|  |  |
| --- | --- |
| TypeScript :  - Généralités | * TypeScript, qui existe depuis 2012, est un sur-ensemble de JavaScript, ajoutant quelques trucs a ES5. Le plus important étant le système de type, lui donnant même son nom. * Depuis la version 1.5, sortie en 2015, cette bibliothèque essaie d’être un sur-ensemble d’ES6, incluant toutes les fonctionnalités vues précédemment, et quelques nouveautés, comme les décorateurs.      * Ecrire du TypeScript ressemble à écrire du JavaScript. Par convention les fichiers sources TypeScript ont l’extension .ts, et seront compiles en JavaScript standard, en général lors du build, avec le compilateur TypeScript. Le code généré reste très lisible. |
| TypeScript :   * Les types * Les paramètres | * Les types - syntaxe : let variable : type. On peut aussi avoir : let variable : type1 | type2 * Les différents types : string, number, any, Array, une classe définie dans l’application, * On peut également définir des enums, le type de retour des fonctions, des interfaces,… * Les paramètres optionnels d’une fonction se définissent avec un ? après le paramètre. |
| TypeScript :   * Les Classes | * Les classes peuvent implémenter des interfaces, * Une interface peut étendre d’autres interfaces * Une classe en TypeScript peut avoir des propriétés et des méthodes, ce n’est pas une fonctionnalité standard de ES6 * Tout est public par défaut et on peut utiliser le mot clé private pour cacher une propriété ou une méthode. * Ajouter public ou private a un parametre de constructeur est un raccourci pour créer et initialiser un membre prive ou public |
| TypeScript :   * Les décorateurs | * Un décorateur est une façon de faire de la meta-programmation. Ils ressemblent beaucoup aux annotations, qui sont principalement utilisées en Java, C#, et Python, et peut-être d’autres langages que je ne connais pas. Suivant le langage, tu peux ajouter une annotation sur une méthode, un attribut, ou une classe. Généralement, les annotations ne sont pas vraiment utiles au langage lui-même, mais plutôt aux frameworks et aux bibliothèques. * Leur but sont multiples : ils peuvent modifier leur cible (classes, methodes, etc.) et par exemple modifier les paramètres ou le résultat retourne, appeler d’autres méthodes quand la cible est appelée, ou ajouter des métadonnées destinées à un Framework. * En TypeScript, les annotations sont préfixées par @, et peuvent être appliquées sur une classe, une propriété de classe, une fonction, ou un paramètre de fonction. Pas sur un constructeur en revanche, mais sur ses paramètres oui. |
| Les web components | * Les Web Components essaient de résoudre le problème suivant : avoir des composants réutilisables et encapsules, et qui marcherait directement et apporterait la fonctionnalité aux utilisateurs sans rien faire.   Ce standard émergent est défini dans quatre spécifications :   * Custom elements ("élements personnalises") : un standard qui permet au développeur de créer ses propres éléments du DOM (par exemple <mon-toto></mon-toto> * Shadow DOM ("DOM de l’ombre") : Le Shadow DOM est une façon d’encapsuler le DOM de ton composant. Cette encapsulation signifie que la feuille de style et la logique JavaScript de ton application ne vont pas s’appliquer sur le composant et le ruiner accidentellement. Cela en fait l’outil idéal pour dissimuler le fonctionnement interne de ton composant, et s’assurer que rien n’en fuit à l’extérieur. * Template : Un template spécifie dans un élément <template> n’est pas affiche par le navigateur. Son but est d’être à terme clone dans un autre élément. * HTML imports : pour permettre d’importer du HTML dans du HTML. C’est ce qui va permettre à un utilisateur d’importer le composant créé. |
| Les enjeux autour d’Angular | * Angular est un framework géré par Google. * Il s’appuie sur le TypeScript un sur-ensemble de javascript. Ses particularités : * Il doit être transcompilé pour être rendu compatible sur les navigateurs. * Il est beaucoup plus stable, rapide et facile * Il possède un typage strict * On peut utiliser des fonctions lambda ou arrow * On peut créer des classes et des interfaces, codage plus modulaire et robuste * Ionic le framework dédié aux applications mobiles utilise Angular. |
| Les composants | * Angular est un Framework orienté composant. * Un composant est un groupe d’éléments dans un template dédiés à une tâche particulière. Assemblés une à une et organisés de façon hiérarchique, ces petits composants vont former une application complète. * Beaucoup d’autres framwork s’appuient sur cette philosophie : React.js, Amber.js, Vue.js, Aurelia... * Un composant est une partie isolée de mon application, de même mon application est un composant comme les autres. * Les composants peuvent être regroupés ensembles en entités cohérentes, appelées modules. |
