

Become a  
**NINJA**  
with



ANGULAR

*ninja*  *squad*

Deviens un ninja avec Angular

Ninja Squad

# Table des matières

1. Introduction .....	1
2. Une rapide introduction à ECMASCIRIPT 6.....	4
2.1. Transpileur .....	4
2.2. <code>let</code> .....	5
2.3. Constantes .....	6
2.4. Création d'objets.....	7
2.5. Affectations déstructurées.....	7
2.6. Paramètres optionnels et valeurs par défaut .....	9
2.7. <i>Rest operator</i> .....	11
2.8. Classes.....	12
2.9. <i>Promises</i> .....	14
2.10. <i>(arrow functions)</i> .....	17
2.11. <i>Set et Map</i> .....	21
2.12. Template de string.....	22
2.13. Modules .....	22
2.14. Conclusion .....	24
3. Un peu plus loin qu'ES6 .....	25
3.1. Types dynamiques, statiques et optionnels.....	25
3.2. Hello TypeScript .....	26
3.3. Un exemple concret d'injection de dépendance.....	26
4. Découvrir TypeScript .....	29
4.1. Les types de TypeScript .....	29
4.2. Valeurs énumérées ( <i>enum</i> ) .....	30
4.3. Return types .....	30
4.4. Interfaces .....	31
4.5. Paramètre optionnel .....	31
4.6. Des fonctions en propriété.....	32
4.7. Classes.....	32
4.8. Utiliser d'autres bibliothèques .....	34
4.9. Decorateurs .....	35
5. Le monde merveilleux des Web Components .....	38
5.1. Le nouveau Monde .....	38
5.2. Custom elements .....	38
5.3. Shadow DOM.....	39
5.4. Template.....	40
5.5. HTML imports.....	41
5.6. Polymer et X-tag .....	41
6. La philosophie d'Angular.....	44

7. Commencer de zéro .....	48
7.1. Créer une application avec TypeScript.....	48
7.2. Notre premier composant .....	50
7.3. Notre premier module Angular .....	52
7.4. Démarrer l'application .....	54
7.5. Commencer de zéro avec Angular CLI .....	57
8. La syntaxe des templates .....	59
8.1. Interpolation .....	59
8.2. Utiliser d'autres composants dans nos templates .....	63
8.3. <i>Binding</i> de propriété .....	64
8.4. Événements .....	68
8.5. Expressions vs instructions .....	71
8.6. Variables locales .....	72
8.7. Directives de structure .....	72
8.8. Autres directives de templating .....	77
8.9. Syntaxe canonique.....	77
8.10. Résumé .....	78
9. Injection de dépendances .....	82
9.1. Cuisine et Dépendances .....	82
9.2. Développement facile .....	82
9.3. Configuration facile .....	86
9.4. Autres types de provider .....	90
9.5. Injecteurs hiérarchiques .....	91
9.6. Injection sans types .....	94
10. Services .....	95
10.1. Service Title.....	95
10.2. Service Meta .....	95
10.3. Créer son propre service .....	96
11. <i>Pipes</i> .....	98
11.1. Ceci n'est pas une pipe .....	98
11.2. json .....	98
11.3. slice .....	99
11.4. uppercase ("majuscule") .....	101
11.5. lowercase ("minuscule") .....	102
11.6. titlecase ("titre") .....	102
11.7. number ("nombre").....	102
11.8. percent ("pourcent") .....	103
11.9. currency ("devise monétaire") .....	103
11.10. date .....	104
11.11. async .....	104
11.12. Créer tes propres <i>pipes</i> .....	105

12. Programmation réactive .....	108
12.1. OK, on vous rappellera .....	108
12.2. Principes généraux .....	108
12.3. RxJS .....	109
12.4. Programmation réactive en Angular .....	111
13. Créer des composants et directives .....	113
13.1. Introduction .....	113
13.2. Directives .....	113
13.3. Composants .....	125
14. Style des composants et encapsulation .....	128
14.1. Stratégie <i>Native</i> .....	129
14.2. Stratégie <i>Emulated</i> ("émulée") .....	129
14.3. Stratégie <i>None</i> ("aucune") .....	130
14.4. Style l'hôte .....	130
15. Tester ton application .....	131
15.1. Tester c'est douter .....	131
15.2. Test unitaire .....	131
15.3. Dépendances factices .....	137
15.4. Tester des composants .....	139
15.5. Tester avec des templates ou providers factices .....	142
15.6. Tests end-to-end ("de bout en bout") .....	144
16. Envoyer et recevoir des données par HTTP .....	146
16.1. Obtenir des données .....	146
16.2. Transformer des données .....	148
16.3. Options avancées .....	149
16.4. Jsonp .....	150
16.5. Tests .....	151
17. Routeur .....	154
17.1. En route .....	154
17.2. Navigation .....	156
18. Formulaires .....	160
18.1. Form, form, form-idable ! .....	160
18.2. Formulaire piloté par le template .....	163
18.3. Formulaire piloté par le code .....	167
18.4. Un peu de validation .....	171
18.5. Erreurs et soumission .....	173
18.6. Un peu de style .....	176
18.7. Créer un validateur spécifique .....	178
18.8. Combiner les approches pilotée par le code et pilotée par le template pour la validation ..	182
18.9. Regrouper des champs .....	182
18.10. Réagir aux modifications .....	184

18.11. Conclusion .....	186
19. Les Zones et la magie d'Angular .....	188
19.1. AngularJS 1.x et le <i>digest cycle</i> .....	188
19.2. Angular et les zones .....	191
20. Observables : utilisation avancée .....	197
20.1. subscribe, unsubscribe et <i>pipe async</i> .....	197
20.2. Le pouvoir des opérateurs .....	202
20.3. Construire son propre Observable .....	206
21. Internationalisation .....	209
21.1. La <i>locale</i> .....	209
21.2. Traduire du texte .....	210
21.3. Processus et outillage .....	211
21.4. Traduire les messages dans le code .....	216
21.5. Pluralisation .....	216
21.6. Bonnes pratiques .....	218
22. Ce n'est qu'un au revoir .....	220
Annexe A: Historique des versions .....	223
A.1. v1.6 - 2017-03-24 .....	223
A.2. v1.5 - 2017-01-25 .....	224
A.3. v1.4 - 2016-11-18 .....	224
A.4. v1.3 - 2016-09-15 .....	225
A.5. v1.2 - 2016-08-25 .....	225
A.6. v1.1 - 2016-05-11 .....	227

# Chapitre 1. Introduction

Alors comme ça on veut devenir un ninja ?! Ça tombe bien, tu es entre de bonnes mains !

Mais pour y parvenir, nous avons un bon bout de chemin à parcourir ensemble, semé d'embûches et de connaissances à acquérir :).

On vit une époque excitante pour le développement web. Il y a un nouvel Angular, une réécriture complète de ce bon vieil AngularJS. Pourquoi une réécriture complète ? AngularJS 1.x ne suffisait-il donc pas ?

J'adore cet ancien AngularJS. Dans notre petite entreprise, on l'a utilisé pour construire plusieurs projets, on a contribué du code au cœur du framework, on a formé des centaines de développeurs (oui, des centaines, littéralement), et on a même écrit [un livre](#) sur le sujet.

AngularJS est incroyablement productif une fois maîtrisé. Mais cela ne nous empêche pas de constater ses faiblesses. AngularJS n'est pas parfait, avec des concepts très difficiles à cerner, et des pièges ardus à éviter.

Et qui plus est, le web a bien évolué depuis qu'AngularJS a été conçu. JavaScript a changé. De nouveaux frameworks sont apparus, avec de belles idées, ou de meilleures implémentations. Nous ne sommes pas le genre de développeurs à te conjurer d'utiliser tel outil plutôt que tel autre. Nous connaissons juste très bien quelques outils, et savons ce qui peut correspondre au projet. AngularJS était un de ces outils, qui nous permettait de construire des applications web bien testées, et de les construire vite. On a aussi essayé de le plier quand il n'était pas forcément l'outil idéal. Merci de ne pas nous condamner, ça arrive aux meilleurs d'entre nous, n'est-ce pas ? ;p

Angular sera-t-il l'outil que l'on utilisera sans aucune hésitation dans nos projets futurs ? C'est difficile à dire pour le moment, parce que ce framework est encore tout jeune, et que son écosystème est à peine bourgeonnant.

En tout cas, Angular a beaucoup de points positifs, et une vision dont peu de frameworks peuvent se targuer. Il a été conçu pour le web de demain, avec ECMAScript 6, les Web Components, et le mobile en tête. Quand il a été annoncé, j'ai d'abord été triste, comme beaucoup de gens, en réalisant que cette version 2.0 n'allait pas être une simple évolution (et désolé si tu viens de l'apprendre).

Mais j'étais aussi très curieux de voir quelles idées allait apporter la talentueuse équipe de Google.

Alors j'ai commencé à écrire ce livre, dès les premiers commits, lisant les documents de conception, regardant les vidéos de conférences, et analysant chaque commit depuis le début. J'avais écrit mon premier livre sur AngularJS 1.x quand c'était déjà un animal connu et bien apprivoisé. Ce livre-ci est très différent, commencé quand rien n'était encore clair dans la tête même des concepteurs. Parce que je savais que j'allais apprendre beaucoup, sur Angular évidemment, mais aussi sur les concepts qui allaient définir le futur du développement web, et certains n'ont rien à voir avec Angular. Et ce fut le cas. J'ai du creuser pas mal, et j'espère que tu vas apprécier revivre ces découvertes avec moi, et comprendre comment ces concepts s'articulent avec Angular.

L'ambition de cet ebook est d'évoluer avec Angular. S'il s'avère qu'Angular devient le grand framework qu'on espère, tu en recevras des mises à jour avec des bonnes pratiques et de nouvelles

fonctionnalités quand elles émergeront (et avec moins de fautes de frappe, parce qu'il en reste probablement malgré nos nombreuses relectures...). Et j'adorerais avoir tes retours, si certains chapitres ne sont pas assez clairs, si tu as repéré une erreur, ou si tu as une meilleure solution pour certains points.

Je suis cependant assez confiant sur nos exemples de code, parce qu'ils sont extraits d'un vrai projet, et sont couverts par des centaines de tests unitaires. C'était la seule façon d'écrire un livre sur un framework en gestation, et de repérer les problèmes qui arrivaient inévitablement avec chaque release.

Même si au final tu n'es pas convaincu par Angular, je suis à peu près sûr que tu vas apprendre deux-trois trucs en chemin.

Si tu as acheté le "pack pro" (merci !), tu pourras construire une petite application morceau par morceau, tout au long du livre. Cette application s'appelle **PonyRacer**, c'est un site web où tu peux parler sur des courses de poneys. Tu peux même [tester cette application ici](#) ! Vas-y, je t'attends.

Cool, non ?

Mais en plus d'être super cool, c'est une application complète. Tu devras écrire des composants, des formulaires, des tests, tu devras utiliser le routeur, appeler une API HTTP (fournie), et même faire des Web Sockets. Elle intègre tous les morceaux dont tu auras besoin pour construire une vraie application.

Chaque exercice viendra avec son squelette, un ensemble d'instructions et quelques tests. Quand tu auras tous les tests en succès, tu auras terminé l'exercice !

Les 6 premiers exercices du Pack Pro sont gratuits. Les autres ne sont accessibles que pour les acheteurs de notre formation en ligne. À la fin de chaque chapitre, nous listerons les exercices du Pack Pro liés aux fonctionnalités expliquées dans le chapitre, en signalant les exercices gratuits avec le symbole suivant :  , et les autres avec le symbole suivant : .

Si tu n'as pas acheté le "pack pro" (tu devrais), ne t'inquiète pas : tu apprendras tout ce dont tu auras besoin. Mais tu ne construiras pas cette application incroyable avec de beaux poneys en pixel art. Quel dommage :) !

Tu te rendras vite compte qu'au-delà d'Angular, nous avons essayé d'expliquer les concepts au cœur du framework. Les premiers chapitres ne parlent même pas d'Angular : ce sont ceux que j'appelle les "chapitres conceptuels", ils te permettront de monter en puissance avec les nouveautés intéressantes de notre domaine.

Ensuite, nous construirons progressivement notre connaissance du framework, avec les composants, les templates, les pipes, les formulaires, http, le routeur, les tests...

Et enfin, nous nous attaquerons à quelques sujets avancés. Mais c'est une autre histoire.

Passons cette trop longue introduction, et jetons-nous sur un sujet qui va définitivement changer notre façon de coder : ECMAScript 6.

**NOTE** Cet ebook utilise Angular version 4.0.0 dans les exemples.

## NOTE

### *Angular et versions*

Le titre de ce livre était à l'origine "Deviens un Ninja avec Angular 2". Car à l'origine, Google appelait ce framework Angular 2. Depuis, ils ont revu leur [politique de versioning](#).

D'après leur plan, on aura une version majeure tous les 6 mois. Et désormais, le framework est simplement nommé "Angular".

Pas d'inquiétudes, ces versions majeures ne seront pas des réécritures complètes sans compatibilité ascendante, comme Angular 2 l'a été pour AngularJS 1.x.

Comme cet ebook sera (gratuitement) mis à jour avec chacune des versions majeures, il est désormais nommé "Deviens un Ninja avec Angular" (sans aucun numéro).

# Chapitre 2. Une rapide introduction à ECMASCIPT 6

Si tu lis ce livre, on peut imaginer que tu as déjà entendu parler de JavaScript. Ce qu'on appelle JavaScript (JS) est une des implémentations d'une spécification standardisée, appelée ECMAScript. La version de la spécification que tu connais le plus est probablement la version 5 : c'est celle utilisée depuis de nombreuses années.

Depuis quelque temps, une nouvelle version de cette spécification est en travaux : ECMASCIPT 6, ES6, ou ECMASCIPT 2015. Je l'appellerai désormais systématiquement ES6, parce que c'est son petit nom le plus populaire. Elle ajoute une tonne de fonctionnalités à JavaScript, comme les classes, les constantes, les *arrow functions*, les générateurs... Il y a tellement de choses qu'on ne peut pas tout couvrir, sauf à y consacrer entièrement ce livre. Mais Angular a été conçu pour bénéficier de cette nouvelle version de JavaScript. Même si tu peux toujours utiliser ton bon vieux JavaScript, tu auras plein d'avantages à utiliser ES6. Ainsi, nous allons consacrer ce chapitre à découvrir ES6, et voir comment il peut nous être utile pour construire une application Angular.

On va laisser beaucoup d'aspects de côté, et on ne sera pas exhaustifs sur ce qu'on verra. Si tu connais déjà ES6, tu peux directement sauter ce chapitre. Sinon, tu vas apprendre des trucs plutôt incroyables qui te serviront à l'avenir même si tu n'utilises finalement pas Angular !

## 2.1. Transpileur

ES6 vient d'atteindre son état final, il n'est pas encore supporté par tous les navigateurs. Et bien sûr, certains navigateurs vont être en retard (mais, pour une fois, Microsoft fait un bon travail avec Edge). Ainsi, on peut se demander à quoi bon présenter le sujet s'il est toujours en pleine évolution ? Et tu as raison, car rares sont les applications qui peuvent se permettre d'ignorer les navigateurs devenus obsolètes. Mais comme tous les développeurs qui ont essayé ES6 ont hâte de l'utiliser dans leurs applications, la communauté a trouvé une solution : un transpileur.

Un transpileur prend du code source ES6 en entrée et génère du code ES5, qui peut tourner dans n'importe quel navigateur. Il génère même les fichiers *source map*, qui permettent de débugger directement le code ES6 depuis le navigateur. Au moment de l'écriture de ces lignes, il y a deux outils principaux pour transpiler de l'ES6 :

- [Traceur](#), un projet Google.
- [Babeljs](#), un projet démarré par Sebastian McKenzie, un jeune développeur de 17 ans (oui, ça fait mal), et qui a reçu beaucoup de contributions extérieures.

Chacun a ses avantages et ses inconvénients. Par exemple Babeljs produit un code source plus lisible que Traceur. Mais Traceur est un projet Google, alors évidemment, Angular et Traceur vont bien ensemble. Le code source d'Angular était d'ailleurs transpilé avec Traceur, avant de basculer en TypeScript. TypeScript est un langage open source développé par Microsoft. C'est un sur-ensemble typé de JavaScript qui compile vers du JavaScript standard, mais nous étudierons cela très bientôt.

Pour parler franchement, Babel est biiiiien plus populaire que Traceur, on aurait donc tendance à

te le conseiller. Le projet devient peu à peu le standard de-facto.

Si tu veux jouer avec ES6, ou le mettre en place dans un de tes projets, jette un œil à ces transpileurs, et ajoute une étape à la construction de ton projet. Elle prendra tes fichiers sources ES6 et générera l'équivalent en ES5. Ça fonctionne très bien mais, évidemment, certaines fonctionnalités nouvelles sont difficiles voire impossibles à transformer, parce qu'elles n'existent tout simplement pas en ES5. Néanmoins, l'état d'avancement actuel de ces transpileurs est largement suffisant pour les utiliser sans problèmes, alors jetons un coup d'œil à ces nouveautés ES6.

## 2.2. let

Si tu pratiques le JS depuis un certain temps, tu dois savoir que la déclaration de variable avec `var` peut être délicate. Dans à peu près tous les autres langages, une variable existe à partir de la ligne contenant la déclaration de cette variable. Mais en JS, il y a un concept nommé *hoisting* ("remontée") qui déclare la variable au tout début de la fonction, même si tu l'as écrite plus loin.

Ainsi, déclarer une variable `name` dans le bloc `if` :

```
function getPonyFullName(pony) {
  if (pony.isChampion) {
    var name = 'Champion ' + pony.name;
    return name;
  }
  return pony.name;
}
```

est équivalent à la déclarer tout en haut de la fonction :

```
function getPonyFullName(pony) {
  var name;
  if (pony.isChampion) {
    name = 'Champion ' + pony.name;
    return name;
  }
  // name is still accessible here
  return pony.name;
}
```

ES6 introduit un nouveau mot-clé pour la déclaration de variable, `let`, qui se comporte enfin comme on pourrait s'y attendre :

```
function getPonyFullName(pony) {  
  if (pony.isChampion) {  
    let name = 'Champion ' + pony.name;  
    return name;  
  }  
  // name is not accessible here  
  return pony.name;  
}
```

L'accès à la variable `name` est maintenant restreint à son bloc. `let` a été pensé pour remplacer définitivement `var` à long terme, donc tu peux abandonner ce bon vieux `var` au profit de `let`. La bonne nouvelle est que ça doit être indolore, et que si ça ne l'est pas, c'est que tu as mis le doigt sur un défaut de ton code !

## 2.3. Constantes

Tant qu'on est sur le sujet des nouveaux mot-clés et des variables, il y en a un autre qui peut être intéressant. ES6 introduit aussi `const` pour déclarer des... constantes ! Si tu déclares une variable avec `const`, elle doit obligatoirement être initialisée, et tu ne pourras plus lui affecter de nouvelle valeur par la suite.

```
const poniesInRace = 6;
```

```
poniesInRace = 7; // SyntaxError
```

Comme pour les variables déclarées avec `let`, les constantes ne sont pas hoisted ("remontées") et sont bien déclarées dans leur bloc.

Il y a un détail qui peut cependant surprendre le profane. Tu peux initialiser une constante avec un objet et modifier par la suite le contenu de l'objet.

```
const PONY = {};  
PONY.color = 'blue'; // works
```

Mais tu ne peux pas assigner à la constante un nouvel objet :

```
const PONY = {};
```

```
PONY = {color: 'blue'}; // SyntaxError
```

Même chose avec les tableaux :

```
const PONIES = [];
PONIES.push({ color: 'blue' }); // works
```

```
PONIES = []; // SyntaxError
```

## 2.4. Crédit d'objets

Ce n'est pas un nouveau mot-clé, mais ça peut te faire tiquer en lisant du code ES6. Il y a un nouveau raccourci pour créer des objets, quand la propriété de l'objet que tu veux créer a le même nom que la variable utilisée comme valeur pour l'attribut.

Exemple :

```
function createPony() {
  const name = 'Rainbow Dash';
  const color = 'blue';
  return { name: name, color: color };
}
```

peut être simplifié en :

```
function createPony() {
  const name = 'Rainbow Dash';
  const color = 'blue';
  return { name, color };
}
```

## 2.5. Affectations déstructurées

Celui-là aussi peut te faire tiquer en lisant du code ES6. Il y a maintenant un raccourci pour affecter des variables à partir d'objets ou de tableaux.

En ES5 :

```
var httpOptions = { timeout: 2000, isCache: true };
// later
var httpTimeout = httpOptions.timeout;
var httpCache = httpOptions.isCache;
```

Maintenant, en ES6, tu peux écrire :

```
const httpOptions = { timeout: 2000, isCache: true };
// later
const { timeout: httpTimeout, isCache: httpCache } = httpOptions;
```

Et tu auras le même résultat. Cela peut être perturbant, parce que la clé est la propriété à lire dans l'objet et la valeur est la variable à affecter. Mais cela fonctionne plutôt bien ! Et même mieux : si la variable que tu veux affecter a le même nom que la propriété de l'objet à lire, tu peux écrire simplement :

```
const httpOptions = { timeout: 2000, isCache: true };
// later
const { timeout, isCache } = httpOptions;
// you now have a variable named 'timeout'
// and one named 'isCache' with correct values
```

Le truc cool est que ça marche aussi avec des objets imbriqués :

```
const httpOptions = { timeout: 2000, cache: { age: 2 } };
// later
const { cache: { age } } = httpOptions;
// you now have a variable named 'age' with value 2
```

Et la même chose est possible avec des tableaux :

```
const timeouts = [1000, 2000, 3000];
// later
const [shortTimeout, mediumTimeout] = timeouts;
// you now have a variable named 'shortTimeout' with value 1000
// and a variable named 'mediumTimeout' with value 2000
```

Bien entendu, cela fonctionne avec des tableaux de tableaux, des tableaux dans des objets, etc.

Un cas d'usage intéressant de cette fonctionnalité est la possibilité de retourner des valeurs multiples. Imagine une fonction `randomPonyInRace` qui retourne un poney et sa position dans la course.

```
function randomPonyInRace() {
  const pony = { name: 'Rainbow Dash' };
  const position = 2;
  // ...
  return { pony, position };
}

const { position, pony } = randomPonyInRace();
```

Cette nouvelle fonctionnalité de déstructuration assigne la `position` retournée par la méthode à la variable `position`, et le poney à la variable `pony`. Et si tu n'as pas usage de la position, tu peux écrire :

```
function randomPonyInRace() {  
  const pony = { name: 'Rainbow Dash' };  
  const position = 2;  
  // ...  
  return { pony, position };  
}  
  
const { pony } = randomPonyInRace();
```

Et tu auras seulement une variable `pony`.

## 2.6. Paramètres optionnels et valeurs par défaut

JS a la particularité de permettre aux développeurs d'appeler une fonction avec un nombre d'arguments variable :

- si tu passes plus d'arguments que déclarés par la fonction, les arguments supplémentaires sont tout simplement ignorés (pour être tout à fait exact, tu peux quand même les utiliser dans la fonction avec la variable spéciale `arguments`).
- si tu passes moins d'arguments que déclarés par la fonction, les paramètres manquants auront la valeur `undefined`.

Ce dernier cas est celui qui nous intéresse. Souvent, on passe moins d'arguments quand les paramètres sont optionnels, comme dans l'exemple suivant :

```
function getPonies(size, page) {  
  size = size || 10;  
  page = page || 1;  
  // ...  
  server.get(size, page);  
}
```

Les paramètres optionnels ont la plupart du temps une valeur par défaut. L'opérateur OR (`||`) va retourner l'opérande de droite si celui de gauche est `undefined`, comme cela serait le cas si le paramètre n'avait pas été fourni par l'appelant (pour être précis, si l'opérande de gauche est *falsy*, c'est-à-dire `undefined`, `0`, `false`, `" "`, etc.). Avec cette astuce, la fonction `getPonies` peut ainsi être invoquée :

```
getPonies(20, 2);  
getPonies(); // same as getPonies(10, 1);  
getPonies(15); // same as getPonies(15, 1);
```

Cela fonctionnait, mais ce n'était pas évident de savoir que les paramètres étaient optionnels, sauf à

lire le corps de la fonction. ES6 offre désormais une façon plus formelle de déclarer des paramètres optionnels, dès la déclaration de la fonction :

```
function getPonies(size = 10, page = 1) {  
    // ...  
    server.get(size, page);  
}
```

Maintenant il est limpide que la valeur par défaut de `size` sera 10 et celle de `page` sera 1 s'ils ne sont pas fournis.

**NOTE**

Il y a cependant une subtile différence, car maintenant `0` ou `""` sont des valeurs valides, et ne seront pas remplacées par les valeurs par défaut, comme `size = size || 10` l'aurait fait. C'est donc plutôt équivalent à `size = size === undefined ? 10 : size;`.

La valeur par défaut peut aussi être un appel de fonction :

```
function getPonies(size = defaultSize(), page = 1) {  
    // the defaultSize method will be called if size is not provided  
    // ...  
    server.get(size, page);  
}
```

ou même d'autres variables, d'autres variables globales, ou d'autres paramètres de la même fonction :

```
function getPonies(size = defaultSize(), page = size - 1) {  
    // if page is not provided, it will be set to the value  
    // of the size parameter minus one.  
    // ...  
    server.get(size, page);  
}
```

Note que si tu essayes d'utiliser des paramètres sur la droite, leur valeur sera toujours `undefined` :

```
function getPonies(size = page, page = 1) {  
    // size will always be undefined, as the page parameter is on its right.  
    server.get(size, page);  
}
```

Ce mécanisme de valeur par défaut ne s'applique pas qu'aux paramètres de fonction, mais aussi aux valeurs de variables, par exemple dans le cas d'une affectation déstructurée :

```
const { timeout = 1000 } = httpOptions;
// you now have a variable named 'timeout',
// with the value of 'httpOptions.timeout' if it exists
// or 1000 if not
```

## 2.7. Rest operator

ES6 introduit aussi une nouvelle syntaxe pour déclarer un nombre variable de paramètres dans une fonction. Comme on le disait précédemment, tu peux toujours passer des arguments supplémentaires à un appel de fonction, et y accéder avec la variable spéciale `arguments`. Tu peux faire quelque chose comme :

```
function addPonies(ponies) {
  for (var i = 0; i < arguments.length; i++) {
    poniesInRace.push(arguments[i]);
  }
}

addPonies('Rainbow Dash', 'Pinkie Pie');
```

Mais tu seras d'accord pour dire que ce n'est ni élégant, ni évident : le paramètre `ponies` n'est jamais utilisé, et rien n'indique que l'on peut fournir plusieurs poneys.

ES6 propose une syntaxe bien meilleure, grâce au *rest operator* `...` ("opérateur de reste").

```
function addPonies(...ponies) {
  for (let pony of ponies) {
    poniesInRace.push(pony);
  }
}
```

`ponies` est désormais un véritable tableau, sur lequel on peut itérer. La boucle `for ... of` utilisée pour l'itération est aussi une nouveauté d'ES6. Elle permet d'être sûr de n'itérer que sur les valeurs de la collection, et non pas sur ses propriétés comme `for ... in`. Ne trouves-tu pas que notre code est maintenant bien plus beau et lisible ?

Le *rest operator* peut aussi fonctionner avec des affectations déstructurées :

```
const [winner, ...losers] = poniesInRace;
// assuming 'poniesInRace' is an array containing several ponies
// 'winner' will have the first pony,
// and 'losers' will be an array of the other ones
```

Le *rest operator* ne doit pas être confondu avec le *spread operator* ("opérateur d'étalement"), même si, on te l'accorde, ils se ressemblent dangereusement ! Le *spread operator* est son opposé : il prend

un tableau, et l'étale en arguments variables. Le seul cas d'utilisation qui me vient à l'esprit serait pour les fonctions comme `min` ou `max`, qui peuvent recevoir des arguments variables, et que tu voudrais appeler avec un tableau :

```
const ponyPrices = [12, 3, 4];
const minPrice = Math.min(...ponyPrices);
```

## 2.8. Classes

Une des fonctionnalités les plus emblématiques, et qui va largement être utilisée dans l'écriture d'applications Angular : ES6 introduit les classes en JavaScript ! Tu pourras désormais facilement faire de l'héritage de classes en JavaScript. C'était déjà possible, avec l'héritage prototypal, mais ce n'était pas une tâche aisée, surtout pour les débutants...

Maintenant c'est les doigts dans le nez, regarde :

```
class Pony {
  constructor(color) {
    this.color = color;
  }

  toString() {
    return `${this.color} pony`;
    // see that? It is another cool feature of ES6, called template literals
    // we'll talk about these quickly!
  }
}
const bluePony = new Pony('blue');
console.log(bluePony.toString()); // blue pony
```

Les déclarations de classes, contrairement aux déclarations de fonctions, ne sont pas *hoisted* ("remontées"), donc tu dois déclarer une classe avant de l'utiliser. Tu as probablement remarqué la fonction spéciale `constructor`. C'est le constructeur, la fonction appelée à la création d'un nouvel objet avec le mot-clé `new`. Dans l'exemple, il requiert une couleur, et nous créons une nouvelle instance de la classe `Pony` avec la couleur "blue". Une classe peut aussi avoir des méthodes, appelables sur une instance, comme la méthode `toString()` dans l'exemple.

Une classe peut aussi avoir des attributs et des méthodes statiques :

```
class Pony {
  static defaultSpeed() {
    return 10;
  }
}
```

Ces méthodes statiques ne peuvent être appelées que sur la classe directement :

```
const speed = Pony.defaultSpeed();
```

Une classe peut avoir des accesseurs (*getters, setters*), si tu veux implémenter du code sur ces opérations :

```
class Pony {  
    get color() {  
        console.log('get color');  
        return this._color;  
    }  
  
    set color(newColor) {  
        console.log('set color ${newColor}');  
        this._color = newColor;  
    }  
}  
  
const pony = new Pony();  
pony.color = 'red';  
// 'set color red'  
console.log(pony.color);  
// 'get color'  
// 'red'
```

Et bien évidemment, si tu as des classes, l'héritage est possible en ES6.

```
class Animal {  
    speed() {  
        return 10;  
    }  
}  
  
class Pony extends Animal {  
}  
  
const pony = new Pony();  
console.log(pony.speed()); // 10, as Pony inherits the parent method
```

Animal est appelée la classe de base, et Pony la classe dérivée. Comme tu peux le voir, la classe dérivée possède toutes les méthodes de la classe de base. Mais elle peut aussi les redéfinir :

```

class Animal {
  speed() {
    return 10;
  }
}
class Pony extends Animal {
  speed() {
    return super.speed() + 10;
  }
}
const pony = new Pony();
console.log(pony.speed()); // 20, as Pony overrides the parent method

```

Comme tu peux le voir, le mot-clé `super` permet d'invoquer la méthode de la classe de base, avec `super.speed()` par exemple.

Ce mot-clé `super` peut aussi être utilisé dans les constructeurs, pour invoquer le constructeur de la classe de base :

```

class Animal {
  constructor(speed) {
    this.speed = speed;
  }
}
class Pony extends Animal {
  constructor(speed, color) {
    super(speed);
    this.color = color;
  }
}
const pony = new Pony(20, 'blue');
console.log(pony.speed); // 20

```

## 2.9. Promises

Les *promises* ("promesses") ne sont pas si nouvelles, et tu les connais ou les utilises peut-être déjà, parce qu'elles tenaient une place importante dans AngularJS 1.x. Mais comme nous les utiliserons beaucoup avec Angular, et même si tu n'utilises que du pur JS sans Angular, on pense que c'est important de s'y attarder un peu.

L'objectif des *promises* est de simplifier la programmation asynchrone. Notre code JS est plein d'asynchronisme, comme des requêtes AJAX, et en général on utilise des *callbacks* pour gérer le résultat et l'erreur. Mais le code devient vite confus, avec des *callbacks* dans des *callbacks*, qui le rendent illisible et peu maintenable. Les *promises* sont plus pratiques que les *callbacks*, parce qu'elles permettent d'écrire du code à plat, et le rendent ainsi plus simple à comprendre. Prenons un cas d'utilisation simple, où on doit récupérer un utilisateur, puis ses droits, puis mettre à jour un menu quand on a récupéré tout ça .

Avec des *callbacks* :

```
getUser(login, function (user) {
  getRights(user, function (rights) {
    updateMenu(rights);
  });
});
```

Avec des *promises* :

```
getUser(login)
  .then(function (user) {
    return getRights(user);
  })
  .then(function (rights) {
    updateMenu(rights);
  })
```

J'aime cette version, parce qu'elle s'exécute comme elle se lit : je veux récupérer un utilisateur, puis ses droits, puis mettre à jour le menu.

Une *promise* est un objet *thenable*, ce qui signifie simplement qu'il a une méthode `then`. Cette méthode prend deux arguments : un callback de succès et un callback d'erreur. Une *promise* a trois états :

- *pending* ("en cours") : quand la *promise* n'est pas réalisée, par exemple quand l'appel serveur n'est pas encore terminé.
- *fulfilled* ("réalisée") : quand la *promise* s'est réalisée avec succès, par exemple quand l'appel HTTP serveur a retourné un status 200-OK.
- *rejected* ("rejetée") : quand la *promise* a échoué, par exemple si l'appel HTTP serveur a retourné un status 404-NotFound.

Quand la promesse est réalisée (*fulfilled*), alors le callback de succès est invoqué, avec le résultat en argument. Si la promesse est rejetée (*rejected*), alors le callback d'erreur est invoqué, avec la valeur rejetée ou une erreur en argument.

Alors, comment crée-t-on une *promise* ? C'est simple, il y a une nouvelle classe `Promise`, dont le constructeur attend une fonction avec deux paramètres, `resolve` et `reject`.

```

const getUser = function (login) {
  return new Promise(function (resolve, reject) {
    // async stuff, like fetching users from server, returning a response
    if (response.status === 200) {
      resolve(response.data);
    } else {
      reject('No user');
    }
  });
};

```

Une fois la *promise* créée, tu peux enregistrer des *callbacks*, via la méthode `then`. Cette méthode peut recevoir deux arguments, les deux *callbacks* que tu veux voir invoqués en cas de succès ou en cas d'échec. Dans l'exemple suivant, nous passons simplement un seul *callback* de succès, ignorant ainsi une erreur potentielle :

```

getUser(login)
  .then(function (user) {
    console.log(user);
  })

```

Quand la promesse sera réalisée, le callback de succès (qui se contente ici de tracer l'utilisateur en console) sera invoqué.

La partie la plus cool c'est que le code peut s'écrire à plat. Si par exemple ton *callback* de succès retourne lui aussi une *promise*, tu peux écrire :

```

getUser(login)
  .then(function (user) {
    return getRights(user) // getRights is returning a promise
      .then(function (rights) {
        return updateMenu(rights);
      });
  })

```

ou plus élégamment :

```

getUser(login)
  .then(function (user) {
    return getRights(user); // getRights is returning a promise
  })
  .then(function (rights) {
    return updateMenu(rights);
  })

```

Un autre truc cool est la gestion d'erreur : tu peux définir une gestion d'erreur par *promise*, ou

globale à toute la chaîne.

Une gestion d'erreur par *promise* :

```
getUser(login)
  .then(function (user) {
    return getRights(user);
  }, function (error) {
    console.log(error); // will be called if getUser fails
    return Promise.reject(error);
})
  .then(function (rights) {
    return updateMenu(rights);
  }, function (error) {
    console.log(error); // will be called if getRights fails
    return Promise.reject(error);
})
```

Une gestion d'erreur globale pour toute la chaîne :

```
getUser(login)
  .then(function (user) {
    return getRights(user);
  })
  .then(function (rights) {
    return updateMenu(rights);
  })
  .catch(function (error) {
    console.log(error); // will be called if getUser or getRights fails
})
```

Tu devrais sérieusement t'intéresser aux *promises*, parce que ça va devenir la nouvelle façon d'écrire des APIs, et toutes les bibliothèques vont bientôt les utiliser. Même les bibliothèques standards : c'est le cas de la nouvelle [Fetch API](#) par exemple.

## 2.10. (*arrow functions*)

Un truc que j'adore dans ES6 est la nouvelle syntaxe *arrow function* ("fonction flèche"), utilisant l'opérateur *fat arrow* ("grosse flèche") : `=>`. C'est très utile pour les *callbacks* et les fonctions anonymes !

Prenons notre exemple précédent avec des *promises* :

```
getUser(login)
  .then(function (user) {
    return getRights(user); // getRights is returning a promise
  })
  .then(function (rights) {
    return updateMenu(rights);
  })
```

Il peut être réécrit avec des *arrow functions* comme ceci :

```
getUser(login)
  .then(user => getRights(user))
  .then(rights => updateMenu(rights))
```

N'est-ce pas super cool ?!

Note que le `return` est implicite s'il n'y a pas de bloc : pas besoin d'écrire `user => return getRights(user)`. Mais si nous avions un bloc, nous aurions besoin d'un `return` explicite :

```
getUser(login)
  .then(user => {
    console.log(user);
    return getRights(user);
  })
  .then(rights => updateMenu(rights))
```

Et les *arrow functions* ont une particularité bien agréable que n'ont pas les fonctions normales : le `this` reste attaché lexicalement, ce qui signifie que ces *arrow functions* n'ont pas un nouveau `this` comme les fonctions normales. Prenons un exemple où on itère sur un tableau avec la fonction `map` pour y trouver le maximum.

En ES5 :

```

var maxFinder = {
  max: 0,
  find: function (numbers) {
    // let's iterate
    numbers.forEach(
      function (element) {
        // if the element is greater, set it as the max
        if (element > this.max) {
          this.max = element;
        }
      });
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);

```

Ca semble pas mal, non ? Mais en fait ça ne marche pas... Si tu as de bons yeux, tu as remarqué que le `forEach` dans la fonction `find` utilise `this`, mais ce `this` n'est lié à aucun objet. Donc `this.max` n'est en fait pas le `max` de l'objet `maxFinder`... On pourrait corriger ça facilement avec un alias :

```

var maxFinder = {
  max: 0,
  find: function (numbers) {
    var self = this;
    numbers.forEach(
      function (element) {
        if (element > self.max) {
          self.max = element;
        }
      });
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);

```

ou en *bindant* le `this` :

```

var maxFinder = {
  max: 0,
  find: function (numbers) {
    numbers.forEach(
      function (element) {
        if (element > this.max) {
          this.max = element;
        }
      }.bind(this));
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);

```

ou en le passant en second paramètre de la fonction `forEach` (ce qui est justement sa raison d'être) :

```

var maxFinder = {
  max: 0,
  find: function (numbers) {
    numbers.forEach(
      function (element) {
        if (element > this.max) {
          this.max = element;
        }
      }, this);
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);

```

Mais il y a maintenant une solution bien plus élégante avec les *arrow functions* :

```

const maxFinder = {
  max: 0,
  find: function (numbers) {
    numbers.forEach(element => {
      if (element > this.max) {
        this.max = element;
      }
    });
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);

```

Les *arrow functions* sont donc idéales pour les fonctions anonymes en *callback* !

## 2.11. Set et Map

On va faire court : on a maintenant de vraies collections en ES6. Youpi \o/!

On utilisait jusque-là de simples objets JavaScript pour jouer le rôle de *map* ("dictionnaire"), c'est à dire un objet JS standard, dont les clés étaient nécessairement des chaînes de caractères. Mais nous pouvons maintenant utiliser la nouvelle classe *Map* :

```

const cedric = { id: 1, name: 'Cedric' };
const users = new Map();
users.set(cedric.id, cedric); // adds a user
console.log(users.has(cedric.id)); // true
console.log(users.size); // 1
users.delete(cedric.id); // removes the user

```

On a aussi une classe *Set* ("ensemble") :

```

const cedric = { id: 1, name: 'Cedric' };
const users = new Set();
users.add(cedric); // adds a user
console.log(users.has(cedric)); // true
console.log(users.size); // 1
users.delete(cedric); // removes the user

```

Tu peux aussi itérer sur une collection, avec la nouvelle syntaxe **for ... of** :

```
for (let user of users) {  
  console.log(user.name);  
}
```

Tu verras que cette syntaxe `for ... of` est celle choisie par l'équipe Angular pour itérer sur une collection dans un template.

## 2.12. Template de string

Construire des strings a toujours été pénible en JavaScript, où nous devions généralement utiliser des concaténations :

```
const fullname = 'Miss ' + firstname + ' ' + lastname;
```

Les templates de string sont une nouvelle fonctionnalité mineure mais bien pratique, où on doit utiliser des accents graves (*backticks `*) au lieu des habituelles apostrophes (*quote '*) ou apostrophes doubles (*double-quotes "*), fournissant un moteur de template basique avec support du multi-ligne :

```
const fullname = `Miss ${firstname} ${lastname}`;
```

Le support du multi-ligne est particulièrement adapté à l'écriture de morceaux d'HTML, comme nous le ferons dans nos composants Angular :

```
const template = `<div>  
  <h1>Hello</h1>  
</div>`;
```

## 2.13. Modules

Il a toujours manqué en JavaScript une façon standard de ranger ses fonctions dans un espace de nommage, et de charger dynamiquement du code. NodeJS a été un leader sur le sujet, avec un écosystème très riche de modules utilisant la convention CommonJS. Côté navigateur, il y a aussi l'API [AMD](#) (Asynchronous Module Definition), utilisé par [RequireJS](#). Mais aucun n'était un vrai standard, ce qui nous conduit à des débats incessants sur la meilleure solution.

ES6 a pour objectif de créer une syntaxe avec le meilleur des deux mondes, sans se préoccuper de l'implémentation utilisée. Le [comité Ecma TC39](#) (qui est responsable des évolutions d'ES6 et auteur de la spécification du langage) voulait une syntaxe simple (c'est indéniablement l'avantage de CommonJS), mais avec le support du chargement asynchrone (comme AMD), et avec quelques bonus comme la possibilité d'analyser statiquement le code par des outils et une gestion claire des dépendances cycliques. Cette nouvelle syntaxe se charge de déclarer ce que tu exportes depuis tes modules, et ce que tu importes dans d'autres modules.

Cette gestion des modules est fondamentale dans Angular, parce que tout y est défini dans des

modules, qu'il faut importer dès qu'on veut les utiliser. Supposons qu'on veuille exposer une fonction pour parier sur un poney donné dans une course, et une fonction pour lancer la course.

Dans `races_service.js`:

```
export function bet(race, pony) {  
    // ...  
}  
export function start(race) {  
    // ...  
}
```

Comme tu le vois, c'est plutôt simple : le nouveau mot-clé `export` fait son travail et exporte les deux fonctions.

Maintenant, supposons qu'un composant de notre application veuille appeler ces deux fonctions.

Dans un autre fichier :

```
import { bet, start } from './races_service';
```

```
// later  
bet(race, pony1);  
start(race);
```

C'est ce qu'on appelle un *named export* ("export nommé"). Ici on importe les deux fonctions, et on doit spécifier le nom du fichier contenant ces deux fonctions, ici '`races_service`'. Evidemment, on peut importer une seule des deux fonctions, si besoin avec un alias :

```
import { start as startRace } from './races_service';
```

```
// later  
startRace(race);
```

Et si tu veux importer toutes les fonctions du module, tu peux utiliser le caractère joker `*`.

Comme tu le ferais dans d'autres langages, il faut utiliser le caractère joker `*` avec modération, seulement si tu as besoin de toutes les fonctions, ou la plupart. Et comme tout ceci sera prochainement géré par ton IDE préféré qui prendra en charge la gestion automatique des imports, on n'aura plus à se soucier d'importer les seules bonnes fonctions.

Avec un caractère joker, tu dois utiliser un alias, et j'aime plutôt ça, parce que ça rend le reste du code plus lisible :

```
import * as racesService from './races_service';
```

```
// later
racesService.bet(race, pony1);
racesService.start(race);
```

Si ton module n'expose qu'une seule fonction, ou valeur, ou classe, tu n'as pas besoin d'utiliser un *named export*, et tu peux bénéficier de l'export par défaut, avec le mot-clé `default`. C'est pratique pour les classes notamment :

```
// pony.js
export default class Pony {
}
// races_service.js
import Pony from './pony';
```

Note l'absence d'accolade pour importer un export par défaut. Tu peux l'importer avec l'alias que tu veux, mais pour être cohérent, c'est mieux de l'importer avec le nom du module (sauf évidemment si tu importes plusieurs modules portant le même nom, auquel cas tu devras donner un alias pour les distinguer). Et bien sûr tu peux mélanger l'export par défaut et l'export nommé, mais un module ne pourra avoir qu'un seul export par défaut.

En Angular, tu utiliseras beaucoup de ces imports dans ton application. Chaque composant et service sera une classe, généralement isolée dans son propre fichier et exportée, et ensuite importée à la demande dans chaque autre composant.

## 2.14. Conclusion

Voilà qui conclue notre rapide introduction à ES6. On a zappé quelques parties, mais si tu as bien assimilé ce chapitre, tu n'auras aucun problème à coder ton application en ES6. Si tu veux approfondir, je te recommande chaudement [Exploring JS](#) par [Axel Rauschmayer](#), ou [Understanding ES6](#) par [Nicholas C. Zakas](#). Ces deux ebooks peuvent être lus gratuitement en ligne, mais pense à soutenir ces auteurs qui ont fait un beau travail ! En l'occurrence, j'ai relu récemment [Speaking JS](#), le précédent livre d'Axel, et j'ai encore appris quelques trucs, donc si tu veux rafraîchir tes connaissances en JS, je te le conseille vivement !

# Chapitre 3. Un peu plus loin qu'ES6

## 3.1. Types dynamiques, statiques et optionnels

Tu sais probablement que les applications Angular peuvent être écrites en ES5, ES6, ou TypeScript. Et tu te demandes peut-être qu'est-ce que TypeScript, et ce qu'il apporte de plus.

JavaScript est dynamiquement typé. Tu peux donc faire des trucs comme :

```
let pony = 'Rainbow Dash';
pony = 2;
```

Et ça fonctionne. Ça offre pleins de possibilités : tu peux ainsi passer n'importe quel objet à une fonction, tant que cet objet a les propriétés requises par la fonction :

```
const pony = { name: 'Rainbow Dash', color: 'blue' };
const horse = { speed: 40, color: 'black' };
const printColor = animal => console.log(animal.color);
// works as long as the object has a `color` property
```

Cette nature dynamique est formidable, mais elle est aussi un handicap dans certains cas, comparée à d'autres langages plus fortement typés. Le cas le plus évident est quand tu dois appeler une fonction inconnue d'une autre API en JS : tu dois lire la documentation (ou pire le code de la fonction) pour deviner à quoi doivent ressembler les paramètres. Dans notre exemple précédent, la méthode `printColor` attend un paramètre avec une propriété `color`, mais encore faut-il le savoir. Et c'est encore plus difficile dans notre travail quotidien, où on multiplie les utilisations de bibliothèques et services développés par d'autres. Un des co-fondateurs de Ninja Squad se plaint souvent du manque de type en JS, et déclare qu'il n'est pas aussi productif, et qu'il ne produit pas du code aussi bon qu'il le ferait dans un environnement plus statiquement typé. Et il n'a pas entièrement tort, même s'il trolle aussi par plaisir ! Sans les informations de type, les IDEs n'ont aucun indice pour savoir si tu écris quelque chose de faux, et les outils ne peuvent pas t'aider à trouver des bugs dans ton code. Bien sûr, nos applications sont testées, et Angular a toujours facilité les tests, mais c'est pratiquement impossible d'avoir une parfaite couverture de tests.

Cela nous amène au sujet de la maintenabilité. Le code JS peut être difficile à maintenir, malgré les tests et la documentation. Refactoriser une grosse application JS n'est pas chose aisée, comparativement à ce qui peut être fait dans des langages statiquement typés. La maintenabilité est un sujet important, et les types aident les outils, ainsi que les développeurs, à éviter les erreurs lors de l'écriture et la modification de code. Google a toujours été enclin à proposer des solutions dans cette direction : c'est compréhensible, étant donné qu'ils gèrent des applications parmi les plus grosses du monde, avec GMail, Google apps, Maps... Alors ils ont essayé plusieurs approches pour améliorer la maintenabilité des applications *front-end* : GWT, Google Closure, Dart... Elles devaient toutes faciliter l'écriture de grosses applications web.

