|  |  |
| --- | --- |
| 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. |
| 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 * 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 |
| 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). * 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. |
| 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 |
| 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. |
| 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. |
| 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. |
| 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. |
| 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) |
| 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 :   * map()  : modifie les valeurs reçues — peut effectuer des calculs sur des chiffres, transformer du texte, créer des objets… * 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.   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 ; * 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.   Dans Angular, par défaut le comportement de validation HTML5 est desactivé. |
| git push |  |
| git revert 6448b12ab54… puis faire git push | Crée un commit inverse à un commit existant. Astuce pour annuler un commit envoyé sur le serveur |
| git branch | Affiche toutes les branches de mon dépôt |
| git branch nom\_branche | Crée une branche sur mon dépôt en local |
| git checkout nom\_branche | Se positionne sur la branche spécifiée et restaure les fichiers de la branche tels qu’ils étaient lors du dernier commit. |
| git checkout master | Se positionne sur la branche principale |
| Fusionner une branche avec la branche principale | git checkout master (se positioner sur la branche principale)  git pull (récupérer les dernières modifications de la branche principale sur le serveur)  git merge nom\_branche (fusionne la branche selectionnée avec la branche principale)  git push (pousse la branche principale sur le serveur) |
| git branch –d nom\_branche git branch –D nom\_branch | Supprime une branche. S’il n’y a pas eu de merge auparavant, git interdit l’opération  Supprime une branche même s’il n’y a pas eu de merge auparavant. |
| git stash | Enregistre et met de côté mes modifications, si je ne veux pas encore commiter mes modifs et changer de branche entre temps. |
| git stash apply | Pour récupérer mes fichiers mis de coté |
| git branch -r | Lister les branches que le serveur connait |
| git branch --track br\_local br\_serveur | Crée une copie en locale d’une branche sur le serveur et la garde connectée à celle-ci |
| git tag nom\_version id\_commit git push --tags  git tag –d nom\_tag | Ajoute un tag ou alias (par exemple v1.2) sur un commit précis  Envoie le tag également lors du push, ce qui n’est pas le cas par défaut.  Supprimer un tag |
| git grep « motif » | Recherche une occurrence dans les fichiers source, par exemple git grep « TODO » |
| Demander à git d’ignorer des fichiers | Créer un fichier **.gitignore** à la racine et y indiquer les noms des fichiers ligne par ligne. On peut utiliser \* comme joker afin de spécifier des ensemble de fichier dans un répertoire, ou qui ont la même extension, etc. Aucun de ces fichiers n’apparaitra dans git status, même modifié. |
|  |  |
|  |  |
|  |  |
|  |  |