.ts
/// <reference path="../validation.ts" />
/// <reference path="../node_modules/systemjs/dist/system.ts" />

console.log(Validation.LettersOnlyValidator);
console.log(System);
```

Déclaration

Elles permettent la prévention de message d'erreur en cas d'utilisation implicites.

```
foo = 123; // Error: `foo` is not defined
```

```
declare var foo:any;
foo = 123; // allowed
```

L'option de compilation `--declaration` permet générer le fichier de déclaration d'un code correspondant.

```
//code.ts

class User {
    public name;
    constructor(name) {
        this.name = name;
    }

    greet(){
        console.log('hi' + name);
    }
}
```

```
tsc code.ts --declaration > code.d.ts
```

Génération du fichier de sortie **code.d.ts**

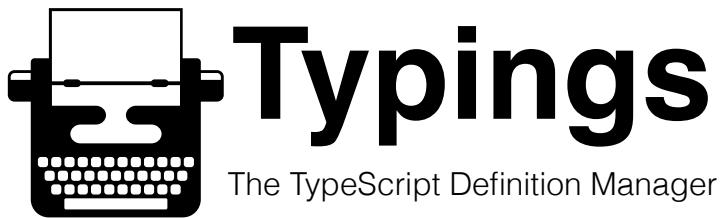
```
declare class User {
    name: any;
    constructor(name: any);
    greet(): void;
}
```

Un autre fichier pourra alors utiliser cette déclaration.

```
//main.ts  
/// <reference path="code.d.ts" />
```

Un fichier particulier **lib.d.ts** est installé avec TypeScript, ce fichier contient les déclaration de librairies javascript commune.

definitelytyped.org centralise un grand nombre de fichiers de **définition** pour l'usage des bibliothèque les plus répandues avec TypeScript.



[Typings](#) est un utilitaire de dépendance pour les fichier **.d.ts** utilisant un fichier **typings.json**

```
npm install typings --global  
typings init  
typings search core-js  
typings i dt~core-js --save --global
```

A noter : Les définition angular sont livrées avec le package npm.

```
npm i angular2
```

TypeScript Best Practices.

Nomenclature

1. Utiliser le PascalCase pour les noms de types.
2. Ne pas préfixer les **interfaces avec "I"**.
3. Utiliser le PascalCase pour les **enum**.
4. Utiliser le camelCase pour les fonction, proriété et variables.
5. Ne pas préfixer les **membres privés de “_”**.

Composants

1. 1 fichier par composant logique.

`null` and `undefined`

1. Use **undefined**, do not use null.

Commentaires

1. Utiliser des commentaires **JSDoc**.

Style

1. Utiliser des préférence les **arrow function**.
2. Ne délimiter les paramètres que si nécessaire.

Exemple, `(x) => x + x` :

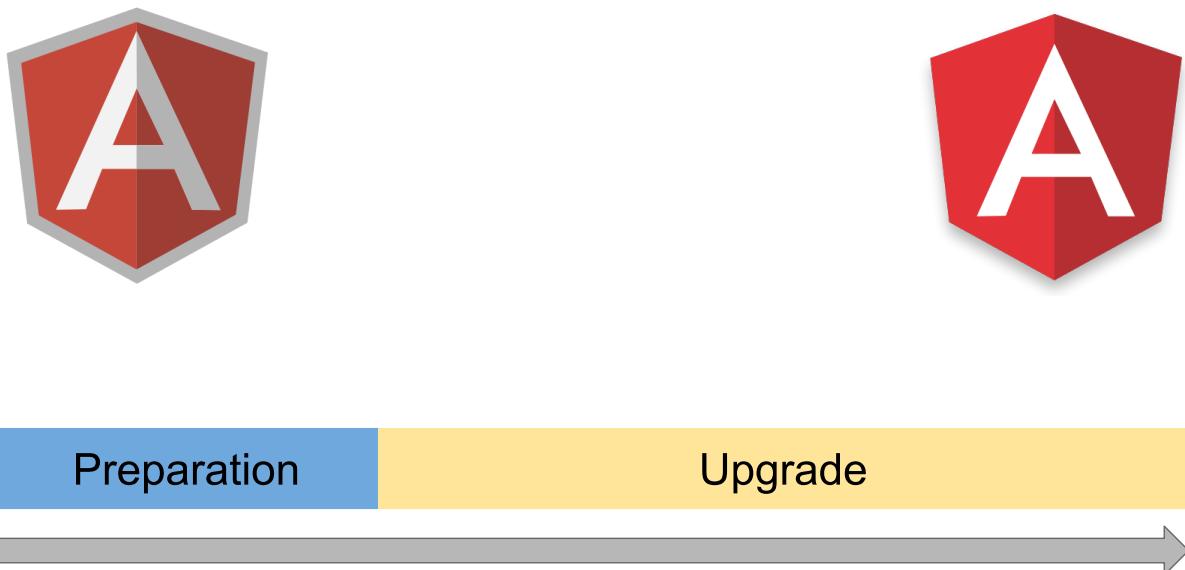
3. `x => x + x`
4. `(x,y) => x + y`
5. `<T>(x: T, y: T) => x === y`
6. Toujours utiliser les `{}` pour les conditions et boucles.
7. Les `{}` ouvrante figure sur la ligne de déclaration.
8. Ne pas utiliser d'espace intérieur avec les parenthèses:
9. `for (var i = 0, n = str.length; i < 10; i++) { }`
10. `if (x < 10) { }`
11. `function f(x: number, y: string): void { }`
12. Utiliser la **single var pattern**
(i.e. use `var x = 1; var y = 2;` over `var x = 1, y = 2;`).
13. Utiliser 4 espaces pour l'indentation.



Migrer d'AngularJS 1.x à AngularJS 2

8/ Migrer d'AngularJS 1.x à AngularJS 2

La migration entre angular 1 et 2 s'opère en deux phases :



- 1. Préparation:** Définir les portions d'application à migrer et les besoin en *tooling* ([plan de migration](#)).
- 2. Migration:** Porter le code en testant la compatibilité commune.

Préparation

1. Suivre les [bonnes pratiques standardisées par la communauté](#)

- Un composant par fichier.
- Strucutre de répertoire par fonctionnalité
- Utiliser un *module loader* (SystemJS, Webpack, ou Browserify)
- Migrer vers TypeScript (notamment pour les annotations)
- Utiliser des directive de composants

Les **directive de composants** sont la première (pré)implémentation des **Web Components** soit :

- `restrict: 'E'` Un composant représente une zone fonctionnelle (élément)
- `scope: {}` Un scope isolé représentant la responsabilité fonctionnelle.
- `bindToController: {}`
- `controller and controllerAs` Un composant expose une logique (class de préférence).
- `template or templateUrl` La création d'un élément nécessite un template.

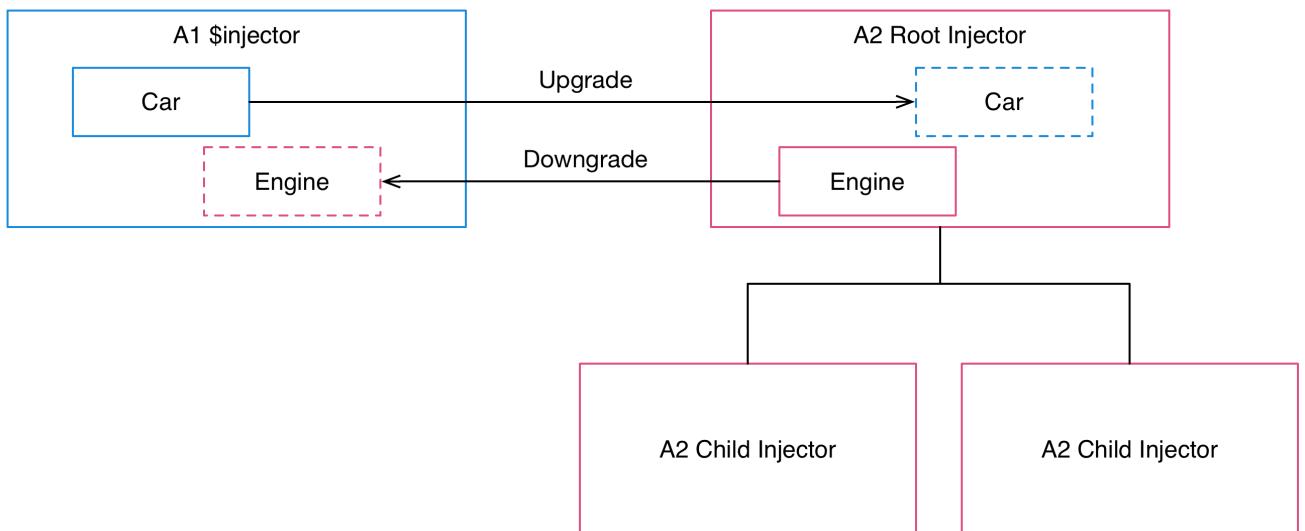
Options possibles:

- `transclude: true`
- `require`

Options à rejeter

- `compile`
- `replace`
- `priority`
- `terminal`

Le **sous-module** `angular2/upgrade` ou **ngUpgrade** permet l'adaption des composants applicatifs dans les deux sens **upgrade (ng1 > ng2)** et **downgrade (ng1 < ng2)**



8.1/ Comparaison et “topographie” des concepts.

Les concepts AngularJS 1.x restent présent dans AngularJS 2 mais se transforment avec l'évolution de JavaScript et des Web components.

Les concepts de **directives** **filtres** **services** et **composants** servent le même besoin conceptuels avec une implémentation basée sur **ES6/2015** dans l'esprit des **Web Components**

A noter: Sur la base **ES6/2015** avec Angular2 tout est `class` encapsulé par un `module`.

Angular2 invite à une approche structurelle du code `hyper-modulaire` nécessitant un workflow pour la gestion du build et du développement.

Tableau comparatif : Angular 1 vs 2

Angular 1	Angular 2
Bindings/interpolation	Bindings/interpolation
<code>{{vm.value}}</code>	<code>{{value}}</code>
Filters	Pipes
<code>{{vm.value uppercase}}</code>	<code>{{value uppercase}}</code>
Variables locales	Variables de template
<code><div ng-repeat="movie in vm.movies">{{movie.title}}</div></code>	<code><div *ngFor="let movie of movies">{{movie.title}}</div></code> OU <code><div *ngFor="#movie of movies">{{movie.title}}</div></code>
ng-app	Bootstrapping

Angular 1	Angular 2
<pre><body ng-app="app"></pre>	<pre>platformBrowserDynamic() .bootstrapModule(AppModule);</pre>
ng-class	ngClass
<pre><div ng-class="{active: isActive}"> <div ng-class="{active: isActive,shazam: isImportant}"></pre>	<pre><div [ngClass]="{active: isActive}"> <div [ngClass]="{active: isActive,shazam: isImportant}"> <div [class.active]="isActive"></pre>
ng-click	Binding événementiel
<pre><button ng- click="vm.toggleImage()"> <button ng- click="vm.toggleImage(\$event)"></pre>	<pre><button (click)="toggleImage()"> <button (click)="toggleImage(\$event)"></pre>
ng-controller	Component decorator
<pre><div ng- controller="MovieListCtrl as vm"></pre>	<pre>@Component({ moduleId: module.id, selector: 'movie-list', templateUrl: 'movie- list.component.html', styleUrls: ['movie- list.component.css'] })</pre>
ng-hide/ng-show	Binding sur l'attribut hidden
<pre></pre>	<pre></pre>
ng-src	Binding sur l'attribut src
<pre></pre>	<pre></pre>

Angular 1	Angular 2
ng-href	Binding sur l'attribut href
<pre><a ng- href="#movies">Movies</pre>	<pre><a [href]="angularDocsUrl">Angular Docs</pre>
Routing : ng-href	Binding spécial : routerLink
<pre><a ng- href="#movies">Movies</pre>	<pre><a [routerLink]=["/movies"]>Movies</pre>
ng-if	*ngIf
<pre><table ng-if="movies.length"></pre>	<pre><table *ngIf="movies.length"></pre>
ng-model	ngModel
<pre><input ng- model="vm.favoriteHero"/></pre>	<pre><input [(ngModel)]="favoriteHero" /></pre>
ng-for	*ngFor
<pre><tr ng-repeat="movie in vm.movies"></pre>	<pre><tr *ngFor="let movie of movies"></pre>
ng-style	ngStyle
<pre><div ng-style="{color: colorPreference}"></pre>	<pre><div [ngStyle]="{{color: colorPreference}}> <div [style.color]="colorPreference"></pre>
ng-switch	ngSwitch
<u>Synthèse</u>	

8.2/ Préparer la migration. Structure d'une application AngularJS 2.

[**ng-metadata**](#) (successeur de **ng-forward**) est la solution courante permettant de porter du code **ng1 (1.4+)** vers **ng2**

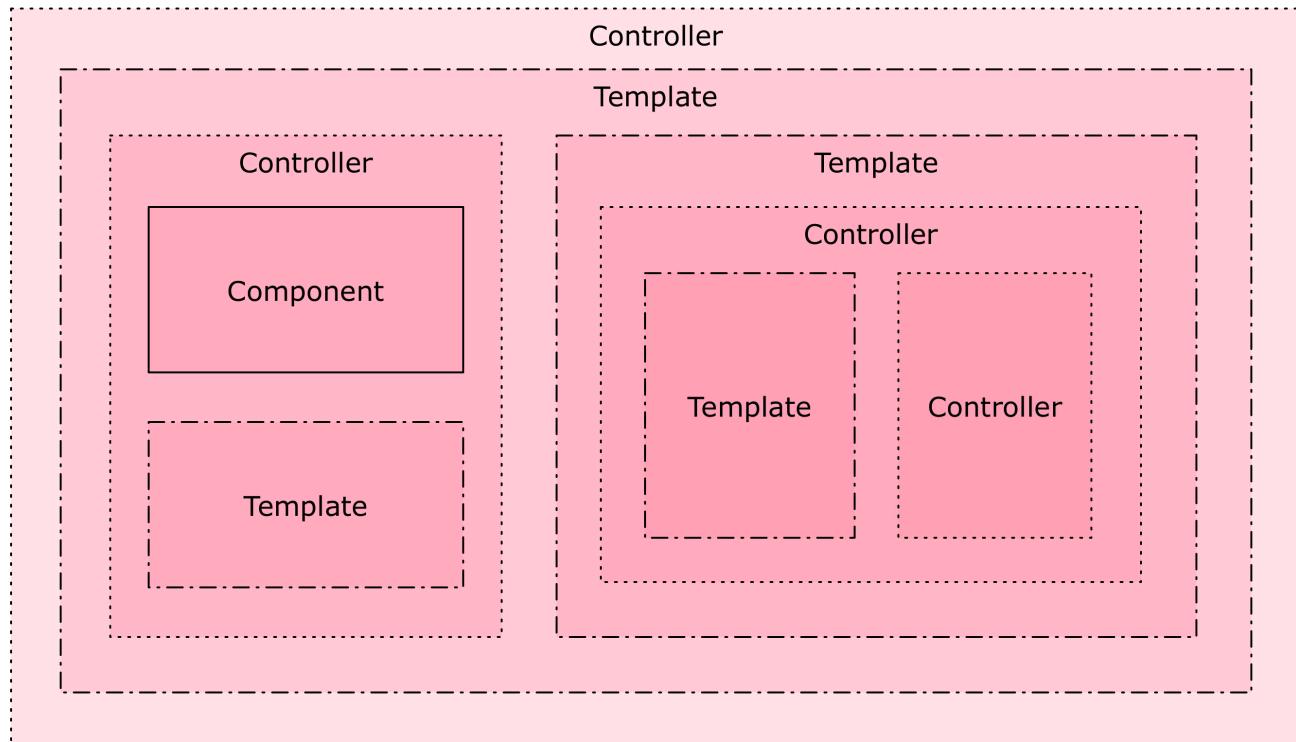
Stratégies

1. Par fichier avec `angular.module`
2. Par fichier avec déclaration au niveau du composant racine @Component et des (metadata providers/directive/pipes)
3. Hybride : en combinaison des deux précédentes.

Bénéfices :

- Angular 2 DI avec Angular 1
- Suppression des abstraction Angular 1 intermédiaires (link function, \$scope, \$element)
- Conception par composant
- Conception d'application en arbre de composants

Le plus grand bénéfice étant que le code de l'application **AngularJS 1.x** est alors pleinement compatible avec **ng2**.



8.3/ Les modules AngularJS 2, “core” et principaux modules.

Les modules définit **@NgModule** sont des conteneurs logiques rassemblant les composants de l'application.

Le décorateur @NgModule indique à angular quels sont les composants (composants, filtres, directives, services) publics ou internes propre au module.

Anatomie d'un module angular

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

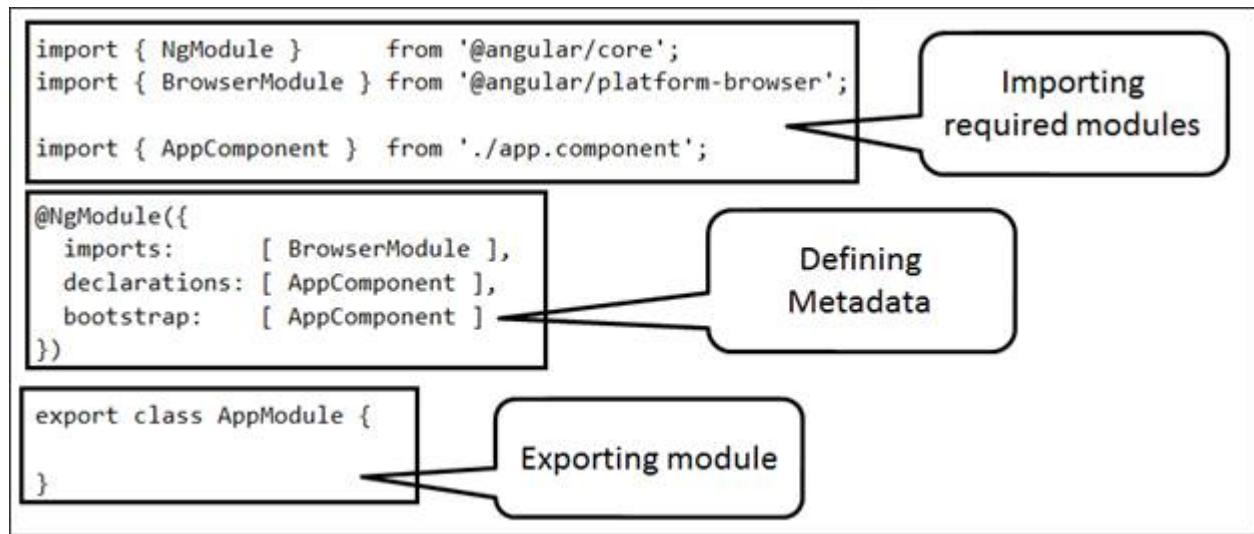
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:   [ AppComponent ]
})

export class AppModule { }
```