| Préparer l’environnement de développement | * Installer Node.js la dernière version stable. * Installer la dernière version de NPM (Node Package Manager) avec la commande : npm install –g npm@latest * Installer le CLI Angular : npm install –g @angular/cli |
| Créer un projet Angular | * ng new nom\_projet --style=scss * ng fix audit pour régler les éventuels problèmes dans les composants téléchargés. * ng serve --open pour lancer l’application sur le navigateur. |
| Ajouter bootstrap au projet | * npm install bootstrap@3.3.7 –save * Ajouter le chemin suivant au fichier angular.json, section ‘style’ : nodes\_modules/bootstrap/dist/css/bootstrap.css |
| Les templates | * Un template permet de rendre du contenu html avec des parties dynamiques dépendant de mes données. * L’opérateur ? (Safe navigation operator) permet de s’assurer que l’application va s’exécuter même si une variable est indéfinie.- |
| Databinding | Le databinding, c'est la communication entre votre code TypeScript et le template HTML qui est montré à l'utilisateur.  Cette communication est divisée en deux directions :   * Les informations venant du code devant être affichées dans le navigateur. Les 2 principales méthodes pour cela sont le string interpolation (afficher le contenu d’une variable) et la propery binding (modifier dynamiquement les propriétés d'un élément du DOM en fonction de données dans votre TypeScript). Exemple de syntaxe : <p>{{ nom\_variable }}</p> est équivalent à <p [textContent]= ’’nom\_variable’’></p> * Event binding : faire réagir mon code TypeScript aux évènements venant du template HTML. * Two-way binding : pratique si l’on mettre en place une communication à double sens, par exemple sur les formulaires. * Les propriétés du DOM ont un avantage sur les attributs HTML : leurs valeurs sont forcément à jour. Dans mon exemple, l'*attribut* value contiendra toujours "hello", alors que la *propriété* value du DOM sera modifiée dynamiquement par le navigateur, et contiendra ainsi la valeur entrée par l’utilisateur dans le champ de saisie. |
| Générer un nouveau composant | ng generate component nom\_composant ou bien ng g c nom\_composant |
| Lancer l’application | ng serve --open |
| Les Directives | Les directives sont des instructions intégrées dans le DOM que vous utiliserez presque systématiquement quand vous créerez des applications Angular. Il existe deux types principaux de directive : les directives structurelles et les directives par attribut. |
| Les directives structurelles | Ce sont des directives qui modifient la structure du document. Exemple :   * \*ngFor : structure itérative * \*ngIf : structure conditionnelle * \*ngSwitch : permet de switcher entre plusieurs templates. |
| Les directives de template | Ce sont des directives qui permettent d’ahgir sur le style d’un éléement du DOM   * ngStyle : pour modifier plusieurs styles en même temps * ngClass : pour ajouter, modifier, enlever dynamiquement des classes d’un élément |
| Les directives par attribut | Modifient le comportement d’un objet déjà existant. Exemple :   * ngModel : ajouter ou modifier un attribut d’un objet déjà existant * ngStyle : pour appliquer des styles à un objet du DOM de manière dynamique * ngClass : Ajouter une classe CSS à un objet déjà existant |
| Les Pipes | Les pipes prennent des données en input, les transforment, et puis affichent les données modifiées dans le DOM.  Il y a des pipes fournis avec Angular, et vous pouvez également créer vos propres pipes si vous en avez besoin. L’idée étant de modifier l’affichage des données, sans en modifier la nature (cf : twig).  Pipe particulier : async permet de spécifier à Angular des données qui sont récupérées de façon asynchrone. Un *pipe* async retourne une chaine de caractères vide jusqu’a ce que les données deviennent disponibles (i.e. jusqu’a ce que la promise soit résolue, dans le cas d’une promise). Une fois résolue, la valeur obtenue est retournée. Et plus important, cela déclenche un cycle de détection de changement une fois la donnée obtenue.  Voici quelques pipes :   * json : permet d’afficher le contenu d’une variable au format json. Il peut être utilisé côté serveur avec le service JsonPipe * slice : pour afficher un sous-ensemble d’une collection. Il peut être appliqué à un tableau ou à une chaine de caractères. * uppercase : transforme une chaine de caractères en majuscules * lowercase : transforme une chaine de caractères en minuscules * titlecase : transforme la première lettre de chaque mot en majuscule * number : permet de formater un nombre. Il peut prendre les paramètres suivants : integerDigits, minFractiondigits, maxFractionDigits. * percent : permet d’afficher un pourcentage * currency : permet d’afficher le symbole d’une devise monétaire. Il faut juste fournir en paramètre le code ISO de la monnaie, avec éventuellement l’option symbol ou symbol-narrow pour afficher le symbole de la monnaie, * date : permet de formater une date   On peut également créer ses propres pipes. Pour cela, on crée une classe :   * à laquelle on applique le décorateur @pipe * qui implémente l’interface PipeTransform. On devra par là même définir la méthode transform qui fera tout le travail de formattage. |
