**Partie I**

**Notion de framework**

**C’est quoi un framework ?**

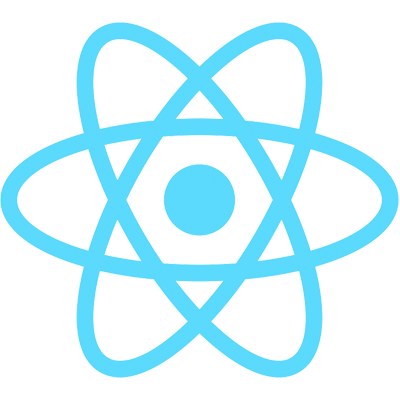
c’est un cadre de travail qui permet de créer des applications web et mobiles. Un framework propose une bibliothèque de fonctionnalités dans laquelle vos développeurs vont pouvoir piocher en fonction de vos besoins. En développement, l’utilisation d’un framework permet donc de gagner du temps et aujourd’hui, il s’agit d’un standard dans la construction d’un projet web ou mobile. Un framework est construit autour d’un langage de programmation : PHP, Python, javascript, Ruby, etc.Le découpage classique frontend/backend s’applique aussi aux frameworks.

## **Les frameworks frontend**

## Ces Frameworks servent à définir ce que va voir le visiteur lorsque qu’il va arriver sur le site. Ils utilisent des langages clients tels que le langage HTML, CSS ou JavaScript. En exemple de Framework front end on peut retrouver Vue.JS, React ou bien AngularJS.

* Etude comparative des Framework

### React :



Créé en 2013 par Facebook, React a été développé pour répondre à des problématiques de performance que rencontrait le site à l'époque. C’est un framework qui est donc léger et optimisé pour gérer un trafic très important. La dernière version en date est la 16.12.0 sortie en novembre 2019.

A noter quelque chose de très important quand on parle de React, c’est que c’est un framework minimaliste (pas seulement une librairie car on utilise notamment son CLI create-react-ap). En effet React n’a pas vocation à gérer tous les aspects de votre application. Il n’y a pas de gestion des formulaires, pas de gestion de l’état de l’application, pas de gestion du routing, pas de gestion de la validation etc. Il faudra à chaque fois utiliser des librairies indépendantes qui n’ont aucun soutien de la part de Facebook : par exemple Formik pour les formulaires, Redux pour la gestion d’état, React-Router pour le routing etc.

Évidemment si nous comprenons cela, nous savons pourquoi React est particulièrement accessible : le nombre de choses à connaître pour débuter est forcément bien moindre qu’un framework comme Angular qui inclut toutes ces fonctionnalités directement.

Cela signifie également que vous pouvez utiliser React très facilement sur un grand nombre d’applications, notamment ce qu’on appelle des applications serveurs (celles qu’on crée par exemple avec symfony, ruby on rails etc …).

### Vue :



Sorti en 2014, ce framework a été créé par un ancien Googler ayant travaillé plusieurs années avec Angular.js du nom de Evan You. Vue.js se présente comme étant la synthèse entre Angular.js et React. La popularité du framework ne cesse de grandir et il s’est fait une place de choix malgré la compétition féroce et les deux titans que sont Google et Facebook. La dernière version est la 2.6.11 sortie le 13 décembre 2019.

Une synthèse entre Angular et React, c’est-à-dire ?

Pour la comparaison avec React, Vue.js permet d'utiliser des composants réactifs et réutilisables avec une syntaxe simple qui est populaire et très simple à comprendre. Ils utilisent tous les deux de nombreuses optimisations comme le DOM virtuel.

La principale différence est que Vue est composé de plusieurs librairies que vous pouvez ajouter à votre application au fur et à mesure que vous avez besoin de plus de fonctionnalités. C’est pourquoi Vue.js est un framework progressif, car vous pourrez ajouter des parties du framework selon vos besoins.

Vue possède ainsi d’excellentes libraires comme par exemple [Vuex](https://github.com/vuejs/vuex) pour la gestion d'état et [vue-router](https://github.com/vuejs/vue-router) pour la création de routes sur votre application. La grande différence avec React, c’est que c’est la même équipe qui développe tous ces outils et vous êtes donc certains qu’ils sont totalement compatibles et bien maintenus.

### Angular :



Il y a deux angular, il y a l'ancêtre de tous les framework Front-End modernes, qu’on appelle Angular.js ou Angular 1, et il y a le nouvel Angular qu’on appelle simplement Angular ou Angular 2. Utilisez le terme officiel Angular car nous sommes aujourd’hui à la version 9 de ce framework et ce n’a plus aucun sens de l’appeler Angular 2.