- **NgModule** décorateur : injecter les propriété descriptives (metadata)
 - **declarations** - Les classes propres au module (components, directives, pipes).
 - **exports** - Ensemble des `declaration` du module utilisable dans les template d'autre modules.
 - **imports** - Autre modules requis.
 - **providers** - Exposition de `service (s)` globaux.
 - **bootstrap** - Logique du **module root** (déclaration réservée au module principale).

Comprendre la structure d'un fichier.

1. Imports ES6/2015.
2. Décoration des meta-data **Angular2**
3. Export ES6/2015.



Bootstrapping du module root

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule); //Référence à : bootstrap: [ ]
```

Ne pas confondre : Les modules ES6/2015 et les modules angular.

Les premiers apportent l'isolation du point de vue du moteur JavaScript et les seconds du point de vue du cycle de compilation angular

Module ES6/2015 : Séparation du code.

Module Angular : Organisation de la logique.

Principaux modules :

Angular 2 prend la forme d'une collection de modules **ES6/2015** (comme autant de librairies) préfixés [**@angular**](#) exposant à leur tour un **module angular**

Chaque module comporte un équivalent destiné aux test eg: **@angular/core*/testing**

- **@angular/common** : Gestion de l'url. Filtres natifs. Directives de template.
- **@angular/core** : Cycle de compilation. Décorateurs.
- **@angular/compiler** : Compilation de templates. CLI -> AoT (Ahead of Time).
- **@angular/http** : Encapsulation réseau (AJAX,JSONP)
- **@angular/forms** : Logique de validation des formulaires. Directives de formulaires.
- **@angular/router** : Gestion du routing de l'abre des composant (indépendant de l'url)
- **@angular/platform-browser** : Gestion du Browser.
- **@angular/platform-browser-dynamic** : Gestion dynamiques du Browser.
- **@angular/platform-server** : Gestion Server.
- **@angular/platform-webworker** : Gestion Web Worker.
- **@angular/platform-webworker-dynamic** : Gestion dynamiques des Worker(s).
- **@angular/upgrade** : Compatibilité ng1 / ng2

8.4/ Principe de l'injection de dépendance.

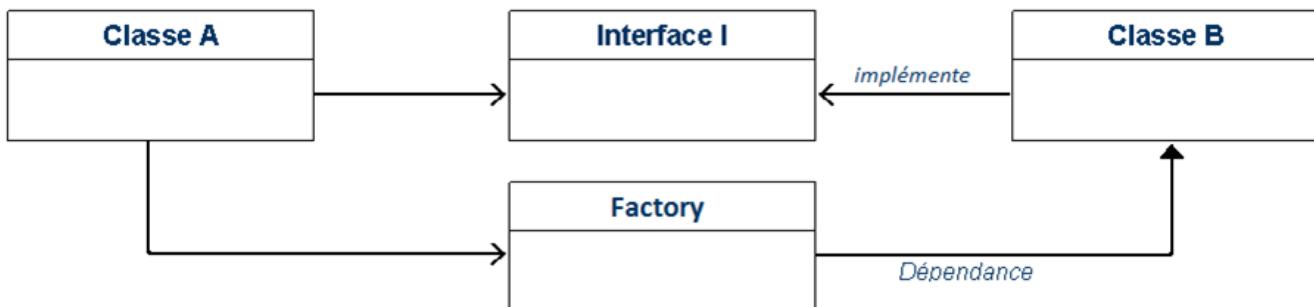
L'injection de dépendances (**DI Dependency Injection en anglais**) est un mécanisme consistant à créer dynamiquement (**injecter**) les dépendances entre les différentes classes en s'appuyant sur une description.

Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution.

On distingue essentiellement **4 types de dépendances**

1. **(dépendance par composition)** un attribut de type X
2. **(dépendance par héritage)** un objet étant une classe X
3. **(dépendance par transitivité)** un objet étant une classe Y dependnat d'une classe X
4. une méthode d'un objet Y appelle une méthode d'un objet X.

Les **interfaces et les factories** sont un moyen de simplification des l'expression des dépendances.



Il existe 4 types d'injections de dépendances :

Injection par constructeur

Injection par interface

Injection par mutateur

Injection par champs

8.5/ Classification des directives: , Composant, Attribut, services.



Une application *Angular2 s'appuie sur 4 briques conceptuelles.

- Component : Definition d'un ensemble fonctionnel de l'Interface Utilisateur.
- Directive : Definition d'une transformation (CSS) ou comportement (Event) exprimé via un attribut .
- Services : Ensemble logique de manipulation des données (Model).
- Routers : La résolution de navigation basée sur les composant et le module routing.

Les composants sont en fait des directives exposant un template .

Directive et **structural directives** on appelle structural directive une directive ayant un impact sur la structure du DOM (NgFor, NgIf)

8.6/ Les décorateurs : définition des hiérarchies.

Les décorateurs sont une fonctionnalité ajoutée par TypeScript (1.5+) pour le support d'Angular et la simplification de syntaxe.

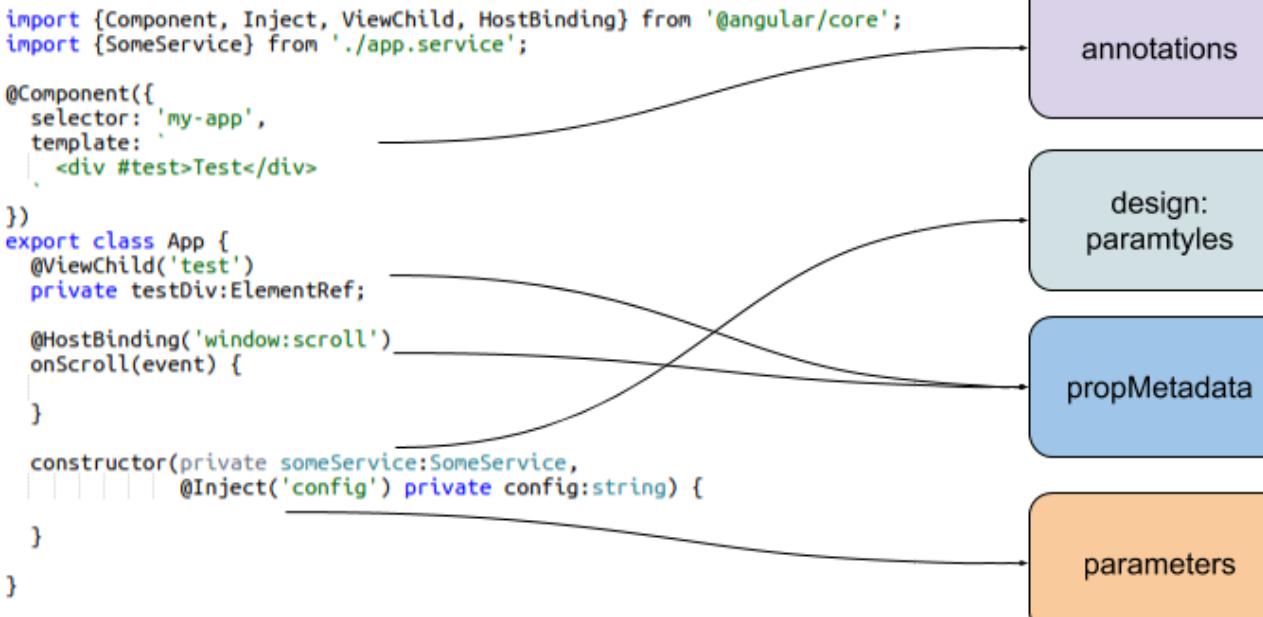
Un décorateur est une façon de faire de la méta-programmation semblable aux annotations (Java, C#)

Le framework Angular 2, fournit des méthodes préfixées par @ (en TypeScript).

Leur rôle est simple: ajouter des métadonnées aux classes, propriétés ou paramètres.

Niveau de décoration angular2

- annotations(class)
- design:paramtypes
- propMetadata(property)
- parameters(method)



