

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 ECMAScript 2015+	4
2.1. Transpileur	4
2.2. <code>let</code>	5
2.3. Constantes	6
2.4. Raccourcis pour la création d'objets	7
2.5. Affectations déstructurées	8
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>Async/await</i>	20
2.12. <i>Set</i> et <i>Map</i>	21
2.13. Template de string	22
2.14. Modules	23
2.15. Conclusion	25
3. Un peu plus loin qu'ES2015+	26
3.1. Types dynamiques, statiques et optionnels	26
3.2. Hello TypeScript	27
3.3. Un exemple concret d'injection de dépendance	27
4. Découvrir TypeScript	30
4.1. Les types de TypeScript	30
4.2. Valeurs énumérées (<i>enum</i>)	31
4.3. Return types	32
4.4. Interfaces	33
4.5. Paramètre optionnel	33
4.6. Des fonctions en propriété	34
4.7. Classes	34
4.8. Utiliser d'autres bibliothèques	36
4.9. Décorateurs	37
5. TypeScript avancé	40
5.1. <code>readonly</code>	40
5.2. <code>keyof</code>	40
5.3. Mapped type	41
5.4. Union de types et gardien de types	43
6. Le monde merveilleux des Web Components	46
6.1. Le nouveau Monde	46

6.2. Custom elements	47
6.3. Shadow DOM	48
6.4. Template	49
6.5. Les bibliothèques basées sur les Web Components	49
7. La philosophie d'Angular	51
8. Commencer de zéro	56
8.1. Node.js et NPM	56
8.2. Angular CLI	56
8.3. Structure de l'application	58
8.4. Notre premier composant <i>standalone</i>	59
8.5. Démarrer l'application	61
9. La syntaxe des templates	63
9.1. Interpolation	64
9.2. Utiliser d'autres composants dans nos templates	67
9.3. <i>Binding</i> de propriété	68
9.4. Événements	71
9.5. Expressions vs instructions	75
9.6. Variables locales	75
9.7. Directives de structure	76
9.8. Autres directives de templating	80
9.9. Résumé	81
10. Créer des composants et directives	85
10.1. Introduction	85
10.2. Directives	85
10.3. Composants	96
11. Style des composants et encapsulation	99
11.1. Stratégie <i>Shadow DOM</i>	100
11.2. Stratégie <i>Emulated</i> ("émulée")	100
11.3. Stratégie <i>None</i> ("aucune")	101
11.4. Style l'hôte	101
12. <i>Pipes</i>	103
12.1. Ceci n'est pas une pipe	103
12.2. json	103
12.3. slice	104
12.4. keyvalue ("clé valeur")	106
12.5. uppercase ("majuscule")	107
12.6. lowercase ("minuscule")	107
12.7. titlecase ("titre")	108
12.8. number ("nombre")	108
12.9. percent ("pourcent")	109
12.10. currency ("devise monétaire")	109

12.11. date	110
12.12. async	110
12.13. Un <i>pipe</i> dans ton code	112
12.14. Créer tes propres <i>pipes</i>	112
13. Injection de dépendances	115
13.1. Cuisine et Dépendances	115
13.2. Développement facile	115
13.3. Configuration facile	118
13.4. Autres types de provider	121
13.5. Injecteurs hiérarchiques	122
13.6. Injection sans types	125
13.7. <code>inject()</code>	126
13.8. Services provided by the framework	127
14. Programmation réactive	129
14.1. OK, on vous appellera	129
14.2. Principes généraux	129
14.3. RxJS	130
14.4. Programmation réactive en Angular	132
15. Tester ton application	134
15.1. Tester c'est douter	134
15.2. Tests unitaires	134
15.3. Dépendances factices	140
15.4. Tester des composants	142
15.5. Tester avec des templates ou providers factices	145
15.6. Des tests unitaires plus simples et plus propres avec <code>ngx-speculoos</code>	147
15.7. Tests de bout en bout	149
16. Envoyer et recevoir des données par HTTP	152
16.1. Obtenir des données (<code>provideHttpClient</code>)	152
16.2. Transformer des données	155
16.3. Options avancées	155
16.4. Intercepteurs	156
16.5. Contexte	157
16.6. Tests	158
17. Routeur	161
17.1. En route (<code>provideRouter</code>)	161
17.2. Navigation	164
17.3. Redirections	167
17.4. Quelle route pour une URL ?	167
17.5. Routes hiérarchiques	167
17.6. Guards	169
17.7. Resolvers	171

17.8. Événements de routage	173
17.9. Paramètres et data	174
17.10. Lier les paramètres et data aux inputs des composants	175
17.11. Chargement à la demande	176
18. Formulaires	179
18.1. Form, form, form-idable !	179
18.2. Formulaire piloté par le template	182
18.3. Formulaire piloté par le code	187
18.4. Un peu de validation	191
18.5. Erreurs et soumission	193
18.6. Un peu de style	196
18.7. Créer un validateur spécifique	197
18.8. Regrouper des champs	200
18.9. Réagir aux modifications	202
18.10. Mettre à jour seulement à la perte de focus ou à la soumission	203
18.11. FormArray et FormRecord	205
18.12. Des formulaires typés	207
18.13. Des messages d'erreur de validation super simples avec ngx-valdemort	209
18.14. Aller plus loin : définir ses propres contrôles de formulaires avec ControlValueAccessor	210
18.15. Conclusion	213
19. Les Zones et la magie d'Angular	214
19.1. AngularJS 1.x et le <i>digest cycle</i>	214
19.2. Angular et les zones	217
20. Compilation Angular : JiT (Just in Time) et AoT (Ahead of Time)	223
20.1. Génération de Code	223
20.2. Compilation En Avance (<i>Ahead of Time</i> , AoT)	225
21. Observables : utilisation avancée	227
21.1. subscribe, unsubscribe et <i>pipe async</i>	227
21.2. Le pouvoir des opérateurs	232
21.3. Construire son propre Observable	235
21.4. Gestion d'état avec un <i>store</i> (NgRx, NGXS, Akita et autres)	237
21.5. Conclusion	237
22. Composants et directives avancées	238
22.1. Transformation d'inputs	238
22.2. Requêtes de vue : ViewChild	239
22.3. Contenu : ng-content	242
22.4. Requêtes de contenu : ContentChild	244
22.5. Projection de contenu conditionnelle et contextuelle : ng-template et ngTemplateOutlet	247
22.6. À l'écoute de l'hôte : HostListener	250
22.7. Binding sur l'hôte : HostBinding	251
23. Les modules Angular	254

23.1. Une unité de compilation	254
23.2. Composition de modules	255
23.3. Modules fonctionnels associés au routage	256
24. Internationalisation	258
24.1. La <i>locale</i>	258
24.2. La devise par défaut	260
24.3. Traduire du texte	261
24.4. Processus et outillage	262
24.5. Traduire les messages dans le code	268
24.6. Pluralisation	269
24.7. Bonnes pratiques	270
25. Performances	272
25.1. Premier chargement	272
25.2. Rechargement	276
25.3. Profilage	277
25.4. Performances à l'exécution	278
25.5. NgZone	291
26. Signaux	299
26.1. Les raisons derrière les signaux	299
26.2. API des signaux	299
26.3. Signaux, composants et détection des changements	302
26.4. Partager un signal entre composants	304
26.5. Fuites mémoire	305
26.6. Signaux et interopérabilité RxJS	306
26.7. Composants basés sur les signaux	307
27. Livrer en production	309
27.1. Environnements et configurations	309
27.2. strictTemplates	311
27.3. Assembler ton application	312
27.4. Configuration du serveur	312
27.5. Conclusion	313
28. Ce n'est qu'un au revoir	314
Annexe A: Historique des versions	317
A.1. v16.2.0 - 2023-08-10	317
A.2. v16.1.0 - 2023-06-14	317
A.3. v16.0.0 - 2023-05-17	317
A.4. v15.2.0 - 2023-02-23	317
A.5. v15.1.0 - 2023-01-11	318
A.6. v15.0.0 - 2022-11-16	318
A.7. v14.2.0 - 2022-08-26	319
A.8. v14.1.0 - 2022-07-21	319

A.9. v14.0.0 - 2022-06-03	319
A.10. v13.3.0 - 2022-03-16	319
A.11. v13.2.0 - 2022-01-27	319
A.12. v13.1.0 - 2021-12-10	320
A.13. v13.0.0 - 2021-11-04	320
A.14. v12.2.0 - 2021-08-05	320
A.15. v12.1.0 - 2021-06-25	320
A.16. v12.0.0 - 2021-05-13	320
A.17. v11.2.0 - 2021-02-12	321
A.18. v11.1.0 - 2021-01-21	321
A.19. v11.0.0 - 2020-11-12	321
A.20. v10.2.0 - 2020-10-22	321
A.21. v10.1.0 - 2020-09-03	321
A.22. v10.0.0 - 2020-06-25	321
A.23. v9.1.0 - 2020-03-26	322
A.24. v9.0.0 - 2020-02-07	322
A.25. v8.2.0 - 2019-08-01	323
A.26. v8.1.0 - 2019-07-02	323
A.27. v8.0.0 - 2019-05-29	323
A.28. v7.2.0 - 2019-01-09	324
A.29. v7.1.0 - 2018-11-27	324
A.30. v7.0.0 - 2018-10-25	325
A.31. v6.1.0 - 2018-07-26	325
A.32. v6.0.0 - 2018-05-04	326
A.33. v5.2.0 - 2018-01-10	327
A.34. v5.0.0 - 2017-11-02	327
A.35. v4.3.0 - 2017-07-16	328
A.36. v4.2.0 - 2017-06-09	329
A.37. v4.0.0 - 2017-03-24	329
A.38. v2.4.4 - 2017-01-25	331
A.39. v2.2.0 - 2016-11-18	331
A.40. v2.0.0 - 2016-09-15	331
A.41. v2.0.0-rc.5 - 2016-08-25	332
A.42. v2.0.0-rc.0 - 2016-05-06	333
A.43. v2.0.0-alpha.47 - 2016-01-15	335

Chapter 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

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.



Cet ebook utilise Angular version 16.2.2 dans les exemples.

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*.

Nous avons maintenant 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 est (gratuitement) mis à jour avec chacune des versions majeures, il est désormais nommé "Deviens un Ninja avec Angular" (sans aucun numéro).

Chapter 2. Une rapide introduction à ECMAScript 2015+

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.

En 2015, une nouvelle version de cette spécification a été validée : ECMAScript 2015, ES2015, ou ES6, puisque c'est la sixième version de cette spécification. Et nous avons depuis une nouvelle version chaque année (ES2016, ES2017, etc.), avec à chaque fois quelques nouvelles fonctionnalités. Je l'appellerai désormais systématiquement ES2015, parce que c'est son petit nom le plus populaire, ou ES2015+ pour parler de ES2015, ES2016, ES2017, etc. 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 ES2015+. Ainsi, nous allons consacrer ce chapitre à découvrir ES2015+, 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 ES2015+, 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

La sixième version de la spécification a atteint son état final en 2015. Il est donc supporté par les navigateurs modernes, mais il y a encore des navigateurs qui ne supportent pas toute la spécification, ou qui la supportent seulement partiellement. Et bien sûr, avec une spécification maintenant annuelle (ES2016, ES2017, etc.), certains navigateurs seront toujours en retard. 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 ES2015+ ont hâte de l'utiliser dans leurs applications, la communauté a trouvé une solution : un transpileur.

Un transpileur prend du code source ES2015+ 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 ES2015+ depuis le navigateur. En 2015, il y avait deux outils principaux pour transpiler de l'ES2015+ :

- [Traceur](#), un projet Google, historiquement le premier, mais maintenant non-maintenu.
- [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.

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 aujourd’hui, on aurait donc tendance à te le conseiller. Le projet est maintenant le standard de-facto.

Si tu veux jouer avec ES2015+, 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 ES2015+ 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 ES2015+.

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;
}
```

ES2015 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. ES2015 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. Raccourcis pour la création d'objets

Ce n'est pas un nouveau mot-clé, mais ça peut te faire tiquer en lisant du code ES2015. 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 };
}
```

Tu peux aussi utiliser un autre raccourci, quand tu veux déclarer une méthode dans un objet :

```
function createPony() {
  return {
    run: () => {
      console.log('Run!');
    }
  };
}
```

qui peut être simplifié en :

```
function createPony() {
  return {
    run() {
      console.log('Run!');
    }
  };
}
```

2.5. Affectations déstructurées

Celui-là aussi peut te faire tiquer en lisant du code ES2015. 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 ES2015, 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` renvoyée 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. ES2015 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.



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);
}

```

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

ES2015 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.

ES2015 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'ES2015. 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 : ES2015 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 ES2015, 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 ES2015.

```
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(new Error(`Bad status ${response.status}`));
    }
  });
};
```

```
    } 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 ES2015 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. Async/await

Nous discutions des promesses précédemment, et il est intéressant de connaître un autre mot-clé introduit pour les gérer de façon plus synchrone : *await*.

Cette fonctionnalité n'est pas introduite par ECMAScript 2015 mais par ECMAScript 2017, et pour utiliser *await*, ta fonction doit être marquée comme *async*. Quand tu utilises le mot-clé *await* devant une promesse, tu pauses l'exécution de la fonction *async*, attends la résolution de la promesse, puis reprends l'exécution de la fonction *async*. La valeur renournée est la valeur résolue par la promesse.

On peut donc écrire notre exemple précédent en utilisant *async/await* comme ceci :

```

async function getUserRightsAndUpdateMenu() {
  // getUser is a promise
  const user = await getUser(login);
  // getRights is a promise
  const rights = await getRights(user);
  updateMenu(rights);
}
await getUserRightsAndUpdateMenu();

```

Et notre code a l'air complètement synchrone ! Une autre fonctionnalité assez cool de `async/await` est la possibilité d'utiliser un simple `try/catch` pour gérer les erreurs :

```
async function getUserRightsAndUpdateMenu() {
  try {
    // getUser is a promise
    const user = await getUser(login);
    // getRights is a promise
    const rights = await getRights(user);
    updateMenu(rights);
  } catch (e) {
    // will be called if getUser, getRights or updateMenu fails
    console.log(e);
  }
}
await getUserRightsAndUpdateMenu();
```

Note que, même si le code ressemble à du code synchrone, il reste asynchrone. L'exécution de la fonction est mise en pause puis reprise, mais, comme avec les callbacks, cela ne bloque pas le fil d'exécution : les autres événements peuvent être gérés pendant que l'exécution est mise en pause.

2.12. Set et Map

On va faire court : on a maintenant de vraies collections en ES2015. 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.

2.13. 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>`;
```

Une dernière fonctionnalité est la possibilité de les "tagger". Tu peux définir une fonction, et l'appliquer sur une chaîne de caractères template. Ici `askQuestion` ajoute un point d'interrogation à la fin de la chaîne de caractère :

```
const askQuestion = strings => strings + '?';  
const template = askQuestion`Hello there`;
```

Mais quelle est la différence avec une fonction classique alors ? Une fonction de tag reçoit en fait plusieurs paramètres :

- un tableau des morceaux statiques de la chaîne de caractères
- les valeurs résultant de l'évaluation des expressions

Par exemple, si l'on a la chaîne de caractère template contenant les expressions suivante :

```
const person1 = 'Cedric';
```

```
const person2 = 'Agnes';
const template = 'Hello ${person1}! Where is ${person2}?';
```

alors la fonction de tag reçoit les différents morceaux statiques et dynamiques. Ici nous avons une fonction de tag qui passe en majuscule les noms des protagonistes :

```
const uppercaseNames = (strings, ...values) => {
  // `strings` is an array with the static parts ['Hello ', '! Where is ', '?']
  // `values` is an array with the evaluated expressions ['Cedric', 'Agnes']
  const names = values.map(name => name.toUpperCase());
  // `names` now has ['CEDRIC', 'AGNES']
  // let's merge the `strings` and `names` arrays
  return strings.map((string, i) => `${string}${names[i] ? names[i] : ''}`).join('');
}
const result = uppercaseNames`Hello ${person1}! Where is ${person2}?`;
// returns 'Hello CEDRIC! Where is AGNES?'
```

Passons maintenant à l'un des grands changements introduits : les modules.

2.14. 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. Node.js 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), utilisée par RequireJS. Mais aucun n'était un vrai standard, ce qui nous conduit à des débats incessants sur la meilleure solution.

ES2015+ 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'ES2015+ et auteur de la spécification du langage) voulait une syntaxe simple (c'est indéniablement l'atout 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.

2.15. Conclusion

Voilà qui conclue notre rapide introduction à ES2015+. On a zappé quelques parties, mais si tu as bien assimilé ce chapitre, tu n'auras aucun problème à coder ton application en ES2015+. 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 !

Chapter 3. Un peu plus loin qu'ES2015+

3.1. Types dynamiques, statiques et optionnels

Tu sais probablement que les applications Angular sont écrites en 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 ECMAScript 4, qui a été abandonnée. Au départ ils annoncèrent AtScript, un sur-ensemble d'ES2015 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, tu auras probablement envie de l'utiliser dans toutes tes applications. 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, 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 !

C'est pourquoi nous allons passer un peu de temps à apprendre TypeScript (TS). Angular est clairement construit pour tirer parti de celui-ci, 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 !

Chapter 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'ES2015+, 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 ponyNumber: number = 0;  
const ponyName: 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  
}
```

Depuis TypeScript 2.4, tu peux même donner une valeur sous forme de chaîne de caractères :

```
enum Position {  
    First = 'First',  
    Second = 'Second',  
    Other = 'Other'  
}
```

Cependant, pour être tout à fait honnêtes, nous n'utilisons pas d'enum dans nos projets : on utilise des unions de types. Elles sont plus simples et couvrent à peu près les mêmes usages :

```
let color: 'blue' | 'red' | 'green';  
// we can only give one of these values to 'color'  
color = 'blue';
```

TypeScript permet même de créer ses propres types, on pourrait donc faire comme suit :

```
type Color = 'blue' | 'red' | 'green';  
const ponyColor: Color = 'blue';
```

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;  
}
```

Tu verras que l'on utilise très souvent les interfaces dans le livre pour représenter nos entités.

On utilise également les interfaces pour représenter nos modèles métiers dans nos projets. Généralement, on ajoute un suffixe `Model` pour le montrer de façon claire. Il est alors très facile de créer une nouvelle entité :

```
interface PonyModel {  
    name: string;  
    speed: number;  
}  
const pony: PonyModel = { name: 'Light Shoe', speed: 56 };
```

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. La définition de cette interface serait :

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

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

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

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: number): void {
        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}`);
    }
}
// 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: number): void {
        logger.log(`pony runs ${meters}`);
    }

    eat(): void {
        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'ES2015+, c'est seulement possible en TypeScript.

```
class SpeedyPony {
    speed = 10;

    run(): void {
        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(): void {
    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(): void {
    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 définitions pour les types et les fonctions exposés par les bibliothèques JavaScript les plus populaires.

Les fichiers contenant ces définitions 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, []);
// 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 définitions pour une dépendance si elles sont packagées avec la dépendance elle-même. 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

Cette fonctionnalité a été ajoutée en TypeScript 1.5, en partie 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 (possible 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: number) {
    // call API
  }
}
```

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

```
const Log = () => {
  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', template: '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 "ES2015+", 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 à partir de maintenant, car Angular, et tout l'outillage autour, est vraiment conçu pour en tirer parti.

Chapter 5. TypeScript avancé

Si tu commences juste ton apprentissage de TypeScript, tu peux sauter sans problème ce chapitre dans un premier temps et y revenir plus tard. Ce chapitre est là pour montrer des utilisations plus avancées de TypeScript, qui n'auront vraiment de sens que si le langage t'est déjà familier.

5.1. readonly

Tu peux utiliser le mot-clé `readonly` (lecture seule) pour marquer une propriété d'un objet ou d'une classe comme étant... en lecture seule. De cette façon, le compilateur refusera de compiler du code qui tente d'assigner une nouvelle valeur à cette propriété :

```
interface Config {  
    readonly timeout: number;  
}  
  
const config: Config = { timeout: 2000 };  
// `config.timeout` is now readonly and can't be reassigned
```

5.2. keyof

Le mot-clé `keyof` peut être utilisé pour un type représentant l'union de tous les noms des propriétés d'un autre type. Par exemple, si tu as une interface `PonyModel` :

```
interface PonyModel {  
    name: string;  
    color: string;  
    speed: number;  
}
```

Tu veux écrire une fonction qui renvoie la valeur d'une propriété. Voici une première implémentation naïve :

```
function getProperty(obj: any, key: string): any {  
    return obj[key];  
}  
  
const pony: PonyModel = {  
    name: 'Rainbow Dash',  
    color: 'blue',  
    speed: 45  
};  
const nameValue = getProperty(pony, 'name');
```

Il y a deux problèmes ici :

- tu peux donner n'importe quelle valeur au paramètre `key`, même une clé qui n'existe pas dans `PonyModel`.
- le type de retour étant `any`, nous perdons beaucoup d'information de typage.

C'est ici que `keyof` peut être utile. `keyof` permet de lister toutes les clés d'un type :

```
type PonyModelKey = keyof PonyModel;
// this is the same as ''name''|'speed'|'color'
let property: PonyModelKey = 'name'; // works
property = 'speed'; // works
// key = 'other' would not compile
```

On peut donc utiliser ce type pour rendre `getProperty` plus strictement typée, en déclarant que :

- le premier paramètre est de type `T` ;
- le second paramètre est de type `K`, qui est une clé de `T`.

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const pony: PonyModel = {
  name: 'Rainbow Dash',
  color: 'blue',
  speed: 45
};
// TypeScript infers that `nameValue` is of type `string`!
const nameValue = getProperty(pony, 'name');
```

On fait ici d'une pierre, deux coups :

- `key` peut maintenant seulement être une propriété existante de `PonyModel` ;
- le type de retour sera déduit par TypeScript (ce qui est sacrément cool !).

Maintenant voyons comment nous pouvons utiliser `keyof` pour aller encore plus loin.

5.3. Mapped type

Disons que tu veux construire un type qui a exactement les mêmes propriétés que `PonyModel`, mais tu veux que chaque propriété soit optionnelle. Tu peux bien sûr le définir manuellement :

```
interface PartialPonyModel {
  name?: string;
  color?: string;
  speed?: number;
}
```

```
const pony: PartialPonyModel = {
  name: 'Rainbow Dash'
};
```

Mais on peut faire quelque chose de plus générique avec un *mapped type*, un "type de transformation" :

```
type Partial<T> = {
  [P in keyof T]?: T[P];
};

const pony: Partial<PonyModel> = {
  name: 'Rainbow Dash'
};
```

Le type `Partial` est une transformation qui applique le modificateur `?` à chaque property du type ! En fait, `Partial` est suffisamment fréquent pour qu'il soit inclus dans TypeScript depuis la version 2.1, et il est déclaré exactement comme ceci dans la bibliothèque standard.

TypeScript offre également d'autres types de transformation.

5.3.1. Readonly

`Readonly` rend toutes les propriétés d'un objet `readonly` :

```
const pony: Readonly<PonyModel> = {
  name: 'Rainbow Dash',
  color: 'blue',
  speed: 45
};
// all properties are `readonly`
```

5.3.2. Pick

`Pick` t'aide à construire un type avec seulement quelques-unes des propriétés d'origine :

```
const pony: Pick<PonyModel, 'name' | 'color'> = {
  name: 'Rainbow Dash',
  color: 'blue'
};
// 'pony' can't have a 'speed' property
```

5.3.3. Record

`Record` t'aide à construire un type avec les mêmes propriétés et un autre type pour ces propriétés :

```

interface FormValue {
  value: string;
  valid: boolean;
}

const pony: Record<keyof PonyModel, FormValue> = {
  name: { value: 'Rainbow Dash', valid: true },
  color: { value: 'blue', valid: true },
  speed: { value: '45', valid: true }
};

```

Il y a [encore d'autres types](#), mais ceux-ci sont les plus utiles.

5.4. Union de types et gardien de types

Les unions de types sont très pratiques. Disons que ton application a des utilisateurs connectés et des utilisateurs anonymes, et que, parfois, tu dois faire une action différente selon le cas. Tu peux modéliser les entités comme ceci :

```

interface User {
  type: 'authenticated' | 'anonymous';
  name: string;
  // other fields
}

interface AuthenticatedUser extends User {
  type: 'authenticated';
  loggedSince: number;
}

interface AnonymousUser extends User {
  type: 'anonymous';
  visitingSince: number;
}

function onWebsiteSince(user: User): number {
  if (user.type === 'authenticated') {
    // this is a LoggedUser
    return (user as AuthenticatedUser).loggedSince;
  } else if (user.type === 'anonymous') {
    // this is an AnonymousUser
    return (user as AnonymousUser).visitingSince;
  }
  // TS doesn't know every possibility was covered
  // so we have to return something here
  return 0;
}

```

Je ne sais pas pour toi, mais je n'aime pas trop ces typages explicites `as ...`. Peut-on faire mieux ?

La première possibilité est d'utiliser un *type guard*, un gardien de types, une fonction spéciale dont le seul but est d'aider le compilateur TypeScript.

```
function isAuthenticated(user: User): user is AuthenticatedUser {
    return user.type === 'authenticated';
}

function isAnonymous(user: User): user is AnonymousUser {
    return user.type === 'anonymous';
}

function onWebsiteSince(user: User): number {
    if (isAuthenticated(user)) {
        // this is inferred as a LoggedUser
        return user.loggedSince;
    } else if (isAnonymous(user)) {
        // this is inferred as an AnonymousUser
        return user.visitingSince;
    }
    // TS still doesn't know every possibility was covered
    // so we have to return something here
    return 0;
}
```

C'est mieux ! Mais on a toujours besoin de retourner une valeur par défaut, même si nous avons couvert tous les cas.

On peut légèrement améliorer la situation si on abandonne les gardiens de types et que l'on utilise une union de types à la place.

```
interface BaseUser {
    name: string;
    // other fields
}

interface AuthenticatedUser extends BaseUser {
    type: 'authenticated';
    loggedSince: number;
}

interface AnonymousUser extends BaseUser {
    type: 'anonymous';
    visitingSince: number;
}

type User = AuthenticatedUser | AnonymousUser;
```

```

function onWebsiteSince(user: User): number {
  if (user.type === 'authenticated') {
    // this is inferred as a LoggedUser
    return user.loggedSince;
  } else {
    // this is narrowed as an AnonymousUser
    // without even testing the type!
    return user.visitingSince;
  }
  // no need to return a default value
  // as TS knows that we covered every possibility!
}

```

C'est encore mieux, car TypeScript comprend automatiquement le type utilisé dans la branche `else`.

Parfois tu sais que ce modèle va grandir dans le futur, et que d'autres cas devront être gérés. Par exemple, si tu ajoutes un `AdminUser`. Dans ce cas, on peut utiliser un `switch`. Un `switch` sur une union de types ne compilera pas si l'un des cas n'est pas géré. Donc introduire notre `AdminUser`, ou un autre type plus tard, ajouterait automatiquement des erreurs de compilation dans tous les endroits de notre code où nous devons les gérer !

```

interface AdminUser extends BaseUser {
  type: 'admin';
  adminSince: number;
}

type User = AuthenticatedUser | AnonymousUser | AdminUser;

function onWebsiteSince(user: User): number {
  switch (user.type) {
    case 'authenticated':
      return user.loggedSince;
    case 'anonymous':
      return user.visitingSince;
    case 'admin':
      // without this case, we could not even compile the code
      // as TS would complain that all possible paths are not returning a value
      return user.adminSince;
  }
}

```

J'espère que ces patterns vous aideront dans vos projets. Penchons-nous maintenant sur les Web Components.

Chapter 6. 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.

6.1. 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 quelque 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 trois spécifications :

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

- Template

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

6.2. 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 `customElements.define` :

```
class PonyComponent extends HTMLElement {
  constructor() {
    super();
    console.log("I'm a pony!");
  }
}

customElements.define('ns-pony', PonyComponent);
```

Et ensuite l'utiliser avec :

```
<ns-pony></ns-pony>
```

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 :

```
class PonyComponent extends HTMLElement {
  constructor() {
    super();
    console.log("I'm a pony!");
  }
}

/**
 * This is called when the component is inserted
 */
connectedCallback() {
  this.innerHTML = '<h1>General Soda</h1>';
```

```
}
```

```
}
```

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.

6.3. 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 :

```
class PonyComponent extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });
    const title = document.createElement('h1');
    title.textContent = 'General Soda';
    shadow.appendChild(title);
  }
}
```

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>`.

6.4. 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-template">
  <style>
    h1 {
      color: orange;
    }
  </style>
  <h1>General Soda</h1>
</template>
```

```
class PonyComponent extends HTMLElement {
  constructor() {
    super();
    const template = document.querySelector('#pony-template');
    const clonedTemplate = document.importNode(template.content, true);
    const shadow = this.attachShadow({ mode: 'open' });
    shadow.appendChild(clonedTemplate);
  }
}
```

6.5. Les bibliothèques basées sur les Web Components

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](#), première tentative de la part de Google ;
- [LitElement](#), projet plus récent de l'équipe Polymer ;

- [X-tag](#) de Mozilla et Microsoft ;
- [Stencil](#).

Je ne vais pas rentrer dans les détails, mais tu peux facilement utiliser un composant 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 -->
<script src="google-map.js"></script>

<!-- 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://www.webcomponents.org/>.

Tu peux faire plein de trucs cools avec LitElement et les autres frameworks similaires, 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. Il est même possible d'exporter nos composants Angular sous forme de Web Components, grâce à [Angular Elements](#).

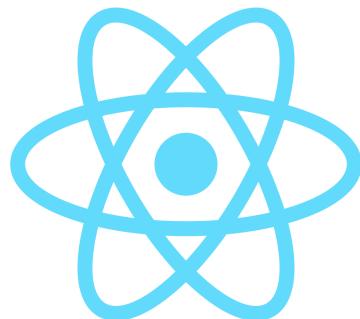
Chapter 7. La philosophie d'Angular

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.

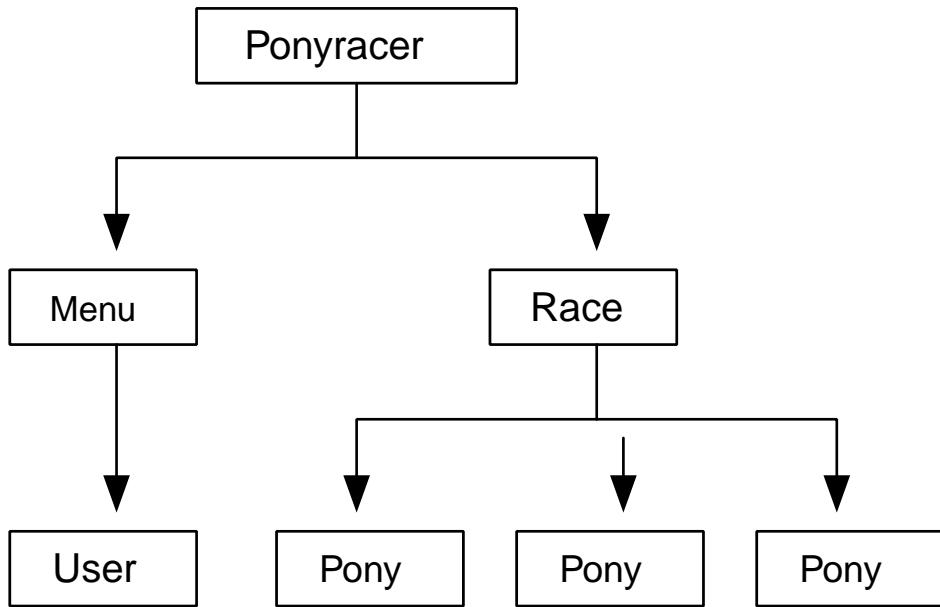
Cette orientation composant est largement partagée par de nombreux frameworks front-end : c'est le cas depuis le début de [React](#), le framework tendance de Facebook ; [Ember](#) et [AngularJS](#) ont leur propre façon de faire quelque chose de similaire ; et les petits nouveaux [Svelte](#) ou [Vue.js](#) 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 ([Ponyracer](#)), affichant un menu ([Menu](#)) avec l'utilisateur connecté ([User](#)) et une vue enfant ([Race](#)), affichant, 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 : ES2015+. Ainsi tu peux écrire tes composants en ES5 (pas cool !) ou en ES2015+ (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';
import { NgFor } from '@angular/common';
import { PonyComponent } from './components';

@Component({
  selector: 'ns-race',
  templateUrl: './race.component.html',
  standalone: true,
  imports: [NgFor, PonyComponent]
})
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 mêmes 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 ES2015), ou bien tu apprendras à te passer de ces modules en faisant de tes composants des composants *standalone*.

Tu pourras aussi prendre des bibliothèques fournies par la communauté, et les utiliser simplement dans ton application pour bénéficier de leurs fonctionnalités.

De telles bibliothèques fournissent 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, ta première application, 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.

Chapter 8. Commencer de zéro

Commençons par créer notre première application Angular et notre premier composant, avec un minimum d'outillage.

8.1. Node.js et NPM

Aujourd'hui, pratiquement tous les outils JavaScript modernes sont faits pour Node.js et NPM. 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`).

8.2. Angular CLI

Nous *pourrions* tout installer à la main par nous-même, en commençant avec un projet TypeScript, puis installer toutes les dépendances nécessaires, etc.

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 ;
- peut-être un *linter* pour vérifier la qualité du code ;
- peut-être un pré-processeur CSS ;
- un outil de construction, pour orchestrer différentes tâches (compiler, tester, packager, etc.).

Mais c'est un peu pénible à faire soi-même, surtout quand il y a taaaaaalement d'outils à apprendre auparavant.

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 eu 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 en évolution continue, avec une équipe Google dédiée travaillant dessus, et l'améliorant chaque jour. C'est maintenant la façon recommandée et le standard *de facto* pour créer et construire des applications Angular. Alors essayons cette CLI, et découvrons la tonne de fonctionnalités qu'elle embarque !



Si tu veux, tu peux suivre notre exercice en ligne [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, et va plus loin que ce que nous voyons dans ce chapitre.

Commençons par installer Angular CLI, et générerons une nouvelle application avec la commande `ng new`. Si tu veux utiliser exactement la même version de la CLI que nous (`16.2.0`), tu peux utiliser `npm install -g @angular/cli@16.2.0` à la place.

```
npm install -g @angular/cli
```

TODO: change the script below to generate a standalone application

```
ng new ponyracer --prefix ns --defaults --standalone
```

Cela va créer un squelette de projet dans un nouveau dossier nommé `ponyracer`. Depuis ce répertoire, tu peux démarrer l'application avec :

```
ng serve
```

Cela va démarrer un petit serveur HTTP localement, avec recharge à chaud. Ainsi, à chaque modification de fichier, l'application sera reconstruite et le navigateur se rechargera automatiquement.

Tada ! Tu as ta première application qui tourne ! 🎉



Peut-être te demandes-tu pourquoi on a passé l'option `--standalone` à la commande `ng new`. Dans sa version 15, Angular a introduit une nouvelle fonctionnalité appelée *standalone components*. Jusque-là, les composants devaient être déclarés dans des modules Angular, qui constituent un concept assez complexe à comprendre et utiliser correctement, surtout quand on démarre avec Angular. Utiliser des composants *standalone* nous permet d'éviter d'avoir à créer des modules Angular et rend les choses plus simples, particulièrement pour les débutants. Nous pensons que les composants *standalone*s vont rapidement s'imposer comme la manière standard d'écrire des applications Angular, donc nous avons choisi d'utiliser cette option. Nous expliquerons néanmoins les modules Angular plus en détails dans un prochain chapitre, parce que, à défaut d'en créer vous-mêmes, vous aurez très certainement besoin de comprendre leur fonctionnement et d'en utiliser dans votre travail quotidien. Pour l'instant en tout cas, nul besoin de trop s'en

préoccuper.

8.3. Structure de l'application

Plongeons-nous dans le code généré.

Tu peux ouvrir le projet dans ton 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. Choisis ton favori : Webstorm, Visual Studio Code... Ils ont tous un bon support de TypeScript.



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 les 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. Comme nous utilisons des *source maps*, tu peux voir le code TypeScript directement dans ton navigateur, et même debugger ton application en positionnant des points d'arrêt directement dedans.

Tu devrais voir tout un tas de fichiers de configuration dans le répertoire racine : bienvenue dans le JavaScript Moderne !

Le premier que tu reconnais peut-être est le fichier `package.json` : C'est là que sont définies les dépendances de l'application. Tu peux regarder à l'intérieur, il devrait contenir les dépendances suivantes (entre autres) :

- les différents packages `@angular`.
- `rxjs`, une bibliothèque vraiment cool pour la programmation réactive. On consacrera un chapitre entier à ce sujet, et à RxJS en particulier.
- `zone.js`, qui assure la plomberie pour détecter les changements (on y reviendra aussi plus tard).
- quelques dépendances pour développer l'application, comme la CLI, TypeScript, des bibliothèques de tests, des *typings*...

TypeScript lui-même a un fichier de configuration `tsconfig.json` (et un autre appelé `tsconfig.app.json`), qui stocke les options de compilation. Comme on l'a vu dans les chapitres précédents, on va utiliser TypeScript avec des décorateurs (d'où les deux options dont le nom contient *decorator*). 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 JavaScript généré et le code TypeScript originel. Ces *source maps* sont utilisés par le navigateur pour te permettre de debugger le code JavaScript qu'il exécute en parcourant le code TypeScript originel que tu as écrit.

Les projets TypeScript sont fréquemment utilisés avec ESLint, un *linter*, pratique pour vérifier si ton code suit un ensemble de bonnes pratiques. ESLint a ses propres options, stockées dans `.eslintrc.json`, où tu peux ajouter/supprimer certaines règles.

Angular CLI a lui-même son fichier de configuration `angular.json` si tu veux changer quelques-unes des options par défaut.

L'ebook utilise Angular version **16.2.2** pour les exemples. Angular CLI va probablement installer la version la plus récente, qui peut ne pas être exactement la même. Pour utiliser la même version que nous, tu peux remplacer la version dans le `package.json` par **16.2.2** pour chacun des packages Angular. Cela peut t'épargner quelques maux de tête ! Ou, encore mieux, suis notre exercice en ligne gratuit [Getting Started](#) 🎉 qui est toujours à jour et à l'épreuve du feu !



Maintenant que nous avons parcouru la configuration, regardons le code applicatif.

8.4. Notre premier composant *standalone*

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). La CLI a d'ores et déjà créé un composant pour nous `src/app/app.component.ts`. Jetons-y un œil :

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

@Component({
  selector: 'ns-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'ponyracer';
}
```

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 comme tu peux le voir en haut du fichier.

Quand tu écris de nouveaux composants, n'oublie pas d'importer ce décorateur `Component`. Tu l'oublieras peut-être 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.

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

@Component({
})
export class AppComponent {
  title = 'ponyracer';
}
```

Le décorateur `@Component` attend un objet de configuration. On verra plus tard en détails ce qu'on

peut y configurer, mais pour le moment commençons par expliquer la propriété `selector`. Elle indique à Angular ce qu'il faudra chercher dans nos pages HTML. À chaque fois que Angular trouve un élément dans notre HTML qui correspond au sélecteur défini dans notre composant, Angular crée une instance de ce composant, et remplace le contenu de l'élément par le template de notre composant.

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

@Component({
  selector: 'ns-root',
})
export class AppComponent {
  title = 'ponyracer';
}
```

Donc ici, chaque fois que notre HTML contiendra un élément `<ns-root></ns-root>`, Angular créera une nouvelle instance de notre classe `AppComponent`.

Il y a une convention de nommage clairement établie, et appliquée par Angular CLI. Les classes de composant se terminent par `Component`, et sont définies dans des fichiers dont le nom se terminent en `.component.ts`. Angular recommande aussi d'utiliser un préfixe dans les sélecteurs des composants, pour éviter un conflit de nom avec des composants externes. Par exemple, comme notre société se nomme Ninja Squad, on a choisi comme préfixe `ns`. Notre composant poney aura donc comme sélecteur `ns-pony`. Tu peux configurer Angular CLI pour qu'il ajoute un préfixe automatiquement à chaque composant généré. Si tu te rappelles bien, lorsqu'on a créé le projet avec `ng new`, on a passé l'option `--prefix ns`. C'est ce que cette option permet : elle configure le projet afin que les composants soient générés avec le préfixe `ns`.



Un composant doit aussi avoir un template. On peut avoir un template *inline* (directement dans le décorateur) ou dans un autre fichier comme le fait la CLI :

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

@Component({
  selector: 'ns-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'ponyracer';
}
```

Le HTML correspondant est défini dans `app.component.html`, avec tout un tas d'éléments statiques, à l'exception du premier `h1` :

```
<span>{{ title }} app is running!</span>
```

Enfin, tu peux voir dans le code généré que le composant a deux options supplémentaires{nbs}: **standalone** et **imports**:

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

@Component({
  selector: 'ns-root',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'ponyracer';
}
```

La propriété **standalone: true** est celle qui rend notre composant *standalone*. Cela signifie que nous n'aurons pas besoin de le déclarer dans un module Angular pour pouvoir l'utiliser.

L'autre propriété, **imports: [CommonModule]** n'est pas toujours nécessaire. Son rôle est de donner à Angular la liste des autres composants, directives et pipes qui peuvent être utilisés à l'intérieur du template de notre composant. La plupart des composants que nous créons utilisent des pipes et des directives fournies par Angular. Ces pipes et directives communément utilisés sont déclarés dans le module Angular **CommonModule**. Importer **CommonModule** dans la configuration rend ainsi possible l'utilisation des pipes et directives de **CommonModule** dans le template de notre composant.

Nous avons expliqué le principe des modules ES2015+ et TS dans les premiers chapitres, qui permettent de définir des imports et des exports. Le compilateur TypeScript, lorsqu'il voit une interface **Race** utilisée dans le code source d'un fichier TypeScript, doit savoir où trouver la définition de cette interface **Race**. C'est la raison pour laquelle on doit l'importer.

Et bien la propriété **imports** du décorateur **@Component** joue un rôle similaire : si Angular rencontrait une balise **<ns-race>** dans le template de **AppComponent**, il faudrait qu'il sache où et comment est défini le composant correspondant, **RaceComponent**. Pour lui indiquer où trouver la définition du composant, nous devons ajouter **RaceComponent** aux imports du décorateur de **AppComponent**.

8.5. Démarrer l'application

Enfin, on doit démarrer l'application, avec la fonction **bootstrapApplication**. Angular CLI génère par défaut un fichier séparé contenant cette logique de démarrage : **main.ts**.