Ces deux frameworks n’ont rien à voir et ici on parlera bien entendu d’A

ngular, Angular.js étant totalement abandonné et plus maintenu par Google.

Angular est officiellement sorti en 2016, c’est donc le framework le plus récent des trois. Angular a connu un début un petit peu difficile car les gens n’aiment pas forcément le changement et étaient habitués à la première version. Donc il fallait repartir de zéro, pour tout le monde, et il oblige à utiliser [Typescript](https://github.com/microsoft/TypeScript) un langage que peu de monde connaissait lors de sa sortie.

Nous sommes actuellement sur la version 9 du framework sortie en février 2020.

Angular, est un framework dit “opinionated” ou orienté, c’est-à-dire que Google pense qu’il y a une bonne manière de développer une application Web et organise toute le framework de cette manière. Lorsque nous utilisons Angular on doit donc développer d’une certaine façon, c’est le framework le plus orienté dans la mesure où il y a toujours une façon recommandée de faire.

Toutes les problématiques du Web sont gérées par le framework lui-même et il n’y a pas besoin de librairies pour les fonctionnalités communes des applications Web (requêtes HTTP, routing, formulaires, applications progressives pour mobile -PWA-, Web workers, rendu serveur -SSR-…).

Cela a plusieurs conséquences, des positives et des négatives.

Commençons par le négatif. Avec Angular vous avez assez peu de libertés sur la façon dont vous allez concevoir vos applications, contrairement à Vue ou à React qui laissent une grande liberté dans la conception de l'architecture. L’autre point important, c’est qu’il y a énormément plus de choses à apprendre pour maîtriser le framework, et que cela nécessite un investissement beaucoup plus important pour l’apprentissage.

Comme dit plus haut, vous allez nécessairement devoir utiliser Typescript, qui est un langage qui compile en JavaScript avec des fonctionnalités supplémentaires et un typage fort - qui permet une meilleure maintenabilité-. Vous allez également devoir vous familiariser avec RxJS qui est une librairie de programmation réactive pas facile à aborder lorsque l’on débute. En résumé il est plus long et plus difficile de développer en Typescript mais le code produit est de qualité supérieure car plus lisible et maintenable.

Avec Typescript, si vous connaissez le JavaScript vous pouvez vous en sortir en quelques heures, c’est vraiment très similaire.

Pour ce qui est de la programmation réactive, cela demande plus de temps pour la maîtriser car c’est vraiment un paradigme moderne qui n’est pas facile à comprendre. Toutes les entreprises technologiques utilisent ce nouveau paradigme (GAFAM et Netflix par exemple).

Maintenant les points positifs, vous arrivez sur une application Angular que vous n’avez jamais vu auparavant ? Pas de problèmes elles sont globalement toutes architecturées de la même façon. Comment travailler en équipe ? Pas de débats interminables pour savoir la bonne manière de faire, le framework met en oeuvre et recommande les bonnes pratiques. De même si vous ne savez pas trop la meilleure façon d’organiser votre code pas de panique : Angular vous l’impose.

Pour ce qui est de Typescript, que ce soit Node.js, React, Vue.js, tout l’écosystème JavaScript s’y met progressivement. C’est le langage le plus soutenu par la communauté professionnelle : il est développé par de [nombreux ingénieurs de Microsoft](https://github.com/microsoft/TypeScript/graphs/contributors) et [bénéficie du soutien de Google qui l'utilise largement](https://github.com/google/gts).

Si nous comparons avec React, par exemple. Les chances sont extrêmement élevées que ce soit ce qu’on appelle du spaghetti code - du code très difficile à maintenir car mal organisé-. Vous avez évidemment des très bons développeurs React et Vue.js bien sûr, mais c’est très variable et la qualité de votre application dépendra énormément du niveau de compétence des développeurs seniors de l’équipe.

**Conclusion** : Pas vraiment de bon ou mauvais point ici, juste des approches vraiment différentes. React est volontairement minimaliste, simple à utiliser et laisse une complète liberté aux développeurs. Angular est plus complet et complexe, et il est beaucoup plus orienté. Vue se situe entre les deux approches et il a l’avantage d’avoir une approche progressive en fonction des besoins.

* Choix du Framework Angular

### Un bref historique

En version courte, au début il y avait le JavaScript seul, puis les sites Web sont devenus complexes, une librairie est alors apparue, jQuery (2006) qui est devenue très populaire.

Mais elle ne résolvait pas le problème de l’architecture des projets et des problématiques avancées (comme des optimisations de rendus). Les sites Web devenaient encore plus complexes et les projets devenaient difficilement maintenables. Le premier framework AngularJS (2009) a donc été créé par Google pour résoudre ces problèmes.