Avec Angular, l'équipe Google voulait nous aider à écrire du meilleur JS, en ajoutant des informations de type à notre code. Ce n'est pas un concept nouveau pour JS, c'était même le sujet de

la spécification ECMASCIPT 4, qui a été abandonnée. Au départ ils annoncèrent AtScript, un sur ensemble d'ES6 avec des annotations (des annotations de type et d'autres, dont je parlerai plus tard). Ils annoncèrent ensuite le support de TypeScript, le langage de Microsoft, avec des annotations de type additionnelles. Et enfin, quelques mois plus tard, l'équipe TypeScript annonçait, après un travail étroit avec l'équipe de Google, que la nouvelle version du langage (1.5) aurait toutes les nouvelles fonctionnalités d'AtScript. L'équipe Angular déclara alors qu'AtScript était officiellement abandonné, et que TypeScript était désormais la meilleure façon d'écrire des applications Angular !

## 3.2. Hello TypeScript

Je pense que c'était la meilleure chose à faire pour plusieurs raisons. D'abord, personne n'a vraiment envie d'apprendre une nouvelle extension de langage. Et TypeScript existait déjà, avec une communauté et un écosystème actifs. Je ne l'avais jamais vraiment utilisé avant Angular, mais j'en avais entendu du bien, de personnes différentes. TypeScript est un projet de Microsoft, mais ce n'est pas le Microsoft de l'ère Ballmer et Gates. C'est le Microsoft de Nadella, celui qui s'ouvre à la communauté, et donc, à l'open-source. Google en a conscience, et c'est tout à leur avantage de contribuer à un projet existant, plutôt que de maintenir le leur. Le framework TypeScript gagnera de son côté en visibilité : *win-win* comme dirait ton manager.

Mais la raison principale de parier sur TypeScript est le système de types qu'il offre. C'est un système optionnel qui vient t'aider sans t'entraver. De fait, après avoir codé quelque temps avec, il s'est fait complètement oublier : tu peux faire des applications Angular en utilisant les trucs de TypeScript les plus utiles (j'y reviendrai) et en ignorant tout le reste avec du pur JavaScript (ES6 dans mon cas). Mais j'aime ce qu'ils ont fait, et on jettera un coup d'oeil à TypeScript dans le chapitre suivant. Tu pourras ainsi lire et comprendre n'importe quel code Angular, et tu pourras décider de l'utiliser, ou pas, ou juste un peu, dans tes applications.

Si tu te demandes "mais pourquoi avoir du code fortement typé dans une application Angular ?", prenons un exemple. Angular 1 et 2 ont été construits sur le puissant concept d'injection de dépendance. Tu le connais déjà peut-être, parce que c'est un *design pattern* classique, utilisé dans beaucoup de frameworks et langages, et notamment AngularJS 1.x comme je le disais.

## 3.3. Un exemple concret d'injection de dépendance

Pour synthétiser ce qu'est l'injection de dépendance, prenons un composant d'une application, disons **RaceList**, permettant d'accéder à la liste des courses que le service **RaceService** peut retourner. Tu peux écrire **RaceList** comme ça :

```

class RaceList {
  constructor() {
    this.raceService = new RaceService();
    // let's say that list() returns a promise
    this.raceService.list()
      // we store the races returned into a member of 'RaceList'
      .then(races => this.races = races);
      // arrow functions, FTW!
  }
}

```

Mais ce code a plusieurs défauts. L'un d'eux est la testabilité : c'est compliqué de remplacer `raceService` par un faux service (un bouchon, un *mock*), pour tester notre composant.

Si nous utilisons le *pattern* d'injection de dépendance (*Dependency Injection*, DI), nous délégons la création de `RaceService` à un framework, lui réclamant simplement une instance. Le framework est ainsi en charge de la création de la dépendance, et il peut nous "l'injecter", par exemple dans le constructeur :

```

class RaceList {
  constructor(raceService) {
    this.raceService = raceService;
    this.raceService.list()
      .then(races => this.races = races);
  }
}

```

Désormais, quand on teste cette classe, on peut facilement passer un faux service au constructeur :

```

// in a test
const fakeService = {
  list: () => {
    // returns a fake promise
  }
};
const raceList = new RaceList(fakeService);
// now we are sure that the race list
// is the one we want for the test

```

Mais comment le framework sait-il quel composant injecter dans le constructeur ? Bonne question ! AngularJS 1.x se basait sur le nom du paramètre, mais cela a une sérieuse limitation : la minification du code va changer le nom du paramètre. Pour contourner ce problème, tu pouvais utiliser la notation à base de tableau, ou ajouter des métadonnées à la classe :

```
RaceList.$inject = ['RaceService'];
```

Il nous fallait donc ajouter des métadonnées pour que le framework comprenne ce qu'il fallait injecter dans nos classes. Et c'est exactement ce que proposent les annotations de type : une métadonnée donnant un indice nécessaire au framework pour réaliser la bonne injection. En Angular, avec TypeScript, voilà à quoi pourrait ressembler notre composant `RaceList` :

```
class RaceList {
  raceService: RaceService;
  races: Array<string>

  constructor(raceService: RaceService) {
    // the interesting part is `: RaceService`
    this.raceService = raceService;
    this.raceService.list()
      .then(races => this.races = races);
  }
}
```

Maintenant l'injection peut se faire sans ambiguïté ! Tu n'es pas obligé d'utiliser TypeScript en Angular, mais clairement ton code sera plus élégant avec. Tu peux toujours faire la même chose en pur ES6 ou ES5, mais tu devras ajouter manuellement des métadonnées d'une autre façon (on y reviendra en détail).

C'est pourquoi nous allons passer un peu de temps à apprendre TypeScript (TS). Angular est clairement construit pour tirer parti d'ES6 et TS 1.5+, et rendre notre vie de développeur plus facile en l'utilisant. Et l'équipe Angular a envie de soumettre le système de type au comité de standardisation, donc peut-être qu'un jour il sera normal d'avoir de vrais types en JS.

Il est temps désormais de se lancer dans TypeScript !

# Chapitre 4. Découvrir TypeScript

TypeScript, qui existe depuis 2012, est un sur-ensemble de JavaScript, ajoutant quelques trucs à 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 compilés en JavaScript standard, en général lors du build, avec le compilateur TypeScript. Le code généré reste très lisible.

```
npm install -g typescript  
tsc test.ts
```

Mais commençons par le début.

## 4.1. Les types de TypeScript

La syntaxe pour ajouter des informations de type en TypeScript est basique :

```
let variable: type;
```

Les différents types sont simples à retenir :

```
const poneyNumber: number = 0;  
const poneyName: string = 'Rainbow Dash';
```

Dans ces cas, les types sont facultatifs, car le compilateur TS peut les deviner depuis leur valeur (c'est ce qu'on appelle l'inférence de type).

Le type peut aussi être défini dans ton application, avec par exemple la classe suivante `Pony` :

```
const pony: Pony = new Pony();
```

TypeScript supporte aussi ce que certains langages appellent des types génériques, par exemple avec un `Array` :

```
const ponies: Array<Pony> = [new Pony()];
```

Cet `Array` ne peut contenir que des poneys, ce qu'indique la notation générique `<>`. On peut se demander quel est l'intérêt d'imposer cela. Ajouter de telles informations de type aidera le compilateur à détecter des erreurs :

```
ponies.push('hello'); // error TS2345
// Argument of type 'string' is not assignable to parameter of type 'Pony'.
```

Et comment faire si tu as besoin d'une variable pouvant recevoir plusieurs types ? TS a un type spécial pour cela, nommé **any**.

```
let changing: any = 2;
changing = true; // no problem
```

C'est pratique si tu ne connais pas le type d'une valeur, soit parce qu'elle vient d'un bout de code dynamique, ou en sortie d'une bibliothèque obscure.

Si ta variable ne doit recevoir que des valeurs de type **number** ou **boolean**, tu peux utiliser l'union de types :

```
let changing: number|boolean = 2;
changing = true; // no problem
```

## 4.2. Valeurs énumérées (enum)

TypeScript propose aussi des valeurs énumérées : **enum**. Par exemple, une course de poneys dans ton application peut être soit **ready**, **started** ou **done**.

```
enum RaceStatus {Ready, Started, Done}
const race = new Race();
race.status = RaceStatus.Ready;
```

Un **enum** est en fait une valeur numérique, commençant à 0. Tu peux cependant définir la valeur que tu veux :

```
enum Medal {Gold = 1, Silver, Bronze}
```

## 4.3. Return types

Tu peux aussi spécifier le type de retour d'une fonction :

```
function startRace(race: Race): Race {
  race.status = RaceStatus.Started;
  return race;
}
```

Si la fonction ne retourne rien, tu peux le déclarer avec **void** :

```
function startRace(race: Race): void {
    race.status = RaceStatus.Started;
}
```

## 4.4. Interfaces

C'est déjà une bonne première étape. Mais comme je le disais plus tôt, JavaScript est formidable par sa nature dynamique. Une fonction marchera si elle reçoit un objet possédant la bonne propriété :

```
function addPointsToScore(player, points) {
    player.score += points;
}
```

Cette fonction peut être appliquée à n'importe quel objet ayant une propriété `score`. Maintenant comment traduit-on cela en TypeScript ? Facile !, on définit une interface, un peu comme la "forme" de l'objet.

```
function addPointsToScore(player: { score: number; }, points: number): void {
    player.score += points;
}
```

Cela signifie que le paramètre doit avoir une propriété nommée `score` de type `number`. Tu peux évidemment aussi nommer ces interfaces :

```
interface HasScore {
    score: number;
}
function addPointsToScore(player: HasScore, points: number): void {
    player.score += points;
}
```

## 4.5. Paramètre optionnel

Y'a un autre truc sympa en JavaScript : les paramètres optionnels. Si tu ne les passes pas à l'appel de la fonction, leur valeur sera `undefined`. Mais en TypeScript, si tu déclares une fonction avec des paramètres typés, le compilateur te gueulera dessus si tu les oublies :

```
addPointsToScore(player); // error TS2346
// Supplied parameters do not match any signature of call target.
```

Pour montrer qu'un paramètre est optionnel dans une fonction (ou une propriété dans une interface), tu ajoutes `?` après le paramètre. Ici, le paramètre `points` est optionnel :

```
function addPointsToScore(player: HasScore, points?: number): void {
  points = points || 0;
  player.score += points;
}
```

## 4.6. Des fonctions en propriété

Tu peux aussi décrire un paramètre comme devant posséder une fonction spécifique plutôt qu'une propriété :

```
function startRunning(pony) {
  pony.run(10);
}
```

La définition de cette interface serait :

```
interface CanRun {
  run(meters: number): void;
}

function startRunning(pony: CanRun): void {
  pony.run(10);
}

const pony = {
  run: (meters) => logger.log(`pony runs ${meters}m`)
};
startRunning(pony);
```

## 4.7. Classes

Une classe peut implémenter une interface. Pour nous, un poney peut courir, donc on pourrait écrire :

```
class Pony implements CanRun {
  run(meters) {
    logger.log(`pony runs ${meters}m`);
  }
}
```

Le compilateur nous obligera à implémenter la méthode `run` dans la classe. Si nous l'implémentons mal, par exemple en attendant une `string` au lieu d'un `number`, le compilateur va crier :

```
class IllegalPony implements CanRun {
    run(meters: string) {
        console.log(`pony runs ${meters}m`);
    }
}
// error TS2420: Class 'IllegalPony' incorrectly implements interface 'CanRun'.
// Types of property 'run' are incompatible.
```

Tu peux aussi implémenter plusieurs interfaces si ça te fait plaisir :

```
class HungryPony implements CanRun, CanEat {
    run(meters) {
        logger.log(`pony runs ${meters}m`);
    }

    eat() {
        logger.log(`pony eats`);
    }
}
```

Et une interface peut en étendre une ou plusieurs autres :

```
interface Animal extends CanRun, CanEat {}

class Pony implements Animal {
    // ...
}
```

Une classe en TypeScript peut avoir des propriétés et des méthodes. Avoir des propriétés dans une classe n'est pas une fonctionnalité standard d'ES6, c'est seulement possible en TypeScript.

```
class SpeedyPony {
    speed = 10;

    run() {
        logger.log(`pony runs at ${this.speed}m/s`);
    }
}
```

Tout est public par défaut. Mais tu peux utiliser le mot-clé `private` pour cacher une propriété ou une méthode. Ajouter `public` ou `private` à un paramètre de constructeur est un raccourci pour créer et initialiser un membre privé ou public :

```

class NamedPony {
  constructor(public name: string, private speed: number) {
  }

  run() {
    logger.log(`pony runs at ${this.speed}m/s`);
  }
}

const pony = new NamedPony('Rainbow Dash', 10);
// defines a public property name with 'Rainbow Dash'
// and a private one speed with 10

```

Ce qui est l'équivalent du plus verbeux :

```

class NamedPonyWithoutShortcut {
  public name: string;
  private speed: number;

  constructor(name: string, speed: number) {
    this.name = name;
    this.speed = speed;
  }

  run() {
    logger.log(`pony runs at ${this.speed}m/s`);
  }
}

```

Ces raccourcis sont très pratiques et nous allons beaucoup les utiliser en Angular !

## 4.8. Utiliser d'autres bibliothèques

Mais si on travaille avec des bibliothèques externes écrites en JS, comment savoir les types des paramètres attendus par telle fonction de telle bibliothèque ? La communauté TypeScript est tellement cool que ses membres ont défini des interfaces pour les types et les fonctions exposés par les bibliothèques JavaScript les plus populaires.

Les fichiers contenant ces interfaces ont une extension spéciale : **.d.ts**. Ils contiennent une liste de toutes les fonctions publiques des bibliothèques. [DefinitelyTyped](#) est l'outil de référence pour récupérer ces fichiers. Par exemple, si tu veux utiliser TS dans ton application AngularJS 1.x, tu peux récupérer le fichier dédié depuis le repository directement avec NPM :

```
npm install --save-dev @types/angular
```

ou le télécharger manuellement. Puis tu inclus ce fichier au début de ton code et hop!, tu profites du bonheur d'avoir une compilation *typesafe* :

```
/// <reference path="angular.d.ts" />
angular.module(10, []); // the module name should be a string
// so when I compile, I get:
// Argument of type 'number' is not assignable to parameter of type 'string'.
```

/// <reference path="angular.d.ts" /> est un commentaire spécial reconnu par TS, qui indique au compilateur de vérifier l'interface angular.d.ts. Maintenant, si tu te trompes dans l'appel d'une méthode AngularJS, le compilateur te le dira, et tu peux corriger sans avoir à lancer manuellement ton application !

Encore plus cool, depuis TypeScript 1.6, le compilateur est capable de trouver par lui-même ces interfaces si elles sont packagées avec ta dépendance dans ton répertoire node\_modules. De plus en plus de projets adoptent cette approche, et Angular fait de même. Tu n'as donc même pas à t'occuper d'inclure ces interfaces dans ton projet Angular : le compilateur TS va tout comprendre comme un grand si tu utilises NPM pour gérer tes dépendances !

## 4.9. Décorateurs

C'est une fonctionnalité toute nouvelle, ajoutée seulement en TypeScript 1.5, juste pour le support d'Angular. En effet, comme on le verra bientôt, les composants Angular peuvent être décrits avec des décorateurs. Tu n'as peut-être jamais entendu parler de décorateurs, car tous les langages ne les proposent pas. Un décorateur est une façon de faire de la métaprogrammation. 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.

Les décorateurs sont vraiment puissants: ils peuvent modifier leur cible (classes, méthodes, etc.) et par exemple modifier les paramètres ou le résultat retourné, appeler d'autres méthodes quand la cible est appelée, ou ajouter des métadonnées destinées à un framework (c'est ce que font les décorateurs d'Angular). Jusqu'à présent, cela n'existe pas en JavaScript. Mais le langage évolue, et il y a maintenant une proposition officielle pour le support des décorateurs, qui les rendra peut-être possibles un jour (probablement avec ES7/ES2016).

En Angular, on utilisera les annotations fournies par le framework. Leur rôle est assez simple: ils ajoutent des métadonnées à nos classes, propriétés ou paramètres pour par exemple indiquer "cette classe est un composant", "cette dépendance est optionnelle", "ceci est une propriété spéciale du composant", etc. Ce n'est pas obligatoire de les utiliser, car tu peux toujours ajouter les métadonnées manuellement si tu ne veux que du pur ES5, mais le code sera plus élégant avec des décorateurs, comme ceux proposés par TypeScript.

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.

Pour mieux comprendre ces décorateurs, essayons d'en construire un très simple par nous-mêmes, `@Log()`, qui va écrire le nom de la méthode à chaque fois qu'elle sera appelée.

Il s'utilisera comme ça :

```
class RaceService {  
  
  @Log()  
  getRaces() {  
    // call API  
  }  
  
  @Log()  
  getRace(raceId) {  
    // call API  
  }  
}
```

Pour le définir, nous devons écrire une méthode renvoyant une fonction comme celle-ci :

```
const Log = function () {  
  return (target: any, name: string, descriptor: any) => {  
    logger.log(`call to ${name}`);  
    return descriptor;  
  };  
};
```

Selon ce sur quoi nous voulons appliquer notre décorateur, la fonction n'aura pas exactement les mêmes arguments. Ici nous avons un décorateur de méthode, qui prend 3 paramètres :

- **target** : la méthode ciblée par notre décorateur
- **name** : le nom de la méthode ciblée
- **descriptor** : le descripteur de la méthode ciblée, par exemple est-ce que la méthode est énumérable, etc.

Nous voulons simplement écrire le nom de la méthode, mais tu pourrais faire pratiquement ce que tu veux : modifier les paramètres, le résultat, appeler une autre fonction, etc.

Ici, dans notre exemple basique, chaque fois que les méthodes `getRace()` ou `getRaces()` sont exécutées, nous verrons une nouvelle trace dans la console du navigateur :

```
raceService.getRaces();  
// logs: call to getRaces  
raceService.getRace(1);  
// logs: call to getRace
```

En tant qu'utilisateur d'Angular, jetons un œil à ces annotations :

```
@Component({ selector: 'ns-home' })
class HomeComponent {

  constructor(@Optional() hello: HelloService) {
    logger.log(hello);
  }
}
```

L'annotation `@Component` est ajoutée à la classe `Home`. Quand Angular chargera notre application, il va trouver la classe `Home`, et va comprendre que c'est un composant grâce au décorateur. Cool, hein ?! Comme tu le vois, une annotation peut recevoir des paramètres, ici un objet de configuration.

Je voulais juste présenter le concept de décorateur, nous aurons l'occasion de revoir tous les décorateurs disponibles en Angular tout au long de ce livre.

Je dois aussi indiquer que tu peux utiliser les décorateurs avec Babel comme transpileur à la place de TypeScript. Il y a même un plugin pour supporter tous les décorateurs Angular : [angular2-annotations](#). Babel supporte aussi les propriétés de classe, mais pas le système de type apporté par TypeScript. Tu peux utiliser Babel, et écrire du code "ES6+", mais tu ne pourras pas bénéficier des types, et ils sont sacrément utiles pour l'injection de dépendances. C'est possible, mais tu devras ajouter d'autres décorateurs pour remplacer les informations de type.

Ainsi mon conseil est d'essayer TypeScript. Tous mes exemples dans ce livre l'utiliseront. Il n'est pas intrusif, tu peux l'utiliser quand c'est utile et l'oublier le reste du temps. Si vraiment tu n'aimes pas, il ne sera pas très compliqué de repasser à ES6 avec Babel ou Traceur, ou même ES5 si tu es complètement fou (mais pour ma part, je trouve le code ES5 d'une application Angular vraiment pas terrible !).

# Chapitre 5. Le monde merveilleux des Web Components

Avant d'aller plus loin, j'aimerais faire une petite pause pour parler des Web Components. Vous n'avez pas besoin de connaître les Web Components pour écrire du code Angular. Mais je pense que c'est une bonne chose d'en avoir un aperçu, car en Angular certaines décisions ont été prises pour faciliter leur intégration, ou pour rendre les composants que l'on construit similaires à des Web Components. Tu es libre de sauter ce chapitre si tu ne t'intéresses pas du tout au sujet, mais je pense que tu apprendras deux-trois choses qui pourraient t'être utiles pour la suite.

## 5.1. Le nouveau Monde

Les composants sont un vieux rêve de développeur. Un truc que tu prendrais sur étagère et lâcherais dans ton application, et qui marcherait directement et apporterait la fonctionnalité à tes utilisateurs sans rien faire.

Mes amis, cette heure est venue.

Oui, bon, peut-être. En tout cas, on a le début d'un truc.

Ce n'est pas complètement neuf. On avait déjà la notion de composants dans le développement web depuis quelques temps, mais ils demandaient en général de lourdes dépendances comme jQuery, Dojo, Prototype, AngularJS, etc. Pas vraiment le genre de bibliothèques que tu veux absolument ajouter à ton application.

Les Web Components essaient de résoudre ce problème : avoir des composants réutilisables et encapsulés.

Ils reposent sur un ensemble de standards émergents, que les navigateurs ne supportent pas encore parfaitement. Mais quand même, c'est un sujet intéressant, même si on ne pourra pas en bénéficier pleinement avant quelques années, ou même jamais si le concept ne décolle pas.

Ce standard émergent est défini dans quatre spécifications :

- Custom elements ("éléments personnalisés")
- Shadow DOM ("DOM de l'ombre")
- Template
- HTML imports

Note que les exemples présentés ont plus de chances de fonctionner dans un Chrome ou un Firefox récent.

## 5.2. Custom elements

Les éléments customs sont un nouveau standard qui permet au développeur de créer ses propres éléments du DOM, faisant de `<ns-pony></ns-pony>` un élément HTML parfaitement valide. La

spécification définit comment déclarer de tels éléments, comment tu peux les faire étendre des éléments existants, comment tu peux définir ton API, etc.

Déclarer un élément custom se fait avec un simple `document.registerElement('ns-pony')` :

```
// new element
var PonyComponent = document.registerElement('ns-pony');
// insert in current body
document.body.appendChild(new PonyComponent());
```

Note que le nom doit contenir un tiret, pour indiquer au navigateur que c'est un élément custom.

Évidemment ton élément custom peut avoir des propriétés et des méthodes, et il aura aussi des callbacks liés au cycle de vie, pour exécuter du code quand le composant est inséré ou supprimé, ou quand l'un de ses attributs est modifié. Il peut aussi avoir son propre template. Par exemple, peut-être que ce `ns-pony` affiche une image du poney, ou seulement son nom :

```
// let's extend HTMLElement
var PonyComponentProto = Object.create(HTMLElement.prototype);
// and add some template using a lifecycle
PonyComponentProto.createdCallback = function() {
  this.innerHTML = '<h1>General Soda</h1>';
};

// new element
var PonyComponent = document.registerElement('ns-pony', {prototype:
PonyComponentProto});
// insert in current body
document.body.appendChild(new PonyComponent());
```

Si tu jettes un coup d'œil au DOM, tu verras `<ns-pony><h1>General Soda</h1></ns-pony>`. Mais cela veut dire que le CSS ou la logique JavaScript de ton application peut avoir des effets indésirables sur ton composant. Donc, en général, le template est caché et encapsulé dans un truc appelé le Shadow DOM ("DOM de l'ombre"), et tu ne verras dans le DOM que `<ns-pony></ns-pony>`, bien que le navigateur affiche le nom du poney.

## 5.3. Shadow DOM

Avec un nom qui claque comme celui-là, on s'attend à un truc très puissant. Et il l'est. 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.

Si on retourne à notre exemple précédent :

```

var PonyComponentProto = Object.create(HTMLElement.prototype);

// add some template in the Shadow DOM
PonyComponentProto.createdCallback = function() {
    var shadow = this.createShadowRoot();
    shadow.innerHTML = '<h1>General Soda</h1>';
};

var PonyComponent = document.registerElement('ns-pony', {prototype:
PonyComponentProto});
document.body.appendChild(new PonyComponent());

```

Si tu essaies maintenant de l'observer, tu devrais voir :

```

<ns-pony>
#shadow-root (open)
<h1>General Soda</h1>
</ns-pony>

```

Désormais, même si tu ajoutes du style aux éléments `h1`, rien ne changera : le Shadow DOM agit comme une barrière.

Jusqu'à présent, nous avions utilisé une chaîne de caractères pour notre template. Mais ce n'est habituellement pas la façon de procéder. La bonne pratique est de plutôt utiliser l'élément `<template>`.

## 5.4. Template

Un template spécifié dans un élément `<template>` n'est pas affiché par le navigateur. Son but est d'être à terme cloné dans un autre élément. Ce que tu déclareras à l'intérieur sera inerte : les scripts ne s'exécuteront pas, les images ne se chargeront pas, etc. Son contenu peut être requêté par le reste de la page avec la méthode classique `getElementById()`, et il peut être placé sans risque n'importe où dans la page.

Pour utiliser un template, il doit être cloné :

```

<template id="pony-tpl">
  <style>
    h1 { color: orange; }
  </style>
  <h1>General Soda</h1>
</template>

var PonyComponentProto = Object.create(HTMLElement.prototype);

// add some template using the template tag
PonyComponentProto.createdCallback = function() {
  var template = document.querySelector('#pony-tpl');
  var clone = document.importNode(template.content, true);
  this.createShadowRoot().appendChild(clone);
};

var PonyComponent = document.registerElement('ns-pony', {prototype:
PonyComponentProto});
document.body.appendChild(new PonyComponent());

```

Et si on pouvait déclarer cela dans un seul fichier, cela nous ferait un composant parfaitement encapsulé... C'est ce que nous allons faire avec les imports HTML !

## 5.5. HTML imports

C'est la dernière spécification. Les imports HTML permettent d'importer du HTML dans du HTML. Quelque chose comme `<link rel="import" href="ns-pony.html">`. Ce fichier, `ns-pony.html`, contiendrait tout ce qui est requis : le template, les scripts, les styles, etc.

Si quelqu'un voulait ensuite utiliser notre merveilleux composant, il lui suffirait simplement d'utiliser un import HTML.

## 5.6. Polymer et X-tag

Toutes ces spécifications constituent les Web Components. Je suis loin d'en être expert, et ils présentent toute sorte de pièges.

Comme les Web Components ne sont pas complètement supportés par tous les navigateurs, il y a un *polyfill* à inclure dans ton application pour être sûr que ça fonctionne. Ce *polyfill* est appelé `web-component.js`, et il est bon de noter qu'il est le fruit d'un effort commun entre Google, Mozilla et Microsoft, entre autres.

Au dessus de ce *polyfill*, quelques bibliothèques ont vu le jour. Elles proposent toutes de faciliter le travail avec les Web Components, et viennent souvent avec un lot de composants tout prêts.

Parmi les initiatives notables, on peut citer :

- [Polymer](#) de Google ;

- X-tag de Mozilla et Microsoft.

Je ne vais pas rentrer dans les détails, mais tu peux facilement utiliser un composant Polymer existant. Supposons que tu veuilles embarquer une carte Google dans ton application :

```
<!-- Polyfill Web Components support for older browsers -->
<script src="webcomponents.js"></script>

<!-- Import element -->
<link rel="import" href="google-map.html">

<!-- Use element -->
<body>
  <google-map latitude="45.780" longitude="4.842"></google-map>
</body>
```

Il y a une tonne de composants disponibles. Tu peux en avoir un aperçu sur <https://customelements.io/>.

Polymer permet aussi de créer tes propres composants :

```
<dom-module id="ns-pony">
  <template>
    <h1>[[name]]</h1>
  </template>
  <script>
    Polymer({
      is: 'ns-pony',
      properties: {
        name: String
      }
    });
  </script>
</dom-module>
```

et de les utiliser :

```
<!-- Polyfill Web Components support for older browsers -->
<script src="webcomponents.js"></script>

<!-- Polymer -->
<link rel="import" href="polymer.html">

<!-- Import element -->
<link rel="import" href="ns-pony.html">

<!-- Use element -->
<body>
  <ns-pony name="General Soda"></ns-pony>
</body>
```

Tu peux faire plein de trucs cools avec Polymer, comme du binding bi-directionnel, donner des valeurs par défaut aux attributs, émettre des événements custom, réagir aux modifications d'attribut, répéter des éléments si tu fournis une collection à un composant, etc.

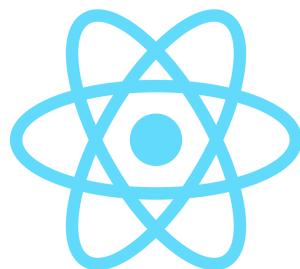
C'est un chapitre trop court pour te montrer sérieusement tout ce que l'on peut faire avec les Web Components, mais tu verras que certains de leurs concepts vont émerger dans la lecture à venir. Et tu verras sans aucun doute que l'équipe Google a conçu Angular pour rendre facile l'utilisation des Web Components aux côtés de nos composants Angular.

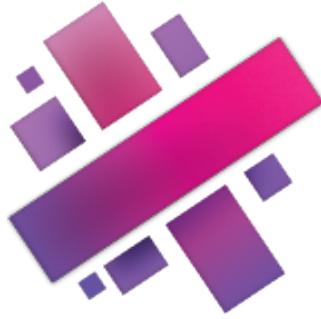
# Chapitre 6. La philosophie d'Angular

Pour construire une application Angular, il te faut saisir quelques trucs sur la philosophie du framework.

Avant tout, Angular est un framework orienté composant. Tu vas écrire de petits composants, et assemblés, ils vont constituer une application complète. Un composant est un groupe d'éléments HTML, dans un template, dédiés à une tâche particulière. Pour cela, tu auras probablement besoin d'un peu de logique métier derrière ce template, pour peupler les données, et réagir aux événements par exemple. Pour les vétérans d'AngularJS 1.x, c'est un peu comme le fameux duo "template / contrôleur", ou une directive.

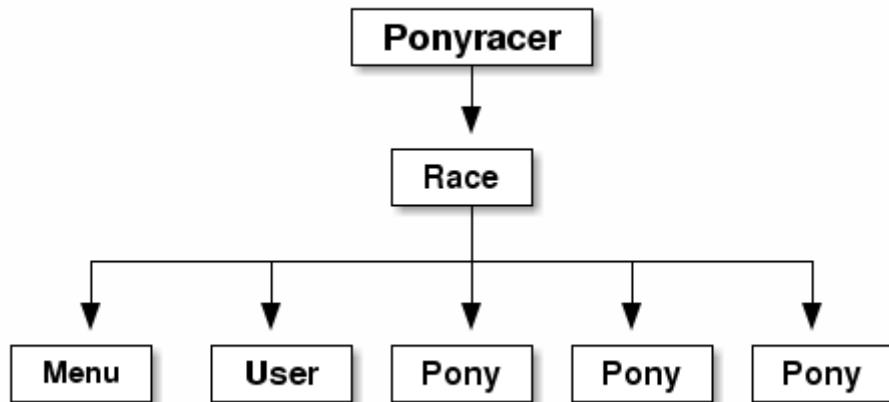
Il faut aussi dire qu'un standard a été défini autour de ces composants : le standard *Web Component* ("composant web"). Même s'il n'est pas encore complètement supporté dans les navigateurs, tu peux déjà construire des petits composants isolés, réutilisables dans différentes applications (ce vieux rêve de programmeur). Cette orientation composant est largement partagée par de nombreux frameworks front-end : c'est le cas depuis le début de [ReactJS](#), le framework tendance de Facebook ; [EmberJS](#) et [AngularJS](#) ont leur propre façon de faire quelque chose de similaire ; et les petits nouveaux [Aurelia](#) ou [Vue.js](#) parient aussi sur la construction de petits composants.





Angular n'est donc pas le seul sur le sujet, mais il est parmi les premiers (ou le premier ?) à considérer sérieusement l'intégration des Web Components (ceux du standard officiel). Mais écartons ce sujet, trop avancé pour le moment.

Tes composants seront organisés de façon hiérarchique, comme le DOM : un composant racine aura des composants enfants, qui auront chacun des composants enfants, etc. Si tu veux afficher une course de poneys (qui ne voudrait pas ?), tu auras probablement une application (Poneyracer), avec une vue enfant (Race), affichant un menu (Menu), l'utilisateur connecté (User), et, évidemment, les poneys (Pony) en course :



Comme tu vas écrire des composants tous les jours (de la semaine au moins), regardons de plus près à quoi ça ressemble. L'équipe Angular voulait aussi bénéficier d'une autre pépite du développement web moderne : ES6 (ou ES2015, si tu préfères). Ainsi tu peux écrire tes composants en ES5 (pas cool !) ou en ES6 (super cool !). Mais cela ne leur suffisait pas, ils voulaient utiliser une fonctionnalité qui n'est pas encore standard : les décorateurs. Alors ils ont travaillé étroitement avec les équipes de transpileurs (Traceur et Babel) et l'équipe Microsoft du projet TypeScript, pour nous permettre d'utiliser des décorateurs dans nos applications Angular. Quelques décorateurs sont disponibles, permettant de déclarer facilement un composant et sa vue. J'espère que tu es au courant, parce que je viens de consacrer deux chapitres à ces sujets !

Par exemple, en simplifiant, le composant Race pourrait ressembler à ça :

```

import { Component } from '@angular/core';
import { RacesService } from './services';

@Component({
  selector: 'ns-race',
  templateUrl: 'race/race.html'
})
export class RaceComponent {

  race: any;

  constructor(racesService: RacesService) {
    racesService.get()
      .then(race => this.race = race);
  }
}

```

Et le template pourrait ressembler à ça :

```

<div>
  <h2>{{ race.name }}</h2>
  <div>{{ race.status }}</div>
  <div *ngFor="let pony of race.ponies">
    <ns-pony [pony]="pony"></ns-pony>
  </div>
</div>

```

Si tu connais déjà AngularJS 1.x, le template doit t'être familier, avec les même expressions entre accolades `{{ }}`, qui seront évaluées et remplacées par les valeurs correspondantes. Certains trucs ont cependant changé : plus de `ng-repeat` par exemple. Je ne veux pas aller trop loin pour le moment, juste te donner un aperçu du code.

Un composant est une partie complètement isolée de ton application. Ton application *est* un composant comme les autres.

Tu regrouperas tes composants au sein d'une ou plusieurs entités cohérentes, appelées des modules (des modules Angular, pas des modules ES6).

Dans un monde idéal, tu prendrais aussi des modules sur étagère fournis par la communauté, et les ajouterais simplement dans ton application pour bénéficier de leurs fonctionnalités.

De tels modules pourraient fournir des composants d'IHM, ou la gestion du glisser-déposer, ou des validations spécifiques pour tes formulaires, et tout ce que tu peux imaginer d'autre.

Dans les chapitres suivants, on explorera quoi mettre en place, comment construire un petit composant, ton premier module, et la syntaxe des templates.

Il y a un autre concept au cœur d'Angular : l'injection de dépendance (*Dependency Injection, DI*).

C'est un pattern très puissant, et tu seras très rapidement séduit après la lecture du chapitre qui lui sera consacré. C'est particulièrement utile pour tester ton application, et j'adore faire des tests, et voir la barre de progression devenir entièrement verte dans mon IDE. Ça me donne l'impression de faire du bon boulot. Il y aura ainsi un chapitre entier consacré à tout tester : tes composants, tes services, ton interface...

Angular a toujours cette sensation magique de la v1, où les modifications sont automatiquement détectées par le framework et appliquées au modèle et à la vue. Mais c'est fait d'une façon très différente : la détection de changement utilise désormais un concept nommé **zones**. On étudiera évidemment tout ça.

Angular est aussi un framework complet, avec plein d'outils pour faciliter les tâches classiques du développement web. Construire des formulaires, appeler un serveur HTTP, du routage d'URL, interagir avec d'autres bibliothèques, des animations, tout ce que tu veux : c'est possible !

Voilà, ça fait pas mal de trucs à apprendre ! Alors commençons par le commencement : initialiser une application et construire notre premier composant.

# Chapitre 7. Commencer de zéro

## 7.1. Créer une application avec TypeScript

Commençons par créer notre première application Angular et notre premier composant, avec un minimum d'outillage. Tu devras installer Node.js et NPM sur ton système. Comme la meilleure façon de le faire dépend de ton système d'exploitation, le mieux est d'aller voir le [site officiel](#). Assure-toi d'avoir une version suffisamment récente de Node.js (en exécutant `node --version`), quelque-chose comme 4.4+. On va écrire notre application en TypeScript, donc tu devras l'installer aussi, via `npm` :

```
npm install -g typescript
```

Ensuite, tu vas créer un nouveau répertoire pour notre expérimentation, et utiliser `tsc` depuis ce nouveau répertoire vide pour y initialiser un projet. `tsc` sont les initiales de **TypeScript Compiler** ("compilateur TypeScript"). Il est fourni par le module NPM `typescript` qu'on vient d'installer globalement :

```
tsc --init --target es5 --sourceMap --experimentalDecorators --emitDecoratorMetadata
```

Cela va créer un fichier, `tsconfig.json`, qui stockera les options de compilation TypeScript. Comme on l'a vu dans les chapitres précédents, on va utiliser TypeScript avec des décorateurs (d'où les deux derniers flags), et on veut que notre code soit transpilé en ECMASCIPT 5, lui permettant d'être exécuté par tout navigateur. L'option `sourceMap` permet de générer les *source maps* ("dictionnaires de code source"), c'est-à-dire des fichiers assurant le lien entre le code ES5 généré et le code TypeScript originel.

Ces *source maps* sont utilisés par le navigateur pour te permettre de débugger le code ES5 qu'il exécute en parcourant le code TypeScript originel que tu as écrit.

On va maintenant utiliser notre IDE préféré. Tu peux utiliser à peu près ce que tu veux, mais tu devrais y activer le support de TypeScript pour plus de confort (et t'assurer qu'il supporte TypeScript 1.5+). Choisis ton IDE préféré: Webstorm, Atom, VisualStudio Code... Ils ont tous un bon support de TypeScript.

Le compilateur TypeScript (et aussi souvent l'IDE) s'appuie sur le fichier `tsconfig.json` pour savoir quelles options il doit utiliser. Le fichier devrait ressembler à celui-ci :

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "experimentalDecorators": true,  
    "emitDecoratorMetadata": true,  
    "sourceMap": true,  
    "module": "commonjs",  
    "noImplicitAny": false  
  },  
  "exclude": [  
    "node_modules"  
  ]  
}
```

Tu peux voir que quelques options ont été ajoutées par défaut. Une intéressante est l'option `module`, qui indique que notre code va être packagé sous forme de modules CommonJS. Cela deviendra important dans un moment.

On doit maintenant ajouter la bibliothèque Angular et notre code. Pour la bibliothèque Angular, on va la télécharger avec NPM, un chouette outil pour gérer ses dépendances.

Pour éviter quelques problèmes, nous allons utiliser NPM version 3. Vérifie quelle version tu as avec :

```
npm -v
```

Si tu n'as pas NPM version 3, tu peux facilement le mettre à jour :

```
npm install -g npm
```

Maintenant que c'est fait, commençons par créer le fichier `package.json`, qui contient toutes les informations dont NPM a besoin. Tu peux répondre par `Entrée` à toutes les questions.

```
npm init
```

Ensuite, installons Angular et ses dépendances .

#### NOTE

Cet ebook utilise Angular version 4.0.0 dans les exemples. La commande ci-dessous va installer la version la plus récente, qui ne sera peut-être pas la même. Pour utiliser la même version que nous, ajoute `@4.0.0` à chaque package Angular, comme `@angular/core@4.0.0`. Cela t'évitera peut-être quelques problèmes ! Angular étant très modulaire, nous devrons donc installer un certain nombre de packages pour le framework lui-même, et pour ses dépendances.

```
npm install --save @angular/core@"$NG" @angular/compiler@"$NG" @angular/common@"$NG"  
@angular/platform-browser@"$NG" @angular/platform-browser-dynamic@"$NG" rxjs reflect-  
metadata zone.js
```

Tu peux jeter un coup d'œil au fichier `package.json`, il devrait désormais contenir les dépendances suivantes :

- les différents packages `@angular`.
- `reflect-metadata`, parce que nous utilisons les décorateurs.
- `rxjs`, une bibliothèque vraiment cool appelée RxJS pour la programmation réactive. On aura un chapitre entier consacré à ce sujet.
- et enfin, le module `zone.js`, qui assure la plomberie pour faire tourner notre code dans des zones isolées et y détecter les changements (on y reviendra aussi plus tard).

Dernière chose pour faire plaisir au compilateur, tu dois installer les *typings* pour tout ce qui a trait à ES6. Le plus simple est d'installer les *typings* pour `core-js` :

```
npm install --save-dev @types/core-js
```

L'outillage est désormais en place, il est temps de créer notre premier composant !

## 7.2. Notre premier composant

Crée un nouveau fichier, nommé `app.component.ts`.

Maintenant on est prêt à lancer le compilateur TypeScript, en utilisant le *watch mode* ("mode de surveillance") pour compiler en tâche de fond dès la sauvegarde. Il se peut aussi que ton IDE puisse s'en charger.

```
tsc --watch
```

Cela devrait afficher quelque chose comme :

```
Compilation complete. Watching for file changes.
```

Tu peux désormais laisser ce compilateur tourner en fond et ouvrir une nouvelle ligne de commande pour la suite.

Dès que tu sauveras ce fichier, tu devrais voir apparaître un nouveau fichier `app.component.js` dans le répertoire : c'est le compilateur TypeScript qui fait son travail. Tu devrais aussi voir le fichier *source map*. Sinon, c'est que tu as probablement arrêté ton compilateur TypeScript qui surveillait les changements, alors tu devrais le lancer à nouveau avec `tsc --watch`, et le laisser tourner en fond.

Comme on l'a vu dans le chapitre précédent, un composant est la combinaison d'une vue (le template) et de logique (notre classe TS). Créons une classe :

```
export class PonyRacerAppComponent {  
}
```

Notre application elle-même est un simple composant. Pour l'indiquer à Angular, on utilise le décorateur **@Component**. Et pour pouvoir l'utiliser, il nous faut l'importer :

```
import {Component} from '@angular/core';  
  
@Component()  
export class PonyRacerAppComponent {  
}
```

Si ton IDE le supporte, la complétion de code devrait fonctionner car la dépendance Angular a ses propres fichiers **d.ts** dans le répertoire **node\_modules**, et TypeScript est capable de le détecter. Tu peux même naviguer vers les définitions de type si tu le souhaites.

TypeScript apporte sa vérification de types, donc tu verras les erreurs dès que tu les tapes. Mais les erreurs ne sont pas nécessairement bloquantes : si tu as oublié d'ajouter les informations de type sur ta variable, le code compilera quand même en JavaScript et s'exécutera correctement.

J'essaie de mon côté de garder le compte des erreurs TypeScript à zéro, mais tu peux faire comme tu veux. Comme nous utilisons des *source maps*, tu peux voir le code TypeScript directement dans ton navigateur, et même débugger ton application en positionnant des points d'arrêt directement dedans.

Le décorateur **@Component** attend un objet de configuration. On verra plus tard en détails ce qu'on peut y configurer, mais pour le moment une seule propriété est requise : **selector**. Elle indiquera à Angular ce qu'il faudra chercher dans nos pages HTML. A chaque fois que le sélecteur défini sera trouvé dans notre HTML, Angular remplacera l'élément sélectionné par notre composant :

```
import {Component} from '@angular/core';  
  
@Component({  
  selector: 'ponyracer-app'  
})  
export class PonyRacerAppComponent {  
}
```

Donc ici, chaque fois que notre HTML contiendra un élément `<ponyracer-app></ponyracer-app>`, Angular créera une nouvelle instance de notre classe **PonyRacerAppComponent**.

**NOTE**

Il n'y a pas de convention de nommage encore clairement établie. J'ai tendance à suffixer mes classes de composants avec **Component**. Les sélecteurs de composant devraient avoir un tiret, comme **ns-pony**, même si ce n'est pas obligatoire. Mais, si tu veux laisser d'autres développeurs utiliser tes composants et éviter de potentiels conflits de nom, il est de bon ton d'adopter une convention comme "namespace-composant". Le *namespace* devrait être court, comme "ns" pour Ninja Squad par exemple. Cela donnerait un composant réutilisable avec un sélecteur **ns-pony**. Enfin, tu peux suffixer tes noms de fichiers pour que leur rôle soit évident au premier coup d'œil, par exemple **pony.component.ts** ou **race.service.ts**.

Un composant doit aussi avoir un template. On pourrait externaliser le template dans un autre fichier, mais contentons-nous de faire simple la première fois, et mettons le en ligne :

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

@Component({
  selector: 'ponyracer-app',
  template: '<h1>PonyRacer</h1>'
})
export class PonyRacerAppComponent { }
```

N'oublie pas d'importer le décorateur **Component**. Tu l'oublieras régulièrement au début, mais tu t'y feras vite, parce que le compilateur ne va cesser de t'insulter ! ;)

Tu verras que l'essentiel de nos besoins se situe dans le module **@angular/core**, mais ce n'est pas toujours le cas. Par exemple, quand on fera du HTTP, on utilisera des imports de **@angular/http**, ou quand on utilisera le routeur, on importera depuis **@angular/router**, etc.

## 7.3. Notre premier module Angular

Comme nous l'avons brièvement évoqué dans le chapitre précédent, nous allons regrouper nos composants et les autres parties que nous verrons plus tard dans des entités cohérentes : des modules Angular.

Un module Angular est différent des modules ES6 que nous avons croisés plus tôt : nous parlons ici de modules **applicatifs**.

Notre application aura toujours au moins un module, le **module racine**. Plus tard, peut-être, lorsque notre application grossira, nous ajouterons d'autres modules, par fonctionnalité. Par exemple, nous pourrions ajouter un module dédié à la partie Administration de notre application, contenant tous les composants et la logique métier de cette partie. Mais nous reviendrons là-dessus plus tard. Nous verrons aussi que les librairies tierces et Angular lui-même exposent des modules, que nous pouvons utiliser dans notre application.

Pour définir un module Angular pour notre petite application, nous devons créer une classe. Habituellement, cela est fait dans un fichier séparé, appelé **app.module.ts** pour le module racine.

Cette classe doit être décorée avec `@NgModule`.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports: [BrowserModule],
})
export class AppModule { }
```

Comme le décorateur `@Component`, il reçoit un objet de configuration.

Puisque nous construisons une application pour le navigateur, le module racine devra importer `BrowserModule`. Ce n'est la seule cible possible pour Angular, nous pouvons choisir de rendre l'application sur le serveur par exemple, et devrions ainsi importer un autre module. `BrowserModule` contient beaucoup de choses très utiles pour la suite. Un module peut choisir d'*exporter* des composants, directives et *pipes*. Quand tu importes un module, tu rends toutes les directives, composants et *pipes* exportés par ce module utilisables dans ton module. Notre module racine ne sera pas importé dans d'autres modules, donc nous n'avons pas d'`exports`', mais nous aurons plusieurs `imports` à la fin.

La terminologie n'est pas simple quand on débute. On parle de modules ES6 et TS dans les premiers chapitres, qui définissent des imports et des exports. Et maintenant nous parlons de modules Angular, qui ont aussi des imports et des exports... Que les mêmes termes désignent des concepts différents ne me semble pas une riche idée, alors laisse-moi expliquer tout ça un peu plus.

Tu peux voir un import ES6 ou TS purement comme une fonctionnalité du langage, comme un import en Java : il permet d'utiliser la classe/fonction importée dans notre code source. Cela déclare aussi une dépendance pour le *bundler* ou le *module loader* (Webpack ou SystemJS, par exemple), qui savent que si `a.ts` est chargé, alors `b.ts` doit être chargé également si `a` importe `b`. Tu as besoin des imports et exports avec ES6 et TypeScript, que tu utilises ou pas Angular ou un autre framework.

D'un autre côté, importer un module Angular (par exemple `BrowserModule`) dans ton propre module Angular (`AppModule`), a une signification fonctionnelle. Cela indique à Angular : tous les composants, directives et *pipes* qui sont exportés par `BrowserModule` doivent être rendus disponibles pour mes composants/templates Angular. Cela n'a aucune signification particulière pour le compilateur TypeScript.

Revenons à `NgModule` : nous devons déclarer les composants qui appartiennent à notre module racine dans l'attribut `declarations` de son objet de configuration. Ajoutons le composant que nous avons développé : `PonyracerAppComponent`.