```
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';
```

```
bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));
```

Comme tu peux le voir, elle prend en argument le composant racine de l'application : [AppComponent](#).

Youpi ! Mais attends voir. On doit bien servir un fichier HTML à nos utilisateurs, non ?

La CLI a créé un fichier [index.html](#) pour nous, qui est la seule page de notre application (d'où le nom *SPA, single page application*). Tu te demandes peut-être comment cela peut fonctionner, car il ne contient aucun élément [script](#).

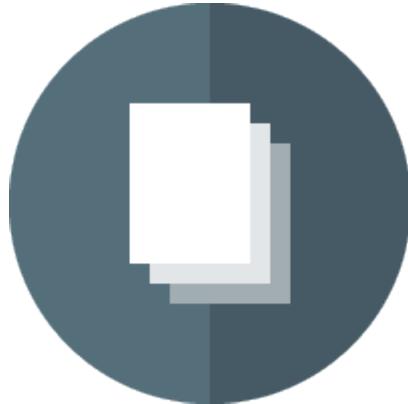
Quand on lance [ng serve](#), la CLI appelle le compilateur TypeScript. Le compilateur génère des fichiers JavaScript. La CLI va alors les assembler (*bundle*) et ajouter les éléments [script](#) nécessaires au fichier [index.html](#) (en utilisant pour cela [Webpack](#)).

Tu devrais maintenant avoir une meilleure compréhension des différentes parties de cette première application Angular. Elle n'est pas encore réellement dynamique, et on aurait pu faire la même chose en une seconde dans une page HTML statique, je te l'accorde. Alors jetons-nous sur les chapitres suivants, et apprenons tout de l'injection de dépendances et du système de templates.

Chapter 9. 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 en simplifiant notre premier composant :

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

@Component({
  selector: 'ns-root',
  template: '<h1>PonyRacer</h1>',
  standalone: true
})
export class AppComponent {}
```

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: 'ns-root',
  template: '<h1>PonyRacer</h1>',
  standalone: true
})
export class AppComponent {
  numberOfWorkers = 146;
}
```

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

9.1. Interpolation

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

Un exemple rapide :

```
@Component({
  selector: 'ns-root',
  template: `
    <h1>PonyRacer</h1>
    <h2>{{ numberOfUsers }} users</h2>
  `,
  standalone: true
})
export class AppComponent {
  numberOfUsers = 146;
}
```

Nous avons un composant `AppComponent`, qui sera activé dès qu'Angular tombera sur une balise `<ns-root>`. La classe `AppComponent` 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 :

```
<ns-root>
  <h1>PonyRacer</h1>
  <h2>146 users</h2>
</ns-root>
```

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

Quand Angular détecte un élément `<ns-root>` dans une page, il crée une instance de la classe `AppComponent`, et cette instance sera le contexte d'évaluation des expressions dans le template. Ici la classe `AppComponent` 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'est pas initialisée, 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: 'ns-root',
    template: `
      <h1>PonyRacer</h1>
      <h2>Welcome {{ user.name }}</h2>
    `,
    standalone: true
)
export class AppComponent {
  user = { name: 'Cédric' };
}

```

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

```

<ns-root>
  <h1>PonyRacer</h1>
  <h2>Welcome Cédric</h2>
</ns-root>

```

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: 'ns-root',
  // typo: users is not user!
  template: `
    <h1>PonyRacer</h1>
    <h2>Welcome {{ users.name }}</h2>
  `,
  standalone: true
)
export class AppComponent {
  user = { name: 'Cédric' };
}

```

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

```
error TS2339: Property 'users' does not exist on type 'AppComponent'
```

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: 'ns-root',
  // user is undefined
  // but the ?. will avoid the error
  template: `
    <h1>PonyRacer</h1>
    <h2>Welcome {{ user?.name }}</h2>
  `,
  standalone: true
})
export class AppComponent {
  user: { name: string } | undefined;
}
```

Et nous n'avons plus d'erreur ! Le `?`. est appelé "*optional chaining operator*" (opérateur de chaînage optionnel).

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>',
  standalone: true
})
export class RacesComponent {}
```

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

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

9.2. Utiliser d'autres composants dans nos templates

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

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

@Component({
  selector: 'ns-root',
  // added the RacesComponent component
  template: `
    <h1>PonyRacer</h1>
    <ns-races></ns-races>
  `,
  standalone: true
})
export class AppComponent {}
```

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 ? Angular ne connaît pas encore ce composant `RacesComponent`.

La solution est simple : il faut ajouter le `RacesComponent` aux `imports` du décorateur de `AppComponent`. De cette manière, lorsque Angular compilera le template de `AppComponent`, il recherchera un composant ayant le sélecteur `ns-races` parmi la liste des composants importés, il y trouvera le `RacesComponent`, et saura que c'est ce composant qu'il faut instancier et afficher.

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

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

@Component({
  selector: 'ns-root',
  template: `
    <h1>PonyRacer</h1>
    <ns-races></ns-races>
  `,
  standalone: true,
  // add RacesComponent to the imports of AppComponent
  imports: [RacesComponent]
})
export class AppComponent {}
```

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

Pour importer `RacesComponent` dans la classe `AppComponent`, tu dois exporter la classe `RacesComponent` dans son fichier source `races.component.ts` (relis le chapitre `modules ES2015` 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>',
  standalone: true
})
export class RacesComponent {}
```

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

```
<ns-root>
  <h1>PonyRacer</h1>
  <ns-races>
    <h2>Races</h2>
  </ns-races>
</ns-root>
```

9.3. *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 :

```

```

Cette syntaxe a un problème majeur : le navigateur va essayer de récupérer l'image dès qu'il lira l'attribut `src`. Tu peux constater l'échec : il réalisera une requête HTTP sur l'URL `{{ pony.avatar.url }}`, qui n'est pas une URL valide...

En AngularJS 1.x, il y avait une directive dédiée à ce problème : `ng-src`.

```

```

Utiliser `ng-src` au lieu de `src` solutionnait le problème, parce que le navigateur ignorerait 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>
```

9.4. 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>
  `,
  standalone: true
})
export class RacesComponent {
  races: Array<RaceModel> = [];

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

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

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

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 `AppComponent` ressemblera alors à :

```
@Component({
  selector: 'ns-root',
  template: `
    <h1>PonyRacer</h1>
    <ns-races (newRaceAvailable)="onNewRace()"></ns-races>
  `,
  standalone: true,
  imports: [RacesComponent]
})
export class AppComponent {
  onNewRace(): void {
    // 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 `AppComponent` 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: 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) {
  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.

9.5. 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 enchaîner 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.

9.6. 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.

9.7. 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 ``.

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 !

9.7.1. 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 dont nous avons discuté un peu plus tôt : `CommonModule`. 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 ne font pas l'objet d'un traitement de faveur : pour pouvoir les utiliser, un composant standalone doit les importer. L'exemple suivant importe explicitement `NgIf`. Tu pourrais choisir d'importer plutôt le `CommonModule` tout entier : cela permettrait au template d'utiliser toutes les directives et tous les pipes qu'il fournit.

```
import { Component } from '@angular/core';
import { RaceModel } from './race.model';
import { NgIf } from '@angular/common';

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

```
    races: Array<RaceModel> = [];
}
```

Il existe aussi une possibilité d'utiliser `else`:

```
import { Component } from '@angular/core';
import { RaceModel } from './race.model';
import { NgIf } from '@angular/common';

@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>
  `,
  standalone: true,
  imports: [NgIf]
})
export class RacesComponent {
  races: Array<RaceModel> = [];
}
```

9.7.2. 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';
import { RaceModel } from './race.model';
import { CommonModule } from '@angular/common';

@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>
  `,
  standalone: true,
  imports: [CommonModule]
})
export class RacesComponent {
  races: Array<RaceModel> = [{ 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>
```



dans l'exemple ci-dessus, qui utilise à la fois `NgIf` and `NgFor`, nous avons choisi d'importer `CommonModule`. Si tu préfères être plus explicite, tu peux à la place ajouter `NgIf` et `NgFor` au tableau `imports`. C'est principalement une question de préférence.

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

```
<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

9.7.3. 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.

9.8. 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.

9.8.1. 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`').

9.8.2. 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 classe avec le binding de propriété :

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

Ou plusieurs :

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

Tu peux également utiliser `ngClass` :

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

9.9. 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.

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';  
import { CommonModule } from '@angular/common';  
  
@Component({  
  selector: 'ns-ponies',
```

```

    template: ``,
    standalone: true,
    imports: [CommonModule]
})
export class PoniesComponent {}

```

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

Notre nouveau composant a une liste de poneys :

```

import { Component } from '@angular/core';
import { PonyModel } from '../pony.model';
import { CommonModule } from '@angular/common';

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

```

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

```

import { Component } from '@angular/core';
import { PonyModel } from '../pony.model';
import { CommonModule } from '@angular/common';

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

```

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

```

import { Component } from '@angular/core';
import { PonyModel } from '../pony.model';
import { CommonModule } from '@angular/common';

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

  refreshPonies(): void {
    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';
import { PonyModel } from '../pony.model';
import { CommonModule } from '@angular/common';

@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>
  `,
  standalone: true,
  imports: [CommonModule]
})
export class PoniesComponent {
  ponies: Array<PonyModel> = [{ name: 'Rainbow Dash' }, { name: 'Pinkie Pie' }];

  refreshPonies(): void {
    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.



Essaye notre [quiz](#) 🐾 et 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.

Chapter 10. Créer des composants et directives

10.1. 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 ?

10.2. 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.

Cela a donc du 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.

10.2.1. 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]',
  standalone: true
})
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>',
  standalone: true,
  imports: [DoNothingDirective]
})
export class TestComponent {}
```

Un sélecteur plus complexe pourrait ressembler à :

```
@Directive({
  selector: 'div.loggable[logText]:not([notLoggable=true])',
  standalone: 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 ! :)



Les sélecteurs CSS à base de descendants, de frères, ou d'identifiants, et les pseudo-sélecteurs (autres que `:not`) ne sont pas supportés.

Bien ! Maintenant on sait déclarer une directive. Il est temps d'en faire une qui sert vraiment à quelque chose !

10.2.2. 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'],
  standalone: true
})
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'],
  standalone: true
})
export class SimpleTextWithSetterDirective {
  set text(value: string) {
    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 (généralement préférée) :

```
<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: binding** :

```
@Directive({
  selector: '[loggable]',
  inputs: ['logText'],
  standalone: true
})
export class SameNameInputDirective {
  set logText(value: string) {
    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]',
  standalone: true
})
export class InputDecoratorDirective {
  @Input('logText') text = '';
}
```

ou, si la propriété et le binding ont le même nom :

```
@Directive({
  selector: '[loggable]',
  standalone: true
})
export class SameNameInputDecoratorDirective {
  @Input() logText = '';
}
```

TypeScript ne permet pas d'avoir un champ et un setter avec le même nom. 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]',
  standalone: true
})
export class InputDecoratorOnSetterDirective {
  @Input('logText')
  set text(value: string) {
    console.log(value);
  }
}
```

et si le setter et le binding ont le même nom :

```
@Directive({
```

```

    selector: '[loggable]',
    standalone: true
})
export class SameNameInputDecoratorOnSetterDirective {
  @Input()
  set logText(value: string) {
    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.

Angular v16 a ajouté la possibilité de marquer un input comme étant obligatoire, avec `@Input({ required: true })`:

```
@Input({ required: true }) pony!: PonyModel;
```

Dans ce cas, si le composant parent ne passe pas l'input, alors le compilateur va lever une erreur.

Note que TypeScript se plaindra peut-être si tu n'initialises pas le champ avec une valeur (si tu as activé la configuration `strict` dans ton fichier `tsconfig`). Je donne généralement une valeur par défaut à mes inputs quand c'est possible, ou j'ajoute un `!` (une assertion non-nulle) après le champ pour indiquer à TypeScript que le champ sera initialisé plus tard (par exemple, quand c'est un input obligatoire). Si l'input n'est pas obligatoire, et si nous n'avons pas de valeur par défaut, alors il faut marquer le champ comme étant optionnel ou nullable.

```

@Component({
  selector: 'ns-pony',
  template: `<div>{{ pony.name }}</div>`,
  standalone: true
})
export class PonyComponent {
  @Input({ required: true }) pony!: PonyModel;
}

@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>
  `,
  standalone: true,
}

```

```

    imports: [NgFor, PonyComponent]
)
export class PoniesComponent {
  ponies: Array<PonyModel> = [
    { id: 1, name: 'Rainbow Dash' },
    { id: 2, name: 'Pinkie Pie' }
  ];
}

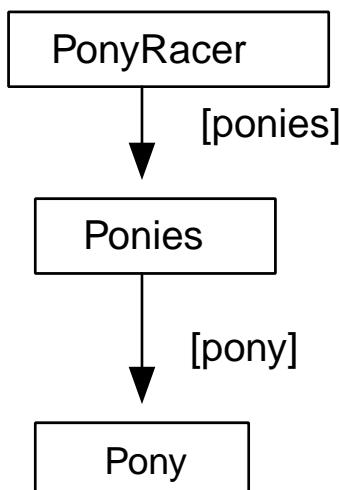
```

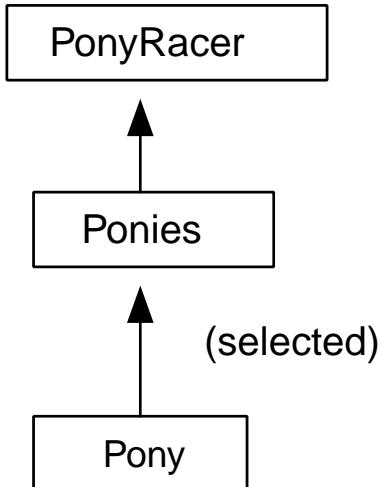
OK, et maintenant, comment passe-t-on des données vers le haut ? On ne peut pas utiliser des propriétés pour passer des données de `PonyComponent` vers `PoniesComponent`. Mais on peut utiliser des événements !

10.2.3. 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` (le guide de style Angular recommande de le marquer `readonly`) ;
- 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>',
  standalone: true
})
export class SelectablePonyComponent {
  @Input({ required: true }) pony!: PonyModel;

  // we declare the custom event as an output,
  // the EventEmitter is used to emit the event
  @Output() readonly ponySelected = new EventEmitter<PonyModel>();

  /**
   * Selects a pony when the component is clicked.
   * Emits a custom event.
   */
  selectPony(): void {
    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: PonyModel): void {  
  // 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>',  
  standalone: true  
)  
export class OtherSelectablePonyComponent {  
  @Input({ required: true }) pony!: PonyModel;  
  // the emitter is called 'emitter'  
  // and the event 'ponySelected'  
  @Output('ponySelected') readonly emitter = new EventEmiter<PonyModel>();  
  
  selectPony(): void {  
    this.emitter.emit(this.pony);  
  }  
}
```

10.2.4. 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]',
  standalone: true
})
export class UndefinedInputsDirective {
  @Input({ required: true }) 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 contenus projetés du composant ont été vérifiés pour la première fois.
- `ngAfterContentChecked` est appelée quand tous les contenus projetés du composant ont été vérifiés, même s'ils n'ont pas changé.
- `ngAfterViewInit` est appelée quand tous les bindings du template ont été vérifiés pour la première fois.
- `ngAfterViewChecked` est appelée quand tous les bindings du template ont été vérifiés, même s'ils

n'ont pas changé. Cela peut être utile si ton composant ou ta directive attend qu'un élément particulier soit disponible pour en faire quelque chose, par exemple mettre le focus dessus.

Si cela semble un peu difficile à comprendre, pas d'inquiétude : continue la lecture, et on reviendra sur le sujet plus en détails dans le [chapitre Composants et directives avancés](#) un peu plus loin.

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]',
  standalone: true
})
export class OnInitDirective implements OnInit {
  @Input({ required: true }) pony!: string;

  ngOnInit(): void {
    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]',
  standalone: true
})
export class OnChangesDirective implements OnChanges {
  @Input({ required: true }) pony!: string;

  ngOnChanges(changes: SimpleChanges): void {
    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]',
  standalone: true
})
export class SetterDirective {
  private ponyModel!: string;

  @Input()
  set pony(newPony: string) {
    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]',
  standalone: true
})
export class OnDestroyDirective implements OnDestroy {
  sayHello: number;

  constructor() {
    this.sayHello = window.setInterval(() => console.log('hello'), 1000);
  }

  ngOnDestroy(): void {
    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)...

10.3. 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.

10.3.1. 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.

10.3.2. 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; }'],
  standalone: true
})
export class StyledPonyComponent {
  @Input() pony: any;
}
```

10.3.3. 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 !



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.

Chapter 11. 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 v0* (la première version de la spécification, maintenant dépréciée) ;
- `ViewEncapsulation.ShadowDom`, qui s'appuie sur le *Shadow DOM v1* (nouvelle option introduite dans Angular 6.1, qui supporte la nouvelle version de la specification 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>',
  standalone: true,
  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.

11.1. Stratégie *Shadow DOM*

Si tu utilises la stratégie `ShadowDom`, 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 `ShadowDom`, 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. Alors sois prudent quand tu l'utilises dans tes applications !

11.2. 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 `ShadowDom`, 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 `ShadowDom`). 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.

11.3. 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.

11.4. 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 `ShadowDom`, 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 `ShadowDom` 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.

Chapter 12. Pipes

12.1. 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 formater, 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") :).

Comme les composants, les *pipes* peuvent être *standalone* ou pas. Les *pipes* fournis par Angular et dont nous allons discuter ici sont tous *standalone*, et font tous partie de `CommonModule`. Ainsi, pour pouvoir les utiliser dans un composant, il faudra les ajouter aux `imports` de ce composant, ou bien y ajouter le `CommonModule`.

Étudions quelques cas d'exemple.

12.2. 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 `|` 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>
```

12.3. 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';
import { NgFor, SlicePipe } from '@angular/common';
```

```

@Component({
  selector: 'ns-ponies',
  template: '<div *ngFor="let pony of ponies | slice: 0 : 2">{{ pony.name }}</div>',
  standalone: true,
  imports: [NgFor, SlicePipe]
})
export class PoniesComponent {
  ponies: Array<PonyModel> = [{ 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';
import { NgFor, SlicePipe } from '@angular/common';

@Component({
  selector: 'ns-ponies',
  template: '<div *ngFor="let pony of ponies | slice: 0 : size">{{ pony.name }}</div>',
  standalone: true,
  imports: [NgFor, SlicePipe]
})
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';
import { NgFor, SlicePipe } from '@angular/common';

@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>
  `,
  standalone: true,
  imports: [NgFor, SlicePipe]
})

```

```

})
export class PoniesComponent {
  ponies: Array<PonyModel> = [{ name: 'Rainbow Dash' }, { name: 'Pinkie Pie' }, {
  name: 'Fluttershy' }];
}

```

12.4. keyvalue ("clé valeur")

Ce *pipe*, introduit par Angular 6.1, permet d'itérer sur une *Map* ou un objet, et ainsi d'afficher les couples clé/valeur dans nos templates.

A noter qu'il ordonne par défaut les clés :

- d'abord par ordre lexicographique si ce sont toutes deux des chaînes de caractères
- puis par valeur si ce sont toutes deux des nombres
- puis par valeur booléenne si ce sont toutes deux des booléens (*false* avant *true*)

Et si les clés sont de type différent, elles seront converties en chaînes de caractères puis comparées.

```

@Component({
  selector: 'ns-ponies',
  template: `
    <ul>
      <!-- entry contains { key: number, value: PonyModel } -->
      <li *ngFor="let entry of ponies | keyvalue">{{ entry.key }} - {{ entry.value.name }}</li>
    </ul>
  `,
  standalone: true,
  imports: [NgFor, KeyValuePipe]
})
export class PoniesComponent {
  ponies = new Map<number, PonyModel>();

  constructor() {
    this.ponies.set(103, { name: 'Rainbow Dash' });
    this.ponies.set(56, { name: 'Pinkie Pie' });
  }
}

```

Si vous avez des clés *null* ou *undefined*, elles seront affichées à la fin.

Il est aussi possible de définir votre propre comparateur :

```

@Component({
  selector: 'ns-ponies',
  template: `

```

```

<ul>
  <!-- entry contains { key: PonyModel, value: number } -->
  <li *ngFor="let entry of poniesWithScore | keyvalue: ponyComparator">{{ entry.key.name }} - {{ entry.value }}</li>
</ul>
,
standalone: true,
imports: [NgFor, KeyValuePipe]
)
export class PoniesComponent {
  poniesWithScore = new Map<PonyModel, number>();

  constructor() {
    this.poniesWithScore.set({ name: 'Rainbow Dash' }, 430);
    this.poniesWithScore.set({ name: 'Rainbow Dash' }, 680);
    this.poniesWithScore.set({ name: 'Pinkie Pie' }, 125);
  }

  /*
   * Defines a custom comparator to order the elements by the name of the PonyModel
   (the key)
  */
  ponyComparator(a: KeyValue<PonyModel, number>, b: KeyValue<PonyModel, number>): -1 | 0 | 1 {
    if (a.key.name === b.key.name) {
      return 0;
    }
    return a.key.name < b.key.name ? -1 : 1;
  }
}

```

12.5. 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' -->
```

12.6. 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' -->
```

12.7. 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' -->
```

12.8. number ("nombre")

Les *pipes* suivants (`number`, `percent`, `currency`, `date`) sont des aides à l'internationalisation. Ils ont été complètement revus avec la version 5.0 d'Angular, et n'utilisent plus l'API Intl des navigateurs (source de nombreux problèmes). L'équipe Angular a donc implémenté elle-même la logique d'internationalisation. Les exemples suivants utilisent la nouvelle implémentation des *pipes* introduite avec Angular 5, sans s'attarder sur l'aspect internationalisation, qui est couvert dans un [chapitre](#) dédié en fin d'ouvrage. Les exemples utilisent ici la locale par défaut d'Angular, à savoir `en-US`.



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' -->
```

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:'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' }}</p>
<!-- will display '12,345.1' -->

<p>{{ 12345.16 | number:'1-1' }}</p>
<!-- will display '12,345.2' -->
```

12.9. percent ("pourcent")

Sur le même principe, `percent` permet d'afficher... un pourcentage !

```
<p>{{ 0.8 | percent }}</p>
<!-- will display '80%' -->

<p>{{ 0.8 | percent:'3' }}</p>
<!-- will display '80.000%' -->
```

12.10. 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, une option indiquant si l'on souhaite afficher le symbole ('€', '\$', 'CA\$') avec '`symbol`' ou le code ISO avec '`code`', voire le symbole raccourci avec '`symbol-narrow`'. Le symbole raccourci est par exemple `$`, alors que le symbole est `CA$` pour les dollars canadiens. Par défaut, l'option est '`symbol`'.
- Optionnellement, une chaîne de formatage du montant, avec la même syntaxe que `number`.

```
<p>{{ 10.6 | currency:'CAD' }}</p>
<!-- will display 'CA$10.60' -->

<p>{{ 10.6 | currency:'CAD':'symbol-narrow' }}</p>
<!-- will display '$10.60' -->
```

```
<p>{{ 10.6 | currency:'EUR':'code':'.3' }}</p>
<!-- will display 'EUR10.600' -->
```

Si tu ne fournis pas le code ISO de la devise, alors **USD** est utilisé, à moins que tu ne le configures globalement (depuis Angular 9). Lis le chapitre sur [l'internationalisation](#) si tu veux savoir comment.

Ce pipe est plus puissant qu'il n'en a l'air : il s'appuie sur la spécification ISO 4217 pour déterminer le nombre de chiffres après la virgule. Par exemple, formater un montant en pesos chiliens donne un résultat sans décimales (car les pesos n'ont pas de centimes), alors que formater un montant en dinars tunisiens donne un résultat avec 3 décimales (car les dinars ont des millimes).

12.11. 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' -->
```

Évidemment, 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' -->
```



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](#).

12.12. async

Le **pipe async** permet d'afficher des données obtenues de manière asynchrone. Il peut gérer des données qui 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 ES2015. Et concernant l'Observable, nous y viendrons rapidement.

Un **pipe async** retourne **null** 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';
import { AsyncPipe } from '@angular/common';

@Component({
  selector: 'ns-greeting',
  template: '<div>{{ asyncGreeting | async }}</div>',
  standalone: true,
  imports: [AsyncPipe]
})
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 *pipe* (quand l'utilisateur naviguera vers une autre page, par exemple, ou s'il fait partie du `ngIf` d'un template qui devient faux).

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` :

```
import { Component } from '@angular/core';
import { AsyncPipe, NgIf } from '@angular/common';

@Component({
  selector: 'ns-user',
  template: '<div *ngIf="asyncUser | async as user">{{ user.name }}</div>',
  standalone: true,
  imports: [NgIf, AsyncPipe]
})
export class UserComponent {
  asyncUser = new Promise<UserModel>(resolve => {
    // after 1 second, the promise will resolve
    window.setTimeout(() => resolve({ name: 'Cédric' }), 1000);
  });
}
```

Si tu veux en savoir plus sur ce *pipe*, regarde le chapitre sur les [Performances](#) vers la fin du livre.

12.13. Un *pipe* dans ton code

Il est aussi possible d'utiliser les fonctions de formatage offertes par le framework directement. Par exemple pour formater un nombre, on peut utiliser la fonction `formatNumber` :

```
import { Component, Inject, LOCALE_ID } from '@angular/core';
// you need to import the function you want to use
import { formatNumber } from '@angular/common';

@Component({
  selector: 'ns-pony',
  template: '<p>{{ formattedSpeed }}</p>',
  standalone: true
})
export class PonyComponent {
  pony = { name: 'Rainbow Dash', speed: 15 };
  formattedSpeed: string;

  constructor(@Inject(LOCALE_ID) locale: string) {
    // use the format function
    this.formattedSpeed = formatNumber(this.pony.speed, locale, '.2');
  }
}
```

12.14. Crée 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 { Pipe, PipeTransform } from '@angular/core';

export class FromNowPipe implements PipeTransform {
  transform(value: string, args: Array<unknown>): string {
    // do something here
  }
}
```

Nous utiliserons la fonction `parseISO` de `date-fns` pour interpréter la date, et la fonction `formatDistanceToNowStrict` pour afficher combien de temps s'est écoulé depuis.

Tu peux installer `date-fns` avec `NPM` si tu le souhaites :

```
npm install date-fns
```

Les types nécessaires pour TypeScript sont déjà inclus dans la dépendance NPM, le compilateur devrait donc être content sans action particulière de notre part.

```
import { Pipe, PipeTransform } from '@angular/core';
import { formatDistanceToNowStrict, parseISO } from 'date-fns';

export class FromNowPipe implements PipeTransform {
  transform(value: string, args: Array<unknown>): string {
    const date = parseISO(value);
    return formatDistanceToNowStrict(date, { addSuffix: true });
  }
}
```

Maintenant, nous devons dire à Angular que cette classe constitue un *pipe*. Pour cela, il y a un décorateur particulier à utiliser: `@Pipe`.

```
import { Pipe, PipeTransform } from '@angular/core';
import { formatDistanceToNowStrict, parseISO } from 'date-fns';

@Pipe({
  name: 'fromNow',
  standalone: true
})
export class FromNowPipe implements PipeTransform {
  transform(value: string, args: Array<unknown>): string {
    const date = parseISO(value);
    return formatDistanceToNowStrict(date, { addSuffix: true });
}
```

Le nom choisi sera celui qui nous permettra d'utiliser le *pipe* dans le template. Et comme nous voulons que le *pipe* soit *standalone*, nous ajoutons la propriété `standalone: true`.

Pour pouvoir utiliser le *pipe* dans le template, la dernière chose à faire est d'ajouter le *pipe* dans les `declarations` du composant.

```
@Component({
  selector: 'ns-race',
  template: 'The race started {{ race.startInstant | fromNow }}',
  standalone: true,
  imports: [FromNowPipe]
})
export class RaceComponent {
```

```
race = {  
    startInstant: '2023-02-10T10:00:00.000Z'  
};  
}
```

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 échappé.



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 date-fns 🦄](#) te fait construire ton premier *pipe* custom !

Chapter 13. Injection de dépendances

13.1. 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.

13.2. 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 `LoggingService`. Pendant le développement, on veut utiliser

une implémentation qui affiche les messages dans la console du navigateur. Mais en production, on veut utiliser une implémentation qui envoie les messages vers un serveur distant.

```
export class LoggingService {
  log(message: string): void {
    console.log(message);
  }
}
```

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 `LoggingService` :

```
import { LoggingService } from './logging.service';

export class RaceService {
  constructor(private loggingService: LoggingService) {}

}
```

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

Maintenant, on peut ajouter une méthode `list()` à notre service, qui appellera notre serveur et tracera un message en utilisant notre `LoggingService` :

```
import { LoggingService } from './logging.service';

export class RaceService {
  constructor(private loggingService: LoggingService) {}

  list(): Array<RaceModel> {
    this.loggingService.log('race-service: get races');
    // ...
  }
}
```

En Angular, tous les services doivent être décorés avec `@Injectable()` :

```
import { Injectable } from '@angular/core';
import { RaceModel } from './race.model';
import { LoggingService } from './logging.service';

@Injectable()
```

```

export class RaceService {
  constructor(private loggingService: LoggingService) {}

  list(): Array<RaceModel> {
    this.loggingService.log('race-service: get races');
    // ...
  }
}

```

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

La façon la plus simple (et recommandée) est d'ajouter le décorateur `@Injectable` sur `LoggingService` et d'ajouter `providedIn` directement à l'intérieur :

```

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

@Injectable({
  providedIn: 'root'
})
export class LoggingService {
  log(message: string): void {
    console.log(message);
  }
}

```

Une autre façon est d'utiliser l'option `providers` de la fonction `bootstrapApplication()` vue précédemment :

```
bootstrapApplication(AppComponent, { providers: [LoggingService] });
```

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

```

import { Injectable } from '@angular/core';
import { LoggingService } from './logging.service';
import { RaceModel } from './race.model';

@Injectable({
  providedIn: 'root'
})
export class RaceService {
  constructor(private loggingService: LoggingService) {}

  list(): Array<RaceModel> {
    this.loggingService.log('race-service: get races');
  }
}

```

```
// ...  
}  
}
```

Et c'est fini !

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

```
export class AppComponent {  
  races: Array<RaceModel> = [];  
  
  // add a constructor with RaceService  
  constructor(private raceService: RaceService) {  
    this.races = this.raceService.list();  
  }  
}
```

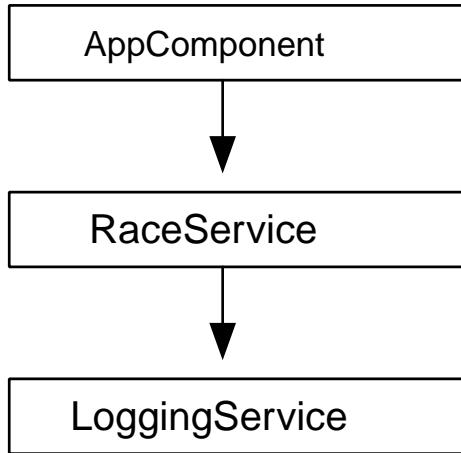
Comme nous voulons utiliser une API pour collecter les logs en production, nous allons créer une implémentation différente de `LoggingService` qui envoie les messages vers un serveur distant.

13.3. 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.

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 :

```
bootstrapApplication(AppComponent, {
  providers: [{ provide: LoggingService, useClass: LoggingService }]
});
```

On explique à l'injecteur qu'on veut un lien entre un token (le type `LoggingService`) et la classe `LoggingService`. 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 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, le token et la classe de l'objet à injecter sont les mêmes, tu peux écrire la même chose en version raccourcie :

```
bootstrapApplication(AppComponent, { providers: [LoggingService] });
```

Le token doit identifier de manière unique la dépendance.

Angular propose aussi une fonction `inject()` qui permet d'injecter une dépendance. On verra des cas d'utilisation plus tard, mais en attendant, c'est un bon moyen de tester comment fonctionne l'injecteur.

```
// in our bootstrapApplication function
providers: [
  LoggingService,
  { provide: LoggingService, useClass: LoggingService },
  // let's add another provider to the same class
  // with another token
  // A token can be declared like this:
```

```
// const token = new InjectionToken<LoggingService>('LoggingServiceToken');
{ provide: token, useClass: LoggingService }
]
```

```
console.log(inject(LoggingService));
// logs "LoggingService"
console.log(inject(token));
// logs "LoggingService" again
console.log(inject(LoggingService) === inject(LoggingService));
// logs "true", as the same instance is returned every time for a token
console.log(inject(LoggingService) === inject(token));
// logs "false", as the providers are different,
// so there are two distinct instances
```

On demande ici une dépendance à l'injecteur grâce à un token. Comme j'ai déclaré le `LoggingService` deux fois, avec deux tokens différents, on a deux providers. L'injecteur va créer une instance de `LoggingService` 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 `LoggingService` 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 `LoggingService`. Je peux écrire une nouvelle classe, faisant le même boulot que `LoggingService`, mais qui cette fois appelle une API distante pour collecter les messages :

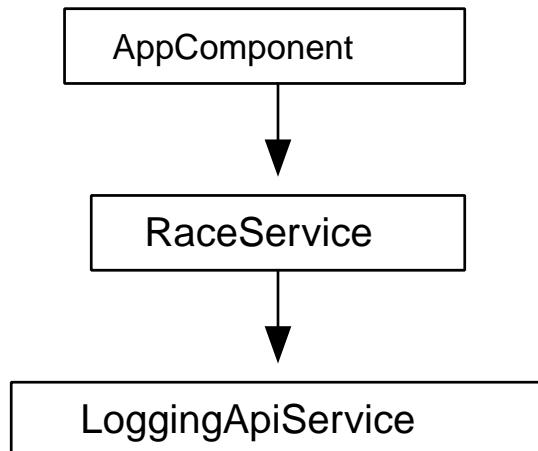
```
@Injectable()
export class LoggingAPIService {
  log(message: string): void {
    // ... calls the logging API
  }
}
```

```
}
```

On peut manipuler la déclaration du provider pour remplacer `LoggingService` par notre `LoggingAPIService` :

```
// in our bootstrapApplication function
providers: [
  // we provide a different implementation of the service
  { provide: LoggingService, useClass: LoggingAPIService }
]
```

Maintenant nos relations de dépendances sont devenues :



Cela peut être très utile pour remplacer une implémentation par une autre dans certains environnements, et aussi, comme nous le verrons bientôt, quand tu écris des tests automatisés.

13.4. Autres types de provider

Toujours dans le cadre de cet exemple, nous aurions probablement envie d'utiliser `LoggingService` pendant le développement, et `LoggingAPIService` en production. Tu peux pour cela utiliser un autre type de provider : `useFactory`.

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

// in our bootstrapApplication function
providers: [
  RaceService,
  // we provide a factory
  {
    provide: LoggingService,
```

```
    useFactory: () => (IS_PROD ? new LoggingAPIService() : new LoggingService())
  }
]
```

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

```
// in our bootstrapApplication function
providers: [RaceService, { provide: LoggingService, useClass: IS_PROD ?
  LoggingAPIService : LoggingService }]
```

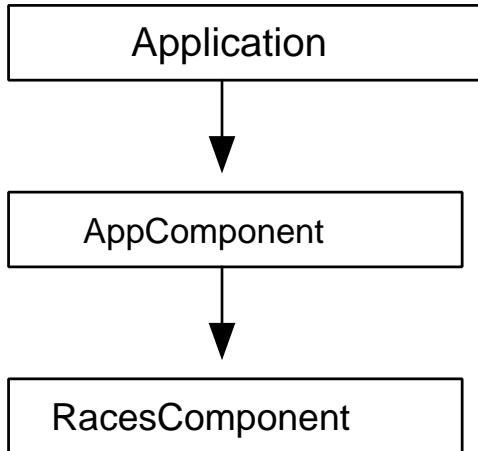
Déclarer une constante `IS_PROD` est un peu pénible : et si nous utilisions aussi ici l'injection de dépendances ?! Oui, je pousse le bouchon un peu loin ! :) Tu n'as certes pas besoin de tout faire rentrer dans le moule de l'injection, mais cela permet de te montrer un autre type de provider : `useValue`.

```
// in our bootstrapApplication function
providers: [
  { provide: 'IS_PROD', useValue: true },
  RaceService,
  {
    provide: LoggingService,
    useFactory: (IS_PROD: boolean) => (IS_PROD ? new LoggingAPIService() : new
  LoggingService()),
    deps: ['IS_PROD']
  }
]
```

13.5. 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.

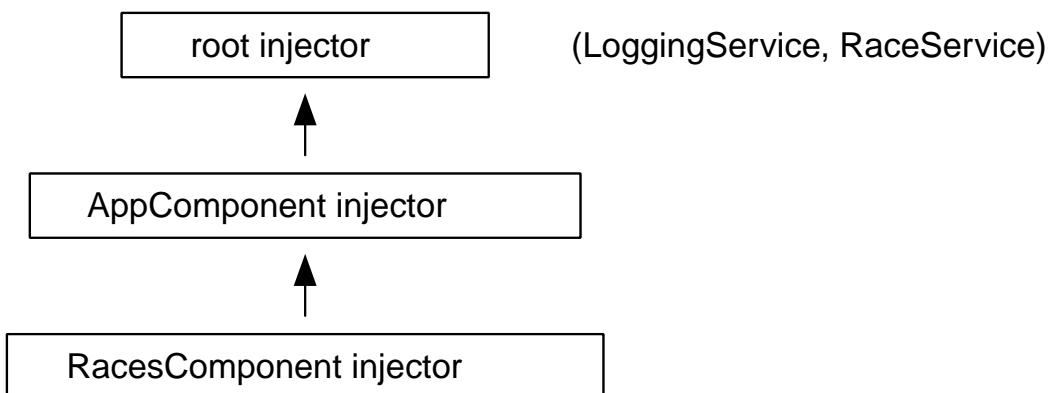
Disons qu'on a une application qui ressemble à :



On a un module racine `AppModule` avec un composant racine `AppComponent`, 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.

Quand tu enregistres un service en utilisant la façon recommandée `providedIn: 'root'`, ou en utilisant les providers du module, tu ajoutes ces services à l'injecteur racine.



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, `LoggingService` 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'option `providers` de `bootstrapApplication()`.

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

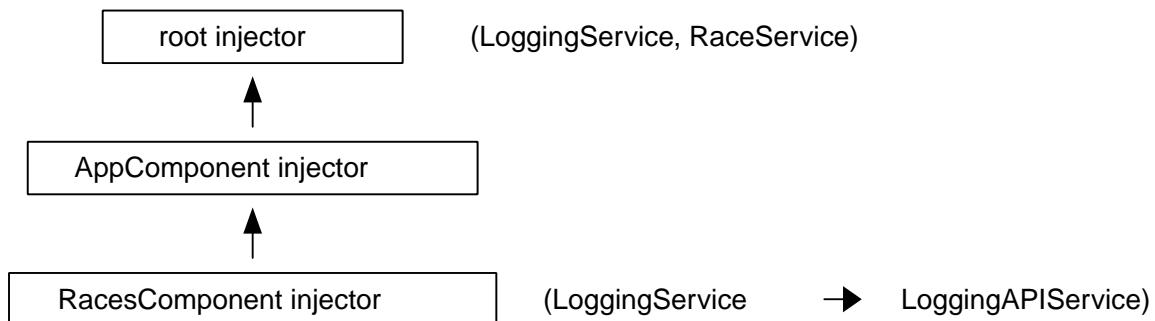
```
@Component({
  selector: 'ns-races',
  providers: [{ provide: LoggingService, useClass: LoggingAPIService }],
  template: '<strong>Races</strong>',
  standalone: true
})
export class RacesComponent {
  constructor(private loggingService: LoggingService) {
    this.loggingService.log('RacesComponent created');
  }
}
```

Dans ce composant, le provider avec le token `LoggingService` retournera toujours une instance de `LoggingAPIService`, quel que 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.

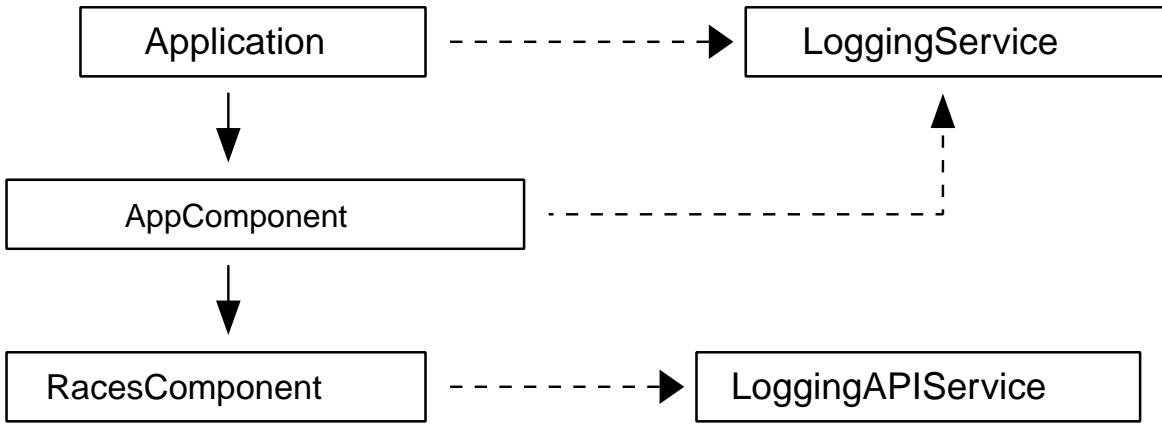


Si tu déclares une dépendance à la fois dans le module de ton application et dans l'attribut `providers` de ton composant, ce seront bien deux instances différentes qui seront créées et utilisées !

Ici, nous avons :



L'injection sera alors résolue en :



Si le service contient un état propre à une instance de composant, alors il devrait être fourni par ce composant. Pour les services sans état, ou qui contiennent un état global, ils devraient être fournis à la racine.

13.6. Injection sans types

Il est aussi possible de ne pas utiliser un type pour l'injection de dépendances, en passant par le décorateur `@Inject()`. On peut ainsi injecter des services ou de simples valeurs comme dans l'exemple suivant :

```

import { Inject, Injectable } from '@angular/core';

import { BACKEND_URL } from './tokens';

@Injectable({
  providedIn: 'root'
})
export class RaceService {
  constructor(@Inject(BACKEND_URL) private url: string) {}

}

```

Pour utiliser `@Inject()`, il est nécessaire de lui passer un *token*. Ce token est défini comme ceci par exemple :

```

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

export const BACKEND_URL = new InjectionToken<string>('API URL');

```

On utilise ensuite ce token dans les providers du module pour définir sa valeur, par exemple :

```
{ provide: BACKEND_URL, useValue: 'http://localhost:8080' }
```

Ou tu peux l'enregistrer directement avec `providedIn` :

```
export const BACKEND_URL_PROVIDED = new InjectionToken<string>('API URL', {
  providedIn: 'root',
  factory: () => 'http://localhost:8080'
});
```

Note que tu pourrais utiliser la fonctionnalité des variables d'environnement de Angular CLI pour ce cas.

Angular utilise ce mécanisme de token également, et nous permet par exemple de définir la locale de l'application grâce au token `LOCALE_ID` :

```
bootstrapApplication(AppComponent, {
  providers: [
    { provide: LOCALE_ID, useValue: 'fr-FR' }
  ]
});

@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' -->
  `,
  standalone: true,
  imports: [DecimalPipe]
})
export class CustomLocaleComponent {
  constructor(@Inject(LOCALE_ID) public locale: string) {}
}
```

Mais on en reparlera plus loin lorsque l'on abordera l'internationalisation !

13.7. inject()

Angular fournit également une fonction `inject()` qui permet d'injecter un service. Cette fonction est disponible depuis la version 14 du framework et est utile dans les cas où il n'est pas possible d'utiliser l'injection par constructeur. Elle a été introduite pour permettre l'utilisation de l'injection de dépendances dans des fonctions, car Angular tend de plus en plus à utiliser celles-ci au lieu des classes pour certaines fonctionnalités. C'est notamment le cas des gardes du routeur ou des

intercepteurs HTTP que l'on croisera plus loin : historiquement, ceux-ci étaient des classes, mais depuis les versions 14 et 15, il est possible de les définir comme des fonctions. Ce pattern est donc de plus en plus utilisé dans les versions récentes d'Angular.