```

import {Component, Inject, ViewChild, HostBinding} from '@angular/core'
import {SomeService} from './app.service';

@Component({
  selector: 'my-app',
  template: `
    <div #test>Test</div>
  `
})
export class App {
  @ViewChild('test')
  private testDiv: ElementRef;

  @HostBinding('window:scroll')
  onScroll(event) {

  }

  constructor(private someService:SomeService, @Inject('config') config:string) {
  }
}

```

Mécanisme

```
// Décoration
@decoratorExpression
class MyClass { }
```

La fonction de décoration serait :

```
function decoratorExpression(target) {
  // Add a property on target
  target.annotated = true;
}
```

Le résultat ES5 est proche du mécanisme d'annotations de l'injection de dépendance d'Angular1.

```
MainCtrl.$inject=['$scope','$http']
function MainCtrl($scope,$http) {
}
```

Comparaison des syntaxes

Angular2 en ES5

```
(function(app) {
  app.AppModule =
    ng.core.NgModule({
      imports: [ ng.platformBrowser.BrowserModule ],
      declarations: [ app.AppComponent ],
      bootstrap: [ app.AppComponent ]
    })
    .Class({
      constructor: function() {}
    });
})(window.app || (window.app = {}));
```

Angular2 en ES6 avec TypeScript :

- Dépendances claires
- Lisibilité
- Code concis

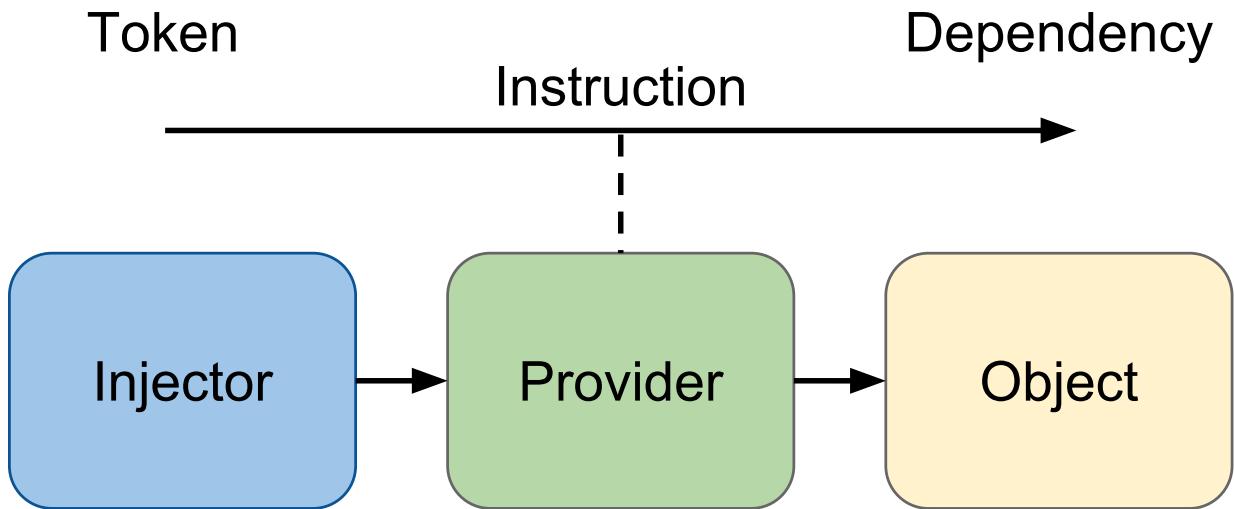
```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```



Injection de dépendances

9/ Injection de dépendances



Avec **Angular2** l'injection de dépendance est résolue sur trois concepts clés :

- **Injector** Moteur de résolution des instantiations.
- **Provider** `factory` utilisant un `token` pour enregistrer la méthode de création d'une dépendance.
- **Dependency** Déclaration du `type` d'objet à créer.

```
import { ReflectiveInjector } from '@angular/core';

var injector = ReflectiveInjector.resolveAndCreate([MyService]);

var car = injector.get(MyService);
```

Ici l'API exposée par `ReflectiveInjector` reçoit une liste de `providers` on pourrait réécrire ce code :

```
import { ReflectiveInjector } from '@angular/core';

var injector = ReflectiveInjector.resolveAndCreate([{ provide: MyService, useClass: My
var car = injector.get(MyService);
```

9.1/ Utiliser les annotations et décorateurs.

Principaux décorateurs

@Component Enregistre le traitement d'une classe comme **composant** d'application.

```
@Component({selector: 'greet', template: 'Hello {{name}}!'})
class Greet {
  name: string = 'World';
}
```

Un **composant** doit appartenir à un **NgModule** (dans les déclarations) pour être réutilisable dans l'application.

Meta-Data

- **animations** Liste des animations.
- **changeDetection** Mode de détection des changements (default onPush)
- **encapsulation** Mode d'encapsulation des CSS (Native None)
- **entryComponents** Liste des composants insérés dynamiquement dans la vue.
- **exportAs** Nom d'export du composant dans un template.
- **host** Correspondance des evenements.
- **inputs** List de propriété d'input.
- **interpolation** Interpolations personnalisées.
- **moduleId** Référence au à l'identifiant de module ES/CommonJS du fichier.
- **outputs** List de propriété exposant un event de sortie.
- **providers** List de providers.
- **queries**
- **selector** CSS identifiant le composant dans un template.
- **styleUrls** List des fichiers CSS à inclure.
- **styles**
- **template**
- **templateUrl** url du fichier de template.
- **viewProviders** List des providers accessibles.

@Directive Enregistre le traitement d'une classe comme **directive** d'application.
Marks a class as an Angular directive and collects directive configuration metadata.

```
import {Directive} from "@angular/core";  
  
@Directive({ selector: "my-directive" })  
export class MyDirective { }
```

Une **directive** doit appartenir à un **NgModule** (dans les déclarations) pour être réutilisable dans une application.

Meta-Data

- **exportAs**
- **host**
- **inputs**
- **outputs**
- **providers**
- **queries**
- **selector**

@Pipe() Enregistre le traitement d'une classe comme **filtre de transformation** d'application.

```
import { Pipe, PipeTransform } from '@angular/core';  
  
@Pipe({name: 'filterName'})  
export class MyFilter implements PipeTransform {  
  transform(value): any {  
    let transformedValue = value + '!';  
    return transformedValue;  
  }  
}
```

Meta-Data

- **name** Nom d'usage du filtre.
- **pure(Boolean)** Mode de détection des changements (shallow深深/deep).

@NgModule Enregistre les `meta-data` d'un module et de ses composants.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import
  { AppComponent }  from './app.component';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

@Inject() Requiert l'injection du paramètre.

```
import { Component, Inject } from '@angular/core';
import { Hamburger } from '../services/hamburger';

@Component({
  selector: 'app',
  template: `Bun Type: {{ bunType }}`
})
export class App {
  bunType: string;
  constructor(@Inject(Hamburger) h) {
    this.bunType = h.bun.type;
  }
}
```

@Injectable() Enregistre une `class` comme candidat à l'injection de dépendance.

```
@Injectable()
class UsefulService {
}
@Injectable()
class NeedsService {
  constructor(public service: UsefulService) {}
}
```

A noter l'injecteur fera une `NoAnnotationError` sur une tentative d'injection de class non `@Injectable`.

@Component, @Directive et @Pipe sont des sous-type de `@Injectable`

@SkipSelf() vs @Self() Spécifie si le mécanisme de DI doit commencer sur l'injecteur parent.

```
class Dependency {}
@Injectable()
class NeedsDependency {
  constructor(@SkipSelf() public dependency: Dependency) { this.dependency = dependency }
}
const parent = ReflectiveInjector.resolveAndCreate([Dependency]);
const child = parent.resolveAndCreateChild([NeedsDependency]);
```

@Host Spécifie la résolution de dépendance via la chaîne des injecteur.

```
class OtherService {}
class HostService {}
@Directive({selector: 'child-directive'})
class ChildDirective {
  logs: string[] = [];
  constructor(@Optional() @Host() os: OtherService, @Optional() @Host() hs: HostService) {
    // os is null: true
    this.logs.push(`os is null: ${os === null}`);
    // hs is an instance of HostService: true
    this.logs.push(`hs is an instance of HostService: ${hs instanceof HostService}`);
  }
}
@Component({
  selector: 'parent-cmp',
  viewProviders: [HostService],
  template: '<child-directive></child-directive>',
})
class ParentCmp {
}
@Component({
  selector: 'app',
  viewProviders: [OtherService],
  template: '<parent-cmp></parent-cmp>',
})
class App {
}
```

@Optional Un paramètre optionnel (initialiser à `null` si non trouvé)

```
class Engine {}
@Injectable()
class Car {
  constructor(@Optional() public engine: Engine) {}
}
```

@Input() et **@Output()** Déclarent un `binding` unidirectionnel.

Autres décorateurs

- `@ViewChildren`
- `@ViewChild`
- `@ContentChild`

- @ContentChildren

9.2/ Configuration de l'injecteur.