```

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

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

```

Comme ce module est notre module racine, nous devons également indiquer à Angular quel composant est le composant racine, c'est à dire le composant que nous devons instancier quand l'application démarre. C'est l'objectif du champ `bootstrap` de l'objet de configuration :

```

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

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

```

Notre module est prêt, démarrons l'application !

## 7.4. Démarrer l'application

Enfin, on doit démarrer l'application, avec la méthode `bootstrapModule`. Cette méthode est présente sur un objet retourné par une méthode appelée `platformBrowserDynamic`. Il nous faut aussi l'importer, depuis `@angular/platform-browser-dynamic`. C'est quoi ce module bizarre ?! Pourquoi n'est-ce pas `@angular/core`? Bonne question : c'est parce que tu pourrais avoir envie de faire tourner ton application ailleurs que dans un navigateur, parce qu'Angular permet le rendu côté serveur, ou peut tourner dans un Web Worker par exemple. Et dans ces cas, la logique de démarrage sera un peu différente. Mais on verra cela plus tard, on va se contenter d'un navigateur pour le moment.

Créons un nouveau fichier, par exemple `main.ts`, pour séparer la logique de démarrage :

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

platformBrowserDynamic().bootstrapModule(AppModule);
```

Youpi ! Mais attends voir. On n'a pas encore de fichier HTML, si ? T'as raison !

Crée un autre fichier nommé `index.html` et ajoutes-y le contenu suivant :

```
<html>

<head></head>

<body>
  <ponyracer-app>
    You will see me while Angular starts the app!
  </ponyracer-app>
</body>

</html>
```

Il nous faut maintenant ajouter nos scripts dans nos fichiers HTML. En AngularJS 1.x, c'était simple : il te suffisait d'ajouter un script pour `angular.js`, et un script pour chacun des fichiers JS que tu avais écrits, et c'était bon. Il y avait cependant un inconvénient : tout devait être chargé statiquement, dès le démarrage, ce qui pouvait allonger lourdement les temps de chargement pour les grosses applications.

Avec Angular, c'est plus complexe, mais aussi bien plus puissant. Angular est maintenant modulaire, et chaque module (module ES6) peut être chargé dynamiquement. Notre application est donc aussi modulaire, comme on l'a vu.

Il y a cependant quelques problèmes :

- la notion de module n'existe pas en ES5, et les navigateurs ne supportent que ES5 pour le moment ;
- les concepteurs d'ES6 ont décidé de spécifier comment les modules étaient définis et importés. Mais ils n'ont pas encore spécifié comment ils devaient être packagés et chargés par les navigateurs.

Pour charger nos modules, il nous faudra donc s'appuyer sur un outil : [SystemJS](#). SystemJS est un petit chargeur de modules : tu l'ajoutes (statiquement) dans ta page HTML, tu lui indiques où sont situés les modules sur le serveur, et tu charges l'un d'eux. Il déterminera automatiquement les dépendances entre les modules, et téléchargerá ceux utilisés par ton application.

Cela va entraîner des pelletées de téléchargements de fichiers JS. Si cela n'est pas un problème pendant le développement, ça le devient en production. Heureusement, SystemJS vient aussi avec un outil qui peut emballer plusieurs petits modules dans un plus gros paquet. Quand un module est requis, le paquet contenant ce module (et plusieurs autres) sera alors téléchargé.

Note que ce n'est pas le seul outil possible pour ce travail, tu peux par exemple utiliser [Webpack](#) si tu veux.

Installons SystemJS :

```
npm install --save systemjs
```

On doit charger [SystemJS](#) statiquement, et lui indiquer où se situe notre module contenant la logique de démarrage (dans `main`). On doit aussi lui indiquer où trouver les dépendances de notre application, comme `@angular`. Mais d'abord, il nous faut inclure `reflect-metadata` et `zone.js` :

```
<html>

<head>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/reflect-metadata/Reflect.js"></script>
  <script src="node_modules/systemjs/dist/system.js"></script>
  <script>
    System.config({
      // the app will need the following dependencies
      map: {
        '@angular/core': 'node_modules/@angular/core/bundles/core.umd.js',
        '@angular/common': 'node_modules/@angular/common/bundles/common.umd.js',
        '@angular/compiler': 'node_modules/@angular/compiler/bundles/compiler.umd.js',
        '@angular/platform-browser': 'node_modules/@angular/platform-
        browser/bundles/platform-browser.umd.js',
        '@angular/platform-browser-dynamic': 'node_modules/@angular/platform-browser-
        dynamic/bundles/platform-browser-dynamic.umd.js',
        'rxjs': 'node_modules/rxjs'
      },
      packages: {
        // we want to import our modules without writing '.js' at the end
        // we declare them as packages and SystemJS will add the extension for us
        '.': {}
      }
    });
    // and to finish, let's boot the app!
    System.import('main');
  </script>
</head>

<body>
  <ponyracer-app>
    You will see me while Angular starts the app!
  </ponyracer-app>
</body>

</html>
```

OK ! Démarrons maintenant un serveur HTTP pour servir notre mini application. Je vais utiliser [http-server](#), un outil Node.js qui fait ce que laisse deviner son nom. Mais tu peux évidemment utiliser ton serveur web préféré : Apache, Nginx, Tomcat, etc. Pour l'installer, on utilisera [npm](#):

```
npm install -g http-server
```

Pour le démarrer, va dans ton répertoire, et entre :

```
http-server
```

Maintenant c'est la grande première ! Ouvre ton navigateur sur <http://localhost:8080>.

Tu devrais y voir brièvement "You will see this while Angular start the app!", puis ensuite "PonyRacer" devrait apparaître ! Ton premier composant est un succès !

Bon, OK, pour le moment ce n'est pas vraiment une application dynamique, on aurait pu faire la même chose en une seconde dans une page HTML statique. Alors jetons-nous sur les chapitres suivants, et apprenons tout de l'injection de dépendances et du système de templates.

## 7.5. Commencer de zéro avec Angular CLI

Dans un vrai projet, il te faudra probablement mettre en place d'autres choses comme :

- des tests pour vérifier que tu n'as pas introduit de régressions ;
- un outil de construction, pour orchestrer différentes tâches (compiler, tester, packager, etc.)

Et c'est un peu laborieux de tout mettre en place tout seul, même si je pense que c'est utile de le faire au moins une fois pour comprendre tout ce qui se passe.

Ces dernières années, plusieurs petits projets ont vu le jour, tous basés sur le formidable [Yeoman](#). C'était déjà le cas avec AngularJS 1.x, et il y a déjà plusieurs tentatives avec Angular.

Mais cette fois-ci, l'équipe Google a travaillé sur le sujet, et ils en ont sorti ceci : [Angular CLI](#).

[Angular CLI](#) est un outil en ligne de commande pour démarrer rapidement un projet, déjà configuré avec Webpack comme un outil de construction, des tests, du packaging, etc.

Cette idée n'est pas nouvelle, et est d'ailleurs piquée d'un autre framework populaire : EmberJS et son [ember-cli](#) largement plébiscité.

L'outil est encore en développement, mais je pense qu'il va devenir le standard de fait pour créer une application Angular dans le futur, alors tu peux l'essayer.

En fait, tu devrais vraiment l'essayer : tu auras l'équivalent de ce que nous venons de faire manuellement, plus une tonne de choses utiles.

```
npm i -g @angular/cli  
ng new ponyracer
```

Cela va créer un squelette de projet. Tu peux démarrer l'application avec :

```
ng serve
```

Cela va démarrer un petit serveur HTTP localement, avec rechargement à chaud. Ainsi, à chaque modification de fichier, l'application sera rafraîchie dans le navigateur.

Tu peux aussi créer un squelette de composant :

```
ng generate component pony
```

Cela va créer un fichier de composant, avec son template associé, sa feuille de style, et un fichier de test.

L'outil n'est pas seulement là pour nous aider à développer notre application : il vient avec un système de plugins qui vont faciliter d'autres tâches comme le déploiement. Par exemple, tu peux rapidement déployer sur Github Pages, avec le plugin [github-pages](#) :

```
ng github-pages:deploy
```

À terme, cela devrait être formidable ! On partagera la même organisation du code à travers les projets, une façon commune de construire et déployer les applications, et probablement un large écosystème de plugins pour simplifier certaines tâches.

Alors va jeter un œil à [Angular CLI](#) !

## MISE EN PRATIQUE

Essaye notre exercice [Getting Started](#) ! Il est gratuit et fait partie de notre Pack pro, où tu apprends à construire une application complète étape par étape. Le premier exercice est consacré à démarrer une application avec Angular CLI !

# Chapitre 8. La syntaxe des templates

On a vu qu'un composant a besoin de sa vue. Pour définir une vue, tu peux définir un template *inline* (dans le code du composant), ou dans un fichier séparé. Tu es probablement familier avec une syntaxe de template, peut-être même avec celle d'AngularJS 1.x. Pour simplifier, un template nous permet de rendre du HTML avec quelques parties dynamiques dépendant de tes données.

Angular a sa propre syntaxe de template que nous devons donc apprendre avant d'aller plus loin.

Prenons un exemple simple. Notre premier composant ressemblait à :

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

@Component({
  selector: 'ponyracer-app',
  template: '<h1>PonyRacer</h1>'
})
export class PonyRacerAppComponent {
```

Supposons qu'on veuille afficher des données dynamiques dans notre première page, par exemple le nombre d'utilisateurs enregistrés dans notre application. Plus tard nous verrons comment récupérer des données depuis un serveur, mais pour le moment ce nombre d'utilisateurs sera directement hardcodé dans notre classe :

```
@Component({
  selector: 'ponyracer-app',
  template: '<h1>PonyRacer</h1>'
})
export class PonyRacerAppComponent {

  numberOfWorkers = 146;

}
```

Maintenant, comment doit-on modifier notre template pour afficher cette variable ? Grâce à l'interpolation !

## 8.1. Interpolation

L'interpolation est un bien grand mot pour un concept simple.

Un exemple rapide :

```

@Component({
  selector: 'ponyracer-app',
  template: `
    <h1>PonyRacer</h1>
    <h2>{{numberOfUsers}} users</h2>
  `,
})
export class PonyRacerAppComponent {
  numberOfUsers = 146;
}

```

Nous avons un composant `PonyRacer`, qui sera activé dès qu'Angular tombera sur une balise `<ponyracer-app>`. La classe `PonyRacerAppComponent` a une propriété, `numberOfUsers`. Et le template a été enrichi avec une balise `<h2>`, utilisant la fameuse notation avec double-accolades (les "moustaches") pour indiquer que cette expression doit être évaluée. Ce type de templating est de l'interpolation.

On devrait maintenant voir dans le navigateur :

```

<ponyracer-app>
  <h1>PonyRacer</h1>
  <h2>146 users</h2>
</ponyracer-app>

```

car `{{numberOfUsers}}` a été remplacé par sa valeur.

Quand Angular détecte un élément `<ponyracer-app>` dans une page, il crée une instance de la classe `PonyRacerAppComponent`, et cette instance sera le contexte d'évaluation des expressions dans le template. Ici la classe `PonyRacerAppComponent` affecte `146` à la propriété `numberOfUsers`, donc nous voyons '146' affiché à l'écran.

La magie surviendra quand la valeur de `numberOfUsers` sera modifiée dans notre objet, et que le template sera alors automatiquement mis à jour ! Cela s'appelle *change detection* ("la détection de changement"), et c'est une des grandes fonctionnalités d'Angular.

Il faut cependant se rappeler une chose importante : si on essaye d'afficher une variable qui n'existe pas, au lieu d'afficher `undefined`, Angular affichera une chaîne vide. Et de même pour une variable `null`.

Maintenant, au lieu d'une valeur simple, disons que notre composant a un objet `user` plus complexe, décrivant l'utilisateur courant.

```

@Component({
  selector: 'ponyracer-app',
  template: `
    <h1>PonyRacer</h1>
    <h2>Welcome {{user.name}}</h2>
  `,
})
export class PonyRacerAppComponent {
  user: any = { name: 'Cédric' };
}

```

Comme tu le vois, on peut interroger des expressions plus complexes, y compris accéder à des propriétés dans un objet.

```

<ponyracer-app>
  <h1>PonyRacer</h1>
  <h2>Welcome Cédric</h2>
</ponyracer-app>

```

Que se passe-t-il si on a une faute de frappe dans notre template, avec une propriété qui n'existe pas dans la classe ?

```

@Component({
  selector: 'ponyracer-app',
  // typo: users is not user!
  template: `
    <h1>PonyRacer</h1>
    <h2>Welcome {{users.name}}</h2>
  `,
})
export class PonyRacerAppComponent {
  user: any = { name: 'Cédric' };
}

```

En chargeant l'application, tu auras une erreur, indiquant que cette propriété n'existe pas :

```
Cannot read property 'name' of undefined in [{{users.name}}] in PonyRacerAppComponent]
```

C'est plutôt cool, parce que tu peux être sûr que tes templates sont corrects. Un des problèmes les plus fréquents en AngularJS 1.x était que ce genre d'erreurs n'était pas détecté, et tu pouvais ainsi perdre pas mal de temps à comprendre ce qu'il se passait (traditionnellement, une faute de frappe genre `{{users.name}}` au lieu de `{{user.name}}`). Pour avoir donné souvent des formations

AngularJS 1.x, je peux te garantir qu'au moins 30% des débutants rencontraient ce problème le premier jour. Ça m'agaçait un peu, alors j'ai même soumis une [pull-request](#) pour afficher un *warning* quand le parseur rencontrait une variable inconnue, mais elle a été refusée pour de bonnes raisons, avec un commentaire de l'équipe précisant qu'ils avaient une idée de comment corriger cela en Angular. La voilà finalement concrétisée !

Dernière petite fonctionnalité : que se passe-t-il si mon objet `user` est en fait récupéré depuis le serveur, et donc indéfini dans mon composant au début ? Que pouvons-nous faire pour éviter les erreurs quand le template est compilé ?

Facile : plutôt que d'écrire `user.name`, nous écrirons `user?.name` :

```
@Component({
  selector: 'ponyracer-app',
  // user is undefined
  // but the ?. will avoid the error
  template: `
    <h1>PonyRacer</h1>
    <h2>Welcome {{user?.name}}</h2>
  `
})
export class PonyRacerAppComponent {
  user: any;
}
```

Et nous n'avons plus d'erreur ! Le `?` est parfois appelé "**Safe Navigation Operator**" (opérateur de navigation sûre).

Ainsi on peut écrire nos templates plus sereinement, et être assurés qu'ils vont se comporter correctement.

Retournons à notre exemple. On affiche désormais un message de bienvenue. Allons un peu plus loin, et essayons d'afficher une liste des courses de poneys à venir.

Cela nous amène à écrire notre deuxième composant. Pour le moment, faisons simple :

```
// in another file, races.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'ns-races',
  template: '<h2>Races</h2>'
})
class RacesComponent {
```

Rien d'extraordinaire : une simple classe, décorée avec `@Component` pour indiquer un `selector` d'ancrage et un template *inline*.

Si on veut maintenant inclure ce composant dans le template de `PonyRacerAppComponent`, comment faire ?

## 8.2. Utiliser d'autres composants dans nos templates

On a notre composant application, `PonyRacerAppComponent`, où on veut afficher le composant listant les courses de poneys, `RacesComponent`.

```
// in ponyracer_app.ts
import { Component } from '@angular/core';

@Component({
  selector: 'ponyracer-app',
  // added the RacesComponent component
  template: `
    <h1>PonyRacer</h1>
    <ns-races></ns-races>
  `
})
export class PonyRacerAppComponent {
```

On a donc ajouté le composant `RacesComponent` dans le template, en incluant une balise dont le nom correspond au `selector` défini dans le composant.

Maaaaais, ça ne fonctionnera pas : le navigateur n'affichera pas ce composant listant les courses.

Pourquoi cela? La raison est simple: Angular ne connaît pas encore ce composant `RacesComponent`.

Mais la solution est simple. Tu te souviens qu'on a dû ajouter `PonyRacerAppComponent` dans les `declarations` du décorateur `@NgModule`? A présent, comme on a un deuxième composant, il doit être déclaré également.

`RacesComponent` n'est pas le composant racine de notre application, donc il doit être dans les déclarations, mais pas dans le tableau `bootstrap`.

```

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

import { PonyRacerAppComponent } from './app.component';
// do not forget to import the component
import { RacesComponent } from './races.component';

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

```

Note aussi que comme tu passes directement la classe, il te faudra l'importer préalablement.

Pour importer `RacesComponent` dans la classe `PonyRacerAppComponent`, tu dois exporter la classe `RacesComponent` dans son fichier source `races.component.ts` (relis le chapitre [modules ES6](#) si ce n'est pas clair). Ainsi `RacesComponent` va ressembler à :

```

// in another file, races.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'ns-races',
  template: '<h2>Races</h2>'
})
export class RacesComponent {
}

```

Désormais, notre composant listant les courses sera fièrement affiché dans le navigateur :

```

<ponyracer-app>
  <h1>PonyRacer</h1>
  <ns-races>
    <h2>Races</h2>
  </ns-races>
</ponyracer-app>

```

## 8.3. Binding de propriété

L'interpolation n'est qu'une des façons d'avoir des morceaux dynamiques dans nos templates.

En fait, l'interpolation n'est qu'une simplification du concept au cœur du moteur de template

d'Angular : le *binding* de propriété.

En Angular, on peut écrire dans toutes les propriétés du DOM via des attributs spéciaux sur les éléments HTML, entourés de crochets `[]`. Ça fait bizarre au premier abord, mais en fait c'est du HTML valide (et ça m'a aussi surpris). Un nom d'attribut HTML peut commencer par n'importe quoi, à l'exception de quelques caractères comme un guillemet `", une apostrophe ', un slash /, un égal =, un espace...`

Je mentionne les propriétés du DOM, mais peut-être n'est-ce pas clair pour toi. On écrit en général dans des attributs HTML, n'est-ce pas ? Tout à fait, en général c'est ce qu'on fait. Prenons ce simple morceau d'HTML :

```
<input type="text" value="hello">
```

La balise `input` ci-dessus a deux *attributs* : un attribut `type` et un attribut `value`. Quand le navigateur rencontre cette balise, il crée un nœud correspondant dans le DOM (un `HTMLInputElement` si on veut être précis), qui a les *propriétés* correspondantes `type` et `value`. Chaque attribut HTML standard a une propriété correspondante dans un nœud du DOM. Mais un nœud du DOM a aussi d'autres propriétés, qui ne correspondent à aucun attribut HTML. Par exemple : `childElementCount`, `innerHTML` ou `textContent`.

L'interpolation que nous utilisions plus haut pour afficher le nom de l'utilisateur :

```
<p>{{user.name}}</p>
```

est juste du sucre syntaxique pour l'écriture suivante :

```
<p [textContent]="user.name"></p>
```

La syntaxe à base de crochets permet de modifier la propriété `textContent` du DOM, et nous lui donnons la valeur `user.name` qui sera évaluée dans le contexte du composant courant, comme pour l'interpolation.

Note que l'analyseur est sensible à la casse, ce qui veut dire que la propriété doit être écrite avec la bonne casse. `textcontent` ou `TEXTCONTENT` ne fonctionneront pas, il faut écrire `textContent`.

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.

Enfin, les propriétés peuvent avoir des valeurs booléennes, alors que certains attributs n'agissent que par leur simple présence ou absence sur la balise HTML. Par exemple, il existe l'*attribut* `selected` sur la balise `<option>` : quelle que soit la valeur que tu lui donnes, il sélectionnera l'option, dès qu'il y est présent.

```
<option selected>Rainbow Dash</option>
<option selected="false">Rainbow Dash</option> <!-- still selected -->
```

Avec l'accès aux propriétés que nous offre Angular, tu peux écrire :

```
<option [selected]="isPonySelected" value="Rainbow Dash">Rainbow Dash</option>
```

Et le poney sera sélectionné si `isPonySelected` vaut `true`, et ne sera pas sélectionné si elle vaut `false`. Et à chaque fois que la valeur de `isPonySelected` changera, la propriété `selected` sera mise à jour.

Tu peux faire plein de trucs cools avec ça, notamment des trucs qui étaient pénibles en AngularJS 1.x. Par exemple, avoir une URL source dynamique pour une image :

```

```

Utiliser `ng-src` au lieu de `src` solutionnait le problème, parce que le navigateur ignorera cet attribut non standard. Une fois qu'AngularJS avait compilé l'application, il ajoutait l'attribut `src` avec une URL correcte après interpolation, déclenchant ainsi le téléchargement de la bonne image. Cool ! Mais ça avait deux inconvénients :

- D'abord, en tant que développeur, il te fallait deviner quelle valeur donner à `ng-src`. Était-ce '`https://gravatar.com`' ? '`"https://gravatar.com"`' ? '`'pony.avatar.url'`' ? '`'{{pony.avatar.url}}'`' ? Aucun moyen de savoir, sauf à lire la documentation.
- Ensuite, l'équipe Angular devait créer une directive pour chaque attribut standard. Ils l'ont fait, et nous devions tous les connaître...

Mais nous sommes désormais dans un monde où ton HTML peut contenir des *Web Components* externes, genre :

```
<ns-pony name="Rainbow Dash"></ns-pony>
```

Si c'est un *Web Component* que tu veux utiliser, tu n'as aucun moyen simple de lui passer une valeur dynamique avec la plupart des frameworks JS. Sauf à ce que le développeur du *Web Component* ait pris un soin particulier pour le rendre possible. Lis le chapitre sur les *Web Components* pour plus d'informations.

Un *Web Component* doit agir comme un élément du navigateur, sans aucune différence avec un élément natif. Ils ont une API DOM basée sur des propriétés, événements, et méthodes. Avec Angular, tu peux écrire :

```
<ns-pony [name]="pony.name"></ns-pony>
```

Et cela fonctionnera !

Angular synchronisera les valeurs des propriétés et des attributs.

Plus aucune directive spécifique à apprendre ! Si tu veux cacher un élément, tu peux utiliser la propriété standard `hidden` :

```
<div [hidden]="isHidden">Hidden or not</div>
```

Et la `<div>` ne sera cachée que si `isHidden` vaut `true`, car Angular travaillera directement avec la propriété `hidden` du DOM. Plus besoin de `ng-hide`, ni des dizaines d'autres directives qui étaient nécessaires en AngularJS 1.x.

Tu peux aussi accéder à des propriétés comme l'attribut `color` de la propriété `style`.

```
<p [style.color]="foreground">Friendship is Magic</p>
```

Si la valeur de l'attribut `foreground` est modifiée à `green`, le texte deviendra vert aussi !

Ainsi Angular utilise les propriétés. Quelles valeurs pouvons-nous leur donner ? Nous avions déjà vu que l'interpolation `property="{{expression}}"` :

```
<ns-pony name="{{pony.name}}></ns-pony>
```

est la même chose que `[property]="expression"` (que l'on préfère généralement) :

```
<ns-pony [name]="pony.name"></ns-pony>
```

Si tu veux plutôt afficher 'Pony ' suivi du nom du poney, tu as deux options :

```
<ns-pony name="Pony {{pony.name}}></ns-pony>
<ns-pony [name]="'Pony ' + pony.name"></ns-pony>
```

Si ta valeur n'est pas dynamique, tu peux simplement écrire `property="value"` :

```
<ns-pony name="Rainbow Dash"></ns-pony>
```

Toutes ces notations sont équivalentes, et la syntaxe ne dépend pas de la façon dont le développeur a choisi d'écrire son composant, comme c'était le cas en AngularJS 1.x où il fallait savoir si le composant attendait une valeur ou une référence par exemple.

Bien sûr, une expression peut aussi contenir un appel de fonction :

```
<ns-pony name="{{pony.fullName()}}></ns-pony>
<ns-pony [name]="pony.fullName()"></ns-pony>
```

## 8.4. Événements

Si tu développes une application web, tu sais qu'afficher des données n'est qu'une partie du travail : il te faut aussi réagir aux interactions de l'utilisateur.

Pour cela, le navigateur déclenche des événements, que tu peux écouter : `click`, `keyup`, `mousemove`, etc. AngularJS 1.x avait une directive par événement : `ng-click`, `ng-keyup`, `ng-mousemove`, etc. En Angular, c'est plus simple, il n'y a plus de directives spécifiques à se remémorer.

Si on retourne à notre composant `RacesComponent`, nous aimerions maintenant un bouton qui affichera les courses de poneys quand il est cliqué.

Réagir à un événement peut être fait comme suit :

```
<button (click)="onButtonClick()">Click me!</button>
```

Un clic sur le bouton de l'exemple ci-dessus déclenchera un appel à la méthode `onButtonClick()` du composant.

Ajoutons ceci à notre composant :

```
@Component({
  selector: 'ns-races',
  template: `
    <h2>Races</h2>
    <button (click)="refreshRaces()">Refresh the races list</button>
    <p>{{races.length}} races</p>
  `
})
export class RacesComponent {
  races: any = [];

  refreshRaces() {
    this.races = [{ name: 'London' }, { name: 'Lyon' }];
  }
}
```

Si tu testes ça dans notre navigateur, au départ tu devrais voir :

```

<ponyracer-app>
  <h1>PonyRacer</h1>
  <ns-races>
    <h2>Races</h2>
    <button (click)="refreshRaces()">Refresh the races list</button>
    <p>0 races</p>
  </ns-races>
</ponyracer-app>

```

Et après le clic, '0 races' doit devenir '2 races'. Hourra !

L'instruction peut être un appel de fonction, mais ça peut être aussi n'importe quelle instruction exécutable, ou même une séquence d'instructions, comme :

```

<button (click)="firstName = 'Cédric'; lastName = 'Exbrayat'">
  Click to change name to Cédric Exbrayat
</button>

```

Mais bon je ne te conseille pas de faire ça. Utiliser des méthodes est une bien meilleure façon d'encapsuler le comportement. Elles rendront ton code plus facile à tester et à maintenir, et rendront la vue plus simple.

Le truc cool c'est que ça fonctionne avec les événements standards du DOM, mais aussi avec tous les événements spécifiques déclenchés par tes composants Angular, ou par des *Web Components*. On verra plus tard comment déclencher des événements spécifiques.

Pour le moment, disons que le composant `RacesComponent` déclenche un événement spécifique pour indiquer la disponibilité d'une nouvelle course de poneys.

Notre template dans le composant `PonyRacerAppComponent` ressemblera alors à :

```

@Component({
  selector: 'ponyracer-app',
  template: `
    <h1>PonyRacer</h1>
    <ns-races (newRaceAvailable)="onNewRace()"></ns-races>
  `
})
export class PonyRacerAppComponent {
  onNewRace() {
    // add a flashy message for the user.
  }
}

```

Sans trop d'effort, on peut imaginer que le composant `<ns-races>` peut déclencher un événement spécifique `newRaceAvailable`, et que quand cet événement est déclenché, la méthode `onNewRace()` de notre `PonyRacerAppComponent` sera appelée.

Angular écoute les événements de l'élément et ceux de ses enfants, il va donc aussi réagir sur les événements "bouillonnants" ("bubbling up", i.e. les événements qui se propagent vers le haut depuis le fond des composants enfants). Considérons le template suivant :

```
<div (click)="onButtonClick()">
  <button>Click me!</button>
</div>
```

Même si l'utilisateur clique sur le bouton dans la div, la méthode `onButtonClick()` sera appelée, car l'événement se propage vers le haut.

Et tu peux accéder à l'événement en question depuis la méthode appelée ! Pour cela, tu dois simplement passer `$event` à ta méthode :

```
<div (click)="onButtonClick($event)">
  <button>Click me!</button>
</div>
```

Et ensuite tu peux gérer cet événement dans la classe du composant :

```
onButtonClick(event) {
  console.log(event);
}
```

Par défaut, l'événement va continuer à "bouillonner", déclenchant potentiellement d'autres *handlers* plus haut dans la hiérarchie de composants.

Tu peux agir sur l'événement pour empêcher ce comportement par défaut, et/ou annuler la propagation si tu le souhaites :

```
onButtonClick(event) {
  event.preventDefault();
  event.stopPropagation();
}
```

Une autre fonctionnalité sympa réside dans la gestion des événements du clavier:

```
<textarea (keydown.space)="onSpacePress()">Press space!</textarea>
```

Chaque fois que tu appuies sur la touche `space`, la méthode `onSpacePress()` sera appelée. Et on peut faire des combos, comme (`keydown.alt.space`), etc.

Pour conclure cette partie, je voudrais t'indiquer une différence fondamentale entre :

```
<component [property]="doSomething()"></component>
```

et

```
<component (event)="doSomething()"></component>
```

Dans le premier cas de *binding* de propriété, la valeur `doSomething()` est une expression, et sera évaluée à chaque cycle de détection de changement pour déterminer si la propriété doit être modifiée.

Dans le second cas de *binding* d'événement, la valeur `doSomething()` est une instruction (*statement*), et ne sera évaluée **que lorsque l'événement est déclenché**.

Par définition, ces deux *bindings* ont des objectifs complètement différents, et comme tu t'en doutes, des restrictions d'usage différentes.

## 8.5. Expressions vs instructions

Les expressions et les instructions (*statements*) présentent des différences.

Une expression sera évaluée plusieurs fois, par le mécanisme de détection de changement. Elle doit ainsi être la plus performante possible. Pour faire simple, une expression Angular est une version simplifiée d'une expression JavaScript.

Si tu utilises `user.name` comme expression, `user` doit être défini, sinon Angular va lever une erreur.

Une expression doit être unique : tu ne peux pas en chaîner plusieurs séparées par des points-virgules.

Une expression ne doit pas avoir d'effets de bord. Par exemple, une affectation est interdite.

```
<!-- forbidden, as the expression is an assignment -->
<!-- this will throw an error -->
<component [property]="user = 'Cédric'"></component>
```

Elle ne doit pas contenir de mot-clés comme `if`, `var`, etc.

De son côté, une instruction est déclenchée par l'événement correspondant. Si tu essayes d'utiliser une instruction comme `race.show()` où `race` serait `undefined`, tu auras une erreur. Tu peux chaîner plusieurs instructions, séparées par un point-virgule. Une instruction peut avoir des effets de bord, et doit généralement en avoir : c'est l'effet voulu quand on réagit à un événement, on veut que des choses se produisent. Une instruction peut contenir des affectations de variables, et peut contenir des mot-clés.

## 8.6. Variables locales

Quand j'explique qu'Angular va regarder dans l'instance du composant pour trouver une variable, ce n'est pas tout à fait exact. En fait, il va regarder dans l'instance du composant et dans les variables locales. Les variables locales sont des variables que tu peux déclarer dynamiquement dans ton template avec la notation `#`.

Supposons que tu veuilles afficher la valeur d'un input :

```
<input type="text" #name>
{{ name.value }}
```

Avec la notation `#`, on crée une variable locale `name` qui référence l'objet `HTMLInputElement` du DOM. Cette variable locale peut être utilisée n'importe où dans le template. Comme `HTMLInputElement` a une propriété `value`, on peut l'afficher dans une expression interpolée. Je reviendrai sur cet exemple plus tard.

Un autre cas d'usage des variables locales est l'exécution d'une action sur un autre élément.

Par exemple, tu peux vouloir donner le focus à un élément quand on clique sur un bouton. C'était un peu pénible à réaliser en AngularJS 1.x, il te fallait créer une directive et tout le tralala.

La méthode `focus()` est standard dans l'API DOM, et on peut maintenant en bénéficier. Avec une variable locale en Angular, c'est super simple :

```
<input type="text" #name>
<button (click)="name.focus()">Focus the input</button>
```

Ça peut aussi être utilisé avec un composant spécifique, créé dans notre application, ou importé d'un autre projet, ou même un véritable *Web Component* :

```
<google-youtube #player></google-youtube>
<button (click)="player.play()">Play!</button>
```

Ici le bouton peut lancer la lecture de la vidéo sur le composant `<google-youtube>`. C'est bien un véritable *Web Component* écrit en `Polymer` ! Ce composant a une méthode `play()` qu'Angular peut appeler quand on clique sur le bouton : la classe !

Les variables locales ont quelques cas d'utilisations spécifiques, et on va progressivement les découvrir. L'un d'entre eux est expliqué dans la section juste ci-dessous.

## 8.7. Directives de structure

Pour le moment, notre `RacesComponent` n'affiche toujours aucune course de poneys :) La façon "canonique" en Angular serait de créer un autre composant `RaceComponent` pour afficher chaque

course. On va faire quelque chose de plus simple, et utiliser une pauvre liste <ul><li>.

Le *binding* de propriété et d'événement est formidable, mais il ne nous permet pas de modifier la structure du DOM, comme pour itérer sur une collection et ajouter un nœud par élément. Pour cela, nous avons besoin de **directives structurelles**. Dans Angular, une directive est assez proche d'un composant, mais n'a pas de template. On les utilise pour ajouter un comportement à un élément.

Les directives structurelles fournies par Angular s'appuient sur l'élément <ng-template>, inspirée de la balise standard **template** de la **specification** HTML. Cet élément s'appelait même **template** avant la version 4.0, mais est maintenant déprécié au profit de **ng-template** :

```
<ng-template>
  <div>Races list</div>
</ng-template>
```

Ici nous avons défini un template, affichant une simple <div>. Seul, il n'est pas très utile, car le navigateur ne va pas l'afficher. Mais si nous ajoutons un élément **template** dans l'une de nos vues, alors Angular pourra utiliser son contenu. Les directives structurelles ont justement la capacité d'utiliser ce contenu, pour l'afficher ou non, le répéter, etc.

Examinons ces fameuses directives !

### 8.7.1. NgIf

Si nous voulons instancier le template seulement lorsqu'une condition est réalisée, alors nous utiliserons la directive **ngIf** :

```
<ng-template [ngIf]="races.length > 0">
  <div><h2>Races</h2></div>
</ng-template>
```

Le framework propose quelques directives, comme **ngIf**. Elles viennent du module que nous avons importé un peu plus tôt : **BrowserModule**. Si besoin, tu peux aussi définir tes propres directives ; on y reviendra.

Ici, le template ne sera instancié que si **races** a au moins un élément, donc s'il existe des courses de poneys. Comme cette syntaxe est un peu longue, il y a une version raccourcie :

```
<div *ngIf="races.length > 0"><h2>Races</h2></div>
```

Et tu utiliseras toujours cette version courte.

La notation utilise **\*** pour montrer que c'est une instantiation de template. La directive **ngIf** va maintenant prendre en charge l'affichage ou non de la **div** à chaque fois que la valeur de **races** va changer : s'il n'y a plus de course, la **div** va disparaître.

Les directives fournies par le framework sont déjà pré-chargées pour nous, nous n'avons donc pas

besoin d'importer et de déclarer `NgIf` dans l'attribut `directives` du décorateur `@Component`.

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

@Component({
  selector: 'ns-races',
  template: '<div *ngIf="races.length > 0"><h2>Races</h2></div>'
})
export class RacesComponent {
  races: Array<any> = [];
}
```

Il existe aussi une possibilité d'utiliser `else` depuis la version 4.0 :

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

@Component({
  selector: 'ns-races',
  template: `
    <div *ngIf="races.length > 0; else empty"><h2>Races</h2></div>
    <ng-template #empty><h2>No races.</h2></ng-template>
  `
})
export class RacesComponent {
  races: Array<any> = [];
}
```

## 8.7.2. NgFor

Travailler avec de vraies données te conduira indubitablement à afficher une liste d'éléments. `ngFor` est alors très utile : elle permet d'instancier un template par élément d'une collection. Notre composant `RacesComponent` possède un attribut `races`, qui est, comme tu peux le deviner, un tableau des courses de poneys disponibles.

```

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

@Component({
  selector: 'ns-races',
  template: '<div *ngIf="races.length > 0">
    <h2>Races</h2>
    <ul>
      <li *ngFor="let race of races">{{race.name}}</li>
    </ul>
  </div>'
})
export class RacesComponent {
  races: Array<any> = [{ name: 'London' }, { name: 'Lyon' }];
}

```

Et ainsi nous avons une magnifique liste, avec une balise `li` par élément de notre collection !

```

<ul>
  <li>London</li>
  <li>Lyon</li>
</ul>

```

Note que `NgFor` utilise une syntaxe particulière, dite "microsyntax" (*microsyntax*).

```

<ul>
  <li *ngFor="let race of races">{{race.name}}</li>
</ul>

```

C'est l'équivalent du plus verbeux (et que l'on n'utilisera jamais) :

```

<ul>
  <ng-template ngFor let-race [ngForOf]="races">
    <li>{{race.name}}</li>
  </ng-template>
</ul>

```

Tu peux reconnaître ici :

- L'élément `template` pour déclarer un template *inline*,
- La directive `NgFor` qui lui est appliquée,
- La propriété `NgForOf` où nous fournissons la collection à parcourir,
- La variable `race` à utiliser dans les expressions interpolées, reflétant l'élément courant.

Au lieu de se rappeler de tout ça, c'est plus simple d'utiliser la notation raccourcie :

```
<ul>
  <li *ngFor="let race of races">{{race.name}}</li>
</ul>
```

C'est aussi possible de déclarer une autre variable locale, associée à l'indice de l'élément courant dans la collection :

```
<ul>
  <li *ngFor="let race of races; index as i">{{i}} - {{race.name}}</li>
</ul>
```

La variable locale **i** recevra l'indice de l'élément courant, commençant à zéro.

**index** est une **variable exportée**. Certaines directives exportent des variables que l'on peut affecter à une variable locale afin de pouvoir les utiliser dans notre template :

```
<ul>
  <li>0 - London</li>
  <li>1 - Lyon</li>
</ul>
```

Il y aussi quelques autres variables exportées qui peuvent être utiles :

- **even**, un booléen qui sera vrai si l'élément a un index pair
- **odd**, un booléen qui sera vrai si l'élément a un index impair
- **first**, un booléen qui sera vrai si l'élément est le premier de la collection
- **last**, un booléen qui sera vrai si l'élément est le dernier de la collection

### 8.7.3. NgSwitch

Comme tu peux le deviner par son nom, celle-ci permet de switcher entre plusieurs templates selon une condition.

```
<div [ngSwitch]="messageCount">
  <p *ngSwitchCase="0">You have no message</p>
  <p *ngSwitchCase="1">You have a message</p>
  <p *ngSwitchDefault>You have some messages</p>
</div>
```

Comme tu peux le voir, **ngSwitch** prend une condition et les **\*ngSwitchCase** attendent les différents cas possibles. Tu as aussi **\*ngSwitchDefault**, qui sera affiché si aucune des autres possibilités n'est remplie.

## 8.8. Autres directives de templating

Deux autres directives peuvent être utiles pour écrire un template, mais ce ne sont pas des directives structurelles comme celles que nous venons de voir. Ces directives sont des directives standards.

### 8.8.1. NgStyle

La première est `ngStyle`. Nous avons déjà vu que nous pouvions agir sur le style d'un élément en utilisant :

```
<p [style.color]="foreground">Friendship is Magic</p>
```

Si tu veux changer plusieurs styles en même temps, tu peux utiliser la directive `ngStyle` :

```
<div [ngStyle]="{fontWeight: fontWeight, color: color}">I've got style</div>
```

Note que la directive attend un objet dont les clés sont les styles à définir. La clé peut être en **camelCase** (`fontWeight`) ou en **dash-case** ('`font-weight`').

### 8.8.2. NgClass

Dans le même esprit, la directive `ngClass` permet d'ajouter ou d'enlever dynamiquement des classes sur un élément.

Comme pour le style, on peut soit définir une seule classe avec le binding de propriété :

```
<div [class.awesome-div]="isAwesomeDiv()">I've got style</div>
```

Ou, si on veut en définir plusieurs en même temps, utiliser `ngClass` :

```
<div [ngClass]="{'awesome-div': isAwesomeDiv(), 'colored-div': isColoredDiv()}">I've got style</div>
```

## 8.9. Syntaxe canonique

Chacune des syntaxes indiquées a une version plus verbeuse appelée la syntaxe canonique (*canonical syntax*). C'est essentiellement intéressant si ton moteur de template côté serveur a du mal avec les notations `[]` ou `()`. Ou si tu ne supportes vraiment pas d'utiliser `[ ]`, `( )`, `* ...`.

Si tu veux déclarer un *binding* de propriété, tu peux écrire :

```
<ns-pony [name]="pony.name"></ns-pony>
```

ou utiliser la notation canonique :

```
<ns-pony bind-name="pony.name"></ns-pony>
```

Pour le *binding* d'événement, tu peux écrire :

```
<button (click)="onButtonClick()">Click me!</button>
```

ou utiliser la syntaxe canonique :

```
<button on-click="onButtonClick()">Click me!</button>
```

Et pour les variables locales, tu utiliseras `ref-` :

```
<input type="text" ref-name>
<button on-click="name.focus()">Focus the input</button>
```

au lieu de la forme raccourcie :

```
<input type="text" #name>
<button (click)="name.focus()">Focus the input</button>
```

## 8.10. Résumé

Le système de template d'Angular nous propose une syntaxe puissante pour exprimer les parties dynamiques de notre HTML. Elle nous permet d'exprimer du *binding* de données, de propriétés, d'événements, et des considérations de templating, d'une façon claire, avec des symboles propres :

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

Elle permet d'interagir avec le standard des *Web Components* comme aucun autre framework. Et comme il n'y a pas d'ambiguité sémantique entre les symboles, on verra nos outils et IDEs s'améliorer progressivement pour nous aider et nous alerter dans nos écritures de templates.

Tous ces symboles sont les versions raccourcies de leur pendant canonique, que tu peux utiliser si tu le souhaites.

Tu auras certainement besoin de temps pour utiliser intuitivement cette syntaxe, mais tu gagneras

rapidement en dextérité, et elle deviendra ensuite facile à lire et à écrire.

Avant de passer à la suite, essayons de voir un exemple complet.

Je veux écrire un composant **PoniesComponent**, affichant une liste de poneys. Chacun de ces poneys devrait être représenté par son propre composant **PonyComponent**, mais nous n'avons pas encore vu comment passer des paramètres à un composant. Donc, pour l'instant, nous allons nous contenter d'afficher une simple liste. La liste devra s'afficher seulement si elle n'est pas vide, et j'aimerais avoir un peu de couleur sur les lignes paires. Et nous voulons pouvoir rafraîchir cette liste d'un simple clic sur un bouton.

Prêt ?

On commence par écrire notre composant, dans son propre fichier :

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

@Component({
  selector: 'ns-ponies',
  template: ``
})
export class PoniesComponent {

}
```

Tu peux l'ajouter au composant **PonyRacerAppComponent** écrit au chapitre précédent pour le tester. Tu devras l'importer, l'ajouter aux directives, et insérer la balise `<ns-ponies></ns-ponies>` dans le template.

Notre nouveau composant a une liste de poneys :

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

@Component({
  selector: 'ns-ponies',
  template: ``
})
export class PoniesComponent {
  ponies: Array<any> = [{ name: 'Rainbow Dash' }, { name: 'Pinkie Pie' }];
}
```

On va afficher cette liste, en utilisant **ngFor** :

```

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

@Component({
  selector: 'ns-ponies',
  template: `<ul>
    <li *ngFor="let pony of ponies">{{pony.name}}</li>
  </ul>`
})
export class PoniesComponent {
  ponies: Array<any> = [{ name: 'Rainbow Dash' }, { name: 'Pinkie Pie' }];
}

```

Il manque un dernier truc, le bouton qui rafraîchit la liste :

```

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

@Component({
  selector: 'ns-ponies',
  template: `<button (click)="refreshPonies()">Refresh</button>
<ul>
  <li *ngFor="let pony of ponies">{{pony.name}}</li>
</ul>`
})
export class PoniesComponent {
  ponies: Array<any> = [{ name: 'Rainbow Dash' }, { name: 'Pinkie Pie' }];

  refreshPonies() {
    this.ponies = [{ name: 'Fluttershy' }, { name: 'Rarity' }];
  }
}

```

Et bien sûr, une touche de couleur pour finir, avec l'utilisation de `[style.color]` et de la variable locale `isEven` :

```

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

@Component({
  selector: 'ns-ponies',
  template: `<button (click)="refreshPonies()">Refresh</button>
<ul>
  <li *ngFor="let pony of ponies; even as isEven"
      [style.color]="isEven ? 'green' : 'black'>
    {{pony.name}}
  </li>
</ul>`
})
export class PoniesComponent {
  ponies: Array<any> = [{ name: 'Rainbow Dash' }, { name: 'Pinkie Pie' }];

  refreshPonies() {
    this.ponies = [{ name: 'Fluttershy' }, { name: 'Rarity' }];
  }
}

```

Et voilà, on a mis en œuvre toute la syntaxe de templating, et on a un composant parfaitement fonctionnel. Nos données sont pour le moment hardcodées, mais on apprendra vite comment utiliser un service pour les charger ! Cela implique auparavant d'en apprendre un peu plus sur l'injection de dépendances, pour que nous puissions utiliser le service HTTP !

## MISE EN PRATIQUE

Essaye nos deux exercices [Templates 🐾](#) et [Liste de courses 🐾](#) ! Ils sont gratuits et font partie de notre Pack pro, où tu apprends à construire une application complète étape par étape. Le premier consiste à construire un petit composant, un menu "responsive", et jouer avec son template. Le second te guide pour construire un autre composant : la liste des courses.

# Chapitre 9. Injection de dépendances

## 9.1. Cuisine et 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 fournit au composant qui en a besoin. Cette façon de procéder se nomme l'inversion de contrôle.

Cela apporte plusieurs bénéfices :

- le développement est simplifié, on exprime juste ce que l'on veut, où on le veut.
- le test est simplifié, en permettant de remplacer les dépendances par des versions bouchonnées.
- la configuration est simplifiée, en permutant facilement différentes implémentations.

C'est un concept largement utilisé côté serveur, mais AngularJS 1.x était un des premiers à l'apporter côté client.

## 9.2. Développement facile

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, récupérera 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.

Prenons un exemple avec un service `ApiService`, déjà écrit par l'un de nos collègues. Comme c'est le feignant de l'équipe, il a seulement écrit une classe vide avec une méthode `get` qui renvoie un tableau vide, mais tu peux déjà deviner qu'à terme ce service sera utilisé pour communiquer avec l'API exposée par le backend.