```
export const loggedInGuard: CanActivateFn = (
  _route: ActivatedRouteSnapshot,
  _state: RouterStateSnapshot
): boolean | UrlTree => {
  const userService = inject(UserService);
  const router = inject(Router);
  // returns 'true' if the user is logged in or redirects to the login page
  // note that you can also use `router.createUrlTree()` to build a `UrlTree` with
  parameters
  return userService.isLoggedIn() || router.parseUrl('/login');
};
```

13.8. Services provided by the framework

Angular fournit quelques services. Certains d'entre eux seront décrits dans des chapitres dédiés. Pour l'instant, parlons seulement de deux services très simples.

13.8.1. Service Title

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';

@Component({
  selector: 'ns-root',
  template: '<h1>PonyRacer</h1>',
  standalone: true
})
export class AppComponent {
  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 !

13.8.2. 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';

@Component({
  selector: 'ns-root',
  template: '<h1>PonyRacer</h1>',
  standalone: true
})
export class AppComponent {
  constructor(meta: Meta) {
    meta.addTag({ name: 'author', content: 'Ninja Squad' });
  }
}
```

Chapter 14. Programmation réactive

14.1. 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.

14.2. 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 a 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.

14.3. 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 :

```
import { filter, from, map } from 'rxjs';
```

```
from([1, 2, 3, 4, 5])
```

```

.pipe(
  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 :

```

import { fromEvent } from 'rxjs';

const input = $('#input');

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 = new Observable(observer => observer.next('hello'));

observable.subscribe(value => console.log(value));
// logs "hello"

```

`new Observable()` 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, au lieu d'une simple fonction de callback, un objet. Cet objet peut définir un callback pour gérer les événements (appelé `next`), et un callback pour gérer les erreurs (appelé `error`).

Ici, la méthode `map` lève une exception, donc le callback d'erreur va la tracer.

```

range(1, 5)
.pipe(
  map(x => {
    if (x % 2 === 1) {
      throw new Error('something went wrong');
    } else {
      return x;
    }
  })
)

```

```

    }
  )),
  filter(x => x > 5)
)
.subscribe({
  next: x => console.log(x),
  error: 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 :

```

range(1, 5)
  .pipe(
    map(x => x * 2),
    filter(x => x > 5)
)
.subscribe({
  next: x => console.log(x),
  error: error => console.log(error),
  complete: () => console.log('done')
});
// 6, 8, 10, done

```

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 avoir une chouette représentation visuelle de chaque fonction, va sur rxmarbles.com.

Maintenant, regardons comment les *observables* seront utilisés dans Angular.

14.4. 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`.

Tu le reconnais ? C'est ce qu'on a utilisé pour émettre les événements sur un `Output` !

```

@Component({
  selector: 'ns-pony',
  // the method `selectPony()` will be called on click
  template: '<div (click)="selectPony()">{{ pony.name }}</div>',
  standalone: true
})
export class SelectablePonyComponent {
  @Input({ required: true }) pony!: PonyModel;

  // we declare the custom event as an output,
  // the EventEmitter is used to emit the event
  @Output() readonly ponySelected = new EventEmitter<PonyModel>();

  /**
   * Selects a pony when the component is clicked.
   * Emits a custom event.
   */
  selectPony(): void {
    this.ponySelected.emit(this.pony);
  }
}

```

Tu vas aussi beaucoup utiliser les observables : dès que quelque chose est asynchrone dans une application Angular, on le représente généralement avec un observable.



Essaye notre [quiz](#) 🐴 et 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 !

Chapter 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. Écrire 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. Tests unitaires

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: number): boolean {  
    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.



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.

Par exemple, si l'on a une classe `Race` avec une méthode `start()`, qui appelle `run()` sur chaque poney dans la course, et filtre ceux qui n'ont pas commencé à courir (`run()` renvoie un booléen) :

Race.ts

```
class Race {
  constructor(private ponies: Array<Pony>) {}

  start(): Array<Pony> {
    return (
      this.ponies
        // start every pony
        // and only keeps the ones that started running
        .filter(pony => pony.run(10))
    );
  }
}
```

Nous voulons tester la méthode `start()`, et vérifier si elle appelle bien `run()`. On peut donc espionner la méthode `run()` de chacun des poneys dans la course :

```

describe('Race', () => {
  let rainbowDash: Pony;
  let pinkiePie: Pony;
  let race: Race;

  beforeEach(() => {
    rainbowDash = new Pony('Rainbow Dash');
    // first pony agrees to run
    spyOn(rainbowDash, 'run').and.returnValue(true);

    pinkiePie = new Pony('Pinkie Pie');
    // second pony refuses to run
    spyOn(pinkiePie, 'run').and.returnValue(false);

    // create a race with these two ponies
    race = new Race([rainbowDash, pinkiePie]);
  });

});

```

et vérifier si les méthodes sont bien appelées :

```

it('should make the ponies run when it starts', () => {
  // start the race
  const runningPonies: Array<Pony> = race.start();
  // should have called `run()` on the ponies
  expect(pinkiePie.run).toHaveBeenCalled();
  // with a speed of 10
  expect(rainbowDash.run).toHaveBeenCalledWith(10);
  // as one pony refused to start, the result should be an array of one pony
  expect(runningPonies).toEqual([rainbowDash]);
});

```

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 renvoyant une liste de courses codée en dur.

```
@Injectable({
  providedIn: 'root'
})
export class RaceService {
  list(): Array<RaceModel> {
    const race1: RaceModel = { name: 'London' };
    const race2: RaceModel = { name: '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);
  });
});
```

Cela 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 `inject`.

Cette méthode est utilisée dans notre test pour récupérer des dépendances spécifiques.

Si on retourne à notre exemple, cette fois-ci en utilisant `TestBed.inject` :

```
import { TestBed } from '@angular/core/testing';

describe('RaceService', () => {
  it('should return races when list() is called', () => {
    const raceService = TestBed.inject(RaceService);
    expect(raceService.list().length).toBe(2);
  });
});
```

Cela va fonctionner, car le service est déclaré avec `providedIn: 'root'`, ce qui le rend disponible dans le test. Il sera alors instancié et injecté paresseusement quand il est demandé.

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.inject` dans un `beforeEach` :

```
import { TestBed } from '@angular/core/testing';

describe('RaceService', () => {
  let service: RaceService;

  beforeEach(() => (service = TestBed.inject(RaceService)));

  it('should return races when list() is called', () => {
    expect(service.list().length).toBe(2);
  });
});
```

On a déplacé la logique de `TestBed.inject` dans un `beforeEach`, et maintenant notre test est plutôt clair.

Évidemment, un véritable `RaceService` n'aura pas une liste de courses écrite en dur, et il est fortement probable que cette réponse soit asynchrone. Disons que la méthode `list` retourne un observable. Qu'est-ce que cela change dans notre test ? Et bien nous devrons mettre `expect` dans le callback `subscribe` :

```

import { TestBed, waitForAsync } from '@angular/core/testing';

describe('RaceService', () => {
  let service: RaceService;

  beforeEach(() => (service = TestBed.inject(RaceService)));

  it('should return an observable of 2 races', waitForAsync(() => {
    service.list().subscribe(races => {
      expect(races.length).toBe(2);
    });
  }));
});

```

Tu te dis peut-être que cela ne fonctionnera pas, car le test se terminera avant que l'observable ne soit appelé, et notre assertion ne sera jamais exécutée.

Mais nous avons utilisé la fonction `waitForAsync()` 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 `waitForAsync()` : 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

La classe `TestBed` va nous aider à déclarer des dépendances factices si besoin.

Même si Angular nous permet maintenant de créer des applications sans définir aucun module Angular, le design de son `TestBed` est toujours basé sur la notion de module de test (*testing module*). Le module de test peut être configuré à peu de choses près de la même manière qu'un module Angular. Comme nous utilisons des composants *standalone*, le but principal du module de test sera de fournir des dépendances factices au lieu des dépendances réelles. La méthode `TestBed.configureTestingModule()` permet de spécifier pour cela un tableau de `providers`, qui prennent le dessus sur les providers réels utilisés pour injecter les dépendances dans le service, composant, pipe ou directive qu'on veut tester.

Illustrons cela par un 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({
  providedIn: 'root'
})
export class RaceService {
  constructor(private localStorage: LocalStorageService) {}

  list(): Array<RaceModel> {
    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 :

```
export class FakeLocalStorage {
  get(key: string): any {
    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 }]
    });
  );

  it('should return 2 races from localStorage', () => {
    const service = TestBed.inject(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>('LocalStorageService', ['get']);

  beforeEach(() =>
    TestBed.configureTestingModule({
      providers: [{ provide: LocalStorageService, useValue: localStorage }]
    })
  );

  it('should return 2 races from localStorage', () => {
    localStorage.get.and.returnValue([ { name: 'Lyon' }, { name: 'London' } ]);

    const service = TestBed.inject(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 tester non seulement la partie TypeScript du composant, mais aussi sa partie HTML.

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() + (running ? '-running' : '')
    + '.png'
      [alt]="pony.name"
      (click)="clickOnPony()"
    />
  `,
  standalone: true,
  imports: [CommonModule]
})
export class PonyComponent {
  @Input({ required: true }) pony!: PonyModel;
  @Input() running = false;
}

```

```

@Output() readonly ponyClicked = new EventEmitter<PonyModel>();

clickOnPony(): void {
  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.

```

import { TestBed } from '@angular/core/testing';

import { PonyComponent } from './pony.component';

describe('PonyComponent', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({});
  });

  it('should have an image', () => {
    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;
    const imageElement = element.querySelector('img');
    expect(imageElement.getAttribute('src')).toContain('/images/pony-blue.png');
    expect(imageElement.getAttribute('alt')).toBe('Rainbow Dash');
  });
});

```

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', () => {
  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>
  `,
  standalone: true,
  imports: [CommonModule, PonyComponent]
})
export class RaceComponent {
  @Input({ required: true }) race!: RaceModel;
}
```

et son test :

```
describe('RaceComponent', () => {
  let fixture: ComponentFixture<RaceComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({});  

    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 requêter avec `query()` et `queryAll()`. Ces méthodes prennent un prédictat 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 hôte de test* qui l'utilise. Cela permet de vérifier que le *binding* d'inputs et d'outputs via le template fonctionne correctement. Prenons en exemple le `PonyComponent` défini plus haut. Afin de tester qu'il peut, ou pas, prendre une

input running, et que le comportement est correct dans les deux cas, il faudrait faire un premier test avec un composant parent qui ne fournit pas l'*input*, et un deuxième test avec un composant parent qui le fournit.

Heureusement, le **TestBed** permet de redéfinir le template du composant hôte (ou de n'importe quel autre composant d'ailleurs) :

```
import { Component } from '@angular/core';
import { PonyComponent, PonyModel } from './pony.component';
import { ComponentFixture, TestBed } from '@angular/core/testing';

@Component({
  selector: 'ns-test-host',
  template: '',
  standalone: true,
  imports: [PonyComponent]
})
class TestHostComponent {
  pony: PonyModel = {
    name: 'Rainbow Dash',
    color: 'BLUE'
  };
}

describe('PonyComponent', () => {
  let fixture: ComponentFixture<TestHostComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({});;
  });

  it('should display a non-running pony by default', () => {
    // given a test host component where the running input is not passed
    TestBed.overrideTemplate(TestHostComponent, '<ns-pony [pony]="pony"></ns-pony>');
    fixture = TestBed.createComponent(TestHostComponent);

    // when we trigger the change detection
    fixture.detectChanges();

    // then we should have a not running pony
    const element = fixture.nativeElement;
    expect(element.querySelector('img').src).toContain('/images/pony-blue.png');
  });

  it('should display a running pony if the running input is set to true', () => {
    // given a test host component where the running input is not passed
    TestBed.overrideTemplate(TestHostComponent, '<ns-pony [pony]="pony"
[running]="true"></ns-pony>');
    fixture = TestBed.createComponent(TestHostComponent);
  });
});
```

```

    // when we trigger the change detection
    fixture.detectChanges();

    // then we should have a not running pony
    const element = fixture.nativeElement;
    expect(element.querySelector('img').src).toContain('/images/pony-blue-running.png');
  );
};

});

```

On peut aller plus loin que ça. Il est aussi possible d'appeler `TestBed.overrideComponent()` pour ajouter, enlever ou remplacer n'importe quel propriété du décorateur d'un composant (`template`, `providers`, `imports`, etc.). Cela peut être utile, par exemple, pour tester un composant parent en lui faisant utiliser un composant enfant factice (`stub`), au lieu du composant enfant réel, afin de simplifier le test. Nous pourrions par exemple remplacer le `PonyComponent` dans les imports de `RaceComponent` par un `PonyStubComponent` qui aurait le même sélecteur, les mêmes inputs et les mêmes outputs, mais qui ne ferait rien.

Maintenant tu es prêt pour tester ton application !

15.6. Des tests unitaires plus simples et plus propres avec `ngx-speculoos`

Les tests Angular peuvent rapidement devenir assez verbeux. Comme on n'aime pas trop ça, nous avons écrit une petite bibliothèque open-source appelée `ngx-speculoos`.

Au lieu d'écrire un test comme celui-ci :

```

let fixture: ComponentFixture<UserComponent>;

beforeEach(() => {
  fixture = TestBed.createComponent(UserComponent);
  fixture.detectChanges();
});

it('should display French cities when selecting the country France', () => {
  const countrySelect = fixture.nativeElement.querySelector('#country'); // countrySelect is of type any
  expect(countrySelect.selectedIndex).toBe(0);
  countrySelect.selectedIndex = 2; // what is at index 2?
  countrySelect.dispatchEvent(new Event('change')); // why do I need to do that?
  fixture.detectChanges();

  const city = fixture.nativeElement.querySelector('#city'); // city is of type any
  expect(city).toBeTruthy();
  expect(city.options.length).toBe(3);
  expect(city.options[0].value).toBe('');
  expect(city.options[0].label).toBe('');
}

```

```

expect(city.options[1].value).toBe('PARIS');
expect(city.options[1].label).toBe('Paris');
expect(city.options[2].value).toBe('LYON');
expect(city.options[2].label).toBe('Lyon');
});

it('should hide cities when selecting the empty country option', () => {
  const countrySelect = fixture.nativeElement.querySelector('#country'); // I did that
previously. What about DRY?
  countrySelect.selectedIndex = 0;
  countrySelect.dispatchEvent(new Event('change')); // why do I need to do that?
  fixture.detectChanges(); // why do I need to do that?

  expect(fixture.nativeElement.querySelector('#city')).toBeFalsy(); // I did that
previously. What about DRY?
});

```

tu peux écrire un test plus propre et plus simple avec `ngx-speculoos` :

```

class UserComponentTester extends ComponentTester<UserComponent> {
  constructor() {
    super(UserComponent);
  }

  get country(): TestSelect {
    return this.select('#country')!; // returns a TestSelect object, not any. Similar
methods exist for inputs, buttons, etc.
  }

  get city(): TestSelect {
    return this.select('#city')!; // returns a TestSelect object, not any
  }
}

let tester: UserComponentTester;

beforeEach(() => {
  tester = new UserComponentTester();
  tester.detectChanges();
});

it('should display French cities when selecting the country France', () => {
  tester.country.selectLabel('France'); // no dispatchEvent, no detectChanges needed

  expect(tester.city.optionValues).toEqual(['', 'PARIS', 'LYON']);
  expect(tester.city.optionLabels).toEqual(['', 'Paris', 'Lyon']);
});

it('should hide cities when selecting empty country option', () => {
  tester.country.selectedIndex(0); // no repetition of the selector, no dispatchEvent,

```

```
no detectChanges needed
```

```
    expect(tester.city).toBeFalsy(); // no repetition of the selector
});
```

Tu peux même aller un cran plus loin avec les "custom matchers" pour Jasmine que nous avons écrits :

```
beforeEach(() => jasmine.addMatchers(speculoosMatchers));

it('should contain a pre-populated form', () => {
  expect(tester.informationMessage).toContainText('Please check that everything is
correct');
  expect(tester.country).toHaveSelectedValue('');
  expect(tester.name).toHaveValue('Doe');
  expect(tester.newsletter).toBeChecked();
});
```

N'hésite pas à essayer !

15.7. Tests de bout en bout

Les tests de bout en bout (*end-to-end*, or *e2e*) sont l'autre type de tests que tu peux écrire et exécuter. Ils consistent à réellement lancer ton application dans un navigateur, et à simuler un utilisateur interagissant avec l'application (en cliquant sur des boutons, remplissant des formulaires, etc.). Ils ont l'avantage de tester réellement l'application dans son ensemble plutôt que des petites unités isolées, mais :

- ils sont plus lents (plusieurs secondes par test) ;
- il est plus difficile de tester les cas à la marge.

Comme tu peux le deviner, tu n'as pas à choisir entre tests unitaires et tests e2e. Combiner les deux est l'idéal pour avoir une bonne couverture et une assurance raisonnable du bon fonctionnement de l'application.

Angular CLI proposait auparavant une intégration par défaut avec Protractor. Mais ce n'est plus le cas depuis la version 12, car Protractor est peu à peu délaissé. Il nous faut donc mettre en place une solution par nous-même. Il y a beaucoup d'options intéressantes possibles : [Cypress](#), [Playwright](#), [Puppeteer](#), etc. Et je suis en pâmoison devant Cypress 🦇.

Tu peux lancer les tests e2e avec :

```
ng e2e
```

15.7.1. Cypress



Lorsque tu exécutes les tests e2e, Cypress lance Electron, Chrome, Firefox ou Edge. Tu peux aussi les lancer en mode *headless* (c'est-à-dire sans afficher la moindre fenêtre) avec :

```
ng e2e --headless
```

ce qui est pratique pour exécuter ces tests sur une plateforme d'intégration continue.

Cypress a plein de fonctionnalités intéressantes :

- il est facile à mettre en place ;
- il permet de *mocker* des réponses HTTP ;
- il permet de tester l'application avec différentes tailles de fenêtre (pratique pour les applications *responsive*) ;
- il offre un DSL relativement simple.

Ce qui m'a conquis cependant est ce qu'il appelle le *time-travel debugging*. À chaque étape du test, Cypress fait un snapshot, qui permet donc de visualiser, directement dans le navigateur, l'aspect graphique et le contenu du DOM. En plaçant simplement sa souris sur une étape d'un test en échec, on peut voir le contenu de la page et interagir avec elle pour rapidement comprendre ce qui ne va pas.

Les tests Cypress sont écrits avec Mocha. Même si c'est une bibliothèque différente de Jasmine, tu verras qu'elle est très similaire. Ce qui importe plus, c'est l'API spécifique de Cypress nous permettant d'interagir avec l'application.

Cypress nous met à disposition un objet `cy`, qui nous permet par exemple d'appeler la méthode `visit()` pour... visiter une page. Ensuite, `get(selector)` permet de sélectionner tous les éléments qui correspondent à un sélecteur CSS. Ou encore `contains(selector, text)` permet de cibler les éléments contenant un texte donné.

Les objets rentrés par ces méthodes offrent eux-mêmes des méthodes permettant de vérifier leur état, comme `should('contain', text)` or `should('be', 'not.displayed')`. Ou encore des méthodes comme `click()` et `type()` qui permettent d'interagir avec eux.

Comme je l'évoquais précédemment, tu peux aussi isoler les tests du backend, en mockant les appels HTTP, et vérifier que les requêtes sont envoyées aux moments opportuns.

Par exemple, supposons que la page d'accueil de l'application affiche un titre.

Un test de cette page ressemblerait à ça :

```
describe('Ponyracer home page', () => {
  it('should display a headline', () => {
    cy.visit('/');
    cy.get('span').should('contain', 'ponyracer app is running!');
  });
});
```

Ces tests peuvent être assez longs à écrire, mais ils sont vraiment utiles. On peut les utiliser à d'autres fins, par exemple enregistrer des vidéos de démonstration, prendre des copies d'écran, les comparer au pixel prêt d'une exécution à l'autre avec [Percy](#), etc.

Avec les tests unitaires et les tests e2e, tu as toutes les clés en main pour construire une application robuste et maintenable !



Tous les exercices du Pack Pro viennent avec leurs tests unitaires et e2e ! 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.

Chapter 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 favorite pour faire des requêtes HTTP.



Une des possibilités est l'API `fetch`, qui est une API standard fournie par les navigateurs. Tu peux parfaitement construire ton application en utilisant `fetch` ou une autre bibliothèque. En fait, c'était ce que 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 la plupart des développeurs Angular utilisent un service fourni par Angular : `HttpClient`.

Si tu veux l'utiliser, il te faudra utiliser les classes et fonctions du package `@angular/common/http`.

Pourquoi préférer ce service 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 client 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 (`provideHttpClient`)

Le module `@angular/common/http` propose un service nommé `HttpClient` que tu peux t'injecter dans n'importe quel constructeur. Ce service n'est pas disponible par défaut dans une application Angular. Il faut configurer l'application afin de pouvoir l'utiliser, en ajoutant un *provider* au démarrage de l'application :

```
import { bootstrapApplication } from '@angular/platform-browser';
import { provideHttpClient, withInterceptors } from '@angular/common/http';

bootstrapApplication(AppComponent, {
  providers: [provideHttpClient()]
}).catch(err => console.error(err));
```

Quand cela est fait, tu peux injecter le service `HttpClient` partout où tu en as besoin :

```
@Component({
  selector: 'ns-races',
  template: '<h1>Races</h1>',
  standalone: true
})
export class RacesComponent {
  constructor(private http: HttpClient) {}
}
```

Dans les applications basées sur les modules Angular, fournir le `HttpClient` consiste à importer le `HttpClientModule` depuis le module racine de l'application :



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

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

Par défaut, le service `HttpClient` réalise des requêtes AJAX avec `XMLHttpRequest`.

Il propose plusieurs méthodes, correspondant au verbes HTTP communs :

- `get`
- `post`
- `put`
- `delete`
- `patch`
- `head`
- `jsonp`

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 disponibles 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 `HttpClient`, une telle requête est triviale :

```
http
  .get<Array<RaceModel>>(`${baseUrl}/api/races`)
```

Cela retourne un Observable, et tu peux donc t'y abonner pour obtenir la réponse. Le corps de la réponse est la partie la plus intéressante, et est directement émis par l'Observable :

```
http.get<Array<RaceModel>>(`${baseUrl}/api/races`).subscribe((response: Array
<RaceModel>) => {
  console.log(response);
  // logs the array of races
});
```

Note que tu n'as pas besoin de dé-sérialiser le corps de la réponse de chaîne de caractères en objet ou tableau JavaScript. Tout est fait automatiquement par Angular. Cependant, Angular ne vérifiera pas si le JSON de la réponse correspond au type générique indiqué, et si l'interface `RaceModel` décrit effectivement le JSON renvoyé par le serveur.

Évidemment, tu peux accéder à la réponse HTTP complète. L'objet retourné est alors une `HttpResponse`, avec quelques champs comme le champ `status` (code de la réponse), le champ `headers`, etc.

```
http
  .get<Array<RaceModel>>(`${baseUrl}/api/races`, { observe: 'response' })
  .subscribe((response: HttpResponse<Array<RaceModel>>) => {
    console.log(response.status); // logs 200
    console.log(response.headers.keys()); // logs []
});
```

L'observable échoue si la réponse a un statut différent de 2xx ou 3xx, et l'erreur récupérée est alors une `HttpErrorResponse`.

Envoyer des données est aussi trivial. Il suffit d'appeler la méthode `post()` ou `put()`, avec l'URL et l'objet à poster :

```
http
.post<RaceModel>(`${baseUrl}/api/races`, newRace)
```

Encore une fois, pas besoin de sérialiser l'objet à envoyer en JSON. Angular le fait pour toi. Le type générique `RaceModel` est ici, comme pour la méthode `get()`, le type du corps de la réponse. Donc cette API exemple prend un `RaceModel` comme entrée et renvoie le `RaceModel` créé.

Je ne te montrerai pas les autres méthodes, je suis sûr que tu as compris le principe.

16.2. Transformer des données

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()
  .pipe(
    // 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 `options` en paramètre optionnel, où tu peux configurer ta requête.

L'option `params` représente les paramètres de recherche (que l'on appelle aussi *query string*) à ajouter à l'URL.

```

const params = {
  sort: 'ascending',
  page: '1'
};

http
  .get<Array<RaceModel>>(`${baseUrl}/api/races`, { params })
  // will call the URL ${baseUrl}/api/races?sort=ascending&page=1
  .subscribe(response => {
    // will return the races sorted
    this.races = response;
  });

```

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 = { Authorization: `Bearer ${token}` };

http.get<Array<RaceModel>>(`${baseUrl}/api/races`, { headers }).subscribe(response =>
{
  // will return the races visible for the authenticated user
  this.races = response;
});

```

16.4. Intercepteurs

Les intercepteurs sont intéressants lorsque l'on veut... intercepter des requêtes ou des réponses dans l'application.

Par exemple, si l'on veut intercepter toutes les requêtes pour ajouter un header particulier à certaines d'entre elles, on peut écrire un intercepteur comme celui-ci :

```

export const githubAPIInterceptor: HttpInterceptorFn = (
  req: HttpRequest<unknown>,
  next: HttpHandlerFn
): Observable<HttpEvent<unknown>> => {
  // if it is a Github API request
  if (req.url.includes('api.github.com')) {
    // we need to add an OAUTH token as a header to access the Github API
    const clone = req.clone({ setHeaders: { Authorization: `token ${OAUTH_TOKEN}` } });
    return next(clone);
  }
  // if it's not a Github API request, we just hand it to the next handler
  return next(req);
}

```

```
};
```

Note que tu dois cloner la requête pour la mettre à jour (les requêtes sont immuables).

Il faut ensuite configurer le *provider* du client HTTP afin qu'il utilise l'intercepteur :

```
providers: [
  provideHttpClient(withInterceptors([githubAPIInterceptor])),
]
```

Maintenant, toute requête passera par notre intercepteur, et recevra le header que nous avons défini si nécessaire (ici les appels à l'API Github).

Il est également possible d'intercepter la réponse, ce qui peut être pratique pour gérer les erreurs de façon générique :

```
export const errorHandlerInterceptor: HttpInterceptorFn = (
  req: HttpRequest<unknown>,
  next: HttpHandlerFn
): Observable<HttpEvent<unknown>> => {
  const router = inject(Router);
  const errorHandler = inject(ErrorHandler);
  return next(req).pipe(
    // we catch the error
    tap({
      error: (errorResponse: HttpErrorResponse) => {
        // if the status is 401 Unauthorized
        if (errorResponse.status === HttpStatusCode.Unauthorized) {
          // we redirect to login
          router.navigateByUrl('/login');
        } else {
          // else we notify the user
          errorHandler.handle(errorResponse);
        }
      }
    })
  );
};
```

Ceci est l'un des cas où la fonction `inject()` doit être utilisée : comme l'intercepteur est une fonction, il n'y a pas de constructeur qui permette d'injecter les dépendances, mais la fonction `inject()` permet néanmoins de les obtenir.

16.5. Contexte

On veut parfois passer des informations à un intercepteur pour lui indiquer de traiter une requête d'une façon particulière. Depuis Angular v12, c'est possible grâce à `HttpContext`. Le contexte utilise

un token typé (`HttpContextToken`), qui permet de définir une clé comme suit :

```
export const SHOULD_NOT_HANDLE_ERROR = new HttpContextToken<boolean>(() => false);
```

On peut alors modifier l'intercepteur :

```
export const errorHandlerInterceptor: HttpInterceptorFn = (
  req: HttpRequest<unknown>,
  next: HttpHandlerFn
): Observable<HttpEvent<unknown>> => {
  const router = inject(Router);
  const errorHandler = inject(ErrorHandler);
  // if there is a context specifically asking for not handling the error, we don't
  handle it
  if (req.context.get(SHOULD_NOT_HANDLE_ERROR)) {
    return next(req);
  }
  return next(req).pipe(
    // ...
```

Toutes les méthodes HTTP acceptent une option `context`, qui est une Map que tu peux construire de façon typée en utilisant la clé précédente :

```
const context = new HttpContext().set(SHOULD_NOT_HANDLE_ERROR, true);
return http.get(`${baseUrl}/api/users`, { context });
```

16.6. Tests

Nous avons maintenant un service appelant une API REST pour récupérer les courses. Comment testons-nous ce service ?

```
@Injectable({
  providedIn: 'root'
})
export class RaceService {
  constructor(private http: HttpClient) {}

  list(): Observable<Array<RaceModel>> {
    return this.http.get<Array<RaceModel>>('/api/races');
  }
}
```

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 `HttpClient` par une implémentation bouchonnée

en important `HttpClientTestingModule`. Nous pouvons ensuite utiliser une classe fournie par le framework appelée `HttpTestingController` pour simuler les réponses HTTP :

Et on peut également ajouter quelques assertions sur la requête HTTP sous-jacente :

```
import { TestBed } from '@angular/core/testing';
import { HttpTestingController, provideHttpClientTesting } from
  '@angular/common/http/testing';
import { HttpClient, provideHttpClient } from '@angular/common/http';

describe('RaceService', () => {
  let raceService: RaceService;
  let http: HttpTestingController;

  beforeEach(() =>
    TestBed.configureTestingModule({
      providers: [provideHttpClient(), provideHttpClientTesting()]
    })
  );

  beforeEach(() => {
    raceService = TestBed.inject(RaceService);
    http = TestBed.inject(HttpTestingController);
  });

  afterEach(() => {
    http.verify();
  });

  it('should return an Observable of 2 races', () => {
    // fake response
    const hardcodedRaces = [{ name: 'London' }, { name: 'Lyon' }];

    // call the service
    let actualRaces: Array<RaceModel> = [];
    raceService.list().subscribe(races => (actualRaces = races));

    // check that the underlying HTTP request was correct
    http
      .expectOne('/api/races')
      // return the fake response when we receive a request
      .flush(hardcodedRaces);

    // check that the returned array is deserialized as expected
    expect(actualRaces.length).toBe(2);
  });
});
```

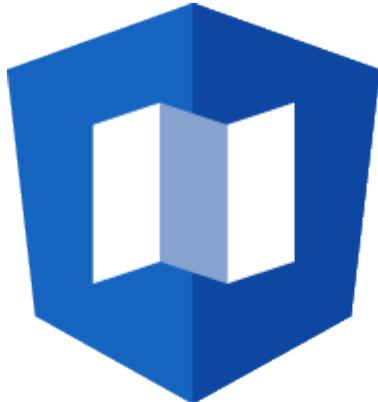
Et le tour est joué !

Essaye notre exercice [HTTP](#) et notre [quiz](#) ! Nous avons préparé une API REST complète, prête à être utilisée. Récupérons donc les courses avec le service [HttpClient](#) ! Plus tard, nous apprendrons comment appeler une API sécurisée avec un système d'authentification et des intercepteurs dans les exercices [HTTP avec authentification](#), [Parier sur un poney](#) et [Annuler un pari](#). Dans le même genre, nous utiliserons aussi les [WebSockets](#).

Chapter 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 (`provideRouter`)

Commençons à utiliser le routeur. C'est un module optionnel ; il n'est donc pas inclus dans le noyau du framework. D'une manière similaire à ce que nous avons fait pour fournir le client HTTP, tu dois fournir le routeur à l'application 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 fournir le routeur dans notre application, initialisé avec la bonne configuration :

```
import { bootstrapApplication } from '@angular/platform-browser';
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';

bootstrapApplication(AppComponent, {
  providers: [provideRouter(routes)]
});
```

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>`.

AppComponent

Header

RouterOutlet

→ le composant sera inclus ici

Footer

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--&gt;
&lt;/main&gt;
&lt;footer&gt;made with &amp;lt;3 by Ninja Squad&lt;/footer&gt;</pre>
```

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**.

Toutes les directives du routeur, y compris **RouterOutlet**, sont des directives *standalone*. Afin de pouvoir les utiliser dans le template d'un composant, elles doivent figurer dans les **imports** de son décorateur. Ou alors, tu peux ajouter le **RouterModule** aux **imports** pour mettre à disposition du template toutes les directives du routeur d'un seul coup.



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 ``. En effet, un clic sur un tel lien entraîne un rechargeement 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. 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 routerLink="/">Home</a>
<!-- same as -->
<a [routerLink]=[ '/' ]>Home</a>
```

A l'exécution, le lien `href` sera calculé par le routeur et pointera sur `/`.



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 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 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(): void {
```

```
// ... 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 [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 a un champ bien pratique : `snapshot`. Ce champ contient tous les paramètres de l'URL dans `paramMap` !

```
export class PonyComponent {
  pony: PonyModel | undefined;

  constructor(ponyService: PonyService, route: ActivatedRoute) {
    const id = route.snapshot.paramMap.get('ponyId')!;
    ponyService.get(id).subscribe(pony => (this.pony = pony));
  }
}
```

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`.



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 ni le constructeur ni le `ngOnInit` ne seront appelés 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 {
  pony: PonyModel | undefined;

  constructor(ponyService: PonyService, route: ActivatedRoute) {
    route.paramMap.subscribe((params: ParamMap) => {
      const id = params.get('ponyId')!;
      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 {
  pony: PonyModel | undefined;

  constructor(ponyService: PonyService, route: ActivatedRoute) {
    route.paramMap
      .pipe(
        map((params: ParamMap) => params.get('ponyId')!),
        switchMap(id => ponyService.get(id))
      )
      .subscribe(pony => (this.pony = pony));
  }
}
```



Essaye notre exercice [Routeur 🚗](#) pour apprendre à configurer le routeur, naviguer entre composants, et tester tout ça.

Ce que tu viens d'apprendre devrait couvrir tes besoins de routage les plus communs. Mais le routeur va bien plus loin que ça, et offre de nombreuses autres fonctionnalités. Les couvrir toutes en détails n'est pas une mince affaire, et on peut vite se sentir submergé lorsqu'on tente de tout connaître sur le routeur.

Cette section tente de présenter la plupart des fonctionnalités additionnelles, aussi brièvement que possible, en expliquant pourquoi elles peuvent être utiles. Mais elle ne couvre pas chaque petit détail. Si tu ressens le besoin d'avoir une couverture exhaustive de toutes les fonctionnalités du

routeur, il y a un [livre entier](#) pour ça, écrit par le contributeur principal du routeur d'Angular, Victor Savkin.

17.3. Redirections

Il est assez commun de vouloir qu'une URL redirige simplement vers une autre URL dans l'application. Par exemple, un site d'actualités pourrait rediriger l'URL racine vers `/breaking`, qui présenterait les toutes dernières nouvelles. Ou encore, on pourrait vouloir conserver une URL utilisée avant un refactoring et la rediriger vers l'URL correspondante après le refactoring. Cela est possible en utilisant

```
{ path: '', pathMatch: 'full', redirectTo: '/breaking' },
```

17.4. Quelle route pour une URL ?

Dans l'exemple précédent, illustrant une redirection, j'ai appliqué une stratégie pour la sélection de la route : `'full'`. La stratégie par défaut est `'prefix'`. Pour une URL donnée, cette stratégie choisit une route si l'URL commence par le *path* de la route. Si j'avais utilisé cette stratégie par défaut ici, toutes les URLs seraient redirigées vers `/breaking`, puisque toutes les URLs commencent par une chaîne vide.

La stratégie du routeur consiste à trouver la première route qui correspond à l'URL complète. Donc, par exemple, si tu définis les routes

```
{ path: 'races/:id', component: RaceComponent },
{ path: 'races/new', component: RaceCreationComponent }
```

et que l'URL est `races/new`, le composant activé par le routeur sera le `RaceComponent`. En effet, `races/:id` correspond à `races/new`, et apparaît en premier dans la liste des routes. Pour résoudre ce problème, il suffit de changer l'ordre des routes :

```
{ path: 'races/new', component: RaceCreationComponent },
{ path: 'races/:id', component: RaceComponent }
```

17.5. Routes hiérarchiques

Les routes peuvent avoir des routes filles. Cela peut être utile pour plusieurs raisons :

- appliquer des *guards* à plusieurs routes à la fois (voir plus loin) ;
- appliquer des *resolvers* à plusieurs routes à la fois (voir plus loin) ;
- appliquer un *template* commun à plusieurs routes.

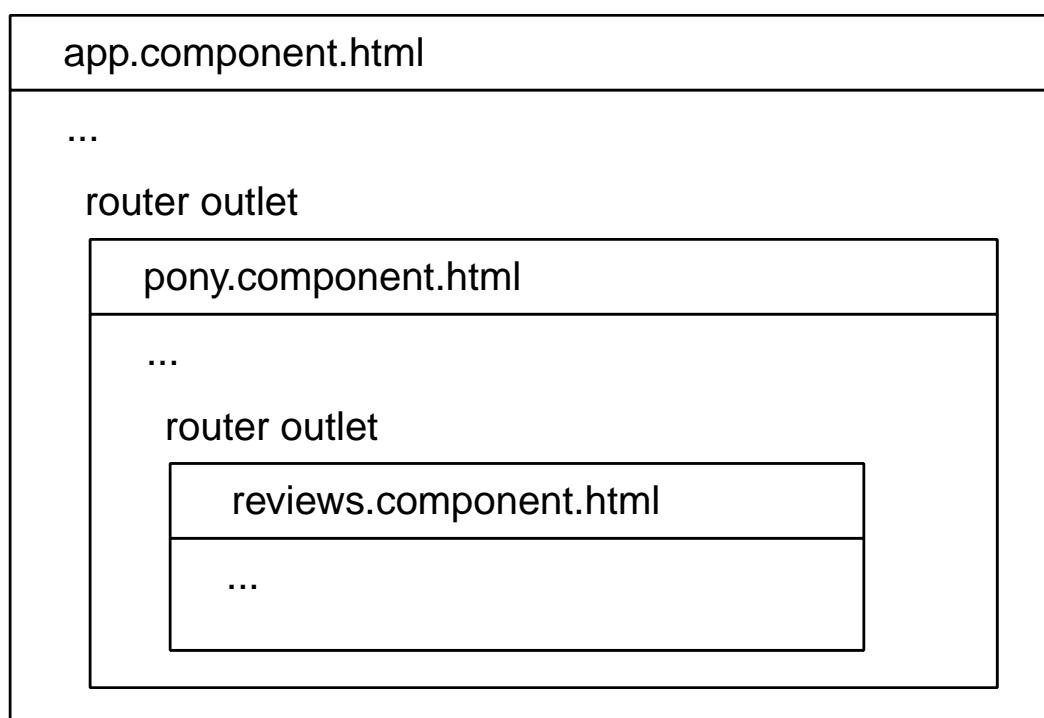
Nous avons expliqué au début de ce chapitre que, lorsque le routeur active une route, le composant de la route est inséré dans la page à l'emplacement marqué par la directive `router-outlet`.

En fait, ce mécanisme peut aussi être utilisé par les composants imbriqués. Suppose que tu veuilles créer une page complexe pour afficher le profil d'un poney. Cette page affichera le nom et le portrait du poney dans sa partie supérieure, et affichera des onglets dans sa partie inférieure : un onglet pour le certificat de naissance du poney, un autre pour son palmarès et un troisième pour les avis des journalistes. Tu veux bien sûr que chaque onglet ait sa propre URL, afin de pouvoir créer des liens directs vers chacun d'eux. Mais tu veux aussi éviter de recharger le poney, et de répéter son nom et son portrait sur chacun des trois composants correspondant aux trois onglets.

La solution est d'utiliser un `router-outlet` dans le template du `PonyComponent`, et de définir une route mère pour le poney, comme ceci :

```
{  
  path: 'ponies/:ponyId',  
  component: PonyComponent,  
  children: [  
    { path: 'birth-certificate', component: BirthCertificateComponent },  
    { path: 'track-record', component: TrackRecordComponent },  
    { path: 'reviews', component: ReviewsComponent }  
  ]  
}
```

Lorsque tu navigues vers `ponies/42/reviews`, par exemple, le routeur insère le `PonyComponent` à l'emplacement marqué par le `router-outlet` principal, dans le composant racine. Le template du `PonyComponent`, en plus du nom et du portrait du poney, contient lui aussi un `router-outlet`. C'est là que le composant de la route fille (`ReviewsComponent`) est inséré.



Si tu navigues vers `ponies/42`, le composant pony sera affiché, mais aucun des trois sous-composants ne le sera. Tâchons d'afficher plutôt le certificat de naissance par défaut. Il suffit pour cela d'ajouter une route fille, avec un chemin vide, et qui redirige vers la route `birth-certificate` :

```
{  
  path: 'ponies/:ponyId',  
  component: PonyComponent,  
  children: [  
    { path: '', pathMatch: 'full', redirectTo: 'birth-certificate' },  
    { path: 'birth-certificate', component: BirthCertificateComponent },  
    { path: 'track-record', component: TrackRecordComponent },  
    { path: 'reviews', component: ReviewsComponent }  
  ]  
}
```

Note que, dans cet exemple, la redirection est relative à la route `ponies/:ponyId` parce qu'elle ne commence pas par un `/`.

Au lieu de rediriger, on peut aussi afficher le certificat de naissance directement à l'adresse `ponies/42`. Là aussi, il suffit d'utiliser un chemin vide :