Il existe différentes manières de configurer un injecteur :

Class (useClass,useExisting)

```
{ provide: MyService, useClass: OtherService }
{ provide: MyService, useClass: MyService }
{ provide: Service, useExisting: MyService }{ provide: 'some value', useValue: 'Hello'
```

Factory (useFactory)

```
{
  provide: MyService,
  useFactory: () => {
    if (ENV_IS_PROD) {
      return new ProdService();
    } else {
      return new DevService();
    }
  }
}
```

Exposer une Factory comme manager :

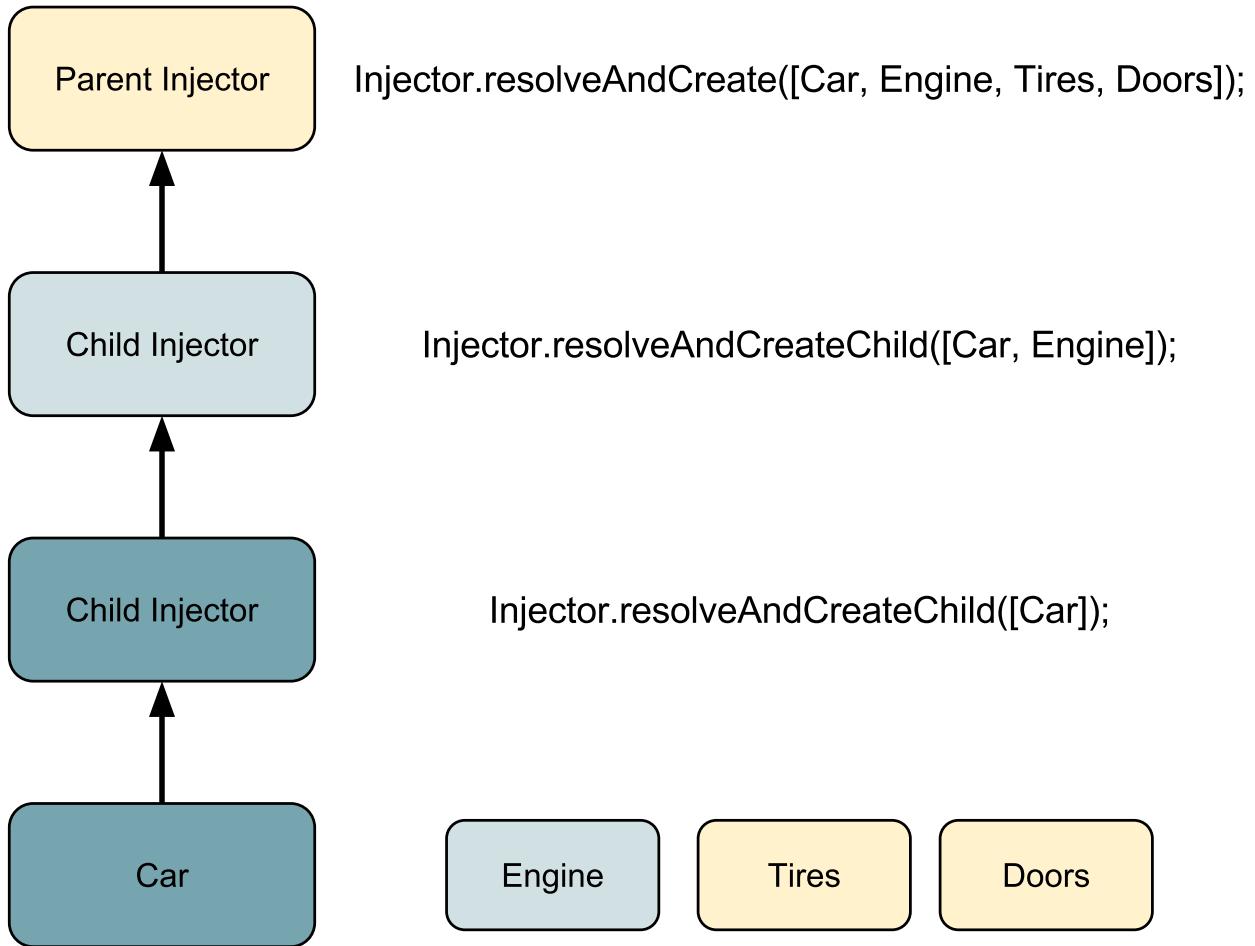
```
{  
  provide: MyService,  
  useFactory: () => {  
    return () => {  
      return new MyService();  
    }  
  }  
}
```

Value (useValue)

```
{ provide: 'myValue', useValue: 'Hello World' }
```

Ces `strategies` sont applicable à chaque fois qu'une définition de `provider` est possible (`NgModule`)

Angular2 construit un arbre d'injecteur qu'il est possible de gérer distinctement :



Les injecteurs fonctionnent à la manière de la **chaine des prototype** en JavaScript.

Si un injecteur enfant ne peut résoudre une injection il la résoudra via sa chaîne de parenté. Cela permet de configurer avec précision la **visibilité des dépendances**.

```
var injector = ReflectiveInjector.resolveAndCreate([MyService,OtherService]);
var childInjector = injector.resolveAndCreateChild([MyService]);

injector.get(MyService) !== childInjector.get(MyService);
injector.get(OtherService) == childInjector.get(OtherService);
```

9.3/ Gestion des modules : bonnes pratiques.

Contrairement à **Angular1**, **Angular2** expose un arbre d'injecteur reflétant l'arbre des composants de l'application.

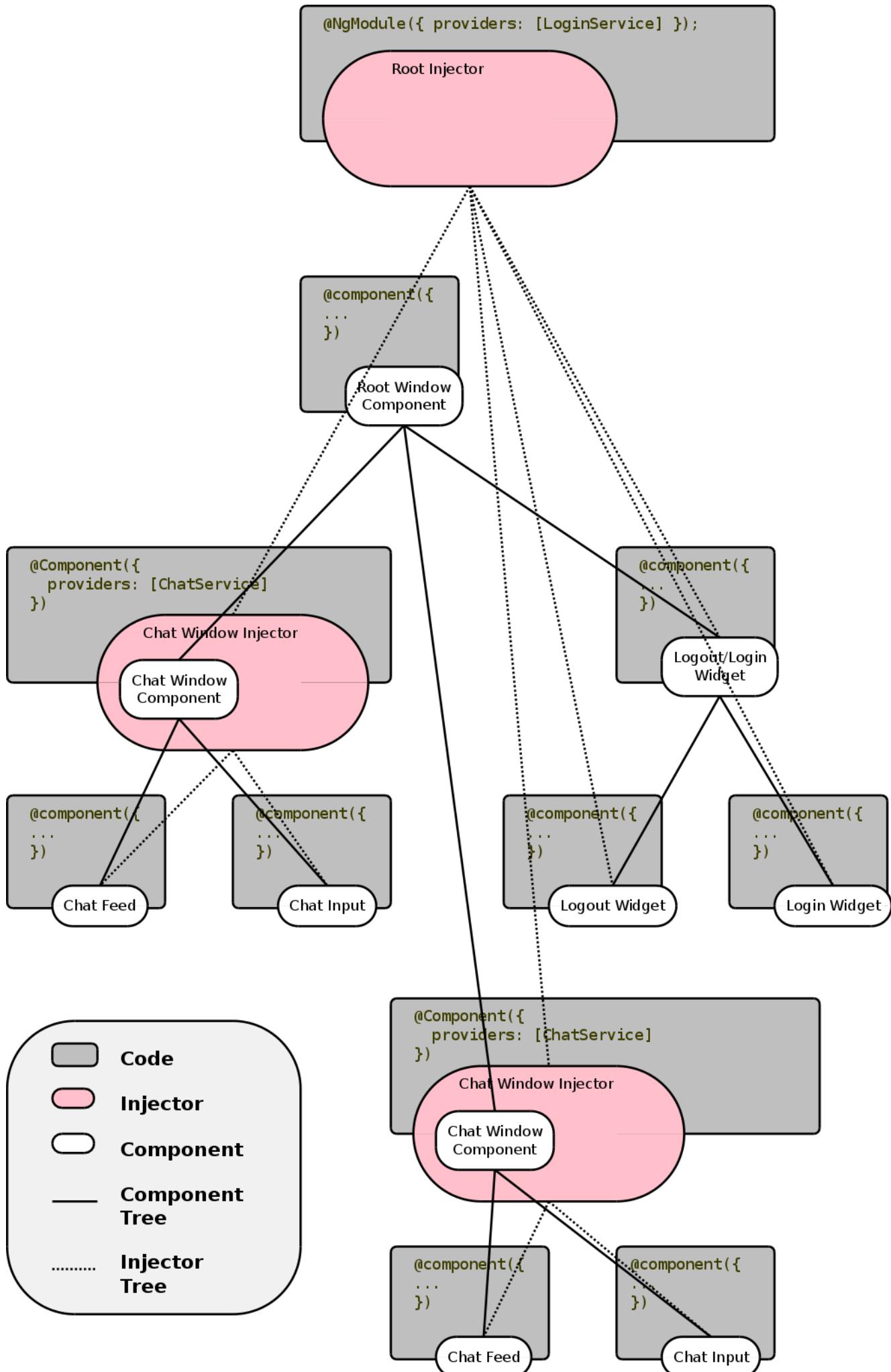
Les dépendances sont des **singleton dans leur scope d'injection** toutefois un **provider** définit dans un **NgModule** sera enregistré au niveau du **RootInjector** et accessible dans toute l'application, tandis qu'une définition via un composant n'est visible que dans son arbre de descendance.

Important Lors de la déclaration de module, Angular peut utiliser la **clé calculée de moduleId:module.id** pour résoudre le chemin de fichier (ajouté automatiquement par WebPack).

Organisation des dépendances:

- Exposer via **@NgModule** les services globaux.
- Une **class** par fichier. (prévient les erreurs de résolution).
- Les **Interface typeScript** ne peuvent pas être utilisées comme **token**.

A noter si un composant redéfinit dans ses **providers** un token identique à celui de son parent, cela bloquera la résolution de dépendance au niveau global.



9.4/ Création de services injectables. Classification des services.

```
import { Injectable } from '@angular/core';

@Injectable()
export class Logger {
  logs: string[] = [];// capture logs for testing
  log(message: string) {
    this.logs.push(message);
    console.log(message);
  }
}
```

9.5/ “BootStrap” d’application.

En suivant ces règles une application sera toujours composée d’au moins quatres fichiers.

- bootstrap : invocation du polyfill de chargement des modules ES6/2015.
- Main Entry : fichier d’entrée vers l’arborescence de mdoules ES6/2015.
- @NgModule : main module de l’application.
- @Component : composant principal de l’application (root).

polyfill

```
<!-- Polyfill(s) for older browsers -->
<script src="node_modules/core-js/client/shim.min.js"></script>
<!-- Zone d'exécution -->
<script src="node_modules/zone.js/dist/zone.js"></script>
<!-- Meta-Data -->
<script src="node_modules/reflect-metadata/Reflect.js"></script>
<!-- Polyfill(s) ES6 modules system -->
<script src="node_modules/systemjs/dist/system.src.js"></script>
<!-- 2. Configure SystemJS -->
<script src="systemjs.config.js"></script>
<script src="bootstrap.js"></script>
```

bootstrap.js

```
System.import('app').catch(function(err){ console.error(err); });
```

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app.module';

const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

main.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

main.compoenet.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: '<h1>My First Angular App</h1>'
})
export class AppComponent { }
```

9.6/ Organisation des modules.

L'Organisation des modules peut se faire selon par destination.

AppModule : module principale de l'application initialisant l'abre des composant.

FeatureModule : Collection d'utilitaires métiers ajoutés à la racine de l'application (AppModule)

RoutingModule : Définition des routes de applicatives.

Structure d'un `FeatureModule`

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { UserService } from './user.service';
import { UserComponent } from './user.component';

@NgModule({
  imports: [CommonModule],
  declarations: [
    UserService,
    UserComponent
  ],
  providers: [UserService],
  exports: [UserComponent]
})
export class UserModule {}
```

Import du `FeatureModule`

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { UserModule } from 'user/usermodule';
import { AppComponent } from './app.component';

@NgModule({
  imports: [
    BrowserModule,
    UserModule
  ],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

A noter Les `Component`, `Directive`, et `Pipe` ne sont visibles que dans leur module à moins d'être explicitement exportés.

Les `Services` sont par contre résolu sur l'arborescence des injecteurs.

En cas de chargment dynamique Angular ne rattache le module chargé à l'injecteur racine.

La méthode `forRoot` permet de spécifier à nouveau cette référence.

Structure d'un `FeatureModule forRoot`

```

import { NgModule , ModuleWithProviders } from '@angular/core';
import { CommonModule } from '@angular/common';

import { UserService } from './user.service';
import { UserComponent } from './user.component';

@NgModule({
  imports: [CommonModule],
  declarations: [
    UserService,
    UserComponent
  ],
  providers: [UserService],
  exports: [UserComponent]
})
export class UserModule {
  //La méthode statique forRoot redéfinit la déclaration de provider
  static forRoot(): ModuleWithProviders {
    return {
      ngModule: UserModule,
      providers: [UserService]
    }
  }
}

```

Import forRoot du FeatureModule

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { UserModule } from 'user/usermodule';
import { AppComponent } from './app.component';

@NgModule({
  imports: [
    BrowserModule,
    UserModule.forRoot() // Appel de la déclaration de provider
  ],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Utilisez Toujours la méthode `forRoot` pour exporter un `Service` de puis un `FeatureModule`.