| Les injections de dépendances | * L’injection de dépendances est un *design pattern* bien connu. Prenons un composant de notre application. Il peut avoir besoin de faire appel à des fonctionnalités qui sont définies dans d’autres parties de l’application (un service, par exemple). C’est ce que l’on appelle une dépendance : le composant dépend du service. Au lieu de laisser au composant la charge de créer une instance du service, l’idée est que le Framework crée l’instance du service lui-même, et la fournisse au composant qui en a besoin. Cette façon de procéder se nomme l’inversion de contrôle. * Cela apporte plusieurs bénéfices : * le développement est simplifie, on exprime juste ce que l’on veut, ou on le veut. * le test est simplifie, en permettant de remplacer les dépendances par des versions bouchonnees. * la configuration est simplifiée, en permutant facilement différentes implémentations. * Pour faire de l’injection de dépendances, on a besoin : * d’une façon d’enregistrer une dépendance, pour la rendre disponible à l’injection dans d’autres composants/services. * d’une façon de déclarer quelles dépendances sont requises dans chaque composant/service.   Le Framework se chargera ensuite du reste. Quand on déclarera une dépendance dans un composant, il regardera dans le registre s’il la trouve, récupèrera l’instance existante ou en créera une, et réalisera enfin l’injection dans notre composant.   * Une dépendance peut être aussi bien un service fourni par Angular, ou un des services que nous avons écrits. |
| Les Services | Un service permet de centraliser des parties de votre code et des données qui sont utilisées par plusieurs parties de votre application ou de manière globale par l'application entière. Pour être utilisé dans l'application, un service doit être injecté, et le niveau choisi pour l'injection est très important.  Il y a trois niveaux possibles pour cette injection :   * dans  AppModule : ainsi, la même instance du service sera utilisée par tous les components de l'application *et* par les autres services ; * dans  AppComponent : comme ci-dessus, tous les components auront accès à la même instance du service mais non les autres services ; * dans un autre component : le component lui-même et tous ses enfants (c'est-à-dire tous les components qu'il englobe) auront accès à la même instance du service, mais le reste de l'application n'y aura pas accès.   Le Framework propose deux service par défaut : Service Title pour modifier le titre de la page, Service Meta pour modifier les balises méta de la page. |
| La méthode ngOnInit() | La méthode  ngOnInit()  d'un component est exécutée une fois par instance au moment de la création du component par Angular, et après son constructeur. On l'utilise très souvent pour initialiser des données une fois le component créé. |
| Autres méthodes | * ngOnChanges sera la 1ere appelée lorsque la valeur d’une propriété bindée est modifiée. * ngOnDestroy sera appelé lorsque le composant sera supprimé. |
| Le Routing | L'un des énormes avantages d'utiliser Angular est de pouvoir créer des "single page application" (SPA). Sur le Web, ces applications sont rapides et lisses : il n'y a qu'un seul chargement de page au début, et même si les données mettent parfois du temps à arriver, la sensation pour l'utilisateur est celle d'une application native.  Au lieu de charger une nouvelle page à chaque clic ou à chaque changement d'URL, on remplace le contenu ou une partie du contenu de la page : on modifie les components qui y sont affichés, ou le contenu de ces components.  On accomplit tout cela avec le "routing", où l'application lit le contenu de l'URL pour afficher le ou les components requis.  Sous Angular, une route est un ensemble d’instructions d'affichage à suivre pour chaque URL, c'est-à-dire quel(s) component(s) il faut afficher à quel(s) endroit(s) pour un URL donné. Puisque le routing d'une application est fondamentale pour son fonctionnement, on déclare les routes dans  app.module.ts.  Lorsqu’un routerLink  se trouve à l'intérieur du  router-outlet , il faut ajouter un  /  au début de l'URL pour naviguer vers la racine, sinon il prend en compte url de la page courante. |