```
{  
  path: 'ponies/:ponyId',  
  component: PonyComponent,  
  children: [  
    { path: '', component: BirthCertificateComponent },  
    { path: 'track-record', component: TrackRecordComponent },  
    { path: 'reviews', component: ReviewsComponent }  
  ]  
}
```

17.6. Guards

Certaines routes de l'application ne devraient pas être accessibles à tous. L'utilisateur est-il authentifié ? A-t-il les permissions nécessaires ? Bien sûr, il faut désactiver ou masquer les liens pointant sur ces routes inaccessibles. Bien sûr, le backend doit aussi empêcher l'accès aux ressources interdites à l'utilisateur courant. Mais cela n'empêchera pas l'utilisateur d'accéder aux routes qui lui sont interdites, simplement en entrant leur URL dans la barre d'adresse du navigateur.

C'est là que les *guards* (gardes, gardiens) entrent en jeu. Il existe 4 types de *guards* :

- `canActivate` : lorsqu'un tel *guard* est appliqué à une route, il peut empêcher l'activation de la route. Il peut renvoyer une URL pour naviguer vers une autre route (pour être plus précis, il peut renvoyer un type Angular appelé `UrlTree`, voir l'exemple plus loin). C'est ce qui permet d'afficher une page d'erreur, ou de naviguer vers la page de connexion lorsqu'un utilisateur anonyme tente d'accéder à une page qui requiert une authentification ;

- **canActivateChild**: Ce *guard* peut empêcher les activations des *enfants* de la route sur lequel il est appliqué. Cela peut être utile, par exemple, pour empêcher l'accès à de nombreuses routes d'un seul coup, en fonction de leur URL ;
- **canDeactivate** : ce *guard* est différent des trois autres. Il est utilisé pour empêcher de *quitter* la route actuelle. Cela peut être utile pour, par exemple, demander une confirmation avant de quitter une page contenant un long formulaire.
- **canMatch** : ajouté en Angular v14.1, ce *guard* indique au routeur si la route peut être utilisée ou non. Si le *guard* renvoie **false**, alors la route est tout simplement ignorée et le routeur continue sa recherche. Un usage typique de ce type de *guard* est d'avoir plusieurs routes pour le même chemin avec des composants différents, et d'utiliser **canMatch** pour indiquer au routeur quel composant afficher en fonction du profil de l'utilisateur par exemple.

Voici comment appliquer un *guard* **canActivate** à une route. Les trois autres types sont appliqués de manière similaire :

```
{ path: 'races', component: RacesComponent, canActivate: [loggedInGuard] }
```

Dans l'exemple ci-dessus, **loggedInGuard** est une fonction.



Dans les premières versions d'Angular (avant la version 15.2), les *guards* étaient définis comme des services, mais cette pratique est maintenant dépréciée.

La fonction consiste simplement à décider si la route peut être activée ou non (en vérifiant si l'utilisateur est authentifié ou non), et à retourner un **boolean**, une **Promise<boolean|UrlTree>**, un **Observable<boolean|UrlTree>** ou un **UrlTree**.

Le routeur naviguera vers la route si la valeur retournée est **true**, ou si la promesse retournée est résolue à **true**, ou si l'observable retourné émet **true**. Si la valeur retournée est un **UrlTree**, alors la navigation actuelle est annulée et une nouvelle navigation vers cet **UrlTree** est déclenchée.

Voilà à quoi pourrait ressembler **loggedInGuard** :

```
export const loggedInGuard: CanActivateFn = (
  _route: ActivatedRouteSnapshot,
  _state: RouterStateSnapshot
): boolean | UrlTree => {
  const userService = inject(UserService);
  const router = inject(Router);
  // returns 'true' if the user is logged in or redirects to the login page
  // note that you can also use `router.createUrlTree()` to build a `UrlTree` with
  // parameters
  return userService.isLoggedIn() || router.parseUrl('/login');
};
```

Les routes hiérarchiques, combinées aux routes à chemin vide, sont très pratiques pour appliquer un *guard* sur de nombreuses routes d'un seul coup. Par exemple, si tu veux que les routes affichant

des courses et des poneys ne soient accessibles qu'aux utilisateurs authentifiés, au lieu de

```
{ path: 'ponies/:ponyId', component: PonyComponent, canActivate: [loggedInGuard] },
{ path: 'races', component: RacesComponent, canActivate: [loggedInGuard] }
```

tu peux introduire une route mère avec un chemin vide, et sans composant. Cette route ne consommera aucun segment d'URL, et n'activera aucun composant, mais ses *guards* seront appelés chaque fois qu'on navigue vers l'une de ses routes filles :

```
{
  path: '',
  canActivate: [loggedInGuard],
  children: [
    { path: 'ponies/:ponyId', component: PonyComponent },
    { path: 'races', component: RacesComponent }
  ]
}
```

17.7. Resolvers

Dans une bonne vieille application multi-pages, où les pages sont générées côté serveur, voilà ce qui se passe lorsqu'on clique sur un lien : une requête est envoyée au serveur, le navigateur affiche une petite animation dans l'onglet, et quand la réponse du serveur arrive enfin, alors seulement le navigateur change l'URL dans la barre d'adresse et affiche le contenu de la nouvelle page téléchargée.

Dans une application Angular mono-page, ça ne marche pas vraiment comme ça. L'utilisateur clique sur un lien pour afficher une course de poneys (par exemple). Le routeur crée une instance de `RaceComponent`, et le composant envoie une requête AJAX pour charger la course. Le routeur insère immédiatement le template du composant à l'endroit indiqué par le router-outlet. A cet instant, immédiatement après le clic, l'URL a déjà changé dans la barre d'adresse et l'utilisateur voit déjà la nouvelle page, mais sans une seule course affichée. Lorsque la réponse arrive enfin du serveur, la course est stockée dans le composant et le DOM est mis à jour pour afficher la course.

Cela a des avantages et des inconvénients :

- la navigation vers la nouvelle page a l'air plus rapide ;
- l'utilisateur peut être perturbé si le chargement de la course est trop long, parce que la page apparaît vide, ce qui ressemble à un bug ;
- le template doit être codé avec soin, parce qu'il doit fonctionner correctement pendant la courte période où la course est `null` ou `undefined` ;
- le template peut cependant donner un feedback immédiat en affichant un message ou une animation indiquant que la course est en cours de chargement ;
- Même si le chargement échoue (à cause d'une perte de connexion par exemple), au lieu de rester sur la page courante, la navigation est effectuée et l'URL change, bien que la page ne

puisse afficher aucune course.

Un *resolver* permet de faire en sorte que notre application Angular se comporte pratiquement comme une application multi-pages traditionnelle. Au lieu de charger la course depuis le composant lui-même, on applique un *resolver* à la route, et ce *resolver* charge la course.

Comme un *guard*, un *resolver* peut retourner les données de manière synchrone (en retournant une course) ou asynchrone (en retournant une promesse ou un observable de course). Le routeur ne navigue vers la route que lorsque la promesse a été résolue, ou lorsque l'observable a émis une course et terminé. Voici à quoi un *resolver* ressemble:

```
export const raceResolver: ResolveFn<RaceModel> = (route: ActivatedRouteSnapshot): Observable<RaceModel> => {
  const id = route.paramMap.get('raceId')!;
  const raceService = inject(RaceService);
  return raceService.get(id);
};
```

Comme tu peux le voir, c'est une simple fonction qui utilise le snapshot de la nouvelle route, fournie par le routeur, pour charger la course identifiée par le paramètre `raceId` et retourner un `Observable<RaceModel>`.

Voici comment le *resolver* est appliqué à la route :

```
{
  path: 'races/:raceId',
  component: RaceComponent,
  resolve: {
    race: raceResolver
  }
}
```

Le resolver est associé à une clé que j'ai choisi de nommer `race`. Le routeur utilise cette clé pour stocker la course chargée par le *resolver* dans une propriété éponyme des `data` de la route. Ainsi, le `RaceComponent` peut obtenir la course de cette manière :

```
export class RaceComponent {
  race: RaceModel;

  constructor(route: ActivatedRoute) {
    this.race = route.snapshot.data['race'];
  }
}
```

Note que, si tu navigues d'une route à la même route, mais avec des paramètres différents (par exemple, si ta page contient un lien *Course Suivante*), les *guards* et les *resolvers* appliqués à la route sont appelés à nouveau. La même instance de composant, dans ce cas, sera néanmoins réutilisée, et

il te faut donc souscrire à un observable pour obtenir la course (ou stocker l'observable dans le composant et utiliser le `pipe async` dans le template) :

```
export class RaceComponent {
  race!: RaceModel;

  constructor(route: ActivatedRoute) {
    route.data.subscribe(data => (this.race = data['race']));
  }
}
```

Les *resolvers* ont de nombreux avantages par rapport au chargement des données depuis le composant qui les affiche :

- ils rendent la navigation plus traditionnelle ;
- ils peuvent être partagés par plusieurs routes ;
- ils simplifient le code du composant et de son template : plus besoin de charger la donnée ; plus besoin de se préoccuper de la période pendant laquelle le modèle est absent ; plus besoin d'appliquer des opérateurs RxJS relativement complexes pour obtenir les données à partir des paramètres ;
- si la navigation échoue, la page courante reste affichée, et il suffit à l'utilisateur de cliquer sur le lien pour refaire une tentative.

Le seul inconvénient que je peux trouver aux *resolvers* est que, dans le cas où le chargement est long, l'application peut sembler ne pas réagir. L'utilisateur clique sur un lien, et rien de visible ne se produit avant que les données aient été chargées par le *resolver*. Si tu sais que le chargement des données d'une page est long (parce que le serveur doit faire des calculs complexes, ou interroger des services externes), naviguer directement vers le composant et lui faire afficher une animation ou un message d'attente peut être plus ergonomique. Un autre moyen de régler ce problème est de s'appuyer sur les événements de routage pour afficher ce message d'attente.

17.8. Événements de routage

Le routeur émet plusieurs événements lorsqu'on navigue vers une route. Tu peux être notifié de ces événements en souscrivant à l'observable `events` du service `Router`. Les événements émis sont de plusieurs types, que tu peux filtrer en utilisant `instanceof`. Par exemple : `event instanceof NavigationStart`.

Voici les différents types d'événements :

- `NavigationStart` : émis lorsqu'une navigation est demandée. Par exemple lorsqu'on clique sur un lien. Cet événement peut être utilisé pour afficher un indicateur animé ;
- `NavigationEnd` : émis lorsque la navigation s'achève avec succès. Il peut être utilisé pour masquer l'indicateur animé. Un autre cas d'utilisation est l'envoi d'un *hit* à un service de statistiques de consultation (comme Google Analytics par exemple). Pratique pour connaître les pages populaires ou à l'inverse les fonctionnalités peu utilisées dans l'application ;

- **NavigationError** : émis lorsque la navigation échoue à cause d'une erreur inattendue (comme un *resolver* retournant un observable vide ou en erreur). Il peut être utilisé pour masquer l'indicateur animé, ou pour tenter d'envoyer une erreur de navigation au serveur, par exemple ;
- **NavigationCancel** : émis lorsqu'une navigation est annulée, parce qu'un *guard* a empêché la navigation, typiquement. Là aussi, l'indicateur animé de navigation doit être masqué.

D'autres événements existent pour le chargement des configurations de routes (**RouteConfigLoadStart**, **RouteConfigLoadEnd**, **RoutesRecognized**) et, depuis la version 4.3, pour les resolvers (**ResolveStart**, **ResolveEnd**) et les guards (**GuardsCheckStart**, **GuardsCheckEnd**). La version 5.0 a également introduit des événements de navigation plus fins (**ChildActivationStart**, **ChildActivationEnd**). La version 6.1 a elle ajouté un événement **Scroll**, en même temps que l'option de configuration **scrollPositionRestoration** qui permet de restaurer la position de scroll lorsque l'on navigue vers un composant précédent.

17.9. Paramètres et data

Nous avons vu plus tôt que les routes pouvaient avoir des paramètres. Par exemple, la route `races/:raceId` a un paramètre nommé `raceId`. La valeur de ce paramètre, lorsqu'on navigue vers `/races/42`, est la chaîne de caractères '`42`'. Mais cette route peut en fait avoir d'autres paramètres additionnels appelés *matrix parameters*.

 Les paramètres *matrix* ne sont pas spécifiques à Angular. Bien qu'assez rarement utilisés et donc moins connus que les paramètres de requête, ils constituent des éléments standards des URIs, qui sont supportés par plusieurs frameworks utilisés côté serveur.

Si tu navigues vers l'URL

```
/races/42;foo=bar;baz=wiz
```

les propriétés `params` et `paramMap` de la route activée contiendront deux paramètres additionnels 'foo' et 'baz' ayant respectivement les valeurs 'bar' et 'wiz'.

Ces paramètres *matrix* sont spécifiques à la route. Ainsi, par exemple, si l'URL est

```
/races/42;foo=bar;baz=wiz/ponies
```

alors le composant associé au segment `ponies` de la route n'aura pas `foo` ni `bar` dans les paramètres de son `ActivatedRoute`. Seul le composant associé au segment `races/42` les aura.

Pour naviguer vers cette URL, on utilise le code suivant :

```
router.navigate(['/races', 42, { foo: 'bar', baz: 'wiz' }, 'ponies']);
```

ou, depuis le template, le router link équivalent :

```
<a [routerLink]="/races", 42, { foo: 'bar', baz: 'wiz' }, 'ponies'">Link</a>
```

Les paramètres de requête (*query parameters*), en revanche, sont partagés par tous les segments de la route. Ils ressemblent à ceci dans l'URL :

```
/races/42/ponies?foo=bar&baz=wiz
```

Ces paramètres de requête sont donc accessibles depuis n'importe quelle route, en utilisant les propriétés `queryParams` ou `queryParamMap`.

Pour naviguer vers une URL comme celle-là, on utilise le code suivant :

```
router.navigate(['/races', 42, 'ponies'], { queryParams: { foo: 'bar', baz: 'wiz' } });
```

ou, depuis le template, le router link équivalent :

```
<a [routerLink]="/races", 42, 'ponies" [queryParams]={{ foo: 'bar', baz: 'wiz' }}>Link</a>
```

Enfin, nous avons vu que les *resolvers* permettaient d'ajouter des propriétés aux `data` de `ActivatedRoute`, avant que la route ne soit activée. Il est aussi possible d'y ajouter des propriétés directement depuis la configuration de la route. Cela peut être utile lorsque le même composant doit pouvoir être utilisé dans deux contextes différents, par exemple :

```
{
  path: 'races',
  component: RacesComponent,
  data: {
    allowDeletion: false
  }
}
```

17.10. Lier les paramètres et data aux inputs des composants

Nous avons vu que les paramètres et les données de la route sont accessibles depuis le composant associé à la route, via des observables sur l'objet `ActivatedRoute`.

Depuis Angular v16, il est possible de lier automatiquement les paramètres et les données de la route aux inputs du composant.

Il suffit pour cela de configurer le router avec `withComponentInputBinding` :

app.config.ts

```
provideRouter(routes, withComponentInputBinding())
```

Avec cette option, un composant peut déclarer un input avec le même nom qu'un paramètre ou une donnée de la route, et Angular se chargera de lier automatiquement la valeur du paramètre ou de la donnée à cet input.

race.component.ts

```
export class RaceComponent implements OnChanges {
  raceModel$!: Observable<RaceModel>;
  @Input({ required: true }) raceId!: string;
```

On peut ensuite utiliser cet input comme un input classique, et réagir à son changement avec `ngOnChanges` ou en utilisant un setter pour cet input :

race.component.ts

```
constructor(private raceService: RaceService) {}

ngOnChanges() {
  this.raceModel$ = this.raceService.get(this.raceId);
}
```

17.11. Chargement à la demande

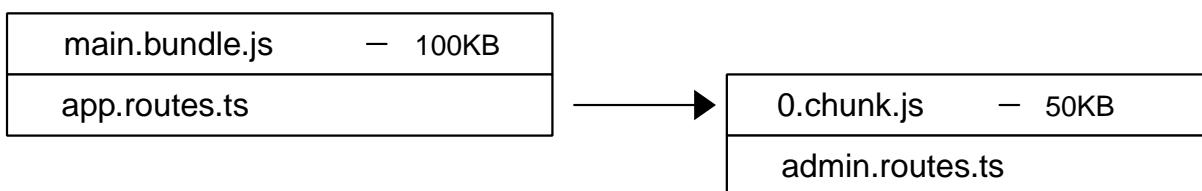
Cette section clôture ce long chapitre sur le routeur d'Angular.

Inévitamment, les applications s'enrichissent de plus en plus de fonctionnalités, et leur taille ne fait qu'augmenter, jusqu'au point où ça devient un problème : le fichier JS consomme trop de bande passante et met trop longtemps à être téléchargé et analysé, en particulier sur les téléphones mobiles et les réseaux de mauvaise qualité. Par ailleurs, le nombre de fonctionnalités augmentant, certaines d'entre elles ne sont utilisées que par certains utilisateurs, ou par tous, mais rarement. Charger ces fonctionnalités dès le démarrage de l'application est un gaspillage de bande passante et une perte de temps pour l'utilisateur. C'est là que le chargement à la demande (*lazy loading*) devient intéressant.



Le chargement à la demande consiste à diviser l'application en plusieurs bundles JavaScript, en chargeant des routes filles à la demande. Le tableau de routes de l'application, au lieu de définir toutes les routes de l'application, contient seulement les routes chargées dès le démarrage, ainsi que des routes mères, dont les routes filles sont inconnues au départ et chargées dynamiquement, à la demande.

Lorsque l'utilisateur navigue vers le chemin d'une route fille (encore inconnue), le routeur télécharge le bundle JavaScript contenant les routes filles (et leurs composants et services associés), puis ajoute ces routes, composants et services à l'application.



Il est en réalité possible de pré-charger les bundles en tâche de fond, après que l'application ait démarré, sans attendre que l'utilisateur ait navigué vers une route *lazy*, grâce à une `preloadingStrategy` alternative.

Pour illustrer comment configurer concrètement un chargement à la demande, nous allons supposer que tu veux définir une section dédiée aux administrateurs de l'application, qui doit être chargée à la demande.

La première étape consiste à définir un composant `admin`, et au moins une route pour ce composant dans un fichier `admin.routes.ts` :

```
export const adminRoutes: Routes = [{ path: '', component: AdminComponent }];
```

L'étape finale (et oui, c'est tout ce qu'il y a à faire) est d'ajouter une route dans le fichier de configuration des routes principales (`app.routes.ts`), et d'indiquer au routeur qu'il faut charger les routes d'administration lorsqu'on navigue vers cette route (ou une des routes filles qu'il pourrait avoir) :

```
{ path: 'admin', loadChildren: () => import('./admin/admin.routes').then(m => m.adminRoutes) }
```

Comme tu le constates, ceci est réalisé grâce à la propriété `loadChildren` de la route d'administration et à la fonction d'import dynamique de TypeScript.

Lorsque l'application est construite, Angular CLI analyse la configuration des routes et détecte que le module d'administration doit être chargé à la demande. Sans aucune autre configuration nécessaire, il génère un bundle JavaScript supplémentaire (nommé `0.chunk.js`), et génère le code JavaScript nécessaire pour télécharger ce fichier lorsque le routeur requiert le chargement de `'./admin/admin.routes'`.

Il est même possible de simplifier un peu l'import si tu utilises un export `default` pour le module :

```
const defaultAdminRoutes: Routes = [{ path: '', component: AdminComponent }];
export default defaultAdminRoutes;
```

Le routeur va alors "développer" l'import automatiquement :

```
{ path: 'admin', loadChildren: () => import('./admin/admin.routes') }
```

Enfin, si comme dans l'exemple ci-dessus, il n'y a qu'un composant à charger à la demande plutôt qu'un ensemble de routes, alors c'est encore plus simple : il n'y a pas besoin de définir de fichier de routes supplémentaire. Tu peux charger à la demande le composant lui-même :

```
{ path: 'admin', loadComponent: () => import('./admin/admin.component').then(m => m.AdminComponent) }
```



Essaye notre [quiz 🎯](#) et nos exercices [Protected routes with guards 🎯](#), [Resolvers](#), [child routes and redirections 🎯](#) et [Lazy loading 🎯](#) pour apprendre les fonctionnalités avancées du routeur.

Chapter 18. Formulaires

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';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'ns-register',
```

```

template: `

<h2>Sign up</h2>
<form></form>
`,

standalone: true,
imports: [CommonModule]
)

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); // '' until the user enters a value
console.log(password.hasError('required')); // false
password.disable(); // disables the control
password.reset(); // resets the value
```

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: ''}
console.log(form.controls.username); // logs the Control
```

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 chaque composant qui utilise un formulaire piloté par le template.



Au contraire des directives de `CommonModule` et `RouterModule`, qui sont *standalone*, les directives de `FormsModule` ne le sont pas. Donc on ne peut pas les importer une par une dans les composants. Le module `FormsModule` tout entier doit être importé.

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'ns-register',
  template: `
    <h2>Sign up</h2>
    <form></form>
  `,
  standalone: true,
  imports: [CommonModule, FormsModule]
})
export class RegisterFormComponent {}
```

`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';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'ns-register',
  template: `
    <h2>Sign up</h2>
    <form (ngSubmit)="register()">
      <button type="submit">Register</button>
    </form>
  `,
  standalone: true,
```

```

    imports: [CommonModule, FormsModule]
)
export class RegisterFormComponent {
  register(): void {
    // 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.

Tu te demandes peut-être pourquoi il y a une directive `NgForm` sur l'élément `form`, alors qu'il ne porte aucun attribut particulier. C'est simplement parce que le sélecteur de la directive `NgForm` est `form` (il est en fait un peu plus spécifique que cela), ce qui signifie que chaque élément HTML `<form>` déclenche la création d'une directive `NgForm`, pour autant que le `FormsModule` ait été importé.

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';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.component.html',
  standalone: true,
  imports: [CommonModule, FormsModule]
})
export class RegisterFormComponent {
  register(): void {
    // 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';
import { UserModel } from './user.model';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: '../code/register-form.component.html',
  standalone: true,
  imports: [CommonModule, FormsModule]
})
export class RegisterFormComponent {
  register(user: UserModel): void {
    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 interface `UserModel` :

```
export interface UserModel {
  username: string;
  password: string;
}
```

Notre `RegisterFormComponent` doit avoir un attribut `user` de type `UserModel` :

```
@Component({
  selector: 'ns-register',
  templateUrl: './register-form.component.html',
  standalone: true,
  imports: [CommonModule, FormsModule]
})
export class RegisterFormComponent {
  user: UserModel = {
    username: '',
    password: ''
  };

  register(): void {
    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, ReactiveFormsModule } from '@angular/forms';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.component.html',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule]
})
export class RegisterFormComponent {
  constructor(fb: FormBuilder) {
    // we will have to build the form
  }

  register(): void {
    // 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 { CommonModule } from '@angular/common';
import { FormControl, FormBuilder, FormGroup, ReactiveFormsModule } from
  '@angular/forms';

@Component({
```

```

    selector: 'ns-register',
    templateUrl: './register-form.component.html',
    standalone: true,
    imports: [CommonModule, ReactiveFormsModule]
)
export class RegisterFormComponent {
  userForm: FormGroup<{
    username: FormControl<string | null>;
    password: FormControl<string | null>;
  }>;
  constructor(fb: FormBuilder) {
    this.userForm = fb.group({
      username: '',
      password: ''
    });
  }
  register(): void {
    // 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 { CommonModule } from '@angular/common';
import { FormControl, FormBuilder, FormGroup, ReactiveFormsModule } from
  '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.component.html',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule]
)
export class RegisterFormComponent {
  userForm: FormGroup<{
    username: FormControl<string | null>;
    password: FormControl<string | null>;
  }>;
  constructor(fb: FormBuilder) {
    this.userForm = fb.group({
      username: '',
      password: ''
    });
  }
  register(): void {
    console.log(this.userForm.value);
  }
}

```

```

}};

constructor(fb: FormBuilder) {
  this.userForm = fb.group({
    username: fb.control(''),
    password: fb.control('')
  });
}

register(): void {
  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 composant. 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 { CommonModule } from '@angular/common';
import { FormBuilder, FormGroup, FormControl, ReactiveFormsModule } from
  '@angular/forms';

```

```

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.component.html',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule]
})
export class RegisterFormComponent {
  usernameCtrl: FormControl<string | null>;
  passwordCtrl: FormControl<string | null>;
  userForm: FormGroup<{
    username: FormControl<string | null>;
    password: FormControl<string | null>;
  }>;
}

constructor(fb: FormBuilder) {
  this.usernameCtrl = fb.control('');
  this.passwordCtrl = fb.control('');
  this.userForm = fb.group({
    username: this.usernameCtrl,
    password: this.passwordCtrl
  });
}

reset(): void {
  this.usernameCtrl.setValue('');
  this.passwordCtrl.setValue('');
}

register(): void {
  console.log(this.userForm.value);
}
}

```

Comme il est un peu pénible de définir les types des champs de la classe, on peut directement utiliser le `FormBuilder` pour les initialiser et laisser TypeScript inférer les types :

```

import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormBuilder, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.component.html',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule]
})
export class RegisterFormComponent {
  usernameCtrl = this.fb.control('');
  passwordCtrl = this.fb.control('');
}

```

```

userForm = this.fb.group({
  username: this.usernameCtrl,
  password: this.passwordCtrl
});

constructor(private fb: FormBuilder) {}

reset(): void {
  this.usernameCtrl.setValue('');
  this.passwordCtrl.setValue('');
}

register(): void {
  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.min(n)` (disponible depuis la version 4.2) pour s'assurer que la valeur entrée vaut au moins *n* ;
- `Validators.max(n)` (disponible depuis la version 4.2) pour s'assurer que la valeur entrée vaut au plus *n* ;
- `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 { CommonModule } from '@angular/common';
import { FormBuilder, ReactiveFormsModule, Validators } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.component.html',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule]
})
export class RegisterFormComponent {
  userForm = this.fb.group({
    username: this.fb.control('', [Validators.required, Validators.minLength(3)]),
    password: this.fb.control('', Validators.required)
  });

  constructor(private fb: FormBuilder) {}

  register(): void {
    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` (`min` et `max` ne sont en revanche pas encore disponibles comme des directives).

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

Évidemment, 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.controls.username.hasError('required')">Username is required</div>
    <div *ngIf="userForm.controls.username.hasError('minlength')">Username should be 3 characters min</div>
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    <div *ngIf="userForm.controls.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.controls.username.dirty &&
userForm.controls.username.hasError('required')">
      Username is required
    </div>
    <div *ngIf="userForm.controls.username.dirty &&
userForm.controls.username.hasError('minlength')">
      Username should be 3 characters min
    </div>
  </div>
  <div>
    <label>Password</label><input type="password" formControlName="password">
    <div *ngIf="userForm.controls.password.dirty &&
userForm.controls.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',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule]
})
export class RegisterFormComponent {
  usernameCtrl = this.fb.control('', Validators.required);
  passwordCtrl = this.fb.control('', Validators.required);
  userForm = this.fb.group({
    username: this.usernameCtrl,
    password: this.passwordCtrl
  });
}

```

```

constructor(private fb: FormBuilder) {}

register(): void {
  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: AbstractControl<Date | null>) => {
  // 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 { CommonModule } from '@angular/common';
import { AbstractControl, FormBuilder, ReactiveFormsModule, ValidationErrors,
Validators } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.component.html',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule]
})

export class RegisterFormComponent {
  usernameCtrl = this.fb.control('', Validators.required);
  passwordCtrl = this.fb.control('', Validators.required);
  birthdateCtrl = this.fb.control('', [Validators.required, RegisterFormComponent
.isOldEnough]);
  userForm = this.fb.group({
    username: this.usernameCtrl,
    password: this.passwordCtrl,
    birthdate: this.birthdateCtrl
  });

  static isOldEnough(control: AbstractControl): ValidationErrors | null {
    // 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(private fb: FormBuilder) {}

  register(): void {
    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',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule]
})
export class RegisterFormComponent {
  usernameCtrl = this.fb.control('', Validators.required, control => this
  .isUsernameAvailable(control));
  userForm = this.fb.group({
    username: this.usernameCtrl
  });

  constructor(
    private fb: FormBuilder,
    private userService: UserService
  ) {}

  isUsernameAvailable(control: AbstractControl): Observable<ValidationErrors | null> {
    const username = control.value;
    return this.userService
  }
}

```

```

    .isUsernameAvailable(username)
    .pipe(map(available => (available ? null : { alreadyUsed: true })));
}

register(): void {
  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...

18.8. 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.

D'abord, on crée un nouveau groupe `passwordGroup`, avec les deux champs, et on l'ajoute dans le groupe `userForm` avec la clé `passwordForm` :

```

import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { AbstractControl, FormBuilder, ReactiveFormsModule, ValidationErrors,
Validators } from '@angular/forms';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.component.html',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule]
})
export class RegisterFormComponent {

```

```

usernameCtrl = this.fb.control('', Validators.required);
passwordCtrl = this.fb.control('', Validators.required);
confirmCtrl = this.fb.control('', Validators.required);
passwordGroup = this.fb.group(
  { password: this.passwordCtrl, confirm: this.confirmCtrl },
  { validators: RegisterFormComponent.passwordMatch }
);

userForm = this.fb.group({ username: this.usernameCtrl, passwordForm: this
.passwordGroup });

static passwordMatch(group: AbstractControl): ValidationErrors | null {
  const password = group.value.password;
  const confirm = group.value.confirm;
  return password === confirm ? null : { matchingError: true };
}

constructor(private fb: FormBuilder) {}

register(): void {
  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="passwordGroup.dirty && passwordGroup.hasError('matchingError')">Your password does not match</div>
  </div>

```

```
<button type="submit" [disabled]="userForm.invalid">Register</button>
</form>
```

Et voilà !

18.9. 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 { CommonModule } from '@angular/common';
import { FormBuilder, ReactiveFormsModule, Validators } from '@angular/forms';
import { debounceTime, distinctUntilChanged } from 'rxjs';

@Component({
  selector: 'ns-register',
  templateUrl: './register-form.component.html',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule]
})
export class RegisterFormComponent {
  usernameCtrl = this.fb.control('', Validators.required);
  passwordCtrl = this.fb.control('', Validators.required);
  userForm = this.fb.group({
    username: this.usernameCtrl,
    password: this.passwordCtrl
  });
  passwordStrength = 0;

  constructor(private fb: FormBuilder) {
    // we subscribe to every password change
    this.passwordCtrl.valueChanges
      .pipe(
        // only recompute when the user stops typing for 400ms
        debounceTime(400),
        // only recompute if the new value is different from the last
        distinctUntilChanged()
      )
      .subscribe(newValue => (this.passwordStrength = newValue ? newValue.length : 0));
  }

  register(): void {
    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 `HttpClient` utilise des observables également.

18.10. Mettre à jour seulement à la perte de focus ou à la soumission

Angular 5.0 a introduit la possibilité d'attendre la perte de focus (`blur`) ou la soumission du formulaire (`submit`) pour mettre à jour la valeur et la validité d'un champ. Pour cela, le constructeur de `FormControl` accepte à présent un objet `options` comme second paramètre, qui permet de définir les validateurs synchrones et asynchrones, ainsi que l'option `updateOn`. Sa valeur peut-être :

- **change**, par défaut, la valeur et la validité sont mises à jour à chaque changement ;
- **blur**, la valeur et la validité sont mises à jour uniquement quand le champ perd le focus ;
- **submit**, la valeur et la validité sont mises à jour uniquement quand le formulaire parent est soumis.

```
this.usernameCtrl = new FormControl('', Validators.required);
this.passwordCtrl = new FormControl('', {
  validators: Validators.required,
  updateOn: 'blur'
});
```

Il est aussi possible de configurer cette option sur tous les champs d'un groupe à la fois :

```
this.userForm = new FormGroup(
{
  username: this.usernameCtrl,
  password: this.passwordCtrl
},
{
  updateOn: 'blur'
});
```

La même fonctionnalité existe aussi dans les formulaires pilotés par le template, grâce à l'input **ngModelOptions** de la directive **NgModel** :

```
<label>Username</label>
<input name="username" #usernameCtrl="ngModel"
  [(ngModel)]="user.username" [ngModelOptions]="{ updateOn: 'blur' }" required>
<div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('required')">Username is
required</div>
```

ou globalement sur un formulaire avec l'input **NgFormOptions** (apparu avec Angular 5.0) de la directive **NgForm** :

```
<form (ngSubmit)="register()" [ngFormOptions]="{ updateOn: 'blur' }">
<div>
  <label>Username</label>
  <input name="username" #usernameCtrl="ngModel"
    [(ngModel)]="user.username" required>
  <div *ngIf="usernameCtrl.dirty && usernameCtrl.hasError('required')">Username is
required</div>
```

18.11. FormArray et FormRecord

`FormControl` et `FormGroup` ne sont pas les seules entités que l'on peut utiliser pour construire un formulaire. Si tu souhaites qu'une partie du formulaire soit dynamique, tu peux utiliser un `FormArray` ou un `FormRecord`.

`FormArray` est un tableau d'éléments qui contiennent les mêmes types de valeur. Un exemple typique serait un formulaire où les utilisateurs peuvent ajouter/supprimer des valeurs, par exemple pour ajouter/supprimer des mot-clés sur un article de blog :

```
export class EditBlogPostComponent {
  // one empty tag by default
  tagsArray = this.fb.array(['']);
  blogPostForm = this.fb.group({
    title: '',
    content: '',
    tags: this.tagsArray
  });

  constructor(private fb: FormBuilder) {}

  addTag() {
    this.tagsArray.push(this.fb.control(''));
  }

  removeTag(index: number) {
    this.tagsArray.removeAt(index);
  }
}
```

On peut ensuite itérer sur les champs du tableau dans le template, et avoir des boutons d'ajout/suppression d'un mot-clé :

```
<div formArrayName="tags">
  <label>Tags</label>
  <button id="add-tag" (click)="addTag()">Add tag</button>
  <div *ngFor="let tagControl of tagsArray.controls; index as i">
    <input class="tag" [formControlName]="i" />
    <button class="remove-tag" (click)="removeTag(i)">Remove tag</button>
  </div>
</div>
```

La valeur du `FormArray` est un tableau de chaînes de caractères dans cet exemple. Bien sûr, un `FormArray` peut contenir n'importe quel type d'élément. Par exemple, si tu veux ajouter des lignes à une facture, le `FormArray` peut contenir des `FormGroup` avec un nom, une quantité, un prix, etc.

`FormRecord` est une autre entité utilisable dans les formulaires. Il a été introduit en Angular v14, et permet de construire des dictionnaires de clés/valeurs.

Disons que l'on veut un formulaire pour lister l'équipement à préparer pour un voyage, avec la possibilité d'ajouter/supprimer des éléments, et de pouvoir les cocher/décocher. `FormRecord` peut aider dans ce cas :

```
export class EditPackingListComponent {
  equipmentRecord = this.fb.record({
    // always bring your toothbrush
    toothbrush: true
  });
  packingListForm = this.fb.group({
    equipments: this.equipmentRecord
  });

  constructor(private fb: NonNullableFormBuilder) {}

  addEquipment(equipment: string) {
    this.equipmentRecord.addControl(equipment, this.fb.control(true));
  }

  removeEquipment(equipment: string) {
    this.equipmentRecord.removeControl(equipment);
  }
}
```

On peut là-aussi itérer sur les champs dans le template, et avoir des boutons d'ajout/suppression d'un équipement :

```
<div formGroupName="equipments">
  <label>Equipements</label>
  <input #equipment name="equipement-to-add" />
  <button id="add-equipment" (click)="addEquipment(equipment.value)">Add</button>
  <div *ngFor="let control of equipmentRecord.controls | keyvalue">
    <label [for]="'eq-' + control.key">{{ control.key }}</label>
    <input [id]="'eq-' + control.key" type="checkbox" class="equipment"
      [formControlName]="control.key" />
    <button class="remove-equipment" (click)="removeEquipment(control.key)">Remove
      equipment</button>
  </div>
</div>
```

La valeur du `FormRecord` sera alors un objet, avec le nom de l'équipement comme clé et un booléen comme valeur. Par exemple :

```
{
  toothbrush: false,
  jacket: true
}
```

18.12. Des formulaires typés

Jusqu'à la version 14 d'Angular, les formulaires n'étaient pas typés. Qu'entend-on par là ?

Et bien la valeur d'un `FormControl` ou d'un `FormGroup` était de type `any`, ce qui évidemment, est loin d'être idéal.

Si vous migrez une application de la version 13 à la version 14, afin de ne pas casser votre code, les types `FormControl`, `FormGroup` et `FormArray` vont être remplacés respectivement par `UntypedFormControl`, `UntypedFormGroup` et `UntypedFormArray`.

Les types originaux sont à présent typés. Mais qu'est-ce qui a changé concrètement avec Angular 14 ?

Les éléments de formulaire sont devenus génériques. Ainsi, on utilise à présent des `FormControl<string>` par exemple, pour indiquer que la valeur du contrôle est une chaîne de caractère. Les `FormGroup`, quant à eux, prennent le type des contrôles qu'ils contiennent. Par exemple, un `FormGroup` utilisé pour l'enregistrement aura le type

```
FormGroup<{
  username: FormControl<string>;
  password: FormControl<string>;
}>;
```

C'est verbeux, mais pas de panique : ces types génériques sont la plupart du temps inférés par le compilateur à l'initialisation.

18.12.1. Nullabilité

J'ai quelque peu simplifié les choses dans la section précédente. En effet, le typage des formulaires doit prendre en compte deux dures réalités :

1. certains éléments de formulaire peuvent être désactivés
2. le comportement de la méthode `reset()` d'Angular est de peupler les champs avec la valeur `null`.

Lorsqu'un contrôle est désactivé, sa valeur n'apparaît plus dans la valeur du `FormGroup` auquel il appartient. Ainsi, si je décide de désactiver le champ `password` de mon formulaire, la valeur du `FormGroup` sera simplement

```
{
  username: 'cedric'
}
```

Et ça, c'est un peu gênant, parce que cela implique que la valeur du `FormGroup` n'est pas de type

```
{
  username: string;
```

```
    password: string;  
}
```

comme on pourrait s'y attendre, mais bien

```
{  
  username?: string;  
  password?: string;  
}
```

En effet, Angular ne peut savoir à l'avance si vous comptez ou pas désactiver les champs du formulaire. Il est donc obligé de rendre chacune des propriétés optionnelles dans sa valeur. À vous, donc, de gérer cela, typiquement en utilisant `value.username!` pour obtenir une valeur de type `string` plutôt qu'une valeur de type `string | undefined`, si vous savez qu'il n'est pas désactivé. Une autre possibilité est d'utiliser la valeur `raw` du `FormGroup` (retournée par `getRawValue()`). Cette valeur `raw` contient toutes les propriétés, que leur contrôle soit désactivé ou pas. Et ces propriétés ne sont donc pas marquées comme optionnelles.

Le deuxième point pose le même genre de problème. Comme Angular ne peut savoir à l'avance si vous appellerez ou pas la méthode `reset()`, la valeur du `FormGroup` est en réalité de type

```
{  
  username?: string | null;  
  password?: string | null;  
}
```

Ce deuxième désagrément, au contraire du premier, peut être évité. Il faut pour cela configurer chacun des éléments du formulaire avec l'option `nonNullable: true`. Cette option change le comportement de `reset()`, qui va peupler le champ avec sa valeur initiale plutôt qu'avec `null`. Configurer cette option sur chacun des éléments de formulaire est assez pénible et verbeux. C'est néanmoins ce que vous devez faire si vous utilisez les constructeurs pour créer vos éléments de formulaire. C'est plus simple si vous choisissez d'utiliser le `FormBuilder`. Il suffit de vous injecter à la place de `FormBuilder` un `NonNullableFormBuilder`, et tous les éléments qu'il créera auront l'option `nonNullable`.