Définition de composants

10/ Définition de composants

Les composants applicatifs **Angular2** sont des `Directives` activant un template dans cycle de compilation.

Il utilise :

Une class de définition.

Des données issues de `Services`.

Une syntaxe de template.

Des directives de structure (`template Directives`).

Des filtres (Pipe).

Des `Directives` personnalisées.

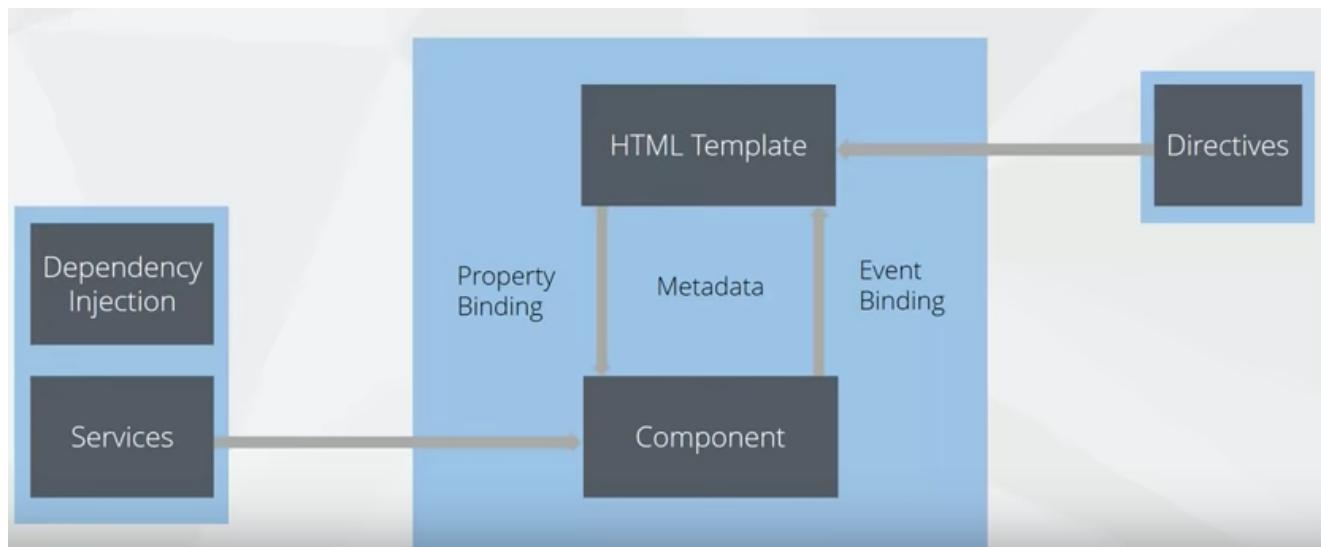
* D'autres composants.

Définition d'un composant

```
import {Component} from '@angular/core';

@Component({
  selector: 'hello',
  template: '<p>Hello, {{name}}</p>'
})
export class Hello {
  name: string;

  constructor() {
    this.name = 'World';
  }
}
```



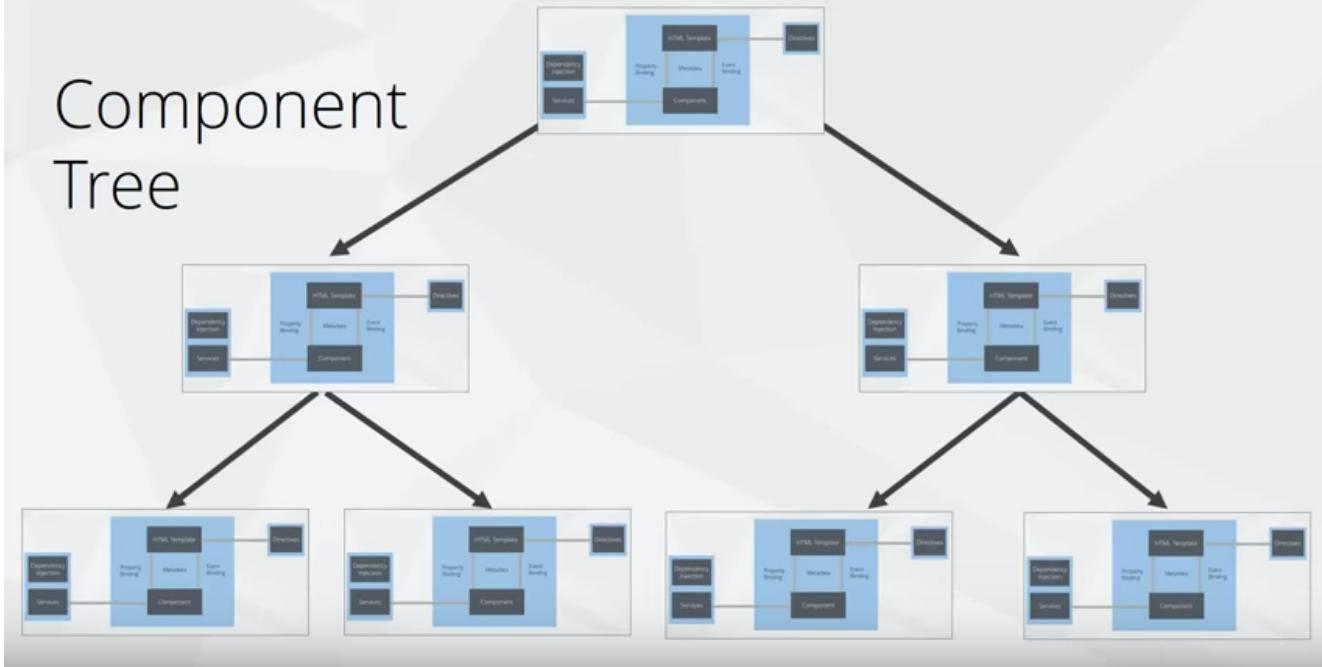
Une application est constituée d'un arbre de composant. **Component Tree**

```
<AppComponent>

  <ListComponent>
    <ListItem></ListItem>
    <ListItem></ListItem>
    <ListItem></ListItem>
  </ListComponent>

  <ListForm></ListForm>
</AppComponent>
```

Component Tree



10.1/ Cycle de vie dans l'application.

Transmission de données.

En considérant le `Component Tree` les données peuvent être transmises à composant via une déclaration d'`input unidirectionnel`.

```
import {Component, Input} from '@angular/core';

@Component({
  selector: 'hello',
  template: '<p>Hello, {{name}}</p>'
})
export class Hello {
  //Décorateur
  @Input() name: string;
}
```

Pour retourner des données le composant peut émettre un événements.

```
import {Component, EventEmitter, Input, Output} from '@angular/core';

@Component({
  selector: 'counter',
  template: `
    <div>
      <p>Count: {{ count }}</p>
      <button (click)="increment()">Increment</button>
    </div>
  `
})
export class Counter {
  @Input() count: number = 0;
  @Output() result: EventEmitter = new EventEmitter();

  increment() {
    this.count++;
    //Décorateur
    this.result.emit(this.count);
  }
}
```

A noter les événements se propagent selon le modèle du DOM le paramètre `$event` permet leur capture.

```
<button (click)="clicked($event)"></button>
```

```
@Component(...)  
class MyComponent {  
  clicked(event) {  
    event.preventDefault();  
  }  
}
```

Two-Way Data Binding

Le mécanisme du Two-Way Data Binding est une combinaison des @Input et @Output .

```
<hero-detail [hero]="currentHero" (deleteRequest)="deleteHero($event)">
```

```
<input [(ngModel)]="name" >  
  
//Similaire à  
  
<input [ngModel]="name" (ngModelChange)="name=$event">
```

Il est nécessaire de prévoir une propriété d' @Output suffixée par “Change”.

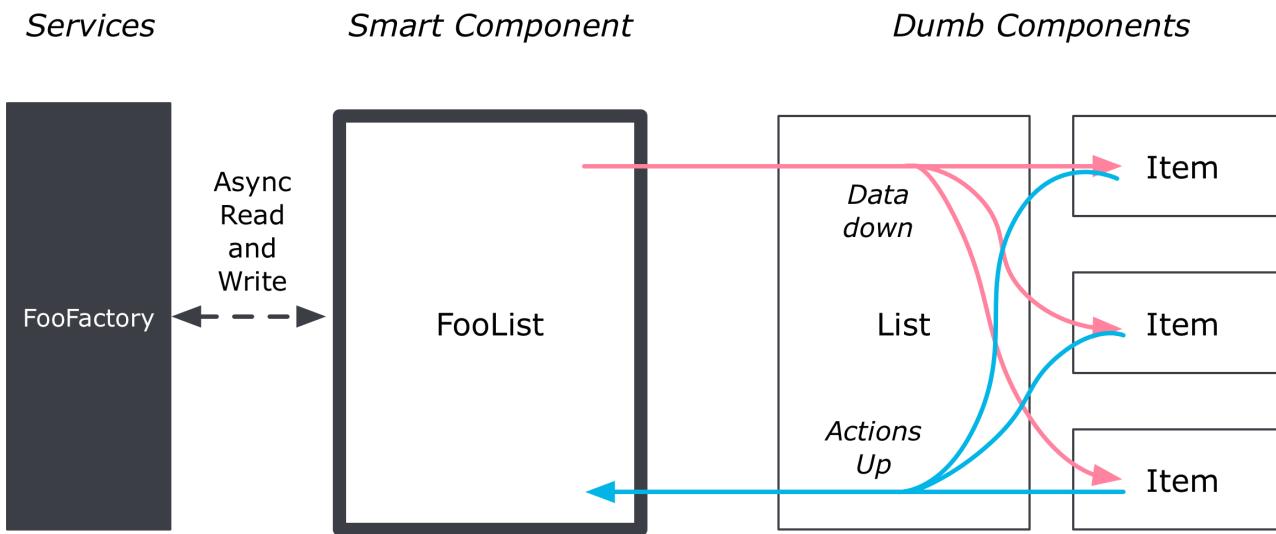
```
@Component({/*....*/})  
export default class Counter {  
  @Input() count: number = 0;  
  @Output() countChange: EventEmitter<number> = new EventEmitter<number>();  
  
  increment() {  
    this.count++;  
    this.countChange.emit(this.count);  
  }  
}
```

Utilisation :

```
@Component({  
  template: '<counter [(count)]="myNumber"></counter>'  
  directives:[Counter]  
})  
class SomeComponent {  
  // ...  
}
```

Accès à un composant enfant.

Il est souvent nécessaire de créer des composants **ignorant la logique** exploitée ou **Dummy Component**. Ces composants sont des briques fonctionnelles réutilisables.



Un composant parent peut accéder au propriété d'un enfant (Component Tree) en le référençant par **DI** ou variable local

Dependency Injection par Inférence :

```
@Component({
  selector: 'my-example',
  templateUrl: 'app/my-example.component.html'
})
export class MyExampleComponent {
  submitForm (form: NgForm) {
    console.log(form.value);
  }
}
```

Variable locale :

```
<red-ball #myBall></red-ball>
The ball is {{ myBall.color }}.
```

Un accès plus explicite est possible grâce au décorateurs **@ViewChild & @ViewChildren**

@ViewChild sélectionne la `class` d'un composant enfant par inférence de type.

@ViewChildren sélectionne un ensemble ou `QueryList`.

Les deux décorateurs supporte un `selector` en paramètres.

```
import {Component, QueryList, ViewChildren} from '@angular/core';
import {Hello} from './hello.component';

@Component({
  selector: 'app',
  template: `
    <div>
      <title></title>
      <hello></hello>
      <hello></hello>
      <hello></hello>
    </div>
    <button (click)="onClick()">Call Child function</button>
  `
})
export class App {
  @ViewChild(Title) child: Title;
  @ViewChildren(Hello) helloChildren: QueryList<Hello>;

  constructor() {}

  onClick() {
    this.helloChildren.forEach((child) => child.exampleFunction());
    this.child.exampleFunction();
  }
}
```

Les décorateurs `@ContentChild` & `@ContentChildren` fonctionnent sur le même principe mais sur le contenu projeté c-a-d inséré depuis un composant parent.