Angular est le framework JavaScript qui a été développé par Google. Là encore, un grand groupe se cache derrière cet outil, ce qui peut rassurer la communauté des développeurs. En parlant de communauté, celle d’Angular est très importante et surtout très active. Elle permet donc à ce framework d’être régulièrement mis à jour afin de s’adapter aux nouveaux besoins.

Il s’agit d’un framework JavaScript très complet et très performant à la condition de savoir correctement l’utiliser. Et c’est là le problème. En effet, Angular est assez complexe à prendre en main. Il vous faudra un peu de persévérance. Mais cela vaut le coup. Comme React, Angular propose de nombreuses fonctionnalités, ce qui en fait un framework Js générique qui peut convenir pour la majorité des applications.

En bref : **il n’y a qu’un seul framework Angular**. La question des versions n’a pas de sens et résulte juste d’une confusion.

## **Historique des versions d’Angular**

Angular sort une nouvelle version majeure tous les 6 mois environ :

* [Angular 2](http://blog.angularjs.org/2016/09/angular2-final.html) est sorti en septembre 2016,
* [Angular 4](http://angularjs.blogspot.fr/2017/03/angular-400-now-available.html) en mars 2017,
* [Angular 5](https://blog.angular.io/version-5-0-0-of-angular-now-available-37e414935ced) en novembre 2017,
* [Angular 6](https://blog.angular.io/version-6-of-angular-now-available-cc56b0efa7a4) en mai 2018,
* [Angular 7](https://blog.angular.io/version-7-of-angular-cli-prompts-virtual-scroll-drag-and-drop-and-more-c594e22e7b8c) en octobre 2018,
* [Angular 8](https://blog.angular.io/version-8-of-angular-smaller-bundles-cli-apis-and-alignment-with-the-ecosystem-af0261112a27) en mai 2019,
* [Angular 9](https://blog.angular.io/version-9-of-angular-now-available-project-ivy-has-arrived-23c97b63cfa3) en février 2020,
* [Angular 10](https://blog.angular.io/version-10-of-angular-now-available-78960babd41) en juin 2020.
* ES6 et Typscript

TypeScript est un pur langage de programmation **open source** orienté objet. Il s'agit d'un **sur** - **ensemble** de JavaScript fortement typé qui se compile en JavaScript brut. TypeScript est développé et maintenu par **Microsoft** sous la licence **Apache 2** . Il n'est pas exécuté directement sur le navigateur. Il a besoin d'un compilateur pour compiler et générer dans un fichier JavaScript. Le fichier source TypeScript a l'extension ".ts". Nous pouvons utiliser n'importe quel fichier " **.js** " valide en le renommant en fichier ".ts". TypeScript est la version ES6 de JavaScript avec quelques fonctionnalités supplémentaires.

### **Histoire de TypeScript**

* **Anders Hejlsberg a** développé TypeScript. Il a été introduit pour la première fois au public au mois du **1er octobre 2012** . Après deux ans de développement interne chez Microsoft, la nouvelle version de TypeScript 0.9 est sortie en 2013. La version actuelle de TypeScript est **TypeScript 3.4.5** qui est sortie le **24 avril 2019** .

## **ES6**

* ECMAScript (ES) est une spécification de **langage de script** normalisée par **ECMA international** . Il a été créé pour standardiser JavaScript. Le langage de script ES contient de nombreuses implémentations, et la plus populaire est **JavaScript**. Les développeurs utilisent ECMAScript principalement pour **les scripts côté client** du World Wide Web (WWW).

# . Classe[▲](https://yahiko.developpez.com/tutoriels/introduction-typescript/)

La notion de classe introduite dans TypeScript anticipe la prochaine évolution de JavaScript (ECMAScript 6).

Définition d'une classe

Sélectionnez

**class** Animal {

name: string;

**constructor**(name: string) {

**this**.name = name;

}

shout(): string {

return "...";

}

}

Comme on peut le voir dans l'exemple ci-dessus, une classe *Animal* y est définie d'une façon proche de la plupart des langages orientés objet.

La classe *Animal* possède ici un attribut (*name*), elle définit un constructeur (*constructor*) et une méthode (*shout*). Son instanciation se fait à l'aide de l'opérateur *new* comme ceci :

Instanciation d'une classe

Sélectionnez

**var** animal = **new** Animal("pokemon");

De pair avec la notion de classe, TypeScript implémente la notion d'héritage simple par l'utilisation du mot-clé *extends*.

L'extension de la classe *Animal* de l'exemple précédent pourrait se faire ainsi :

Héritage

Sélectionnez

**class** Lion **extends** Animal {

sex: string;

**constructor**(name: string, sex: string) {

super(name);

**this**.sex = sex;

}

shout(): string {

return "Rooooaarrr!"