```
export class RegisterFormComponent {  
  userForm = this.fb.group({  
    username: '',  
    password: ''  
  });  
  
  constructor(private fb: NonNullableFormBuilder) {}  
}
```

Il faut noter que le nom `NonNullableFormBuilder` est trompeur. L'utiliser n'empêche pas d'avoir des contrôles avec la valeur `null`. Prenons l'exemple d'un champ de type `number`. Il n'y a pas vraiment

d'autre valeur par défaut possible que `null` dans ce cas. Et même si vous préinitialisez le champ avec une valeur non-null, l'utilisateur pourra toujours vider le champ et donc lui donner la valeur `null`. Dans ce cas, il vous faudra typer explicitement le contrôle :

```
birthYearCtrl = new FormControl<number | null>(null);
```

ou, avec le `FormBuilder` ou le `NonNullableFormBuilder` :

```
birthYearCtrl = this.formBuilder.control<number | null>(null);
```

ou si ce contrôle fait partie d'un `FormGroup` :

```
formGroup = this.formBuilder.group({
  // ...
  birthYear: null as number | null
});
```

Toute cette complexité additionnelle peut sembler rebutante, mais il ne faut pas perdre de vue ce qu'on y a gagné : des objets avec des propriétés dont le nom et le type sont connus, et qui peuvent être auto-complétées par l'IDE. Au final donc, du code plus maintenable et plus robuste.

18.13. Des messages d'erreur de validation super simples avec `ngx-valdemort`

Comme tu l'as sans doute remarqué, les templates des formulaires deviennent rapidement bavards avec leurs messages d'erreurs à répéter pour chaque type d'erreur de chaque champ de chaque formulaire. On se retrouve vite avec des `ngIf` à rallonge, et des copier/coller entre composants.

Et comme on trouve ça désagréable nous aussi, nous avons écrit une petite bibliothèque open-source pour simplifier tout ça (fortement inspirée de `ngMessages` en AngularJS) : `ngx-valdemort`.

À la place de :

```
<input id="email" formControlName="email" class="form-control" type="email" />
<div
  class="invalid-feedback"
  *ngIf="form.controls.email.invalid && (f.submitted || form.controls.email.touched)">
</div>
<div *ngIf="form.controls.email.hasError('required')">The email is required</div>
<div *ngIf="form.controls.email.hasError('email')">The email must be a valid email address</div>
```

la bibliothèque permet d'écrire :

```

<input id="email" formControlName="email" class="form-control" type="email" />
<val-errors controlName="email">
  <ng-template valError="required">The email is required</ng-template>
  <ng-template valError="email">The email must be a valid email address</ng-template>
</val-errors>

```

On peut même faire encore mieux en définissant des messages d'erreur par défaut une bonne fois pour toutes :

```

<val-default-errors>
  <ng-template valError="required" let-label> {{ label || 'This field' }} is required
  </ng-template>
  <ng-template valError="email" let-label> {{ label || 'This field' }} must be a valid
  email address </ng-template>
  <ng-template valError="min" let-error="error" let-label>
    {{ label || 'This field' }} must be at least {{ error.min | number }}
  </ng-template>
  <!-- same for the other types of error -->
</val-default-errors>

```

Et ensuite simplement utiliser dans un formulaire :

```

<input id="email" formControlName="email" class="form-control" type="email" />
<val-errors controlName="email" label="The email"></val-errors>

```

Une intégration avec Bootstrap 4 et Material est fournie, pour avoir des messages d'erreur avec un style cohérent si tu utilises un de ces frameworks CSS. Fais un essai, on pense que tu ne le regretteras pas !

18.14. Aller plus loin : définir ses propres contrôles de formulaires avec `ControlValueAccessor`

HTML définit un nombre important de contrôles standards : texte, mot de passe, case à cocher, etc. Mais parfois ces contrôles standards ne conviennent pas.

Angular permet de définir des composants bien sûr, et il est en fait possible d'en faire des éléments de formulaires, c'est-à-dire de les lier à un *form control* en leur appliquant la directive `NgModel` ou `FormControlName`, et donc de les intégrer dans un formulaire.

Ce qui permet de lier un composant Angular à un contrôle de formulaire est une interface fournie par Angular : `ControlValueAccessor`.

Remplir le contrat de cette interface est en fait assez simple. Il faut

- accepter la valeur du `FormControl` et l'afficher dans le composant (`writeValue`) ;

- prévenir Angular que l'utilisateur a changé la valeur, en appelant une fonction callback (`registerOnChange`) ;
- prévenir Angular quand le contrôle doit être considéré comme *touched*, en appelant une fonction callback (`registerOnTouched`) ;
- honorer la demande d'activer ou de désactiver le contrôle (`setDisabledState`).

Nous allons illustrer tout cela en utilisant un composant *rating*. Ce composant permet de donner une note de 0 à 5 à un film, par exemple. Mais au lieu d'utiliser un champ de type `number` ou `range`, on voudrait que l'utilisateur clique simplement sur un bouton parmi 6 boutons affichés (qui, typiquement, seraient présentés sous forme d'étoiles, mais nous laisserons ça de côté dans l'exemple qui suit).

Voici le code d'un tel composant :

```
export class RatingComponent implements ControlValueAccessor {
  private onChange: (rating: number) => void = () => {
    // do nothing by default
  };

  onTouched: () => void = () => {
    // do nothing by default
  };

  value: number | null = null;
  disabled = false;
  pickableValues = [0, 1, 2, 3, 4, 5];

  registerOnChange(fn: any): void {
    this.onChange = fn;
  }

  registerOnTouched(fn: any): void {
    this.onTouched = fn;
  }

  setDisabledState(isDisabled: boolean): void {
    this.disabled = isDisabled;
  }

  writeValue(v: number | null | undefined): void {
    this.value = v ?? null;
  }

  setValueAndPropagateChanges(value: number) {
    this.value = value;
    // tell Angular the value has changed
    this.onChange(this.value);
  }
}
```

```
}
```

Et voici son template :

```
<button
  *ngFor="let pickableValue of pickableValues"
  [class.selected]="value != null && pickableValue <= value"
  type="button"
  (click)="setValueAndPropagateChanges(pickableValue)"
  [disabled]="disabled"
  (blur)="onTouched()"
>
  {{ pickableValue }}
</button>
```

Le code qui gère les deux fonctions callback et l'état d'activation est pratiquement identique dans chaque CVA ([ControlValueAccessor](#)).

La partie intéressante est la gestion de la valeur. Angular appelle `writeValue()` pour dire au composant quelle valeur il doit afficher. Ici, on stocke simplement cette valeur, et on l'utilise dans le template pour afficher les premiers boutons en jaune.

Lorsqu'on clique sur un bouton, on change la valeur du composant bien sûr, mais on doit aussi notifier Angular de ce changement de valeur. C'est ce qui lui permet de changer la valeur du [FormControl](#), de déclencher la validation, etc.

L'état `disabled` est honoré en désactivant les boutons.

Et finalement, on choisit de rendre le contrôle `touched` dès que l'un des boutons perd le focus. On fait ça en appelant la fonction de callback `onTouched`, fournie par Angular, lorsqu'un bouton lève un événement `blur`.

Dernière chose à faire : enregistrer ce composant comme l'un des *control value accessors* de l'application. Nous faisons cela en ajoutant un *provider* dans le décorateur du composant `rating`. Ne te prends pas trop la tête avec cette syntaxe : tu peux simplement la copier-coller chaque fois que tu définis un nouveau CVA.

```
providers: [
  {
    provide: NG_VALUE_ACCESSOR,
    useExisting: forwardRef(() => RatingComponent),
    multi: true
  }
]
```

Et voilà. Nous disposons à présent d'un joli composant qui peut être utilisé dans n'importe quel formulaire, en lui appliquant simplement l'une des directives des formulaires Angular :

```
<ns-rating formControlName="rating"></ns-rating>
```

18.15. 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 bidirectionnel.
- 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.

C'est peut-être l'approche la plus pragmatique : commence avec une approche orientée template et le binding bidirectionnel 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.



Essaye notre [quiz](#), et 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 !

Chapter 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.

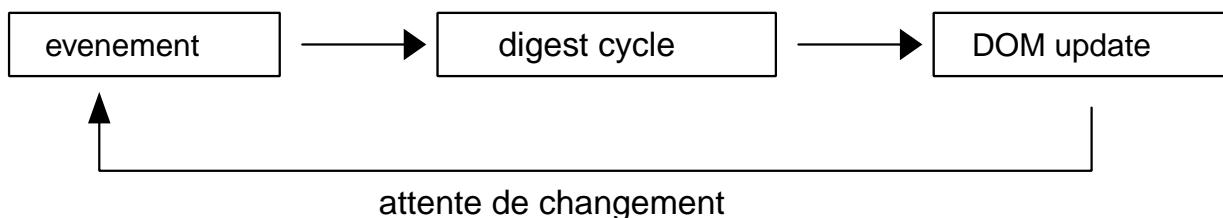
Le fait d'utiliser ses directives et services permet au framework d'être parfaitement informé du fait qu'un événement vient de se produire. C'est ça la première partie de la magie ! Et c'est cette première partie qui va déclencher la seconde : il faut maintenant que le framework analyse le changement 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
```

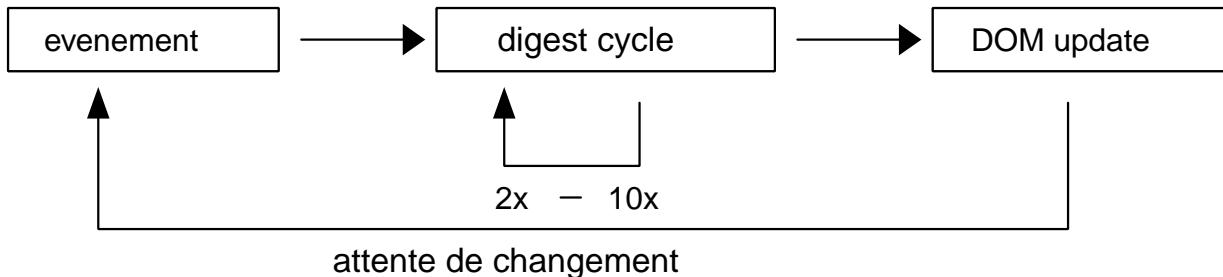
- "user.name" -> "Cédric"
- "user.password" -> "h"
- "passwordStrength" -> 3

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 language 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(playerOne, 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(playerOne, 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(playerOne, 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(playerOne, 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: ZoneDelegate, current: Zone, target: Zone, task: Function,
 applyThis: any) {
 // start the timer
 startTimer();
 return delegate.invoke(target, task, applyThis);
 },
 onHasTask(delegate: ZoneDelegate, current: Zone, target: Zone, hasTaskState:
 HasTaskState): void {
 delegate.hasTask(target, hasTaskState);
 if (!hasTaskState.macroTask) {
 // if the zone run is done, stop the timer
 stopTimer();
 }
 }
});
scoreZone.run(() => {
 const score = computeScore();
 updatePlayer(playerOne, 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 *monomorphique*, 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 *mégamorphe*. 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 mémoristique, 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 (monomorphe 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 !

# Chapter 20. Compilation Angular : JiT (Just in Time) et AoT (Ahead of Time)

## 20.1. Génération de Code

Dans le chapitre précédent, nous avons brièvement évoqué comment le framework *générait* une fonction de détection de changement pour chaque composant.

C'est une caractéristique intéressante d'Angular, que tu ne retrouveras pas dans d'autres frameworks : Angular, au démarrage de ton application, va *compiler* tes templates et *générer* du code dynamique pour chaque composant.

Le HTML que tu écris dans tes templates n'est jamais lu directement par le navigateur. À la place, Angular génère une *component definition* pour chaque composant qui représente exactement ce que tu as écrit dans ton template. Cette définition est générée dans un champ statique du composant.

Prenons par exemple notre fameux [PonyComponent](#). Le template est essentiellement une image avec une source dynamique, et une légende avec interpolation.

```
<figure>

 <figcaption>{{ ponyModel.name }}</figcaption>
</figure>
```

Quand Angular le compile, il commence par *parser* le template pour générer un arbre de syntaxe abstraite (*Abstract Syntax Tree*, AST). Un AST est un arbre représentant la structure du template, que les compilateurs utilisent traditionnellement pour représenter la structure d'un programme. C'est le résultat de la phase *d'analyse syntaxique* de la compilation. Cet AST sera ensuite utilisé pour générer le code JS dynamique, une *component definition* par composant. Une définition de composant contient différentes choses, et parmi elles, le template, représenté par une fonction.

```
elementStart(0, 'figure');
{
 element(1, 'img');
}
{
 elementStart(2, 'figcaption');
 text(3);
 elementEnd();
}
elementEnd();
```



Ce chapitre décrit le code généré par le compilateur/moteur de rendu introduit en Angular 8.0 et appelé "Ivy". Nous avons écrit [un article assez détaillé](#) sur Ivy si cela

t'intéresse. Ce moteur de rendu en est à sa troisième itération. En effet, il a déjà été réécrit en Angular 4.0. Ces itérations ont amené soit de meilleures performances soit des améliorations de la taille des bundles ou les deux, tout en gardant la même syntaxe de template, afin d'être rétro-compatible avec les applications existantes, et de nous permettre de migrer facilement. La plupart des développeurs n'ont même pas remarqué le changement de moteur introduit par Angular 4.0 par exemple.

Avec cette fonction, Angular est capable de créer le DOM correspondant à notre `PonyComponent` : la fonction crée les `HTMLElement` correspondants à chaque élément et les attache au DOM.

Mais comment gère-t-il la détection de changement ? La fonction définissant le template est en fait un peu plus longue :

```
template: (renderFlags: RenderFlags, component: PonyComponent) => {
 if (renderFlags & RenderFlags.Create) {
 elementStart(0, 'figure');
 {
 element(1, 'img');
 }
 {
 elementStart(2, 'figcaption');
 text(3);
 elementEnd();
 }
 elementEnd();
 }
 if (renderFlags & RenderFlags.Update) {
 advance(1);
 property('src', component.getPonyImageUrl());
 advance(2);
 textInterpolate(component.ponyModel.name);
 }
},
```

Cette fonction de template a deux parties :

- la création du composant comme expliqué ci-dessus ;
- la mise à jour du composant qui réalise essentiellement ce que tu aurais pu écrire à la main (c'est à dire que l'image a une propriété `src` qui doit refléter le résultat de la méthode `getPonyImageUrl()` de mon composant, et une `figcaption` dont le texte doit refléter le nom du poney).

Elle est invoquée par le framework à chaque fois qu'il a besoin de vérifier si un changement s'est produit (voir le chapitre précédent). Elle récupère la valeur de ses parties dynamiques (l'attribut `src` et son interpolation), puis appelle une fonction du framework avec l'indice de l'élément à mettre à jour pour le sélectionner, puis une autre avec la propriété et sa nouvelle valeur. Le framework compare alors la valeur stockée précédemment pour cette propriété, et si elle a changé,

met à jour la valeur stockée, et remplace la valeur de la propriété par cette nouvelle valeur.

Ce code généré est très rapide, et peut être optimisé par les moteurs JavaScript (voir la partie sur le monomorphisme et le *inline caching* du chapitre précédent). Cette génération de code prend du temps, et elle était auparavant faite dans le navigateur, au démarrage de l'application. C'est ce que l'on appelait la compilation Juste à Temps (*Just in Time*, JiT). Le compilateur Ivy est maintenant suffisamment rapide pour que cette compilation se fasse directement quand tu lances un `ng serve` ou un `ng build`. C'est ce que l'on appelle la compilation AoT, que l'on détaille ci-dessous.

Pour résumer, quand tu utilises la compilation JiT : tu écris du code TypeScript et des templates en HTML, tu compiles le TypeScript en JavaScript, et tu envoies du JS et du HTML à tes utilisateurs. Et à l'exécution, le HTML est aussi compilé en JS.

## 20.2. Compilation En Avance (*Ahead of Time*, AoT)

On peut donc générer ce code *avant* le démarrage de l'application, grâce à un compilateur fourni par l'équipe Angular ; le compilateur *Ahead of Time* (AoT). Tu peux l'invoquer manuellement dans ton projet (`ngc`), ou, si tu utilises la CLI (tu devrais), cela se fait par défaut quand tu exécutes `ng serve` ou `ng build` depuis Angular CLI v8.1.

Le build va alors utiliser le compilateur Angular pour compiler les templates en fichiers TypeScript. En fichiers TypeScript ? Oui, en fichier TypeScript, parce que cela permet de vérifier qu'on n'a pas fait d'erreur de syntaxe dans nos templates ! Le code TypeScript généré et notre code applicatif seront ensuite compilés (`ngc` va appeler le compilateur TypeScript immédiatement) et si tu as fait une faute dans un template, tu verras une erreur :

```
<figure>
<!-- wrong method name -->

<figcaption>{{ ponyModel.name }}</figcaption>
</figure>
```

Le compilateur Angular génère ensuite ce code TypeScript :

```
template: (component: PonyComponent, create: boolean) => {
 if (create) {
 // ...
 }
 advance(1);
 <!-- wrong method name in the generated code -->
 property('src', component.getPonyImageUr());
 advance(2);
 textInterpolate(component.ponyModel.name);
},
```

Et ce code lèvera une erreur de compilation TypeScript :

Property 'getPonyImageUr' does not exist on type 'PonyComponent'.

C'est plutôt cool, parce que cela signifie que tu peux vérifier tous tes templates avant même de démarrer l'application. Ainsi, tes refactorings futurs seront sans douleur : si tu renommes une méthode ou une propriété d'un composant, tu sauras immédiatement si un template doit aussi être mis à jour, parce qu'il cassera le build.

Cela signifie aussi que la compilation peut lever une erreur alors que ton code semblait fonctionner en mode JIT. Par exemple, si tu as un `@Input()` privé d'un composant utilisé dans un template, cela ne produira pas d'erreur en mode JIT (parce que le JavaScript n'a pas la notion de propriété privée), mais cela échouera en mode AoT (parce que le code TypeScript généré pour un autre composant doit accéder à cette propriété).

Compiler ton application avant de l'envoyer à tes utilisateurs va aussi grandement améliorer le temps de démarrage de ton application, puisque la compilation sera déjà faite !

Pour résumé, en mode AoT : tu écris toujours du TypeScript et du HTML, tu compiles le HTML en TypeScript, puis tout le code TypeScript en JavaScript, puis tu envoies tout le code JS à tes utilisateurs.

L'inconvénient sera la taille de tes modules JavaScript à envoyer à tes utilisateurs : le code généré est plus large que les templates non compilés. Cela est partiellement compensé par le fait que tu n'as plus besoin d'envoyer le compilateur Angular à tes utilisateurs, puisque les templates sont déjà compilés. Et le compilateur est un bon gros morceau de code, alors le gain n'est pas négligeable. Sur des applications de taille moyenne ou large, cela ne compense cependant pas, en général, l'augmentation de la taille par le code généré. Si tu veux avoir les bénéfices de la compilation AoT ET conserver des modules de taille raisonnable, il te faudra creuser les fonctionnalités de *lazy-loading* que nous avons expliquées dans le chapitre sur le Routeur.

# Chapter 21. 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 : [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.

## 21.1. 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 `HttpClient` 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 Ponyracer, 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`).

Depuis Angular v16, on peut même faire plus simple en utilisant l'opérateur `takeUntilDestroyed` fourni par le framework dans le package `@angular/core/rxjs-interop`.

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>
 `,
 standalone: true,
 imports: [AsyncPipe]
})
export class RaceComponent implements OnInit {
 race!: Observable<RaceModel>;

 constructor(private raceService: RaceService) {}

 ngOnInit(): void {
 this.race = this.raceService.get();
 }
}
```

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 `shareReplay` par exemple :

```
import { Component, OnInit } from '@angular/core';
import { Observable, shareReplay } from 'rxjs';
import { RaceService, RaceModel } from './race.service';
import { AsyncPipe } from '@angular/common';

@Component({
 selector: 'ns-race',
 template: `
```

```

<div>
 <h2>{{ (race | async)?.name }}</h2>
 <small>{{ (race | async)?.date }}</small>
</div>
',
standalone: true,
imports: [AsyncPipe]
})
export class RaceComponent implements OnInit {
 race!: Observable<RaceModel>;
 constructor(private raceService: RaceService) {}

 ngOnInit(): void {
 this.race = this.raceService.get().pipe(
 // will share the subscription between the subscribers
 shareReplay()
);
 }
}

```

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 `tap()` et/ou un `catchError()` 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';
import { RaceService, RaceModel } from './race.service';
import { AsyncPipe, NgIf } from '@angular/common';

@Component({
 selector: 'ns-race',
 template: `
 <div *ngIf="race | async as raceModel">
 <h2>{{ raceModel.name }}</h2>
 <small>{{ raceModel.date }}</small>
 </div>
 `,
 standalone: true,
 imports: [NgIf, AsyncPipe]
})
export class RaceComponent implements OnInit {
 race!: Observable<RaceModel>;

```

```

constructor(private raceService: RaceService) {}

ngOnInit(): void {
 this.race = this.raceService.get();
}
}

```

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 *ngIf="race | async as r" [raceModel]="r"></ns-race>
 </div>
 `,
 standalone: true,
 imports: [NgIf, AsyncPipe, RaceComponent]
})
export class RaceContainerComponent implements OnInit {
 race!: Observable<RaceModel>;
 error = false;

 constructor(private raceService: RaceService) {}

 ngOnInit(): void {
 this.race = this.raceService.get().pipe(
 // the 'catchError' operator allows to handle a potential error
 // it must return an observable (here an empty one).
 catchError(error => {
 this.error = true;
 console.error(error);
 return EMPTY;
 })
);
 }
}

@Component({
 selector: 'ns-race',
 template: `
 <div>

```

```

<h2>{{ raceModel.name }}</h2>
<small>{{ raceModel.date }}</small>
</div>
,
standalone: true
})
export class RaceComponent {
 @Input({ required: true }) 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.

## 21.2. 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, ReactiveFormsModule } from '@angular/forms';
import { PonyService, PonyModel } from './pony.service';
import { NgFor } from '@angular/common';

@Component({
 selector: 'ns-typeahead',
 template: `
 <div>
 <input [formControl]="input" />

 <li *ngFor="let pony of ponies">{{ pony.name }}

 </div>
 `
})

```

```

 </div>
 ,
 standalone: true,
 imports: [NgFor, ReactiveFormsModule]
 })
export class PonyTypeaheadComponent implements OnInit {
 input = new FormControl();
 ponies: Array<PonyModel> = [];

 constructor(private ponyService: PonyService) {}

 ngOnInit(): void {
 // 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
 .pipe(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 propose 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
 .pipe(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
 .pipe(
 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
 .pipe(
 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
 .pipe(
 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
 .pipe(
 filter(query => query.length >= 3),
 debounceTime(400),
 distinctUntilChanged(),
 switchMap(value => this.ponyService.search(value).pipe(catchError(error =>
 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.

## 21.3. 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 `new Observable(observer => {})`.

La fonction passée en paramètre au constructeur 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 = new Observable(observer => {
 observer.next(1);
 observer.next(2);
 observer.complete();
}

```

```
});
```

On pourrait alors s'abonner à un tel observable :

```
numbers.subscribe({
 next: number => console.log(number),
 error: error => console.log(error),
 complete: () => 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';

export class HelloService {
 get(): Observable<string> {
 return new Observable(observer => {
 const interval = setInterval(() => observer.next('hello'), 2000);
 });
 }
}
```

La fonction passée à `new Observable()` 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';

export class HelloService {
 get(): Observable<string> {
 return new Observable(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".

## 21.4. Gestion d'état avec un *store* (NgRx, NGXS, Akita et autres)

L'écosystème Angular a beaucoup de bibliothèques de gestion d'état, habituellement appelées *stores* : NgRx, NgRx component store, NGXS, Elf, et probablement d'autres.

Pendant nos formations, on nous demande souvent : "est-ce que je devrais en utiliser un ?" ou "lequel est le meilleur ?". On ne peut pas vraiment répondre à ces questions. Chaque projet a ses propres besoins. Chaque équipe de développeurs a ses préférences et ses priorités.

Nous avons un avis, cependant : ce ne sont pas des outils magiques, et ils ne vous éviteront pas d'avoir à apprendre RxJS. Ils sont tous basés sur RxJS, et tu as intérêt à maîtriser RxJS avant de les utiliser.

Pour *nos* différents projets, on a toujours trouvé que leurs bénéfices ne compensaient pas la complexité additionnelle et le *boilerplate* que leur utilisation implique. Mais tu peux avoir un avis différent. Fais tes propres expériences et décide par toi-même.

Attention à ce biais cependant : les stores paraissent intelligents ; le fait de les utiliser peut te faire paraître intelligent ; mais ça ne veut pas dire pour autant qu'ils sont une solution à promouvoir pour ton projet et ton équipe. Nous recevons aussi beaucoup de témoignages nous disant que le développeur principal a fait le choix d'utiliser un store, et que tous les autres développeurs de l'équipe pâtissent de ce choix.

## 21.5. Conclusion

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 !

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](#) 🦄
- [Live](#) 🦄
- [Astuces sur les Observables](#) 🦄
- [Booster un poney](#) 🦄
- [Score réactif](#) 🦄

# Chapter 22. Composants et directives avancés

## 22.1. Transformation d'inputs

Depuis Angular v16.1, il est possible de transformer un input avec l'option `transform` du décorateur `@Input()`.

Cela permet de transformer la valeur passée à l'input avant qu'elle soit assignée à la propriété. L'option `transform` attend une fonction qui recevra la valeur en entrée et renvoie la valeur transformée. Comme les cas les plus courants sont de transformer une chaîne de caractères en nombre ou en booléen, Angular fournit deux fonctions pour cela : `numberAttribute` et `booleanAttribute` dans `@angular/core`.

Voici comment on utilise `booleanAttribute` :

```
@Input({ transform: booleanAttribute }) disabled = false;
```

Cela va transformer la valeur passée en input en booléen, rendant possible ces utilisations :

```
<ns-button disabled></ns-button>
<ns-button disabled="true"></ns-button>
<!-- Before, only the following was properly working -->
<ns-button [disabled]="true"></ns-button></pre>
```

La fonction `numberAttribute` est similaire et transforme la chaîne de caractère en nombre :

```
@Input({ transform: numberAttribute }) value = 0;
```

Elle permet de définir une valeur par défaut, dans le cas où l'input ne serait pas un nombre valide (la valeur par défaut est `NaN`) :

```
@Input({ transform: (value: unknown) => numberAttribute(value, 42) }) value = 0;
```

On peut alors utiliser le composant comme ceci :

```
<ns-value value="42"></ns-value>
<ns-value value="not a number"></ns-value>
<!-- Before, only the following was properly working -->
<ns-value [value]="42"></ns-value></pre>
```

## 22.2. Requêtes de vue : ViewChild

Dans le chapitre sur les templates, on a mentionné les "variables locales", une chouette fonctionnalité qui permet d'obtenir dans le template une référence vers un élément du DOM. Par exemple, tu peux facilement donner le focus à un champ grâce à un bouton :

```
<input #myInput />
<button (click)="myInput.focus()">Focus</button>
```

On a aussi rencontré cette fonctionnalité dans le chapitre sur les Formulaires, quand on voulait obtenir une référence vers une directive précise :

```
<input name="login" [(ngModel)]="user.login" required #loginCtrl="ngModel" />
<div *ngIf="loginCtrl.dirty && loginCtrl.hasError('required')">The login field is
required</div>
```

Mais comment faire si on a besoin de ces mêmes références dans le code de notre composant, et pas seulement dans son template ? C'est là que les requêtes de vue (*view queries*) arrivent à la rescousse.

Par exemple, tu pourrais vouloir donner le focus à un champ dès que le composant est affiché. Pour ce faire, il nous faut obtenir une référence vers l'input, grâce au décorateur **ViewChild**.

```
@Component({
 selector: 'ns-login',
 template: '<input #loginInput name="login" [(ngModel)]="credentials.login" required
/>',
 standalone: true,
 imports: [FormsModule]
})
export class LoginComponent implements AfterViewInit {
 credentials = { login: '' };

 @ViewChild('loginInput') loginInput!: ElementRef<HTMLInputElement>;

 ngAfterViewInit(): void {
 this.loginInput.nativeElement.focus();
 }
}
```

On déclare un attribut nommé `loginInput`, que l'on décore avec **ViewChild**. Ce décorateur a besoin d'un sélecteur en paramètre : ici on utilise la variable locale déclarée dans notre template. Le décorateur indique ainsi au framework qu'il veut requérir la vue pour y trouver un élément avec ce nom de variable locale. L'attribut sera initialisé avec cet élément, de type `ElementRef<T>`. Ce type n'a qu'un seul attribut, `nativeElement` de type `T`, qui est une référence à l'élément du DOM sous-jacent.

Cet exemple montre aussi une utilisation intéressante du cycle de vie avec l'usage de la méthode `ngAfterViewInit`. Cette méthode est appelée dès que la vue est créée, ainsi nous sommes sûrs que l'élément qu'on attend sera effectivement présent. Si tu essaies de faire la même chose dans `ngOnInit`, cela ne fonctionnera pas. En fait, cela dépendait du template du composant dans les premières versions d'Angular. Une option `static` a été ajoutée en Angular 8.0 pour rendre ce timing de résolution explicite. Si tu passes `static: true` comme option au décorateur, la requête sera disponible dans `ngOnInit`. Mais il est conseillé de plutôt ne pas utiliser cette option et de récupérer la requête dans `ngAfterViewInit`. La [documentation officielle](#) explique les rares cas où tu pourrais avoir besoin de `static: true`.

Il y a aussi une autre méthode nommée `ngAfterViewChecked`, qui est appelée à chaque fois que la vue est vérifiée (après chaque détection de changement).

Si tu veux implémenter cette fonctionnalité de focus dans plusieurs composants différents, tu pourrais créer une directive pour cela, au lieu de dupliquer ce code dans chaque composant.

```
@Directive({
 selector: '[nsFocus]',
 standalone: true
})
export class FocusDirective implements AfterViewInit {
 ngAfterViewInit(): void {
 }
}
```

Cette directive ne fait rien pour le moment, mais elle serait utilisée comme suit dans un template :

```
<input nsFocus />
```

Cette directive doit accéder à son élément hôte pour lui donner le focus. C'est alors que `ElementRef` devient très intéressant, puisqu'il peut être injecté dans notre directive :

```
@Directive({
 selector: '[nsFocus]',
 standalone: true
})
export class FocusDirective implements AfterViewInit {
 constructor(private element: ElementRef<HTMLElement>) {}

 ngAfterViewInit(): void {
 this.element.nativeElement.focus();
 }
}
```

Et voilà : grâce à cette directive, nous donnerons le focus à son élément hôte !



Accéder directement au `nativeElement` comme dans ces exemples ne fonctionnera

pas si tu décides de faire tourner ton application sur une autre plateforme qu'un navigateur, comme dans des Web Workers, ou sur le serveur.

Retournons à notre décorateur `ViewChild` : il peut aussi accepter un *type* comme sélecteur.

Par exemple, dans le chapitre sur les formulaires, on a vu que pour soumettre un formulaire avec la façon pilotée par le template, on peut utiliser du binding bi-directionnel, ou on peut obtenir une référence vers le formulaire et passer sa valeur à la méthode submit :

```
<form (ngSubmit)="authenticate(form.value)" #form="ngForm">
 <!-- ... -->
</form>
```

Mais on pourrait aussi utiliser `ViewChild` :

```
@Component({
 selector: 'ns-login',
 template: `
 <form (ngSubmit)="authenticate()">
 <!-- ... -->
 </form>
 `,
 standalone: true,
 imports: [NgIf, FormsModule]
})
export class LoginFormComponent {
 @ViewChild(NgForm) credentialsForm!: NgForm;

 authenticate(): void {
 console.log(this.credentialsForm.value);
 }
}
```

Le truc cool avec `ViewChild` c'est que c'est une requête dynamique : il sera toujours à jour du template. Si l'élément requêté est détruit, l'attribut deviendra `undefined`.

Ce décorateur a aussi un frère jumeau nommé `ViewChildren`. À la différence de `ViewChild` qui permet d'obtenir la référence vers un seul élément satisfaisant le sélecteur (le premier s'il y en a plusieurs), `ViewChildren` permet d'obtenir une référence vers tous les éléments. Il retourne une `QueryList`, un type avec quelques attributs bien pratiques :

- `first` retourne le premier élément satisfaisant la requête ;
- `last` retourne le dernier élément satisfaisant la requête ;
- `length` retourne le nombre d'éléments satisfaisant la requête ;
- `changes` retourne un observable qui émet la nouvelle `QueryList` chaque fois qu'un élément satisfaisant la requête est ajouté, enlevé, ou déplacé.

Supposons que nous avons un `RaceComponent` affichant une collection de `PonyComponent`. On peut facilement être notifié chaque fois qu'un `PonyComponent` est ajouté ou supprimé :

```
@Component({
 selector: 'ns-race',
 templateUrl: './race.component.html',
 standalone: true,
 imports: [NgFor, PonyComponent]
})
export class RaceComponent implements AfterViewInit {
 @Input({ required: true }) raceModel!: RaceModel;

 @ViewChildren(PonyComponent) ponies!: QueryList<PonyComponent>;

 ngAfterViewInit(): void {
 this.ponies.changes
 // this will log how many ponies are displayed in the component
 .subscribe(newList => console.log(newList.length));
 }
}
```

`QueryList` propose aussi plein d'autres méthodes comme `toArray()`, `map()`, `filter()`, `find()`...

## 22.3. Contenu : `ng-content`

Un autre besoin récurrent des développeurs est la capacité de construire des composants graphiques dont le contenu serait dynamique.

Par exemple, supposons que tu veuilles construire [un composant "carte"](#) à l'aide du framework CSS Bootstrap 4. Le template d'une telle carte ressemblerait à :

```
<div class="card">
 <div class="card-body">
 <h4 class="card-title">Card title</h4>
 <p class="card-text">Some quick example text</p>
 </div>
</div>
```

Tu pourrais évidemment dupliquer ce code HTML chaque fois que tu en as besoin dans ton application. Mais à ce point de la lecture, tu penses déjà probablement à créer un composant. Il y a deux parties dynamiques dans cette carte (le titre et le contenu), alors tu arriverais probablement à quelque-chose comme :

```
@Component({
 selector: 'ns-card',
 template: `
 <div class="card">
```

```

<div class="card-body">
 <h4 class="card-title">{{ title }}</h4>
 <p class="card-text">{{ text }}</p>
</div>
</div>
',
standalone: true
})
export class CardComponent {
 @Input() title = '';
 @Input() text = '';
}

```

Et tu l'utiliserais comme cela :

```
<ns-card title="Card title" text="Some quick example text"></ns-card>
```

Cela fonctionne parfaitement. Mais en regardant plus précisément le besoin, tu te rends compte que le contenu de la carte peut aussi être du HTML complexe, et pas seulement du texte, ce que supporte Bootstrap !

Heureusement, Angular va encore une fois te sauver la mise : il est facile de "passer" du HTML à un composant enfant, grâce à `<ng-content>`. C'était ce qu'on appelait la *transclusion* en AngularJS 1.x.

`ng-content` est un tag spécial que tu peux utiliser dans tes templates pour inclure le HTML fourni par le composant parent :

```

<div class="card">
 <div class="card-body">
 <h4 class="card-title">{{ title }}</h4>
 <p class="card-text">
 <ng-content></ng-content>
 </p>
 </div>
</div>

```

Et tu peux désormais utiliser ce composant comme cela :

```
<ns-card title="Card title"> Some quick example text </ns-card>
```

Plus tard, tu te rends compte que le titre peut aussi contenir du HTML. Et bien sûr, il y a une solution pour passer plusieurs contenus au composant carte, en utilisant plusieurs `ng-content` avec un sélecteur.

```

<div class="card">
 <div class="card-body">

```

```

<h4 class="card-title">
 <ng-content select=".title"></ng-content>
</h4>
<p class="card-text">
 <ng-content select=".content"></ng-content>
</p>
</div>
</div>

```

et tu l'utiliseras comme cela :

```

<ns-card>
 <p class="title">Card title</p>
 <p class="content">Some quick example text</p>
</ns-card>

```

Cela produira le résultat suivant :

```

<div class="card">
 <div class="card-body">
 <h4 class="card-title">
 <p class="title">Card title</p>
 </h4>
 <p class="card-text">
 <p class="content">Some quick example text</p>
 </p>
 </div>
</div>

```

## 22.4. Requêtes de contenu : ContentChild

Quand tu utilises ces tags `ng-content`, le contenu projeté ne sera pas requêtable par `ViewChild` ou `ViewChildren`. Pour ces contenus, il te faut utiliser deux autres décorateurs : `ContentChild` et `ContentChildren`.

Disons que tu es en train de créer un autre composant graphique en Bootstrap 4, cette fois-ci un composant "onglet". D'après la documentation, le HTML doit ressembler à :

```

<ul class="nav nav-tabs">
 <li class="nav-item">
 Races

 <li class="nav-item">
 About


```

Mais on voudrait proposer un composant plus agréable d'utilisation à notre équipe, qui ressemblerait à :

```
<ns-tabs>
 <ns-tab title="Races"></ns-tab>
 <ns-tab title="About"></ns-tab>
</ns-tabs>
```

On a aussi besoin d'un composant **Tabs** englobant, qui doit découvrir combien de directives **ns-tab** sont embarquées dans le template du composant, doit itérer sur chacune d'elles, et générer le balisage adéquat.

Pour ce faire, commençons par créer une directive **TabDirective** :

```
@Directive({
 selector: 'ns-tab',
 standalone: true
})
export class TabDirective {
 @Input() title = '';
}
```

Cette directive ne fait pas grand-chose : elle n'a qu'une seule entrée pour le titre de l'onglet. Note qu'on utilise un élément comme sélecteur : **ns-tab**.

Maintenant il nous faut construire le **TabsComponent**:

```
@Component({
 selector: 'ns-tabs',
 template: `
 <ul class="nav nav-tabs">
 <li class="nav-item" *ngFor="let tab of tabs">
 {{ tab.title }}

 `,
 standalone: true,
 imports: [NgFor]
})
export class TabsComponent {
 @ContentChildren(TabDirective) tabs!: QueryList<TabDirective>;
}
```

Comme tu le vois, le template itère sur un tableau d'onglets **tabs** et génère un élément **li** pour chacun. Mais d'où provient ce tableau **tabs** ? Comment le composant peut-il connaître les deux directives **ns-tab** imbriquées dans le composant **ns-tabs** ? C'est ce que permet la directive **ContentChildren**.

Pour récupérer la liste des onglets, on doit utiliser `ContentChildren`, ici avec `TabDirective`. Cela nous fournit une liste d'onglets, que l'on peut parcourir avec un `NgFor`. Chaque élément dans cette liste est une `TabDirective`, on peut donc accéder à sa propriété `title`, et afficher le titre de l'onglet !

Note que si, pour n'importe quelle raison, tu avais un template comme celui-ci :

```
<ns-tabs>
 <div>
 <ns-tab title="Races"></ns-tab>
 </div>
 <ns-tabgroup>
 <ns-tab title="About"></ns-tab>
 </ns-tabgroup>
</ns-tabs>
```

alors le `QueryList` dans `TabComponent` ne contiendrait que la première `TabDirective`. `ContentChild` et `ContentChildren` ne recherchent en effet que les descendants directs, et stopperaient leur recherche sur le composant `ns-tabgroup`.

Si tu voulais que ton composant fonctionne malgré cela, il te faudrait ajouter une option à ton décorateur :

```
@Component({
 selector: 'ns-tabs',
 template: `
 <ul class="nav nav-tabs">
 <li class="nav-item" *ngFor="let tab of tabs">
 {{ tab.title }}

 `,
 standalone: true,
 imports: [NgFor]
})
export class TabsWithDescendantsComponent {
 @ContentChildren(TabDirective, { descendants: true })
 tabs!: QueryList<TabDirective>;
}
```

Et maintenant, il trouverait de nouveau toutes les `TabDirective` !

Le champ `tabs` est une `QueryList`, donc tu peux aussi t'abonner à ses changements, comme on l'a vu pour `ViewChildren`. Encore une fois, cette `QueryList` n'est pas accessible dans le constructeur du composant, ni-même dans `ngOnInit`. Pour être sûr que le contenu puisse être requêté (i.e. que la `QueryList` soit correctement peuplée) tu dois utiliser la méthode `ngAfterContentInit` du cycle de vie. Tu peux aussi utiliser `ngAfterContentChecked`, qui est appelée à chaque fois que la détection de changement est lancée.



Essaye notre exercice [Composants avancés](#) ! Tu construiras un composant avec `ng-content` !

## 22.5. Projection de contenu conditionnelle et contextuelle : `ng-template` et `ngTemplateOutlet`

`ng-content` ne convient pas lorsqu'on veut insérer du contenu HTML dynamique de manière conditionnelle, ou en boucle. Il ne permet pas non plus de fournir un contexte à ce contenu dynamique inséré.

Illustrons un tel cas par un exemple simple. On voudrait créer un composant *progress*. Ce composant accepte une valeur minimale, une valeur maximale et une valeur, calcule le pourcentage, et l'affiche dans une barre de progression.

Commençons par le squelette de ce composant :

```
export class ProgressComponent implements OnChanges {
 @Input() min = 0;
 @Input() max = 100;
 @Input() value = 0;
 percentage = 0;

 ngOnChanges(): void {
 this.percentage = (100 * (this.value - this.min)) / (this.max - this.min);
 }
}
```

Et son template :

```
<div class="progress">
 <div
 class="progress-bar"
 role="progressbar"
 [style.width.%]="percentage"
 [attr.aria-valuenow]="value"
 [attr.aria-valuemin]="min"
 [attr.aria-valuemax]="max"
 >
 </div>
</div>
```

Rien de nouveau jusqu'ici.

Mais on voudrait aussi afficher la valeur du pourcentage. Et l'utilisateur/utilisatrice du composant doit pouvoir customiser l'affichage du pourcentage si il ou elle le désire.

Donc, dans le template du composant *progress*, on veut ceci, où le `formatter` est quelque chose qui

est fourni en *input* du composant.

```
<div class="progress">
 <div
 class="progress-bar"
 role="progressbar"
 [style.width.%]="percentage"
 [attr.aria-valuenow]="value"
 [attr.aria-valuemin]="min"
 [attr.aria-valuemax]="max"
 >

 <ng-template #noFormatter>{{ percentage | number }}%</ng-template>
 </div>
</div>
```

Ce *formatter* n'est pas une valeur. Ce n'est pas non plus une fonction, parce que le pourcentage doit pouvoir être affiché en utilisant du code HTML, enrichi grâce à Angular.

Angular modélise un tel bout de HTML enrichi, qu'on peut passer en argument et insérer où et quand on le veut, par un `<ng-template>`. Son équivalent TypeScript est un objet de type `TemplateRef`.

Dans notre cas, la responsabilité du *formatter* est d'afficher un pourcentage. Donc il a besoin d'un contexte qui contient ce pourcentage. Ce contexte d'affichage constitue le type générique du `TemplateRef` que le composant accepte en *input*.

```
interface ProgressContext {
 percentage: number;
}

export class ProgressComponent implements OnChanges {
 // ...
 @Input() formatter?: TemplateRef<ProgressContext>;
 formatterContext: ProgressContext = { percentage: 0 };

 ngOnChanges(): void {
 this.percentage = (100 * (this.value - this.min)) / (this.max - this.min);
 this.formatterContext = { percentage: this.percentage };
 }
}
```

À présent, comment insérer ce `TemplateRef` dans le template de notre composant `progress` et lui fournir son contexte? C'est le rôle de la directive structurelle `*ngTemplateOutlet`. Voici comment on l'utilise :

```

 <ng-container *ngTemplateOutlet="formatter; context: formatterContext"></ng-
```

```
container>

<ng-template #noFormatter>{{ percentage | number }}%</ng-template>
```

Enfin, comment utilise-t-on ce composant *progress* et comment lui fournit-on un *formatter*? Simplement en utilisant la balise `<ng-template>`, en lui assignant une variable, et en passant cette variable comme valeur de l'*input formatter* au composant *progress*.

Dans le `<ng-template>`, on peut bien sûr accéder à toutes les propriétés du composant courant. Mais on peut en outre accéder aux propriétés du contexte, qui vient du composant *progress*, en utilisant cette syntaxe bizarre `let-p="percentage"`. `p` est simplement un alias pour l'élément du contexte nommé `percentage` (on pourrait choisir n'importe quel autre alias, comme par exemple `let-progress="percentage"`).