(Ce mécanisme est similaire à la transclusion ng1)

```
@Component({
  selector: 'app',
  template: `
<div style="border: 2px solid black; padding: 1rem;">
  <h4>App Component</h4>
  <child>
    <header>Contenu projeté dans le composant *child*.</header>
    <p>
      <b>Count:</b> {{ count }} <br/>
      <b>Child Count:</b> {{ childCount || 'N/A' }}
    </p>
  </child>
`)

})
```

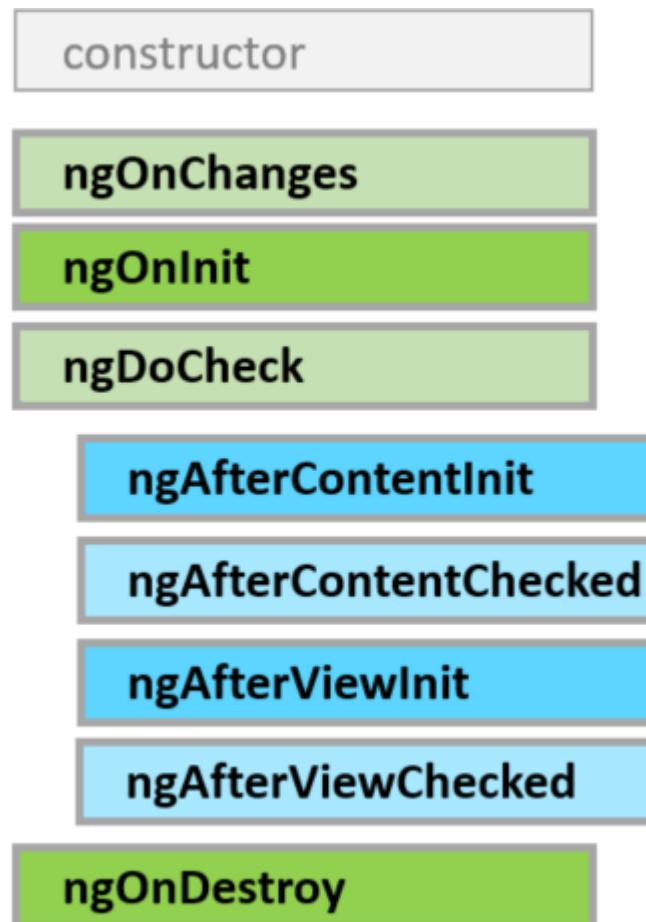
ng-content select= définit l'insertion du contenu projeté.

```
@Component({
  selector: 'child',
  template: `
<div style="border: 2px solid red; padding: 1rem; margin: 2px;">
  <h4>Child Component with Select</h4>
  <div style="border: 2px solid orange; padding: 1rem; margin: 2px">
    <ng-content select="header"></ng-content>
  </div>
</div>
`)

})
export class Child {
```

Cycle de vie.

Les composants suivent le cycle de gestion d'angular (création, rendu, insertion de données...) il est possible de réimplémenter (surcharge) les méthodes correspondant à ce cycle (LifeCycle Hooks)



- `ngOnChanges` - un `@Input` change.
- `ngOnInit` - Début du cycle.
- `ngDoCheck` - Après chaque détection de changement.
- `ngAfterContentInit` - Après initialisation du contenu.
- `ngAfterContentChecked` - Après vérification du contenu.
- `ngAfterViewInit` - Après initialisation des `views`.
- `ngAfterViewChecked` - Après vérification des `views`.
- `ngOnDestroy` - Après la destruction du composant.

[Cycle de vie](#) par l'exemple ou [en détail](#)

Manipulation du DOM

ElementRef expose un accès au DOM natif.

```
import {AfterContentInit, Component, ElementRef} from '@angular/core';

@Component({
  selector: 'app',
  template: `
    <h1>My App</h1>
    <pre style="background: #eee; padding: 1rem; border-radius: 3px; overflow: auto;">
      <code>{{ node }}</code>
    </pre>
  `
})
export class App implements AfterContentInit {
  node: string;

  constructor(private elementRef: ElementRef) {}

  ngAfterContentInit() {
    const tmp = document.createElement('div');
    const el = this.elementRef.nativeElement.cloneNode(true);

    tmp.appendChild(el);
    this.node = tmp.innerHTML;
  }
}
```

10.2/ Syntaxe des templates : interpolation/expression, “Binding” et filtres.

Comparé au 43 directives de angular1, le système de template d'**Angular2** propose une syntaxe puissante pour exprimer les parties dynamiques du code HTML.

- `{{}}` pour l'interpolation.
- `[]` pour le binding de propriété.
- `()` pour le binding d'événement.
- `#` pour la déclaration de variable.
- `*` pour les directives structurelles.

Les **Expressions de Template** sont évaluées afin de produire la valeur à représenté. Elles se réfèrent **exclusivement** au propriétés de leur composant.

- Ne pas affecter les propriétés du composant (`side - effect`)
- Simple
- Rapide d'exécution.

```
 {{1 + 1}}
```

ou

```
[property]="expression"
```

Les **Template Statement** sont les expression réponse à un événement.

Opérateurs non supportés :

- `new`
- `++ et --`
- `+= et -=`
- `| and &`
- Opérateurs de template.

Syntaxe canonique

Chacune de ces syntaxe supporte une version plus verbeuse appelée `canonical syntax`.

Binding de propriété.

```

```

Angular recherche toujours la propriété sur l'élément (!=attribut)

Canonical Syntax

```

```

Binding sur attribut.

Lorsqu'il n'existe pas la propriété recherchée sur un élément il est possible d'utiliser un attribut.

```
Template parse errors:  
Can't bind to 'colspan' since it isn't a known native property
```

```
<table border=1>  
  <!-- expression calculates colspan=2 -->  
  <tr><td [attr.colspan]="1 + 1">One-Two</td></tr>  
  
  <!-- ERROR: There is no `colspan` property to set!  
        <tr><td colspan="{{1 + 1}}>Three-Four</td></tr>  
  -->  
  
  <tr><td>Five</td><td>Six</td></tr>  
</table>
```

Binding sur une Classe CSS.

Fonctionne comme pour les attributs en utilisant le mot clé **class**

```
<!-- reset/override all class names with a binding -->
<div class="bad curly special"
    [class]="badCurly">Bad curly</div>

<!-- toggle the "special" class on/off with a property -->
<div [class.special]="isSpecial">The class binding is special</div>

<!-- binding to `class.special` trumps the class attribute -->
<div class="special"
    [class.special]!="isSpecial">This one is not so special</div>
```

Binding sur l'attribut style.

Fonctionne comme pour les attributs en utilisant le mot clé **style**

Les propriétés peuvent être notées en `dash-case` ou `camelCase`

```
<button [style.color] = "isSpecial ? 'red': 'green'">Red</button>
<button [style.backgroundColor] = "canSave ? 'cyan': 'grey'">Save</button>

<button [style.fontSize.em] = "isSpecial ? 3 : 1" >Big</button>
<button [style.fontSize.%] = "!isSpecial ? 150 : 50" >Small</button>
```

Alternativement il est possible d'utiliser la directive **ngStyle**

```
<div [ngStyle] = "setStyles()">
  This div is italic, normal weight, and extra large (24px).
</div>
```

```
setStyles() {
  return {
    // CSS property names
    'font-style': this.canSave ? 'italic' : 'normal', // italic
    'font-weight': !this.isUnchanged ? 'bold' : 'normal', // normal
    'font-size': this.isSpecial ? '24px' : '8px' // 24px
  };
  return;
}
```

Binding événementiel.

```
<button (click)="onSave()">Save</button>
```

Canonical Syntax

```
<button on-click="onSave()">On Save</button>
```

Custom Event

Angular peut associer un événement personnalisé défini dans le composant.

```
<div (myClick)="clickMessage=$event">click with myClick</div>
```

Utilisation des Pipe (filtres)

Les [Pipe](#) sont des fonctions de transformation (paramétrable) utilisables dans les expressions.

```
<!-- Simple -->
<p>The hero's birthday is {{ birthday | date }} </p>

<!-- Parameter -->
<p>The hero's birthday is {{ birthday | date:'MM/dd/yy' }} </p>

<!-- Chain -->
<p>The hero's birthday is {{ birthday | pipe1 | pipe2 }} </p>
```

Les Pipes natifs :

- **AsyncPipe** - `{{ value | async }}`
- **CurrencyPipe** - `{{ value | currency:ISO }}`
- **DatePipe** - `{{ value | date }}`
- **DecimalPipe** - `{{ value | decimal }}`
- **I18nPluralPipe** - `{{ value | json }}`
- **I18nSelectPipe** - `{{ value | json }}`
- **JsonPipe** - `{{ value | json }}`
- **LowerCasePipe** - `{{ value | lowercase }}`
- **PercentPipe** - `{{ value | json }}`
- **SlicePipe** - `{{ value | slice:0:2 }}`
- **UpperCasePipe** - `{{ value | uppercase }}`

DecimalPipe

- `integerDigits` : complète la partie gauche du format.
- `minFractionDigits` : complète la partie droite du format.
- `maxFractionDigits` : précision décimale.

```
<p>{{ 12345.16 | number:'6.1-1' }}</p>
<!-- will display '012,345.2' -->
```

CurrencyPipe

- Code ISO de la devise ('EUR', 'USD'...)
- Optionnellement, un flag d'utilisation du symbole ('€', '\$') ou le code ISO.
- Optionnellement, une chaîne de formatage du montant.

```
<p>{{ 10.6 | currency:'EUR' }}</p>
<!-- will display 'EUR10.60' -->

<p>{{ 10.6 | currency:'USD':true }}</p>
<!-- will display '$10.60' -->

<p>{{ 10.6 | currency:'USD':true:'.'3' }}</p>
<!-- will display '$10.600' -->
```

DatePipe

Le format spécifié peut être soit un pattern comme 'dd/MM/yyyy' ou 'MM-yy', soit un des formats prédéfinis.

```
<p>{{ birthday | date:'dd/MM/yyyy' }}</p>
<!-- will display '16/07/1986' -->

<p>{{ birthday | date:'longDate' }}</p>
<!-- will display 'July 16, 1986' -->

<p>{{ birthday | date:'HH:mm' }}</p>
<!-- will display '15:30' -->

<p>{{ birthday | date:'shortTime' }}</p>
<!-- will display '3:30 PM' -->
```

Utilisation depuis le code.

Les Pipes sont également utilisable depuis le code au moyen de la `DI`

```
//...
constructor(jsonPipe: JsonPipe) {
  this.value = jsonPipe.transform(this.value);
}
```

10.3/ Directives de transformation.

Directives Natives

- `NgStyle` - Modification de l'attribut **style** d'un élément.
- `NgClass` - Modification de l'attribut **style** d'un élément.
- `NgIf` - Conditionne le **rendu** d'un composant.
- `NgFor` - Conditionne le **rendu par itération** d'un composant.
- `NgSwitch` - Conditionne le **rendu** de plusieurs composants.

NgStyle

```
<p style="padding: 1rem"
    [ngStyle]="{
      color: 'red',
      'font-weight': 'bold',
      borderBottom: borderStyle
    }">
  <ng-content></ng-content>
</p>
```

NgClass

```
<!-- toggle the "special" class on/off with a property -->
<div [class.special]="isSpecial">The class binding is special</div>
```

NgIf

```
<div *ngIf="currentHero">Hello, {{currentHero.firstName}}</div>
```

NgFor

```
<!-- html -->
<div *ngFor="let hero of heroes">{{hero.fullName}}</div>
<!-- Component -->
<hero-detail *ngFor="let hero of heroes" [hero]="hero"></hero-detail>
<!-- Index -->
<div *ngFor="let hero of heroes; let i=index">{{i + 1}} - {{hero.fullName}}</div>
<!-- NgForTrackBy -->
<div *ngFor="let hero of heroes; trackBy:trackByHeroes">({{hero.id}}) {{hero.fullName}}
```

NgSwitch

1. ngSwitch: Expression de **base** de la condition.
2. ngSwitchCase: Expression de **test** condition.
3. ngSwitchDefault

```
<span [ngSwitch]="toeChoice">
  <span *ngSwitchCase="'Eenie'">Eenie</span>
  <span *ngSwitchCase="'Meanie'">Meanie</span>
  <span *ngSwitchCase="'Miney'">Miney</span>
  <span *ngSwitchCase="'Moe'">Moe</span>
  <span *ngSwitchDefault>other</span>
</span>
```

Directives Personnalisées

Il est possible de créer des directives personnalisées se basant de préférence sur un sélecteur par attribut (`attribute directive`)

A noter: préfixer les directive d'un identifiant autre que `ng` afin de les identifier plus facilement.

```
export class HighlightDirective {
  private _defaultColor = 'red';

  constructor(private el: ElementRef, private renderer: Renderer) { }

  @Input('myHighlight') highlightColor: string;

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || this._defaultColor);
  }
  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', color);
  }
}
```

L'ajout de l' `@Input` offre la personnalisation de la directive.

```
<p [myHighlight]="color">Highlight me!</p>
```

Utilisation

```
<p myHighlight>Highlight me!</p>
```

10.4/ Définition syntaxique, le symbole (*). Variables locales.