```
export class ApiService {  
  get(path) {  
    // todo: call the backend API  
  }  
}
```

Avec TypeScript, c'est ultra-simple de déclarer une dépendance dans un de nos composants ou services, il suffit d'utiliser le système de type.

Disons que l'on veut écrire un `RaceService` qui utilise  `ApiService` :

```
import { ApiService } from './api.service';

export class RaceService {

  constructor(private apiService: ApiService) {
  }

}
```

Angular récupérera le service  `ApiService` pour nous, et l'injectera dans notre constructeur. Quand le service `RaceService` est utilisé, le constructeur sera appelé, et le champ `apiService` référencera le service  `ApiService`.

Maintenant, on peut ajouter une méthode `list()` à notre service, qui appellera notre serveur via le service  `ApiService` :

```
import { ApiService } from './api.service';

export class RaceService {

  constructor(private apiService: ApiService) {
  }

  list() {
    return this.apiService.get('/races');
  }

}
```

Pour indiquer à Angular que ce service a lui-même des dépendances, on doit lui ajouter un décorateur `@Injectable()` :

```

import { Injectable } from '@angular/core';
import { ApiService } from './api.service';

@Injectable()
export class RaceService {

  constructor(private apiService: ApiService) {
  }

  list() {
    return this.apiService.get('/races');
  }
}

```

Comme nous utilisons `ApiService`, nous avons besoin de "l'enregistrer", pour le rendre disponible à l'injection.

Une façon simple est d'utiliser l'attribut `providers` du décorateur `@NgModule` vu précédemment.

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { PonyRacerAppComponent } from './app.component';
import { ApiService } from './services/api.service';

@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent],
  providers: [
    ApiService
  ],
  bootstrap: [PonyRacerAppComponent]
})
export class AppModule {
}

```

Maintenant, si on veut rendre notre `RaceService` disponible à l'injection dans d'autres services et composants, il nous faut aussi l'enregistrer :

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { PonyRacerAppComponent } from './app.component';
import { RaceService } from './services/race.service';
import { ApiService } from './services/api.service';

@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent],
  providers: [
    RaceService,
    ApiService
  ],
  bootstrap: [PonyRacerAppComponent]
})
export class AppModule {
}

```

Et c'est fini !

On peut désormais utiliser notre nouveau service où on le souhaite. Testons-le dans notre composant `PonyRacerAppComponent` :

```

import { Component } from '@angular/core';
import { RaceService } from './services/race.service';

@Component({
  selector: 'ponyracer-app',
  template: '<h1>PonyRacer</h1>
  <p>{{list()}}</p>'
})
export class PonyRacerAppComponent {

  // add a constructor with RaceService
  constructor(private raceService: RaceService) {}

  list() {
    return this.raceService.list();
  }

}

```

Comme notre collègue a simplement renvoyé un tableau vide dans la méthode `get` de `ApiService` tu ne vas rien recevoir si tu appelles la méthode `list()`.

Mais on peut peut-être y faire quelque chose...

## 9.3. Configuration facile

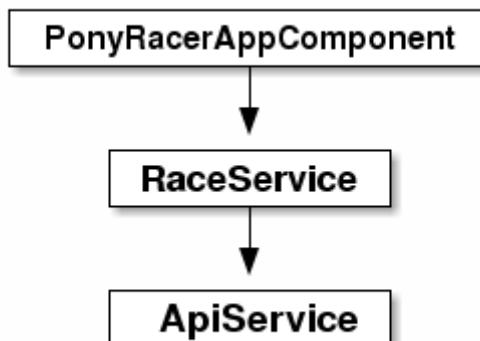
Je reviendrai sur les bénéfices de testabilité apportés par l'injection de dépendances dans un chapitre ultérieur, on va se concentrer sur les questions de configuration pour le moment. Ici, on appelait un serveur qui n'existe pas. Soit l'équipe backend n'est pas prête, soit tu veux le faire plus tard. Dans tous les cas, on voudrait bouchonner tout ça.

L'injection de dépendances permet de le faire facilement. Retournons à l'enregistrement de nos dépendances :

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { PonyRacerAppComponent } from './app.component';
import { RaceService } from './services/race.service';
import { ApiService } from './services/api.service';

@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent],
  providers: [
    RaceService,
    ApiService
  ],
  bootstrap: [PonyRacerAppComponent]
})
export class AppModule { }
```

On peut représenter les relations entre composants et services comme ceci, si une flèche signifie "dépend de" :



En fait, ce que l'on a écrit est la version courte de :

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { PonyRacerAppComponent } from './app.component';
import { RaceService } from './services/race.service';
import { ApiService } from './services/api.service';

@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent],
  bootstrap: [PonyRacerAppComponent],
  providers: [
    { provide: RaceService, useClass: RaceService },
    { provide: ApiService, useClass: ApiService }
  ]
})
export class AppModule {
}

```

On explique à l'injecteur qu'on veut un binding entre un token (le type `RaceService`) et la classe `RaceService`. L'injecteur (`Injector`) est un service qui maintient un registre des composants injectables, et qui les injecte quand ils sont réclamés. Le registre est un dictionnaire associant des clés, appelés tokens, à des classes. Les tokens ne sont pas, contrairement à de nombreux frameworks d'injection de dépendances, limités à des seules chaînes de caractères. Ils peuvent être de n'importe quel type, comme par exemple une référence à une classe. Et ce sera en général le cas.

Comme dans notre exemple, si le token et la classe de l'objet à injecter sont les mêmes, tu peux écrire la même chose en version raccourcie :

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { PonyRacerAppComponent } from './app.component';
import { RaceService } from './services/race.service';
import { ApiService } from './services/api.service';

@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent],
  providers: [
    RaceService,
    ApiService
  ],
  bootstrap: [PonyRacerAppComponent]
})
export class AppModule {
}

```

Le token doit identifier de manière unique la dépendance.

L'injecteur est retourné par la promesse `bootstrapModule`, alors on peut jouer un peu avec :

```
// in our module
providers: [
  ApiService,
  { provide: RaceService, useClass: RaceService },
  // let's add another provider to the same class
  // with another token
  { provide: 'RaceServiceToken', useClass: RaceService }
]

// let's bootstrap the module
platformBrowserDynamic().bootstrapModule(AppModule)
  .then(
    // and play with the returned injector
    appRef => playWithInjector(appRef.injector)
  );
```

La partie intéressante est la fonction `playWithInjector` :

```
function playWithInjector(inj) {
  console.log(inj.get(RaceService));
  // logs "RaceService {apiService: ApiService}"
  console.log(inj.get('RaceServiceToken'));
  // logs "RaceService {apiService: ApiService}" again
  console.log(inj.get(RaceService) === inj.get(RaceService));
  // logs "true", as the same instance is returned every time for a token
  console.log(inj.get(RaceService) === inj.get('RaceServiceToken'));
  // logs "false", as the providers are different,
  // so there are two distinct instances
}
```

On demande ici une dépendance à l'injecteur via la méthode `get` et un token. Comme j'ai déclaré le `RaceService` deux fois, avec deux tokens différents, on a deux providers. L'injecteur va créer une instance de `RaceService` à la première demande d'un token donné, et retournera la même instance lors de chaque demande ultérieure. Il fera de même pour chaque provider, alors nous aurons en fait dans ce cas deux instances de `RaceService` dans notre application, une pour chaque token.

Cependant, tu ne manipuleras pas souvent le token, voire pas du tout. En TypeScript, tu compteras sur les types pour faire le boulot, où le token sera une référence de type, en général associé à la classe correspondante. Si tu veux spécifiquement définir un autre token, tu devras utiliser le décorateur `@Inject()` : on y reviendra à la fin de ce chapitre.

Cet exemple ne servait qu'à pointer du doigt quelques détails :

- un provider associe un token à un service.

- l'injecteur retourne la même instance chaque fois que le même token est demandé.
- on peut définir un token différent de la classe de l'objet à injecter.

Le fait de créer une instance lors du premier appel, et de retourner ensuite toujours la même, est un *design pattern* bien connu : un *singleton*. C'est très utile, parce que tu peux y stocker des informations, et les partager entre composants via un service, parce qu'ils utiliseront la même instance de ce service.

Maintenant, retournons à notre problème de `RaceService` bouchonné. Je peux écrire une nouvelle classe, faisant le même boulot que `RaceService`, mais qui retourne des données codées en dur :

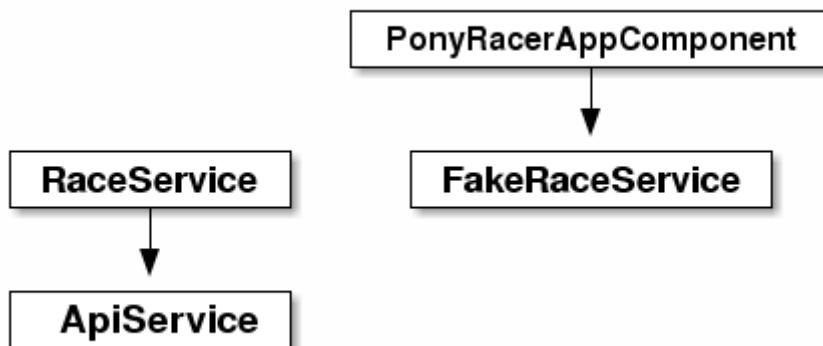
```
class FakeRaceService {
  list() {
    return [{ name: 'London' }];
  }
}
```

On peut manipuler la déclaration du provider pour remplacer `RaceService` par notre `FakeRaceService` :

```
// in our module
providers: [
  // we provide a fake service
  { provide: RaceService, useClass: FakeRaceService }
]
```

Si tu redémarres ton application, tu constateras que nous avons bien cette fois-ci une course disponible, car notre application utilise le service bouchonné au lieu du premier !

Maintenant nos relations de dépendances sont devenues :



Cela peut être très utile quand tu testes manuellement ton application, et aussi, comme nous le verrons prochainement, quand tu écris des tests automatisés.

## 9.4. Autres types de provider

Toujours dans le cadre de cet exemple, nous aurions probablement envie d'utiliser `FakeRaceService` pendant le développement, et `RaceService` en production. Tu peux évidemment changer le provider manuellement, mais tu peux aussi utiliser un autre type de provider : `useFactory`.

```
// we just have to change this constant when going to prod
const IS_PROD = false;

// in our module
providers: [
  // we provide a factory
  {
    provide: RaceService,
    useFactory: () => IS_PROD ? new RaceService(null) : new FakeRaceService()
  }
]
```

Ici, on utilise `useFactory` au lieu de `useClass`. Une *factory* est une fonction avec un seul job, créer une instance. Notre exemple teste une constante, et retourne le service bouchonné ou le véritable en fonction.

Mais attends, si nous basculons sur l'utilisation du véritable service, comme nous utilisons un simple `new` pour créer l'instance de `RaceService`, il n'aura pas sa dépendance `ApiService` instanciée ! Et oui, si on veut que cet exemple fonctionne, on doit passer au constructeur une instance de `ApiService`. Bonne nouvelle: `useFactory` prend un paramètre supplémentaire, un tableau, où tu peux spécifier ses dépendances :

```
// we just have to change this constant when going to prod
const IS_PROD = true;

// in our module
providers: [
  ApiService,
  // we provide a factory
  {
    provide: RaceService,
    // the apiService instance will be injected in the factory
    // so we can pass it to RaceService
    useFactory: apiService => IS_PROD ? new RaceService(apiService) : new
    FakeRaceService(),
    deps: [ApiService]
  }
]
```

Hourra !

**NOTE**

Sois prudent : l'ordre des paramètres doit correspondre à l'ordre dans le tableau, si tu as plusieurs dépendances !

Mais bon, cet exemple permettait juste de démontrer l'utilisation de `useFactory`. En vrai, tu peux, tu DOIS même, écrire :

```
// in our module
providers: [
  ApiService,
  { provide: RaceService, useClass: IS_PROD ? RaceService : FakeRaceService }
]
```

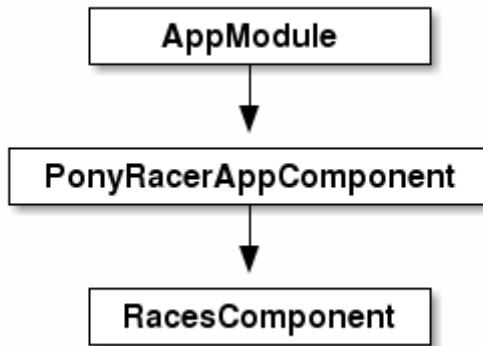
Déclarer une constante `IS_PROD` est un peu pénible : et si nous utilisions aussi ici l'injection de dépendances ?! Oui, je pousse le bouchon un peu loin ! :) Tu n'as certes pas besoin de tout faire rentrer dans le moule de l'injection, mais cela permet de te montrer un autre type de provider : `useValue`.

```
// in our module
providers: [
  ApiService,
  // we provide a factory
  { provide: 'IS_PROD', useValue: true },
  {
    provide: RaceService,
    useFactory: (IS_PROD, apiService) => IS_PROD ? new RaceService(apiService) : new
FakeRaceService(),
    deps: ['IS_PROD', ApiService]
  }
]
```

## 9.5. Injecteurs hiérarchiques

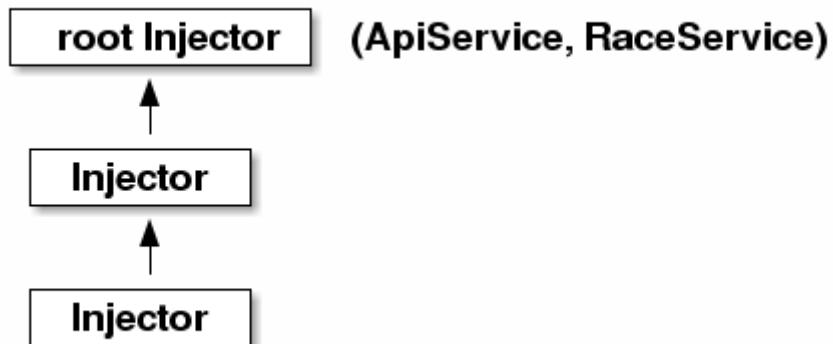
Un dernier point crucial à comprendre en Angular : il y a plusieurs injecteurs dans ton application. En fait, il y a un injecteur par composant, et chaque injecteur hérite de l'injecteur de son parent.

Disons qu'on a une application qui ressemble à :



On a un module racine `AppModule` avec un composant racine `PonyRacerAppComponent`, avec un composant enfant `RacesComponent`.

Quand on bootstrappe l'application, on crée un injecteur racine pour le module. Ensuite, chaque composant va créer son propre injecteur, héritant de celui de son parent.



Cela signifie que si tu déclares une dépendance dans un composant, Angular va commencer sa recherche dans l'injecteur courant. S'il y trouve la dépendance, parfait, il la retourne. Sinon, il va chercher dans l'injecteur parent, puis son parent, et tous ses ancêtres, jusqu'à ce qu'il trouve cette dépendance. S'il ne trouve pas, il lève une exception.

De cela, on peut déduire deux affirmations :

- les providers déclarés dans l'injecteur racine sont disponibles pour tous les composants de l'application. Par exemple,  `ApiService`  et  `RaceService`  peuvent être utilisés partout.
- on peut déclarer des dépendances à d'autres niveaux que le module racine. Comment fait-on cela ?

Le décorateur `@Component` peut recevoir une autre option de configuration, appelée `providers`. Cet attribut `providers` peut recevoir un tableau avec une liste de dépendances, comme nous l'avons fait avec l'attribut `providers` de `@NgModule`.

On peut donc imaginer un `RacesComponent` qui déclarerait son propre provider de `RaceService` :

```

@Component({
  selector: 'ns-races',
  providers: [{ provide: RaceService, useClass: FakeRaceService }],
  template: '<strong>Races list: {{list()}}</strong>'
})
export class RacesComponent {

  constructor(private raceService: RaceService) {}

  list() {
    return this.raceService.list();
  }
}

```

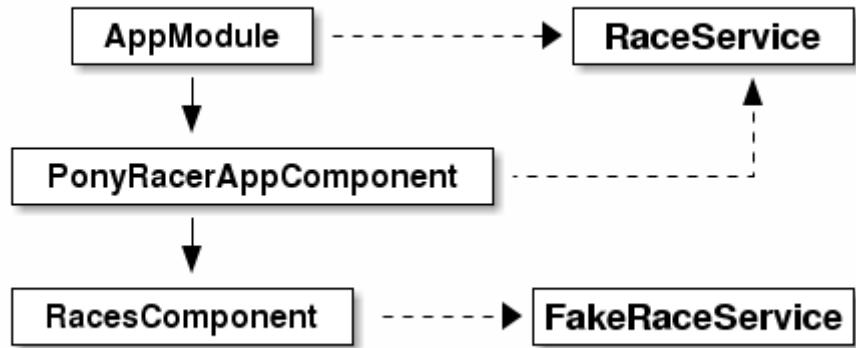
Dans ce composant, le provider avec le token `RaceService` retournera toujours une instance de `FakeRaceService`, quelque soit le provider défini dans l'injecteur racine. C'est vraiment pratique si tu veux utiliser une instance différente d'un service dans un composant donné, ou si tu tiens à avoir des composants parfaitement encapsulés qui déclarent tout ce dont ils dépendent.

**ATTENTION** Si tu déclares une dépendance à la fois dans le module de ton application et dans l'attribut `providers` de ton composant, ce seront bien deux instances différentes qui seront créées et utilisées !

Ici nous avons :



L'injection sera alors résolue en :



En règle générale, on peut dire que si un seul composant a besoin d'un service donné, c'est une bonne idée de définir ce service dans l'injecteur du composant, via l'attribut `providers`. Si une dépendance peut être utilisée ailleurs dans l'application, on la déclare dans le module racine.

## 9.6. Injection sans types

Si tu n'aimes pas TypeScript, et ne veux pas utiliser de types, tu auras besoin d'un décorateur sur chaque dépendance que tu veux injecter. Le même `RaceService` s'écrirait en ES6 avec des décorateurs :

```

import { Injectable, Inject } from '@angular/core';
import { ApiService } from './api.service';

@Injectable()
export class RaceService {

  constructor(@Inject(ApiService) apiService) {
    this.apiService = apiService;
  }

  list() {
    return this.apiService.get('/races');
  }
}

```

Pour que les décorateurs fonctionnent avec cet exemple sans TypeScript, il est nécessaire d'utiliser babel avec le preset `es2015` et les plugins `angular2-annotations` et `syntax-decorators`.

# Chapitre 10. Services

Angular propose le concept de services : des classes que tu peux injecter dans une autre.

Quelques services sont fournis par le framework, certains par les modules communs, et tu peux en construire d'autres. On verra ceux fournis par les modules communs dans un chapitre dédié, alors jetons d'abord un œil à ceux fournis par le framework, et découvrons comment on peut créer les nôtres.

## 10.1. Service Title

Le cœur du framework fournit vraiment peu de services, et encore moins que tu utiliseras réellement dans tes applications. Je dis peu, mais en fait, ils ne sont même que deux :).

Une question qui revient souvent est comment modifier le titre de ma page ? Facile ! Il y a un service **Title** que tu peux t'injecter, et il propose un getter et un setter :

```
import { Component } from '@angular/core';
import { Title } from '@angular/platform-browser';

@Component({
  selector: 'ponyracer-app',
  template: '<h1>PonyRacer</h1>'
})
export class PonyRacerAppComponent {

  constructor(title: Title) {
    title.setTitle('PonyRacer - Bet on ponies');
  }
}
```

Le service créera automatiquement l'élément **title** dans la section **head** si besoin et positionnera correctement la valeur !

## 10.2. Service Meta

L'autre service est sensiblement équivalent : il permet de récupérer et mettre à jour les valeurs "meta" de la page.

```

import { Component } from '@angular/core';
import { Meta } from '@angular/platform-browser';

@Component({
  selector: 'ponyracer-app',
  template: '<h1>PonyRacer</h1>'
})
export class PonyRacerAppComponent {

  constructor(meta: Meta) {
    meta.addTag({ name: 'author', content: 'Ninja Squad' });
  }
}

```

## 10.3. Créer son propre service

C'est vraiment simple. Il suffit d'écrire une classe, et voilà !

```

export class RacesService {

  list() {
    return [{ name: 'London' }];
  }
}

```

Comme en AngularJS 1.x, un service est un singleton, donc la même instance unique sera injectée partout. Cela en fait le candidat idéal pour partager un état parmi plusieurs composants séparés !

Si le service a lui aussi des dépendances, alors il faut lui ajouter le décorateur `@Injectable()`.

Sans ce décorateur, le framework ne pourra pas faire l'injection de dépendances.

Plutôt que renvoyer la même liste à chaque fois, notre `RacesService` va probablement interroger une API REST. Pour faire une requête HTTP, le framework nous donne le service `Http`. Ne t'inquiète pas, on verra très bientôt comment il fonctionne.

Notre service a une dépendance vers `Http` pour récupérer les courses, nous allons donc devoir ajouter un constructeur avec le service `Http` en argument, et ajouter le décorateur `@Injectable()` sur la classe.

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

@Injectable()
export class RacesServiceWithHttp {

  constructor(private http: Http) {
  }

  list() {
    return this.http.get('/api/races');
  }
}
```

Bon c'est pas tout ça, mais il faut que l'on se penche sur les tests maintenant !

#### MISE EN PRATIQUE

Essaye notre exercice [Race service](#) 🐾 ! Il est gratuit et fait partie de notre Pack pro, où tu apprends à construire une application complète étape par étape. Cet exercice te fait construire ton premier service !

# Chapitre 11. Pipes

## 11.1. Ceci n'est pas une pipe

Souvent, les données brutes n'ont pas la forme exacte que l'on voudrait afficher dans la vue. On a envie de les transformer, les filtrer, les tronquer, etc. AngularJS 1.x avait une fonctionnalité bien pratique pour ça, mais mal-nommée : les filters ("filtres"). Ils en ont tiré une leçon, et ces transformateurs de données ont désormais un nom pertinent ! Non, c'est une blague : ils se nomment désormais *pipes* ("tuyaux") :).

Un *pipe* peut être utilisé dans le HTML, ou dans le code applicatif. Etudions quelques cas d'exemple.

## 11.2. json

`json` est un *pipe* pas tellement utile en production, mais bien pratique pour le débug. Ce *pipe* applique simplement `JSON.stringify()` sur tes données. Si tu as dans tes données un tableau de poneys nommé `ponies`, et que tu veux rapidement voir ce qu'il contient, tu aurais pu écrire :

```
<p>{{ ponies }}</p>
```

Pas de chance, cela affichera `[object Object]...`

Mais `JsonPipe` arrive à la rescousse. Tu peux l'utiliser dans n'importe quelle expression, au sein de ton HTML :

```
<p>{{ ponies | json }}</p>
```

Et cela affichera la représentation JSON de ton objet :

```
<p>[ { "name": "Rainbow Dash" }, { "name": "Pinkie Pie" } ]</p>
```

On peut aussi comprendre avec cet exemple d'où vient le nom 'pipe'. Pour utiliser un *pipe*, il faut ajouter le caractère *pipe* (`|`) après tes données, puis le nom du *pipe* à utiliser. L'expression est évaluée, et le résultat traverse le *pipe*. Et il est possible de chaîner plusieurs *pipes*, genre :

```
<p>{{ ponies | slice:0:2 | json }}</p>
```

On reviendra sur le *pipe* `slice`, mais tu constates qu'on enchaîne le *pipe* `slice` et le *pipe* `json`.

Tu peux en utiliser dans une expression interpolée, ou une expression de propriété, mais **pas** dans une instruction d'événement.

```
<p [textContent] = "ponies | json"></p>
```

On peut aussi l'utiliser dans le code , via l'injection de dépendance:

```
import { Component } from '@angular/core';
// you need to import the pipe you want to use
import { JsonPipe } from '@angular/common';

@Component({
  selector: 'ns-ponies',
  template: '<p>{{poniesAsJson}}</p>'
})
export class PoniesComponent {
  ponies: Array<any> = [{ name: 'Rainbow Dash' }, { name: 'Pinkie Pie' }];

  poniesAsJson: string;

  // inject the Pipe you want
  constructor(jsonPipe: JsonPipe) {
    // and then call the transform method on it
    this.poniesAsJson = jsonPipe.transform(this.ponies);
  }
}
```

Mais prends garde : le *pipe* doit être ajouté aux *providers* de ton *@NgModule* (ou de ton *@Component*) pour pouvoir l'injecter et l'utiliser dans ton composant.

Comme l'utilisation programmatique est identique pour tous les *pipes*, je ne te montrerai désormais que des exemples HTML d'utilisation dans une interpolation.

### 11.3. slice

Si tu as envie de n'afficher qu'un sous-ensemble d'une collection, *slice* est ton ami. Il fonctionne comme la méthode du même nom en JavaScript, et prend deux paramètres : un indice de départ et, éventuellement, un indice de fin. Pour passer un paramètre à un *pipe*, il faut lui ajouter un caractère **:** et le premier paramètre, puis éventuellement un autre **:** et le second argument, etc.

Cet exemple va afficher les deux premiers éléments de ma liste de poneys.

```
<p>{{ ponies | slice:0:2 | json }}</p>
```

*slice* fonctionne non seulement avec des tableaux, mais aussi avec des chaînes de caractères, pour n'en afficher qu'une partie :

```
<p>{{ 'Ninja Squad' | slice:0:5 }}</p>
```

et cela affichera seulement 'Ninja'.

Si tu passes à `slice` seulement un entier positif N, il prendra les éléments de N à la fin.

```
<p>{{ 'Ninja Squad' | slice:3 }}</p>
<!-- will display 'ja Squad' -->
```

Si tu lui passes un entier négatif, il prendra les N derniers éléments.

```
<p>{{ 'Ninja Squad' | slice:-5 }}</p>
<!-- will display 'Squad' -->
```

Comme nous l'avons vu, tu peux aussi passer à `slice` un indice de fin : il prendra alors les éléments de l'indice de départ jusqu'à l'indice de fin.

Si cet indice est négatif, il prendra les éléments jusqu'à cet indice, mais en partant cette fois de la fin.

Comme tu peux utiliser `slice` dans n'importe quelle expression, tu peux aussi l'utiliser avec `NgFor` :

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

@Component({
  selector: 'ns-ponies',
  template: '<div *ngFor="let pony of ponies | slice:0:2">{{pony.name}}</div>'
})
export class PoniesComponent {
  ponies: Array<any> = [
    { name: 'Rainbow Dash' },
    { name: 'Pinkie Pie' },
    { name: 'Fluttershy' }
  ];
}
```

Le composant ne créera ici que deux `div`, pour les deux premiers poneys, parce qu'on a appliqué `slice` à la collection.

Le paramètre d'un `pipe` peut bien évidemment avoir une valeur dynamique :

```

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

@Component({
  selector: 'ns-ponies',
  template: `<div *ngFor="let pony of ponies | slice:0:size">{{pony.name}}</div>`
})
export class PoniesComponent {
  size = 2;
  ponies = [
    { name: 'Rainbow Dash' },
    { name: 'Pinkie Pie' },
    { name: 'Fluttershy' }
  ];
}

```

Voici l'exemple parfait d'affichage dynamique où l'utilisateur décide combien d'éléments il veut voir affichés.

Note qu'il est aussi possible de stocker le résultat du `slice` dans une variable, grâce à la syntaxe `as` introduite avec la version 4.0 :

```

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

@Component({
  selector: 'ns-ponies',
  template: `<div *ngFor="let pony of ponies | slice:0:2 as total; index as i">
    {{i+1}}/{{total.length}}: {{pony.name}}
  </div>`
})
export class PoniesComponent {
  ponies: Array<any> = [
    { name: 'Rainbow Dash' },
    { name: 'Pinkie Pie' },
    { name: 'Fluttershy' }
  ];
}

```

## 11.4. uppercase ("majuscule")

Comme son nom le révèle merveilleusement aux bilingues, ce *pipe* transforme une chaîne de caractères dans sa version MAJUSCULE :

```

<p>{{ 'Ninja Squad' | uppercase }}</p>
<!-- will display 'NINJA SQUAD' --&gt;
</pre>

```

## 11.5. lowercase ("minuscule")

Pendant du précédent, ce *pipe* transforme une chaîne de caractères dans sa version minuscule :

```
<p>{{ 'Ninja Squad' | lowercase }}</p>
<!-- will display 'ninja squad' -->
```

## 11.6. titlecase ("titre")

Angular 4 ajoute un nouveau pipe **titlecase**. Celui-ci change la première lettre de chaque mot en majuscule :

```
<p>{{ 'ninja squad' | titlecase }}</p>
<!-- will display 'Ninja Squad' -->
```

## 11.7. number ("nombre")

Ce *pipe* permet de formater un nombre.

Il prend un seul paramètre, une chaîne de caractères, sous la forme **{integerDigits}.{minFractionDigits}-{maxFractionDigits}**, où chaque partie est facultative. Ces parties indiquent:

- combien de chiffres veut-on dans la partie entière;
- combien de chiffres de précision minimale veut-on dans la partie décimale;
- combien de chiffres de précision maximale veut-on dans la partie décimale.

Quelques exemples, à commencer sans *pipe* :

```
<p>{{ 12345 }}</p>
<!-- will display '12345' -->
<p>{{ 12345 }}</p>
<!-- will display '12345' -->
```

Utiliser le *pipe* **number** va grouper la partie entière, même sans aucun paramètre :

```
<p>{{ 12345 | number }}</p>
<!-- will display '12,345' -->
```

Le paramètre **integerDigits** va compléter à gauche avec des zéros si besoin :

```
<p>{{ 12345 | number:'6.' }}</p>
<!-- will display '012,345' -->
```

Le paramètre `minFractionDigits` est la précision minimale de la partie décimale, donc il complétera avec des zéros à droite :

```
<p>{{ 12345 | number:'1.2' }}</p>
<!-- will display '12,345.00' -->
```

Le paramètre `maxFractionDigits` est la précision maximale de la partie décimale. Il est obligatoire de donner une valeur à `minFractionDigits`, fut-elle 0, si tu veux l'utiliser. Si le nombre a plus de décimale que cette valeur, alors il sera arrondi :

```
<p>{{ 12345.13 | number:'1.1-1' }}</p>
<!-- will display '12,345.1' -->

<p>{{ 12345.16 | number:'1.1-1' }}</p>
<!-- will display '12,345.2' -->
```

#### ATTENTION

Ce `pipe` (ainsi que `percent` et `currency`) s'appuie sur l'API d'internationalisation des navigateurs, et cette API n'est pas [disponible dans tous les navigateurs à l'heure actuelle](#). C'est un problème connu, et il oblige à utiliser un *polyfill* pour cette API dans certains navigateurs (notamment Safari/iOS).

## 11.8. percent ("pourcent")

Sur le même principe, `percent` permet d'afficher... un pourcentage !

```
<p>{{ 0.8 | percent }}</p>
<!-- will display '80%' -->

<p>{{ 0.8 | percent:'1.3' }}</p>
<!-- will display '80.000%' -->
```

## 11.9. currency ("devise monétaire")

Comme on l'imagine, ce `pipe` permet de formater une somme d'argent dans la devise que tu veux. Il faut lui fournir au moins un paramètre :

- le code ISO de la devise ('EUR', 'USD'...)
- Optionnellement, un flag booléen indiquant si l'on souhaite afficher le symbole ('€', '\$') ou le code ISO. Par défaut, le flag est à `false`, et le symbole ne sera pas utilisé.
- Optionnellement, une chaîne de formatage du montant, avec la même syntaxe que `number`.

```

<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' -->

```

## 11.10. date

Le *pipe date* transforme une date en chaîne de caractères au format désiré. Cette date peut être un objet *Date*, ou un nombre de millisecondes. Le format spécifié peut être soit un *pattern* comme 'dd/MM/yyyy' ou 'MM-yy', soit un des formats prédéfinis comme 'short', 'longDate', etc.

```

<p>{{ birthday | date:'dd/MM/yyyy' }}</p>
<!-- will display '16/07/1986' -->

<p>{{ birthday | date:'longDate' }}</p>
<!-- will display 'July 16, 1986' -->

```

Evidemment, tu peux aussi afficher la seule partie horaire d'une date :

```

<p>{{ birthday | date:'HH:mm' }}</p>
<!-- will display '15:30' -->

<p>{{ birthday | date:'shortTime' }}</p>
<!-- will display '3:30 PM' -->

```

**NOTE** Pour plus d'informations sur l'internationalisation en général, et en particulier sur la manière d'indiquer la langue à utiliser pour le formatage des nombres et des dates en particulier, tu peux te référer au chapitre sur [l'internationalisation](#).

## 11.11. async

Le *pipe async* permet d'afficher des données obtenues de manière asynchrone. Sous le capot, il utilise *PromisePipe* ou *ObservablePipe* selon que tes données viennent d'une Promise ou d'un Observable. Si tu ne te rappelles pas ce qu'est une Promise, c'est le moment de retourner au chapitre ES6. Et concernant l'Observable, nous y viendrons rapidement.

Un *pipe async* retourne une chaîne de caractères vide jusqu'à ce que les données deviennent disponibles (i.e. jusqu'à 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.

L'exemple suivant utilise une Promise :

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

@Component({
  selector: 'ns-greeting',
  template: '<div>{{ asyncGreeting | async }}</div>'
})
export class GreetingComponent {
  asyncGreeting = new Promise(resolve => {
    // after 1 second, the promise will resolve
    window.setTimeout(() => resolve('hello'), 1000);
  });
}
```

Tu peux voir que le *pipe* `async` est appliqué à la variable `asyncGreeting`. Celle-ci est une promise, résolue après une seconde. Une fois la promise résolue, le navigateur affichera :

```
<div>hello</div>
```

Encore plus intéressant, si la source de données est un Observable, alors le *pipe* se chargera de se désabonner de la source de données à la destruction du composant (quand l'utilisateur naviguera vers une autre page, par exemple).

Et pour éviter de créer de multiples souscriptions à ton Observable ou de multiples appels à ta promesse, tu peux stocker le résultat de l'appel avec `as` (depuis la 4.0) :

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

@Component({
  selector: 'ns-user',
  template: '<div *ngIf="asyncUser | async as user">{{ user.name }}</div>'
})
export class UserComponent {
  asyncUser = new Promise(resolve => {
    // after 1 second, the promise will resolve
    window.setTimeout(() => resolve({ name: 'Cédric' }), 1000);
  });
}
```

## 11.12. Créer tes propres pipes

On peut bien évidemment créer ses propres *pipes*. C'est parfois très utile. En AngularJS 1.x, on utilisait souvent des filtres customs. Par exemple, nous avions créé et utilisé dans plusieurs de nos applications un filtre permettant d'afficher le temps écoulé depuis une action utilisateur (genre "il y a 12 secondes", ou "il y a 3 jours"). Voyons comment faire la même chose en Angular !

Tout d'abord, nous devons créer une nouvelle classe. Elle doit implémenter l'interface `PipeTransform`, ce qui nous amène à écrire une méthode `transform()`, celle qui fait tout le travail.

Ça n'a pas l'air très compliqué, alors essayons !

```
import { PipeTransform, Pipe } from '@angular/core';

export class FromNowPipe implements PipeTransform {
  transform(value, args) {
    // do something here
  }
}
```

Nous utiliserons la fonction `fromNow` de `Moment.js` pour afficher combien de temps s'est écoulé depuis une date.

Tu peux installer Moment.js avec `NPM` si tu le souhaites :

```
npm install moment
```

Et l'ajouter à la configuration de SystemJS :

```
map: {
  'angular2': 'node_modules/angular2',
  'rxjs': 'node_modules/rxjs',
  'moment': 'node_modules/moment/moment'
}
```

Les typings nécessaires pour TypeScript sont déjà inclus dans la dépendance NPM, le compilateur devrait donc être content sans action particulière de notre part.

```
import { PipeTransform, Pipe } from '@angular/core';
import * as moment from 'moment';

export class FromNowPipe implements PipeTransform {
  transform(value, args) {
    return moment(value).fromNow();
  }
}
```

Maintenant, nous devons rendre disponible notre *pipe* dans l'application. Pour cela, il y a un décorateur particulier à utiliser: `@Pipe`.

```

import { PipeTransform, Pipe } from '@angular/core';
import * as moment from 'moment';

@Pipe({ name: 'fromNow' })
export class FromNowPipe implements PipeTransform {
  transform(value, args) {
    return moment(value).fromNow();
  }
}

```

Le nom choisi sera celui qui nous permettra d'utiliser le *pipe* dans le template. Pour pouvoir utiliser le *pipe* dans le template, la dernière chose à faire est d'ajouter le *pipe* dans les `declarations` de ton `@NgModule`.

```

@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent, RacesComponent, FromNowPipe],
  bootstrap: [PonyRacerAppComponent]
})
export class AppModule {
}

```

## N.B.

A sa grande déception, mais à notre soulagement, le traducteur n'a pas réussi à placer le calembour "casser ta pipe" dans le chapitre. N'hésite toutefois pas à te manifester en cas de faute de goût qui nous aurait échappée.

### MISE EN PRATIQUE

Essaye notre exercice [Pipes 🐾](#) ! Il est gratuit et fait partie de notre Pack pro, où tu apprends à construire une application complète étape par étape. Cet exercice te fait utiliser ton premier *pipe*. Plus tard, l'exercice [Custom pipe avec Moment.js 🦄](#) te fait construire ton premier *pipe* custom !

# Chapitre 12. Programmation réactive

## 12.1. OK, on vous rappellera

Tu as probablement entendu parler récemment de programmation réactive ou programmation fonctionnelle réactive (*reactive programming*). C'est devenu assez populaire dans la plupart des plateformes, comme en .Net avec la bibliothèque *Reactive Extensions*, qui est désormais disponible dans la plupart des langages (RxJava, RxJS, etc.).

La programmation réactive n'est pas quelque chose de fondamentalement nouveau. C'est une façon de construire une application avec des événements, et d'y réagir (d'où le nom). Les événements peuvent être combinés, filtrés, groupés, etc. en utilisant des fonctions comme `map`, `filter`, etc. C'est pourquoi tu croiseras parfois le terme de "programmation fonctionnelle réactive" (*functional reactive programming*). Mais pour être tout à fait précis, la programmation réactive n'est pas forcierement fonctionnelle, parce qu'elle n'inclue pas forcément les concepts d'immuabilité, l'absence d'effets de bord, etc. Tu as probablement déjà réagi à des événements :

- dans le navigateur, en enregistrant des listeners sur des actions utilisateurs ;
- côté serveur, en traitement des événements d'un bus de messages.

Dans la programmation réactive, toute donnée entrante sera dans un flux. Ces flux peuvent être écoutés, évidemment modifiés (filtrés, fusionnés, ...), et même devenir un nouveau flux que l'on pourra aussi écouter. Cette technique permet d'obtenir des programmes faiblement couplés : tu n'as pas à te soucier des conséquences de ton appel de méthode, tu te contentes de déclencher un événement, et toutes les parties de l'application intéressées réagiront en conséquence. Et peut-être même qu'une de ces parties va aussi déclencher un événement, etc.

Maintenant, pourquoi est-ce que je parle de tout cela ? Quel est le rapport avec Angular ?

Et bien Angular est construit sur de la programmation réactive, et nous utiliserons aussi cette technique pour certaines parties. Répondre à une requête HTTP ? Programmation réactive. Lever un événement spécifique dans un de nos composants ? Programmation réactive. Gérer un changement de valeurs dans un de nos formulaires ? Programmation réactive.

Alors arrêtons-nous quelques minutes sur ce sujet. Rien de bien compliqué, mais c'est mieux d'avoir les idées claires.

## 12.2. Principes généraux

Dans la programmation réactive, tout est un flux. Un flux est une séquence ordonnée d'événements. Ces événements représentent des valeurs (hé, regarde, une nouvelle valeur !), des erreurs (oups, ça a merdé), ou des terminaisons (voilà, j'ai fini). Tous ces événements sont poussés par un producteur de données, vers un consommateur. En tant que développeur, ton job sera de t'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*.

Ils sont différents des *promises*, même s'ils y ressemblent, car ils gèrent tous deux des valeurs asynchrones. Mais un *observer* n'est pas une chose à usage unique : il continuera d'écouter jusqu'à ce qu'il reçoive un événement de terminaison.

Pour le moment, les *observables* ne font pas partie de la spécification ECMAScript officielle, mais ils feront peut-être partie d'une version future, un effort en cours va dans ce sens.

Les *observables* sont très similaires à des tableaux. Un tableau est une collection de valeurs, comme un *observable*. Un *observable* ajoute juste la notion de valeur reportée dans le temps : dans un tableau, toutes les valeurs sont disponibles immédiatement, dans un *observable*, les valeurs viendront plus tard, par exemple dans plusieurs minutes.

La bibliothèque la plus populaire de programmation réactive dans l'écosystème JavaScript est [RxJS](#). Et c'est celle choisie par Angular.

Jetons-y un œil.

## 12.3. RxJS

Tout *observable*, comme un tableau, peut être transformé avec des fonctions classiques :

- `take(n)` va piocher les n premiers éléments.
- `map(fn)` va appliquer la fonction `fn` sur chaque événement et retourner le résultat.
- `filter(predicate)` laissera passer les seuls événements qui répondent positivement au prédictat.
- `reduce(fn)` appliquera la fonction `fn` à chaque événement pour réduire le flux à une seule valeur unique.
- `merge(s1, s2)` fusionnera les deux flux.
- `subscribe(fn)` appliquera la fonction `fn` à chaque événement qu'elle reçoit.
- et bien d'autres...

Si tu as un tableau de nombres que tu veux tous multiplier par 2, filtrer ceux inférieurs à 5, et les afficher enfin, tu peux écrire :

```
[1, 2, 3, 4, 5]
  .map(x => x * 2)
  .filter(x => x > 5)
  .forEach(x => console.log(x)); // 6, 8, 10
```

RxJS nous permet de construire un *observable* à partir d'un tableau. Et nous pouvons ainsi faire exactement la même chose :

```
Observable.from([1, 2, 3, 4, 5])
  .map(x => x * 2)
  .filter(x => x > 5)
  .subscribe(x => console.log(x)); // 6, 8, 10
```

Mais un *observable* est bien plus qu'une simple collection. C'est une collection asynchrone, dont les événements arrivent au cours du temps. Les événements dans le navigateur sont un bon exemple : ils arriveront au cours du temps, donc ils sont de bons candidats pour l'utilisation d'un *observable*. Voici un exemple avec jQuery :

```
const input = $('input');

Observable.fromEvent(input, 'keyup')
  .subscribe(() => console.log('keyup!'));

input.trigger('keyup'); // logs "keyup!"
input.trigger('keyup'); // logs "keyup!"
```

Tu peux construire des *observables* depuis une requête AJAX, un événement navigateur, une réponse de Websocket, une *promise*, et tout ce que tu peux imaginer. Et depuis une fonction évidemment :

```
const observable = Observable.create((observer) => observer.next('hello'));

observable.subscribe((value) => console.log(value));
// logs "hello"
```

`Observable.create` reçoit une fonction qui émet des événements sur son paramètre `observer`. Ici, elle n'émet qu'un seul événement pour la démonstration.

Tu peux aussi traiter les erreurs, parce qu'un *observable* peut partir en sucette. La méthode `subscribe` accepte un deuxième callback, consacré à la gestion des erreurs.

Ici, la méthode `map` lève une exception, donc le deuxième callback de la méthode `subscribe` va le tracer.

```
Observable.range(1, 5)
  .map(x => {
    if (x % 2 === 1) {
      throw new Error('something went wrong');
    } else {
      return x;
    }
  })
  .filter(x => x > 5)
  .subscribe(x => console.log(x), error => console.log(error)); // something went
  // wrong
```

Une fois que l'*observable* sera terminé, il enverra un événement de terminaison, que tu peux détecter avec un troisième callback. Ici, la méthode `range` utilisée pour générer des événements va boucler de 1 à 5, puis émettre un signal de terminaison :

```
Observable.range(1, 5)
  .map(x => x * 2)
  .filter(x => x > 5)
  .subscribe(x => console.log(x), error => console.log(error), () => console.log('done'));
// 6, 8, 10, done
```

Et tu peux faire plein plein de trucs avec un *observable* :

- transformation (delaying, debouncing...)
- combinaison (merge, zip, combineLatest...)
- filtrage (distinct, filter, last...)
- mathématique (min, max, average, reduce...)
- conditions (amb, includes...)

Il nous faudrait un livre entier pour en venir à bout ! Si tu veux en savoir plus, jette un œil au livre [Rx Book](#). Il contient la meilleure introduction que j'ai vue sur le sujet. Et si tu veux avoir une chouette représentation visuelle de chaque fonction, va sur [rxmarbles.com](#).

Maintenant, regardons comment les *observables* seront utilisés dans Angular.

## 12.4. Programmation réactive en Angular

Angular utilise RxJS, et nous permet aussi de l'utiliser. Le framework propose un adaptateur autour de l'objet `Observable` : `EventEmitter`. `EventEmitter` a une méthode `subscribe()` pour réagir aux événements, et cette méthode reçoit trois paramètres :

- une méthode pour réagir aux événements.
- une méthode pour réagir aux erreurs.
- une méthode pour réagir à la terminaison.

Un `EventEmitter` peut émettre un événement avec la méthode `emit()`.

```

const emitter = new EventEmitter();

emitter.subscribe(
  value => console.log(value),
  error => console.log(error),
  () => console.log('done')
);

emitter.emit('hello');
emitter.emit('there');
emitter.complete();

// logs "hello", then "there", then "done"

```

Note que la méthode `subscribe` retourne un objet *subscription*, avec une méthode `unsubscribe` pour se désabonner.

```

const emitter = new EventEmitter();

const subscription = emitter.subscribe(
  value => console.log(value),
  error => console.log(error),
  () => console.log('done')
);

emitter.emit('hello');
subscription.unsubscribe(); // unsubscribe
emitter.emit('there');

// logs "hello" only

```

Maintenant qu'on en sait un peu plus sur la programmation réactive, voyons l'usage qu'en fait Angular.

#### MISE EN PRATIQUE

Essaye notre exercice [Observables](#) 🐾 ! Il est gratuit et fait partie de notre Pack pro, où tu apprends à construire une application complète étape par étape. Dans cet exercice, tu transformeras le `RaceService` pour le rendre réactif !

# Chapitre 13. Créer des composants et directives

## 13.1. Introduction

Jusque-là, on a surtout vu des petits composants. Et bien sûr, tu peux imaginer que dans la vraie vie, comme ils sont la structure de nos applications, ils seront bien plus complexes que ceux qu'on a croisés. Comment leur fournir des données ? Comment gérer leur cycle de vie ? Quelles sont les bonnes pratiques pour construire ses composants ?

Et les directives : c'est quoi, pourquoi, comment ?

## 13.2. Directives

Une directive est très semblable à un composant, sauf qu'elle n'a pas de template. En fait, la classe `Component` hérite de la classe `Directive` dans le framework.

Il fait donc sens de commencer par étudier les directives, car tout ce que nous verrons pour les directives s'appliquera également pour les composants. On regardera les options de configuration qui te seront le plus utiles. Les options plus exotiques seront réservées à un chapitre ultérieur, quand tu maîtriseras déjà les bases.

Comme pour un composant, ta directive sera annotée d'un décorateur, mais au lieu de `@Component`, ce sera `@Directive` (logique).

Les directives sont des briques minimalistes. On peut les concevoir comme des décorateurs pour ton HTML : elles attacheront du comportement aux éléments du DOM. Et tu peux avoir plusieurs directives appliquées à un même élément.

Une directive doit avoir un sélecteur CSS, qui indique au framework où l'activer dans notre template.

### 13.2.1. Sélecteurs

Les sélecteurs peuvent être de différents types :

- un élément, comme c'est généralement le cas pour les composants : `footer`.
- une classe, mais c'est plutôt rare : `.alert`.
- un attribut, ce qui est le plus fréquent pour une directive : `[color]`.
- un attribut avec une valeur spécifique : `[color=red]`.
- une combinaison de ceux précédents : `footer[color=red]` désignera un élément `footer` avec un attribut `color` à la valeur `red`. `[color], footer.alert` désignera n'importe quel élément avec un attribut `color`, ou `(,)` un élément `footer` portant la classe CSS `alert`. `footer:not(.alert)` désignera un élément `footer` qui ne porte pas `(:not())` la classe CSS `alert`.

Par exemple, voici une directive très simple qui ne fait rien mais est activée si un élément possède

l'attribut `doNothing` :

```
@Directive({
  selector: '[doNothing]'
})
export class DoNothingDirective {

  constructor() {
    console.log('Do nothing directive');
  }
}
```

Cette directive sera activée sur un composant comme `TestComponent` :

```
@Component({
  selector: 'ns-test',
  template: '<div doNothing>Click me</div>'
})
export class TestComponent {
```

Un sélecteur plus complexe pourrait ressembler à :

```
@Directive({
  selector: 'div.loggable[logText]:not([notLoggable=true])'
})
export class ComplexSelectorDirective {

  constructor() {
    console.log('Complex selector directive');
  }
}
```

Celui-là désignera tous les éléments `div` portant la class `loggable` et un attribut `logText`, mais n'ayant pas d'attribut `notLoggable` avec la valeur `true`.

Ainsi, ce template déclenchera la directive :

```
<div class="loggable" logText="text">Hello</div>
```

Mais celui-là, non :

```
<div class="loggable" logText="text" notLoggable="true">Hello</div>
```

Mais pour être honnête, si tu en es à écrire quelque chose comme cela, c'est qu'il y a un truc qui ne va pas ! :)

**NOTE** Les sélecteurs CSS à base de descendants, de frères, ou d'identifiants, et les pseudo-sélecteurs (autres que `:not`) ne sont pas supportés.

**NOTE** Ne nomme pas tes sélecteurs avec un préfixe `bind-`, `on-`, `let-` ou `ref-` : ils ont une autre signification pour le parseur, car ils font partie de la syntaxe canonique des templates.

Bien ! Maintenant on sait déclarer une directive. Il est temps d'en faire une qui sert vraiment à quelque chose !

### 13.2.2. Entrées

Le binding de données est généralement une étape capitale du travail de création d'un composant ou d'une directive. À chaque fois que tu voudras qu'un composant fournit des données à l'un de ses enfants, tu devras utiliser du binding de propriété.