```
<!-- without formatter -->
<ns-progress [value]="75"></ns-progress>

<!-- with formatter -->
<ng-template #myFormatter let-p="percentage">Your progress: {{ p | number }}%</ng-template>
<ns-progress [value]="75" [formatter]="myFormatter"></ns-progress>
```

Si, dans l'interface du contexte, on choisit de nommer une propriété `$implicit` (au lieu de `percentage` par exemple), alors on peut utiliser simplement `let-p` au lieu de `let-p="percentage"` pour accéder à cette propriété.

C'est d'ailleurs ainsi que fonctionnent les directives structurelles en Angular. Toutes les directives que l'on utilise avec une `*`, comme `ngIf` ou `ngFor`, sont en réalité des instances de `<ng-template>`. Angular définit une *micro-syntaxe* pour les directives structurelles, qui permet d'appliquer plus succinctement une directive à un `<ng-template>`. Prenons l'exemple de `ngFor`. Quand on écrit un `ngFor` comme celui-ci :

```
<div *ngFor="let pony of ponies; i as index">
 {{ i }} - {{ pony }}
</div>
```

Cela revient au même qu'écrire le template suivant :

```
<ng-template ngFor [ngForOf]="ponies" let-pony let-i="index">
 <div>{{ i }} - {{ pony }}</div>
</ng-template>
```

C'est moins lisible, on est d'accord. C'est pour ça que l'on utilise toujours `*ngFor`. Mais cela montre qu'en réalité Angular utilise lui-même ce pattern à base de `ng-template` ! `ngFor` n'est rien d'autre qu'une directive s'appliquant sur un `ng-template`, avec un input `ngForOf` qui attend une collection, et un contexte contenant une propriété `$implicit` qui est l'élément courant, et d'autres propriétés

comme l'index.

Ce pattern est extrêmement puissant, et est d'une grande utilité pour concevoir des composants customisables.

## 22.6. À l'écoute de l'hôte : HostListener

Quand on écrit une directive, on a souvent envie d'interagir avec l'élément hôte.

Prenons un exemple simple : notre client voudrait rapidement vider le contenu de certains champs texte en double-cliquant dessus. C'est typiquement le genre de comportement que tu peux encapsuler dans une directive custom, que nous appellerons `InputClearDirective`. Son sélecteur sera un attribut, disons `nsInputClear`. Quand cet attribut sera ajouté sur un élément, on voudra écouter les double-clics sur cet élément hôte.



Cet exemple est certes simple, mais pas vraiment réaliste. Une application plus réaliste (mais aussi plus complexe) de cette fonctionnalité serait, par exemple, d'afficher une bulle d'aide (*tooltip*) quand un élément est survolé ou cliqué.

Créons cette directive :

```
@Directive({
 selector: '[nsInputClear]',
 standalone: true
})
export class InputClearDirective {
 constructor(private element: ElementRef<HTMLInputElement>) {}
}
```

On utilisera notre directive comme cela :

```
<input nsInputClear />
```

Il nous faut maintenant réagir à l'événement 'dblclick' sur notre élément hôte (ici, le `<input>`) pour y vider sa valeur.

C'est là que nous pouvons utiliser le décorateur `HostListener` ! Ce décorateur peut être ajouté sur une méthode d'une directive, pour indiquer au framework que cette méthode doit être appelée si l'événement passé en paramètre au décorateur est déclenché sur l'élément hôte.

Dans notre cas, nous pourrions écrire :

```
@Directive({
 selector: '[nsInputClear]',
 standalone: true
})
export class InputClearDirective {
```

```

constructor(private element: ElementRef<HTMLInputElement>) {}

@HostListener('dblclick')
clearContent(): void {
 this.element.nativeElement.value = '';
}
}

```

Désormais, chaque fois qu'un événement 'dblclick' est déclenché sur l'élément hôte, la directive va vider la valeur de l'input.

Note qu'il est aussi possible de se mettre à l'écoute d'événements globaux, comme `window:resize` par exemple :

```

@Directive({
 selector: '[nsWindowResize]',
 standalone: true
})
export class WindowResizeDirective {
 @HostListener('window:resize', ['$event'])
 resize(event: Event): void {
 const innerWidth = (event.target as Window).innerWidth;
 console.log(`The screen is being resized to ${innerWidth}`);
 }
}

```

Note également que tu peux accéder à l'événement dans la méthode, en indiquant `[$event]` comme second paramètre du décorateur.

## 22.7. Binding sur l'hôte : HostBinding

Un autre décorateur disponible pour la création de directives et composants avancés est `HostBinding`. Alors que `HostListener` permet d'interagir avec les événements sur l'élément hôte, `HostBinding` permet d'interagir avec ses propriétés.

Supposons qu'on veuille ajouter une classe CSS particulière (`is-required`) à un input s'il fait l'objet d'une erreur de validation donnée (`required`). Cette classe pourrait ajouter une jolie bordure autour du champ, ou un petit astérisque, peu importe. Cela ne validera en aucune façon la saisie, on se contentera de consommer le résultat de la validation native de formulaires en Angular, et d'utiliser ce résultat pour styler le champ.

C'est de nouveau un travail idéal pour une directive :

```

@Directive({
 selector: '[nsAddClassIfRequired]',
 standalone: true
})
export class AddClassIfRequiredDirective {

```

```
}
```

On pourra l'utiliser dans un formulaire Angular piloté par le code :

```
<input formControlName="firstName" nsAddClassIfRequired />
```

Ou dans un formulaire piloté par le template :

```
<input [(ngModel)]="user.name" required nsAddClassIfRequired>
```

Il nous faut ensuite, dans cette directive, une référence à l'état du champ. Angular valide automatiquement les champs, et ajoute l'erreur `required` si un champ est requis mais vide. Le puissant système d'injection de dépendances va encore une fois pouvoir nous aider ! On peut en effet demander à Angular d'injecter dans une directive une autre directive appliquée au même hôte, ou à un des ses descendants.

Comme nous voulons que notre directive fonctionne avec `FormControlName` ou `NgModel`, on pourrait demander à Angular de nous injecter les deux. Mais ces injections ne fonctionneraient pas, car seulement une seule des deux serait disponible (on utilise généralement soit l'une, soit l'autre, sur un input donné), et Angular lève une erreur si une dépendance ne peut être injectée. Il existe une astuce qui permet de demander à Angular de continuer même si une dépendance s'avère manquante : le décorateur `Optional`.

Ainsi, un tel code pourrait fonctionner :

```
constructor(
 @Optional() private formControl: FormControlName,
 @Optional() private ngModel: NgModel
) {}
```

Mais on peut faire mieux. En fait, ces deux directives héritent de la même classe de base : `NgControl`. Alors, au lieu d'injecter l'une ou l'autre, on peut lui demander de nous fournir le `NgControl` commun !

```
@Directive({
 selector: '[nsAddClassIfRequired]',
 standalone: true
)
export class AddClassIfRequiredDirective {
 constructor(private control: NgControl) {}
}
```

Maintenant qu'on a une référence au `NgControl`, il est facile de savoir si le champ présente l'erreur `required`, en utilisant sa méthode `hasError()`. La dernière étape est d'ajouter la classe `is-required` à l'élément hôte si c'est le cas. C'est alors que le décorateur `HostBinding` entre en scène ! Ce décorateur

permet de binder une propriété de l'élément hôte sur un attribut de notre directive. On l'utilise généralement comme ceci :

```
@HostBinding('value') innerValue;
```

Et cela mettrait à jour automatiquement la valeur de l'hôte lors de chaque modification de la propriété `innerValue` de la directive.

Dans notre cas, nous n'avons pas de propriété sur laquelle se binder. Mais on peut définir un accesseur qui retourne `true` ou `false`, selon si le champ est en erreur, et décorer cet accesseur avec `HostBinding` pour automatiquement ajouter ou enlever la classe `is-required` :

```
@Directive({
 selector: '[nsAddClassIfRequired]',
 standalone: true
})
export class AddClassIfRequiredDirective {
 constructor(private control: NgControl) {}

 @HostBinding('class.is-required')
 getisRequired(): boolean {
 return this.control.hasError('required');
 }
}
```

Ces quelques lignes de code sont vraiment puissantes : à chaque fois que cette directive sera utilisée dans un formulaire, Angular ajoutera ou enlèvera automatiquement notre classe spécifique selon la saisie utilisateur !

Cela peut être utilisé pour binder d'autres types de caractéristiques, et pas uniquement des classes CSS. Par exemple, certaines bibliothèques de composants l'utilisent pour ajouter les attributs d'accessibilité (`aria.xxx`) à l'élément hôte.

Note que notre directive utilise un sélecteur spécifique. Mais si tu veux l'appliquer absolument partout, tu peux simplement changer son sélecteur pour `input`, et tous les champs de ton application en bénéficieront.



Essaye notre exercice [Directives avancées](#) ! Tu construiras plusieurs directives qui interagissent entre elles et avec `ContentChild` et `HostBinding` ! Tu pourras ensuite voir comment des bibliothèques comme `ng-bootstrap` utilisent ces patterns dans l'exercice [Intégration avec une bibliothèque de composants](#) et comment ajouter des graphiques dans une application avec l'exercice [Charts](#).

# Chapter 23. Les modules Angular

Jusqu'à la version 14 d'Angular, les applications étaient organisées en modules Angular. Beaucoup d'applications existantes le sont probablement encore. Et même dans les applications récentes, qui utilisent des composants, *pipes* et directives standalone, le concept de module Angular, ou **NgModule**, est toujours présent : on peut, voire on doit, importer certains d'entre eux (**CommonModule**, **ReactiveFormsModule** par exemple) dans nos composants.

Peut-être lis-tu ce livre en vue de travailler sur un projet existant où les modules Angular sont toujours utilisés. Ce chapitre explique leurs principes, règles, et usages. Nous t'encourageons néanmoins à utiliser des composants standalone dans tes nouvelles applications, et même à migrer tes applications existantes. La CLI fournit des commandes permettant d'[automatiser la plus grande partie de cette migration](#).

## 23.1. Une unité de compilation

Dans la plus simple des applications Angular utilisant des modules, tu en trouveras au moins un, le module racine (*root*), conventionnellement appelé **AppModule**. Ce module ressemble typiquement à ceci :

```
@NgModule({
 declarations: [AppComponent, HomeComponent, AboutComponent, PonyComponent],
 imports: [BrowserModule, HttpClientModule, RouterModule.forRoot(APP_ROUTES)],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule {}
```

Que peut-on en dire ?

La propriété **declarations** déclare les composants, *pipes* et directives qui appartiennent à ce module. Ils ne sont pas standalone. S'ils l'étaient, alors ils seraient listés dans la propriété **imports**.

 La seule différence entre un composant (ou un *pipe*, ou une directive) standalone et un composant "normal", non standalone, est que les composants non-standalone n'ont ni **standalone: true**, ni **imports** dans leur décorateur. Le module auquel ils appartiennent définit ce qu'ils peuvent utiliser dans leur template.

Un module Angular définit une unité de compilation. Comme ils font partie du même module, tous ces composants peuvent s'utiliser l'un l'autre. Par exemple, les composants **HomeComponent** et **AboutComponent** peuvent utiliser **PonyComponent** dans leur template.

La propriété **imports** définit la liste des autres modules Angular, mais aussi des composants, *pipes* et directives standalones, qui sont importés dans ce module. Comme nous importons **RouterModule**, les composants peuvent utiliser les directives du module de routage, comme **routerLink**. Comme nous n'importons pas **ReactiveFormModule**, ils ne peuvent pas utiliser les directives de ce module, comme **formGroup**. Importer **HttpClientModule** permet d'importer le *provider* qu'il définit pour le service

`HttpClient`, ce qui signifie qu'il peut être injecté dans n'importe quel service de l'application. Importer `BrowserModule`, qui lui-même importe et exporte `CommonModule`, est ce qui rend possible l'utilisation des directives et *pipes* communs (`*ngIf`, `ngClass`, `date` etc.) dans les composants de ce module.

Le tableau `providers` définit les *providers*, de la même manière qu'ils sont définis dans l'appel de la fonction `bootstrapApplication()` des applications standalone. Comme la plupart des services utilisent `providedIn: 'root'`, il n'y a pas besoin de les lister ici. Mais cela peut être nécessaire si un service doit être fourni explicitement pour configurer l'application.

Enfin, `bootstrap` contient le composant racine (ou, beaucoup plus rarement, les composants racines) de l'application, qui doivent être *bootstrappés*, parce qu'ils sont à la racine de l'arbre des composants.

## 23.2. Composition de modules

Les applications plus grandes sont composées de modules supplémentaires, qu'on peut ranger dans deux catégories :

1. Les modules contenant des composants, directives, *pipes* réutilisables, et parfois aussi des *providers* de services. Ces modules sont destinés à être importés par les modules fonctionnels de l'application.
2. Les modules fonctionnels, souvent chargés à la demande, correspondant à un ensemble de routes de l'application.

Commençons avec les modules de composants réutilisables. Supposons que le `PonyComponent` doive aussi être utilisé par le `RacesComponent`, qui est lui-même déclaré dans un module fonctionnel `RacesModule`. Comme nous l'avons vu, pour que le `RacesComponent` ait accès au `PonyComponent`, nous devrions ajouter le `PonyComponent` aux `declarations` de `RacesModule`. Angular, cependant, ne te laissera pas faire ça : un composant doit être déclaré dans un et un seul module de l'application. Or il est déjà déclaré dans `AppModule` (qui en a aussi besoin pour pouvoir l'utiliser dans `HomeComponent`).

La solution consiste à extraire `PonyComponent` dans un module supplémentaire, parfois appelé module partagé (*shared module*) :

```
@NgModule({
 declarations: [PonyComponent],
 imports: [CommonModule],
 exports: [PonyComponent]
})
export class PonyModule {}
```

Ce module déclare le `PonyComponent` qu'il contient. Il importe `CommonModule` afin que `PonyComponent` puisse utiliser les directives communes. Et, le plus important, sa propriété `exports` contient aussi `PonyComponent`. C'est cet export qui permet de rendre disponible le composant aux autres modules qui importent le `PonyModule`. C'est ce que nous allons faire dans `AppModule` et `RacesModule` afin de pouvoir utiliser `PonyComponent`:

```

@NgModule({
 declarations: [AppComponent, HomeComponent, AboutComponent],
 imports: [BrowserModule, HttpClientModule, RouterModule.forRoot(APP_ROUTES),
 PonyModule],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule {}

```

La tentation est grande de mettre tous les composants réutilisables dans un seul module `SharedModule`. Au fur et à mesure que l'application grossit, cependant, ce module va devenir énorme. Une meilleure approche est de créer de nombreux minuscules modules partagés, afin que les autres modules n'importent que ce dont ils ont réellement besoin. De tels modules, qui n'exportent qu'un seul composant, sont connues sous le nom de *SCAMs (Single Component Angular Modules)*. Le *boilerplate* que tous ces modules représentent est l'une des raisons pour lesquelles Angular a décidé de supporter les composants standalone : tu peux voir un composant standalone comme étant à la fois un composant et le module qui le déclare et l'exporte.

### 23.3. Modules fonctionnels associés au routage

Les modules fonctionnels ne sont pas bien différents des modules de composants réutilisables. Les composants qu'ils déclarent, en revanche, ne sont pas destinés à être utilisés par les composants d'autres modules. Donc, ils sont déclarés, mais pas exportés.

```

@NgModule({
 declarations: [RacesComponent],
 imports: [CommonModule, PonyModule, RouterModule.forChild(RACES_ROUTES)]
})
export class RacesModule {}

```

Le module principal peut importer les modules fonctionnels. Mais alors, ils seront intégrés au *bundle* principal, et donc chargés immédiatement, avant le démarrage de l'application. La plupart du temps, on préférera donc charger les modules fonctionnels à la demande. La manière de procéder est similaire à celle que l'on a décrite précédemment. Simplement, au lieu de charger dynamiquement des routes, on chargera le module fonctionnel lui-même. Ainsi, dans les routes du module principal, on trouvera :

```

{
 path: 'races',
 loadChildren: () => import('./races/races.module').then(m => m.RacesModule)
}

```

Tu devrais maintenant être en mesure de comprendre comment une application est organisée en modules Angular. Les règles ne sont pas si difficiles à maîtriser, mais il faut bien reconnaître que les modules Angular n'apportent aucune fonctionnalité à l'application, et rendent les dépendances

des composants plus difficiles à trouver. C'est la raison pour laquelle, dans ce livre, nous avons préféré promouvoir l'utilisation des composants standalone.

# Chapter 24. 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.



## 24.1. La *locale*

On a déjà évoqué l'internationalisation auparavant, dans le chapitre sur les *pipes*. Quatre des *pipes* fournis par Angular concernent l'internationalisation. Ces *pipes* sont les *pipes* `number`, `percent`, `currency` et `date`. Jusqu'à Angular 5, ils utilisaient l'API JavaScript standard `Internationalization`, censée être fournie par le navigateur. Mais cette API n'étant pas toujours fournie par le navigateur et parfois source de bugs et d'incohérences entre navigateurs, ces *pipes* ont été complètement réécrits en Angular 5.0.

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' -->
 ,
 standalone: true,
 imports: [DecimalPipe]
)
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 les *providers* de l'application. Attention cependant : il faut également faire en sorte que les données nécessaires à la locale (les traduction des mois, les règles de formattage des nombres, etc.) soient livrées avec l'application. En effet, Angular ne contient par défaut que les données de [en-US](#).

```
import '@angular/common/locales/global/fr';
```

Voici un exemple, qui montre l'effet que cela produit sur notre composant :

```

bootstrapApplication(AppComponent, {
 providers: [
 { provide: LOCALE_ID, useValue: 'fr-FR' }
]
});

@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' -->
 ,
 standalone: true,
 imports: [DecimalPipe]
)
export class CustomLocaleComponent {
 constructor(@Inject(LOCALE_ID) public locale: string) {}
}

```

Tous les *pipes* qui gèrent l'internationalisation peuvent aussi prendre comme dernier paramètre la locale. Il est donc possible de la changer dynamiquement si besoin :

```

@Component({
 selector: 'ns-locale',
 template: `
 <p>The locale is {{ locale }}</p>
 <!-- will display 'en-US' -->
 `,
 standalone: true,
 imports: [DecimalPipe]
})
class DefaultLocaleOverriddenComponent {
 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.

## 24.2. La devise par défaut

Le pipe `currency` permet de préciser la devise à utiliser en fournissant un code ISO comme `USD`, `EUR`... Si tu n'en fournis pas, le pipe utilise `USD` par défaut. C'est donc un peu pénible de devoir préciser la devise à chaque fois si ton application n'affiche que des euros par exemple.

Angular 9 a introduit la possibilité de configurer la devise par défaut globalement en utilisant le token `DEFAULT_CURRENCY_CODE`.

```

bootstrapApplication(AppComponent, {
 providers: [
 { provide: DEFAULT_CURRENCY_CODE, useValue: 'EUR' },
 { provide: LOCALE_ID, useValue: 'fr-FR' }
]
});

```

Tu peux ensuite utiliser `currency` sans devoir préciser `EUR` dans un composant. Et tu peux bien sûr récupérer la devise par défaut via l'injection de dépendances :

```

@Component({
 selector: 'ns-currency',
 template: `
 <p>The currency is {{ currency }}</p>
 <!-- will display 'EUR' -->

 <p>{{ 1234.56 | currency }}</p>
 <!-- will display '1 234,56 €' -->
 `
})

```

```

 ,
 standalone: true,
 imports: [CurrencyPipe]
})
class DefaultCurrencyOverriddenComponent {
 constructor(@Inject(DEFAULT_CURRENCY_CODE) public currency: string) {}
}

```

## 24.3. 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 pas 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'elle ait été réécrite en Angular 9.0. 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 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.

À partir de la version 9.0, l'équipe Angular a introduit un nouveau package [@angular/localize](#). Si tu utilises la CLI, tu as juste à lancer `ng add @angular/localize` et elle se chargera de l'ajouter au bon endroit pour toi.

Ce nouveau package introduit une fonction globale appelée `$localize`, qui est utilisée sous le capot pour la localisation, et que l'on pourra utiliser dans le futur pour traduire des messages dans notre code.

## 24.4. 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. 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 au [guide de l'internationalisation](#) 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.

Examinons ces différentes étapes en détail.

### 24.4.1. 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 `extract-i18n` fournie par Angular CLI :

```
ng extract-i18n --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-US" datatype="plaintext" original="ng2.template">
 <body>
 <trans-unit id="7627914200888412251" datatype="html">
 <source>Welcome to Ponyracer</source>
 <context-group purpose="location">
 <context context-type="sourcefile">src/app/app.component.html</context>
 <context context-type="linenumber">2</context>
 </context-group>
 </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" target-
language="fr">
 <body>
 <trans-unit id="7627914200888412251" 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="6888363845978110110" datatype="html">
 <source> Welcome to Ponyracer.
</source>
 <context-group purpose="location">
 <context context-type="sourcefile">src/app/app.component.html</context>
 <context context-type="linenumber">5,7</context>
 </context-group>
</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>
 <context-group purpose="location">
 <context context-type="sourcefile">src/app/app.component.html</context>
 <context context-type="linenumber">10</context>
 </context-group>
</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>
```

```

<context-group purpose="location">
 <context context-type="sourcefile">src/app/app.component.html</context>
 <context context-type="linenumber">13</context>
</context-group>
<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" equiv-text="{{ user.firstName }}"/> <x id="INTERPOLATION_1" equiv-text="{{ user.lastName }}"/>!</source>
 <context-group purpose="location">
 <context context-type="sourcefile">src/app/app.component.html</context>
 <context context-type="linenumber">16</context>
 </context-group>
</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 sont clairement indiquées ;
- en revanche, si ce que ces deux expressions représentent est clair pour les développeurs, cela peut ne pas l'être pour les traducteurs. 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 (à part les indications des interpolations), 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>
 <context-group purpose="location">
 <context context-type="sourcefile">src/app/app.component.html</context>
 <context context-type="linenumber">21</context>
 </context-group>
</trans-unit>
<trans-unit id="home.ponyImage.title" datatype="html">
 <source>Ponies are cool, aren't they?</source>
 <context-group purpose="location">
 <context context-type="sourcefile">src/app/app.component.html</context>
 <context context-type="linenumber">23</context>
 </context-group>
</trans-unit>
```

Enfin, comment traduire le dernier morceau de texte ? Il n'y a aucun élément qui puisse porter l'attribut **i18n**. Une solution est possible : utiliser un élément **ng-container**, qui ne fera pas partie du HTML généré par le template :

```
<ng-container i18n="@@home.startMessage">Let's start playing.</ng-container>
```

#### 24.4.2. 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.



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" equiv-text="{{ user.firstName }}"/> <x id="INTERPOLATION_1" equiv-text="{{ user.lastName }}"/>!</source>
 <target>Bienvenue dans Ponyracer <x id="INTERPOLATION" equiv-text="{{ user.firstName }}"/> <x id="INTERPOLATION_1" equiv-text="{{ user.lastName }}"/>&nbsp!</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` en utilisant la propriété `i18n` du fichier `angular.json` :

Avant la version 9, nous devions passer de nombreuses options aux commandes `ng serve` et `ng build` pour lancer ou construire l'application en français.

Depuis la version 9, nous devons préciser les locales que nous voulons supporter, avec leurs messages associés, en utilisant la propriété `i18n` du fichier `angular.json` :

```
"prefix": "ns",
"i18n": {
 "locales": {
 "fr": "src/locale/messages.fr.xlf"
 }
},
```

Tu peux ajouter autant de locales que tu veux, chacune liée à son fichier de traduction.

Tu peux ensuite lancer :

```
ng build --localize
```

et la CLI construit l'application; et génère autant de versions que de locales définies. Dans notre cas, elle génère une version pour la locale par défaut `en-US`, dans le répertoire `dist/i18n/en-US` (si le projet s'appelle `i18n`), et une autre version pour la locale `fr` dans le dossier `dist/i18n/fr`.

Si tu veux lancer `ng serve`, tu dois spécifier la locale à utiliser, car on ne peut servir qu'une seule version.

Le plus simple est de définir une `configuration` de build pour cette locale dans le fichier `angular.json`, exactement comme la configuration `production`, mais en beaucoup plus simple :

```
"fr": {
 "localize": ["fr"]
}
```

et ajouter une configuration pour `serve`:

```
"fr": {
 "browserTarget": "i18n:build:fr"
}
```

et ensuite lancer :

```
ng serve --configuration=fr
```

ou construire l'application pour une locale spécifique avec :

```
ng build --configuration=production,fr
```

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 une seule fois, puis l'outil de localization générera trois versions (une pour chaque langue) de l'application (c'est très rapide). Ensuite, il faudra 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.

## 24.5. 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. Depuis Angular 9.0, on peut utiliser `$localize` pour faire cela. `$localize` est une fonction de tag qui peut être appliquée à une chaîne de caractères template (revois le chapitre [ECMAScript 2015](#) si tu veux te rafraîchir la mémoire).

```
status = $localize`PENDING`;
```

Ensuite tu peux construire ton application avec la CLI, et les appels à `$localize` sont remplacés par leurs traductions !

Tu peux aussi définir un ID avec la même syntaxe que dans les templates, et avoir des parties dynamiques dans la chaîne de caractères :

```
greetings = $localize`:@@home.greetings:Welcome ${this.user.firstName}!`;
```

`ng extract-i18n` supporte l'extraction des appels à `$localize` dans le code depuis la CLI v10.1, et génère :

```
<trans-unit id="home.greetings" datatype="html">
 <source>Welcome <x id="PH" equiv-text="this.user.firstName"/>!</source>
 <context-group purpose="location">
```

```
<context context-type="sourcefile">src/app/app.component.ts</context>
<context context-type="linenumber">17</context>
</context-group>
</trans-unit>
```

Tu peux ensuite le traduire comme d'habitude (avec **PH** le placeholder pour la partie dynamique) :

```
<trans-unit id="home.greetings" datatype="html">
 <source>Welcome <x id="PH" equiv-text="{{ this.user.firstName }}"/>!</source>
 <target>Bonjour <x id="PH" equiv-text="{{ this.user.firstName }}"/>!&nbsp!</target>
</trans-unit>
```

## 24.6. 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 génère deux éléments **trans-unit** : l'un pour le message lui-même, et l'autre pour l'expression contenue dans le message.

```
<trans-unit id="home.racesPlanned" datatype="html">
```

```

<source> Hello, <x id="ICU" equiv-text="{racesPlanned, plural, =0 {no race is
planned} =1 {only one race is planned} other
{{ racesPlanned }} races are planned}" xid="2482273160942454889"/>.
</source>
<context-group purpose="location">
<context context-type="sourcefile">src/app/app.component.html</context>
<context context-type="linenumber">31,34</context>
</context-group>
</trans-unit>
<trans-unit id="5232470321856793506" datatype="html">
<source>{VAR_PLURAL, plural, =0 {no race is planned} =1 {only one race is
planned} other {<x id="INTERPOLATION"/> races are planned}</source>
<context-group purpose="location">
<context context-type="sourcefile">src/app/app.component.html</context>
<context context-type="linenumber">32,33</context>
</context-group>
</trans-unit>

```

Malheureusement, la deuxième `trans-unit` a un identifiant auto-généré, et la syntaxe de pluralisation doit être comprise et respectée par le traducteur. 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" equiv-text="{racesPlanned, plural, =0 {...} =1 {...} other
...}"/>..
</source>
<target>Bonjour, <x id="ICU" equiv-text="{racesPlanned, plural, =0 {...} =1
...} other {...}"/>.</target>
</trans-unit>
<trans-unit id="5232470321856793506" datatype="html">
<source>{VAR_PLURAL, plural, =0 {no race is planned} =1 {only one race is
planned} other {<x id="INTERPOLATION" equiv-text="{{ racesPlanned }}"/> races are
planned}</source>
<target>{VAR_PLURAL, plural, =0 {aucune course n'est planifiée} =1 {seule une
course est planifiée} other {<x id="INTERPOLATION" equiv-text="{{ racesPlanned }}"/>
courses sont planifiées}</target>
</trans-unit>

```

## 24.7. 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.

# Chapter 25. Performances



Attention à ne pas optimiser prématurément. Il faut toujours mesurer avant et après. Fais attention également aux benchmarks que l'on trouve sur les internets : il est assez facile de leur faire dire ce que l'on veut.

Le terme performance peut vouloir dire beaucoup de choses : vitesse, utilisation du CPU (et donc consommation de la batterie), pression sur la mémoire... Tout n'est pas important pour tout le monde : on peut avoir des besoins différents selon que l'on développe un site mobile, une plate-forme e-commerce, ou une application de gestion classique.

Les performances peuvent être découpées en plusieurs catégories, qui, une fois de plus, peuvent ne pas être aussi importantes les unes que les autres pour toi : premier chargement, rechargement, et performances durant la vie de l'application, ce que l'on appelle le *runtime*.

Le premier chargement se produit lorsqu'un utilisateur charge l'application pour la première fois. Le rechargement se produit quand il ou elle revient consulter l'application. Les performances au *runtime* concernent quant à elles ce qui se passe quand l'application est utilisée. Certains des conseils suivants sont très génériques, et pourraient être appliqués à n'importe quel framework. Nous en parlons quand même parce que l'on pense que c'est intéressant et important à savoir. Et aussi parce que, lorsque l'on parle de performances, le framework est parfois le goulot d'étranglement, mais très (très) souvent, ce n'est pas le cas.

## 25.1. Premier chargement

Quand tu charges une application web moderne dans ton navigateur, un certain nombre de choses se passent. Premièrement, le fichier `index.html` est chargé et interprété par le navigateur. Puis les scripts JS et les autres ressources référencées sont téléchargés. Quand une des ressources demandées est reçue, le navigateur l'analyse, et l'exécute si c'est un fichier JS.

### 25.1.1. Taille des ressources

La première astuce est relativement évidente : fais attention à la taille des ressources chargées !

La phase de chargement des ressources dépend du nombre de ressources que tu veux charger. Beaucoup de ressources veut dire que ce sera long. Des ressources volumineuses également. Particulièrement si le réseau n'est pas très bon, ce qui arrive plus souvent que tu ne le penses : tu testes peut-être l'application sur une bonne connection fibre, mais certains utilisateurs sont peut-être au milieu de nulle part, contraints d'utiliser une 3G lente. Voici ce que tu peux faire pour les aider.

### 25.1.2. Faire un beau paquet : le *bundle*

Quand on écrit une application Angular, on a des imports un peu partout, et notre code est réparti dans des dizaines, centaines, voire milliers de fichiers. Mais on ne veut pas que nos utilisateurs chargent des milliers de fichiers ! Donc avant de livrer notre application, nous allons faire ce que l'on appelle un "bundle" : un seul fichier contenant tous les fichiers JavaScript.

C'est le travail de [Webpack](#) de prendre tous nos fichiers JavaScript (et CSS, et les fichiers de template HTML) et de construire ces *bundles*. Ce n'est pas un outil simple à maîtriser pour être honnête, mais Angular CLI fait un assez bon travail pour cacher sa complexité. Si tu n'utilises pas la CLI, tu peux quand même construire ton application avec Webpack, ou tu peux choisir un autre outil qui produira des résultats peut-être même encore meilleurs (comme [Rollup](#) par exemple). Mais attention, cela demande pas mal d'expertise (et de travail) pour ne pas tout casser, et souvent juste pour gagner quelques kilobytes supplémentaires. J'aurais tendance à te recommander de rester avec la CLI. L'équipe travaillant dessus fait un très bon travail pour être à jour avec les dernières versions d'Angular, de TypeScript et de Webpack.

Plus que cela, ils ont également construit quelques outils pour diminuer la taille des *bundles*. Par exemple, ils ont écrit un plugin qui passe à travers tout le code JavaScript généré, et ajoute des commentaires spécifiques pour aider [Terser](#) à supprimer le code mort.

### 25.1.3. *Tree-shaking*

Webpack (ou l'outil que tu utilises) commence par le point d'entrée de ton application (le fichier `main.ts` que la CLI a généré pour toi, et que tu n'as probablement jamais touché), et résout ensuite l'arbre des imports, puis produit le *bundle*. C'est assez cool parce que le bundle ne contient ainsi que les fichiers de ta base de code et des bibliothèques tierces que tu as importés. Le reste n'est pas embarqué. Donc même si tu as des dépendances dans ton `package.json` que tu n'utilises plus (et donc que tu n'importe plus), elles ne seront pas dans le *bundle* final.

Webpack est même un petit peu plus malin que ça. Si tu as un fichier `models` qui exporte deux classes, disons `PonyModel` et `RaceModel`, et que tu n'importe que `PonyModel` dans le reste de l'application, mais jamais `RaceModel`, alors Webpack mettra seulement `PonyModel` dans le *bundle* final, et pas `RaceModel`. Ce procédé est appelé **tree-shaking**. Et tous les frameworks et bibliothèques dans l'éco-système font de leur mieux pour être "tree-shakable". En théorie, cela veut dire que ton bundle final ne contient que ce qui est réellement nécessaire. En pratique, Webpack (et les autres) sont sensiblement plus conservateurs, et ne sont pas capables de comprendre deux ou trois choses à l'heure actuelle. Par exemple, si tu as une classe `Pony` avec deux méthodes `eat` et `run`, mais que tu n'utilise que `run`, le code de la méthode `eat` sera quand même dans le *bundle* final. Donc ce n'est pas parfait, mais cela fait déjà un bon boulot.

Quelques techniques peuvent être utilisées avec Angular spécifiquement pour avoir un meilleur *tree-shaking*. Premièrement, n'ajoute aux imports de ton NgModule que les modules que tu utilises vraiment. Parfois tu essaies une bibliothèque offrant un super composant, et tu ajoutes le NgModule de cette bibliothèque aux imports de ton NgModule. Puis tu ne l'utilises plus, mais tu as peut-être oublié l'import du module, et tu ne l'as pas supprimé... Mauvaise nouvelle : ce module et la bibliothèque seront dans le bundle final (en tout cas à l'heure actuelle, peut-être que ce sera mieux à l'avenir). Donc pense bien à n'importer que ce que tu utilises vraiment.

Une autre astuce est d'utiliser `providedIn` pour tes services. Si tu déclares un service dans les providers de ton NgModule, il va toujours atterrir dans le bundle, que tu l'utilises en fait vraiment ou non, simplement parce qu'il est importé et référencé dans le module. Alors que si tu ne l'enregistres pas dans les providers de ton NgModule, mais utilises `providedIn: 'root'` à la place, et que tu n'utilises pas ce service, alors il ne finira pas dans le bundle.

## 25.1.4. Minification et élimination de code mort

Quand le *bundle* est construit, le code est généralement minifié et le code mort est éliminé. Cela veut dire que toutes les variables, noms de méthodes, noms de classes... sont renommés pour n'utiliser qu'un ou deux caractères à travers toute la base de code. C'est un peu effrayant au premier abord et laisse penser que cela pourrait casser des choses, mais Terser fait un bon travail. Terser va aussi éliminer le code mort qu'il va trouver. Il fait de son mieux, mais comme je le disais plus haut, l'équipe de la CLI a construit un outil qui prépare le code avec des commentaires spéciaux sur du code non-nécessaire, pour que Terser puisse le supprimer en toute sécurité.

## 25.1.5. Autres types de ressources

Alors que les sections précédentes concernent spécifiquement le JavaScript, une application contient aussi souvent d'autres types de ressources, comme des styles, des images, des polices de caractères... Tu devrais avoir le même souci d'efficacité à leurs propos, et faire de ton mieux pour les garder à une taille raisonnable. Appliquer toutes sortes de techniques dingues pour optimiser la taille des *bundles* JS, mais charger des images de plusieurs Mo serait contre-productif sur le temps de chargement et la bande passante ! Comme ce n'est pas réellement le sujet de cet ebook, je ne vais pas creuser ce sujet, mais cette ressource en ligne de Addy Osmani à propos de l'optimisation des images est vraiment excellente : [Essential Image Optimization](#).

Angular v14.2 a ajouté une directive `NgOptimizedImage` qui force un certain nombre de ces bonnes pratiques.

```

```

## 25.1.6. Compression

Tous les navigateurs modernes supportent la compression des ressources, et indiquent au serveur, via un header de la requête HTTP, que le serveur peut compresser la ressource demandée avant de l'envoyer. Cela veut dire que tu peux servir une version compressée à tes utilisateurs, et le navigateur se chargera de la décompresser avant de l'interpréter. C'est vraiment obligatoire car cela sauve une tonne de bande passante et de temps de chargement !

Tous les serveurs du marché ont une option qui permet d'activer la compression des ressources. Généralement le premier utilisateur qui demandera la ressource payera le coût de la compression à la volée, et ensuite les suivants recevront la ressource compressée directement.

L'algorithme de compression le plus courant est GZIP, mais d'autres comme `Brotli` sont également populaires.

## 25.1.7. Chargement fainéant : le *lazy-loading*

Parfois, en dépit de nos efforts pour garder le *bundle* JS le plus petit possible, on finit avec un gros fichier parce que notre application contient plusieurs dizaines de composants, et utilise plusieurs bibliothèques tierces. Et non seulement ce gros *bundle* va augmenter le temps nécessaire pour télécharger ce JavaScript, mais il va aussi augmenter le temps nécessaire à l'analyse du code JavaScript qu'il contient.

Une solution courante à ce problème est d'utiliser un chargement différé ou *lazy-loading*. Cela veut dire que plutôt que d'avoir un seul gros *bundle*, tu découpes ton application en plusieurs parties et demande à Webpack de construire plusieurs *bundles*.

La bonne nouvelle est qu'Angular (son routeur et son système de modules en particulier) rend cette tâche assez facile à accomplir. L'autre bonne nouvelle est que la CLI sait comment lire la configuration de ton routeur pour construire automatiquement plusieurs *bundles*. Tu peux lire le [chapitre](#) sur le routeur pour en apprendre plus à ce sujet.

Le *lazy-loading* peut améliorer radicalement le temps de chargement : tu peux rendre le premier bundle chargé très petit, avec seulement le code nécessaire pour afficher la page d'accueil, et laisser Angular charger le reste à la demande quand ton utilisateur navigue vers une autre partie de l'application. Tu peux aussi utiliser des stratégies de pré-chargement pour indiquer à Angular de démarrer le chargement des autres *bundles* dès qu'il est inactif.

Note que ce *lazy-loading* ajoute tout de même un peu de complexité à ton application (et quelques pièges avec l'injection de dépendances), donc je te conseille plutôt de ne l'utiliser que si cela a réellement du sens pour ton projet.

### 25.1.8. Rendu côté serveur

J'aimerais commencer par préciser que cette technique n'est sans doute destinée qu'à 0.0001% d'entre vous. Le rendu côté serveur (*server side rendering* ou *universal rendering*) est une technique qui consiste à pré-rendre l'application sur le serveur avant de la servir à tes utilisateurs. Avec cette technique, quand une utilisatrice demande [/dashboard](#), elle recevra une version pré-rendue du tableau de bord, plutôt que de recevoir [index.html](#) et ensuite laisser le routeur faire son travail une fois qu'Angular aura fini de démarrer.

Cela peut donner de grandes améliorations dans les temps de démarrage perçus. Angular offre un package [@angular/universal](#) qui permet de lancer une application dans un environnement différent du navigateur comme par exemple un serveur. Tu peux pré-rendre les pages et les servir à tes utilisateurs. La page va alors s'afficher très vite, puis Angular prendra le relais et fera son travail habituel.

C'est aussi un gros avantage si tu veux que ton site soit découvrable par les moteurs de recherche qui n'exécutent pas le JavaScript, puisque tu peux leur servir une page pré-rendue, plutôt qu'une page blanche.

C'est également une façon de permettre aux réseaux sociaux comme Twitter et Facebook d'afficher un aperçu de ton site. Ces réseaux sociaux vont essayer de faire une capture d'écran d'une URL partagée, mais comme ils n'exécutent pas le JavaScript, ils ne verront rien de ta page générée dynamiquement, à moins que tu ne leur serves une page pré-rendue par le serveur. Donc si tu veux être sûr que l'aperçu soit parfait, par exemple si ton application est un site médiatique, ou un site e-commerce, tu auras peut-être besoin de rendu côté serveur.

La mauvaise nouvelle est que ce n'est pas aussi simple que d'ajouter le package [@angular/universal](#). Ton application devra suivre les bonnes pratiques (pas de manipulation directe du DOM par exemple, car le serveur n'a pas de vrai DOM à manipuler). Il faudra ensuite mettre en place le serveur et réfléchir à la stratégie à adopter. Veux-tu pré-rendre toutes les pages ou seulement

quelques-unes ? Veux-tu pré-rendre toute la page, avec les données déjà récupérées et les vérifications de sécurité nécessaires, ou juste des parties critiques de la page ? Veux-tu pré-rendre les pages au build, ou les pré-rendre à la demande et les mettre en cache ? Veux-tu faire cela pour tous les profils d'utilisateur et toutes les langues supportées ou juste pour certaines combinaisons ? Toutes ces questions dépendent du type d'application que tu construis, et l'effort peut beaucoup varier pour atteindre ces objectifs.

Donc, encore une fois, je te conseille de te pencher sur cette technique uniquement si c'est critique pour ton application, et non pas seulement pour la hype...

## 25.2. Rechargement

Une fois que les utilisateurs ont ouvert l'application une première fois, il est possible d'accélérer les chargements suivants.

### 25.2.1. Cache

Tu devrais toujours mettre en cache les ressources statiques de ton application (images, styles, *bundles JS*...). Tu peux facilement le faire en configurant ton serveur et en utilisant les headers HTTP `Cache-Control` et `ETag`. Tous les serveurs du marché permettent de faire cela, ou tu peux aussi utiliser un CDN. Si tu actives le cache, la prochaine fois que tes utilisateurs chargeront l'application, leur navigateur n'aura même pas à faire de requête HTTP pour récupérer ces ressources car il les aura déjà en cache !

Mais un cache est toujours un peu traître : tu peux avoir besoin de dire au navigateur "Hé ! J'ai déployé une nouvelle version en production, merci de récupérer les nouvelles ressources !".

La façon la plus simple de faire ceci est d'avoir un nom différent pour la ressource mise à jour. Cela veut dire qu'au lieu de déployer une ressource `main.js`, tu déploies `main.xxxx.js` où `xxxx` est un identifiant unique. Cette technique est appelée du *cache busting*, ce qui pourrait se traduire approximativement par "faire péter le cache". Et, encore une fois, la CLI est là pour toi : en mode production, elle va automatiquement nommer les ressources avec un *hash* unique, calculé grâce au contenu du fichier. Elle met également automatiquement à jour les sources des scripts dans `index.html` pour refléter ces noms uniques, les sources des polices de caractères, les sources des fichiers de style, etc.

Si tu utilises la CLI, tu peux donc déployer en toute sécurité une nouvelle version et tout mettre en cache, à l'exception du `index.html` (puisque il contiendra les liens vers les nouvelles ressources déployées) !

### 25.2.2. Service Worker

Si tu veux aller un cran plus loin, tu peux utiliser les *service workers*.

Les *Service Workers* sont une API supportée par la plupart des navigateurs modernes et, pour simplifier, ils se comportent comme un proxy dans le navigateur. Tu peux enregistrer un *service worker* dans ton application et toutes les requêtes GET passeront par lui, te permettant de décider si tu veux vraiment charger la ressource demandée, ou si tu veux la servir depuis un cache. Tu peux alors tout mettre en cache, même le `index.html`, ce qui garantit un temps de démarrage

extrêmement rapide (pas de requête au serveur).

Tu te demandes peut-être comment une nouvelle version peut être déployée si tout est en cache, mais tout est prévu : le *service worker* va servir la ressource depuis le cache puis ira vérifier si une nouvelle version est disponible. Il peut alors forcer un rafraîchissement, ou demander à l'utilisateur s'il veut la nouvelle version tout de suite ou plus tard.

Cela permet même de fonctionner en mode déconnecté, puisque tout est en cache !

Angular propose un package dédié appelé `@angular/service-worker`. C'est un petit package, mais rempli de fonctionnalités sympathiques. Savais-tu que tu peux l'ajouter à ton application Angular CLI, et qu'en ajoutant un *flag* (`"serviceWorker": true` dans `angular.json`), la CLI va automatiquement générer tout le nécessaire pour mettre en cache tes ressources statiques par défaut ? Et l'application téléchargera alors seulement ce qui a changé quand tu déploieras une nouvelle version, permettant des démarrages ultra-rapides !

Cela peut aller encore plus loin, en permettant de mettre en cache les ressources externes (comme les polices de caractères ou les icônes chargés depuis un CDN...), de faire des redirections, et même de mettre en cache des données dynamiques (comme les appels à ton API), avec différentes stratégies possibles (toujours télécharger des données fraîches, ou toujours servir depuis le cache pour gagner en vitesse...). Ce package fournit également un module appelé `ServiceWorkerModule` que tu peux utiliser dans ton application pour réagir aux événements `push` et aux notifications !

C'est assez facile à mettre en œuvre, et offre un gain rapide pour le temps de recharge. C'est aussi l'une des étapes nécessaires pour construire une PWA (*Progressive Web App*, une application web progressive), et atteindre le score parfait de 100% sur [Lighthouse](#). Cela peut donc être utile de s'y intéresser.

## 25.3. Profilage

Maintenant que nous avons évoqué le premier chargement et les rechargements de l'application, nous pouvons parler des performances à l'exécution. Si tu détectes un problème de performances, avant d'appliquer quelque changement que ce soit, l'idéal est d'avoir des chiffres, et donc de profiler l'application pour mesurer ses performances.

Les navigateurs modernes viennent avec des outils de développement sophistiqués. Chrome, en particulier, permet de faire un "enregistrement" de l'application, et de fournir une analyse assez détaillée de son comportement.

Il permet en outre de simuler un environnement spécifique, comme par exemple un processeur lent, ou un réseau médiocre. Une fois l'enregistrement terminé, dans ces conditions spécifiques, on peut plonger dans l'arbre des appels de fonctions, et savoir le temps passé dans chacune d'elles.

Mais l'outillage ne se limite pas au navigateur. Angular lui-même vient avec un outil précieux : `ng.profiler`. Il est assez méconnu, mais peut s'avérer utile pour mesurer le temps perdu par la détection de changements dans la page courante.

Après avoir utilisé les outils fournis par le navigateur, et le *profiler* fourni par Angular, il est plus simple de savoir ce qu'il faut optimiser. Et surtout, il est possible de relancer une mesure après

avoir appliqué l'optimisation, pour avoir une idée précise de son efficacité.

Dans le fichier `main.ts`, remplace le code qui démarre l'application par le code suivant :