Attention les directive ayant un impact sur la structure même du Component Tree sont préfixées de *.

Il est important de ne pas omettre les symbole *

Les **Variable locale** ou (Template reference variables) sont des références au DOM ou à une directive exprimées dans le template.

```
<!-- phone refers to the input element; pass its `value` to an event handler -->
<input #phone placeholder="phone number">
<button (click)="callPhone(phone.value)">Call</button>

<!-- fax refers to the input element; pass its `value` to an event handler -->
<input ref=fax placeholder="fax number">
<button (click)="callFax(fax.value)">Fax</button>
```

Créer une directive de structure

La démarche est la même que pour une directive d'attribut mais il faut indiquer à Angular l'impact sur la structure.

```
@Directive({
  selector: '[delay]'
})
export class DelayDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainerRef: ViewContainerRef
  ) { }

  @Input('delay')
  set delayTime(time: number): void {
    setTimeout(()=>{
      this.viewContainerRef.createEmbeddedView(this.templateRef);
    }, time);
  }
}
```

Utilisation

```
<div *ngFor="let item of [1,2,3,4,5,6]">
  <card *delay="500 * item">
    {{item}}
  </card>
</div>
```

10.5/ Configuration de directives: selector, provider.

- `exportAs`
- `host`
- `inputs`
- `outputs`
- `providers`
- `queries`
- `selector`

De toutes les [options de configuration](#) des directives, le `selector` offre la syntaxe la plus riche.

En effet la syntaxe s'appuie sur les sélecteurs CSS. Cela permet de personnalier l'application de la directive.

Exemples

```
@Directive({
  selector: 'div.active[delay]'
})

@Directive({
  selector: 'div.active[delay]:not([hidden])'
})
```

10.6/ Directives et évènements utilisateur.

Les décorateurs `@HostListener` permet de se référer au DOM accueillant la directive.

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight('yellow');
}

@HostListener('mouseleave') onMouseLeave() {
  this.highlight(null);
}

private highlight(color: string) {
  this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', color);
}
```



Gestion des formulaires, “Routing” et requête HTTP

11/ Gestion des formulaires, “Routing” et requête HTTP

Angular2 en tirant parti du **Two Way Data-Binding** reproduit le cycle de validation des formulaires et en simplifie la programmation.

11.1/ NgForm et FormControl et FormGroup.

La détection des balise `form` et `input NgModel` entraînera l'instanciation d'un **NgForm** contenant les **FormControl**

A noter la clé ou attribut `name` est indispensable.

Création d'un composant de formulaire.

```
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule
  ],
  declarations: [
    AppComponent,
    HeroFormComponent
  ],
  bootstrap: [ AppComponent ]
})
```

A noter l'application devra exposer `FormsModule`

```

import { Component } from '@angular/core';

class Hero {
  constructor(
    public id: number,
    public name: string,
    public power: string,
    public alterEgo?: string
  ) { }
}

@Component({
  moduleId: module.id,
  selector: 'hero-form',
  template: `<div class="container">
    <h1>Hero Form</h1>
    <form>
      <div class="form-group">
        <label for="name">Name</label>
        <input type="text" class="form-control" id="name"
          required
          [(ngModel)]="model.name" name="name"
          #spy >
        <br>TODO: remove this: {{spy.className}}
      </div>
      <div class="form-group">
        <label for="alterEgo">Alter Ego</label>
        <input type="text" class="form-control" id="alterEgo"
          [(ngModel)]="model.alterEgo" name="alterEgo">
      </div>
      <div class="form-group">
        <label for="power">Hero Power</label>
        <select class="form-control" id="power"
          required
          [(ngModel)]="model.power" name="power">
          <option *ngFor="let p of powers" [value]="p">{{p}}</option>
        </select>
      </div>
      <button type="submit" class="btn btn-default">Submit</button>
    </form>
  </div>`)
})

export class HeroFormComponent {
  powers = ['Really Smart', 'Super Flexible',
            'Super Hot', 'Weather Changer'];
  model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');
  submitted = false;
  onSubmit() { this.submitted = true; }
  // TODO: Remove this when we're done
  get diagnostic() { return JSON.stringify(this.model); }
}

```

Ce formulaire basée sur le template est considéré `Template Driven From`.

La validité du formulaire se fait sur la condition de validité de tous ses composant.

Chaque composant de formulaire `NgModel` reçoit les propriété :

- **valid/invalid**
- **pristine/dirty**
- **touched/untouched**

A noter chacune de ses propriétés sont aussi représentée dans le DOM sous la forme de classe CSS préfixées de `ng-`

Il est possible de prévoir les CSS correspondant :

```
.ng-valid[required], .ng-valid.required {  
    border-left: 5px solid #42A948; /* green */  
}  
  
.ng-invalid:not(form) {  
    border-left: 5px solid #a94442; /* red */  
}
```

Afficher / Masquer un message de validation

L'utilisation d'une **variable locale** permet de simplement conditionner la structure sur l'état de validité d'un contrôle.

```
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name"
       #name="ngModel" >
<div [hidden]="name.valid || name.pristine"
      class="alert alert-danger">
  Name is required
</div>
```

Soumission du formulaire

L'événement **ngSubmit** détermine la méthode à déclenché lors de la validation du formulaire.

```
<form *ngIf="active" (ngSubmit)="onSubmit()" #myForm="ngForm">
  <button type="submit" class="btn btn-default" [disabled]="!heroForm.form.valid">Submit</button>
</form>
```

11.2/ Validation et gestion d'erreur personnalisée.

Les messages d'erreurs s'appuient sur le cycle de validation HTML5.

- **required**
- **minlength**
- **maxlength**
- **pattern**

Il est possible de définir un object d'erreurs dans la `class`.

```
formErrors = {  
  'name': '',  
  'power': ''  
};
```

Le dictionnaire `formErrors` recherchera le message de validation correspondant.

```
validationMessages = {  
  'name': {  
    'required': 'Name is required.',  
    'minlength': 'Name must be at least 4 characters long.',  
    'maxlength': 'Name cannot be more than 24 characters long.',  
    'forbiddenName': 'Someone named "Bob" cannot be a hero.'  
  },  
  'power': {  
    'required': 'Power is required.'  
  }  
};
```

Utilisation

```
<div *ngIf="formErrors.name" class="alert alert-danger">  
  {{ formErrors.name }}  
</div>
```

Règle de validation personnalisée

```
import { Directive } from '@angular/core';

@Directive({
  selector: '[validateEmail][ngModel]'
})
export class EmailValidator {}
```

```
import { Directive } from '@angular/core';

@Directive({
  ...
})
export class EmailValidator {

  validator: Function;

  constructor(emailBlackList: EmailBlackList) {
    this.validator = validateEmailFactory(emailBlackList);
  }

  validate(c: FormControl) {
    return this.validator(c);
  }
}
```

11.3/ “FormBuilder”, composants avancés de formulaire.

Le module `@angular/forms` expose `ReactiveFormsModule` permettant la construction dynamique d'un formulaire de puis les données ou `Data Driven Form`

Exemple Détailé

```
import { Component, Input, OnInit } from '@angular/core';
import { FormGroup } from '@angular/forms';

import { QuestionBase } from './question-base';
import { QuestionControlService } from './question-control.service';
@Component({
  moduleId: module.id,
  selector: 'dynamic-form',
  templateUrl: 'dynamic-form.component.html',
  providers: [ QuestionControlService ]
})
export class DynamicFormComponent implements OnInit {
  @Input() questions: QuestionBase<any>[] = [];
  form: FormGroup;
  payLoad = '';
  constructor(private qcs: QuestionControlService) { }
  ngOnInit() {
    this.form = this.qcs.toFormGroup(this.questions);
  }
  onSubmit() {
    this.payLoad = JSON.stringify(this.form.value);
  }
}
```

```
<div>
  <form (ngSubmit)="onSubmit()" [formGroup]="form">
    <div *ngFor="let question of questions" class="form-row">
      <df-question [question]="question" [form]="form"></df-question>
    </div>
    <div class="form-row">
      <button type="submit" [disabled]="!form.valid">Save</button>
    </div>
  </form>
  <div *ngIf="payLoad" class="form-row">
    <strong>Saved the following values</strong><br>{{payLoad}}
  </div>
</div>
```

11.4/ Liaison de données via HTTP.

Angular 2 expose un module http, en laissant place aux alternatives, comme les WebSockets.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpModule } from '@angular/http';

@NgModule({
  imports: [BrowserModule, HttpModule],
  declarations: [PonyRacerAppComponent],
  bootstrap: [PonyRacerAppComponent]
})
export class AppModule {
```

Par défaut, le service **Http** réalise des requêtes AJAX avec XMLHttpRequest.
Il propose plusieurs méthodes, correspondant au verbes HTTP communs :

- get
- post
- put
- delete
- patch
- head

Pour accéder au corps de la réponse on utilise les méthodes :

```
text()  
json()
```

Le service **Http** retourne par défaut un observable

```
this.http.get(`http://backand.com/posts`)
  .map(res => res.json())
  .catch(this.handleError);
```

Il est possible de revenir à une réponse sous la forme de `Promise`

```
this.http.get(this.heroesUrl)
    .toPromise()
    .then(this.extractData)
    .catch(this.handleError);
```

Envoyer des données au serveur

Il faut importer la `class Headers et RequestOptions` pour parémétrer la requête.

```
import { Headers, RequestOptions } from '@angular/http';
```

```
let body = JSON.stringify({ name });
let headers = new Headers({ 'Content-Type': 'application/json' });
let options = new RequestOptions({ headers });

return this.http.post(this.heroesUrl, body, options)
    .map(this.extractData)
    .catch(this.handleError);
```

11.5/ Création de routes. Paramétrage et wildcard.

Le routage applicatif débute par la mise en oeuvre de l'url de base de l'application, fixé au niveau HTML.

```
<base href="/">
```

Il est courant de vouloir associer une URL à un état de l'application. La partie en charge de ce travail s'appelle un routeur.

```
import { RouterModule } from '@angular/router';
```

Configuration des routes

Les `path` définissent l'association d'un composant pour une URL.

- **path** : URL déclenchant la navigation.
- **component** : Composant à initialiser et afficher.

```
@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot([
      { path: 'hero/:id', component: HeroDetailComponent },
      { path: 'crisis-center', component: CrisisListComponent },
      {
        path: 'heroes',
        component: HeroListComponent,
        data: {
          title: 'Heroes List'
        }
      },
      { path: '', component: HomeComponent },
      { path: '**', component: PageNotFoundComponent }
    ])
  ],
  declarations: [
    AppComponent,
    HeroListComponent,
    HeroDetailComponent,
    CrisisListComponent,
    PageNotFoundComponent
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {
```

11.6/ Ciblage, “router-outlet” événements de routage.

Par défaut le Routeur cible un tag nommé `<router-outlet>`.

```
<!-- Routed views go here -->
<router-outlet></router-outlet>
```

Pour changer de route il faut **spécifiquement** informer le routeur via `routerLink`.

`routerLinkActive` permet de spécifier une `class CSS`.

Il est possible d'animer les [transitions de route](#)

```
<h1>Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<router-outlet></router-outlet>
```

Le routage fonctionne en arbre comme les composants et une route peu déclencher une cascade de mise à jour vers des [child routes](#).



Tests unitaires.

12/ Tests unitaires. Bonnes pratiques et outils.