Pour ce faire, nous allons définir toutes les propriétés qui acceptent du binding de données, grâce à l'attribut `inputs` du décorateur `@Directive`. Cet attribut accepte un tableau de chaînes de caractères, chacune sous la forme `property: binding`, où `property` désigne la propriété de l'instance du composant, et `binding` une propriété du DOM qui contiendra l'expression.

Par exemple, cette directive binde la propriété `logText` du DOM sur la propriété `text` de l'instance de directive :

```
@Directive({
  selector: '[loggable]',
  inputs: ['text: logText']
})
export class SimpleTextDirective {
```

Si la propriété n'existe pas dans ta directive, elle est créée. Et, à chaque fois que l'entrée est modifiée, la propriété est mise à jour automatiquement.

```
<div loggable logText="Some text">Hello</div>
```

Si tu veux être notifié quand la propriété est modifiée, tu peux ajouter un setter à ta directive. Le `setter` sera appelé lors de chaque modification de la propriété `logText`.

```

@Directive({
  selector: '[loggable]',
  inputs: ['text: logText']
})
export class SimpleTextWithSetterDirective {

  set text(value) {
    console.log(value);
  }
}

```

Ainsi, si on l'utilise :

```

<div loggable logText="Some text">Hello</div>
// our directive will log "Some text"

```

Il y a également une autre façon de faire que l'on verra dans quelques minutes.

Ici, le texte est statique, mais il pourrait évidemment être une valeur dynamique, avec une interpolation :

```

<div loggable logText="{{expression}}>Hello</div>
// our directive will log the value of 'expression' in the component

```

ou avec la syntaxe crochets :

```

<div loggable [logText]="expression">Hello</div>
// our directive will log the value of 'expression' in the component

```

C'est un des chouettes avantages de la nouvelle syntaxe de template : en tant que développeur de composant, tu n'as pas à te préoccuper de la façon dont ton composant sera utilisé, tu définis juste quelles propriétés peuvent être bindées (si tu as écrit un peu d'AngularJS 1.x, tu sais que c'était différent avec les notations `@` et `=`).

Tu peux aussi utiliser des *pipes* dans tes bindings :

```

<div loggable [logText]="expression | uppercase">Hello</div>
// our directive will log the value of 'expression' in the component in uppercase

```

Si tu veux binder une propriété du DOM sur un attribut de ta directive qui porte le même nom, tu peux écrire simplement `property` au lieu de `property: binding` :

```
@Directive({
  selector: '[loggable]',
  inputs: ['logText']
})
export class SameNameInputDirective {

  set logText(value) {
    console.log(value);
  }
}
```

```
<div loggable logText="Hello">Hello</div>
// our directive will log "Hello"
```

Il y a une autre façon de déclarer une entrée dans ta directive : avec le décorateur `@Input`. Je l'aime bien, et le guide de style officiel indique de préférer cette façon de faire, alors beaucoup d'exemples vont désormais l'utiliser.

Les exemples ci-dessus peuvent être ainsi réécrits :

```
@Directive({
  selector: '[loggable]'
})
export class InputDecoratorDirective {

  @Input('logText') text: string;
}
```

ou, si la propriété et le binding ont le même nom :

```
@Directive({
  selector: '[loggable]'
})
export class SameNameInputDecoratorDirective {

  @Input() logText: string;
}
```

Cela va fonctionner mais avoir un champ et un setter avec le même nom va faire crier le compilateur TypeScript. Une façon de réparer cela (si vous avez besoin du setter, ce qui n'est pas toujours le cas) est d'utiliser le décorateur `@Input` directement sur le setter.

```
@Directive({
  selector: '[loggable]'
})
export class InputDecoratorOnSetterDirective {

  @Input('logText')
  set text(value) {
    console.log(value);
  }
}
```

et si le setter et le binding ont le même nom :

```
@Directive({
  selector: '[loggable]'
})
export class SameNameInputDecoratorOnSetterDirective {

  @Input()
  set logText(value) {
    console.log(value);
  }
}
```

Les entrées sont parfaites pour passer des données d'un élément supérieur à un élément inférieur. Par exemple, si tu veux avoir un composant affichant une liste de poneys, il est probable que tu auras un composant supérieur contenant la liste, et un autre composant inférieur affichant un poney :

```

@Component({
  selector: 'ns-pony',
  template: `<div>{{pony.name}}</div>`
})
export class PonyComponent {
  @Input() pony: Pony;
}

@Component({
  selector: 'ns-ponies',
  template: `<div>
    <h2>Ponies</h2>
    // the pony is handed to PonyComponent via [pony]="currentPony"
    <ns-pony *ngFor="let currentPony of ponies" [pony]="currentPony"></ns-pony>
  </div>`
})
export class PoniesComponent {
  ponies: Array<Pony> = [
    { id: 1, name: 'Rainbow Dash' },
    { id: 2, name: 'Pinkie Pie' }
  ];
}

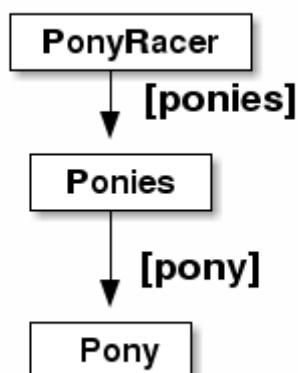
```

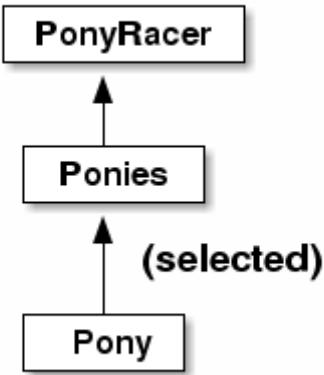
OK, et maintenant, comment passe-t-on des données vers le haut ? On ne peut pas utiliser des propriétés pour passer des données de `PonyComponent` vers `PoniesComponent`. Mais on peut utiliser des événements !

### 13.2.3. Sorties

Retournons à notre exemple précédent, et disons que nous voudrions sélectionner un poney en cliquant dessus, et en informer le composant parent. Pour cela, on utilisera un événement spécifique.

C'est quelque chose de fondamental. En Angular, les données entrent dans un composant via des propriétés, et en sortent via des événements.





Tu te rappelles du dernier chapitre sur la programmation réactive ? Cool, parce que ça va devenir très utile ! Les événements spécifiques sont émis grâce à un `EventEmitter`, et doivent être déclarés dans le décorateur, via l'attribut `outputs`. Comme l'attribut `inputs`, il accepte un tableau contenant la liste des événements que ta directive ou ton composant peut déclencher. Et, comme pour les inputs, on préfère généralement le décorateur `@Output()`.

Disons qu'on veut émettre un événement appelé `ponySelected`. Il y aura trois étapes à réaliser :

- déclarer la sortie dans le décorateur ;
- créer un `EventEmitter` ;
- et émettre un événement quand le poney est sélectionné.

```

@Component({
  selector: 'ns-pony',
  // the method `selectPony()` will be called on click
  template: '<div (click)="selectPony()">{{pony.name}}</div>'
})
export class SelectablePonyComponent {

  @Input() pony: Pony;

  // we declare the custom event as an output,
  // the EventEmitter is used to emit the event
  @Output() ponySelected = new EventEmitter<Pony>();

  /**
   * Selects a pony when the component is clicked.
   * Emits a custom event.
   */
  selectPony() {
    this.ponySelected.emit(this.pony);
  }
}

```

Pour l'utiliser dans le template :

```
<ns-pony [pony]="pony" (ponySelected)="betOnPony($event)"></ns-pony>
```

Dans l'exemple ci-dessus, chaque fois que l'utilisateur clique sur le nom d'un poney, un événement `ponySelected` est émis, avec le poney qui en constitue la valeur (le paramètre de la méthode `emit()`). Le composant parent écoute cet événement, comme tu peux le voir dans le template, et il appellera sa méthode `betOnPony` avec la valeur de l'événement `$event`. `$event` est la syntaxe qui doit être utilisée pour accéder à l'événement déclenché : dans notre cas, ce sera un poney.

Le composant parent doit ensuite implémenter une méthode `betOnPony()`, qui sera appelée avec le poney sélectionné :

```
betOnPony(pony) {  
  // do something with the pony  
}
```

Si tu veux, tu peux définir un nom d'événement différent de celui choisi par l'émetteur, avec la syntaxe `emitter: event` :

```
@Component({  
  selector: 'ns-pony',  
  template: `<div (click)="selectPony()">{${pony.name}}</div>`  
})  
export class OtherSelectablePonyComponent {  
  
  @Input() pony: Pony;  
  // the emitter is called 'emitter'  
  // and the event 'ponySelected'  
  @Output('ponySelected') emitter = new EventEmitter<Pony>();  
  
  selectPony() {  
    this.emitter.emit(this.pony);  
  }  
}
```

### 13.2.4. Cycle de vie

Tu peux avoir besoin que ta directive réagisse à certains moments de sa vie.

C'est un sujet plutôt avancé, et tu n'en auras pas besoin tous les jours, alors je vais aller vite.

Il y a cependant un point important à comprendre, qui te sauvera pas mal de temps si tu l'intègres bien : **les entrées d'un composant ne sont pas encore évaluées dans son constructeur**.

Cela signifie qu'un composant comme celui-ci ne fonctionnera pas :

```

@Directive({
  selector: '[undefinedInputs]'
})
export class UndefinedInputsDirective {

  @Input() pony: string;

  constructor() {
    console.log(`inputs are ${this.pony}`);
    // will log "inputs are undefined", always
  }

}

```

Si tu veux accéder à la valeur d'une entrée, pour par exemple charger des données complémentaires depuis un serveur, il te faudra utiliser une phase du cycle de vie.

Il y a plusieurs phases accessibles, chacune avec ses spécificités propres :

- **ngOnChanges** sera la première appelée quand la valeur d'une propriété bindée est modifiée. Elle recevra une map **changes**, contenant les valeurs courante et précédente du binding, emballées dans un **SimpleChange**. Elle ne sera pas appelée s'il n'y a pas de changement.
- **ngOnInit** sera appelée une seule fois après le premier changement (alors que **ngOnChanges** est appelée à chaque changement). Cela en fait la phase parfaite pour du travail d'initialisation, comme son nom le laisse à penser.
- **ngOnDestroy** est appelée quand le composant est supprimé. Utile pour y faire du nettoyage.

D'autres phases sont disponibles, mais pour des cas d'usage plus avancés :

- **ngDoCheck** est légèrement différente. Si elle est présente, elle sera appelée à chaque cycle de détection de changements, redéfinissant l'algorithme par défaut de détection, qui inspecte les différences pour chaque valeur de propriété bindée. Cela signifie que si une propriété au moins est modifiée, le composant est considéré modifié par défaut, et ses enfants seront inspectés et réaffichés. Mais tu peux redéfinir cela si tu sais que la modification de certaines entrées n'a pas de conséquence. Cela peut être utile si tu veux accélérer le cycle de détection de changements en n'inspectant que le minimum, mais en règle générale tu ne devrais pas avoir à le faire.
- **ngAfterContentInit** est appelée quand tous les bindings du composant ont été vérifiés pour la première fois.
- **ngAfterContentChecked** est appelée quand tous les bindings du composant ont été vérifiés, même s'ils n'ont pas changé.
- **ngAfterViewInit** est appelée quand tous les bindings des directives enfants ont été vérifiés pour la première fois.
- **ngAfterViewChecked** est appelée quand tous les bindings des directives enfants ont été vérifiés, même s'ils n'ont pas changé. Cela peut être utile si ton composant attend quelque chose de ses composants enfants. Comme **ngAfterViewInit**, cela n'a de sens que pour un composant (une directive n'a pas de vue).

Notre exemple précédent fonctionnera mieux en utilisant `ngOnInit`. Angular appelle la méthode `ngOnInit()` si elle est présente, pour que tu n'aies qu'à l'implémenter dans ta directive. Si tu utilises TypeScript pour écrire ton application, tu peux bénéficier de l'interface `OnInit` qui t'oblige à implémenter cette méthode :

```
@Directive({
  selector: '[initDirective]'
})
export class OnInitDirective implements OnInit {

  @Input() pony: string;

  ngOnInit() {
    console.log(`inputs are ${this.pony}`);
    // inputs are not undefined \o/
  }
}
```

Maintenant on a accès à nos entrées !

Si tu veux faire un truc lors de chaque changement de la propriété, utilise `ngOnChanges` :

```
@Directive({
  selector: '[changeDirective]'
})
export class OnChangesDirective implements OnChanges {

  @Input() pony: string;

  ngOnChanges(changes: SimpleChanges) {
    const ponyValue = changes['pony'];
    console.log(`changed from ${ponyValue.previousValue} to ${ponyValue.currentValue}`);
    console.log(`is it the first change? ${ponyValue.isFirstChange()}`);
  }
}
```

Le paramètre `changes` est une map, dont les clés sont les noms de bindings, et les valeurs un objet `SimpleChange` avec deux attributs (la valeur précédente et la valeur courante), ainsi qu'une méthode `isFirstChange()` afin de savoir si c'est le premier changement.

Tu peux aussi utiliser un `setter` si tu veux réagir sur le changement d'un seul de tes bindings. L'exemple suivant produira le même affichage que le précédent.

```

@Directive({
  selector: '[setterDirective]'
})
export class SetterDirective {

  private ponyModel: string;

  @Input()
  set pony(newPony) {
    console.log(`changed from ${this.ponyModel} to ${newPony}`);
    this.ponyModel = newPony;
  }

}

```

`ngOnChanges` est encore plus pratique si tu dois surveiller plusieurs bindings en même temps. Elle ne sera invoquée que si au moins un binding a changé, et elle ne contiendra que les propriétés modifiées.

La phase `ngOnDestroy` est parfaite pour nettoyer le composant. Par exemple, pour y annuler les tâches de fond. Ici `OnDestroyDirective` trace "hello" toute les secondes quand elle est créée. Quand le composant sera supprimé de la page, on veut arrêter le `setInterval` pour éviter des fuites mémoire :

```

@Directive({
  selector: '[destroyDirective]'
})
export class OnDestroyDirective implements OnDestroy {

  sayHello: number;

  constructor() {
    this.sayHello = window.setInterval(() => console.log('hello'), 1000);
  }

  ngOnDestroy() {
    window.clearInterval(this.sayHello);
  }

}

```

Si tu ne fais pas cela, tu auras un thread traçant "hello" jusqu'à la fin de l'application (ou son plantage)...

### 13.2.5. Providers

On a déjà parlé des providers dans le [chapitre Injection de dépendances](#). Cet attribut permet de déclarer des services qui seront injectables dans la directive courante et ses enfants.

```

@Directive({
  selector: '[providersDirective]',
  providers: [PoniesService]
})
export class ProvidersDirective {

  constructor(poniesService: PoniesService) {
    const ponies = poniesService.list();
    console.log(`ponies are: ${ponies}`);
  }
}

```

## 13.3. Composants

Un composant n'est pas vraiment différent d'une directive : il a simplement deux attributs supplémentaires, optionnels, et **doit** avoir une vue associée. Il n'apporte pas beaucoup d'attributs nouveaux comparés à la directive.

### 13.3.1. Providers limités à la vue

On a vu que tu peux spécifier les services injectables via `providers.viewProviders` est similaire, mais les providers donnés ne seront disponibles que dans la vue du composant courant, pas dans ses enfants.

### 13.3.2. Template / URL de template

La caractéristique principale d'un `@Component` est d'avoir un template, alors qu'une directive n'en a pas. Tu peux soit déclarer ton template en ligne, avec l'attribut `template`, ou utiliser une URL pour le placer dans un fichier séparé avec `templateURL` (mais tu ne peux pas définir les deux simultanément).

En règle générale, si ton template est petit (1-2 lignes), c'est parfaitement acceptable de le garder en ligne. Quand il commence à grossir, le déplacer dans son propre fichier est une bonne façon d'éviter l'encombrement de ton composant.

Tu peux utiliser un chemin absolu pour ton URL, ou un relatif, ou même une URL HTTP complète.

Quand le composant est chargé, Angular résout l'URL et essaye de charger le template. S'il y parvient, le template deviendra le *Shadow Root* du composant, et ses expressions seront évaluées.

Si j'ai un gros composant, je place en général le template dans un fichier séparé du même répertoire, et j'utilise une URL relative pour le charger.

```

@Component({
  selector: 'ns-templated-pony',
  templateUrl: 'components/pony/templated-pony.html'
})
export class TemplatedPonyComponent {
  @Input() pony: any;
}

```

Si tu utilises une URL relative, l'URL sera résolue en utilisant l'URL de la base de ton application. Cette URL peut être un peu lourde, car si ton composant est dans un répertoire `components/pony`, ton URL de template sera `components/pony/pony.html`.

Mais tu peux faire légèrement mieux si tu packages ton application avec des modules CommonJS, en utilisant la propriété `moduleId`. Sa valeur doit être `module.id`, une valeur qui sera remplacée par CommonJS à l'exécution. Angular peut alors utiliser cette valeur et construire l'URL relative correcte. Ton URL de template ressemble alors à :

```

@Component({
  selector: 'ns-templated-pony',
  templateUrl: 'templated-pony.html',
  moduleId: module.id
})
export class ModuleIdPonyComponent {
  @Input() pony: any;
}

```

Et tu peux maintenant mettre ton template dans le même dossier que ton composant !

Encore mieux: si tu utilises Webpack, avec un peu de configuration (déjà faite pour toi par Angular CLI), tu peux carrément enlever `module.id` et utiliser un chemin relatif directement. Webpack générera le chemin absolu pour toi !

### 13.3.3. Styles / URL de styles

Tu peux aussi définir les styles de ton composant. C'est particulièrement utile si tu comptes faire des composants vraiment isolés. Tu peux spécifier cela avec `styles` ou `styleUrls`.

Comme tu peux le voir ci-dessous, l'attribut `styles` reçoit un tableau de règles CSS sous forme de chaînes de caractères. Comme tu peux l'imaginer, elles deviennent vite énormes, alors utiliser un fichier séparé et `styleUrls` est une bonne idée. Comme le nom le laisse deviner, tu peux y indiquer un tableau d'URLs.

```
@Component({
  selector: 'ns-styled-pony',
  template: '<div class="pony">{{pony.name}}</div>',
  styles: ['.pony{ color: red; }']
})
export class StyledPonyComponent {
  @Input() pony: any;
}
```

### 13.3.4. Déclarations

Rappelle-toi qu'il te faut déclarer dans les `declarations` de ton `@NgModule` chaque directive et composant que tu utilises. Si tu ne le fais pas, ton composant ne sera pas déclenché par le template, et tu perdras beaucoup de temps à comprendre pourquoi.

Les deux erreurs les plus fréquentes sont **oublier de déclarer les directives** et **définir le mauvais sélecteur**. Si un jour tu ne comprends pas pourquoi rien ne se passe, il serait de bon goût de vérifier cela !

Nous avons laissé de côté deux ou trois points pour l'instant, comme les requêtes, les détections de changements, les exports, les options d'encapsulation, etc. Comme ce sont des options plus avancées, tu n'en auras pas besoin immédiatement, mais ne t'inquiète pas, nous les verrons bientôt dans un chapitre dédié aux sujets avancés !

#### MISE EN PRATIQUE

Essaye nos exercices [Détail d'une course](#)  et [Composant Pony](#)  !

Ces exercices vont te guider dans la construction de deux composants avancés avec entrées et sorties.

# Chapitre 14. Style des composants et encapsulation

Faisons une pause pour parler un peu style et CSS quelques instants. Oh nooOon, mais pourquoi parler de CSS ?! Parce qu'Angular fait beaucoup pour nous sur ce sujet.

En tant que développeur web, tu ajoutes souvent des classes CSS sur tes éléments. Et l'essence même de CSS, c'est que ce style va *cascader* (rappelons que CSS signifie *Cascading Style Sheet*). C'est souvent un comportement voulu (pour par exemple changer la police globalement dans toute l'application), mais des fois non. Imaginons que tu veux ajouter un style sur l'élément sélectionné d'une liste : tu vas probablement utiliser un sélecteur CSS très restreint, comme `li.selected`. Ou encore plus restreint, en utilisant une convention comme `BEM`, parce que tu veux styler l'élément sélectionné dans une partie bien spécifique de ton application.

Et sur ce point Angular peut se rendre utile. Les styles que tu définis dans un composant (soit dans l'attribut `styles`, soit dans un fichier CSS dédié avec `styleUrls`) sont limités par Angular à ce composant et seulement celui-ci. Cela s'appelle l'encapsulation de style (*style encapsulation*). Mais comment est-ce possible ?

Tout commence par l'écriture d'un style. Ensuite, cela dépend de la stratégie que tu auras choisie pour l'attribut `encapsulation` du décorateur du composant. Cet attribut peut prendre trois valeurs :

- `ViewEncapsulation.Emulated`, qui est la valeur par défaut ;
- `ViewEncapsulation.Native`, qui s'appuie sur le *Shadow DOM* ;
- `ViewEncapsulation.None`, qui signifie que tu ne veux pas d'encapsulation.

Chaque valeur va évidemment engendrer un comportement différent. Prenons par exemple notre composant `PonyComponent`, que tu dois commencer à bien connaître. C'est une version simplifiée, affichant seulement le nom du poney dans une `div`. Pour l'exemple, ajoutons une classe CSS `red` à cette `div` :

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

@Component({
  selector: 'ns-pony',
  template: '<div class="red">{{name}}</div>',
  styles: ['.red {color: red;}'],
  // that's the same as the default mode
  encapsulation: ViewEncapsulation.Emulated
})
export class PonyComponent {

  name = 'Rainbow Dash';

}
```

Cette classe est ensuite utilisée dans les styles du composant :

```
.red {  
  color: red;  
}
```

Comme tu le vois, on veut afficher le nom du poney avec une police rouge.

## 14.1. Stratégie Native

Si tu utilises la stratégie **Native**, tu demandes à Angular d'utiliser le *Shadow DOM* du navigateur pour assurer l'encapsulation. Le *Shadow DOM* fait partie de la spécification récente des Web Component. Cette spécification permet de créer des éléments dans un DOM spécial, parfaitement encapsulé. Avec cette stratégie, si on inspecte le DOM généré dans notre navigateur, on y verra :

```
<ns-pony>  
#shadow-root (open)  
  <style>.red {color: red}</style>  
  <div class="red">Rainbow Dash</div>  
</ns-pony>
```

Tu peux y repérer `#shadow-root (open)` que Chrome affiche dans l'inspecteur : c'est parce que notre composant a été ajouté dans un élément du *Shadow DOM* ! Et on peut aussi voir que notre style a été ajouté en tête du contenu du composant.

Avec la stratégie **Native**, tu peux être sûr que les styles de ton composant ne vont pas "transpirer" sur tes composants enfants. Si nous avions un autre composant à l'intérieur de **PonyComponent**, il pourrait aussi définir sa propre classe CSS `red` avec un style différent : tu serais sûr que le bon serait utilisé, sans confusion entre eux !

Mais rappelle-toi, le *Shadow DOM* est une spécification vraiment récente, qui n'est pas encore disponible dans tous les navigateurs. Tu peux en vérifier la disponibilité sur le merveilleux site [caniuse.com](http://caniuse.com). Alors sois prudent quand tu l'utilises dans tes applications !

## 14.2. Stratégie Emulated ("émulée")

Comme je le disais précédemment, c'est la stratégie par défaut. Et la raison en est simple : elle émule (d'où le nom) la stratégie **Native**, mais sans utiliser le *Shadow DOM*. Elle peut donc être utilisée partout, et aura le même comportement.

Pour y parvenir, Angular va *inliner* le CSS que nous définirons dans le `<head>` de la page (et non pas dans chaque composant comme on l'a vu avec la stratégie *Native*). Mais avant de l'*inliner*, Angular va réécrire le sélecteur CSS, pour lui ajouter un attribut identifiant unique. Cet attribut unique sera ensuite ajouté sur tous les éléments du template du composant ! Comme ceci, le style s'appliquera sur notre composant uniquement. Le même exemple donnerait donc ceci :

```

<html>
  <head>
    <style>.red[_ngcontent-dvb-3] {color: red}</style>
  </head>
  <body>
    ...
    <ns-pony _ngcontent-dvb-2="" _nghost-dvb-3="">
      <div _ngcontent-dvb-3="" class="red">Rainbow Dash</div>
    </ns-pony>
  </body>
</html>

```

Le sélecteur de classe `red` a été réécrit en `.red[_ngcontent-dvb-3]`, alors il ne s'appliquera que sur les éléments qui auront à la fois la classe `red` et l'attribut `_ngcontent-dvb-3`. Tu peux aussi vérifier que cet attribut a bien été ajouté automatiquement à notre `div`, pour que tout fonctionne parfaitement. L'élément `<ns-pony>` a aussi quelques attributs : `_ngcontent-dvb-2` qui est l'identifiant unique généré pour son parent, et `_nghost-dvb-3` qui est l'identifiant unique de l'élément hôte lui-même. Oui, parce que l'on peut aussi ajouter des styles sur l'élément hôte, comme nous le verrons bientôt.

## 14.3. Stratégie `None` ("aucune")

Cette stratégie ne réalise aucune encapsulation. Les styles seront *inlinés* en haut de la page (comme pour la stratégie `Emulated`), mais ne seront pas réécrits. Ils se comporteront donc comme des styles "normaux", qui cascaderont sur les enfants.

## 14.4. Style l'hôte

Un sélecteur CSS spécial existe pour styler spécifiquement l'élément hôte. Il s'appelle `:host`, et il vient de la spécification des Web Components :

```

:host {
  display: block;
}

```

Il sera conservé tel quel avec la stratégie `Native`, comme dans la spécification du *Shadow DOM*, ou sera réécrit en `[_nghost-xxx]` si tu utilises `Emulated`.

Pour conclure, il n'y a pas grand chose à faire pour avoir des styles parfaitement encapsulés, car la stratégie `Emulated` fait tout le boulot pour nous. Tu peux basculer sur la stratégie `Native` si tu ne cibles que des navigateurs spécifiques, ou sur `None` si tu n'as pas besoin d'encapsuler tes styles. Cette stratégie peut se définir au sein de chaque composant, ou globalement pour l'application complète dans le module racine.

# Chapitre 15. Tester ton application

## 15.1. Tester c'est douter

J'adore les tests automatisés. Ma vie professionnelle tourne autour de cette barre de progression qui devient verte dans mon IDE, et me félicite d'avoir bien travaillé. J'espère que les tests sont aussi importants pour toi, car ils sont le filet de sécurité ultime quand on écrit du code. Rien n'est plus pénible que tester du code manuellement.

Angular fait du bon boulot en ce qui concerne la facilité d'écriture de tests. AngularJS 1.x était déjà très bon, et c'était une des raisons pour lesquelles j'appréciais ce framework. Comme en AngularJS 1.x, on peut écrire deux types de tests :

- tests unitaires ;
- tests *end-to-end* ("de bout en bout").

Les premiers sont là pour affirmer qu'une petite portion de code (un composant, un service, un *pipe*) fonctionne correctement en isolation, c'est à dire indépendamment de ses dépendances. Ecrire un tel test unitaire demande d'exécuter chacune des méthodes d'un composant/service/*pipe*, et de vérifier que les sorties sont celles attendues pour les entrées fournies. On peut aussi vérifier que les dépendances utilisées par cette portion sont correctement invoquées, par exemple qu'un service fait la bonne requête HTTP.

On peut aussi écrire des tests *end-to-end* ("de bout en bout"). Ceux-ci émulent une interaction utilisateur réelle avec ton application, en démarrant une vraie instance et en pilotant le navigateur pour saisir des valeurs dans les champs, cliquer sur les boutons, etc. On vérifiera ensuite que la page affichée contient ce qui est attendu, que l'URL est correcte, et tout ce que tu peux imaginer.

On va parler de tout cela, mais commençons par les tests unitaires.

## 15.2. Test unitaire

Comme on l'a vu, les tests unitaires vérifient une petite portion de code en **isolation**. Ces tests garantissent seulement qu'une petite partie de l'application fonctionne comme prévu, mais ils ont de nombreux avantages :

- ils sont vraiment rapides, tu peux en exécuter plusieurs centaines en quelques secondes.
- ils sont très efficaces pour tester (quasiment) l'intégralité de ton code, et particulièrement les cas limites, qui peuvent être difficiles à reproduire manuellement dans l'application réelle.

L'un des concepts au cœur d'un test unitaire est celui d'*isolation* : on ne veut pas que notre test soit biaisé par ses dépendances. Alors on utilise généralement des objets bouchonnés (*mock*) comme dépendances. Ce sont des objets factices créés juste pour les besoins du test.

Pour tout cela, on va compter sur quelques outils. D'abord, on a besoin d'une bibliothèque pour écrire des tests. Une des plus populaires (sinon la plus populaire) est [Jasmine](#), on va donc utiliser celle-ci !

## 15.2.1. Jasmine & Karma

Jasmine nous propose plusieurs méthodes pour déclarer nos tests :

- `describe()` déclare une suite de tests (un groupe de tests) ;
- `it()` déclare un test ;
- `expect()` déclare une assertion.

Un test JavaScript basique ressemble à :

```
class Pony {  
    constructor(public name: string, public speed: number) {  
    }  
  
    isFasterThan(speed) {  
        return this.speed > speed;  
    }  
}  
  
describe('My first test suite', () => {  
    it('should construct a Pony', () => {  
        const pony = new Pony('Rainbow Dash', 10);  
        expect(pony.name).toBe('Rainbow Dash');  
        expect(pony.speed).not.toBe(1);  
        expect(pony.isFasterThan(8)).toBe(true);  
    });  
});
```

L'appel à `expect()` peut être chaîné avec plein de méthodes comme `toBe()`, `toBeLessThan()`, `toBeUndefined()`, etc. Chaque méthode peut être rendue négative avec l'attribut `not` de l'objet retourné par `expect()`.

Le fichier de test est un fichier séparé du code que tu veux tester, en général avec une extension comme `.spec.ts`.

Le test d'une classe `Pony` écrite dans un fichier `pony.ts` sera normalement dans un fichier nommé `pony.spec.ts`. Tu peux soit poser ce fichier de test juste à côté du fichier à tester, soit dans un répertoire dédié contenant tous les tests. J'ai tendance à placer le code et le test dans le même répertoire, mais les deux approches sont parfaitement acceptables : choisis ton camp.

### NOTE

Une astuce : si tu utilises `fdescribe()` au lieu de `describe()` alors cette seule suite de tests sera exécutée (le "f" ajouté signifie "focus"). Même chose si tu ne veux exécuter qu'un seul test : utilise `fit()` au lieu de `it()`. Si tu veux exclure un test, utilise `xit()`, ou `xdescribe()` pour une suite de tests.

Tu peux aussi utiliser la méthode `beforeEach()` pour initialiser un contexte avec chaque test. Si j'ai plusieurs tests sur le même poney, il est préférable d'utiliser `beforeEach()` pour initialiser ce poney, plutôt que copier/coller le même morceau de code dans tous les tests.

```

describe('Pony', () => {
  let pony: Pony;

  beforeEach(() => {
    pony = new Pony('Rainbow Dash', 10);
  });

  it('should have a name', () => {
    expect(pony.name).toBe('Rainbow Dash');
  });

  it('should have a speed', () => {
    expect(pony.speed).not.toBe(1);
    expect(pony.speed).toBeGreaterThanOrEqual(9);
  });
});

```

Il existe aussi une méthode `afterEach`, mais je n'en ai juste jamais eu besoin...

Une dernière astuce : Jasmine nous permet de créer des objets factices (bouchons ou espions, comme tu veux), ou même d'espionner une méthode d'un véritable objet. On peut alors faire quelques assertions sur ces méthodes, comme `toHaveBeenCalled()` qui vérifie que la méthode a bien été invoquée, ou `toHaveBeenCalledWith()` qui vérifie les paramètres exacts utilisés lors de l'invocation de la méthode espionnée. Tu peux aussi vérifier combien de fois la méthode a été invoquée, ou vérifier si elle ne l'a pas été, etc.

```

describe('My first test suite with spyOn', () => {
  let pony: Pony;

  beforeEach(() => {
    pony = new Pony('Rainbow Dash', 10);
    // define a spied method
    spyOn(pony, 'isFasterThan').and.returnValue(true);
  });

  it('should test if the Pony is fast', () => {
    const runPonyRun = pony.isFasterThan(60);
    expect(runPonyRun).toBe(true); // as the spied method always returns true
    expect(pony.isFasterThan).toHaveBeenCalled();
    expect(pony.isFasterThan).toHaveBeenCalledWith(60);
  });
});

```

Quand tu écris des tests unitaires, garde à l'esprit qu'ils doivent rester courts et lisibles. Et n'oublie pas de les faire d'abord échouer, pour être sûr que tu testes la bonne chose.

La dernière étape est l'exécution de nos tests. Pour cela, l'équipe Angular a développé [Karma](#), dont le seul but est d'exécuter nos tests dans un ou plusieurs navigateurs. Cette bibliothèque peut aussi

surveiller nos fichiers pour réexécuter nos tests lors de chaque enregistrement. Comme l'exécution est très rapide, c'est vraiment bien d'avoir ça, pour avoir un retour instantané sur notre code.

Je ne vais pas me lancer dans les détails de la mise en place de Karma, mais c'est un projet très intéressant avec une tonne de plugins à disposition, pour le faire fonctionner avec tes outils préférés, pour mesurer la couverture de tests, etc. Si comme moi tu écris ton code en TypeScript, la stratégie que tu peux adopter est de laisser le compilateur TypeScript surveiller ton code et tes tests, et produire les fichiers compilés dans un répertoire séparé, et avoir Karma qui surveille ce dernier répertoire.

Donc on sait maintenant comment écrire un test unitaire en JavaScript. Ajoutons maintenant Angular à la recette.

### 15.2.2. Utiliser l'injection de dépendances

Disons que j'ai une application Angular avec un service simple, comme `RaceService`, avec une méthode retournant une liste de courses codée en dur.

```
export class RaceService {
  list() {
    const race1 = new Race('London');
    const race2 = new Race('Lyon');
    return [race1, race2];
  }
}
```

Essayons de tester cela.

```
describe('RaceService', () => {
  it('should return races when list() is called', () => {
    const raceService = new RaceService();
    expect(raceService.list().length).toBe(2);
  });
});
```

Ca fonctionne. Mais on pourrait aussi bénéficier de l'injection de dépendances proposée par Angular pour récupérer le `RaceService` et l'injecter dans notre test. C'est d'autant plus utile que notre `RaceService` a lui aussi des dépendances : plutôt que devoir instancier nous-même chacune des ses dépendances, on peut compter sur l'injecteur pour le faire à notre place en lui disant "hé, j'ai besoin de `RaceService`, débrouille-toi pour le créer, lui fournir ce dont il a besoin, et me le donner".

Pour utiliser le système d'injection de dépendances dans notre test, le framework a une méthode utilitaire dans la classe `TestBed` nommée `get`.

Cette méthode est utilisée dans notre test pour récupérer des dépendances spécifiques.

Si on retourne à notre exemple, cette fois-ci en utilisant `TestBed.get` :

```

import { TestBed } from '@angular/core/testing';

describe('RaceService', () => {
  it('should return races when list() is called', () => {
    const raceService = TestBed.get(RaceService);
    expect(raceService.list().length).toBe(2);
  });
});

```

Ça ne fonctionnera pas exactement comme cela, parce qu'il nous faudra aussi indiquer au test ce qui est disponible à l'injection, comme on le fait avec le module racine quand on démarre l'application.

La classe `TestBed` va nous aider. Sa méthode `configureTestingModule` permet de déclarer ce qui pourra être injecté dans le test, en créant un module de test ne contenant que ce dont nous avons besoin. Essaye de n'injecter que le strict nécessaire dans ton test, pour le rendre le plus faiblement couplé possible du reste de l'application. La méthode est appelée dans le `beforeEach` de Jasmine pour être exécutée avant nos tests, et prend une configuration de module comme paramètre, très proche de ce que nous passons au décorateur `@NgModule`. L'attribut `providers` reçoit un tableau de dépendances qui deviendront disponibles à l'injection.

```

import { TestBed } from '@angular/core/testing';

describe('RaceService', () => {

  beforeEach(() => TestBed.configureTestingModule({
    providers: [RaceService]
  });

  it('should return races when list() is called', () => {
    const raceService = TestBed.get(RaceService);
    expect(raceService.list().length).toBe(2);
  });
});

```

Maintenant ça fonctionne, cool ! Note que si notre `RaceService` avait lui aussi des dépendances, nous aurions dû les déclarer dans l'attribut `providers`, pour les rendre disponibles à l'injection.

Comme on l'a fait dans l'exemple simple de Jasmine, on devrait déplacer l'initialisation de `RaceService` dans une méthode `beforeEach`. On peut aussi utiliser `TestBed.get` dans un `beforeEach` :

```

import { TestBed } from '@angular/core/testing';

describe('RaceService', () => {
  let service: RaceService;

  beforeEach(() => TestBed.configureTestingModule({
    providers: [RaceService]
  }));

  beforeEach(() => service = TestBed.get(RaceService));

  it('should return races when list() is called', () => {
    expect(service.list().length).toBe(2);
  });
});

```

On a déplacé la logique de `TestBed.get` dans un `beforeEach`, et maintenant notre test est plutôt clair. Pense bien à toujours invoquer `TestBed.configureTestingModule` (qui initialise l'injecteur) avant d'utiliser l'injecteur avec `TestBed.get`, sinon ton test échouera.

Évidemment, un véritable `RaceService` n'aura pas une liste de courses écrite en dure, et il est fortement probable que cette réponse soit asynchrone. Disons que la méthode `list` retourne une promesse. Qu'est-ce que cela change dans notre test ? Et bien nous devrons mettre `expect` dans le callback `then` :

```

import { async, TestBed } from '@angular/core/testing';

describe('RaceService', () => {
  let service: RaceService;

  beforeEach(() => TestBed.configureTestingModule({
    providers: [RaceService]
  }));

  beforeEach(() => service = TestBed.get(RaceService));

  it('should return a promise of 2 races', async(() => {
    service.list().then(races => {
      expect(races.length).toBe(2);
    });
  }));
});

```

Tu te dis peut-être que cela ne fonctionnera pas, car le test se terminera avant que la promesse soit résolue, et notre assertion ne sera jamais exécutée.

Mais nous avons utilisé la fonction `async()` et cette méthode est vraiment intelligente : elle enregistre tous les appels asynchrones faits dans le test et attend leur résolution.

Angular utilise un nouveau concept appelé **zones**. Ces **zones** sont des contextes d'exécution, et, pour simplifier, elles enregistrent tout ce qui se passe en leur sein (timeouts, event listeners, callbacks, ...). Elles proposent aussi des points d'extensions qui peuvent être appelés quand on entre ou sort de la zone. Une application Angular tourne dans une zone, et c'est comme cela que le framework sait qu'il doit rafraîchir le DOM quand une action asynchrone est réalisée.

Ce concept est aussi utilisé dans les tests si ton test utilise `async()` : chaque test tourne dans une zone, ainsi le framework sait quand toutes les actions asynchrones sont terminées, et ne se finira pas jusqu'alors.

Et donc nos assertions asynchrones seront exécutés. Cool !

Il y a une autre façon de gérer les tests asynchrones en Angular, en utilisant `fakeAsync()` et `tick()`, mais on garde ça pour un chapitre plus avancé.

## 15.3. Dépendances factices

Pouvoir déclarer les dépendances dans le module de test a une autre utilité : on peut sans problème y déclarer une dépendance factice plutôt qu'une vraie.

Pour le bien de mon exemple, disons que mon `RaceService` utilise le *local storage* pour stocker les courses, avec une clé `races`. Tes collègues ont développé un service appelé `LocalStorageService` qui gère la sérialisation JSON, etc. que notre `RaceService` utilise. La méthode `list()` ressemble à :

```
@Injectable()
export class RaceService {
  constructor(private localStorage: LocalStorageService) {}

  list() {
    return this.localStorage.get('races');
  }
}
```

Maintenant, nous ne voulons pas vraiment tester le service `LocalStorageService`, nous voulons juste tester notre `RaceService`. Cela peut être fait en bénéficiant du système d'injection de dépendances pour donner un `LocalStorageService` factice :

```
class FakeLocalStorage {
  get(key) {
    return [{ name: 'Lyon' }, { name: 'London' }];
  }
}
```

au `RaceService` dans notre test, en utilisant `provide` :

```

import { TestBed } from '@angular/core/testing';

describe('RaceService', () => {

  beforeEach(() => TestBed.configureTestingModule({
    providers: [
      { provide: LocalStorageService, useClass: FakeLocalStorage },
      RaceService
    ]
  }));

  it('should return 2 races from localStorage', () => {
    const service = TestBed.get(RaceService);
    const races = service.list();
    expect(races.length).toBe(2);
  });
});

```

Cool ! Mais je ne suis pas complètement satisfait de ce test. Créer manuellement un service factice est laborieux, et Jasmine peut nous aider à espionner le service et à remplacer son implémentation par une factice. Elle permet aussi de vérifier que la méthode `get()` a été invoquée avec la bonne clé 'races'.

```

import { TestBed } from '@angular/core/testing';

describe('RaceService', () => {

  const localStorage = jasmine.createSpyObj('LocalStorageService', ['get']);

  beforeEach(() => TestBed.configureTestingModule({
    providers: [
      { provide: LocalStorageService, useValue: localStorage },
      RaceService
    ]
  }));

  it('should return 2 races from localStorage', () => {
    localStorage.get.and.returnValue([{ name: 'Lyon' }, { name: 'London' }]);
    const service = TestBed.get(RaceService);
    const races = service.list();

    expect(races.length).toBe(2);
    expect(localStorage.get).toHaveBeenCalledWith('races');
  });
});
;
```

## 15.4. Tester des composants

Après avoir testé un service simple, l'étape suivante est de tester un composant. Un test de composant est légèrement différent car il nous faut créer le composant. On ne peut pas compter sur le système d'injection de dépendances pour nous fournir une instance du composant à tester (tu as probablement déjà remarqué que les composants ne sont pas injectables dans d'autres composants :)).

Commençons par écrire un composant à tester. Pourquoi pas notre composant `PonyComponent` ? Il prend un poney en entrée et émet un événement `ponyClicked` quand le composant est cliqué.