```
platformBrowserDynamic()
 .bootstrapModule(AppModule)
 .then(moduleRef => {
 const applicationRef = moduleRef.injector.get(ApplicationRef);
 const componentRef = applicationRef.components[0];
 // allows to run `ng.profiler.timeChangeDetection();`
 enableDebugTools(componentRef);
 })
 .catch(err => console.log(err));
```

Ensuite, navigue vers la page que tu désires profiler, et exécute la ligne de code suivante dans la console du navigateur :

```
> ng.profiler.timeChangeDetection()
ran 489 change detection cycles
1.02 ms per check
```

Comme tu peux le voir, cette commande indique le nombre de cycles de détection de changements exécutés (au moins 5 cycles ou au moins pendant un temps de 500ms), et le temps moyen par cycle. Cette métrique est très pratique, parce que la détection de changement est au cœur des performances de l'application, et la plupart des astuces que nous allons présenter agissent directement sur le système de détection de changements. Grâce à cette commande, il est donc très simple de comparer les performances avant et après l'application des optimisations.

Tu peux aussi enregistrer le profil CPU pendant ces détections de changements pour pouvoir les analyser avec `ng.profiler.timeChangeDetection({ record: true })`.

L'équipe Angular recommande un temps par cycle en dessous de 3ms, pour laisser assez de temps à la logique applicative, à la mise à jour de l'interface et au rendu du navigateur de s'exécuter dans une fenêtre de 16ms (en supposant que l'on vise 60FPS).

Découvrons ces différentes astuces !

## 25.4. Performances à l'exécution

La magie d'Angular réside dans son mécanisme de détection de changements : le framework détecte les changements de l'état de l'application et modifie le DOM en conséquence. C'est le principal travail d'Angular, et donc, en règle générale, le meilleur moyen de rendre l'application performante est de limiter le nombre et la taille des détections de changements et la quantité de modifications à apporter au DOM (créations, modifications, suppressions).

Pour être honnête, la plupart des applications Angular n'auront aucun problème de performance, sans avoir à appliquer une quelconque optimisation. Lis le [chapitre](#) sur la magie d'Angular pour

savoir pourquoi Angular est performant, et en particulier plus performant qu'AngularJS.

Malheureusement, certains d'entre nous vont devoir recréer Excel dans le navigateur pour leur entreprise, ou ne vont pas échapper à un arbre devant absolument afficher 10 000 clients, ou vont être confrontés à une quelconque autre demande déraisonnable pour une application web. C'est là que, quel que soit le framework utilisé, les choses deviennent plus compliquées. Ces fonctionnalités nécessitent un nombre énorme d'éléments dans le DOM, et de composants Angular dans l'arbre de composants. Les trucs et astuces qui suivent peuvent être efficaces dans ce genre de situation. Et quelques-uns sont **absolument** nécessaires, comme le premier d'entre eux.

### 25.4.1. Mode production

En mode *développement* (qui est le mode par défaut), Angular exécute la détection de changements deux fois de suite à chaque changement. Cela permet de s'assurer que le code applicatif ne fait pas de choses étranges et problématiques comme mettre à jour des données sans respecter le principe de flux unidirectionnel des données. Si tu violes les règles, cette double détection de changements permet à Angular de t'en avertir pendant le développement en levant une exception, qui t'obligerait à corriger le code fautif. Mais bien sûr, cette double détection a un coût, qu'il serait dommage de subir dans l'application déployée en production.

Pour déployer notre application, il nous faut donc basculer en mode *production*. Cela va désactiver la double détection et rendre aussi le DOM généré plus "léger" (suppression des attributs utilisés pour le débogage).

Comme d'habitude, la CLI gère cette optimisation automatiquement : il suffit de construire l'application avec la configuration `production`, ce qui est le cas par défaut de `ng build`. La CLI utilise pour cela une variable interne appelée `ngDevMode` que le framework utilise pour savoir s'il doit faire du travail supplémentaire ou pas.

### 25.4.2. `trackBy` dans `*ngFor`

Cette astuce peut réellement améliorer les performances de `*ngFor` : ajouter un `trackBy`. Pour que tu comprennes le principe de `trackBy`, laisse-moi d'abord t'expliquer comment les frameworks JavaScript modernes (du moins, les plus populaires d'entre eux) gèrent les collections. Suppose qu'on veuille afficher une liste de 3 poneys. Le code devrait ressembler à ça :

```

 <li *ngFor="let pony of ponies">{{ pony.name }}

```

Si tu ajoutes un poney, Angular va insérer un nouvel élément DOM à la bonne position. Si tu modifies le nom de l'un des poneys, Angular modifiera le texte de l'élément `li` correspondant.

Mais comment fait-il cela ? En créant un lien entre les éléments du DOM et les objets contenus dans le tableau. Angular utilise une table de correspondances qui ressemble à ça :

```
node li 1 -> pony #e435 // { id: 3, color: blue }
```

```
node li 2 -> pony #8fa4 // { id: 4, color: red }
```

Ça fonctionne très bien. Si tu remplaces un objet par un autre dans le tableau, Angular détruira l'élément du DOM correspondant et en créera un nouveau.

```
node li 1 (recreated) -> pony #c1ea // { id: 1, color: green }
node li 2 -> pony #8fa4 // { id: 4, color: red }
```

Si tous les éléments du tableau sont remplacés par de nouveaux objets, tous les éléments `li` du DOM seront détruits et recréés. Tout cela ne pose pas de problème en général, excepté lorsque le but est simplement de rafraîchir la liste avec un contenu qui sera pratiquement identique. Dans ce cas, il serait bien plus efficace de garder les éléments du DOM existants, et de faire les quelques modifications nécessaires. Le problème, c'est que si tu vas rechercher cette même liste de poneys sur le serveur pour la rafraîchir, même si l'état des objets sera sensiblement identique à l'état précédent, les objets, eux (i.e. leur référence) seront tous différents.

La solution est d'aider Angular à maintenir cette table de correspondances sans tout détruire à chaque rafraîchissement, en lui disant d'identifier (*track*) les objets non plus par leur référence, mais plutôt par une propriété fonctionnelle qui permet de les identifier, typiquement un ID de base de données.

C'est ce que permet de faire `trackBy`, qui attend une méthode en argument :

```

 <li *ngFor="let pony of ponies trackBy: ponyById">{{ pony.name }}

```

avec la méthode définie dans le composant :

```
ponyById(index: number, pony: PonyModel): number {
 return pony.id;
}
```

Comme tu peux le voir, cette méthode reçoit l'indice courant dans le tableau et l'entité correspondante, ce qui permet de retourner n'importe quelle propriété de l'objet, ou de simplement identifier les objets par leur indice dans le tableau (mais ce n'est pas recommandé).

Dans l'exemple ci-dessus, Angular ne créera un nouvel élément dans le DOM que si l'identifiant du poney change. Sur des longues listes dont le contenu change peu, cela permet d'économiser des tas de suppressions et de créations d'éléments. Cette optimisation est plutôt simple à mettre en oeuvre et n'a pas d'inconvénient, donc il ne faut pas hésiter à l'utiliser. Il ne s'agit d'ailleurs pas que d'une optimisation. `trackBy` est aussi nécessaire en cas d'utilisation d'animations. Si le style d'un élément est supposé s'animer (en transitionnant progressivement de la valeur précédente à la nouvelle valeur), et que la liste de poneys est remplacée par une nouvelle liste à chaque rafraîchissement, alors `trackBy` est indispensable. Sans lui, l'animation ne va jamais se déclencher, puisque le style des éléments ne change jamais : l'élément lui-même est remplacé par un autre.

### 25.4.3. Stratégies de détection de changements

Lorsque nous avons expliqué comment Angular détecte les changements, nous avons montré qu'il partait du composant à la racine de l'arbre des composants, puis appliquait la détection (*check*) sur ses fils, puis ses petit-fils, et ainsi de suite jusqu'à ce que tous les composants soient vérifiés. Ensuite seulement, tous les changements nécessaires sont appliqués au DOM en une passe.

Mais est-ce vraiment une bonne idée de vérifier **chaque** composant à **chaque** changement ? Bien souvent, ce n'est pas nécessaire.

Angular offre une stratégie alternative de détection de changements : **OnPush**. Et chaque composant peut opter pour cette stratégie.

Si un composant utilise **OnPush**, il ne sera vérifié que dans 2 cas :

- lorsque l'un des *inputs* du composant a changé (pour être plus précis, lorsque la **référence** d'une *input* a changé) ;
- lorsqu'un gestionnaire d'événement du composant a été déclenché.

Cette stratégie est donc adéquate lorsque le template du composant dépend uniquement de ses *inputs*, et elle peut améliorer substantiellement les performances d'une application qui affiche un grand nombre de composants à l'écran ! Mais une fois encore, sois très prudent avant d'appliquer cette optimisation : si les préconditions s'avèrent n'être pas respectées, tu vas perdre tes cheveux à te demander pourquoi le composant, ou l'un de ses descendants, n'est pas rafraîchi après un changement.

Prenons un exemple simple pour illustrer le problème.

Imagine que nous ayons 3 composants. Un composant très simple **ImageComponent** :

```
@Component({
 selector: 'ns-img',
 template: `
 <p>{{ check() }}</p>

 `,
 standalone: true
})
export class ImageComponent {
 @Input() src = '';

 check(): void {
 console.log('image component view checked');
 }
}
```

utilisé par **PonyComponent** :

```
@Component({
```

```

 selector: 'ns-pony',
 template: `
 <p>{{ check() }}</p>
 <ns-img [src]="getPonyImageUrl()"></ns-img>
 `,
 standalone: true,
 imports: [ImageComponent]
)
export class PonyComponent {
 @Input({ required: true }) ponyModel!: PonyModel;

 check(): void {
 console.log('pony component view checked');
 }

 getPonyImageUrl(): string {
 return `images/pony-${this.ponyModel.color}-running.gif`;
 }
}

```

lui-même utilisé par `RaceComponent` :

```

@Component({
 selector: 'ns-race',
 template: `
 <h2>Race</h2>
 <p>{{ check() }}</p>
 <div *ngFor="let pony of ponies">
 <ns-pony [ponyModel]="pony"></ns-pony>
 </div>
 <button (click)="changeColor()">Change color</button>
 `,
 standalone: true,
 imports: [NgFor, PonyComponent]
)
export class RaceComponent {
 ponies: Array<PonyModel> = [
 { id: 1, color: 'green' },
 { id: 2, color: 'orange' }
];
 colors: Array<string> = ['green', 'orange', 'blue'];

 check(): void {
 console.log('race component view checked');
 }

 changeColor(): void {
 this.ponies[0].color = this.randomColor();
 }
}

```

```
}
```

Le `RaceComponent` affiche deux poneys, et l'utilisateur peut changer la couleur du premier poney en cliquant sur le bouton `Change color`.

Avec la stratégie de détection par défaut, chaque événement déclenche la détection de changements sur les 3 composants.

Nous avons ajouté une méthode `check()` dans chacun des composants, appelée depuis leur template. Elle nous permet de savoir si le composant est vérifié ou pas. Et en effet, en exécutant notre exemple, on voit s'afficher dans la console :

```
pony component view checked
image component view checked
pony component view checked
image component view checked
race component view checked
```

(on peut même voir ces traces deux fois à chaque clic, parce que l'application est en mode *développement*. Vois la section précédente à propos du mode *production*).

## OnPush

Dans notre exemple, nous savons que la plupart de ces détections sont inutiles : si le poney ne change pas, le template du `PonyComponent` n'a pas besoin d'être modifié. Il en va de même pour `ImageComponent`: si la valeur de l'`input src` ne change pas, il n'y a pas besoin de recalculer la valeur de l'URL. Adoptons donc la stratégie `OnPush` pour ces deux composants, en ajoutant une propriété `changeDetection` dans leur décorateur `@Component` :

```
@Component({
 selector: 'ns-img',
 template: `
 <p>{{ check() }}</p>

 `,
 standalone: true,
 changeDetection: ChangeDetectionStrategy.OnPush
})
export class ImageComponent {
 @Input() src = '';

 check(): void {
 console.log('image component view checked');
 }
}
```

```
@Component({
```

```

 selector: 'ns-pony',
 template: `
 <p>{{ check() }}</p>
 <ns-img [src]="getPonyImageUrl()"></ns-img>
 `,
 standalone: true,
 imports: [ImageComponent],
 changeDetection: ChangeDetectionStrategy.OnPush
 })
export class PonyComponent {
 @Input({ required: true }) ponyModel!: PonyModel;

 check(): void {
 console.log('pony component view checked');
 }

 getPonyImageUrl(): string {
 return `images/pony-${this.ponyModel.color}-running.gif`;
 }
}

```

Cette fois, lorsqu'on clique sur le bouton, la seule ligne qui apparaît dans la console est :

race component view checked

Parfait. La stratégie est donc bien efficace puisque les composants ne sont plus vérifiés par Angular \o/.

## OnPush et le piège de la mutabilité

Mais... il ya comme un problème : la couleur du poney ne change plus !

J'ai choisi cet exemple à dessein : **OnPush** est efficace, mais il s'agit de bien comprendre son fonctionnement et de ne pas ajouter **OnPush** ici et là.

Pourquoi ne fonctionne-t-il pas comme nous l'espérions ?

Relis attentivement le code de **RaceComponent**, et en particulier de sa méthode **changeColor** :

```

changeColor(): void {
 this.ponies[0].color = this.randomColor();
}

```

Cette méthode **modifie l'état** du poney dans le tableau **ponies**, et ce poney est *l'input* de **PonyComponent**. Maintenant que nous avons adopté la stratégie **OnPush** pour ce composant, Angular ne déclenchera la détection de changements sur ce composant que si la **référence** de *l'input pony* est modifiée. Lorsqu'on change l'état du poney, l'objet lui-même reste identique, et Angular considère donc qu'aucune détection de changements n'est nécessaire.

Cette stratégie est-elle donc inutile ? Non, pas du tout, mais elle requiert beaucoup d'attention.

Le moyen de résoudre notre problème est tout simple. Plutôt que de changer l'état du poney dans `changeColor`, il suffit de remplacer le poney par un autre poney :

```
changeColor(): void {
 const pony = this.ponies[0];
 // create a new pony with the old attributes and the new color
 this.ponies[0] = { ...pony, color: this.randomColor() };
}
```

Une fois ce changement effectué, l'application est plus rapide et correcte. L'utilisateur clique sur le bouton ; la méthode `changeColor` crée un nouvel objet avec les mêmes propriétés excepté la nouvelle couleur. Comme il s'agit d'un nouvel objet passé en `input` au `PonyComponent`, Angular déclenche la détection de changements sur ce composant. Comme la valeur de l'`input src` de `ImageComponent` s'en trouve elle aussi modifiée, l'image affiche désormais le poney avec la bonne couleur. Et, bien sûr, si un autre changement se produit sur `RaceComponent` sans affecter la valeur des poneys, les instances de `PonyComponent` échapperont à une détection de changements inutile.

Comme tu le vois, il est assez facile de tomber dans ce piège lorsqu'on utilise `OnPush`. Sois donc prudent ! (Les tests unitaires sont tes amis).

Une façon d'éviter ce piège est d'utiliser une librairie qui oblige à utiliser des objets immuables. `Immutable.js` (de Facebook) est l'une de ces librairies. Je ne l'ai jamais utilisée professionnellement, donc je ne peux pas savoir si c'est un choix judicieux pour une application Angular ou pas, mais elle est fréquemment mentionnée sur les internets.

Il y a un dernier sujet dont il faut parler lorsqu'on évoque `OnPush` : les observables.

### OnPush, les Observables et le *pipe* `async`

Supposons à présent que nous utilisions uniquement notre désormais fameux composant `PonyComponent`. Il souscrit à un observable exposé par le service `ColorService`, qui émet une nouvelle couleur toutes les secondes. On s'attend évidemment à voir l'image affichée par le composant changer, elle aussi, toutes les secondes. Le développeur de ce composant se dit qu'utiliser `OnPush` sur ce composant ne peut pas faire de mal. Qu'en penses-tu ?

```
@Component({
 selector: 'ns-pony',
 template: `
 <p>New color every 1s</p>

 `,
 standalone: true,
 changeDetection: ChangeDetectionStrategy.OnPush
})
export class PonyComponent {
 color = 'green';
```

```

constructor(colorService: ColorService) {
 colorService
 .get()
 .pipe(takeUntilDestroyed())
 .subscribe(color => (this.color = color));
}
}

```

Malheureusement, cette fois encore, ça ne fonctionne pas. Avec la stratégie `OnPush`, Angular ne rafraîchit un composant que si l'un de ses *inputs* a changé (ici, pas d'*input*), ou si un gestionnaire d'événement a été appelé sur le composant (il n'y en a pas non plus). Donc la propriété `color` est bien modifiée toutes les secondes, mais la vue n'est jamais rafraîchie...

Ce problème peut être résolu en utilisant le *pipe* `async`. Nous en avons parlé dans le [chapitre sur les pipes](#) mais peut-être as-tu besoin d'un petit rappel.

Le *pipe* `async` permet de s'abonner à un Observable ou une Promise. Utilisons-le dans `PonyComponent`:

```

@Component({
 selector: 'ns-observable-on-push-with-async',
 template: '<img *ngIf="color | async as c" [src]="'pony-' + c + '.gif'" [alt]="c"
/>',
 standalone: true,
 imports: [NgIf, AsyncPipe],
 changeDetection: ChangeDetectionStrategy.OnPush
})
export class PonyComponent {
 color: Observable<string>;

 constructor(colorService: ColorService) {
 this.color = colorService.get();
 }
}

```

Cette fois, le composant fonctionne ! Le *pipe* `async` déclenche la détection de changements chaque fois qu'il reçoit un nouvel événement de l'Observable auquel il a souscrit. Comme tu peux le voir, on stocke le résultat de `async`, pour pouvoir l'utiliser dans le template, grâce à `as`. L'utilisation de `async` nous permet aussi d'éviter d'avoir à s'abonner à l'observable dans le code du composant, et d'avoir à se désabonner lorsque le composant est détruit : `async` fait tout ça pour nous.

Attention cependant : utiliser `async` plusieurs fois dans le template, sur le même observable, ajoute plusieurs souscripteurs à l'observable. Si l'observable est un observable HTTP, cela déclenchera l'envoi de plusieurs requêtes HTTP. L'utilisation du pattern "smart/dumb" permet d'éviter ce problème, et rend l'utilisation de `OnPush` et de `async` plus simple et plus sûre. Consulte le [chapitre Observables : utilisation avancée](#) pour plus d'informations sur ce pattern.

#### 25.4.4. ChangeDetectorRef

Il existe encore d'autres moyens d'influer sur la détection de changements. Ceux-ci concernent des cas d'utilisation plus avancés, mais qui sait ? Ils pourraient t'être utiles un jour ou l'autre.

Considérons la situation suivante : un observable émet des données très fréquemment. Dans mon exemple, il s'agit d'une horloge qui émet toutes les 10 millisecondes.

```
@Component({
 selector: 'ns-clock',
 template: `
 <h2>Clock</h2>
 <p>{{ getTime() }}</p>
 <button (click)="start()">Start</button>
 `,
 standalone: true
})
export class ClockComponent {
 time = 0;
 interval$ = interval(10).pipe(
 take(1001), // 0, 1, ..., 1000
 map(time => time * 10),
 takeUntilDestroyed()
);

 start(): void {
 this.interval$.subscribe(time => (this.time = time));
 }

 getTime(): number {
 return this.time;
 }
}
```

Le composant utilise la stratégie de détection de changements `Default`. Donc chaque fois que l'observable émet une nouvelle valeur, la détection de changements est déclenchée, la vue est rafraîchie, et la valeur affichée est modifiée.

Mais est-ce vraiment utile de modifier la valeur affichée cent fois par seconde ? Nos yeux ne peuvent pas suivre un tel rythme, qui sollicite le navigateur pour rien. Et ce n'est pas tout. Il n'y a pas que ce composant qui est vérifié cent fois par seconde. Toute l'application l'est !

Peut-être dans ce cas pourrait-on rafraîchir le temps affiché chaque seconde, par exemple. Pour ce faire, on peut complètement se désengager de la détection de changements automatique dans notre composant, et gérer la chose soi-même, en injectant `ChangeDetectorRef`. Cette classe expose les méthodes suivantes :

- `detach()`
- `detectChanges()`

- `markForCheck()`
- `reattach()`

Les deux premières vont de pair : tu peux indiquer à Angular de ne pas déclencher la détection de changements sur le composant en appelant la méthode `detach`, et ensuite appeler `detectChanges` pour déclencher explicitement, quand tu le désires, une détection de changements :

```
@Component({
 selector: 'ns-clock',
 template: `
 <h2>Clock</h2>
 <p>{{ getTime() }}</p>
 <button (click)="start()">Start</button>
 `,
 standalone: true
})
export class ClockComponent {
 time = 0;
 interval$ = interval(10).pipe(
 take(1001), // 0, 1, ..., 1000
 map(time => time * 10),
 takeUntilDestroyed()
);

 constructor(private ref: ChangeDetectorRef) {}

 start(): void {
 this.ref.detach();
 this.interval$.subscribe(time => {
 this.time = time;
 // manually trigger the change detection every second
 if (this.time % 1000 === 0) {
 this.ref.detectChanges();
 }
 });
 }

 getTime(): number {
 return this.time;
 }
}
```

Dans l'exemple ci-dessus, c'est ce qu'on fait. On détache le composant de la détection de changements automatique grâce au `ChangeDetectorRef` injecté. Et, chaque seconde, on appelle explicitement `detectChanges()` pour déclencher une détection. La propriété `time` est toujours mise à jour 100 fois par seconde, mais la valeur affichée n'est plus modifiée que toutes les secondes.

Notez que, lorsqu'on appelle `detectChanges()`, la détection est seulement déclenchée sur le composant et ses enfants.

Nous pouvons aller plus loin, et gérer complètement la mise à jour du DOM par nous-mêmes, sans déclencher une quelconque détection de changements :

```
@Component({
 selector: 'ns-clock',
 template: `
 <h2>Clock</h2>
 <p #clock></p>
 <button (click)="start()">Start</button>
 `,
 standalone: true
})
export class ClockComponent {
 time = 0;
 @ViewChild('clock') clock!: ElementRef<HTMLParagraphElement>;
 interval$ = interval(10).pipe(
 take(1001), // 0, 1, ..., 1000
 map(time => time * 10),
 takeUntilDestroyed()
);

 constructor(private ref: ChangeDetectorRef) {}

 start(): void {
 this.ref.detach();
 this.interval$.subscribe(time => {
 this.time = time;
 if (this.time % 1000 === 0) {
 this.clock.nativeElement.textContent = `${time}`;
 }
 });
 }
}
```

Ici, on obtient une référence à l'élément du DOM à modifier, et on change son texte chaque fois qu'on le désire, sans utiliser la détection de changements.

Comme je le disais en introduction de cette section, cet exemple montre un cas assez exceptionnel et avancé, mais `ChangeDetectorRef` peut réellement se montrer utile dans certains cas. Imaginons que l'exemple qui change la couleur du poney toutes les secondes n'utilise pas un observable, mais un simple `setInterval`.

```
@Component({
 selector: 'ns-pony',
 template: '',
 standalone: true,
 changeDetection: ChangeDetectionStrategy.OnPush
})
export class PonyComponent implements OnInit, OnDestroy {
```

```

@Input({ required: true }) ponyModel!: PonyModel;
private intervalId: number | null = null;

ngOnInit(): void {
 this.intervalId = window.setInterval(() => {
 this.ponyModel.color = this.randomColor();
 }, 1000);
}

ngOnDestroy(): void {
 if (this.intervalId) {
 window.clearInterval(this.intervalId);
 }
}

```

A cause de `OnPush`, la vue n'est pas mise à jour. Et dans ce cas, on ne peut pas utiliser le `pipe async`, puisqu'on n'a plus d'observable...

On peut corriger le problème en appelant `markForCheck` sur le `ChangeDetectorRef`. Cette méthode permet, sur un composant qui utilise `OnPush`, de signaler à Angular que l'état a été modifié et qu'il faut déclencher une détection de changements :

```

@Component({
 selector: 'ns-pony',
 template: '',
 standalone: true,
 changeDetection: ChangeDetectionStrategy.OnPush
})
export class PonyComponent implements OnInit, OnDestroy {
 @Input({ required: true }) ponyModel!: PonyModel;
 private intervalId: number | null = null;

 constructor(private ref: ChangeDetectorRef) {}

 ngOnInit(): void {
 this.intervalId = window.setInterval(() => {
 this.ponyModel.color = this.randomColor();
 this.ref.markForCheck();
 }, 1000);
 }

 ngOnDestroy(): void {
 if (this.intervalId) {
 window.clearInterval(this.intervalId);
 }
 }
}

```

Et ça fonctionne à nouveau !

## 25.5. NgZone

Une autre façon d'éviter la détection de changements est possible : on peut exécuter une partie du code hors de la "surveillance" de Zone.js, qui est la librairie permettant de déclencher la détection de changements. Pour se mettre "hors-zone", on injecte le service `NgZone`, et on utilise sa méthode `runOutsideAngular` pour exécuter du code hors de sa surveillance :

Cela peut être très pratique pour envelopper une portion de code qui produit beaucoup d'événements, typiquement du code qui vient d'une bibliothèque tierce. Par exemple, imaginons un composant qui affiche un graphique construit avec `Chart.js`. Lorsque nos utilisateurs vont survoler le graphique, des centaines d'événements DOM se déclencheront. Le fait que des centaines d'événements soient déclenchés n'est pas vraiment un problème. C'est la même chose pour n'importe quel élément DOM. Le problème est que la bibliothèque utilise `addEventListener()` pour gérer ces événements, par exemple pour afficher des info-bulles, et c'est ce qui déclenche des centaines de détections de changements !

```
@ViewChild('chart') canvas!: ElementRef<HTMLCanvasElement>;
chart: Chart | null = null;

ngAfterViewInit(): void {
 const ctx = this.canvas.nativeElement;
 this.chart = new Chart(ctx, {
 type: 'bar',
 data: {
 labels: ['Green', 'Red'],
 datasets: [{ label: 'Score', data: [12, 21] }]
 }
 });
}
```

Dans ce cas tu peux injecter `NgZone` et utiliser `runOutsideAngular` pour ignorer les événements issus de `Chart.js` :

```
@ViewChild('chart') canvas!: ElementRef<HTMLCanvasElement>;
chart: Chart | null = null;

constructor(private zone: NgZone) {}

ngAfterViewInit(): void {
 const ctx = this.canvas.nativeElement;
 this.zone.runOutsideAngular(() => {
 this.chart = new Chart(ctx, {
 type: 'bar',
 data: {
 labels: ['Green', 'Red'],
 datasets: [{ label: 'Score', data: [12, 21] }]
 }
 });
 });
}
```

```
});
}
```

Cela produit le même résultat visuel et les autres changements effectués sur le composant seraient toujours détectés automatiquement par Angular. Mais on ne déclenchera pas de détection de changement quand on utilisera le graphique. `runOutsideAngular` est donc plus adapté aux situations où seules certaines portions du code doivent échapper à la détection de changements.

### 25.5.1. Pipes purs

Comme tu le sais maintenant, tu peux définir tes propres *pipes*, la plupart du temps pour formater des données. Par exemple, pour afficher le nom complet d'un utilisateur, tu pourrais utiliser une méthode dans le composant :

```
@Component({
 selector: 'ns-menu',
 template: `
 <p>{{ user.name() }}</p>
 <p>...</p>
 <p>{{ user.name() }}</p>
 `,
 standalone: true
)
export class MenuComponent {
 user: UserModel = {
 id: 1001,
 firstName: 'Jane',
 lastName: 'Doe',
 title: 'Miss'
 };

 userName(): string {
 return `${this.user.title} ${this.user.firstName} ${this.user.lastName}`;
 }
}
```

ou écrire un *pipe* pour encapsuler cette logique :

```
@Component({
 selector: 'ns-menu',
 template: `
 <p>{{ user | displayName }}</p>
 <p>...</p>
 <p>{{ user | displayName }}</p>
 `,
 standalone: true,
 imports: [DisplayNamePipe]
)
```

```

export class MenuComponent {
 user: UserModel = {
 id: 1001,
 firstName: 'Jane',
 lastName: 'Doe',
 title: 'Miss'
 };
}

```

```

@Pipe({
 name: 'displayName',
 standalone: true
})
export class DisplayNamePipe implements PipeTransform {
 transform(user: UserModel): string {
 return `${user.title} ${user.firstName} ${user.lastName}`;
 }
}

```

Ça nécessite un petit peu plus de travail, mais ça permet d'utiliser cette logique dans n'importe quel template.

Ce que tu ne réalises peut-être pas encore, c'est que l'utilisation d'un *pipe* est aussi plus performante. Par défaut, un *pipe* est "pur". En programmation fonctionnelle, on appelle "pure" une fonction qui n'a pas d'effet de bord, et dont le résultat dépend exclusivement de ses arguments. Sachant que le *pipe* est "pur", Angular applique une optimisation : la méthode `transform` du *pipe* n'est appelée que si la référence de l'objet qu'il transforme a changé ou si celle d'un autre argument de la méthode a changé (le principe est similaire à la stratégie `OnPush` des composants).

Cela signifie que, alors qu'une méthode du composant sera appelée à chaque détection de changements, un *pipe* pur ne sera exécuté que lorsque c'est nécessaire. Dans notre exemple comme l'utilisateur ne change jamais, le *pipe* ne sera même appelé qu'une seule fois, parce que les deux utilisations du *pipe* utilisent exactement les mêmes arguments.

Comme un *pipe* est pur par défaut, cette optimisation est gratuite et sans effort ! Mais parfois, elle pose problème.

Dans l'exemple précédent, si on modifie l'état de l'utilisateur en assignant une autre valeur à sa propriété `firstName`, la valeur affichée à l'écran reste identique... Il s'agit là du même problème que celui évoqué dans la section sur la stratégie `OnPush` : la référence à l'objet passé en argument au *pipe* ne change pas, donc le *pipe* n'est pas réexécuté.

Deux solutions sont possibles :

- utiliser des objets immuables (ne pas modifier l'état de l'utilisateur, mais créer un nouvel objet avec un nouveau `firstName`) ;
- marquer le *pipe* comme "impur", pour désactiver l'optimisation. Les performances vont être un peu amoindries, mais la valeur sera toujours rafraîchie, quelle que soit la manière dont les

changements sont effectués.

Pour rendre un *pipe* impur, il suffit d'ajouter `pure: false` à son décorateur :

```
@Pipe({
 name: 'displayName',
 pure: false,
 standalone: true
})
export class DisplayNameImpurePipe implements PipeTransform {
 transform(user: UserModel): string {
 return `${user.title} ${user.firstName} ${user.lastName}`;
 }
}
```

Pour résumer :

- un *pipe* pur n'est pas appelé autant de fois qu'une méthode d'un composant
- mais il n'est pas appelé si ses arguments sont "mutés", donc il s'agit d'être prudent et rigoureux.

### 25.5.2. Diviser efficacement son template

Avec tout ce qu'on vient d'apprendre, voici une astuce qui n'utilise aucune nouvelle API spécifique d'Angular mais qui peut être comprise aisément.

Supposons que nous voulions une longue liste de résultats, et un champ de recherche permettant de modifier le contenu de cette liste. Comme on ne veut pas mettre à jour la liste après chaque saisie d'un nouveau caractère, on peut utiliser `debounceTime`. Par exemple :

```
@Component({
 selector: 'ns-results',
 template: `
 <input [formControl]="search" />
 <h1>{{ resultsTitle() }}</h1>
 <div *ngFor="let result of results">{{ result }}</div>
 `,
 standalone: true,
 imports: [NgFor, ReactiveFormsModule]
})
export class ResultsComponent implements OnInit {
 search = new FormControl('', { nonNullable: true });
 results: Array<string> = [];

 constructor(private searchService: SearchService) {}

 ngOnInit(): void {
 this.search.valueChanges
 .pipe(
 debounceTime(500),
 map((value) => this.searchService.search(value))
)
 .subscribe((result) => this.results = result);
 }
}
```

```

 switchMap(query => this.searchService.updateResults(query))
)
 .subscribe(results => (this.results = results));
}

resultsTitle(): string {
 return `${this.results.length} results`;
}
}

```

Cette technique permet d'éviter de modifier la liste trop souvent, puisque la liste n'est effectivement modifiée que lorsque l'utilisateur s'arrête d'écrire. Mais cela réduit-il le nombre de *détections* de changements ? Pas du tout. La détection de changements est toujours déclenchée chaque fois que la valeur du champ est modifiée. Tu peux le vérifier facilement en ajoutant un simple `console.log` dans la méthode `resultTitle`, et constater qu'elle est appelée à chaque saisie clavier.

La détection de changements sur cette longue liste est inutile tant qu'on n'a pas effectivement modifié la liste. La solution est relativement simple. L'idée est de diviser la vue en deux parties, et d'introduire un sous-composant dédié à l'affichage de la longue liste de résultats. Ce sous-composant est très basique : il prend une liste de résultats en entrée, et se contente de l'afficher. Comme sa vue dépend exclusivement de ses *inputs*, il peut utiliser la stratégie `OnPush`. Et le nombre de détections de changements va donc être significativement réduit, puisque seule la mise à jour de la liste de résultats déclenchera la vérification du sous-composant.

Voici le composant principal :

```

@Component({
 selector: 'ns-results',
 template: `
 <input [formControl]="search" />
 <ns-results-list [results]="results"></ns-results-list>
 `,
 standalone: true,
 imports: [ReactiveFormsModule, ResultsListComponent]
})
export class ResultsComponent implements OnInit {
 search = new FormControl('', { nonNullable: true });
 results: Array<string> = [];

 constructor(private searchService: SearchService) {}

 ngOnInit(): void {
 this.search.valueChanges
 .pipe(
 debounceTime(500),
 switchMap(query => this.searchService.updateResults(query))
)
 .subscribe(results => (this.results = results));
 }
}

```

```
}
```

et le sous-composant qui affiche les résultats :

```
@Component({
 selector: 'ns-results-list',
 template: `
 <h1>{{ resultsTitle() }}</h1>
 <div *ngFor="let result of results">{{ result }}</div>
 `,
 changeDetection: ChangeDetectionStrategy.OnPush,
 standalone: true,
 imports: [NgFor]
})
export class ResultsListComponent {
 @Input() results: Array<string> = [];

 resultsTitle(): string {
 return `${this.results.length} results`;
 }
}
```

Cette technique est fréquemment utilisée, et on l'appelle le *smart/dumb component pattern* : un composant *smart* gère le chargement des données, la gestion des événements, etc., et fournit simplement des données à afficher à un second composant *dumb*. L'unique responsabilité du composant *dumb* est d'afficher les données passées en *inputs*, et éventuellement d'émettre des événements à destination du composant *smart* en utilisant des *outputs*. C'est le composant *dumb* qui a le template le plus grand, contenant de nombreuses expressions et affichant de nombreuses données. Mais comme son état ne change que lorsque le *smart* component lui fournit de nouveaux *inputs* il peut utiliser la stratégie *OnPush* et donc éviter de nombreuses évaluations d'expressions.

### 25.5.3. Décorateur Attribute

Lorsqu'on utilise un `@Input()` dans un composant, Angular suppose que la valeur passée en *input* peut changer, et fait ce qu'il faut pour détecter le changement et passer la nouvelle valeur au composant. Parfois, ce n'est pas vraiment nécessaire, parce qu'on veut simplement passer une valeur une seule fois pour initialiser le composant, et ne jamais la modifier. Dans ce cas très précis, tu peux utiliser le décorateur `@Attribute()` au lieu de `@Input()`.

Considérons un composant bouton, auquel on veut passer un type pour spécifier l'aspect qu'il doit avoir (par exemple, `primary`, `success`, `warning`, `danger`...).

Avec un *input*, ça donnerait ceci :

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

@Component({
 selector: 'ns-button',
```

```

template: `
 <button type="button" class="btn" [class]="'btn-' + btnType">
 <ng-content></ng-content>
 </button>
`,
standalone: true
})
export class ButtonComponent {
 @Input() btnType: 'primary' | 'success' = 'primary';
}

```

qu'on peut utiliser de cette manière :

```

<ns-button btnType="primary">Hello!</ns-button>
<ns-button btnType="success">Success</ns-button>

```

Comme l'*input* est une simple chaîne de caractères qui ne change jamais, on peut utiliser un attribut :

```

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

@Component({
 selector: 'ns-button',
 template: `
 <button type="button" class="btn" [class]="'btn-' + btnType">
 <ng-content></ng-content>
 </button>
 `,
 standalone: true
})
export class ButtonComponent {
 constructor(@Attribute('btnType') public btnType: string) {}
}

```

Le *binding* n'est fait qu'une seule fois, ce qui rend les choses un peu plus rapides. Mais retiens que cela ne fonctionne que pour des chaînes de caractères, statiques.

#### 25.5.4. Conclusion

Ce chapitre, j'espère, t'as appris quelques techniques permettant de résoudre d'éventuels problèmes de performance. Mais avant tout, retiens les règles d'or de l'optimisation des performances (en anglais dans le texte) :

- don't
- don't... yet
- profile before optimizing.

Comme l'a dit un célèbre informaticien :

l'optimisation prématuée est la source de tous les maux.

— Donald Knuth

Alors pense d'abord à écrire du code aussi simple, correct et lisible que possible, et ne pense à profiler l'application, puis à l'optimiser, que si tu as réellement un problème de performance.



Essaye notre exercice [Performance 🚀](#) ! Tu optimiseras l'application et mesureras chaque progrès effectué !

# Chapter 26. Signaux

L'équipe Angular a travaillé sur une nouvelle façon de gérer la réactivité dans les applications pendant l'année passée. Le résultat de leur travail est un nouveau concept dans Angular appelé les signaux.

Voyons ce qu'ils sont, comment ils fonctionnent, comment ils interopèrent avec RxJS, et ce qu'ils vont changer pour les développeurs Angular.

## 26.1. Les raisons derrière les signaux

Les signaux sont un concept utilisé dans de nombreux autres frameworks, comme SolidJS, Vue, Preact, et même le vénérable KnockoutJS. L'idée est d'offrir quelques primitives pour définir l'état réactif dans les applications et de permettre au framework de savoir quels composants sont impactés par un changement, plutôt que de devoir détecter les changements dans l'arbre complet des composants.

Cela va être un changement significatif dans le fonctionnement d'Angular, car il repose actuellement sur zone.js pour détecter les changements dans l'arbre complet des composants par défaut. À la place, avec les signaux, le framework ne va vérifier que les composants impactés par un changement, ce qui rend le processus de rendu plus efficace.

Cela ouvre la porte aux applications "zoneless", c'est-à-dire des applications Angular qui n'ont pas besoin d'inclure Zone.js (ce qui les rend plus légères), n'ont pas besoin de patcher toutes les APIs des navigateurs (ce qui les rend plus rapides au démarrage) et qui sont plus efficaces dans leur détection des changements (pour ne vérifier que les composants impactés par un changement).

Les signaux sont sortis dans Angular v16, en tant qu'API *developer preview*, ce qui signifie qu'il peut y avoir des changements dans le nommage ou le comportement dans le futur. Mais cette preview n'est qu'une petite partie des changements qui vont arriver avec les signaux. Dans le futur, des composants basés sur les signaux avec des inputs et des queries basées sur les signaux, ainsi que des hooks de cycle de vie différents, vont être ajoutés à Angular. D'autres APIs comme les paramètres du routeur et les valeurs et états des contrôles de formulaire, etc. devraient aussi être impactées.

## 26.2. API des signaux

Un signal est une fonction qui contient une valeur qui peut changer au fil du temps. Pour créer un signal, Angular offre une fonction `signal()` dans `@angular/core` :

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

Tu peux ensuite créer un signal avec :

```
// define a signal
const count = signal(0);
```

Le type de `count` est `WritableSignal<number>`, qui est une fonction qui retourne un nombre.