| Les Guards | Une guard est en effet un service qu'Angular exécutera au moment où l'utilisateur essaye de naviguer vers la route sélectionnée.  Ce service implémente l'interface  canActivate , et donc doit contenir une méthode du même nom qui prend les arguments ActivatedRouteSnapshot et RouterStateSnapshot (qui lui seront fournis par Angular au moment de l'exécution) et retourne une valeur booléenne, soit de manière synchrone (boolean), soit de manière asynchrone (sous forme de Promise ou d'Observable).  Il existe 4 types de Guards :   * canActivate : pour autoriser ou empêcher l’activation d’une route * canActivateChild : pour autoriser ou empêcher l’activation des routes enfants. * CanLoad : pour permettre de télécharger un module applicatif à la demande (lazy loading), contenant des routes filles * CanDesactivate : pour empêcher de quitter la route actuelle, par exemple demander une confirmation avant de quitter une page contenant un long formulaire. |
| Programmation Réactive | C’est une façon de construire une application avec des évènements, et d’y réagir. Les évènements peuvent être combines, filtres, groupes, etc. en utilisant des fonctions comme map, filter, etc.  Dans la programmation réactive, toute donnée entrante sera dans un flux. Ces flux peuvent être écoutes, évidemment modifiés (filtres, fusionnes, …), et même devenir un nouveau flux que l’on pourra aussi écouter. Cette technique permet d’obtenir des programmes faiblement couples : on n’a plus à se soucier des conséquences d’un appel de méthode, on se contente tu te contentes de déclencher un évènement, et toutes les parties de l’application intéressées réagiront en conséquence.  Angular est construit sur de la programmation réactive, et nous utiliserons aussi cette technique pour certaines parties. Répondre a une requête HTTP ? Programmation réactive. Lever un évènement spécifique dans un de nos composants ? Programmation réactive. Gérer un changement de valeurs dans un de nos formulaires ? Programmation réactive.  Dans la programmation réactive, tout est un flux. Un flux est une séquence ordonnée d’évènements. Ces évènements représentent des valeurs (hé, regarde, une nouvelle valeur !), des erreurs (oups, ca a planté), ou des terminaisons (voila, j’ai fini). Tous ces évènements sont poussés par un producteur de données, vers un consommateur. Le travail consiste à sera de s’abonner (*subscribe*) à ces flux, i.e. définir un listener capable de gérer ces trois possibilités. Un tel listener sera appelé un *observer*, et le flux, un *observable*. Ces termes ont été définis il y longtemps, car ils constituent un *design pattern* bien connu : l'*observer*.  La bibliothèque la plus populaire de programmation réactive dans l’écosystème JavaScript est RxJS. |
| Observer les données Les observables  Les Subjects  RxJs | Pour réagir à des événements ou à des données de manière asynchrone (c'est-à-dire ne pas devoir attendre qu'une tâche, par exemple un appel HTTP, soit terminée avant de passer à la ligne de code suivante), il y a eu plusieurs méthodes depuis quelques années.  Il y a le système de callback, par exemple, ou encore les Promise.  Avec l'API RxJS, fourni et très intégré dans Angular, la méthode proposée est celle des Observables. |
| Un Observable est un objet qui émet des informations auxquelles on souhaite réagir.  Ces informations peuvent venir d'un champ de texte dans lequel l'utilisateur rentre des données, ou de la progression d'un chargement de fichier, par exemple.  Elles peuvent également venir de la communication avec un serveur : le client HTTP.  Les observables sont mis à disposition par RxJs, un package tiers qui est fourni avec Angular.  À cet Observable, on associe un Observer — un bloc de code qui sera exécuté à chaque fois que l'Observable émet une information.  L'Observable émet trois types d'information : des données, une erreur, ou un message complete.  Du coup, tout Observer peut avoir trois fonctions : une pour réagir à chaque type d'information.  Maintenant que vous avez un Observable, il faut l'observer !  On utilisera la fonction subscribe(), qui prendra comme arguments entre une et trois fonctions anonymes pour gérer les trois types d'informations que cet Observable peut envoyer :   * la première se déclenche à chaque fois que l'Observable émet de nouvelles données, et reçoit ces données comme argument ; * la deuxième se déclenche si l'Observable émet une erreur, et reçoit cette erreur comme argument ; * la troisième se déclenche si l'Observable s'achève, et ne reçoit pas d'argument.   