```
@Component({
  selector: 'ns-pony',
  template: '<img [src]="/images/pony-' + pony.color.toLowerCase() + '.png"'
  (click)="clickOnPony()"
})
export class PonyComponent {

  @Input() pony: PonyModel;
  @Output() ponyClicked = new EventEmitter<PonyModel>();

  clickOnPony() {
    this.ponyClicked.emit(this.pony);
  }
}
```

Il a un template plutôt simple : une image avec une source dynamique dépendante de la couleur du poney, et un gestionnaire de clic.

Pour tester un tel composant, tu as d'abord besoin de créer une instance. Pour ce faire, nous allons aussi utiliser `TestBed`. Cette classe vient avec une méthode utilitaire, nommée `createComponent`, pour créer un composant. Cette méthode retourne un `ComponentFixture`, une représentation de notre composant.

Note que pour créer un composant, il doit être connu du module de test, nous devons donc l'ajouter à l'attribut `declarations` :

```

import { TestBed } from '@angular/core/testing';
import { PonyComponent } from './pony_cmp';

describe('PonyComponent', () => {
  it('should have an image', () => {
    TestBed.configureTestingModule({
      declarations: [PonyComponent]
    });
    const fixture = TestBed.createComponent(PonyComponent);
    // given a component instance with a pony input initialized
    const ponyComponent = fixture.componentInstance;
    ponyComponent.pony = { name: 'Rainbow Dash', color: 'BLUE' };

    // when we trigger the change detection
    fixture.detectChanges();

    // then we should have an image with the correct source attribute
    // depending of the pony color
    const element = fixture.nativeElement;
    expect(element.querySelector('img').getAttribute('src')).toBe('/images/pony-
blue.png');
  });
});

```

Ici, on suit le modèle "Given/When/Then" ("soit/quand/alors") pour écrire notre test. Tu trouveras une littérature complète sur le sujet, mais il se résume à :

- une phase "Given", où on met en place le contexte du test. On y récupère l'instance du composant créé, et lui donne un poney. Cela simule une entrée qui viendrait d'un composant parent dans la vraie application.
- une phase "When", où on déclenche manuellement la détection de changements, via la méthode `detectChanges()`. Dans un test, la détection de changement est de notre responsabilité : ce n'est pas automatique comme ça l'est dans une application.
- et une phase "Then", contenant les assertions. On peut récupérer l'élément natif et interroger le DOM comme on le ferait dans un navigateur (avec `querySelector()` par exemple). Ici on teste si la source de l'image est la bonne.

On peut également tester que le composant émet bien un événement :

```

it('should emit an event on click', () => {
  TestBed.configureTestingModule({
    declarations: [PonyComponent]
  });
  const fixture = TestBed.createComponent(PonyComponent);

  // given a pony
  const ponyComponent = fixture.componentInstance;
  ponyComponent.pony = { name: 'Rainbow Dash', color: 'BLUE' };

  // we fake the event emitter with a spy
  spyOn(ponyComponent.ponyClicked, 'emit');

  // when we click on the pony
  const element = fixture.nativeElement;
  const image = element.querySelector('img');
  image.dispatchEvent(new Event('click'));

  // and we trigger the change detection
  fixture.detectChanges();

  // then the event emitter should have fired an event
  expect(ponyComponent.ponyClicked.emit).toHaveBeenCalled();
});

```

Examinons un autre composant :

```

@Component({
  selector: 'ns-race',
  template: `<div>
    <h1>{{race.name}}</h1>
    <ns-pony *ngFor="let currentPony of race.ponies" [pony]="currentPony"></ns-pony>
  </div>`
})
export class RaceComponent {

  @Input() race: any;

}

```

et son test :

```

describe('RaceComponent', () => {

  let fixture: ComponentFixture<RaceComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [RaceComponent, PonyComponent]
    });
    fixture = TestBed.createComponent(RaceComponent);
  });

  it('should have a name and a list of ponies', () => {
    // given a component instance with a race input initialized
    const raceComponent = fixture.componentInstance;
    raceComponent.race = { name: 'London', ponies: [{ name: 'Rainbow Dash', color: 'BLUE' }] };

    // when we trigger the change detection
    fixture.detectChanges();

    // then we should have a title with the race name
    const element = fixture.nativeElement;
    expect(element.querySelector('h1').textContent).toBe('London');

    // and a list of ponies
    const ponies = fixture.debugElement.queryAll(By.directive(PonyComponent));
    expect(ponies.length).toBe(1);
    // we can check if the pony is correctly initialized
    const rainbowDash = ponies[0].componentInstance.pony;
    expect(rainbowDash.name).toBe('Rainbow Dash');
  });
});

```

Ici on requête toutes les directives de type `PonyComponent` et on teste si le premier poney est correctement initialisé.

Tu peux récupérer les composants au sein de ton composant avec `children` ou les requéter avec `query()` et `queryAll()`. Ces méthodes prennent un prédicat en argument qui peut être soit `By.css`, soit `By.directive`. C'est ce qu'on fait pour obtenir les poneys affichés, parce qu'elles sont des instances de `PonyComponent`. Garde à l'esprit que c'est différent d'une requête DOM avec `querySelector()`: elle ne trouvera que les éléments gérés par Angular, et retourne un `ComponentFixture`, pas un élément du DOM (ce qui te donnera accès à `componentInstance` du résultat par exemple).

## 15.5. Tester avec des templates ou providers factices

Quand on teste un composant, on veut parfois créer un composant parent qui l'utilise. Et s'il y a plusieurs cas d'utilisation, on devrait créer plusieurs composants parents juste pour essayer les

différents *inputs* par exemple.

Heureusement, quand nous sommes dans un test, nous pouvons modifier n'importe quel composant pour le réutiliser dans différents tests, en surchargeant son template.

Pour ce faire, le `TestBed` propose une méthode `overrideComponent()`, à appeler avant `createComponent()`:

```
describe('RaceComponent', () => {  
  
  let fixture: ComponentFixture<RaceComponent>;  
  
  beforeEach(() => {  
    TestBed.configureTestingModule({  
      declarations: [RaceComponent, PonyComponent]  
    });  
    TestBed.overrideComponent(RaceComponent, { set: { template:  
      '<h2>{{race.name}}</h2>' } });  
    fixture = TestBed.createComponent(RaceComponent);  
  });  
  
  it('should have a name', () => {  
    // given a component instance with a race input initialized  
    const raceComponent = fixture.componentInstance;  
    raceComponent.race = { name: 'London' };  
  
    // when we trigger the change detection  
    fixture.detectChanges();  
  
    // then we should have a name  
    const element = fixture.nativeElement;  
    expect(element.querySelector('h2').textContent).toBe('London');  
  });  
});
```

Comme tu peux le voir, cette méthode prend deux arguments :

- le composant que tu veux surcharger ;
- la méta donnée que tu veux ajouter, supprimer ou remplacer (par exemple ici on remplace le template).

Cela signifie que tu peux modifier le template du composant que tu testes, ou celui d'un de ses enfants (pour remplacer le template complexe d'un composant par un plus simple).

`template` n'est pas la seule méta donnée disponible, tu peux aussi utiliser :

- `providers` pour remplacer les dépendances d'un composant ;
- `styles` pour remplacer le style utilisé dans le template d'un composant ;
- ou n'importe quelle propriété que tu peux passer au décorateur `@Component...`

Comme remplacer un template est le cas le plus courant, Angular 4 a introduit une nouvelle méthode `overrideTemplate()` :

```
describe('RaceComponent', () => {

  let fixture: ComponentFixture<RaceComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [RaceComponent, PonyComponent]
    });
    TestBed.overrideTemplate(RaceComponent, '<h2>{{race.name}}</h2>');
    fixture = TestBed.createComponent(RaceComponent);
  });

  it('should have a name', () => {
    // given a component instance with a race input initialized
    const raceComponent = fixture.componentInstance;
    raceComponent.race = { name: 'London' };

    // when we trigger the change detection
    fixture.detectChanges();

    // then we should have a name
    const element = fixture.nativeElement;
    expect(element.querySelector('h2').textContent).toBe('London');
  });
});
```

Maintenant tu es prêt pour tester ton application !

## 15.6. Tests end-to-end ("de bout en bout")

Les tests end-to-end ("de bout en bout", abrégés en *e2e*) sont l'autre type de tests qu'on peut faire. Un test end-to-end consiste à lancer réellement ton application dans un navigateur et à simuler l'interaction d'un utilisateur (clic sur les boutons, saisie de formulaires, etc.). Ils ont l'avantage de vraiment tester l'application dans son ensemble, mais :

- ils sont bien plus lents (plusieurs secondes par test) ;
- c'est souvent difficile de tester les cas limites.

Comme tu l'as deviné, tu n'as pas besoin de choisir entre les tests unitaires et les tests e2e : tu combineras les deux pour avoir à la fois une bonne couverture et quelques garanties que ton application complète fonctionne comme prévue.

Les tests e2e s'appuient sur un outil appelé [Protractor](#). C'est le même outil qu'on utilisait en AngularJS 1.x. Et la bonne nouvelle, c'est qu'il fonctionne de la même façon avec AngularJS 1.x et Angular !

Tu écriras ta suite de tests avec Jasmine comme pour un test unitaire, mais tu utiliseras l'API de Protractor pour interagir avec ton application.

Un test simple ressemblerait à :

```
describe('Home', () => {  
  it('should display title, tagline and logo', () => {  
    browser.get('/');  
    expect(element.all(by.css('img')).count()).toEqual(1);  
    expect($('h1').getText()).toContain('PonyRacer');  
    expect($('small').getText()).toBe('Always a pleasure to bet on ponies');  
  });  
});
```

Protractor nous fournit un objet `browser`, avec quelques méthodes utilitaires comme `get()` pour aller à une URL. Puis tu as `element.all()` pour sélectionner tous les éléments répondant à un prédicat donné. Ce prédicat s'appuie souvent sur `by` et ses méthodes variées (`by.css()` pour faire une requête CSS, `by.id()` pour récupérer un élément par son identifiant, etc.). `element.all()` retournera une promesse, avec une méthode spéciale `count()` utilisée dans le test ci-dessus.

`$(‘h1’)` est un raccourci, équivalent de `element(by.css(‘h1’))`. Il récupérera le premier élément correspondant à la requête CSS.

Tu peux utiliser plusieurs méthodes sur la promesse retournée par `$()`, comme `getText()` et `getAttribute()` pour récupérer des informations, ou `click()` et `sendKeys()` pour agir sur l'élément.

Ces tests peuvent être bien plus longs à écrire et à débugger (bien plus que des tests unitaires), mais ils sont vraiment utiles. Tu peux faire plein de trucs incroyables avec ces tests, comme tester dans plusieurs navigateurs, prendre une capture d'écran à chaque échec, etc.

Avec les tests unitaires et les tests e2e, tu as toutes les cartes en main pour construire une application solide et maintenable !

#### MISE EN PRATIQUE

Tous les exercices du Pack Pro viennent avec leurs tests unitaires ! Si tu veux en apprendre plus, nous t'encourageons fortement à leur jeter un œil : nous avons testé toutes les parties possibles de l'application (100% de couverture de code) ! À la fin tu auras des dizaines d'exemples de test, que tu pourras utiliser dans tes propres projets.

# Chapitre 16. Envoyer et recevoir des données par HTTP

Ce ne sera pas une surprise, mais une grande part de notre travail consiste à demander des données à un serveur, et à lui en envoyer d'autres.

Traditionnellement, on utilise HTTP, même si de nos jours il y a des alternatives, comme les WebSockets. Angular fournit un module http, mais ne te l'impose pas. Si tu préfères, tu peux utiliser ta bibliothèque HTTP favorite pour faire des requêtes asynchrones.

Une des nouvelles candidates est l'API `fetch`, pour le moment disponible sous forme de *polyfill*, mais elle devrait devenir standard dans les navigateurs. Tu peux parfaitement construire ton application en utilisant `fetch` ou une autre bibliothèque. En fait, c'était ce que je j'utilisais avant que la partie HTTP existe dans Angular. Elle fonctionne très bien, sans besoin d'aucune plomberie pour informer le framework de la réception de données et du besoin de déclencher le cycle de détection de changements (contrairement à AngularJS 1.x, où il fallait invoker `$scope.apply()` si tu utilisais une bibliothèque externe : c'est la magie d'Angular et de ses zones !).

Mais si tu es content du framework, tu utiliseras un petit module appelé `HttpModule`, codé par l'équipe interne. C'est un module indépendant, donc tu peux vraiment faire comme tu préfères. Note d'ailleurs qu'il ressemble beaucoup à la proposition de spécification de l'API Fetch.

Si tu veux l'utiliser, il te faudra utiliser les classes du package `@angular/http`.

Pourquoi préférer ce module à `fetch` ? La réponse est simple : pour les tests. Comme on le verra, le module Http permet de bouchonner ton serveur, et de retourner des réponses données. C'est vraiment, vraiment très utile.

Une dernière chose avant de plonger dans l'API : le module Http utilise largement le paradigme de la programmation réactive. Alors si tu as sauté le [chapitre Programmation réactive](#), c'est le bon moment d'y retourner et de le lire vraiment cette fois ;).

## 16.1. Obtenir des données

Le module Http propose un service nommé `Http` que tu peux t'injecter dans n'importe quel constructeur. Comme le service est fourni par un autre module, il te faut le rendre disponible à tes composants et services. Pour cela il faut importer le module `HttpModule` dans le module racine :

```

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 {
}

```

Quand cela est fait tu peux injecter le service `Http` partout où tu en as besoin :

```

@Component({
  selector: 'ponyracer-app',
  template: '<h1>PonyRacer</h1>'
})
export class PonyRacerAppComponent {

  constructor(private http: Http) {
  }

}

```

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`

Toutes ces méthodes sont en fait du simple sucre syntaxique, qui sous le capot utilisent la méthode `request`.

Si tu as déjà utilisé le service `$http` en AngularJS 1.x, tu dois te rappeler qu'il utilisait largement les promesses (*Promises*). Mais en Angular, toutes ces méthodes retournent un objet `Observable`.

L'utilisation des Observables apporte plusieurs avantages, comme la possibilité d'annuler une requête, de la retenter, d'en composer facilement plusieurs, etc.

Commençons par récupérer les courses enregistrées dans PonyRacer. On supposera qu'un serveur est déjà prêt et disponible, fournissant une belle API RESTful. Pour charger les courses, on enverra une requête GET sur une URL genre '`http://backend.url/api/races`'.

Généralement, l'URL de base de tes appels HTTP sera stockée dans une variable ou un service, que tu pourras facilement variabiliser selon ton environnement. Ou, si l'API REST est servie par le même serveur que l'application Angular, tu peux simplement utiliser une URL relative : '/api/races'.

Avec le service `Http`, une telle requête est triviale :

```
http.get('${baseUrl}/api/races')
```

Cela retourne un Observable, et tu peux donc t'y abonner pour obtenir la réponse. La réponse est un objet `Response`, avec quelques champs et méthodes bien pratiques. Tu peux ainsi facilement accéder au code `status`, au `headers`, etc.

```
http.get('${baseUrl}/api/races')
  .subscribe(response => {
    console.log(response.status); // logs 200
    console.log(response.headers); // logs []
  });
```

Évidemment, le corps de la réponse est la partie la plus intéressante. Mais pour y accéder, il te faudra utiliser une méthode :

- `text()` si tu t'attends à du texte;
- `json()` si tu t'attends à un objet JSON, le parsing étant fait pour toi.

```
http.get('${baseUrl}/api/races')
  .subscribe(response => {
    console.log(response.json());
    // logs the array of races
  });
```

Envoyer des données est aussi trivial. Il suffit d'appeler la méthode `post()`, avec l'URL et l'objet à poster :

```
// you currently need to stringify the object you send
http.post('${baseUrl}/api/races', newRace)
```

Je ne te montrerai pas les autres méthodes, je suis sûr que tu as compris le principe.

## 16.2. Transformer des données

Comme on obtient un objet Observable en réponse, n'oublie pas que tu peux facilement transformer les données.

Tu veux une liste des noms des courses ? Il suffit de requêter les courses, et de mapper (*map*) leur nom !

Pour faire cela, il faut actuellement importer l'opérateur `map` pour RxJS. L'équipe Angular ne veut pas inclure toute la library RxJS dans nos applications, il faut donc importer explicitement les opérateurs requis :

```
import 'rxjs/add/operator/map';
```

On peut ensuite l'utiliser :

```
http.get(`${baseUrl}/api/races`)
// extract json body
.map(res => res.json())
.subscribe(races => {
  // store the array of the races in the component
  this.races = races;
});
```

Ce genre de travail sera généralement réalisé dans un service dédié. J'ai tendance à créer un service, genre `RaceService`, où tout le travail sera réalisé. Ainsi, mon composant n'a qu'à s'abonner à la méthode de mon service, sans savoir ce qui est réalisé sous le capot.

```
raceService.list()
.subscribe(races => {
  // store the array of the races in the component
  this.races = races;
});
```

Tu peux aussi bénéficier de la puissance de RxJS pour par exemple retenter plusieurs fois une requête en échec.

```
raceService.list()
// if the request fails, retry 3 times
.retry(3)
.subscribe(races => {
  // store the array of the races in the component
  this.races = races;
});
```

## 16.3. Options avancées

Évidemment, tu peux ajuster tes requêtes plus finement. Chaque méthode accepte un objet `RequestOptions` en paramètre optionnel, où tu peux configurer ta requête. Quelques options sont vraiment pratiques, et tu peux tout surcharger dans la requête. Certaines de ces options ont exactement les même valeurs que celles proposées par l'API Fetch. L'option `url` est évidente, et surchargera l'URL de la requête. L'option `method` est le verbe HTTP à utiliser, comme `RequestMethod.Get`. Si tu veux construire une requête manuellement, tu peux écrire :

```
http.request(`${baseUrl}/api/races/3`, { method: RequestMethod.Get })
  .subscribe(response => {
    // will get the race with id 3
 });
```

L'option `params` représente les paramètres de recherche (que l'on appelle aussi *query string*) à ajouter à l'URL.

```
http.get(`${baseUrl}/api/races`, { params: { sort: 'ascending', page: 1 } })
  // will call the URL ${baseUrl}/api/races?sort=ascending&page=1
  .subscribe(response => {
    // will return the races sorted
    this.races = response.json();
 });
```

L'option `headers` est pratique pour ajouter quelques headers custom à ta requête. Cela est notamment nécessaire pour certaines techniques d'authentification, comme par exemple JSON Web Token :

```
const headers = new Headers();
headers.append('Authorization', `Bearer ${token}`);

http.get(`${baseUrl}/api/races`, { headers })
  .subscribe(response => {
    // will return the races visible for the authenticated user
    this.races = response.json();
 });
```

## 16.4. Jsonp

Pour te permettre d'utiliser leur API sans être bloqué 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 chargé, tu peux alors accéder à la valeur JSON contenue dans la réponse.

En plus du module `HttpModule`, il y a aussi un `JsonpModule`, qui fournit un service `Jsonp` qui facilite les interactions avec de telles APIs, et assure tout le travail de plomberie pour nous. Tout ce que tu as à faire est de fournir l'URL du service à appeler, et d'ajouter `JSONP_CALLBACK` comme valeur du paramètre de callback.

Dans l'exemple ci-dessous, je vais récupérer tous les dépôts publics de notre organisation Github via JSONP.

```
jsonp.get('https://api.github.com/orgs/Ninja-Squad/repos?callback=JSONP_CALLBACK')
// extract json
.map(res => res.json())
// extract data
.map(res => res.data)
.subscribe(response => {
  // will return the public repos of Ninja-Squad
  this.repos = response;
});
```

## 16.5. Tests

Nous avons maintenant un service appelant une API REST pour récupérer les courses. Comment testons-nous ce service ?

```
@Injectable()
export class RaceService {
  constructor(private http: Http) {}

  list() {
    return this.http.get('/api/races').map(res => res.json());
  }
}
```

Dans un test unitaire, on ne veut pas vraiment appeler le serveur HTTP : ce n'est pas ce que nous testons. Nous voulons faire un "faux" appel HTTP pour retourner de fausses données. Pour cela, nous pouvons remplacer la dépendance au service `Http` par une implémentation bouchonnée en utilisant une classe fournie par le framework appelée `MockBackend` :

```

import { async, TestBed } from '@angular/core/testing';
import { BaseRequestOptions, Response, RequestOptions, RequestMethod } from
  '@angular/http';
import { MockBackend, MockConnection } from '@angular/http/testing';

import 'rxjs/add/operator/map';

describe('RaceService', () => {

  let raceService;
  let mockBackend;

  beforeEach(() => TestBed.configureTestingModule({
    providers: [
      MockBackend,
      BaseRequestOptions,
      {
        provide: Http,
        useFactory: (backend, defaultOptions) => new Http(backend, defaultOptions),
        deps: [MockBackend, BaseRequestOptions]
      },
      RaceService
    ]
  }));
}

beforeEach(() => {
  raceService = TestBed.get(RaceService);
  mockBackend = TestBed.get(MockBackend);
});

it('should return an Observable of 2 races', async(() => {
  // fake response
  const hardcodedRaces = [new Race('London'), new Race('Lyon')];
  const response = new Response(new RequestOptions({ body: hardcodedRaces }));

  // on a the connection
  mockBackend.connections.subscribe((connection: MockConnection) => {
    // return the fake response when we receive a request
    connection.mockRespond(response);
  });

  // call the service
  raceService.list().subscribe(races => {
    // check that the returned array is deserialized as expected
    expect(races.length).toBe(2);
  });
}));
});

```

Et on peut également ajouter quelques assertions sur la requête HTTP sous-jacente :

```
// on a the connection
mockBackend.connections.subscribe((connection: MockConnection) => {
  // return the fake response when we receive a request
  connection.mockRespond(response);

  // check that the underlying HTTP request was correct
  expect(connection.request.method).toBe(RequestMethod.Get);
  expect(connection.request.url).toBe('/api/races');
});
```

Et le tour est joué !

## MISE EN PRATIQUE

Essaye notre exercice [Http 🐾](#) ! Nous avons préparé une API REST complète, prête à être utilisée. Récupérons donc les courses avec le service [Http](#) ! Plus tard, nous apprendrons comment appeler une API sécurisée avec un système d'authentification dans les exercices [Http avec authentification 🐾](#), [Parier sur un poney 🐾](#) et [Annuler un pari 🐾](#). Dans le même genre, nous utiliserons aussi les [WebSockets 🐾](#).

# Chapitre 17. Routeur

Il est classique de vouloir associer une URL à un état de l'application. En effet, on veut qu'un utilisateur puisse mettre une page en favori et y revenir plus tard, et ça donne une meilleure expérience en général.

La partie en charge de ce travail s'appelle un routeur, et chaque framework a le sien (voire même plusieurs).

Le routeur d'Angular a un objectif simple : permettre d'avoir des URLs compréhensibles qui reflètent l'état de notre application, et déterminer pour chaque URL quels composants initialiser et insérer dans la page. Tout cela sans rafraîchir la page et sans lancer de requête auprès de notre serveur : c'est tout l'intérêt d'avoir une *Single Page Application*.

Tu sais probablement qu'il y a un routeur natif dans AngularJS 1.x, maintenu par l'équipe du framework, dans un module nommé `ngRoute`. Tu sais aussi peut-être qu'il est très basique : il suffit pour des applications simples, mais il ne permet qu'une seule vue par URL, sans imbrication possible. C'est donc trop limité pour de grosses applications, où on a souvent des vues dans des vues. Il y a cependant un autre module très populaire dans la communauté, nommé `ui-router`, que beaucoup de monde utilise et qui fait du très bon boulot.

Avec Angular, l'équipe a décidé de réduire l'écart et a écrit un nouveau module `RouterModule`. Et ce module va probablement répondre à tous tes besoins !

Quelques-unes des ses nouvelles fonctionnalités sont vraiment intéressantes. Alors lançons-nous !

## 17.1. En route

Commençons à utiliser le routeur. C'est un module optionnel ; il n'est donc pas inclus dans le noyau du framework. Comme nous l'avons vu avec les autres modules, tu dois l'inclure dans le module racine si tu veux l'utiliser. Mais pour ça, nous avons besoin d'une configuration pour définir les associations entre les URLs et les composants. Cela peut se faire dans un fichier dédié, généralement nommé `app.routes.ts`, et contenant un tableau représentant la configuration :

```
import { Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { RacesComponent } from './races/races.component';

export const ROUTES: Routes = [
  { path: '', component: HomeComponent },
  { path: 'races', component: RacesComponent }
];
```

Ensuite nous devons importer le module routeur dans notre module racine, initialisé avec la bonne configuration :

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule } from '@angular/router';
import { ROUTES } from './app.routes';
import { HomeComponent } from './home/home.component';
import { RacesComponent } from './races/races.component';

@NgModule({
  imports: [BrowserModule, RouterModule.forRoot(ROUTES)],
  declarations: [PonyRacerAppComponent, HomeComponent, RacesComponent],
  bootstrap: [PonyRacerAppComponent]
})
export class AppModule {
}

```

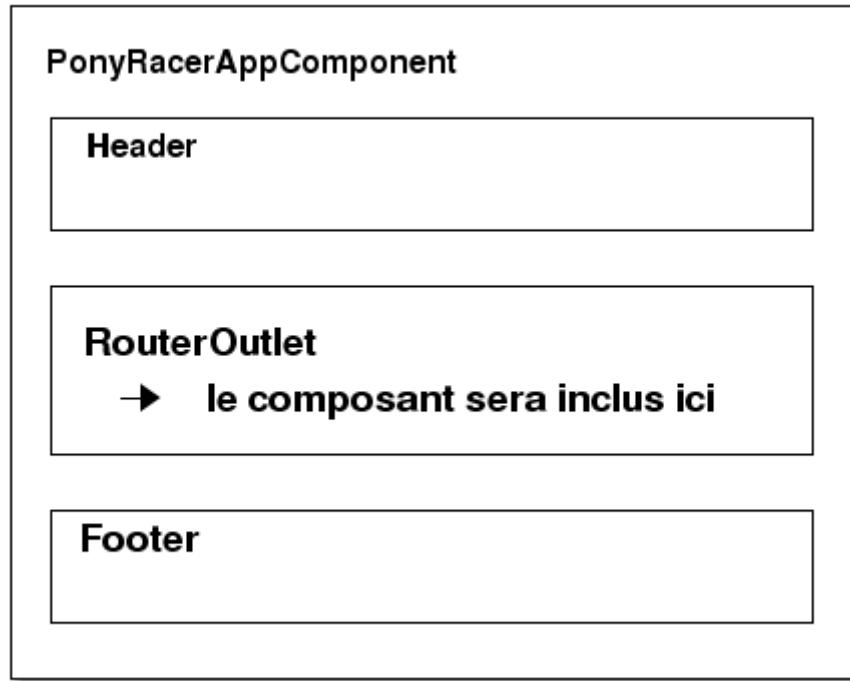
**NOTE**

Tu dois aussi déclarer tous les composants utilisés par le routeur dans les **declarations** du module racine.

Comme tu le vois, les **Routes** sont un tableau d'objets, chacun d'eux étant une... route. Une configuration de route est en général une paire :

- **path** : quelle URL va déclencher la navigation ;
- **component** : quel composant sera initialisé et affiché .

Tu te demandes peut-être où le composant sera inclus dans la page, et c'est une bonne question. Pour qu'un composant soit inclus dans notre application, comme le **RacesComponent** de l'exemple ci-dessus, il faut utiliser un tag spécial dans le template du composant principal : <**router-outlet**>.



C'est évidemment une directive Angular, dont le seul rôle est de marquer l'emplacement du template du composant de la route actuelle. Le template de notre application ressemblera donc à :

```

<header>
  <nav>...</nav>
</header>
<main>
  <router-outlet></router-outlet>
  <!-- the component's template will be inserted here-->
</main>
<footer>made with &lt;3 by Ninja Squad</footer>
  
```

Quand on naviguera, tout restera en place (le *header*, le *main*, et le *footer* ici), et le composant correspondant à la route actuelle sera inséré à la suite de la directive **RouterOutlet**.

## 17.2. Navigation

Comment peut-on naviguer entre différents composants ? Bien sûr, tu peux saisir manuellement l'URL correspondante et recharger la page, mais cela n'est pas très pratique. On ne veut pas non plus utiliser des liens "classiques", avec `<a href=""></a>`. En effet, un clic sur un tel lien entraîne un recharge de la page, et redémarre donc l'application Angular toute entière. Le but d'Angular est justement d'éviter ces rechargements : on construit une *Single Page Application*. Bien sûr, une solution existe.

Dans un template, tu peux insérer un lien avec la directive **RouterLink** pointant sur le chemin où tu veux aller. On peut utiliser cette directive car notre module racine importe **RouterModule**, rendant toutes les directives exportées par **RouterModule** disponibles dans le module racine. La directive

`RouterLink` peut recevoir soit une constante représentant le chemin vers lequel tu veux naviguer, soit un tableau de chaînes de caractères, représentant le chemin de la route et ses paramètres. Par exemple, dans le template de `RacesComponent`, si on veut aller sur `HomeComponent`, on peut imaginer écrire :

```
<a href="" routerLink="/">Home</a>
<!-- same as -->
<a href="" [routerLink]="['/']">Home</a>
```

A l'exécution, le lien `href` sera calculé par le routeur et pointera sur `/`.

#### NOTE

Le *slash* de début dans le chemin est nécessaire. S'il n'est pas inclus, `RouterLink` construit une URL relativement au chemin courant (ce qui peut être utile en cas de composants imbriqués, comme on le verra plus tard). Ajouter un *slash* indique que l'URL doit être calculée depuis l'URL de base de l'application.

La directive `RouterLink` peut être utilisée avec la directive `RouterLinkActive`, qui peut ajouter une classe CSS automatiquement si le lien pointe sur la route courante. Cela permet notamment de styler une entrée de menu comme active quand elle pointe sur la page courante.

```
<a href="" routerLink="/" routerLinkActive="selected-menu">Home</a>
```

On peut même récupérer une référence sur cette directive, pour savoir si la route est active et s'en servir dans le template :

```
<a href="" routerLink="/" routerLinkActive #route="routerLinkActive">Home {{ route.isActive ? '(here)' : '' }}</a>
```

Il est aussi possible de naviguer depuis le code, en utilisant le service `Router` et sa méthode `navigate()`. C'est souvent pratique quand on veut rediriger notre utilisateur suite à une action :

```
export class RacesComponent {
  constructor(private router: Router) {}

  saveAndMoveBackToHome() {
    // ... save logic ...
    this.router.navigate(['']);
  }
}
```

La méthode prend en paramètre un tableau dont le premier élément est le chemin vers lequel tu souhaites rediriger l'utilisateur.

Il est également possible d'avoir des paramètres dans l'URL, et c'est très pratique pour définir des

URLs dynamiques. Par exemple, on pourrait afficher une page de détail pour un poney, et cette page aurait une URL significative comme `ponies/id-du-poney/le-nom-du-poney`.

Pour ce faire, rien de compliqué. On définit une route dans la configuration avec un (ou plusieurs) paramètre(s) dynamique(s).

```
export const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'races', component: RacesComponent },
  { path: 'races/:raceId/ponies/:ponyId', component: PonyComponent }
];
```

On peut alors définir des liens dynamiques avec `routerLink` :

```
<a href="" [routerLink]="['/races', race.id, 'ponies', pony.id]">See pony</a>
```

Et bien sûr on peut récupérer ces paramètres assez facilement dans le composant cible. Ici, grâce à l'injection de dépendances, notre composant `PonyComponent` reçoit un objet du type `ActivatedRoute`. Cet objet peut être utilisé dans `ngOnInit`, et a un champ bien pratique : `snapshot`. Ce champ contient tous les paramètres de l'URL dans `paramMap` !

```
export class PonyComponent implements OnInit {
  pony: any;

  constructor(private ponyService: PonyService, private route: ActivatedRoute) {}

  ngOnInit() {
    const id = this.route.snapshot.paramMap.get('ponyId');
    this.ponyService.get(id).subscribe(pony => this.pony = pony);
  }
}
```

Ce hook est aussi le bon endroit pour faire travail d'initialisation du composant.

Comme tu l'as peut-être remarqué, nous utilisons `snapshot`. Cela veut-il dire qu'il y a une version non instantanée ? En effet. Et cela nous donne une façon de nous abonner aux changements de paramètre, avec, tu l'as deviné, un observable. Cet observable est appelé `paramMap`.

#### ATTENTION

Très important: le routeur va réutiliser le composant s'il le peut ! Supposons que notre composant a un bouton "Suivant" pour le poney suivant. Quand l'utilisateur cliquera sur le bouton, l'URL va changer de `/ponies/1` à `/ponies/2` par exemple. Le routeur va alors réutiliser notre instance de composant : cela veut dire que `ngOnInit` ne sera pas appellée à nouveau ! Si tu veux que ton composant se mette à jour pour ce genre de navigation, il n'y a pas d'autre choix que d'utiliser l'observable `paramMap` !

```

export class PonyReusableComponent implements OnInit {
  pony: any;

  constructor(private ponyService: PonyService, private route: ActivatedRoute) {}

  ngOnInit() {
    this.route.paramMap.subscribe((params: ParamMap) => {
      const id = params.get('ponyId');
      this.ponyService.get(id).subscribe(pony => this.pony = pony);
    });
  }
}

```

Ici nous nous abonnons à l'observable offert par `ActivatedRoute`. Maintenant, à chaque fois que l'URL changera de `/ponies/1` à `/ponies/2` par exemple, l'observable `paramMap` va émettre un événement, et nous pourrons récupérer le bon poney à afficher à l'écran.

Note qu'au lieu de nous abonner au résultat du `PonyService` dans l'abonnement des mises à jour des paramètres on peut utiliser l'opérateur, plus élégant, `switchMap` :

```

export class PonySwitchMapComponent implements OnInit {
  pony: any;

  constructor(private ponyService: PonyService, private route: ActivatedRoute) {}

  ngOnInit() {
    this.route.paramMap
      .map((params: ParamMap) => params.get('ponyId'))
      .switchMap(id => this.ponyService.get(id))
      .subscribe(pony => this.pony = pony);
  }
}

```

## MISE EN PRATIQUE

Essaye notre exercice [Routeur](#) 🚀 pour apprendre à configurer le routeur, naviguer entre composants, et tester tout ça.

# Chapitre 18. Formulaires

## ATTENTION

Ce chapitre utilise la nouvelle API de formulaire (du package `@angular/forms` qui est apparu en [rc.2](#)). Si tu utilises l'ancienne API maintenant dépréciée, c'est le moment de mettre à jour. Ce chapitre t'aidera à faire la migration !

## 18.1. Form, form, form-idable !

La gestion des formulaires a toujours été super soignée en Angular. C'était une des fonctionnalités les plus mises en avant en 1.x, et, comme toute application inclut en général des formulaires, elle a gagné le cœur de beaucoup de développeurs.

Les formulaires, c'est compliqué : tu dois valider les saisies de l'utilisateur, afficher les erreurs correspondantes, tu peux avoir des champs obligatoires ou non, ou qui dépendent d'un autre champ, tu veux pouvoir réagir sur les changements de certains, etc. On a aussi besoin de tester ces formulaires, et c'était impossible dans un test unitaire en AngularJS 1.x. C'était seulement possible avec un test end-to-end, ce qui peut être lent.

En Angular, le même soin a été appliqué aux formulaires, et le framework nous propose une façon élégante d'écrire nos formulaires. En fait, il en propose même deux !

Tu peux écrire ton formulaire en utilisant seulement des directives dans ton template : c'est la façon "pilotée par le template". D'après notre expérience, c'est particulièrement utile pour des formulaires simples, sans trop de validation.

L'autre façon de procéder est la façon "pilotée par le code", où tu écris une description du formulaire dans ton composant, puis utilises ensuite des directives pour lier ce formulaire aux inputs/textareas/selects de ton template. C'est plus verbeux, mais aussi plus puissant, notamment pour faire de la validation custom, ou pour générer des formulaires dynamiquement.

Alors prenons un cas d'utilisation, codons-le de chacune des deux façons, et étudions les différences.

On va écrire un formulaire simple, pour enregistrer de nouveaux utilisateurs dans notre merveilleuse application PonyRacer. Comme on a besoin d'un composant de base pour chacun des cas d'utilisation, commençons par celui-ci :

```

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

@Component({
  selector: 'ns-register',
  template: `
    <h2>Sign up</h2>
    <form></form>
  `
})
export class RegisterFormComponent {
}

```

Rien d'extraordinaire : un composant avec un simple template contenant un formulaire. Dans les minutes qui viennent, on y ajoutera les champs permettant d'enregistrer un utilisateur avec un nom et un mot de passe.

Quelque soit la méthode choisie (piloté par le template ou par le code), Angular crée une représentation de notre formulaire.

Dans la méthode "pilotée par le template", c'est automatique : on a juste besoin d'ajouter les bonnes directives dans le template et le framework s'occupe de créer la représentation du formulaire.

Dans la méthode "pilotée par le code", on crée cette représentation manuellement, et on lie ensuite la représentation du formulaire aux inputs en utilisant des directives.

Sous le capot, un champ du formulaire, comme un `input` ou un `select`, sera représenté par un `FormControl` en Angular. C'est la plus petite partie d'un formulaire, et il encapsule l'état du champ et sa valeur.

Un `FormControl` a plusieurs attributs :

- `valid`: si le champ est valide, au regard des contraintes et des validations qui lui sont appliquées.
- `invalid`: si le champ est invalide, au regard des contraintes et des validations qui lui sont appliquées.
- `errors`: un objet contenant les erreurs du champ.
- `dirty`: `false` jusqu'à ce que l'utilisateur modifie la valeur du champ.
- `pristine`: l'opposé de `dirty`.
- `touched`: `false` jusqu'à ce que l'utilisateur soit entré dans le champ.
- `untouched`: l'opposé de `touched`.
- `value`: la valeur du champ.
- `valueChanges`: un *Observable* qui émet à chaque changement sur le champ.

Il propose aussi quelques méthodes comme `hasError()` pour savoir si le contrôle a une erreur donnée.

Tu peux ainsi écrire des trucs comme :

```
const password = new FormControl();
console.log(password.dirty); // false until the user enters a value
console.log(password.value); // null until the user enters a value
console.log(password.hasError('required'));
```

Note que tu peux passer un paramètre au constructeur, qui deviendra sa valeur initiale.

```
const password = new FormControl('Cédric');
console.log(password.value); // logs "Cédric"
```

Ces contrôles peuvent être regroupés dans un **FormGroup** ("groupe de formulaire") pour constituer une partie du formulaire qui a des règles de validation communes. Un formulaire lui-même est un groupe de contrôle.

Un **FormGroup** a les mêmes propriétés qu'un **FormControl**, avec quelques différences :

- **valid** : si tous les champs sont valides, alors le groupe est valide.
- **invalid** : si l'un des champs est invalide, alors le groupe est invalide.
- **errors** : un objet contenant les erreurs du groupe, ou **null** si le groupe est entièrement valide. Chaque erreur en constitue la clé, et la valeur associée est un tableau contenant chaque contrôle affecté par cette erreur.
- **dirty** : **false** jusqu'à ce qu'un des contrôles devienne "dirty".
- **pristine** : l'opposé de **dirty**.
- **touched** : **false** jusqu'à ce qu'un des contrôles devienne "touched".
- **untouched** : l'opposé de **touched**.
- **value** : la valeur du groupe. Pour être plus précis, c'est un objet dont les clé/valeurs sont les contrôles et leur valeur respective.
- **valueChanges** : un **Observable** qui émet à chaque changement sur un contrôle du groupe.

Un groupe propose les mêmes méthodes qu'un **FormControl**, comme **hasError()**. Il a aussi une méthode **get()** pour récupérer un contrôle dans le groupe.

Tu peux en créer un comme cela :

```
const form = new FormGroup({
  username: new FormControl('Cédric'),
  password: new FormControl()
});
console.log(form.dirty); // logs false until the user enters a value
console.log(form.value); // logs Object {username: "Cédric", password: null}
console.log(form.get('username'));
```

Commençons avec un formulaire piloté par le template !

## 18.2. Formulaire piloté par le template

Dans cette méthode, on va mettre en œuvre un paquet de directives dans notre formulaire, et laisser le framework construire les instances de `FormControl` et `FormGroup` nécessaires. Par exemple, la directive `NgForm` qui transforme l'élément `<form>` en sa version Angular super-puissante : tu peux le voir comme la différence entre Bruce Wayne et Batman.

Toutes ces directives sont incluses dans le module `FormsModule`, nous devons donc l'importer dans notre module racine.

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

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

`FormsModule` contient les directives pour la façon "pilotée par le template". Nous verrons plus tard qu'il existe un autre module, `ReactiveFormsModule`, dans le même package `@angular/forms`, qui est nécessaire pour la façon "pilotée par le code".

Ajoutons un bouton *submit* :

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

@Component({
  selector: 'ns-register',
  template: `
    <h2>Sign up</h2>
    <form (ngSubmit)="register()">
      <button type="submit">Register</button>
    </form>
  `
})
export class RegisterFormComponent {
  register() {
    // we will have to handle the submission
  }
}
```

J'ai ajouté un bouton, et défini un handler d'événement `ngSubmit` sur l'élément `<form>`. L'événement

`ngSubmit` est émis par la directive `form` lors de la soumission du formulaire. Cela invoquera la méthode `register()` de notre composant, qu'on implémentera plus tard.

Une dernière chose : comme notre template va rapidement grossir, extrayons-le tout de suite dans un fichier dédié, grâce à `templateUrl` :

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

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  register() {
    // we will have to handle the submission
  }
}
```

Dans la façon "pilotée par le template", tu écris tes formulaires à peu près comme tu l'aurais fait en AngularJS 1.x, avec beaucoup de trucs dans ton template et peu dans ton composant.

Dans sa forme la plus simple, tu ajoutes simplement les directives `ngModel` dans le template, et c'est tout. La directive `ngModel` crée le `FormControl` pour toi, et le `<form>` crée automatiquement le `FormGroup`. Note que tu dois donner un `name` à l'input, qui sera utilisé par le framework pour construire le `FormGroup`.

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
  <div>
    <label>Username</label><input name="username" ngModel>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel>
  </div>
  <button type="submit">Register</button>
</form>
```

Maintenant, on doit évidemment faire quelque-chose pour la soumission, et récupérer les valeurs saisies. Pour cela, on va utiliser une variable locale et lui assigner l'objet `NgForm` créé par Angular pour le formulaire. Tu te rappelles le chapitre [Template](#) ? On va définir une variable, `userForm`, qui référencera le formulaire. C'est possible parce que la directive exporte l'instance de la directive `NgForm`, qui a les mêmes méthodes que la classe `FormGroup`. On verra comment exporter des données plus en détail quand on étudiera comment construire des directives.

```

<h2>Sign up</h2>
<!-- we use a local variable #userForm -->
<!-- and give its value to the register method -->
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel>
  </div>
  <button type="submit">Register</button>
</form>

```

Notre méthode `register` est désormais appelée avec la valeur du formulaire en paramètre :

```

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

@Component({
  selector: 'ns-register',
  templateUrl: 'register.component.html'
})
export class RegisterFormComponent {
  register(user) {
    console.log(user);
  }
}

```

C'est seulement du binding uni-directionnel cependant. Si tu mets à jour le champ, le modèle sera mis à jour, mais mettre à jour le modèle ne mettra pas à jour la valeur du champ. Mais `ngModel` est plus puissant qu'on ne le croit !

### 18.2.1. Binding bi-directionnel

Si tu as utilisé AngularJS 1.x, ou même juste lu un article dessus, tu as du voir le fameux exemple avec un champ et une expression affichant la valeur de celui-ci, qui se mettait à jour toute seule dès que l'utilisateur entrait une valeur dans le champ, et le champ se mettant à jour automatiquement dès que le modèle changeait. Le fameux "Binding Bi-directionnel", quelque chose comme :

```

<!-- AngularJS 1.x code example -->
<input type="text" ng-model="username">
<p>{{username}}</p>

```

On peut faire quelque chose de similaire en Angular.

Tu commences par définir un modèle de ce qui sera saisi dans le formulaire. On va faire cela dans une classe `User` :

```
class User {  
  username: string;  
  password: string;  
}
```

Notre `RegisterFormComponent` doit avoir un attribut `user` de type `User` :

```
import { Component } from '@angular/core';  
  
class User {  
  username: string;  
  password: string;  
}  
  
@Component({  
  selector: 'ns-register',  
  templateUrl: 'register-form.component.html',  
})  
export class RegisterFormComponent {  
  user = new User();  
  
  register() {  
    console.log(this.user);  
  }  
}
```

Comme tu peux le voir, la méthode `register()` trace désormais directement l'objet `user`.

On est donc prêt à ajouter les champs dans notre formulaire. Il nous faut lier les champs au modèle défini. Et pour cela, il y a donc la directive `ngModel` :

```
<h2>Sign up</h2>  
<form (ngSubmit)="register()">  
  <div>  
    <label>Username</label><input name="username" [(ngModel)]="user.username">  
  </div>  
  <div>  
    <label>Password</label><input type="password" name="password" [(ngModel)]="user.password">  
  </div>  
  <button type="submit">Register</button>  
</form>
```

Wow ! `[(ngModel)]` ? Qu'est-ce que c'est que ce truc ?! Ce n'est que du sucre syntaxique ajouté pour exprimer la même chose que :

```
<input name="username" [ngModel]="user.username" (ngModelChange)="user.username = $event">
```

La directive `NgModel` met à jour la valeur de l'`input` à chaque changement du modèle lié `user.username`, d'où la partie `[ngModel]="user.username"`. Et elle génère un événement depuis un output nommé `ngModelChange` à chaque fois que l'`input` est modifié par l'utilisateur, où l'événement est la nouvelle valeur, d'où la partie `(ngModelChange)="user.username = $event"`, qui met donc à jour le modèle `user.username` avec la valeur saisie.

Au lieu d'écrire cette forme verbeuse, on peut donc utiliser la syntaxe raccourcie `[]()`. Si, comme moi, tu as du mal à te rappeler si c'est `[]()` ou `([])`, il y a un très bon moyen mnémotechnique : c'est une boîte de bananes ! Mais si, regarde : le `[]` est une boîte, et, dedans, il y a deux bananes qui se regardent `()` !

Maintenant chaque fois qu'on tapera quelque chose dans le champ, le modèle sera mis à jour. Et si le modèle est mis à jour dans notre composant, le champ affichera automatiquement la nouvelle valeur :

```
<h2>Sign up</h2>
<form (ngSubmit)="register()">
  <div>
    <label>Username</label><input name="username" [(ngModel)]="user.username">
    <small>{{ user.username }} is an awesome username!</small>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" [(ngModel)]="user.password">
  </div>
  <button type="submit">Register</button>
</form>
```

Si tu essayes l'exemple ci-dessus, tu verras que le binding bi-directionnel fonctionne. Et notre formulaire aussi : si on le soumet, le composant tracera notre objet `user` !

## 18.3. Formulaire piloté par le code

En AngularJS 1.x, tu devais essentiellement construire tes formulaires dans tes templates. Angular introduit une déclaration impérative, qui permet de construire les formulaires programmatiquement.

Désormais, on peut manipuler les formulaires directement depuis le code. C'est plus verbeux mais plus puissant.

Pour construire un formulaire dans notre code, nous allons utiliser les abstractions dont nous avons parlé: `FormControl` et `FormGroup`.

Avec ces briques de bases on peut construire un formulaire dans notre composant. Mais au lieu

d'utiliser `new FormControl()` ou `new FormGroup()`, on va utiliser une classe utilitaire, `FormBuilder`, qu'on peut s'injecter :

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

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {

  constructor(fb: FormBuilder) {
    // we will have to build the form
  }

  register() {
    // we will have to handle the submission
  }
}
```

`FormBuilder` est une classe utilitaire, avec plein de méthodes bien pratiques pour créer des contrôles et des groupes. Faisons simple, et créons un petit formulaire avec deux contrôles, un nom d'utilisateur et un mot de passe.

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  userForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.userForm = fb.group({
      username: '',
      password: ''
    });
  }

  register() {
    // we will have to handle the submission
  }
}
```

On a créé un `<form>` avec deux contrôles. Tu peux voir que chaque contrôle est créé avec la valeur

''. C'est la même chose que d'utiliser la méthode `control()` du `FormBuilder` avec cette chaîne de caractères comme paramètre, et la même chose que d'appeler le constructeur `new FormControl('')` : la chaîne de caractères représente la valeur initiale que nous voulons afficher dans le formulaire. Ici elle est vide, donc l'input sera vide. Mais tu peux mettre une valeur bien sûr, si tu veux éditer une entité existante par exemple. La méthode utilitaire peut aussi recevoir d'autres attributs, que nous verrons plus tard.

On veut maintenant implémenter la méthode `register`. Comme on l'a vu, l'objet `FormGroup` a un attribut `value`, alors on peut facilement tracer son contenu avec :

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  userForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.userForm = fb.group({
      username: fb.control(''),
      password: fb.control('')
    });
  }

  register() {
    console.log(this.userForm.value);
  }
}
```

On veut maintenant affiner un peu le template. On va utiliser d'autres directives que celles que nous avons dans les formulaires "pilotés par le template". Ces directives sont dans le module `ReactiveFormsModule` que nous devons importer dans notre module racine. Leurs noms commencent par `form` au lieu de `ng` comme c'était le cas avec la façon "pilotée par le template".

Le formulaire doit être relié à notre objet `userForm`, grâce à la directive `formGroup`. Chaque champ de saisie est relié à un `FormControl`, grâce à la directive `formControlName` :

```
<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
  </div>
  <button type="submit">Register</button>
</form>
```

On utilise la notation avec crochets `[formGroup]="userForm"` pour relier notre objet `userForm` à `formGroup`. Chaque `input` reçoit la directive `formControlName` avec pour valeur le nom du contrôle auquel il est relié. Si tu indiques un nom qui n'existe pas, tu auras une erreur. Comme on passe une valeur (et pas une expression), on n'utilise pas les `[]` autour de `formControlName`.

Et c'est tout : un clic sur le bouton `submit` va tracer un objet contenant le nom d'utilisateur et le mot de passe choisi !

Si tu en as besoin, tu peux mettre à jour la valeur d'un `FormControl` depuis ton composant en utilisant `setValue()` :

```

import { Component } from '@angular/core';
import { FormBuilder, FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html',
})
export class RegisterFormComponent {
  usernameCtrl: FormControl;
  passwordCtrl: FormControl;
  userForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.usernameCtrl = fb.control('');
    this.passwordCtrl = fb.control('');
    this.userForm = fb.group({
      username: this.usernameCtrl,
      password: this.passwordCtrl
    });
  }

  reset() {
    this.usernameCtrl.setValue('');
    this.passwordCtrl.setValue('');
  }

  register() {
    console.log(this.userForm.value);
  }
}

```

## 18.4. Un peu de validation

La validation de données est traditionnellement une partie importante de la construction de formulaire. Certains champs sont obligatoires, certains dépendent d'autres, certains doivent respecter un format spécifique, certains ne doivent pas avoir de valeur plus grande ou plus petite que X, par exemple.

Commençons par ajouter quelques règles basiques : tous nos champs sont obligatoires.

### 18.4.1. Dans un formulaire piloté par le code

Pour spécifier que chaque champ est requis, on va utiliser `Validator`. Un validateur retourne une *map* des erreurs, ou `null` si aucune n'a été détectée.

Quelques validateurs sont fournis par le framework :

- `Validators.required` pour vérifier qu'un contrôle n'est pas vide ;

- `Validators.minLength(n)` pour s'assurer que la valeur entrée a au moins  $n$  caractères ;
- `Validators.maxLength(n)` pour s'assurer que la valeur entrée a au plus  $n$  caractères ;
- `Validators.email()` (disponible depuis la version 4.0) pour s'assurer que la valeur entrée est une adresse email valide (bon courage pour trouver l'expression régulière par vous-même...) ;
- `Validators.pattern(p)` pour s'assurer que la valeur entrée correspond à l'expression régulière  $p$  définie.

Les validateurs peuvent être multiples, en passant un tableau, et peuvent s'appliquer sur un `FormControl` ou un `FormGroup`. Comme on veut que tous les champs soient obligatoires, on peut ajouter le validator `required` sur chaque contrôle, et s'assurer que le nom de l'utilisateur fait 3 caractères au minimum.

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html',
})
export class RegisterFormComponent {
  userForm: FormGroup;

  constructor(fb: FormBuilder) {
    this.userForm = fb.group({
      username: fb.control('', [Validators.required, Validators.minLength(3)]),
      password: fb.control('', Validators.required)
    });
  }

  register() {
    console.log(this.userForm.value);
  }
}
```

#### 18.4.2. Dans un formulaire piloté par le template

Ajouter un champ requis dans un formulaire piloté par le template est aussi rapide. Il suffit de rajouter l'attribut `required` sur les `<input>`. `required` est une directive fournie par le framework, qui ajoutera automatiquement le validateur sur le champ. Même chose avec `minlength`, `maxlength` et `email`.

En partant de l'exemple de binding bi-directionnel :

```

<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel required minlength="3">
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required>
  </div>
  <button type="submit">Register</button>
</form>

```

Note que cela peut s'utiliser dans un formulaire piloté par le code également.

## 18.5. Erreurs et soumission

Evidemment, on veut que l'utilisateur ne puisse pas soumettre le formulaire tant qu'il reste des erreurs, et ces erreurs doivent être parfaitement affichées.

Si tu utilises les exemples plus haut, tu verras que même si les champs sont requis, on peut quand même soumettre le formulaire. Peut-être qu'on peut y faire quelque chose ?

On sait qu'on peut facilement désactiver un bouton avec l'attribut `disabled`, mais encore faut-il lui passer une expression qui reflète l'état de validité du formulaire courant.

### 18.5.1. Erreurs et soumission dans un formulaire piloté par le code

Nous avons ajouté un champ `userForm`, du type `FormGroup`, à notre composant. Ce champ fournit une vision complète de l'état du formulaire et de ses champs, incluant ses erreurs de validation.

Par exemple, on peut désactiver la soumission si le formulaire n'est pas valide :

```

<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>

```

Comme tu le vois dans la dernière ligne, il suffit de lier `disabled` à la propriété `invalid` du `userForm`.

Désormais, on ne pourra soumettre que si tous les contrôles sont valides. Pour informer l'utilisateur de la raison de cette impossibilité, il faut encore afficher les messages d'erreur.

Toujours grâce au `userForm`, on peut écrire :

```

<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
    <div *ngIf="userForm.get('username').hasError('required')">Username is required</div>
    <div *ngIf="userForm.get('username').hasError('minlength')">Username should be 3 characters min</div>
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    <div *ngIf="userForm.get('password').hasError('required')">Password is required</div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>

```

Cool ! Les erreurs sont désormais affichées si les champs sont vides, et elles disparaissent quand ils ont une valeur. Mais elles sont affichées dès l'apparition du formulaire. Peut-être qu'on devrait les masquer jusqu'à ce que l'utilisateur remplisse une valeur ?