Quand tu veux obtenir la valeur d'un signal, tu dois appeler le signal créé :

```
// get the value of the signal
const value = count();
```

Cela peut être fait à la fois dans le code TypeScript et dans les templates HTML :

```
<div>{{ count() }}</div>
```

Tu peux aussi définir la valeur d'un signal :

```
// set the value of the signal
count.set(1);
```

Ou le mettre à jour :

```
// update the value of the signal, based on the current value
count.update(value => value + 1);
```

Il y a aussi une méthode `mutate` qui peut être utilisée pour muter la valeur d'un signal :

```
// mutate the value of the signal (handy for objects/arrays)
const user = signal({ name: 'JB', favoriteFramework: 'Angular' });
user.mutate(user => (user.name = 'Cédric'));
```

Tu peux aussi créer un signal en lecture seule, qui ne peut pas être mis à jour, avec `asReadonly` :

```
const readonlyCount = count.asReadonly();
```

Quand tu as défini des signaux, tu peux définir des valeurs calculées qui en dépendent :

```
const double = computed(() => count() * 2);
```

Les valeurs calculées sont automatiquement recalculées quand un des signaux dont elles dépendent change.

```
count.set(2);
console.log(double()); // logs 4
```

Note que les valeurs sont calculées de façon paresseuse et ne sont recalculées que quand un des signaux dont elles dépendent produit une nouvelle valeur. Elles ne sont pas modifiables, donc tu ne peux pas utiliser `.set()` ou `.update()` ou `.mutate()` dessus (leur type est `Signal<T>`).

Sous le capot, un signal a un ensemble d'abonnés, et quand la valeur du signal change, il notifie tous ses abonnés. Angular le fait de façon intelligente, pour éviter de tout recalculer quand un signal change, en utilisant des compteurs internes pour savoir quels signaux ont vraiment changé depuis la dernière fois qu'une valeur a été calculée.

Enfin, tu peux utiliser la fonction `effect` pour réagir aux changements dans tes signaux :

```
// log the value of the count signal when it changes
effect(() => console.log(this.count));
```

Un `effect` retourne un objet avec une méthode `destroy` qui peut être utilisée pour arrêter l'effet :

```
const effectRef: EffectRef = effect(() => console.log(this.count));
// stop executing the effect
effectRef.destroy();
```

Cela ressemble à un `BehaviorSubject`, mais il y a quelques différences subtiles, la plus importante étant que se désabonner n'est pas nécessaire grâce à l'utilisation de `weak references`.

Un `effect` ou une valeur calculée sera recalculée quand un des signaux dont elle dépend change, sans aucune action du développeur. C'est généralement ce que tu veux.

Bien que les signaux et les valeurs calculées seront courants dans les applications Angular, les effets seront utilisés moins souvent, car ils sont plus avancés et de plus bas niveau. Si tu as besoin de réagir à un changement de signal, par exemple pour récupérer des données depuis un serveur, tu peux utiliser la couche d'interopérabilité RxJS que nous détaillons ci-dessous.

Si tu veux exclure un signal des dépendances d'un effet ou d'une valeur calculée, tu peux utiliser la fonction `untracked` :

```
const multiplier = signal(2);
// 'total' will not re-evaluate when 'multiplier' changes
// it only re-evaluates when 'count' changes
const total = computed(() => count() * untracked(() => multiplier));
```

Tu ne peux *pas* écrire dans un signal à l'intérieur d'une valeur calculée, car cela créerait une boucle infinie :

```
// this will throw an error NG0600
const total = computed(() => {
 multiplier.set(2);
 return count() * multiplier();
```

```
});
```

C'est la même chose pour les effets, mais cela peut être surchargé en utilisant `allowSignalWrites` :

```
// this will not throw an error
effect(
() => {
 if (this.count() > 10) {
 this.count.set(0);
 }
},
{ allowSignalWrites: true }
);
```

Toutes ces fonctionnalités (et terminologie) sont assez courantes dans d'autres frameworks, donc tu ne seras pas surpris si tu as utilisé SolidJS ou Vue 3 auparavant.

Une fonctionnalité que je *pense* être assez unique à Angular est la possibilité de passer une `ValueEqualityFn` à la fonction `signal`. Pour décider si un signal a changé (et savoir si une valeur calculée ou un effet doit être exécuté), Angular utilise `Object.is()` par défaut, mais tu peux passer une fonction personnalisée pour comparer les anciennes et nouvelles valeurs :

```
const user = signal(
{
 id: 1,
 name: 'Cédric'
},
{ equal: (previousUser, newUser) => previousUser.id === newUser.id }
);
// uppercaseName will not re-evaluate when the user changes if the ID stays the same
const uppercaseName = computed(() => user().name.toUpperCase());
```

## 26.3. Signaux, composants et détection des changements

Comme indiqué ci-dessus, tu peux utiliser des signaux et des valeurs calculées dans tes templates :

```
<div>{{ count() }}</div>
```

Si le compteur change, Angular le détecte et met à jour le composant.

Mais que se passe-t-il si le composant est marqué comme `OnPush`? Jusqu'à présent, `OnPush` signifiait qu'Angular ne détecterait les changements sur un composant que si un de ses inputs changeait, ou si un pipe `async` utilisé dans le template rentrait une nouvelle valeur, ou si le composant utilisait `ChangeDetectorRef#markForCheck()`.

Le framework gère maintenant une autre raison de rafraîchir un composant : quand un signal change. Considérons le composant suivant (qui n'utilise pas de signal, mais un simple champ qui sera mis à jour après 2 secondes) :

```
@Component({
 selector: 'ns-count',
 standalone: true,
 template: '<div>{{ count }}</div>',
 changeDetection: ChangeDetectionStrategy.OnPush
})
export class CountSimpleFieldComponent {
 count = 0;

 constructor() {
 // set the counter to 1 after 2 seconds
 setTimeout(() => (this.count = 1), 2000);
 }
}
```

Comme le composant est `OnPush`, utiliser un `setTimeout` déclenchera une détection des changements, mais le composant ne sera pas mis à jour (car il ne sera pas marqué comme "dirty").

Mais si nous utilisons un signal à la place :

```
@Component({
 selector: 'ns-user',
 standalone: true,
 template: '<div>{{ count() }}</div>',
 changeDetection: ChangeDetectionStrategy.OnPush
})
export class CountSignalComponent {
 count = signal(0);

 constructor() {
 // set the counter to 1 after 2 seconds
 setTimeout(() => this.count.set(1), 2000);
 }
}
```

Alors le composant sera mis à jour après 2 secondes, car Angular détecte le changement du signal et rafraîchit le template.

Pour rendre cela possible, si une vue lit la valeur d'un signal, alors Angular marque le template du composant comme consommateur du signal. Ensuite, quand la valeur du signal change, il marque le composant et tous ses ancêtres comme "dirty".

Pour nous développeurs, cela signifie que nous pouvons utiliser des signaux dans nos composants dès Angular v16, et nous n'avons pas besoin de nous soucier de la détection des changements,

même si les composants utilisent [OnPush](#).

## 26.4. Partager un signal entre composants

Dans l'exemple précédent, le signal était défini dans le composant lui-même. Mais que se passe-t-il si nous voulons partager un signal entre plusieurs composants?

Dans l'écosystème Vue, tu rencontres fréquemment le pattern des "composables" : des fonctions qui retournent un objet contenant des signaux, des valeurs calculées et des fonctions pour les manipuler. Si un signal doit être partagé, il est défini en dehors de la fonction et retourné par la fonction :

```
const count = signal(0);
export function useCount() {
 return {
 count,
 increment: () => count.set(count() + 1)
 };
}
```

En Angular, nous pouvons faire la même chose et nous pouvons également utiliser des services au lieu de fonctions (et c'est probablement ce que nous ferons). Nous pouvons définir un [CountService](#) comme suit :

```
@Injectable({ providedIn: 'root' })
export class CountService {
 count = signal(0);

 increment() {
 this.count.update(value => value + 1);
 }
}
```

Puis, dans nos composants, nous pouvons injecter le service et utiliser le signal :

```
@Component({
 selector: 'ns-user',
 standalone: true,
 template: '<div>{{ count() }}</div>',
 changeDetection: ChangeDetectionStrategy.OnPush
})
export class CountWithServiceComponent {
 count = inject(CountService).count;
}
```

Le service pourrait également définir des valeurs calculées, des effets, des méthodes pour

manipuler les signaux, etc.

```
@Injectable({ providedIn: 'root' })
export class CountService {
 count = signal(0);
 double = computed(() => this.count() * 2);

 increment() {
 this.count.update(value => value + 1);
 }
}
```

## 26.5. Fuites mémoire

Les consommateurs et producteurs de signaux sont liés entre eux en utilisant des références faibles, ce qui signifie que si tous les consommateurs d'un signal cessent d'exister, alors le signal sera également "garbage collecté". En d'autres termes : pas besoin de se soucier de "se désabonner" d'un signal pour éviter les fuites mémoire, comme nous devons le faire avec les observables RxJS `\o/`.

Tu peux également utiliser un `effect` dans ton composant (même si cela sera probablement très rare) pour observer une valeur et réagir à ses changements, par exemple, le `count` défini dans un service comme ci-dessus.

Note que tu n'as pas besoin d'arrêter manuellement un effet quand le composant est détruit, car Angular le fera pour toi pour éviter les fuites mémoire. Sous le capot, un effet utilise un `DestroyRef`, une nouvelle fonctionnalité introduite en Angular v16, qui permet de nettoyer automatiquement les ressources quand un composant ou un service est détruit.

Tu peux cependant changer ce comportement en créant un effet avec une option spécifique :

```
this.logEffect = effect(() => console.log(this.count()), { manualCleanup: true });
```

Dans ce cas, tu devras arrêter manuellement l'effet quand le composant est détruit :

```
ngOnDestroy() {
 this.logEffect.destroy();
}
```

Les effets peuvent également recevoir une fonction de nettoyage, qui est exécutée quand l'effet est exécuté à nouveau. Cela peut être pratique quand tu as besoin d'arrêter une action précédente avant d'en démarrer une nouvelle. Dans l'exemple ci-dessous, nous démarrons un intervalle qui s'exécute toutes les `count` secondes, et nous voulons l'arrêter et en démarrer un nouveau quand `count` change :

```

this.intervalEffect = effect(onCleanup => {
 const intervalId = setInterval(() => console.log(`count in intervalEffect ${this.count()}`), this.count() * 1000);
 return onCleanup(() => clearInterval(intervalId));
});

```

Note que les effets s'exécutent pendant la détection des changements, donc ce n'est pas un bon endroit pour définir des valeurs de signaux. C'est pourquoi tu obtiens une erreur d'Angular si tu essaies de le faire :

```
// ERROR: Cannot set a signal value during change detection
```

Les effets seront probablement utilisés très rarement, mais ils peuvent être utiles dans certains cas :

- logger / tracer
- synchroniser l'état avec le DOM ou un stockage, etc.

## 26.6. Signaux et interopérabilité RxJS

RxJS est là pour rester, même si son utilisation pourrait être plus limitée à l'avenir. Angular ne va pas supprimer RxJS, et il ne va pas nous forcer à utiliser des signaux à la place des observables. En fait, RxJS est probablement un meilleur moyen de réagir aux changements de signaux que d'utiliser des effets.

Nous pouvons utiliser des signaux et des observables ensemble, et nous pouvons les convertir l'un en l'autre. Deux fonctions pour faire cela sont disponibles dans le tout nouveau package [@angular/core/rxjs-interop](#).

Pour convertir un signal en observable, nous pouvons utiliser la fonction `toObservable` :

```
count$ = toObservable(this.count);
```

Note que l'observable créé ne recevra pas tous les changements de valeur du signal, car cela est fait en utilisant un effet, et les effets ne sont exécutés que pendant la détection des changements :

```

count$.subscribe(value => console.log(value));
count.set(1);
count.set(2);
// logs only 2, not 0 and 1, as this is the value when the under-the-hood effect runs

```

Pour convertir un observable en signal, nous pouvons utiliser la méthode `toSignal` :

```
countFromObservable = toSignal(count$);
```

Le signal contiendra la dernière valeur émise par l'observable, ou émettra une erreur si l'observable émet une erreur. Note que l'abonnement créé par `toSignal()` est automatiquement annulé quand le composant qui l'a déclaré est détruit. Comme les observables peuvent être asynchrones, tu peux passer une valeur initiale à la fonction :

```
countFromObservableWithInitialValue = toSignal(this.count$, { initialValue: 0 });
```

Si tu ne fournis pas de valeur initiale, la valeur du signal sera `undefined` si elle est lue avant que l'observable n'émette une valeur. Tu peux également utiliser l'option `requireSync` pour faire en sorte que le signal lève une erreur si elle est lue avant que l'observable n'émette une valeur :

```
countFromObservableWithRequireSync = toSignal(this.otherCount$, { requireSync: true });
```

## 26.7. Composants basés sur les signaux

Dans le futur (v17 ? v18 ?), nous serons en mesure de construire un composant entièrement basé sur les signaux, même pour ses inputs et requêtes. Le framework serait notifié quand une expression a changé grâce aux signaux, et n'aurait donc besoin de vérifier que les composants affectés par le changement, sans avoir besoin de vérifier les changements sur les composants non affectés, sans avoir besoin de zone.js. Il serait même capable de ne mettre à jour que la partie du template qui a changé, au lieu de vérifier le template entier d'un composant comme il le fait actuellement. Mais il y a encore du chemin à parcourir, car plusieurs choses doivent être repensées dans le framework pour que cela fonctionne (inputs, outputs, requêtes, méthodes de cycle de vie, etc).

Une RFC est disponible avec les détails de la proposition, et tu peux suivre les progrès sur le [repo Angular](#).

Actuellement, la RFC propose d'utiliser `signals: true` pour marquer un composant comme "basé sur les signaux" :

```
@Component({
 signals: true,
 selector: 'ns-temperature',
 template: `
 <p>C: {{ celsius() }}</p>
 <p>F: {{ fahrenheit() }}</p>
 `
})
export class TemperatureComponent {
 celsius = signal(25);

 // The computed only re-evaluates if celsius() changes.
 fahrenheit = computed(() => this.celsius() * 1.8 + 32);
}
```

Inputs, outputs et requêtes seraient définis via des fonctions au lieu de décorateurs dans ces composants, et retourneraient un signal :

```
firstName = input<string>(); // Signal<string|undefined>
```

Rien n'a encore été implémenté pour cette partie, donc tu ne peux pas essayer cela dans la v16. Mais tu peux déjà essayer d'utiliser des signaux dans les composants existants, comme nous l'avons mentionné ci-dessus (mais garde à l'esprit qu'ils ne sont pas prêts pour la production).

C'est de toute façon assez intéressant de voir comment les frameworks s'inspirent les uns des autres, avec Angular s'inspirant de Vue et SolidJS pour la partie réactivité, alors que d'autres frameworks adoptent de plus en plus l'approche de compilation des templates d'Angular, sans avoir besoin de Virtual DOM à l'exécution.

Le futur d'Angular va être palpitant !

# Chapter 27. Livrer en production

Voilà, maintenant tu as construit une belle application, et tu penses sérieusement à la mettre à disposition du monde entier. Discutons donc de ce qui doit être fait pour livrer en production !

## 27.1. Environnements et configurations

Si tu utilises Angular CLI, il est possible de définir plusieurs environnements.

Pour cela, la CLI fournit une schematic `environment` :

```
ng generate environments
```

Cela génère des fichiers nommés `environment.ts` et `environment.development.ts`.

Ces fichiers contiennent un objet vide appelé `environment`, dans lequel tu peux ajouter autant de propriétés que tu le souhaites.

`environment.ts`

```
export const environment = {};
```

Tu vas ensuite seulement importer `environment.ts` dans l'application. C'est un peu étrange, mais la CLI va ensuite utiliser le bon fichier selon l'environnement.

Quand tu sers (avec `ng serve`) ton application, la CLI (Webpack, pour être exact) va utiliser `environment.development.ts`.

Mais tu peux aussi servir ou construire ton application avec une configuration spécifique. Par défaut la CLI propose une autre configuration nommée `production`.

Donc, tu peux aussi exécuter `ng serve --configuration=production`. Les différences entre ces deux configurations sont visibles dans les fichiers `angular.json` :

`angular.json`

```
"configurations": {
 "production": {
 "budgets": [
 {
 "type": "initial",
 "maximumWarning": "500kb",
 "maximumError": "1mb"
 },
 {
 "type": "anyComponentStyle",
 "maximumWarning": "2kb",
 "maximumError": "4kb"
 }
]
 }
}
```

```

 }
],
"outputHashing": "all"
},
"development": {
 "buildOptimizer": false,
 "optimization": false,
 "vendorChunk": true,
 "extractLicenses": false,
 "sourceMap": true,
 "namedChunks": true,
 "fileReplacements": [
 {
 "replace": "src/environments/environment.ts",
 "with": "src/environments/environment.development.ts"
 }
]
},

```

Comme tu le vois, il y a une configuration `production` avec son lot de propriétés. Les `budgets`, par exemple, vérifient que le bundle initial et les styles des composants ne sont pas trop lourds.

Il y a une propriété hyper utile dans la configuration de développement, que la schematic a ajouté : `fileReplacements`.

`angular.json`

```

"fileReplacements": [
{
 "replace": "src/environments/environment.ts",
 "with": "src/environments/environment.development.ts"
}
]

```

Tu peux voir que le fichier `environment.ts` est remplacé par `environment.development.ts` : c'est comme cela que Webpack sait quel fichier utiliser.

Tu vas ensuite toujours importer ces variables depuis `environment.ts` dans ton code, et, pendant le build, la CLI choisira le bon fichier d'environnement selon la configuration.

Cela signifie que tu peux donc définir autant de configurations que tu le souhaites. Par exemple, tu pourras ajouter une configuration `preprod` avec son fichier `environment.preprod.ts` dédié.

Cela signifie aussi que tu peux remplacer autant de fichiers que tu veux dans ton application. On peut donc imaginer des choses un peu folles comme remplacer `pony.component.ts` par une version différente (mais je ne vois pas bien pourquoi tu voudrais faire ça ^^).

Un fichier environnement peut contenir tout ce que tu veux. Mais comme son nom l'indique, il est censé contenir du code spécifique à un environnement donné (développement, production, pré-production, etc.). Par exemple, tu peux avoir des URLs d'API différentes en développement et dans

la version en production.

Comme l'environnement `production` est très fréquemment celui que tu veux quand tu construis l'application avec `ng build`, la CLI l'utilise par défaut depuis la version 12.0 (plutôt que de devoir préciser `--configuration=production`).

Depuis la CLI version 9.0, il est possible de spécifier plusieurs configurations :

```
ng build --configuration=production,preprod
```

La commande utilise alors la configuration `production`, fusionnée avec la configuration `preprod`. Si la même propriété est définie dans les deux configurations, celle de la configuration `preprod` prend le dessus sur celle de `production`.

Note que ce mécanisme de remplacement est aussi disponible pour les assets et les styles dans la CLI, t'offrant ainsi la possibilité de proposer des thèmes différents de ton application par le seul jeu des configurations.

## 27.2. strictTemplates

Lorsque tu compiles ton application en AoT (`aot: true`, valeur par défaut depuis Angular 9.0), les templates sont vérifiés par le compilateur Angular.

Par défaut, seule une vérification superficielle est réalisée. Pour aller un peu plus loin, tu peux utiliser l'option de compilation `strictTemplates` :

`tsconfig.json`

```
"angularCompilerOptions": {
 "strictTemplates": true,
}
```

Avec cette option, le compilateur va vérifier que les valeurs passées aux inputs, les événements DOM utilisés, les références dans les templates, etc. ont tous un typage correct. Par exemple, essayer de donner un nombre à un input qui attend un booléen provoquera une erreur de compilation.

Angular 13.2 a également ajouté une nouvelle option appelée `extendedDiagnostics`, qui fait quelques vérifications supplémentaires. Par exemple, cela vérifie que tu utilise bien la syntaxe du lien bidirectionnel correctement (la "syntaxe boîte de bananes" `[(ngModel)]`, et pas `([ngModel])`). L'option trace des avertissements par défaut, mais tu peux la configurer pour qu'elle émette des erreurs :

`tsconfig.json`

```
"angularCompilerOptions": {
 "strictTemplates": true,
 "extendedDiagnostics": {
 "defaultCategory": "error"
 }
```

```
}
```

## 27.3. Assembler ton application

J'ai un peu divugalché la suite dans les étapes précédentes. Si tu veux assembler ton application pour la production avec la CLI, il te suffit juste d'exécuter :

```
ng build
```

Cela va créer un répertoire `dist` contenant le résultat de la construction. L'option `prod` déclenche le remplacement de fichier dont on a parlé, mais ajoute aussi quelques options supplémentaires, et certaines sont vraiment intéressantes.

Cela active également le *tree-shaking* et l'élimination du code mort, grâce à `optimization: true`. Pour rendre ce processus encore plus efficace, l'équipe d'Angular CLI a écrit un outil nommé "build optimizer" (*l'optimisateur de construction*), qui est activé par l'option `buildOptimizer: true`.

Pour réduire encore plus le volume de code JavaScript généré, les bibliothèques tierces ne sont pas packagées dans un fichier séparé (`vendorChunk: false`), et les licences sont supprimées (`extractLicences: true`).

Tu n'as certainement pas envie de déboguer en production. Et tu n'as probablement pas envie non plus de fournir le code source non-minifié à tous tes visiteurs. Ainsi `sourceMap: false` désactive la coûteuse génération des *source maps*.

Une dernière option intéressante est `outputHashing` : elle indique à Webpack que les fichiers générés ne doivent pas être nommés `main.js` mais `main.xxxxxx.js`, où `xxxxxx` est le résultat d'une fonction cryptographique de hachage sur le contenu du fichier. Pourquoi ? Pour être sûr que ces fichiers puissent être mis en cache sans problème : rappelle-toi de la partie "cache busting" du [chapitre Performances](#) plus haut.

Comme tu le vois, la CLI déclenche plein d'optimisations pour nous. Si tu choisis de ne pas utiliser la CLI, il te faudra imaginer des solutions pour faire la même chose avec Webpack ou ton outil préféré.

## 27.4. Configuration du serveur

La dernière étape est de prendre le résultat de l'étape de construction (dans le répertoire `dist`) et le déployer sur ton serveur préféré. Cela peut être un serveur statique comme Apache ou Nginx par exemple.



N'utilise pas `ng serve` en production, même avec l'option `--configuration=production`. Ce n'est qu'un outil de développement, qui n'est ni optimisé ni sécurisé pour la production.

Il reste cependant quelques détails à configurer. Et ces détails dépendent du serveur web choisi.

Tout d'abord, tu vas t'assurer que toutes les ressources sont servies compressées (probablement avec gzip). Ensuite, tu vas t'assurer que toutes ces ressources sont mises en cache pour longtemps. Ne te tracasse pas à propos des soucis classiques de cache, puisqu'on a généré les assets avec un nom unique fonction de leur contenu.

La dernière étape est moins évidente et souvent oubliée (oui, ça nous est arrivé aussi 🤦) : il te faut configurer ton serveur pour que toutes les routes servent `index.html`.

C'est logique : tu as déployé ton application sur <https://ng-ponyracer.ninja-squad.com>, tu l'as tweetée au monde entier, et les gens ont commencé à y affluer. Ton serveur va leur servir le fichier `index.html`, et jusque-là tout va bien. Ils vont naviguer dans l'application, et peut-être atterrir sur la liste des courses à <https://ng-ponyracer.ninja-squad.com/races>.

Mais que va-t-il se passer si quelqu'un appuie sur `F5/Cmd + R` ? La requête envoyée au serveur sera pour `/races`, et si tu ne l'as pas prévu, ton serveur va retourner une erreur `404`...

Donc il faut t'assurer, d'une façon ou d'une autre, que toute requête vers toute route de l'application servira `index.html`. L'application Angular redémarrera entièrement, le routeur analysera l'URL, et il naviguera vers la bonne route immédiatement.

## 27.5. Conclusion

Je pense qu'on a abordé les points importants de la livraison en production. Comme tu l'as vu, c'est assez facile si tu utilises Angular CLI, aussi je ne peux que t'encourager fermement à le faire.



Essaye l'exercice [Aller en production](#) ! Tu apprendras quelques astuces intéressantes pour tes prochaines mises en prod.

# Chapter 28. Ce n'est qu'un au revoir

Merci d'avoir lu ce livre !

Quelques chapitres seront ajoutés dans des versions ultérieures sur des sujets avancés, et d'autres surprises. Ils ont encore besoin d'être fignolés, 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 livres seront gratuites, bien évidemment !

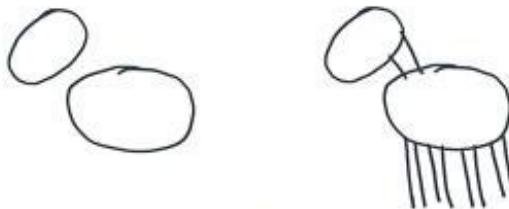
Si tu as aimé cette lecture, parles-en à tes amis !

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 :

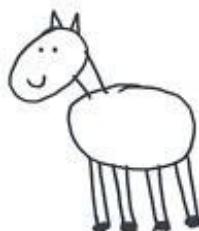
# HOW TO: DRAW A HORSE

BY VAN OKTOP

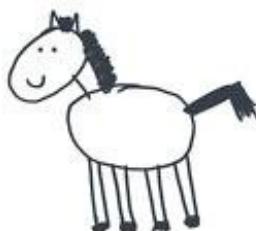


① DRAW 2 CIRCLES

② DRAW THE LEGS



③ DRAW THE FACE



④ DRAW THE HAIR



⑤  
ADD  
SMALL  
DETAILS.

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.

Current versions:

- Angular: [16.2.2](#)
- Angular CLI: [16.2.0](#)

## A.1. v16.2.0 - 2023-08-10

### Building components and directives

- Add a section about the `transform` option of `@Input`, introduced in Angular v16.1. (2023-06-24)

## A.2. v16.1.0 - 2023-06-14

## A.3. v16.0.0 - 2023-05-17

### Building components and directives

- Introduce required inputs, as added in Angular v16 (2023-05-03)

### Router

- Add a section about `withComponentInputBinding` to get router parameters and data as component inputs, as introduced in Angular v16 (2023-05-03)

### Signals

- New chapter about Signals! (2023-05-17)

### Advanced observables

- Use the `takeUntilDestroyed` RxJS operator introduced in Angular v16 (2023-05-03)

## A.4. v15.2.0 - 2023-02-23

### Router

- As Angular v15.2 deprecates class-based resolvers and guards, we now use functional resolvers and guards in all examples. (2023-02-23)

## A.5. v15.1.0 - 2023-01-11

### Dependency Injection

- Use a better example for DI configuration, with a logging service that logs to the console in development and calls an API in production. (2023-01-05)
- Add a section about the `inject()` function. (2022-12-01)

### Router

- Remove the section about the `CanLoad` guard as it is now deprecated (use `CanMatch` instead). (2023-01-11)

### Standalone components

- Add a section about HTTP with `provideHttpClient` and functional interceptors. (2022-11-30)

### Going to production

- Explains how to use `ng generate environments`. (2023-01-11)

## A.6. v15.0.0 - 2022-11-16

### Dependency Injection

- Remove the `providedIn: NgModule` syntax now that it is deprecated in Angular v15 (2022-11-16)

### Router

- The router automatically unwraps default module exports in lazy-loading routes in Angular v15 (2022-11-16)
- Showcases an example of a functional resolver (2022-11-14)
- Showcases an example of functional guard (2022-11-14)

### Standalone components

- Use the `NgFor` alias introduced in Angular v15 for the `NgForOf` directive (2022-11-16)
- The router now automatically unwraps default component exports in lazy-loading routes (2022-11-16)

### Going to production

- Replace the explanation of `enableProdMode` by a section about production mode and mention the `ngDevMode` variable. (2022-11-16)
- We now explain how to use `fileReplacements` as it is no longer included by default in CLI v15. (2022-11-16)

## A.7. v14.2.0 - 2022-08-26

### Standalone components

- Mention `provideRouter(routes)` (2022-08-26)

### Performances

- Mention the experimental `NgOptimizedImage` directive introduced in v14.2 (2022-08-26)

## A.8. v14.1.0 - 2022-07-21

### Router

- Add a section on the new `CanMatch` guard introduced in v14.1 (2022-07-21)

## A.9. v14.0.0 - 2022-06-03

### Forms

- Add a section about `FormArray` and `FormRecord` (2022-06-03)
- Add a section about typed forms (2022-06-03)
- We nows use and explain the new "strictly typed forms API"  (2022-06-03)

### Standalone components

- New chapter about standalone APIs! (2022-06-03)

### Performances

- Better example of `NgZone.runOutsideAngular` usage (2022-05-11)

## A.10. v13.3.0 - 2022-03-16

## A.11. v13.2.0 - 2022-01-27

### Forms

- The forms chapter has a new section about control value accessors, explaining how to create custom form controls (2021-12-14)

### Advanced components and directives

- The advanced components chapter has a new section about ng-template, explaining how to create customizable components using conditional, contextual content projection (2021-12-17)

### Going to production

- Section about the new `extendedDiagnostics` option introduced in v13.2 (2022-01-27)

## A.12. v13.1.0 - 2021-12-10

## A.13. v13.0.0 - 2021-11-04

### The templating syntax

- Remove the canonical `bind-`, `on-`, `ref-` syntax that has been deprecated in Angular v13 (2021-11-04)

### Going to production

- Remove the section about differential loading as it has been removed in Angular v13 (2021-11-04)
- Remove the `fullTemplateTypeCheck` explanation, as it is deprecated in Angular v13, and only keep its replacement `strictTemplates`. (2021-11-04)

## A.14. v12.2.0 - 2021-08-05

### Global

- Add links to our quizzes! (2021-07-29)

### Reactive Programming

- RxJS v7.2 allows to import operators directly from `rxjs`, so all imports have been simplified. (2021-08-05)

## A.15. v12.1.0 - 2021-06-25

## A.16. v12.0.0 - 2021-05-13

### Global

- All examples now use strict null checks. (2021-05-13)

### From zero to something

- The ebook now uses ESLint as its linter. (2021-05-13)

### Testing your app

- The e2e tests section now introduces Cypress (2021-05-13)

### Send and receive data with Http

- Section about the new `HttpContext` introduced in Angular v12. (2021-05-13)
- The HTTP examples now use the human-readable `HttpStatusCode` enum. (2021-05-13)

## Going to production

- The CLI uses the production configuration by default for `ng build` since v12, and the `--prod` flag is deprecated. (2021-05-13)

## A.17. v11.2.0 - 2021-02-12

## A.18. v11.1.0 - 2021-01-21

## A.19. v11.0.0 - 2020-11-12

### Internationalization

- `ng xi18n` has been renamed `ng extract-i18n` in CLI v11 (2020-11-12)

## A.20. v10.2.0 - 2020-10-22

### Internationalization

- `xi18N` now extracts messages from the `$localize` calls in TypeScript code (2020-09-10)

## A.21. v10.1.0 - 2020-09-03

### Testing your app

- `async` has been deprecated and renamed `waitForAsync` (2020-09-01)

### Internationalization

- Import the global variants of the locale data. It's simpler, supports all formatting options, and doesn't trigger an optimization bailout warning when building the app with the CLI. (2020-07-01)

## A.22. v10.0.0 - 2020-06-25

### Global

- Bump to ng **10.0.0** (2020-06-25)

### The wonderful world of Web Components

- Use `customElements.define` instead of the deprecated `document.registerElement`. (2020-06-17)

### Reactive Programming

- Pass an object as argument to the `Observable.subscribe()` method when an error or a completion must be handled, instead of 2 or 3 functions, because passing several functions will be deprecated in RxJS 7. (2020-06-05)

## A.23. v9.1.0 - 2020-03-26

### Global

- Bump to ng [9.1.0](#) (2020-03-26)

### From zero to something

- Bump to cli [9.1.0](#) (2020-03-26)

## A.24. v9.0.0 - 2020-02-07

### Global

- Bump to ng [9.0.0](#) (2020-02-07)
- Bump to ng [9.0.0-next.5](#) (2020-02-06)

### A gentle introduction to ECMAScript 2015+

- Add a section about tagged template strings. (2019-08-02)

### Diving into TypeScript

- Showcase interface usage for modeling entities (2019-08-10)
- Improve the `enum` section with examples of how to use union types (2019-08-10)

### Advanced TypeScript

- Introduce a new chapter about advanced TypeScript patterns, like `keyof`, mapped types, type guards, and other things! (2019-08-10)

### From zero to something

- Bump to cli [9.0.1](#) (2020-02-07)
- Bump to cli [9.0.0-next.3](#) (2020-02-06)
- Bump to cli [8.3.2](#) (2019-08-30)
- Bump to cli [8.3.0](#) (2019-08-22)

### Testing your app

- Use `TestBed.inject` instead of the deprecated `TestBed.get` in ng [9.0.0](#) (2020-02-06)

### Internationalization

- Explains how to configure the default currency code (2020-02-07)
- Introduce `@angular/localize` usage in ng [9.0.0](#) (2020-02-07)

### Going to production

- Mention the multiple configurations support introduced in CLI v9.0 (2020-02-07)
- Explain the `fullTemplateTypeCheck` and `strictTemplates` options (2020-02-07)

## A.25. v8.2.0 - 2019-08-01

### Global

- Bump to ng [8.2.0](#) (2019-08-01)

### From zero to something

- Bump to cli [8.2.0](#) (2019-08-01)

### Testing your app

- Use a more strictly typed `createSpyObj` syntax. (2019-07-31)

## A.26. v8.1.0 - 2019-07-02

### Global

- Bump to ng [8.1.0](#) (2019-07-02)

### The wonderful world of Web Components

- Mention more recent alternatives to Polymer, remove the dead HTML import spec and mention Angular Elements (2019-06-01)

### From zero to something

- Bump to cli [8.1.0](#) (2019-07-02)

## A.27. v8.0.0 - 2019-05-29

### Global

- Bump to ng [8.0.0](#) (2019-05-29)

### A gentle introduction to ECMAScript 2015+

- How to use `async/await` with promises (2019-05-19)

### From zero to something

- Bump to cli [8.0.0](#) (2019-05-29)
- Bump cli to [7.3.0](#) (2019-02-28)

### Testing your app

- Showcase the awesome `ngx-speculoos` library for cleaner unit tests (2019-05-20)

## Forms

- Showcase the awesome `ngx-valdemort` library for better validation error messages (2019-05-19)

## Router

- Use `import` for lazy-loading routes as introduced by ng [8.0.0](#) (2019-05-20)

## Angular compiler

- Update the AoT explanation and generated code for Angular 8.0.0 (Ivy) (2019-05-20)

## Advanced components and directives

- Add and explain the `static` flag for `ViewChild` and `ContentChild` introduced by Angular [8.0.0](#) (2019-05-27)

## Going to production

- Differential loading using `browserslist` as introduced by the cli [8.0.0](#). (2019-05-20)

# A.28. v7.2.0 - 2019-01-09

## Global

- Bump to ng [7.2.0](#) (2019-01-07)
- Bump to ng [7.2.0-rc.0](#) (2019-01-03)
- Bump to ng [7.2.0-beta.2](#) (2018-12-14)

## From zero to something

- Bump to cli [7.2.0](#) (2019-01-09)
- Bump to cli [7.2.0-rc.0](#) (2019-01-07)
- Bump to cli [7.2.0-beta.2](#) (2019-01-07)

# A.29. v7.1.0 - 2018-11-27

## Global

- Bump to ng [7.1.0](#) (2018-11-22)
- Bump to ng [7.1.0-rc.0](#) (2018-11-20)
- Bump to ng [7.0.2](#) (2018-11-05)

## From zero to something

- Bump to cli [7.1.0](#) (2018-11-27)
- Bump to cli [7.0.4](#) (2018-11-05)

## Router

- Use `UrlTree` in `CanActivate` guard, as introduced by 7.1 (2018-11-22)

# A.30. v7.0.0 - 2018-10-25

## Global

- Bump to ng [7.0.0](#) (2018-10-18)
- Bump to ng [7.0.0-rc.1](#) (2018-10-18)
- Bump to ng [7.0.0-rc.0](#) (2018-10-18)
- Bump to ng [7.0.0-beta.6](#) (2018-10-18)
- Bump to ng [7.0.0-beta.4](#) (2018-10-18)
- Bump to ng [7.0.0-beta.0](#) (2018-10-18)

## From zero to something

- Bump to cli [7.0.2](#) (2018-10-24)
- Bump to cli [7.0.1](#) (2018-10-18)
- Bump to cli [6.2.1](#) (2018-09-07)
- Bump to cli [6.2.0-rc.0](#) (2018-09-07)

## Performances

- Adds a performances chapter! (2018-08-30)

## Going to production

- Adds a new chapter about Going to production! (2018-10-25)

# A.31. v6.1.0 - 2018-07-26

## Global

- Bump to ng [6.1.0](#) (2018-07-26)
- Bump to ng [6.1.0-rx.0](#) (2018-07-26)
- Bump to ng [6.1.0-beta.1](#) (2018-07-26)
- Bump to ng [6.0.7](#) (2018-07-06)

## From zero to something

- Bump to cli [6.1.0](#) (2018-07-26)
- Bump to cli [6.0.8](#) (2018-07-06)
- Bump cli to [6.0.7](#) (2018-05-30)

## Pipes

- Add the `keyvalue` pipe introduced in Angular 6.1 (2018-07-26)
- Show usage of formatting functions available since Angular 6.0 (2018-06-15)

## Styling components and encapsulation

- New `ShadowDom` encapsulation option with Shadow DOM v1 support (the old and soon deprecated `Native` option uses Shadow DOM v0) (2018-07-26)

## Send and receive data with Http

- HTTP tests now use `verify` every time (2018-07-06)

## Router

- Adds the `Scroll` event and `scrollPositionRestoration` option introduced in 6.1 (2018-07-26)

## Advanced observables

- Use `shareReplay` instead of `publishReplay` and `refCount` (2018-07-20)

## Internationalization

- Update for CLI 6.0 and use a dedicated configuration (2018-05-09)

# A.32. v6.0.0 - 2018-05-04

## Global

- Bump to ng `6.0.0` (2018-05-04)
- Bump to ng `6.0.0-rc.4` (2018-04-13)
- Bump to ng `6.0.0-rc0` (2018-04-05)
- Bump to ng `6.0.0-beta.7` (2018-04-05)
- Bump to ng `6.0.0-beta.6` (2018-04-05)
- Bump to ng `6.0.0-beta.1` (2018-04-05)

## The wonderful world of Web Components

- Replace `customelements.io` by `webcomponents.org` (2018-01-19)

## From zero to something

- Bump to cli `6.0.0` (2018-05-04)
- The chapter now uses Angular CLI from the start! (2018-03-19)

## Dependency Injection

- Use `providedIn` to register services, as recommended for Angular 6.0 (2018-04-15)

- Updates the dependency injection via token section with a better example (2018-03-19)

## Reactive Programming

- We now use the pipeable operators introduced in RxJS 5.5 (2018-01-28)

## Services

- Use `providedIn` to register the service, as recommended for Angular 6.0 (2018-04-15)

## Testing your app

- Simplify service unit tests now that they use `providedIn` from ng 6.0 (2018-04-15)

## Advanced components and directives

- Angular 6.0+ allows to type `ElementRef<T>` (2018-04-05)

## Advanced observables

- We now use the imports introduced in RxJS 6.0 (`import { Observable, of } from 'rxjs'`) (2018-04-05)
- We now use the pipeable operators introduced with RxJS 5.5 (2018-01-28)

# A.33. v5.2.0 - 2018-01-10

## Global

- Bump to ng [5.2.0](#) (2018-01-10)
- Bump to ng [5.1.0](#) (2017-12-07)

## Building components and directives

- Better lifecycle explanation (2017-12-13)

## Forms

- Reintroduce the `min` and `max` validators from version 4.2.0, even if they are not available as directives. (2017-12-13)

## Send and receive data with Http

- Remove remaining mentions to the deprecated `HttpModule` and `Http` (2017-12-08)

# A.34. v5.0.0 - 2017-11-02

## Global

- Bump to ng [5.0.0](#) (2017-11-02)
- Bump to ng [5.0.0-rc.5](#) (2017-11-02)

- Bump to ng [5.0.0-rc.3](#) (2017-11-02)
- Bump to ng [5.0.0-rc.2](#) (2017-11-02)
- Bump to ng [5.0.0-rc.0](#) (2017-11-02)
- Bump to ng [5.0.0-beta.6](#) (2017-11-02)
- Bump to ng [5.0.0-beta.5](#) (2017-11-02)
- Bump to ng [5.0.0-beta.4](#) (2017-11-02)
- Bump to ng [5.0.0-beta.1](#) (2017-11-02)
- Bump to ng [4.4.1](#) (2017-09-16)

## Pipes

- Use the new i18n pipes introduced in ng 5.0.0 (2017-11-02)

## Forms

- Add a section on the `updateOn: 'blur'` option for controls and groups introduced in 5.0 (2017-11-02)
- Remove the section about combining template-based and code-based approaches (2017-09-01)

## Send and receive data with Http

- Use object literals for headers and params for the new http client, introduced in 5.0.0 (2017-11-02)

## Router

- Adds ng 5.0 `ChildActivationStart/ChildActivationEnd` to the router events (2017-11-02)

## Internationalization

- Remove deprecated i18n comment with ng 5.0.0 (2017-11-02)
- Show how to load the locale data as required in ng 5.0.0 and uses the new i18n pipes (2017-11-02)
- Placeholders now displays the interpolation in translation files to help translators (2017-11-02)

# A.35. v4.3.0 - 2017-07-16

## Global

- Bump to ng [4.3.0](#) (2017-07-16)
- Bump to ng [4.2.3](#) (2017-06-17)

## Forms

- Remove min/max validators mention, as they have been removed temporarily in ng 4.2.3 (2017-06-17)

## Send and receive data with Http

- Updates the chapter to use the new [HttpClientModule](#) introduced in ng 4.3.0. (2017-07-16)

## Router

- List the new router events introduced in 4.3.0 (2017-07-16)

## Advanced components and directives

- Add a section about [HostBinding](#) (2017-06-29)
- Add a section about [HostListener](#) (2017-06-29)
- New chapter on advanced components, with [ViewChild](#), [ContentChild](#) and [ng-content!](#) (2017-06-29)

# A.36. v4.2.0 - 2017-06-09

## Global

- Bump to ng [4.2.0](#) (2017-06-09)
- Bump to ng [4.1.0](#) (2017-04-28)

## Forms

- Introduce the [min](#) and [max](#) validators from version 4.2.0 (2017-06-09)

## Router

- New chapter on advanced router usage: protected routes with guards, nested routes, resolvers and lazy-loading! (2017-04-28)

## Angular compiler

- Adds a chapter about the Angular compiler and the differences between JiT and AoT. (2017-05-02)

# A.37. v4.0.0 - 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.38. v2.4.4 - 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.39. v2.2.0 - 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.40. v2.0.0 - 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.41. v2.0.0-rc.5 - 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 HttpClientModule 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.42. v2.0.0-rc.0 - 2016-05-06

## 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` (2016-03-19)
- 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 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-

## 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)

## A.43. v2.0.0-alpha.47 - 2016-01-15

### Global

- First public release of the ebook! (2016-01-15)