}

}

Cette nouvelle classe *Lion* ajoute un nouvel attribut *sex* à la classe *Animal* et redéfinit la méthode *shout*.

Puisque toutes les classes définies dans TypeScript sont considérées comme de nouveaux types, la classe *Lion* est du type *Lion*, et en vertu de l'héritage est aussi du type *Animal*.

En remarque, malgré cet apport orienté objet à la syntaxe initiale de JavaScript, il faut avoir conscience que les limitations intrinsèques de JavaScript se reflètent également dans TypeScript. Par exemple, la notion d'attribut privé (*private*) d'une classe qui existe dans la plupart des langages orientés objet, bien que syntaxiquement présente dans TypeScript, n'est pas véritablement restrictive dans la mesure où un attribut privé pourra malgré tout être utilisé en dehors de sa classe avec une approche dynamique.

# Interface[▲](https://yahiko.developpez.com/tutoriels/introduction-typescript/)

Une interface peut être vue tout d'abord comme une sorte de contrat minimum que doit respecter une structure de données en termes d'attributs et de méthodes. Cette structure de données peut être un objet {…} ou une classe.

Définition d'interfaces

Sélectionnez

**interface** I1 {

a: number;

}

**interface** I2 {

b: string;

}

L'interface *I1* ci-dessus indique que la structure de données implémentant *I1* doit avoir à minima un attribut nommé *a* de type *number*. L'interface *I2* indique que la structure de données l'implémentant doit avoir à minima un attribut nommé *b* de type *string*.

Une interface est considérée comme un type à part entière, ce qui signifie qu'il est possible de définir une fonction qui prendrait un paramètre de type *I1* comme l'exemple ci-dessous :

Interface en tant que type

Sélectionnez

**function** fct(x: I1) {

alert(x.a);

}

Cette fonction *fct* pouvant être appelée en passant en paramètre un objet ayant au moins un attribut nommé *a* :

Objet implémentant une interface

Sélectionnez

fct({ a: 20, z: -1 });

Un autre intérêt d'une interface, c'est qu'elle peut être considérée comme une classe n'ayant que des méthodes abstraites (ou pas de méthode du tout). Cela permet dans un langage comme TypeScript ne gérant que l'héritage simple, de se rapprocher du concept de l'héritage multiple, une classe pouvant implémenter (*implements*) plusieurs interfaces comme le montre l'exemple ci-dessous :

Classe implémentant deux interfaces

Sélectionnez

**class** C **implements** I1, I2 {

a: number;

b: string;

**constructor**(a: number, b: string) {

**this**.a = a;

**this**.b = b;

}

}

Une instance de cette classe *C* pourrait tout à fait être passée en paramètre de notre fonction *fct* définie plus haut.

Sélectionnez

**var** c = **new** C(15, "bonjour");

fct(c);

Enfin, comme nous venons de le voir, une interface pouvant être considérée comme une classe abstraite, il est possible de faire hériter une interface d'une autre à l'aide du mot-clé *extends* :

Héritage entre interfaces

Sélectionnez

**interface** I3 **extends** I2 {

c: boolean;

}

Dans l'exemple ci-dessus, la nouvelle interface *I3* dérive de l'interface *I2* définie plus haut en ajoutant un attribut *c* de type *boolean*.

## LES FONCTIONS FLÉCHÉES

La fonction fléchée, ou Arrow function, est une autre façon de définir une fonction de façon plus concise et donc de simplifier la syntaxe. Elle se présente de la façon suivante :

|  |  |
| --- | --- |
|  | (param1: string, param2: number): void => { |
|  | /\*implémentation de la fonction\*/ |
|  | } |

[**view raw**](https://gist.github.com/tminguy/556259c5ff0d6fb65d821a09021d5043/raw/cbf95ba94b5a5978123f832e51235f4f4add2a9b/.ts)[**.ts**](https://gist.github.com/tminguy/556259c5ff0d6fb65d821a09021d5043#file-ts) hosted with ❤ by [**GitHub**](https://github.com/)

Cette syntaxe se décompose de la façon suivante :

* Les paramètres sont définis de la même façon que la syntaxe classique. Ils sont écrits entre parenthèses et chaque paramètre est séparé par une virgule ;
* La flèche qui marque la séparation entre les paramètres et l’implémentation de la fonction ;
* Le corps de la fonction entre des accolades.

|  |  |
| --- | --- |
|  | let greeting = (name: string): string => { |
|  | return "Hi, " + name; |
|  | } |

[**view raw**](https://gist.github.com/tminguy/49c546b4459e14eab815db0b8b40919c/raw/cf75eef4cc8150798330e7f629df28e93eb12e1e/.ts)[**.ts**](https://gist.github.com/tminguy/49c546b4459e14eab815db0b8b40919c#file-ts) hosted with ❤ by [**GitHub**](https://github.com/)