```

<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
    <div *ngIf="userForm.get('username').dirty && userForm.get('username').hasError('required')">
      Username is required
    </div>
    <div *ngIf="userForm.get('username').dirty && userForm.get('username').hasError('minlength')">
      Username should be 3 characters min
    </div>
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    <div *ngIf="userForm.get('password').dirty && userForm.get('password').hasError('required')">
      Password is required
    </div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>

```

C'est un peu verbeux, mais on peut créer des références sur chaque contrôle dans le composant pour améliorer cela :

```

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  userForm: FormGroup;
  usernameCtrl: FormControl;
  passwordCtrl: FormControl;

  constructor(fb: FormBuilder) {
    this.usernameCtrl = fb.control('', Validators.required);
    this.passwordCtrl = fb.control('', Validators.required);

    this.userForm = fb.group({
      username: this.usernameCtrl,
      password: this.passwordCtrl
    });
  }

  register() {
    console.log(this.userForm.value);
  }
}

```

Et ensuite on utilise ces références dans le template :

```

<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
    <div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('required')>Username is required</div>
    <div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('minlength')>Username should be 3 characters min</div>
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    <div *ngIf="passwordCtrl.dirty && passwordCtrl.hasError('required')>Password is required</div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>

```

## 18.5.2. Erreurs et soumission dans un formulaire piloté par le template

Dans un formulaire piloté par le template, nous n'avons pas de champ dans notre composant référençant le `FormGroup`, mais nous avons déjà déclaré une variable locale dans le template, référençant l'objet `NgForm` exporté par la directive. Une fois encore, cette variable permet de

connaître l'état du formulaire et d'accéder aux champs.

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel required>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

Maintenant il nous faut afficher les erreurs sur chaque champ.

Comme le formulaire, chaque contrôle expose son objet `FormControl`, on peut donc créer une variable locale pour accéder aux erreurs :

```
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel required #username="ngModel">
      <div *ngIf="username.dirty && username.hasError('required')">Username is required</div>
    </div>
    <div>
      <label>Password</label><input type="password" name="password" ngModel required #password="ngModel">
        <div *ngIf="password.dirty && password.hasError('required')">Password is required</div>
      </div>
    <button type="submit" [disabled]="userForm.invalid">Register</button>
  </form>
```

Yeah !

## 18.6. Un peu de style

Quelle que soit la façon que tu as choisie pour créer tes formulaires, Angular réalise une autre tâche bien pratique pour nous : il ajoute et enlève automatiquement certaines classes CSS sur chaque champ (et le formulaire) pour nous permettre d'affiner le style visuel.

Par exemple, un champ aura la classe `ng-invalid` si un de ses validateurs échoue, ou `ng-valid` si tous ses validateurs passent. Cela signifie que tu peux facilement ajouter du style, comme la traditionnelle bordure rouge autour des champs invalides :

```

<style>
  input.ng-invalid {
    border: 3px red solid;
  }
</style>
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel required>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>

```

`ng-dirty` est une autre classe CSS utile, qui sera présente si l'utilisateur a modifié la valeur. Son contraire est `ng-pristine`, présente si l'utilisateur n'a jamais modifié la valeur. En général, je n'affiche une bordure rouge qu'une fois que l'utilisateur a modifié la valeur :

```

<style>
  input.ng-invalid.ng-dirty {
    border: 3px red solid;
  }
</style>
<h2>Sign up</h2>
<form (ngSubmit)="register(userForm.value)" #userForm="ngForm">
  <div>
    <label>Username</label><input name="username" ngModel required>
  </div>
  <div>
    <label>Password</label><input type="password" name="password" ngModel required>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>

```

Et enfin, il y a une dernière classe CSS : `ng-touched`. Elle sera présente si l'utilisateur est entré et sorti du champ au moins une fois (même s'il n'a rien modifié) Son contraire est `ng-untouched`.

Quand tu afficheras un formulaire pour la première fois, un champ portera généralement les classes CSS `ng-pristine ng-untouched ng-invalid`. Ensuite, quand l'utilisateur sera rentré puis sorti du champ, elle basculeront à `ng-pristine ng-touched ng-invalid`. Quand l'utilisateur modifiera la valeur, toujours invalide, on aura `ng-dirty ng-touched ng-invalid`. Et enfin, quand la valeur deviendra valide : `ng-dirty ng-touched ng-valid`.

## 18.7. Créer un validateur spécifique

Les courses de poneys sont très addictives (TRÈS), alors on ne peut s'inscrire que si on a plus de 18 ans. Et on veut que l'utilisateur entre son mot de passe deux fois, pour éviter toute erreur de saisie.

Comment fait-on cela ? En créant un validateur spécifique.

Pour ce faire, il suffit de créer une méthode qui accepte un `FormControl`, teste sa `value`, et retourne un objet avec les erreurs, ou `null` si la valeur est valide.

```
const isOldEnough = (control: FormControl) => {
  // control is a date input, so we can build the Date from the value
  const birthDatePlus18 = new Date(control.value);
  birthDatePlus18.setFullYear(birthDatePlus18.getFullYear() + 18);
  return birthDatePlus18 < new Date() ? null : { tooYoung: true };
};
```

Notre validation est plutôt simple : on prend la valeur du contrôle, on crée une date, on vérifie si le 18ème anniversaire est avant aujourd'hui, et on retourne une erreur avec la clé `tooYoung` ("trop jeune") sinon.

Maintenant il nous faut inclure ce validateur.

### 18.7.1. Utiliser un validateur dans un formulaire piloté par le code

On doit ajouter un nouveau contrôle dans notre formulaire, via le `FormBuilder` :

```

import { Component } from '@angular/core';
import { FormBuilder, FormControl, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  usernameCtrl: FormControl;
  passwordCtrl: FormControl;
  birthdateCtrl: FormControl;
  userForm: FormGroup;

  static isOldEnough(control: FormControl) {
    // control is a date input, so we can build the Date from the value
    const birthDatePlus18 = new Date(control.value);
    birthDatePlus18.setFullYear(birthDatePlus18.getFullYear() + 18);
    return birthDatePlus18 < new Date() ? null : { tooYoung: true };
  }

  constructor(fb: FormBuilder) {
    this.usernameCtrl = fb.control('', Validators.required);
    this.passwordCtrl = fb.control('', Validators.required);
    this.birthdateCtrl = fb.control('', [Validators.required, RegisterFormComponent
      .isOldEnough]);
    this.userForm = fb.group({
      username: this.usernameCtrl,
      password: this.passwordCtrl,
      birthdate: this.birthdateCtrl
    });
  }

  register() {
    console.log(this.userForm.value);
  }
}

```

Comme tu le vois, on a ajouté un nouveau contrôle `birthdate` ("date de naissance"), avec deux validateurs combinés. Le premier validateur est `required`, et le second est la méthode statique `isOldEnough` de notre classe. Bien sûr, la méthode pourrait appartenir à une autre classe (`required` est statique par exemple).

N'oublie pas d'ajouter le champ et d'afficher les erreurs dans le formulaire :

```

<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
    <div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('required')">Username is required</div>
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    <div *ngIf="passwordCtrl.dirty && passwordCtrl.hasError('required')">Password is required</div>
  </div>
  <div>
    <label>Birth date</label><input type="date" formControlName="birthdate">
    <div *ngIf="birthdateCtrl.dirty">
      <div *ngIf="birthdateCtrl.hasError('required')">Birth date is required</div>
      <div *ngIf="birthdateCtrl.hasError('tooYoung')">You're way too young to be betting on pony races</div>
    </div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>

```

Plutôt facile, non ?

A noter qu'il est possible de créer et d'ajouter des validateurs asynchrones également (pour vérifier auprès du backend si un nom d'utilisateur est disponible par exemple).

```

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  usernameCtrl: FormControl;
  userForm: FormGroup;

  constructor(fb: FormBuilder, private userService: UserService) {
    this.usernameCtrl = fb.control('', Validators.required, control => this
      .isUsernameAvailable(control));
    this.userForm = fb.group({
      username: this.usernameCtrl
    });
  }

  isUsernameAvailable(control: AbstractControl) {
    const username = control.value;
    return this.userService.isUsernameAvailable(username)
      .map(available => available ? null : { alreadyUsed: true });
  }

  register() {
    console.log(this.userForm.value);
  }
}

```

Le validateur asynchrone n'est cette fois pas une méthode statique car il accède au service du composant.

La méthode du service renvoie un `Observable` émettant soit `null` si il n'y a pas d'erreur (le username est disponible), soit un objet dont la clé sera le nom de l'erreur comme pour les validateurs synchrones.

Fonctionnalité intéressante, la classe `ng-pending` est ajoutée dynamiquement au champ tant que le validateur asynchrone n'aura pas terminé son travail. Cela permet par exemple d'afficher un spinner pour indiquer que la validation est en cours.

### 18.7.2. Utiliser un validateur dans un formulaire piloté par le template

Pour cela, on doit définir une directive custom qui s'applique sur un `input`, mais franchement c'est beaucoup plus simple en utilisant l'approche pilotée par le code...

Ou tu peux combiner le meilleur des deux mondes !

## 18.8. Combiner les approches pilotée par le code et pilotée par le template pour la validation

Tu peux tout faire dans le template, à l'exception de la validation custom ! Tu aurais quelque chose comme ça dans la vue :

```
<label>Username</label><input name="username" [FormControl]="usernameCtrl" [(ngModel)]="user.username" required>
<div class="error" *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('notAllowed')">Username is not allowed</div>
```

Et dans le composant :

```
usernameCtrl = new FormControl('', RegisterFormComponent.usernameValidator);

static usernameValidator(control: FormControl) {
  return control.value !== 'admin' ? null : { notAllowed: true };
}
```

C'est un très bon compromis :

- pas besoin de tout écrire comme dans l'approche pilotée par le code ;
- pas besoin de composer les validateurs `required` et `usernameAllowed` toi-même : Angular le fait pour toi ;
- tu as toujours le binding bi-directionnel, que l'on n'a pas dans l'approche pilotée par le code.

C'est une solution à garder en tête : combiner l'approche pilotée par le template pour les choses simples et le binding bi-directionnel, et l'approche pilotée par le code quand de la validation custom est nécessaire.

## 18.9. Regrouper des champs

Jusqu'à présent, on n'avait qu'un seul groupe : le formulaire global. Mais on peut déclarer des groupes au sein d'un groupe. C'est très pratique si tu veux valider un groupe de champs globalement, comme une adresse postale par exemple, ou, comme dans notre exemple, si tu veux vérifier que le mot de passe et sa confirmation sont bien identiques.

La solution est d'utiliser un formulaire piloté par le code (que l'on peut combiner avec un formulaire piloté par le template si besoin, comme au-dessus).

D'abord, on crée un nouveau groupe `passwordForm`, avec les deux champs, et on l'ajoute dans le groupe `userForm` :

```

import { Component } from '@angular/core';
import { FormBuilder, FormControl, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  passwordForm: FormGroup;
  userForm: FormGroup;
  usernameCtrl: FormControl;
  passwordCtrl: FormControl;
  confirmCtrl: FormControl;

  static passwordMatch(group: FormGroup) {
    const password = group.get('password').value;
    const confirm = group.get('confirm').value;
    return password === confirm ? null : { matchingError: true };
  }

  constructor(fb: FormBuilder) {
    this.usernameCtrl = fb.control('', Validators.required);
    this.passwordCtrl = fb.control('', Validators.required);
    this.confirmCtrl = fb.control('', Validators.required);

    this.passwordForm = fb.group(
      { password: this.passwordCtrl, confirm: this.confirmCtrl },
      { validator: RegisterFormComponent.passwordMatch }
    );

    this.userForm = fb.group({ username: this.usernameCtrl, passwordForm: this
      .passwordForm });
  }

  register() {
    console.log(this.userForm.value);
  }
}

```

Comme tu le vois, on a ajouté un validateur sur le groupe, `passwordMatch`, qui sera appelé à chaque fois qu'un des champs est modifié.

Mettons à jour le template pour refléter le nouveau formulaire, grâce à la directive `formGroupName` :

```

<h2>Sign up</h2>
<form (ngSubmit)="register()" [formGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
    <div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('required')">Username is required</div>
  </div>
  <div formGroupName="passwordForm">
    <div>
      <label>Password</label><input type="password" formControlName="password">
      <div *ngIf="passwordCtrl.dirty && passwordCtrl.hasError('required')">Password is required</div>
    </div>
    <div>
      <label>Confirm password</label><input type="password" formControlName="confirm">
      <div *ngIf="confirmCtrl.dirty && confirmCtrl.hasError('required')">Confirm your password</div>
    </div>
    <div *ngIf="passwordForm.dirty && passwordForm.hasError('matchingError')">Your password does not match</div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>

```

Et voilà !

## 18.10. Réagir aux modifications

Dernier bénéfice d'un formulaire piloté par le code : tu peux facilement réagir aux modifications, grâce à l'*observable valueChanges*. La programmation réactive pour les champions ! Par exemple, disons que notre champ de mot de passe doit afficher un indicateur de sécurité. On veut en calculer la robustesse lors de chaque changement du mot de passe :

```

import { Component } from '@angular/core';
import { FormBuilder, FormControl, FormGroup, Validators } from '@angular/forms';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';

@Component({
  selector: 'ns-register',
  templateUrl: 'register-form.component.html'
})
export class RegisterFormComponent {
  userForm: FormGroup;
  usernameCtrl: FormControl;
  passwordCtrl: FormControl;
  passwordStrength = 0;

  constructor(fb: FormBuilder) {
    this.usernameCtrl = fb.control('', Validators.required);
    this.passwordCtrl = fb.control('', Validators.required);

    this.userForm = fb.group({
      username: this.usernameCtrl,
      password: this.passwordCtrl
    });

    // we subscribe to every password change
    this.passwordCtrl.valueChanges
      // only recompute when the user stops typing for 400ms
      .debounceTime(400)
      // only recompute if the new value is different than the last
      .distinctUntilChanged()
      .subscribe(newValue => this.passwordStrength = newValue.length);
  }

  register() {
    console.log(this.userForm.value);
  }
}

```

Maintenant on a un attribut `passwordStrength` dans notre instance de composant, qu'on peut afficher à notre utilisateur :

```

<h2>Sign up</h2>
<form (ngSubmit)="register()" [FormGroup]="userForm">
  <div>
    <label>Username</label><input formControlName="username">
    <div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('required')>Username is required</div>
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    <div>Strength: {{passwordStrength}}</div>
    <div *ngIf="passwordCtrl.dirty && passwordCtrl.hasError('required')>Password is required</div>
  </div>
  <button type="submit" [disabled]="userForm.invalid">Register</button>
</form>

```

On utilise les opérateurs RxJS pour ajouter quelques fonctionnalités bien cools :

- `debounceTime(400)` émettra seulement des valeurs lorsque l'utilisateur arrête de taper pendant 400ms. Cela évite de devoir calculer la force du mot de passe à chaque valeur entrée par l'utilisateur. Cela peut être très intéressant si le calcul prend beaucoup de temps, ou lance une requête HTTP.
- `distinctUntilChanged()` émettra seulement des valeurs si la nouvelle valeur est différente de l'ancienne. Très intéressant également : imagine que l'utilisateur entre '`password`' puis s'arrête. On calcule la force du mot de passe. Puis il entre un nouveau caractère avant de l'effacer rapidement (en moins de 400ms). Le prochain événement qui sortira de `debounceTime` sera à nouveau '`password`'. Cela n'a pas de sens de recalculer la force du mot de passe à nouveau ! Cet opérateur n'émettra même pas la valeur, et nous économise un recalcul.

RxJS peut faire une tonne de travail pour toi : imagine si tu devais recoder toi-même ce que l'on a fait en deux lignes. Cela peut également se combiner facilement avec des requêtes HTTP, puisque le service `Http` utilise des observables également.

## 18.11. Conclusion

Angular propose deux façons de construire un formulaire :

- une en mettant tout en place dans le template. Mais, comme tu l'as vu, ça oblige à écrire des directives custom pour la validation, et c'est plus dur à tester. Cette façon est donc plus adaptée aux formulaires simples, qui n'ont par exemple qu'un seul ou quelques champs, et nous donne du binding bi-directionnel.
- une en mettant quasiment tout en place dans le composant. Cette façon est plus adaptée à la validation et au test, avec plusieurs niveaux de groupes si besoin. C'est donc le bon choix pour construire des formulaires complexes. Et tu peux même réagir aux changements d'un groupe, ou d'un champ.
- combiner à la fois la verbosité réduite de l'approche pilotée par le template, et la puissance de l'approche pilotée par le code pour la validation.

C'est peut-être l'approche la plus pragmatique : commence avec une approche orientée template et le binding bi-directionnel si ça te plaît, et dès que tu as besoin d'accéder à un champ ou un groupe de champs, pour par exemple ajouter de la validation custom ou de la programmation réactive, alors tu peux déclarer ce dont tu as besoin dans le composant, et lier les inputs et divs à ces déclarations avec les directives adéquates.

## MISE EN PRATIQUE

Essaye nos exercices [Formulaire d'enregistrement](#) 🐾, [Validateurs custom](#) 🐾 et [Formulaire de login](#) 🐾. Tu apprendras à construire un simple formulaire en utilisant la technique "pilotée par le code" et un autre avec la technique "pilotée par le template". Tu apprendras également à écrire des validateurs custom, comment tester les formulaires et authentifier tes utilisateurs !

# Chapitre 19. Les Zones et la magie d'Angular

Développer avec AngularJS 1.X dégageait un sentiment de "magie", et Angular procure toujours ce même effet : tu entres une valeur dans un `input` et hop!, magiquement, toute la page se met à jour en conséquence.

J'adore la magie, mais je préfère comprendre comment fonctionnent les outils que j'utilise. Si tu es comme moi, je pense que cette partie devrait t'intéresser : on va se pencher sur le fonctionnement interne d'Angular !

Mais d'abord, laisse moi t'expliquer comment fonctionne AngularJS 1.x, ce qui devrait être intéressant, même si tu n'en as jamais fait.

Tous les frameworks JavaScript fonctionnent d'une façon assez similaire : ils aident le développeur à réagir aux événements de l'application, à mettre à jour son état et à rafraîchir la page (le DOM) en conséquence. Mais ils n'ont pas forcément tous le même moyen d'y parvenir.

EmberJS par exemple, demande aux développeurs d'utiliser des *setters* sur les objets manipulés pour que le framework puisse "intercepter" l'appel au setter et connaître ainsi les changements appliqués au modèle, et pouvoir modifier le DOM en conséquence. React, lui, a opté pour recalculer tout le DOM à chaque changement mais, comme mettre à jour tout le DOM est une opération coûteuse, il fait d'abord ce rendu dans un DOM virtuel, puis n'applique que la différence entre le DOM virtuel et le DOM réel.

Angular, lui, n'utilise pas de *setter*, ni de DOM virtuel. Alors comment fait-il pour savoir ce qui doit être mis à jour ?

## 19.1. AngularJS 1.x et le *digest cycle*

La première étape est donc de détecter une modification du modèle. Un changement est forcément déclenché par un événement provenant soit directement de l'utilisateur (par exemple un clic sur un bouton, ou une valeur entrée dans un champ de formulaire) soit par le code applicatif (une réponse HTTP, une exécution de méthode asynchrone après un `timeout`, etc.).

Comment fait donc le framework pour savoir qu'un événement est survenu ? Et bien c'est simple, il nous force à utiliser ses directives, par exemple `ng-click` pour surveiller un clic, ou `ng-model` pour surveiller un `input`, et ses services, par exemple `$http` pour les appels HTTP ou `$timeout` pour exécuter du code asynchrone.

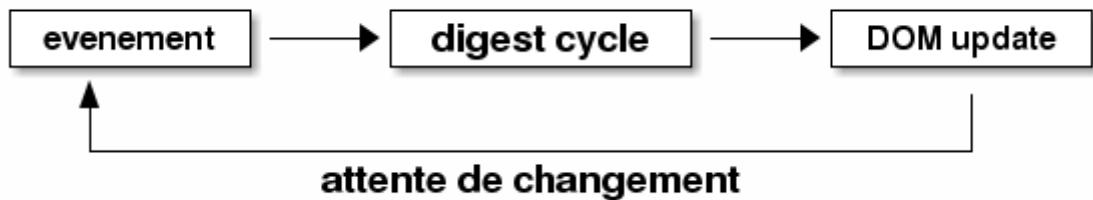
Le fait d'utiliser ses directives et services permet au framework d'être parfaitement informé du fait qu'un événement vient de se produire. C'est ça la première partie de la magie ! Et c'est cette première partie qui va déclencher la seconde : il faut maintenant que le framework analyse le changement qui vient de survenir, et puisse déterminer quelle partie du DOM doit être mise à jour.

Pour cela, en version 1.x, le framework maintient une liste de *watchers* (des *observateurs*) qui représente la liste de ce qu'il doit surveiller. Pour simplifier, un *watcher* est créé pour chaque expression dynamique utilisée dans un template. Il y a donc un *watcher* pour chaque petit bout dynamique de l'application, et on peut donc avoir facilement plusieurs centaines de *watchers* dans une page.

Ces *watchers* ont un rôle central dans AngularJS 1.x : ils sont la mémoire du framework pour connaître l'état de notre application.

Ensuite, à chaque fois que le framework détecte un changement, il déclenche ce que l'on appelle le *digest* (la *digestion*).

Ce *digest* évalue alors toutes les expressions stockées dans les *watchers* et compare leur nouvelle valeur avec leur ancienne valeur. Si un changement est constaté entre la nouvelle valeur d'une expression et son ancienne valeur, alors le framework sait que l'expression en question doit être remplacée par sa nouvelle valeur dans le DOM pour refléter ce changement. Cette technique est ce que l'on appelle du *dirty checking*.



Pendant ce *digest*, AngularJS parcourt toute la liste des *watchers*, et évalue chacun d'eux pour connaître la nouvelle valeur de l'expression surveillée. Avec une subtilité de taille : ce cycle va être effectué dans son intégralité tant que les résultats de tous les *watchers* ne sont pas stables, c'est à dire tant que la dernière valeur calculée n'est pas la même que la nouvelle valeur. Car bien sûr, un *watcher*, lorsqu'il détecte un changement de valeur de l'expression qu'il observe, déclenche un callback. Et ce callback peut lui-même modifier à nouveau le modèle, et donc changer la valeur d'une ou plusieurs expressions surveillées !

Prenons un exemple minimaliste : une page avec deux champs à remplir par l'utilisateur, son nom et son mot de passe, et un indice de robustesse du mot de passe. Un *watcher* est utilisé pour surveiller les changements du mot de passe, et recalculer sa robustesse chaque fois qu'il change.

Nous avons alors cette liste de *watchers* après la première itération du cycle de *digest*, lorsque l'utilisateur a saisi le premier caractère de son mot de passe :

```
$$watchers (expression -> value)
- "user.name" -> "Cédric"
- "user.password" -> "h"
- "passwordStrength" -> 0
```

La fonction de callback du *watcher* qui observe le mot de passe est alors appelée, et elle calcule la nouvelle valeur de `passwordStrength` : 3.

Mais Angular n'a aucune idée des changements appliqués au modèle. Il lance donc une deuxième itération pour savoir si le modèle est stable.

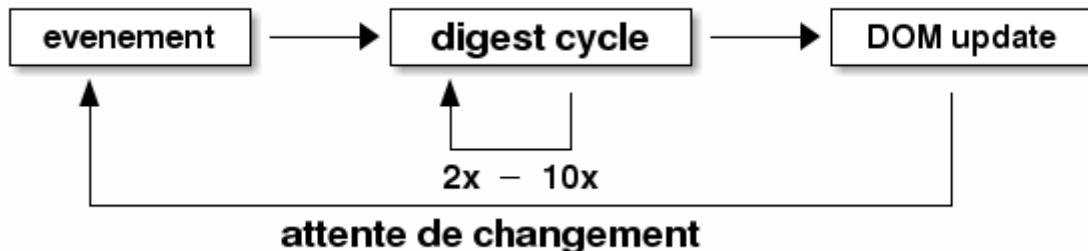
```
$$watchers
- "user.name" -> "Cédric"
- "user.password" -> "h"
- "passwordStrength" -> 3
```

Ce n'est pas le cas : la valeur de `passwordStrength` a changé depuis la première itération. Une nouvelle itération est donc lancée.

```
$$watchers
- "user.name" -> "Cédric"
- "user.password" -> "h"
- "passwordStrength" -> 3
```

Cette fois, c'est stable. C'est seulement à ce moment-là qu'AngularJS 1.x applique les résultats sur le DOM. Cette boucle de `digest` est donc jouée au moins 2 fois à chaque changement dans l'application. Elle peut être jouée jusqu'à 10 fois mais pas plus : après 10 cycles, si les résultats ne sont toujours pas stables, le framework considère qu'il y a une boucle infinie et lance une exception.

Donc mon schéma précédent ressemble en fait plus à :



Et j'insiste : c'est ce qui se passe après chaque événement de notre application. Cela veut dire que si l'utilisateur entre 5 caractères dans un champ, le `digest` sera lancé 5 fois, avec 3 boucles à chaque fois dans notre petit exemple, soit 15 boucles d'exécution. Dans une application réelle, on peut avoir facilement plusieurs centaines de `watchers` et donc plusieurs milliers d'évaluation d'expression à chaque événement.

Même si ça semble fou, cela marche très bien car les navigateurs modernes sont vraiment rapides, et qu'il est possible d'optimiser deux ou trois choses si nécessaire.

Tout cela signifie deux choses pour AngularJS 1.x :

- il faut utiliser les services et les directives du framework pour tout ce qui peut déclencher un changement ;
- modifier le modèle à la suite d'un événement non géré par Angular nous oblige à déclencher nous-même le mécanisme de détection de changement (en ajoutant le fameux `$scope.$apply()` qui lance le `digest`). Par exemple, si l'on veut faire une requête HTTP sans utiliser le service `$http`, il faut appeler `$scope.$apply()` dans notre callback de réponse pour dire au framework :

"Hé, j'ai des nouvelles données, lance le *digest* s'il te plaît !".

Et la magie du framework peut être scindée en deux parties :

- le déclenchement de la détection de changement à chaque événement ;
- la détection de changement elle-même, grâce au *digest*, et à ses multiples cycles.

Maintenant voyons comment Angular fonctionne, et quelle est la différence.

## 19.2. Angular et les zones

Angular conserve les mêmes principes, mais les implémente d'une façon différente, et on pourrait même dire, plus intelligente.

Pour la première partie du problème — le déclenchement de la détection de changement — l'équipe Angular a construit un petit projet annexe appelé [Zone.js](#). Ce projet n'est pas forcément lié à Angular, car les *zones* sont un outil qui peut être utilisé dans d'autres projets. Les *zones* ne sont pas vraiment un nouveau concept : elles existent dans le langage Dart (un autre projet Google) depuis quelques temps déjà. Elles ont aussi quelques similarités avec les *Domains* de Node.js (abandonnés depuis) ou les *ThreadLocal* en Java.

Mais c'est probablement la première fois que l'on voit les *Zones* en JavaScript : pas d'inquiétude, on va les découvrir ensemble.

### 19.2.1. Zones

Une zone est un contexte d'exécution. Ce contexte va recevoir du code à exécuter, et ce code peut être synchrone ou asynchrone. Une zone va nous apporter quelques petits bonus :

- une façon d'exécuter du code avant et après le code reçu à exécuter ;
- une façon d'intercepter les erreurs éventuelles d'exécution du code reçu ;
- une façon de stocker des variables locales à ce contexte.

Prenons un exemple. Si j'ai du code dans une application qui ressemble à :

```
// score computation -> synchronous
const score = computeScore();
// player score update -> synchronous
updatePlayer(player, score);
```

Quand on exécute ce code, on obtient :

```
computeScore: new score: 1000
updatePlayer: player 1 has 1000 points
```

Admettons que je veuille savoir combien de temps prend un tel code. Je peux faire quelque chose comme ça :

```
startTimer();
const score = computeScore();
updatePlayer(player, score);
stopTimer();
```

Et cela produirait ce résultat :

```
start
computeScore: new score: 1000
updatePlayer: player 1 has 1000 points
stop: 12ms
```

Facile. Mais maintenant, que se passe-t-il si `updatePlayer` est une fonction asynchrone ? JavaScript fonctionne de façon assez particulière : les opérations asynchrones sont placées à la fin de la queue d'exécution, et seront donc exécutées après les opérations synchrones.

Donc cette fois mon code précédent :

```
startTimer();
const score = computeScore();
updatePlayer(player, score); // asynchronous
stopTimer();
```

va en fait donner :

```
start
computeScore: new score: 1000
stop: 5ms
updatePlayer: player 1 has 1000 points
```

Mon temps d'exécution n'est plus bon du tout, vu qu'il ne mesure que le code synchrone ! Et c'est par exemple là que les zones peuvent être utiles. On va lancer le code en question en utilisant `zone.js` pour l'exécuter dans une zone :

```
const scoreZone = Zone.current.fork({ name: 'scoreZone' });
scoreZone.run(() => {
  const score = computeScore();
  updatePlayer(player, score); // asynchronous
});
```

Pourquoi est-ce que cela nous aide dans notre cas ? Hé bien, si la librairie `zone.js` est chargée dans notre navigateur, elle va commencer par patcher toutes les méthodes asynchrones de celui-ci. Donc à chaque fois que l'on fera un `setTimeout()`, un `setInterval()`, que l'on utilisera une API asynchrone comme les Promises, XMLHttpRequest, WebSocket, FileReader, Geolocation... on appellera en fait

la version patchée de zone.js. Zone.js connaît alors exactement quand le code asynchrone est terminé, et permet aux développeurs comme nous d'exécuter du code à ce moment là, par le biais d'un *hook*.

Une zone offre plusieurs *hooks* possibles :

- `onInvoke` qui sera appelé avant l'exécution du code encapsulé dans la zone ;
- `onHasTask` qui sera appelé après l'exécution du code encapsulé dans la zone ;
- `onHandleError` qui sera appelé dès que l'exécution du code encapsulé dans la zone lance une erreur ;
- `onFork` qui sera appelé à la création de la zone.

On peut donc utiliser une zone et ses hooks pour mesurer le temps d'exécution de mon code asynchrone :

```
const scoreZone = Zone.current.fork({
  name: 'scoreZone',
  onInvoke(delegate, current, target, task, applyThis, applyArgs, source) {
    // start the timer
    startTimer();
    return delegate.invoke(target, task, applyThis, applyArgs, source);
  },
  onHasTask(delegate, current, target, hasTaskState) {
    delegate.hasTask(target, hasTaskState);
    if (!hasTaskState.macroTask) {
      // if the zone run is done, stop the timer
      stopTimer();
    }
  }
});
scoreZone.run(() => {
  const score = computeScore();
  updatePlayer(player, score);
});
```

Et cette fois-ci ça marche !

```
start
computeScore: new score: 1000
updatePlayer: player 1 has 1000 points
stop: 12ms
```

Vous voyez maintenant peut-être le lien qu'il peut y avoir avec Angular. En effet, le premier problème du framework est de savoir quand la détection de changement doit être lancée. En utilisant les zones, et en faisant tourner le code que nous écrivons dans une zone, le framework a une très bonne vue de ce qu'il est en train de se passer. Il est ainsi capable de gérer les erreurs assez finement, mais surtout de lancer la détection de changement dès qu'un appel asynchrone est

terminé !

Pour simplifier, Angular fait quelque chose comme :

```
const angularZone = Zone.current.fork({
  name: 'angular',
  onHasTask: triggerChangeDetection
});
angularZone.run(() => {
  // your application code
});
```

Et le premier problème est ainsi résolu ! C'est pour cela qu'en Angular, contrairement à AngularJS 1.x, il n'est pas nécessaire d'utiliser des services spéciaux pour profiter de la détection de changements. Vous pouvez utiliser ce que vous voulez, les zones se chargeront du reste !

A noter que les zones sont en voie de standardisation, et pourraient faire partie de la spécification officielle ECMAScript dans un futur proche. Autre information intéressante, l'implémentation actuelle de zone.js embarque également des informations pour WTF (qui ne veut pas dire *What The Fuck* ici, mais [Web Tracing Framework](#)). Cette librairie permet de profiler votre application en mode développement, et de savoir exactement quel temps a été passé dans chaque partie de votre application et du framework. Bref, plein d'outils pour analyser les performances si besoin !

### 19.2.2. La détection de changement en Angular

La seconde partie du problème est la détection de changement en elle-même. C'est bien beau de savoir quand on doit la lancer, mais comment fonctionne-t-elle ?

Tout d'abord, il faut se rappeler qu'une application Angular est un arbre de composants. Lorsque la détection de changement se lance, le framework va parcourir l'arbre de ces composants pour voir si les composants ont subi des changements qui impactent leurs templates. Si c'est le cas, le DOM du composant en question sera mis à jour (seulement la petite portion impactée par le changement, pas le composant en entier). Ce parcours d'arbre se fait de la racine vers les enfants, et contrairement à AngularJS 1.x, il ne se fait qu'une seule fois. Car il y a maintenant une grande différence : la détection de changement en Angular ne change pas le modèle de l'application, là où un *watcher* en AngularJS 1.x pouvait changer le modèle lors de cette phase. Et en Angular, un composant ne peut maintenant modifier que le modèle de ses composants enfants et pas de son parent. Finis les changements de modèle en cascade !

La détection de changement est donc seulement là pour vérifier les changements et modifier le DOM en conséquence. Il n'y a plus besoin de faire plusieurs passages comme c'était le cas dans la version 1.x, puisque le modèle n'aura pas changé !

Pour être tout à fait exact, ce parcours se fait deux fois lorsque l'on est en mode développement pour vérifier qu'il n'y a pas d'effet de bords indésirables (par exemple un composant enfant modifiant le modèle utilisé par son composant parent). Si le second passage détecte un changement, une exception est lancée pour avertir le développeur.

Ce fonctionnement a plusieurs avantages :

- il est plus facile de raisonner sur nos applications, car on ne peut plus avoir de cas où un composant parent passe des informations à un composant enfant qui lui aussi passe des informations à son parent. Maintenant les données sont transmises dans un seul sens ;
- la détection de changement ne peut plus avoir de boucle infinie ;
- la détection de changement est bien plus rapide.

Sur ce dernier point, c'est assez simple à visualiser : là où précédemment la version effectuait ( $M$  watchers) \* ( $N$  cycles) vérifications, la version 2 ne fait plus que  $M$  vérifications.

Mais un autre paramètre entre en compte dans l'amélioration des performances d'Angular : le temps qu'il faut au framework pour faire cette vérification. Là encore, l'équipe de Google fait parler sa connaissance profonde des sciences informatiques et des machines virtuelles.

Pour cela, il faut se pencher sur la façon dont sont comparées deux valeurs en AngularJS 1.x et en Angular. Dans la version 1.x, le mécanisme est très générique : il y a une méthode dans le framework qui est appelée pour chaque *watcher* et qui est capable de comparer l'ancienne valeur et la nouvelle. Seulement, les machines virtuelles, comme celle qui exécute le code JavaScript dans notre navigateur (V8 si tu utilises Google Chrome par exemple), n'aiment pas vraiment le code générique.

Et si tu le permets, je vais faire une petite parenthèse sur le fonctionnement des machines virtuelles. Avoue que tu ne t'y attendais pas dans un article sur un framework JavaScript ! Les machines virtuelles sont des programmes assez extraordinaires : on leur donne un bout de code et elles sont capables de l'exécuter sur n'importe quelle machine. Vu que peu d'entre nous (certainement pas moi) sont capables de produire du code machine performant, c'est quand même assez pratique. On code avec notre language de haut niveau, et on laisse la VM se préoccuper du reste. Evidemment, les VMs ne se contentent pas de traduire le code, elles vont aussi chercher à l'optimiser. Et elles sont plutôt fortes à ce jeu là, à tel point que les meilleures VMs ont des performances aussi bonnes que du code machine optimisé (voire bien meilleures, car elle peuvent profiter d'informations au runtime, qu'il est plus difficile voire impossible de connaître à l'avance quand on fait l'optimisation à la compilation).

Pour améliorer les performances, les machines virtuelles, notamment celles qui font tourner des langages dynamiques comme JavaScript, utilisent un concept nommé *inline caching*. C'est une technique très ancienne (inventée pour SmallTalk je crois, soit près de 40 ans, une éternité en informatique), pour un principe finalement assez simple : si un programme appelle une méthode beaucoup de fois avec le même type d'objet, la VM devrait se rappeler de quelle façon elle évalue les propriétés des objets en question. Il y a donc un cache qui est créé, d'où le nom, *inline caching*. La VM commence donc par regarder dans le cache si elle connaît le type d'objet qu'elle reçoit, et si c'est le cas, utilise la méthode optimisée de chargement.

Ce genre de cache ne fonctionne vraiment que si les arguments de la méthode ont la même forme. Par exemple `{name: 'Cédric'}` et `{name: 'Cyril'}` ont la même forme. Par contre `{name: 'JB', skills: []}` n'a pas la même forme. Lorsque les arguments ont toujours la même forme, on dit que le cache est *monomorphe*, un bien grand mot pour dire qu'il n'a qu'une seule entrée, ce qui donne des résultats très rapides. Si il a quelques entrées, on dit qu'il est *polymorphe*, cela veut dire que la méthode peut être appelée avec des types d'objets différents, et le code est un peu plus lent. Enfin, il arrive que la VM laisse tomber le cache s'il y a trop de types d'objet différents, c'est ce

qu'on appelle un état *mégamorphique*. Et tu l'as compris, c'est le cas le moins performant.

Si j'en reviens à notre détection de changement en AngularJS 1.x, on comprend vite que la méthode générique utilisée n'est pas optimisable par la machine virtuelle : on est dans un état mégamorphique, là où le code est le plus lent. Et même si les navigateurs et machines modernes permettent de faire plusieurs milliers de vérification de watchers par seconde, on pouvait quand même atteindre les limites.

D'où l'idée de faire un peu différemment en Angular ! Cette fois, plutôt qu'avoir une seule méthode capable de comparer tous les types d'objet, l'équipe Google a pris le parti de générer dynamiquement des comparateurs pour chaque type. Cela veut dire qu'au démarrage de l'application, le framework va parcourir l'arbre des composants et générer un arbre de *ChangeDetectors* spécifiques.

Par exemple, pour un composant `User` avec un champ `name` affiché dans le template, on aura un `ChangeDetector` qui ressemble à :

```
class User_ChangeDetector {
  detectChanges() {
    if (this.name !== this.previousName) {
      this.previousName = this.name;
      hasChanged = true;
    }
  }
}
```

Un peu comme si on avait écrit le code de comparaison à la main. Ce code est du coup très rapide (monomorphique si vous suivez), permet de savoir si le composant a changé, et donc de mettre à jour le DOM en conséquence.

Donc non seulement Angular fait moins de comparaison que la version 1.x (une seule passe suffit) mais en plus ces comparaisons sont beaucoup plus rapides !

Depuis le début, l'équipe Google surveille d'ailleurs les performances, avec des `benchmarks` entre AngularJS 1.x, Angular et même Polymer et React sur des cas d'utilisation un peu tordus afin de voir si la nouvelle version est toujours la plus rapide. Il est même possible d'aller encore plus loin, puisque la stratégie de *ChangeDetection* peut même être modifiée de sa valeur par défaut, et être encore plus rapide dans certains cas. Mais ça c'est pour un autre chapitre !

# Chapitre 20. Observables : utilisation avancée

Il me faut confesser une erreur : j'ai sous-estimé la valeur de RxJS et des Observables. Et c'est triste, parce que j'avais déjà fait la même erreur avec AngularJS 1.x et les *promises* ("promesses"). Les *promises* sont très utiles en AngularJS 1.x : une fois qu'on les maîtrise, on peut gérer les parties asynchrones de nos applications très élégamment. Mais cela demande du temps, et il y a quelques pièges à connaître (jette un œil à l'article de blog que nous avons écrit sur le sujet : [http://blog.ninja-squad.com/2015/05/28/angularjs-promises/\[Traps, anti-patterns and tips about AngularJS promises\]](http://blog.ninja-squad.com/2015/05/28/angularjs-promises/[Traps, anti-patterns and tips about AngularJS promises]), si tu veux en savoir plus).

Angular s'appuie sur RxJS, et expose des Observables dans certaines de ses APIs (Http, Forms, Router...). Après avoir codé plus longuement avec RxJS, je me suis dit que cet ebook méritait un chapitre plus poussé sur les Observables, leur création, l'abonnement, les opérateurs disponibles, et leur utilisation possible avec Angular. J'espère que cela te donnera quelques idées, ou au moins l'envie de t'y plonger plus sérieusement, car RxJS pourrait jouer un grand rôle dans l'orchestration de ton application, et probablement te simplifier la vie.

## 20.1. subscribe, unsubscribe et *pipe* `async`

Pour synthétiser ce qu'on a appris dans le chapitre sur la Programmation Réactive, un Observable représente une séquence d'événements à laquelle tu peux t'abonner.

Ce flux d'événements peut survenir à tout moment, et il pourrait n'y avoir qu'un seul événement, ou dix mille d'entre eux. Mais il y a une subtile distinction à percevoir entre deux familles d'Observables : les Observables "chauds" ("*hot*"), et les Observables "froids" ("*cold*").

Les observables froids vont émettre des événements uniquement quand on s'y est abonné. Tu peux faire l'analogie avec le visionnage d'une vidéo sur Youtube : le flux vidéo ne démarrera qu'après avoir appuyé sur lecture. Par exemple, les observables retournés par la classe `Http` sont des observables froids : ils ne déclenchent une requête que quand on invoque `subscribe`.

Les observables chauds sont un peu différents : ils émettent des événements dès leur création. L'analogie serait celle de la télévision : si tu allumes la TV, tu tombes au milieu d'une émission, qui a pu démarrer il y a quelques minutes comme quelques heures. L'observable `valueChanges` dans un `FormControl` est un observable chaud : tu ne recevras pas les valeurs émises avant ton abonnement, mais seulement celles émises après.

Quand tu t'abones à un observable, tu peux passer trois paramètres :

- une fonction pour gérer le prochain événement ;
- une fonction pour gérer une erreur ;
- une fonction pour gérer la terminaison.

La première est plutôt évidente. L'observable est un flux d'événements, et tu définis donc que faire quand un tel événement survient.

La deuxième permet de gérer une hypothétique erreur. Ce paramètre est facultatif. Dans un flux d'événements représentant des clics, il n'y a aucune erreur possible, même si ton utilisateur/utilisatrice se pète les doigts. (cette blague n'est pas de moi, elle est tirée d'une [chouette présentation de André Staltz](#) sur RxJS à NgEurope 2016). Mais dans la plupart des cas, c'est pertinent d'avoir une gestion d'erreur. Cela te permet par exemple de définir quoi faire si tu reçois une erreur en réponse du serveur HTTP.

Il faut retenir un point important sur ce sujet : une erreur est un événement terminal, l'observable n'émettra plus aucun autre événement ensuite. Alors si c'est important pour toi de continuer à écouter, il te faudra gérer correctement ce cas (on y reviendra dans une minute).

La troisième fonction que tu peux passer en paramètre permet de gérer la terminaison : en effet, un observable peut se terminer (une vidéo Youtube par exemple). Et parfois, tu veux réaliser une action spéciale, comme prévenir ton utilisateur-trice, ou calculer une valeur.

Dans notre application Ponycracer, une course est représentée par un observable, qui émet la position des poneys. Quand la course se termine, l'observable arrête d'émettre des événements. A ce moment, on veut alors calculer quel poney a gagné la course, et modifier l'interface en conséquence, etc.

Mais peut-être que l'utilisateur-trice ne va pas regarder la course jusqu'au bout. Que se passe-t-il alors ? Le composant va être détruit. Mais, si on s'est abonné à l'observable dans ce composant avant sa destruction, alors la fonction `next` va continuer à faire son travail à chaque événement. Même si le composant n'est plus affiché ! Cela peut conduire à des fuites mémoire et toute sorte de problème...

La bonne pratique est donc de stocker cet abonnement retourné par la fonction `subscribe`, et à la destruction du composant, appeler `unsubscribe` sur cet abonnement (typiquement dans la méthode `ngOnDestroy`). Enveloppe le `unsubscribe` dans un test pour vérifier si l'abonnement est bien présent (il se peut que l'abonnement n'ait pas encore été initialisé, et tu aurais dans ce cas une erreur en appelant `unsubscribe` sur un objet `undefined`).

Cette règle générale de désabonnement présente quelques exceptions. Cela n'est pas nécessaire pour les observables qui n'émettent qu'un seul événement puis se terminent, comme une requête HTTP. Et elle n'est définitivement pas nécessaire non plus pour les observables du routeur, comme les observables `params` : le routeur s'en chargera pour toi, youpi !

Enfin, un dernier sujet, et non des moindres : le *pipe* `async`. Angular fournit un *pipe* spécial, appelé `async`. Tu peux l'utiliser dans tes templates pour t'abonner directement à un observable.

Cela présente quelques avantages :

- tu peux stocker l'observable directement dans ton composant, sans avoir à t'y abonner manuellement, puis stocker ensuite la valeur émise dans un attribut de ton composant.
- le *pipe* `async` va s'occuper du désabonnement pour toi à la destruction du composant.
- il peut être utilisé pour améliorer magiquement les performances (on y reviendra).

Mais ce *pipe* présente aussi un inconvénient : il te faudra être prudent quand tu utiliseras plusieurs fois cette syntaxe dans un template.

Permet-moi d'illustrer ce dernier point, dans un `RaceComponent`, qui afficherait les caractéristiques d'une course :

```
@Component({
  selector: 'ns-race',
  template: `<div>
    <h2>{{ (race | async)?.name }}</h2>
    <small>{{ (race | async)?.date }}</small>
  </div>`
})
export class RaceComponent implements OnInit {

  race: Observable<RaceModel>;
  error: boolean;

  constructor(private raceService: RaceService) {
  }

  ngOnInit() {
    this.race = this.raceService.get()
      // the 'catch' operator allows to handle a potential error
      // it must return an observable (here an empty one).
      .catch(error => {
        this.error = true;
        console.error(error);
        return Observable.empty();
      });
  }
}
```

Ce bout de code suppose que la méthode `raceService.get()` retourne un Observable émettant un `RaceModel`. On stocke cet observable dans un champ nommé `race` (tu verras parfois le nom `race$` dans certains articles de blogs, car d'autres frameworks utilisent cette convention pour les variables de type Observable). Ensuite on utilise l'observable `race` deux fois dans le template avec le pipe `async` : une première fois pour afficher le nom de la course, et une seconde pour sa date. Le code du composant est plutôt élégant : il n'est pas nécessaire de s'abonner à l'observable.

Mais tu as peut-être déjà repéré le problème. On appelle deux fois le pipe `async`, ce qui veut dire qu'on s'y abonne deux fois. Si la méthode `raceService.get()` déclenche une requête HTTP pour obtenir les détails de la course, cette requête sera déclenchée deux fois !

Une première solution consisterait à modifier l'observable pour le partager entre les différents abonnés. Cela peut se faire avec l'opérateur `publishReplay` par exemple :

```

import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/empty';
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/publishReplay';
import { RaceService, RaceModel } from './race.service';

@Component({
  selector: 'ns-race',
  template: `<div>
    <h2>{{ (race | async)?.name }}</h2>
    <small>{{ (race | async)?.date }}</small>
  </div>`
})
export class RaceComponent implements OnInit {

  race: Observable<RaceModel>;
  error: boolean;

  constructor(private raceService: RaceService) {}

  ngOnInit() {
    this.race = this.raceService.get()
      // the 'catch' operator allows to handle a potential error
      // it must return an observable (here an empty one).
      .catch(error => {
        this.error = true;
        console.error(error);
        return Observable.empty();
      })
      // will share the subscription between the subscribers
      .publishReplay().refCount();
  }
}

```

Mais, même si cela est désormais correct et ne produit pas deux requêtes HTTP différentes, je n'aime pas trop le template. La plupart du temps, le composant a aussi besoin d'accéder à la course pour implémenter des détails de présentation. Et il lui faut aussi gérer les erreurs. Un abonnement, et des opérateurs `do()` et/ou un `catch()` dans le composant sont donc souvent nécessaires. Alors stocker la course dans un champ n'est pas un poids trop lourd, et rend le template plus simple.

Note que la version 4.0 a introduit la syntaxe `as` qui permet de résoudre le problème en ne s'abonnant qu'une seule fois et en stockant le résultat dans une variable du template :

```

import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/empty';
import 'rxjs/add/operator/catch';
import { RaceService, RaceModel } from './race.service';

@Component({
  selector: 'ns-race',
  template: `<div *ngIf="race | async as raceModel">
    <h2>{{ raceModel.name }}</h2>
    <small>{{ raceModel.date }}</small>
  </div>`
})
export class RaceComponent implements OnInit {

  race: Observable<RaceModel>;
  error: boolean;

  constructor(private raceService: RaceService) {
  }

  ngOnInit() {
    this.race = this.raceService.get()
      // the 'catch' operator allows to handle a potential error
      // it must return an observable (here an empty one).
      .catch(error => {
        this.error = true;
        console.error(error);
        return Observable.empty();
      });
  }
}

```

C'est assez élégant.

Une autre solution possible est de couper notre composant en deux : un intelligent ("smart component") qui se chargera de la récupération de la course, et un plus bête ("dumb component") qui se contentera simplement d'afficher la course reçue en entrée.

Cela ressemblerait à :

```

@Component({
  selector: 'ns-racecontainer',
  template: `<div>
    <div *ngIf="error">An error occurred while fetching the race</div>
    <ns-race [raceModel]="race | async"></ns-race>
  </div>`
})
export class RaceContainerComponent implements OnInit {

  race: Observable<RaceModel>;
  error: boolean;

  constructor(private raceService: RaceService) {
  }

  ngOnInit() {
    this.race = this.raceService.get()
    // the 'catch' operator allows to handle a potential error
    // it must return an observable (here an empty one).
    .catch(error => {
      this.error = true;
      console.error(error);
      return Observable.empty();
    });
  }
}

@Component({
  selector: 'ns-race',
  template: `<div>
    <h2>{{ raceModel.name }}</h2>
    <small>{{ raceModel.date }}</small>
  </div>`
})
export class RaceComponent {

  @Input() raceModel: RaceModel;
}

```

Ce modèle peut être utile dans certaines parties de ton application, mais, comme tout modèle, tu n'as pas à l'utiliser systématiquement. C'est parfois satisfaisant de n'avoir qu'un seul composant qui se charge de ces deux aspects. D'autres fois, tu auras besoin d'extraire un composant tout bête pour le réutiliser ailleurs dans ton application, et dans ce cas construire deux composants fera sens.

## 20.2. Le pouvoir des opérateurs

On a déjà croisé quelques opérateurs, mais j'aimerais prendre un peu de temps pour en décrire quelques autres dans un exemple pas à pas. On va coder un champ de saisie *typeahead*. Un champ

typeahead permet à ton utilisateur-trice de saisir du texte dans un champ, puis l'application propose quelques suggestions basées sur cette saisie (comme la boîte de recherche Google).

Un bon typeahead a quelques particularités :

- il affiche des propositions qui correspondent à la recherche (évidemment) ;
- il permet de n'afficher les résultats que si la saisie contient déjà quelques caractères ;
- il ne chargera pas les résultats pour chaque frappe de l'utilisateur-trice, mais attendra un peu pour être sûr qu'il/elle a fini de taper ;
- il ne déclenchera pas deux fois la même requête si l'utilisateur-trice saisit la même recherche.

Tout cela peut se faire à la main, mais c'est loin d'être trivial. Heureusement, Angular et RxJS se combinent plutôt bien pour résoudre ce problème !

Tout d'abord, regardons à quoi ressemblerait un tel composant :

```
import { Component, OnInit } from '@angular/core';
import { FormControl } from '@angular/forms';
import { PonyService, PonyModel } from './pony.service';

@Component({
  selector: 'ns-typeahead',
  template: `<div>
    <input [formcontrol]="input">
    <ul>
      <li *ngFor="let pony of ponies">{{ pony.name }}</li>
    </ul>
  </div>`
})
export class PonyTypeAheadComponent implements OnInit {
  input = new FormControl();
  ponies: Array<PonyModel> = [];

  constructor(private ponyService: PonyService) {}

  ngOnInit() {
    // todo: do something with the input
  }
}
```

Dans la méthode `ngOnInit`, on commence par s'abonner à l'observable `valueChanges` exposé par le `FormControl` (relit le chapitre sur les formulaires si tu as besoin de te rafraîchir la mémoire).