Il existe un type d'Observable qui permet non seulement de réagir à de nouvelles informations, mais également d'en émettre.  Imaginez une variable dans un service, par exemple, qui peut être modifié depuis plusieurs components ET qui fera réagir tous les components qui y sont liés en même temps.  Voici l'intérêt des *Subjects*.  L'API RxJS propose énormément de possibilités — beaucoup trop pour tout voir dans ce cours.  Cependant, j'aimerais vous parler rapidement de l'existence des opérateurs.  Un opérateur est une fonction qui se place entre l'Observable et l'Observer (la Subscription, par exemple), et qui peut filtrer et/ou modifier les données reçues avant même qu'elles n'arrivent à la Subscription.  Voici quelques exemples rapides :   * take(n) : va piocher les premiers éléments * map()  : modifie les valeurs reçues — peut effectuer des calculs sur des chiffres, transformer du texte, créer des objets… * merge : fusionnera deux flux * filter()  : comme son nom l'indique, filtre les valeurs reçues selon la fonction qu'on lui passe en argument. * throttleTime()  : impose un délai minimum entre deux valeurs — par exemple, si un Observable émet cinq valeurs par seconde, mais ce sont uniquement les valeurs reçues toutes les secondes qui vous intéressent, vous pouvez passer  throttleTime(1000)  comme opérateur. * scan()  et  reduce()  : permettent d'exécuter une fonction qui réunit l'ensemble des valeurs reçues selon une fonction que vous lui passez — par exemple, vous pouvez faire la somme de toutes les valeurs reçues. La différence basique entre les deux opérateurs :  reduce()  vous retourne uniquement la valeur finale, alors que  scan()  retourne chaque étape du calcul. * Subscribe appliquera la fonction à chaque évènement qu’il reçoit.   En savoir plus sur RxJs et les Observables : <http://reactivex.io/> |
| Les formulaires | En Angular, il y a deux grandes méthodes pour créer des formulaires :   * la méthode **template** : vous créez votre formulaire dans le template, et Angular l'analyse pour comprendre les différents inputs et pour en mettre à disposition le contenu ; Le formulaire utilisé est de type ngForm * la méthode **réactive** : vous créez votre formulaire en TypeScript et dans le template, puis vous en faites la liaison manuellement — cette approche est plus complexe, mais elle permet beaucoup plus de contrôle et une approche dynamique. Le formulaire utilisé est de type FormGroup.   Dans Angular, par défaut le comportement de validation HTML5 est desactivé. |
| Interagir avec un serveur avec HttpClient | Angular met à disposition un service appelé  HttpClient  qui permet de créer et d'exécuter des appels HTTP (fait par AJAX - Asynchronous JavaScript and XML) et de réagir aux informations retournées par le serveur. Il existe plusieurs méthodes correspondant aux verbes http communs : get, post, put, delete, patch , head, jsonp.  L’option params represente les parametres de recherche (que l’on appelle aussi *query string*) à ajouter à l’URL.  L’option headers est pratique pour ajouter quelques headers custom à la requête. Cela est notamment nécessaire pour certaines techniques d’authentification comme par exemple Json Web Token.  Pour te permettre d’utiliser leur API sans être bloque par la *Same Origin Policy* ("politique de même origine") assurée par les navigateurs web, certains services web n’utilisent pas CORS, mais JSONP (*JSON with Padding*). Le serveur ne retournera pas le JSON directement, mais l’emballera dans une fonction passée en callback. La réponse arrive alors sous forme de script, et les scripts ne sont pas soumis à la *Same Origin Policy*. Une fois charge, tu peux alors accéder à la valeur JSON contenue dans la réponse.  Les intercepteurs permettent d’intercepter des appels de requêtes ou des réponses pour ajouter un header particulier à certaines d’entre elles ou bien gérer des types de réponse. |
| Le service Firebase de Google | Ce service permet la création d'un backend complet sans coder, et comprend énormément de services, dont l'authentification, une base de données NoSQL et un stockage de fichiers. |
| Les tests | * Sous Angular, les fichiers de tests portent l’extension spec.ts. On peut écrire 2 types de tests : * - Les tests unitaires : permettent de vérifier qu’un bout de code fonctionne parfaitement en isolation. : L’un des concepts au cœur d’un test unitaire est celui d’isolation : on ne veut pas que notre test soit biaise par ses dépendances. Alors on utilise généralement des objets bouchonnes (*mock*) comme dépendances. Ce sont des objets factices crées juste pour les besoins du test. Jasmine & Karma : une bibliothèque pour faire des tests unitaires. |
|  |  |
|  |  |
|  |  |