Ici nous avons réécrit la fonction **greeting**en utilisant la syntaxe de la fonction flèche (écriture lambda).

|  |  |
| --- | --- |
|  | var numbers = [1,2,3,4,5,6]; |
|  |  |
|  | var evens = numbers.filter((value) => value % 2 === 0); |
|  |  |
|  | var odds = numbers.filter(function (value) { |
|  | value % 2 !== 0 |
|  | }); |

[**view raw**](https://gist.github.com/tminguy/79229cf4dd5622604ceccef392fad502/raw/dcd80e97bcb9af84aa2c88739b6dd4e2fb08f64d/.ts)[**.ts**](https://gist.github.com/tminguy/79229cf4dd5622604ceccef392fad502#file-ts) hosted with ❤ by [**GitHub**](https://github.com/)

NB : la fonction flèche en Typescript possède une gestion différente du mot clé **this**par rapport à la syntaxe classique mais nous détaillerons ce point dans la section dédiée.

* Architecture d’Angular

Angular est un framework permettant de créer des applications clientes en HTML et en JavaScript ou dans un langage tel que Type Script qui se compile en JavaScript.

Le framework se compose de plusieurs bibliothèques, certaines d'entre elles de base et d'autres facultatives.

Vous écrivez des applications angulaires en composant des *modèles* HTML, en écrivant *des* classes de *composants* pour gérer ces modèles, en ajoutant une logique d'application dans les *services* et en intégrant des composants et des services dans des *modules*.

*Ensuite, vous lancez l'application en*amorçant*le*module racine*. Angular présentant le contenu de votre application dans un navigateur et répondant aux interactions des utilisateurs selon les instructions que vous avez fournies.*

**Pour l'instant, concentrez-vous sur la vue d'ensemble.**

Image pour publication

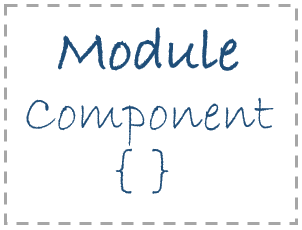
## Image pour publication

**Les huit principaux blocs de construction d'une application angulaire:**

* [Modules](https://v2.angular.io/docs/ts/latest/guide/architecture.html#modules)
* [Composants](https://v2.angular.io/docs/ts/latest/guide/architecture.html#components)
* [Modèles](https://v2.angular.io/docs/ts/latest/guide/architecture.html#templates)
* [Métadonnées](https://v2.angular.io/docs/ts/latest/guide/architecture.html#metadata)
* [Liaison de données](https://v2.angular.io/docs/ts/latest/guide/architecture.html#data-binding)
* [Directives](https://v2.angular.io/docs/ts/latest/guide/architecture.html#directives)
* [Prestations de service](https://v2.angular.io/docs/ts/latest/guide/architecture.html#services)
* [Injection de dépendance](https://v2.angular.io/docs/ts/latest/guide/architecture.html#dependency-injection)

**Modules**

Image pour publication



Chaque application Angular possède un *module racine*, appelé conventionnellement App Module, qui fournit le mécanisme d'amorçage qui lance l'application. Une application contient généralement de nombreux modules fonctionnels.

// app.module.ts importer {Browser Module} depuis '@ angular / platform-browser';   
import {NgModule} depuis '@ angular / core';import {App Routing Module} depuis './app-routing.module';   
import {App Component} depuis './app.component';@NgModule ({   
 déclarations: [   
 AppComponent   
 ],   
 importations: [   
 BrowserModule,   
 AppRoutingModule   
 ],   
 fournisseurs: [],   
 bootstrap: [AppComponent]   
})   
classe d'exportation AppModule {}

Si nous voulons utiliser un autre module angulaire personnalisé, nous devons enregistrer ce module dans le fichier **app.module.ts** . L'organisation de votre code en modules fonctionnels distincts aide à gérer le développement d'applications complexes et à concevoir pour la réutilisabilité.

**Bibliothèques angulaires**

Angular est livré comme une collection de modules JavaScript. Vous pouvez les considérer comme des modules de bibliothèque.

Chaque nom de bibliothèque angulaire commence par le @angular préfixe.

Vous les installez avec le **gestionnaire de** packages **npm** et en importez des parties avec des import instructions JavaScript.