Les tests Angular (recommandés avec Jasmine) peuvent s'appuyer sur l'utilitaire `karma`.

Anciennement connu sous le nom de Testacular. `karma` un gestionnaire de suites de tests.

Installation

```
npm i -g karma-cli  
npm i karma  
karma init  
karma start
```

12.1/ Configurer l'environnement de test.

La mise en œuvre de `karma` avec Angular s'appuie sur un ensemble de fichiers de définition.

- `karma.conf.js`
- `karma-test-shim.js`
- `systemjs.config.js`
- `systemjs.config.extras.js`

Bien que l'utilitaire `angular-cli` prépare l'environnement pour le développeur, il est utile d'en comprendre la structure.

Par convention les fichiers de test ou `specs` sont nommés sur le modèle `*.spec.ts`.

Le framework inclut des utilitaires `@angular/core/testing` et particulièrement une `class` nommée `TestBed` facilitant l'isolation d'un module.

On appelle `fixture` le composant fonctionnel extrait du module pour être testé.

```
beforeEach(() => {

  // refine the test module by declaring the test component
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  });

  // create component and test fixture
  fixture = TestBed.createComponent(BannerComponent);

  // get test component from the fixture
  comp = fixture.componentInstance;
});

it('should display original title', () => {

  // trigger change detection to update the view
  fixture.detectChanges();

  // query for the title <h1> by CSS element selector
  de = fixture.debugElement.query(By.css('h1'));

  // confirm the element's content
  expect(de.nativeElement.textContent).toContain(comp.title);
});
```

12.2/ Ecrire les tests avec Jasmine. Couverture.

Jasmine n'est pas le seul Framework qui permet de créer des tests en JavaScript, il en existe d'autres tel que Mocha ou Tape.

Jasmine offre deux **ensembles syntaxique** à travers différentes fonction.

La structuration de la suite de test :

describe : famille de test.

it : pas de test.

```
describe('Multiplication', function(){
    it('should work', function(){
        expect(value).toBe(valueTest)
    });
});
```

La [documentation](#) de jasmine est complète et simple.

jasmine.done

Le `callback done()` fournit par jasmine permet de retarder l'exécution d'un test.

```
beforeEach(function(done) {
    setTimeout(function() {
        value = 0;
        done();
    }, 1);
});
```

This spec will not start until the done function **is called in the call to beforeEach** a

```
it("should support async execution of test preparation and expectations", function(done) {
    value++;
    expect(value).toBeGreaterThan(0);
    done();
});
```

12.3/ Cas de test : pipe, composant, application.

Il existe différentes [stratégies de test](#)

Composant

```
let comp: BannerComponent;
let fixture: ComponentFixture<BannerComponent>;
let de: DebugElement;
let el: HTMLElement;

describe('BannerComponent', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ],
      /* pour ne pas activer manuellement le cycle de détection il est possible d'ajouter
       providers: [
         { provide: ComponentFixtureAutoDetect, useValue: true }
       ]
    */
  });
  fixture = TestBed.createComponent(BannerComponent);
  comp = fixture.componentInstance; // BannerComponent Instance

  // query for the title <h1> by CSS element selector
  de = fixture.debugElement.query(By.css('h1'));
  el = de.nativeElement;
  });
});

it('should display original title', () => {
  fixture.detectChanges();
  /* pour ne pas activer manuellement le cycle de détection il est possible d'ajouter
   providers: [
     { provide: ComponentFixtureAutoDetect, useValue: true }
   ]
  */
  expect(el.textContent).toContain(comp.title);
});

it('should display a different test title', () => {
  comp.title = 'Test Title';
  fixture.detectChanges();
  expect(el.textContent).toContain('Test Title');
});
```

Composant avec dépendance

Stratégie simuler la dépendance.

```
beforeEach(() => {
  // Mock de la dépendance
  userServiceStub = {
    isLoggedIn: true,
    user: { name: 'Test User' }
  };

  TestBed.configureTestingModule({
    declarations: [ WelcomeComponent ],
    //Redéfinition de la dépendance
    providers: [ {provide: UserService, useValue: userServiceStub} ]
  });

  fixture = TestBed.createComponent(WelcomeComponent);
  comp = fixture.componentInstance;

  // UserService from the root injector
  userService = TestBed.get(UserService);

  // get the "welcome" element by CSS selector (e.g., by class name)
  de = fixture.debugElement.query(By.css('.welcome'));
  el = de.nativeElement;
});

it('should welcome the user', () => {
  fixture.detectChanges();
  const content = el.textContent;
  expect(content).toContain('Welcome', '"Welcome ..."]');
  expect(content).toContain('Test User', 'expected name');
});

it('should welcome "Bubba"', () => {
  userService.user.name = 'Bubba'; // welcome message hasn't been shown yet
  fixture.detectChanges();
  expect(el.textContent).toContain('Bubba');
});

it('should request login if not logged in', () => {
  userService.isLoggedIn = false; // welcome message hasn't been shown yet
  fixture.detectChanges();
  const content = el.textContent;
  expect(content).not.toContain('Welcome', 'not welcomed');
  expect(content).toMatch(/log in/i, '"log in"');
});
```

Composant avec service asynchrone

Stratégie intercepter le service.

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ TwainComponent ],
    providers:    [ TwainService ],
  });

  fixture = TestBed.createComponent(TwainComponent);
  comp    = fixture.componentInstance;

  // TwainService actually injected into the component
  twainService = fixture.debugElement.injector.get(TwainService);

  // Les SPY de Jasmine permettent d'intercepter la méthode du service.
  spy = spyOn(twainService, 'getQuote')
    .and.returnValue(Promise.resolve(testQuote));

  // Get the Twain quote element by CSS selector (e.g., by class name)
  de = fixture.debugElement.query(By.css('.twain'));
  el = de.nativeElement;
});
```

Composant avec templateUrl

Stratégie utiliser la méthode `.compileComponents()`.

La fonction `async` est un utilitaire simplifiant la gestion asynchrone du code.

```
beforeEach( async(() => {
  TestBed.configureTestingModule({
    declarations: [ DashboardHeroComponent ],
  })
  .compileComponents(); // compile template and css
}));
```

service

```
it('#getTimeoutValue should return timeout value', done => {
  service = new FancyService();
  service.getTimeoutValue().then(value => {
    expect(value).toBe('timeout value');
    done();
  });
});
```

Pipe

```
describe('TitleCasePipe', () => {
  // This pipe is a pure, stateless function so no need for BeforeEach
  let pipe = new TitleCasePipe();
  it('transforms "abc" to "Abc"', () => {
    expect(pipe.transform('abc')).toBe('Abc');
  });
  it('transforms "abc def" to "Abc Def"', () => {
    expect(pipe.transform('abc def')).toBe('Abc Def');
  });
  // ... more tests ...
});
```

12.4/ AngularJS2 “Coding guide Style”.

L'équipe d'**Angular2** à pris le soin de réaliser un guide des [bonnes pratiques](#) et de fournir une [cheat sheet](#)

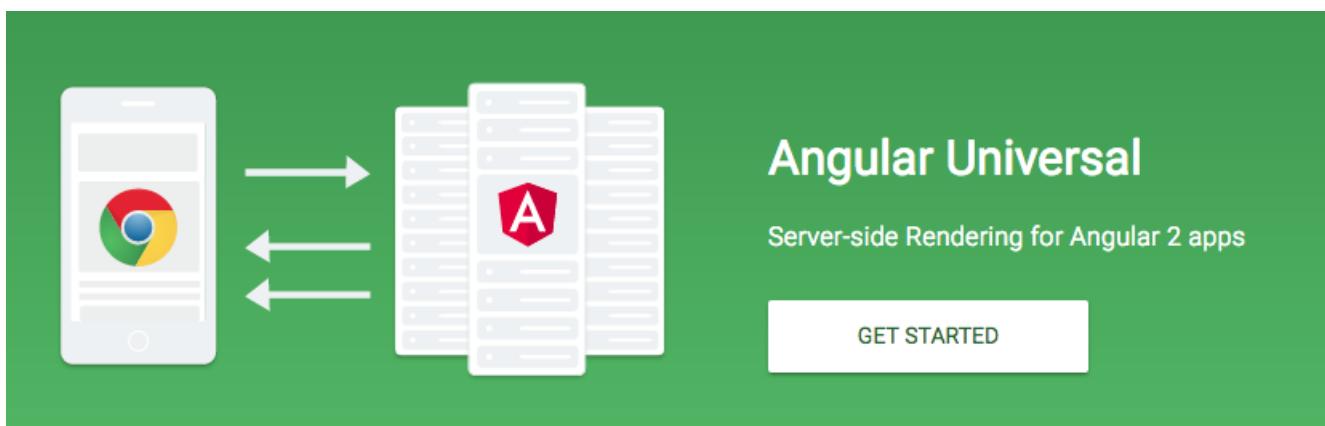
Pour les “[CookBook](#)” présente les `boilerplate` commun.

Evolution

Angular est conçu comme une plate-forme à destination multiple.

universal (universal.angular.io)

Disponible sur sur **NodeJS** et **.NET** implémente le **pré rendu applicatif** au niveau du serveur.



```
> npm install body-parser angular2-universal preboot express --save  
> typings install node express body-parser serve-static express-serve-static-core mime
```

Code du serveur

```
import 'angular2-universal/polyfills';
import * as path from 'path';
import * as express from 'express';

// Angular 2 Universal
import {provideRouter} from '@angular/router';
import {enableProdMode} from '@angular/core';
import {
  expressEngine,
  BASE_URL,
  REQUEST_URL,
  ORIGIN_URL,
  NODE_LOCATION_PROVIDERS,
  NODE_HTTP_PROVIDERS,
  ExpressEngineConfig
} from 'angular2-universal';

// replace this line with your Angular 2 root component
import {App, routes} from './app';

const app = express();
const ROOT = path.join(path.resolve(__dirname, '..'));

enableProdMode();

// Express View
app.engine('.html', expressEngine);
app.set('views', __dirname);
app.set('view engine', 'html');

function ngApp(req, res) {
  let baseUrl = '/';
  let url = req.originalUrl || '/';

  let config: ExpressEngineConfig = {
    directives: [ App ],

    // dependencies shared among all requests to server
    platformProviders: [
      {provide: ORIGIN_URL, useValue: 'http://localhost:3000'},
      {provide: BASE_URL, useValue: baseUrl},
    ],
    providers: [
      {provide: REQUEST_URL, useValue: url},
      provideRouter(routes),
      NODE_LOCATION_PROVIDERS,
      NODE_HTTP_PROVIDERS,
    ],
    // if true, server will wait for all async to resolve before returning response
    async: true,
  };
}
```

```
// if you want preboot, you need to set selector for the app root
// you can also include various preboot options here (explained in separate document)
preboot: false // { appRoot: 'app' }
};

res.render('index', config);
}

// Serve static files
app.use(express.static(ROOT, {index: false}));

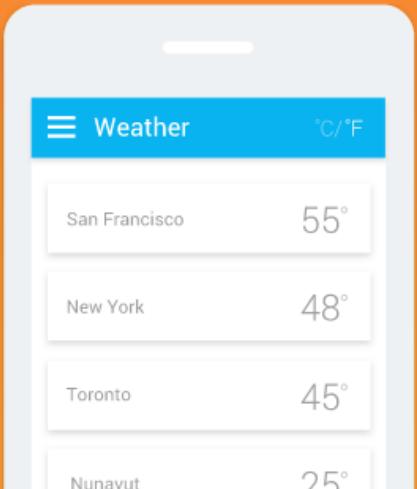
// send all requests to Angular Universal
// if you want Express to handle certain routes (ex. for an API) make sure you adjust
app.get('/', ngApp);
app.get('/home', ngApp);
app.get('/about', ngApp);

// Server
app.listen(3000, () => {
  console.log('Listening on: http://localhost:3000');
});
```

mobile (mobile.angular.io)

Cible le développement d'application mobile.

```
$ ng new hello-mobile --mobile  
$ cd hello-mobile  
$ ng serve
```



The screenshot shows a mobile application interface titled "Weather". At the top right is a "°C/°F" switch. Below the title, there are four cards, each containing a city name and its temperature: San Francisco (55°), New York (48°), Toronto (45°), and Nunavut (25°).

Angular Mobile Toolkit

All the tools and techniques to build high-performance mobile apps

[PREVIEW ON GITHUB](#)



...