```
this.input.valueChanges
  .subscribe(value => console.log(value));
```

Ensuite, on veut utiliser cette saisie pour trouver les poneys correspondant. Notre service

PonyService possède justement une méthode `search()` dont c'est exactement le travail ! On peut supposer que cette méthode réalise une requête HTTP pour obtenir les résultats du serveur, alors elle retourne un `Observable<Array<PonyModel>>`, un observable qui émet des tableaux de poneys.

Abonnons-nous à cette méthode pour mettre à jour le champ `ponies` de notre composant :

```
this.input.valueChanges
  .subscribe(value => {
    this.ponyService.search(value)
      .subscribe(results => this.ponies = results);
  });
}
```

Cool, ça fonctionne. Mais quand je vois un truc comme ça, cela me rappelle les *promises* et les invocations de `then` imbriquées, qui pourraient être mises à plat. Et effectivement, tu peux faire de même avec les Observables, grâce à l'opérateur `concatMap` par exemple :

```
this.input.valueChanges
  .concatMap(value => this.ponyService.search(value))
  .subscribe(results => this.ponies = results);
```

Voilà, c'est bien plus élégant ! `concatMap` "aplatit" notre code. Il remplace chaque événement émis par l'observable source (i.e. le nom de poney saisi) par les événements émis par l'observable de poneys. Mais ce n'est pas encore l'opérateur idéal dans cette situation. Comme notre méthode `search` déclenche une requête par recherche, on peut rencontrer quelques problèmes si une requête est trop longue. Notre utilisateur-trice pourrait rechercher `n` puis `ni`, et la réponse pourrait mettre beaucoup de temps à venir pour `n`, et moins pour `ni`. Cela signifie que notre code ci-dessus n'affichera les seconds résultats qu'une fois les premiers affichés, alors qu'on s'en moque désormais ! Cela pourrait être traité manuellement, mais ça serait vraiment pénible.

RxJS propos un opérateur super pratique pour ce cas d'utilisation : `switchMap`. À la différence de `concatMap`, `switchMap` ne se préoccupera que de la dernière valeur émise, et ignorera les valeurs précédentes. Ainsi, nous sommes sûrs que les résultats d'une recherche précédente ne seront jamais affichés.

```
this.input.valueChanges
  .switchMap(value => this.ponyService.search(value))
  .subscribe(results => this.ponies = results);
```

OK, maintenant ignorons les recherches de moins de trois caractères. Trop facile : il nous suffit d'utiliser l'opérateur `filter` !

```
this.input.valueChanges
  .filter(query => query.length >= 3)
  .switchMap(value => this.ponyService.search(value))
  .subscribe(results => this.ponies = results);
```

On ne veut également pas que notre recherche se déclenche immédiatement après la frappe : on aimerait la déclencher seulement quand l'utilisateur-trice s'est arrêté-e de taper pendant 400ms. Oui, tu l'as deviné : il y bien a un opérateur pour cela, et il s'appelle `debounceTime` :

```
this.input.valueChanges
  .filter(query => query.length >= 3)
  .debounceTime(400)
  .switchMap(value => this.ponyService.search(value))
  .subscribe(results => this.ponies = results);
```

Donc maintenant un-e utilisateur-trice peut saisir une recherche, corriger quelques caractères, puis en ajouter d'autres, et la requête se déclenchera uniquement quand 400ms se seront écoulées depuis la dernière frappe. Mais que se passe-t-il si l'utilisateur-trice saisit "Rainbow", attend 400ms (ce qui déclenchera donc une requête), puis saisit "Rainbow Dash" et immédiatement supprime "Dash" pour revenir à "Rainbow" ? Cela déclencherait deux requêtes successives pour "Rainbow" ! Pourrait-on déclencher une requête seulement si la recherche est différente de la précédente ? Bien sûr, avec `distinctUntilChanged` :

```
this.input.valueChanges
  .filter(query => query.length >= 3)
  .debounceTime(400)
  .distinctUntilChanged()
  .switchMap(value => this.ponyService.search(value))
  .subscribe(results => this.ponies = results);
```

Une dernière chose : il nous faut encore gérer proprement les erreurs. On sait déjà que `valueChanges` ne va jamais émettre d'erreur, mais notre observable `ponyService.search()` le pourrait tout à fait car il dépend du réseau. Et le problème avec les observables c'est qu'une erreur va définitivement stopper le flux : si la moindre requête échoue, le composant ne fonctionnera plus du tout... Ce n'est évidemment pas ce que l'on souhaite, alors attrapons ces maudites erreurs :

```
this.input.valueChanges
  .filter(query => query.length >= 3)
  .debounceTime(400)
  .distinctUntilChanged()
  .switchMap(value => this.ponyService.search(value).catch(error => Observable.of([])))
  .subscribe(results => this.ponies = results);
```

Plutôt cool, tu ne trouves pas ? On ne déclenche une recherche que si l'utilisateur saisit un texte de plus de 3 caractères et attend au moins 400ms. On garantit qu'on ne déclenchera pas deux fois la même requête, et les suggestions sont toujours en accord avec la recherche ! Et le tout avec 5 lignes de code seulement. Bonne chance pour implémenter la même chose à la main sans introduire de problèmes...

Bon, d'accord, c'est le cas d'utilisation idéal pour RxJS, mais il faut retenir qu'il propose une tonne

d'opérateurs, dont certains sont de véritables pépites. Apprendre à les maîtriser demande du temps, mais cela vaut le coup car ils peuvent être d'une aide précieuse pour ton application.

## 20.3. Construire son propre Observable

Parfois, malheureusement, il te faudra utiliser des bibliothèques qui produisent des événements mais sans utiliser un Observable. Tout espoir n'est pas perdu, car tu peux évidemment créer tes propres Observables, en utilisant, par exemple, la méthode `Observable.create(observer => {})`.

La fonction passée en paramètre est appelée la fonction `subscribe` : elle sera en charge d'émettre les événements et les erreurs, et de terminer le flux.

Par exemple, si tu veux créer un Observable qui émettra 1, puis 2, puis se termine, tu pourrais écrire :

```
const numbers = Observable.create(observer => {
  observer.next(1);
  observer.next(2);
  observer.complete();
});
```

On pourrait alors s'abonner à un tel observable :

```
numbers.subscribe(
  number => console.log(number),
  error => console.log(error),
  () => console.log('Complete!')
);
// Will log:
// 1
// 2
// Complete!
```

Maintenant, disons qu'on veut émettre 'hello' toutes les deux secondes, sans jamais se terminer. On pourrait faire cela facilement avec des opérateurs fournis, mais essayons de le faire nous-même, pour l'exemple :

```

import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/create';

export class HelloService {
  get(): Observable<string> {
    return Observable.create(observer => {
      const interval = setInterval(() => observer.next('hello'), 2000);
    });
  }
}

```

La fonction callback passée à `Observable.create()` peut aussi retourner une fonction qui sera appelée lors du désabonnement. C'est très pratique si tu as du nettoyage à réaliser. Et c'est le cas dans notre `HelloService`, parce qu'il nous faut stopper le `setInterval` au désabonnement de l'observable.

```

import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/create';

export class HelloService {
  get(): Observable<string> {
    return Observable.create(observer => {
      const interval = setInterval(() => observer.next('hello'), 2000);
      return () => clearInterval(interval);
    });
  }
}

```

Comme l'intervalle ne sera pas créé avant l'abonnement, nous venons donc de créer un observable "froid".

J'espère que tu as apprécié ce court chapitre sur les observables. Ils peuvent aussi servir à séquencer tes requêtes HTTP, ou à communiquer entre tes composants (on y reviendra bientôt). Mais tu as désormais un bon aperçu de ce qui est possible !

## MISE EN PRATIQUE

Nous avons de nombreux exercices qui utilisent RxJS et te permettent de découvrir les opérateurs :

- [Afficher l'utilisateur](#) 
- [Page d'accueil connecté](#) 
- [Remember me](#) 
- [Logout](#) 
- [Astuces sur les Observables](#) 
- [Booster un poney](#) 

# Chapitre 21. Internationalisation

*So you want to internationalize your application, huh?*

Bon, ne te tracasse pas si ton niveau d'anglais est si bas que tu n'as pas pu comprendre cette petite introduction. Ton rôle de développeur n'est pas de traduire ton application en anglais, espagnol, ou quelque autre dialecte exotique. Ce que tu peux faire par contre, c'est rendre cette traduction possible. Ce chapitre explique comment y arriver.

## 21.1. La locale

On a déjà évoqué l'internationalisation auparavant, dans le chapitre sur les *pipes*. Trois des *pipes* fournis par Angular concernent l'internationalisation, et utilisent l'API JavaScript standard *Internationalization*, censée être fournie par le navigateur. Ces *pipes* sont les *pipes* `number`, `currency` et `date`.

Ce que nous ne savons pas encore, c'est comment ces *pipes* décident de formater les nombres et les dates. Doivent-ils utiliser le point ou la virgule comme séparateur décimal ? Doivent-ils utiliser *January* ou *Janvier* pour désigner le premier mois de l'année ? Tu pourrais penser que la décision est basée sur la langue préférée configurée dans le navigateur. En réalité, ce n'est pas le cas. La décision est basée sur une valeur injectable nommée `LOCALE_ID`. Et la valeur par défaut de `LOCALE_ID` est '`en-US`'.

Voici un exemple qui montre comment obtenir la valeur de `LOCALE_ID`. Comme tu peux le voir, il s'agit d'une simple chaîne de caractères. Pour l'injecter dans tes composants ou services, tu ne peux pas compter sur son type (`string`). Il est nécessaire d'indiquer à Angular le *token* qui identifie cette valeur, en utilisant `@Inject(LOCALE_ID)`. Ceci peut être utile si la logique du composant ou du service dépend de la locale que l'application utilise.

```
@Component({
  selector: 'ns-locale',
  template: `
    <p>The locale is {{ locale }}</p>
    <!-- will display 'en-US' -->

    <p>{{ 1234.56 | number }}</p>
    <!-- will display '1,234.56' -->
  `
})
class DefaultLocaleComponent {
  constructor(@Inject(LOCALE_ID) public locale: string) { }
```

Tout ça est bien beau, mais comment peut-on faire pour *changer* la locale ? En fait, on ne peut pas. La locale est une constante, qu'on ne peut changer pendant l'exécution de l'application. Mais on peut en revanche fournir une autre valeur que '`en-US`' *avant* que l'application ne démarre. Ceci est possible, simplement en fournissant une autre valeur pour le token `LOCALE_ID`, dans le module

Angular principal.

Voici un exemple, qui montre l'effet que cela produit sur notre composant :

```
@NgModule({
  imports: [BrowserModule],
  declarations: [CustomLocaleComponent], // and other components
  providers: [
    { provide: LOCALE_ID, useValue: 'fr-FR' }
  ]
// ...
})
export class AppModule { }

@Component({
  selector: 'ns-locale',
  template: `
    <p>The locale is {{ locale }}</p>
    <!-- will display 'fr-FR' -->

    <p>{{ 1234.56 | number }}</p>
    <!-- will display '1 234,56' -->
  `
})
class CustomLocaleComponent {
  constructor(@Inject(LOCALE_ID) public locale: string) { }
}
```

Si tu veux créer une application entièrement francophone (par exemple), c'est tout ce dont tu as besoin. Mais bien souvent, les utilisateurs de ton application parlent diverses langues, et il faut donc aller plus loin, et réellement internationaliser l'application.

## 21.2. Traduire du texte

Si tu as utilisé AngularJS 1.x pour construire une application internationalisée, tu sais qu'AngularJS ne propose pas de solution pour afficher du texte traduit dans la langue de préférence de l'utilisateur.

Ce manque est comblé par des librairies externes au framework, dont l'une, très populaire, est [angular-translate](#). La stratégie qu'elle utilise est assez commune : le template HTML contient des clés (comme par exemple '`home.welcome`'), qui sont traduites grâce à une directive ou un filtre. Chaque clé identifie donc un message, et chaque message est traduit dans chacune des langues que l'application supporte (par exemple : 'Welcome' et 'Bienvenue'). A l'exécution, la directive ou le filtre utilise la langue préférée pour obtenir la traduction adéquate, et met à jour le DOM avec le message traduit. Tu peux changer de langue préférée à l'exécution, et tous les messages de la page sont immédiatement traduits dans cette nouvelle langue.

Avec Angular, l'internationalisation est à présent directement supportée, sans avoir à recourir à des librairies externes (bien que cette fonctionnalité ne soit vraiment utilisable que depuis la version

4.0, et qu'il reste encore du chemin à parcourir). Angular utilise la même stratégie basée sur des clés, mais avec une différence importante : le remplacement des clés par les messages traduits est effectué pendant la phase de compilation plutôt qu'à l'exécution. Lorsque l'application démarre, ou, si tu utilises la compilation *AOT*, lorsque tu construis l'application, Angular analyse les templates HTML de tous les composants, et les compile en code JavaScript qui, essentiellement, analyse les changements dans le modèle et modifie le DOM en conséquence. C'est pendant cette phase de compilation du HTML en JavaScript que la traduction est réalisée. Cela a des conséquences importantes :

- on ne peut pas changer la locale (et donc le texte affiché dans l'application) pendant l'exécution. L'application entière doit être rechargée et redémarrée pour changer de langue ;
- une fois démarrée, l'application est plus rapide, parce qu'elle ne doit pas traduire les clés encore et encore ;
- si tu utilises la compilation *AOT* (et tu devrais, au moins en production), tu dois construire et servir autant de versions de l'application que de locales supportées.

## 21.3. Processus et outillage

Dans la suite de ce chapitre, nous supposerons que tu utilises Angular CLI pour construire ton application. Les outils sont en fait utilisables sans Angular CLI, parce qu'ils font partie de `ngc`, le compilateur Angular. Mais comme ils sont bien intégrés et simples à utiliser dans Angular CLI, et que c'est l'outil recommandé pour construire ton application, c'est ce que nous utiliserons.

Nous utiliserons aussi la compilation *AOT* pour construire et servir notre application internationalisée. C'est en effet le moyen le plus simple d'y parvenir. Si tu veux tester l'internationalisation en mode *JIT* (i.e. sans précompiler les templates), cela nécessite des modifications substantielles du code utilisé pour lancer l'application. Réfère-toi à [l'internationalization cookbook](#) de la documentation officielle pour plus de détails.

Enfin, nous supposerons que tu es en fait un parfait anglophone qui a écrit son application en anglais et qui désire la traduire en français, langue que tu ne maîtrises que de façon parcellaire.

Cela étant dit, comment procède-t-on. Tu devrais à présent savoir comment écrire des composants et leurs templates. Va-t-il falloir tous les réécrire pour les internationaliser ? Heureusement non. La procédure est la suivante :

1. tu marques les parties des templates qui doivent être traduites en utilisant un attribut `i18n` ;
2. tu exécutes une commande permettant d'extraire ces parties marquées vers un fichier, par exemple `messages.xlf`. Deux formats de fichiers, des standards de l'industrie basés sur XML, sont supportés ;
3. tu demandes à un traducteur compétent de fournir une version traduite de ce fichier, par exemple `messages.fr.xlf` ;
4. tu construis l'application en fournissant la locale ('`fr`' par exemple) et ce fichier contenant les traductions françaises. (`messages.fr.xlf`). Le compilateur Angular et la CLI remplacent les parties marquées via l'attribut `i18n` par les traductions trouvées dans le fichier, et configurent l'application avec le `LOCALE_ID` fourni.

Exammons ces différentes étapes en détail.

### 21.3.1. Marquer le texte à traduire et l'extraire

Commençons avec un template d'exemple :

```
<h1>Welcome to Ponyracer</h1>
<p>Welcome to Ponyracer {{ user.firstName }} {{ user.lastName }}!</p>

Let's start playing.
```

Cinq morceaux de texte doivent être traduits dans ce template. Bien sûr, on pourrait considérer tout le template comme un seul long bloc à traduire. Mais dans un exemple plus réaliste, cela exposerait beaucoup de code HTML aux traducteurs. Et devoir retraduire tout à chaque fois que la structure du HTML change n'est vraiment pas acceptable. Il faut donc traduire les 5 morceaux séparément.

L'un d'eux, le contenu de l'élément `h1`, est du texte purement statique. L'un d'eux est du texte contenant deux expressions interpolées. Deux d'entre eux sont des attributs d'un élément HTML. Le dernier est du texte statique qui n'est contenu dans aucun élément.

Voici comment les marquer. Commençons avec le plus simple, le premier :

```
<h1 i18n>Welcome to Ponyracer</h1>
```

A présent qu'on a ajouté l'attribut `i18n` sur l'élément, utilisons la commande `xi18n` fournie par Angular CLI :

```
ng xi18n --output-path src/locale
```

Cela va générer un fichier `messages.xlf` dans le répertoire `src/locale`. Voici ce qu'il contient :

```
<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en" datatype="plaintext" original="ng2.template">
    <body>
      <trans-unit id="5e3335d7f1a430ef14a91507531838c57138b7f2" datatype="html">
        <source>Welcome to Ponyracer</source>
        <target/>
      </trans-unit>
    </body>
  </file>
</xliff>
```

Comme tu peux le voir, cela génère une `trans-unit` contenant, comme valeur source, notre texte statique. Le rôle du traducteur francophone est de fournir un fichier `messages.fr.xlf` qui ressemble

à ça :

```
<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en" datatype="plaintext" original="ng2.template">
    <body>
      <trans-unit id="5e3335d7f1a430ef14a91507531838c57138b7f2" datatype="html">
        <source>Welcome to Ponyracer</source>
        <target>Bienvenue dans Ponyracer</target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

Cela est relativement simple, parce que le message source est facile à comprendre. Il n'y a pas besoin de beaucoup d'informations contextuelles supplémentaires pour comprendre de quoi il s'agit, et comment traduire ce message. Néanmoins, cette façon de faire a un gros inconvénient. Si tu changes le code source du template pour y ajouter des sauts de lignes insignifiants par exemple, ou un point à la fin du titre, voici ce qui se produit lorsqu'on extrait à nouveau les messages :

```
<h1 i18n>
  Welcome to Ponyracer .
</h1>
```

```
<trans-unit id="6e37e34598c734b3649ba478cac5f2d29e67c331" datatype="html">
  <source>
    Welcome to Ponyracer .
  </source>
  <target/>
</trans-unit>
```

Non seulement le nouveau message source a changé, reflétant la nouvelle valeur dans le template, ce qui est normal. Mais l'identifiant du message a aussi changé. Cela rend la maintenance des traductions plus pénible et complexe que nécessaire. Heureusement, il y a un meilleur moyen. Tu peux fournir l'identifiant unique du message de manière explicite :

```
<h1 i18n="@@home.title">Welcome to Ponyracer</h1>
```

qui produit :

```
<trans-unit id="home.title" datatype="html">
  <source>Welcome to Ponyracer</source>
  <target/>
</trans-unit>
```

En fait, afin de fournir plus de contexte aux traducteurs, tu peux même aller plus loin et indiquer une signification et une description en plus de l'identifiant du message :

```
<h1 i18n="welcome title|the title of the home page@@home.fullTitle">Welcome to  
Ponyracer</h1>
```

```
<trans-unit id="home.fullTitle" datatype="html">  
  <source>Welcome to Ponyracer</source>  
  <target/>  
  <note priority="1" from="description">the title of the home page</note>  
  <note priority="1" from="meaning">welcome title</note>  
</trans-unit>
```

Passons à présent au second morceau de texte :

```
<p i18n="@@home.welcome">Welcome to Ponyracer {{ user.firstName }} {{ user.lastName }}!</p>
```

Voici ce que l'extraction produit pour ce message :

```
<trans-unit id="home.welcome" datatype="html">  
  <source>Welcome to Ponyracer <x id="INTERPOLATION"/> <x id="INTERPOLATION_1"/>  
!</source>  
  <target/>  
</trans-unit>
```

Comme tu peux le voir, ce format a des caractéristiques intéressantes :

- le traducteur n'est pas confronté aux expressions interpolées. Il ne peut pas malencontreusement traduire le contenu des expressions, qui doivent rester telles quelles, puisqu'elles ne sont pas présentes dans le message extrait ;
- en revanche, ce que ces deux expressions représentent n'est pas clair du tout. Dans ce cas, ce serait donc une bonne idée d'expliquer cela dans la description du message ;
- si, dans certaines langues, le nom de famille doit venir avant le prénom, le traducteur est libre de réordonner les deux interpolations ;
- si le développeur choisit de renommer l'attribut `user` du composant, ou les attributs `firstName` et `lastName` de l'utilisateur, le message extrait reste identique, et rien ne doit être re-traduit.

Passons maintenant aux deux attributs de l'élément `img`. La syntaxe pour traduire des attributs est la suivante :

```

```

Cela génère les `trans-unit` suivantes :

```
<trans-unit id="home.ponyImage.alt" datatype="html">
  <source>running pony</source>
  <target/>
</trans-unit>
<trans-unit id="home.ponyImage.title" datatype="html">
  <source>Ponies are cool, aren't they?</source>
  <target/>
</trans-unit>
```

Enfin, comment traduire le dernier morceau de texte ? Il n'y a aucun élément qui puisse porter l'attribut `i18n`. Deux solutions sont possibles : utiliser un commentaire, ou utiliser un élément `ng-container`, qui ne fera pas partie du HTML généré par le template :

```
<!--i18n: @@home.startMessage -->Let's start playing.<!--/i18n-->
```

ou

```
<ng-container i18n="@@home.startMessage">Let's start playing.</ng-container>
```

Ma préférence va à la deuxième solution, qui est plus cohérente avec les autres types de messages.

### 21.3.2. Traduire, construire et servir l'application

Maintenant que le fichier `messages.xlf` est extrait et contient tous les messages à traduire, quelqu'un doit s'en charger.

#### ATTENTION

Une erreur classique est de simplement remplacer le message source dans le fichier par sa traduction. Cela ne fonctionnera pas. Les traductions doivent être saisies dans l'élément `<target>` de chaque élément `trans-unit`. Le contenu de l'élément `<source>` doit être conservé tel quel : il fournit le message original qui doit être traduit. Voici un exemple de message correctement traduit :

```
<trans-unit id="home.welcome" datatype="html">
  <source>Welcome to Ponyracer <x id="INTERPOLATION"/> <x id="INTERPOLATION_1"/>
!</source>
  <target>Bienvenue dans Ponyracer <x id="INTERPOLATION"/> <x id="INTERPOLATION_1"
/>&ampnbsp!</target>
</trans-unit>
```

Pour exécuter ou construire l'application en français, tu dois indiquer la locale, ainsi que l'emplacement du fichier de messages à utiliser, à la commande `ng serve` ou `ng build`:

```
# pour exécuter :  
ng serve --aot --locale fr --i18n-file src/locale/messages.fr.xlf  
  
# pour construire :  
ng build --aot --locale fr --i18n-file src/locale/messages.fr.xlf
```

Le compilateur AOT invoqué par ces commandes va localiser tous les morceaux de texte marqués par l'attribut `i18n` dans les templates, trouver les traductions correspondantes dans le fichier XLF, et transformer le texte des templates en texte traduit. Ensuite, il va transformer comme d'habitude ces templates HTML traduits en code JavaScript, et construire le *bundle* de l'application.

Si tu veux supporter l'anglais, le français et l'espagnol, par exemple, il te faudra construire ton application trois fois (une pour chaque langue), et déployer ces trois applications sur ton serveur web de production. Tu devras aussi décider quelle application est servie à quel utilisateur. Cela peut être fait côté serveur, en détectant la locale préférée dans le header de la requête HTTP et en servant la page `index.html` adéquate. Si l'utilisateur est authentifié, sa langue préférée peut aussi être stockée avec le reste de ses informations dans la base de données. Cette détection peut aussi être réalisée côté client, en servant les trois applications sur trois URLs différentes (`ponyracer.com`, `ponyracer.fr` et `ponyracer.es`, ou bien `ponyracer.com/en`, `ponyracer.com/fr` et `ponyracer.com/es`), et en redirigeant de `ponyracer.com` vers l'une des trois URLs en fonction de la langue configurée dans le navigateur.

## 21.4. Traduire les messages dans le code

Parfois, le texte à traduire n'est pas dans les templates, mais dans le code TypeScript. Par exemple, les trois états PENDING, RUNNING et FINISHED d'une course de poneys devraient être traduits d'une manière ou d'une autre. Le plan actuel est de pouvoir utiliser quelque chose comme ça :

```
const RaceStatus = {  
  PENDING: __('pending'),  
  RUNNING: __('running'),  
  FINISHED: __('finished')  
}
```

Malheureusement, cette fonctionnalité n'est pas encore implémentée. Une solution temporaire peut être, par exemple, de charger des fichiers de traductions en JSON depuis le serveur, en se basant sur le `LOCALE_ID` pour télécharger le fichier approprié.

## 21.5. Pluralisation

Parfois, le message à afficher dépend du nombre d'éléments dans une collection, ou d'un nombre quelconque stocké dans une propriété.

Par exemple, supposons que notre page d'accueil affiche le nombre de courses prévues pour la journée. On pourrait simplement afficher "*Nombre de course(s) prévue(s) : 4*". Mais la page serait plus accueillante si elle affichait plutôt "*Aucune course n'est prévue*", ou "*Seule une course est prévue*", ou encore "*N courses sont prévues*" dans les autres cas.

Angular, en fait, a une syntaxe spécifique pour faire cela. Elle est assez difficile à lire, sauf peut-être pour les programmeurs LISP, mais elle fait le boulot et est relativement simple à comprendre à partir de l'exemple suivant. Supposons que notre composant ait une propriété `racesPlanned`, contenant le nombre de courses prévues. On peut l'afficher, en anglais, de la manière suivante :

```
<p>Hello, {racesPlanned, plural, =0 {no race is planned}
           =1 {only one race is planned}
           other '{{racesPlanned}} races are planned}</p>
```

Pour internationaliser un tel message, on utilise l'attribut `i18n`, comme d'habitude :

```
<p i18n="@@home.racesPlanned">
  Hello, {racesPlanned, plural, =0 {no race is planned}
           =1 {only one race is planned}
           other '{{racesPlanned}} races are planned}.
</p>
```

L'extraction de ce message, cependant, génère deux éléments `trans-unit`, qui sont loin d'être aisés à comprendre pour les traducteurs : l'un pour le message lui-même, et l'autre pour l'expression contenue dans le message, avec un identifiant auto-généré, et pas de valeur source. La fonctionnalité laisse donc encore beaucoup à désirer, et devrait, on l'espère, être améliorée à l'avenir.

```
<trans-unit id="home.racesPlanned" datatype="html">
  <source>
    Hello, <x id="ICU"/>.
  </source>
  <target/>
</trans-unit>
<trans-unit id="b9fa22342555fb5d241fadfd19c6ac99c69fdaca" datatype="html">
  <source/>
  <target/>
</trans-unit>
```

Il est possible de traduire ces deux `trans-unit`, cependant, et la traduction fonctionne correctement :

```

<trans-unit id="home.racesPlanned" datatype="html">
  <source>
    Hello, <x id="ICU"/>.
  </source>
  <target>Bonjour, <x id="ICU"/>.</target>
</trans-unit>
<trans-unit id="b9fa22342555fb5d241fadfd19c6ac99c69fdaca" datatype="html">
  <source/>
  <target>{racesPlanned, plural, =0 {aucune course n'est planifiée}
            =1 {seule une course est planifiée}
           other {{racesPlanned}} courses sont planifiées}</target>
</trans-unit>

```

## 21.6. Bonnes pratiques

Ces bonnes pratiques, acquises par des années d'expérience de développement d'applications internationalisées, ne sont pas nécessairement liées à Angular, mais à l'internationalisation en général.

Spécifie toujours un identifiant unique, de manière explicite, pour les messages. Si tu choisis un identifiant clair, qui a du sens, il est souvent inutile de fournir une description pour le message, parce que l'identifiant est suffisant. Préfixer les identifiants avec le nom du composant dans lequel ils sont utilisés (comme je l'ai fait dans tous les exemples précédents avec le préfixe `home.`) permet de savoir où ils sont utilisés, et de les retrouver rapidement dans le code. Compter sur des identifiants auto-générés ne permet pas d'avoir des traductions différentes pour deux messages identiques utilisés dans deux contextes différents. Faire la différence entre les messages d'une précédente version de l'application et la version actuelle est aussi bien plus difficile.

Même si les traducteurs ne sont pas toujours des développeurs, stocke les fichiers de messages dans ton système de gestion des sources (Git, etc.). Cela permet à chaque branche d'avoir ses propres modifications dans les fichiers de messages, qui peuvent n'être intégrées dans la branche principale que lorsque la branche est prête. Cela rend aussi la comparaison entre branches ou releases plus aisée.

La duplication n'est pas nécessairement une mauvaise chose. Tu pourrais penser que deux pages partageant un même libellé "Enregistrer" ou "OK" devraient utiliser le même identifiant de message. Mais peut-être devront-ils être renommés respectivement "OK, je vais le faire" et "OK, j'accepte". Ou peut-être ces termes assez vagues nécessitent-ils d'être différenciés dans certaines langues étrangères. Il est d'autant plus important d'utiliser des identifiants distincts lorsqu'un même mot est utilisé, mais avec des significations différentes. Par exemple, "Valider" peut vouloir dire "vérifier la cohérence des données" mais aussi "accepter la proposition". Ce même mot serait traduit en anglais par "Validate" dans le premier cas, et "Accept" dans le deuxième.

Ne confond pas *langue* et *pays*. N'utilise pas des drapeaux de pays pour représenter une langue. Certaines langues sont parlées dans de nombreux pays (comme le français ou l'anglais), et certains pays utilisent plusieurs langues (comme la Belgique, qui utilise le français, le néerlandais et l'allemand).

Évite d'utiliser la concaténation pour traduire des messages paramétrés. Par exemple, pour traduire "*Bonjour, je m'appelle X et j'ai Y ans*", n'utilise pas une première clé pour ""*Bonjour, je m'appelle* ", une seconde clé pour *\_*" et *j'ai* " et une troisième pour " *ans*". Utilise une clé unique, contenant des expressions interpolées.

Tu devrais à présent être prêt à conquérir le monde avec ta splendide application multilingue.

# Chapitre 22. Ce n'est qu'un au revoir

Merci d'avoir lu ce livre !

Quelques chapitres seront ajoutés dans des versions ultérieures : l'utilisation du routeur (qui n'était pas finalisé pour cette première version), des sujets avancés, et d'autres surprises. Ils ont encore besoin d'être signés, mais je pense que tu les apprécieras. Et bien sûr, nous tiendrons à jour ce contenu avec les nouvelles versions du framework, pour que tu ne rates pas les nouvelles fonctionnalités à venir. Et toutes ces révisions du livre seront gratuites, bien évidemment !

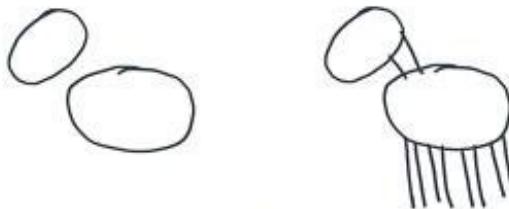
Si tu as aimé cette lecture, parles-en à tes amis !

Et si tu ne l'as pas déjà, sache qu'on propose une version "pro" de cet ebook. Cette version donne accès à toute une série d'exercices qui permettent de construire un projet complet pas à pas, en partant de zéro. Pour chaque étape on fournit les tests unitaires nécessaires à une couverture de code de 100%, des instructions détaillées (qui ne sont pas un simple copier/coller, elles te feront réfléchir et comprendre ce que tu utilises) et une solution (qui sans trop se vanter est probablement la plus élégante, ou en tout cas celle conforme à l'état de l'art). Un outil maison calcule ton score pour l'exercice, et ta progression est visible sur un tableau de bord. Si tu cherches des exemples de code concrets, toujours à jour, qui peuvent te faire gagner des heures de développement, cette version pro est là pour toi{nbs})! Tu peux même [essayer gratuitement les premiers exercices](#), pour te faire une idée concrète. Et vu que tu es déjà le possesseur de cet ebook, on te remercie de ton soutien historique avec un généreux code de réduction pour le pack pro que tu peux aller chercher [à cette adresse](#).

On a essayé de te donner toutes les clés, mais l'apprentissage du web se passe souvent comme ça :

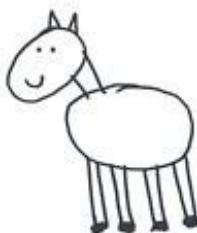
# HOW TO: DRAW A HORSE

BY VAN OKTOP

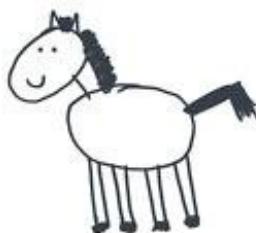


① DRAW 2 CIRCLES

② DRAW THE LEGS



③ DRAW THE FACE



④ DRAW THE HAIR



How to draw a horse. Credit to Van Oktop.

Alors on propose aussi une formation, en France et en Europe essentiellement, mais pourquoi pas dans le monde entier. On peut aussi faire du conseil pour aider ton équipe, ou même travailler avec toi pour t'aider à construire ton produit. Envoie un email à [hello@ninja-squad.com](mailto:hello@ninja-squad.com) et on en discutera !

Mais surtout, j'adorerais avoir ton retour sur ce que tu as aimé, adoré, ou détesté dans ce livre. Cela peut être une petite faute de frappe, une grosse erreur, ou juste pour nous raconter comment tu as trouvé le job de tes rêves grâce à ce livre (on ne sait jamais...).

Je ne peux pas conclure sans remercier quelques personnes. Ma petite amie tout d'abord, qui a été d'un soutien formidable, même quand je passais mon dimanche sur la dixième réécriture d'un chapitre dans une humeur détestable. Mes collègues ninjas, pour leur travail et retours sans

relâche, leur gentillesse et encouragements, et pour m'avoir laissé le temps nécessaire à cette folle idée. Et mes amis et ma famille, pour les petits mots qui redonnent l'énergie au bon moment.

Et merci surtout à toi, pour avoir acheté ce livre, et l'avoir lu jusqu'à cette toute dernière ligne.

À très bientôt.

# Annexe A: Historique des versions

Voici ci-dessous les changements que nous avons apportés à cet ebook depuis sa première version. C'est en anglais, mais ça devrait t'aider à voir les nouveautés depuis ta dernière lecture !

N'oublie pas qu'acheter cet ebook te donne droit à toutes ses mises à jour ultérieures, gratuitement. Rends-toi sur la page <https://books.ninja-squad.com/claim> pour obtenir la toute dernière version.

## A.1. v1.6 - 2017-03-24

### Global

- Bump to stable release [4.0.0](#) (2017-03-24)
- Bump to [4.0.0-rc.6](#) (2017-03-23)
- Bump to [4.0.0-rc.5](#) (2017-03-23)
- Bump to [4.0.0-rc.4](#) (2017-03-23)
- Bump to [4.0.0-rc.3](#) (2017-03-23)
- Bump to [4.0.0-rc.1](#) (2017-03-23)
- Bump to [4.0.0-beta.8](#) (2017-03-23)
- Bump to [ng 4.0.0-beta.7](#) and TS 2.1+ is now required (2017-03-23)
- Bump to [4.0.0-beta.5](#) (2017-03-23)
- Bump to [4.0.0-beta.0](#) (2017-03-23)
- Each chapter now has a link to the corresponding exercise of our [Pro Pack](#) Chapters are slightly re-ordered to match the exercises order. (2017-03-22)

### The templating syntax

- Use `as`, introduced in 4.0.0, instead of `let` for variables in templates (2017-03-23)
- The `template` tag is now deprecated in favor of `ng-template` in 4.0 (2017-03-23)
- Introduces the `else` syntax from version 4.0.0 (2017-03-23)

### Dependency Injection

- Fix the Babel 6 config for dependency injection without TypeScript (2017-02-17)

### Pipes

- Introduce the `as` syntax to store a `NgIf` or `NgFor` result, which can be useful with some pipes like `slice` or `async`. (2017-03-23)
- Adds `titlecase` pipe introduced in 4.0.0 (2017-03-23)

### Services

- New `Meta` service in 4.0.0 to get/set meta tags (2017-03-23)

## Testing your app

- `overrideTemplate` has been added in 4.0.0 (2017-03-23)

## Forms

- Introduce the `email` validator from version 4.0.0 (2017-03-23)

## Send and receive data with Http

- Use `params` instead of the deprecated `search` in 4.0.0 (2017-03-23)

## Router

- Use `paramMap` introduced in 4.0 instead of `params` (2017-03-23)

## Advanced observables

- Shows the `as` syntax introduced in 4.0.0 as an alternative for the multiple async pipe subscriptions problem (2017-03-23)

## Internationalization

- Add a new chapter on internationalization (i18n) (2017-03-23)

# A.2. v1.5 - 2017-01-25

## Global

- Bump to [2.4.4](#) (2017-01-25)
- The big rename: "Angular 2" is now known as "Angular" (2017-01-13)
- Bump to [2.4.0](#) (2016-12-21)

## Forms

- Fix the `NgModel` explanation (2017-01-09)
- `Validators.compose()` is no longer necessary, we can apply several validators by just passing an array. (2016-12-01)

# A.3. v1.4 - 2016-11-18

## Global

- Bump to [2.2.0](#) (2016-11-18)
- Bump to [2.1.0](#) (2016-10-17)
- Remove typings and use `npm install @types/…` (2016-10-17)
- Use `const` instead of `let` and TypeScript type inference whenever possible (2016-10-01)
- Bump to [2.0.1](#) (2016-09-24)

## Testing your app

- Use `TestBed.get` instead of `inject` in tests (2016-09-30)

## Forms

- Add an async validator example (2016-11-18)
- Remove the useless (2.2+) `.control` in templates like `username.control.hasError('required')`. (2016-11-18)

## Router

- `routerLinkActive` can be exported (2.2+). (2016-11-18)
- We don't need to unsubscribe from the router params in the `ngOnDestroy` method. (2016-10-07)

## Advanced observables

- New chapter on Advanced Observables! (2016-11-03)

# A.4. v1.3 - 2016-09-15

## Global

- Bump to stable release [2.0.0](#) (2016-09-15)
- Bump to [rc.7](#) (2016-09-14)
- Bump to [rc.6](#) (2016-09-05)

## From zero to something

- Update the SystemJS config for [rc.6](#) and bump the RxJS version (2016-09-05)

## Pipes

- Remove the section about the replace pipe, removed in rc.6 (2016-09-05)

# A.5. v1.2 - 2016-08-25

## Global

- Bump to [rc.5](#) (2016-08-23)
- Bump to [rc.4](#) (2016-07-08)
- Bump to [rc.3](#) (2016-06-28)
- Bump to [rc.2](#) (2016-06-16)
- Bump to [rc.1](#) (2016-06-08)
- Code examples now follow the official style guide (2016-06-08)

## From zero to something

- Small introduction to NgModule when you start your app from scratch (2016-08-12)

## The templating syntax

- Replace the deprecated `ngSwitchWhen` with `ngSwitchCase` (2016-06-16)

## Dependency Injection

- Introduce modules and their role in DI. Changed the example to use a custom service instead of `Http`. (2016-08-15)
- Remove deprecated `provide()` method and use `{provide: ...}` instead (2016-06-09)

## Pipes

- Date pipe is now fixed in `rc.2`, no more problem with Intl API (2016-06-16)

## Styling components and encapsulation

- New chapter on styling components and the different encapsulation strategies! (2016-06-08)

## Services

- Add the service to the module's providers (2016-08-21)

## Testing your app

- Tests now use the TestBed API instead of the deprecated TestComponentBuilder one. (2016-08-15)
- Angular 2 does not provide Jasmine wrappers and custom matchers for unit tests in `rc.4` anymore (2016-07-08)

## Forms

- Forms now use the new form API (`FormsModule` and `ReactiveFormsModule`). (2016-08-22)
- Warn about forms module being rewritten (and deprecated) (2016-06-16)

## Send and receive data with Http

- Add the `HttpModule` import (2016-08-21)
- `http.post()` now autodetects the body type, removing the need of using `JSON.stringify` and setting the `ContentType` (2016-06-16)

## Router

- Introduce `RouterModule` (2016-08-21)
- Update the router to the API v3! (2016-07-08)
- Warn about router module being rewritten (and deprecated) (2016-06-16)

## Changelog

- Mention free updates and web page for obtaining latest version (2016-07-25)

## A.6. v1.1 - 2016-05-11

### Global

- Bump to `rc.0`. All packages have changed! (2016-05-03)
- Bump to `beta.17` (2016-05-03)
- Bump to `beta.15` (2016-04-16)
- Bump to `beta.14` (2016-04-11)
- Bump to `beta.11` (2016-03-20)
- Bump to `beta.9` (2016-03-11)
- Bump to `beta.8` (2016-03-10)
- Bump to `beta.7` (2016-03-04)
- Display the Angular 2 version used in the intro and in the chapter "Zero to something". (2016-03-04)
- Bump to `beta.6` (`beta.4` and `beta.5` were broken) (2016-03-04)
- Bump to `beta.3` (2016-03-04)
- Bump to `beta.2` (2016-03-04)

### Diving into TypeScript

- Use `typings` instead of `tsd`. (2016-03-04)

### The templating syntax

- `*ngFor` now uses `let` instead of `to` to declare a variable `*ngFor="let pony of ponies"` [small](2016-05-03) #
- `*ngFor` now also exports a `first` variable (2016-04-16)

### Dependency Injection

- Better explanation of hierarchical injectors (2016-03-04)

### Pipes

- A `replace` pipe has been introduced (2016-04-16)

### Reactive Programming

- Observables are not scheduled for ES7 anymore (2016-03-04)

### Building components and directives

- Explain how to remove the compilation warning when using `@Input` and a setter at the same time (2016-03-04)

- Add an explanation on `isFirstChange` for `ngOnChanges` (2016-03-04)

## Testing your app

- `injectAsync` is now deprecated and replaced by `async` (2016-05-03)
- Add an example on how to test an event emitter (2016-03-04)

## Forms

- A pattern validator has been introduced to make sure that the input matches a regexp (2016-04-16)
- Add a mnemonic tip to rememeber the `[()`] syntax: the banana box! (2016-03-04)
- Examples use `module.id` to have a relative `templateUrl` (2016-03-04)
- Fix error `ng-no-form` → `ngNoForm` (2016-03-04)
- Fix errors `(ngModel)` → `(ngModelChange)`, `is-old-enough` → `isOldEnough` (2016-03-04)

## Send and receive data with Http

- Use `JSON.stringify` before sending data with a POST (2016-03-04)
- Add a mention to `JSONP_PROVIDERS` (2016-03-04)

## Router

- Introduce the new router (previous one is deprecated), and how to use parameters in URLs! (2016-05-06)
- `RouterOutlet` inserts the template of the component just after itself and not inside itself (2016-03-04)

## Zones and the Angular magic

- New chapter! Let's talk about how Angular 2 works under the hood! First part is about how AngularJS 1.x used to work, and then we'll see how Angular 2 differs, and uses a new concept called zones. (2016-05-03)