Par exemple, importez le Component décorateur Angular de la @angular/core bibliothèque comme ceci :

Import {Component} de '@ angular / core' ;

Vous importez également des *modules* angulaires à partir de *bibliothèques* angulaires à l’aide d'instructions d'importation JavaScript:

Importer {Browser Module} depuis '@ angular / platform-browser';

Dans l'exemple du module racine simple ci-dessus, le module d'application a besoin de matériel provenant de celui-ci Browser Module. Pour accéder à ce matériel, ajoutez-le aux @Ng Module métadonnées imports comme ceci.

Importations : [BrowserModule],

De cette façon, vous utilisez les deux systèmes de modules angulaires et JavaScript *ensemble* .

**Composants**

Chaque projet Angular a au moins un composant, le *composant racine et* le composant racine connectent la hiérarchie des composants à un modèle d'objet de document de page (DOM). Chaque composant définit la classe qui contient les données et la logique de l'application, et il est associé au modèle HTML qui définit la vue à afficher dans une application cible. Un *composant* contrôle un patch d'écran appelé *vue*.

Le décorateur @Component identifie la classe immédiatement en dessous comme composant et fournit le modèle et les métadonnées spécifiques au composant associé.

*// app.component.ts*@Component ({   
 sélecteur: 'app-root',   
 template Url: './app.component.html',   
 style Urls: ['./app.component.css']   
})

**Modèles**

Image pour publication



Le modèle angulaire combine le HTML avec le balisage angulaire qui peut modifier les éléments HTML avant qu'ils ne soient affichés. Les directives de modèle fournissent la logique du programme et le balisage de liaison connecte vos données d'application et le DOM. Il existe deux types de liaison de données.

* **La liaison d'événement** permet à votre application de répondre aux entrées de l'utilisateur dans l'environnement cible en mettant à jour les données de votre application.
* **La liaison de propriété** vous permet d'interpoler les valeurs calculées à partir des données de votre application dans le HTML.

<div style = "text-align: center">   
 <h1>   
 {{2 | puissance: 5}}   
 </h1>   
</div>

Dans le fichier HTML ci-dessus, nous avons utilisé un modèle. Nous avons également utilisé le tube à l'intérieur du modèle pour transformer les valeurs en sortie souhaitée.

**Métadonnées**

Image pour publication

Image pour publication

Les métadonnées indiquent à Angular comment traiter une classe. Elles sont utilisées pour décorer la classe afin qu'elle puisse configurer le comportement attendu d'une classe. Les décorateurs sont le concept de base lors du développement avec Angular (versions 2 et supérieures). L'utilisateur peut utiliser les métadonnées d'une classe pour indiquer à l'application Angular qu'App Component est le composant. Les métadonnées peuvent être attachées au TypeScript à l'aide du décorateur.

*// app.component.ts*@Component ({   
 sélecteur: 'app-root',   
 templateUrl: './app.component.html',   
 styleUrls: ['./app.component.css']   
})

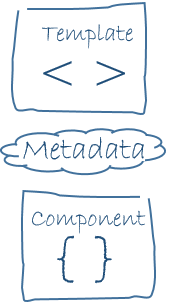
Voici le @Component décorateur, qui identifie la classe immédiatement en dessous comme une classe de composant.

Le @Component décorateur prend un objet de configuration requis avec les informations dont Angular a besoin pour créer et présenter le composant et sa vue.

Voici quelques-unes des @Component options de configuration les plus utiles :

* selector: Sélecteur CSS qui indique à Angular de créer et d'insérer une instance de ce composant là où il trouve une balise <app-root> qui est Parent Component.
* Template Url : adresse relative au module du modèle HTML de ce composant.
* Providers : tableau de **fournisseurs d'injection** de **dépendances** pour les services requis par le composant.

Image pour publication



Les métadonnées dans le @Component indique à Angular où obtenir les principaux blocs de construction que vous spécifiez pour le composant.

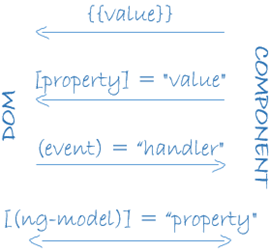
Le modèle, les métadonnées et le composant décrivent ensemble une vue.

Appliquez d'autres décorateurs de métadonnées de la même manière pour guider le comportement angulaire. @Injectable, @Input et @Output sont quelques-uns des décorateurs les plus populaires.

**Liaison de données**

La liaison de données joue un rôle important dans la communication entre un modèle et son composant.

Image pour publication



La liaison de données est également importante pour la communication entre les composants parents et enfants.Angular permet de définir la communication entre un composant et le DOM, ce qui facilite la définition d'applications interactives sans se soucier d'extraire et de pousser les données.

**Du composant au DOM**

Interpolation : {{value}}: l'interpolation ajoute la valeur de la propriété du composant.

<p> Nom: {{student.name}} </p>   
<p> Collège: {{student.college}} </p>

**Liaison de propriété : [propriété] = "valeur"**

Avec la liaison de propriété, une valeur est transmise d'un composant à une propriété spécifiée, qui peut souvent être un simple attribut html.

<input type = "text" [value] = "student.name" />   
<input type = "text" [value] = "student.college" />

**Liaison d’événement : (Event) = "myFunction ($ event)"**

Avec la liaison d'événement, une valeur est transmise d'un composant enfant au composant parent, qui peut souvent être un simple attribut html.

<input type = "text" (my Event) = "onClick ($ event)" />

**Liaison bidirectionnelle :**[(ngModel)] **= "propriété"**

C'est une quatrième forme importante qui combine la liaison de propriété et d'événement dans une seule notation, en utilisant la directive ngModel.

<input [(ngModel)] = "hero.name">

**Directives**

Image pour publication



Un composant angulaire n'est pas plus qu'une directive avec le modèle. Quand nous disons que les composants sont les blocs de construction des applications Angular, nous disons que les directives sont les blocs de construction des projets Angular. Utilisons la directive Angular intégrée comme ngClass, qui est un meilleur exemple de la directive d'attribut Angular existante.

<p [ngClass] = "{'coffee' = true, 'red' = false}">   
 Exemple de directives angulaires 7   
</p><style>   
 .coffee {color: coffee}   
 .red {color: red}   
</style>

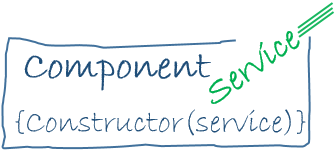
Ici, en fonction de la valeur de la directive [ngClass], le texte a une couleur. Dans notre exemple, le texte sera café parce que c'est vrai.

**Prestations de service**

Pour les données ou la logique qui ne sont pas associées à une vue spécifique et que vous souhaitez partager entre les composants, vous créez une classe de service. Le décorateur @Injectable précède immédiatement la définition de la classe de service. Le décorateur fournit les métadonnées qui permettent à votre service d'être *injecté* dans les composants client en tant que dépendance. Angular distingue les composants des services pour augmenter la modularité et la réutilisabilité. En séparant les fonctionnalités liées à la vue d'un composant des autres types de traitement, vous pouvez rendre vos classes de composants allégées et efficaces.

**Injection de dépendance**

Image pour publication



L'injection de dépendances (DI) vous permet de garder vos classes de composants simples et efficaces. DI ne récupère pas les données d'un serveur, ne valide pas l'entrée de l'utilisateur ou ne se connecte pas directement à la console au lieu de cela, il délègue ces tâches aux services. DI est câblé dans un cadre angulaire et utilisé partout pour fournir de nouveaux composants avec les services ou autres éléments dont ils ont besoin. Les composants consomment des services ; autrement dit, vous pouvez *injecter* un service dans un composant, en donnant au composant l'accès à cette classe de service.

**Partie II**

**Directives(Templates, Structurelles)**

Les directives structurelles sont responsables de la mise en page HTML.Ils façonnent ou remodèlent la*structure*du DOM, généralement en ajoutant, supprimant ou manipulant des éléments.

Comme avec les autres directives, vous appliquez une directive structurelle à un*élément hôte*.La directive fait alors tout ce qu'elle est censée faire avec cet élément hôte et ses descendants.

Les directives structurelles sont faciles à reconnaître.Un astérisque (\*) précède le nom de l'attribut de directive comme dans cet exemple.



**Templating et data-binding**

Le Data Binding est un élément essentiel dans les frameworksde Single Page Application. Il permet de synchroniser la vue au modèle JavaScript sous-jacent. Voici un schéma général du fonctionnement du Data Binding

Angular2 a défini quatre sortes de Data Binding pour synchroniser le templateet le component. Il est ainsi possible de propager une donnée du component vers le DOM, du DOM vers le component et dans les deux sens. Ces formes de Data Binding sont communément nommées:

**Interpolation**: ce mécanisme permet de modifier le DOM à partir du modèle, si un changement est intervenu sur une valeur de ce dernier.

**PropertyBinding** : ce mécanisme permet de valoriser une propriété d'un composant ou d'une directive à partir du modèle, si un changement est intervenu sur une valeur de ce dernier.

**Event Binding**: ce mécanisme permet d'exécuter une fonction portée par un component suite à un évènement émis par un élément du DOM.

**Le "two-way" Data Binding** : c'est une combinaison du PropertyBinding et du Event Binding sous une unique annotation. Dans ce cas-là, le component se charge d'impacter le DOM en cas de changement du modèle et le DOM avertit le component d'un changement via l'émission d'un évènement. Le mécanisme se rapproche du fameux ngModelAngular1, mais avec des algorithmes de Data Binding différents.

**Composants(Définition, Passage de Paramètres entre**

**Components)**

Un composant est une classe, qui contrôle une portion de l'écran. Cette portion de l'écran contrôlée par le composant, on l'appelle une vue: une vue peut être l'ensemble de la page web, une fenêtre de tchat, une barre de navigation, etc. Cette vue est définie dans le templatedu composant.

Pour le passage de paramètres entre components, on peut:

Passer la référence d’un composant à un autre

Communication par le composant parent

Communication à travers le service

Passer la référence d’un composant à un autre

Cette solution doit être utilisée lorsque les composants ont une dépendance entre eux. Par exemple, dropdownet dropdowntoggle. Ils ne peuvent généralement pas exister l’un sans l’autre.

Nous allons créer le composant side-bar-togglequi aura le composant side-baren entrée et en cliquant sur le bouton bascule nous ouvrirons / fermerons le composant side-bar. Voici le code pertinent:



Communication par le composant parent

Peut être utilisé lorsqu’il est facile de contrôler l’état partagé entre des composants via leur composant parent et que vous ne souhaitez pas créer de nouveau service ou créer du code standard, à cause d’une variable.

La mise en oeuvrede cette approche est presque la même que la précédente, mais le composant side-bar-togglene reçoit pas de composant side-bar. Au lieu de cela, le composant parent contient la propriété sideBarIsOpenedqui est transmise au composant side-bar.

Communication par le composant parent





Communication à travers le service

Enfin, cette option est utile et doit être utilisée lorsque vous avez un composant contrôlé ou que son état est demandé à plusieurs instances.



Nous avons maintenant plusieurs emplacements dans l’application qui devront accéder à notre composant sibe-bar. Voyons comment nous le faisons. Nous allons maintenant créer side-bar.service.tsdonc nous aurons:

side-bar.service.ts

side-bar.component.ts

side-bar.component.html

Les services de la barre latérale auront une méthode et un événement à changer pour que chaque composant qui injectera ce service puisse être averti de l’ouverture ou de la bascule d’un panneau. Dans cet exemple, aucun composant latéral ni composant latéral ne comporte de paramètres d’entrée, car ils communiquent via le service.





**Routing(Parent Root, Child Route)**

Le routeur angularest ce qui fait d'une application angulaire une application à page unique ou SPA.

La première chose à faire est simplement d'écrire une configuration de routage.Ce que nous faisons dans cette configuration initiale est de mapper certains chemins d'URL vers des composants angulaires: ce qui signifie que s'il y a une correspondance de chemin, le composant est affiché

**Services et injection de dépendances**

L'injection de dépendance est une notion importante à bien comprendre et à bien implémenter. Déjà présent dans AngularJS, cette notion est toujours présente dans le nouveau frameworkde Google mais à bien changé. Nous allons prendre l'exemple d'un service de Log pour illustrer nos propos. Un service est une class avec un décorateur spécifique @Injectable().

Le décorateur @Injectable()

@Injectable() est un décorateur un peu particulier. Il ne permet pas l’injection à proprement parlé, mais plutôt d’initialiser un contexte détectabilité. Si vous injectez dans un de vos services ( sans ce décorateur) un autre service, le moteur d'injection retournera une erreur. Angularconseille de toujours mettre cette annotation sur un service même si vous n'utilisez pas les injections dans les premiers développements de votre service afin d'éviter de se poser la question plus tard.

**Pipes**

Les Pipes sont des filtres utilisables directement depuis la vue afin de transformer les valeurs lors du "binding".

Utilisation des Pipes

Syntaxe

La syntaxe des Pipes est simplement inspirée des Pipes des shellUNIX que l'on retrouve dans de nombreux systèmes de templating.



Paramètres

Les Pipes peuvent prendre des paramètres qu'il faut mettre après le Pipe et séparés avec le symbole ":".



Chaînage

Les "pipes" peuvent être chaînés.



Les Pipes natifs

Angular dispose de plusieurs "pipes" natifs : https://angular.io/api?type=pipe.

Le Pipe async

Le Pipe asyncest un Pipe capable de consommer des Observables (ou Promise) en appelant implicitement la méthode subscribe (ou then) afin de "binder" les valeurs contenus dans l'Observable (ou la Promise).Dans les cas des Observables, ce Pipe unsubscribe automatiquement à la destruction de la vue.

Cf. Gestion de la Subscription.

Pipes personnalisés

Pour créer un Pipe personnalisé, il faut :

1.implémenter une classe suivant l'interface PipeTransform,

2.décorer cette classe avec le décorateur @Pipe() en indiquant le nom du Pipe.

3.ajouter la classe aux declarations(et exports) du module associé.