Become a NINJA with

ANGULAR

ninja squad

Deviens un ninja avec Angular (extrait gratuit)

Ninja Squad

Table des matières

1. Extrait gratuit . . . 1
2. Introduction . . . 2
3. Une rapide introduction à ECMAScript 6 . . . 5
   - 3.1. Transpileur . . . 5
   - 3.2. `let` . . . 6
   - 3.3. Constantes . . . 7
   - 3.4. Raccourcis pour la création d'objets . . . 8
   - 3.5. Affectations déstructurées . . . 9
   - 3.6. Paramètres optionnels et valeurs par défaut . . . 11
   - 3.7. *Rest operator* . . . 12
   - 3.8. Classes . . . 13
   - 3.9. *Promises* . . . 16
   - 3.10. (*arrow functions*) . . . 19
   - 3.11. Async/await . . . 23
   - 3.12. *Set* et *Map* . . . 24
   - 3.13. Template de string . . . 25
   - 3.14. Modules . . . 26
   - 3.15. Conclusion . . . 28
4. Un peu plus loin qu'ES6 . . . 29
   - 4.1. Types dynamiques, statiques et optionnels . . . 29
   - 4.2. Hello TypeScript . . . 30
   - 4.3. Un exemple concret d'injection de dépendance . . . 30
5. Découvrir TypeScript . . . 33
   - 5.1. Les types de TypeScript . . . 33
   - 5.2. Valeurs énumérées (*enum*) . . . 34
   - 5.3. Return types . . . 35
   - 5.4. Interfaces . . . 36
   - 5.5. Paramètre optionnel . . . 36
   - 5.6. Des fonctions en propriété . . . 37
   - 5.7. Classes . . . 37
   - 5.8. Utiliser d'autres bibliothèques . . . 39
   - 5.9. Décorateurs . . . 40
6. TypeScript avancé . . . 43
   - 6.1. readonly . . . 43
   - 6.2. keyof . . . 43
   - 6.3. Mapped type . . . 44
   - 6.4. Union de types et gardien de types . . . 46
7. Le monde merveilleux des Web Components . . . 51

7.1. Le nouveau Monde . . . 51
7.2. Custom elements . . . 52
7.3. Shadow DOM . . . 53
7.4. Template . . . 53
7.5. Les bibliothèques basées sur les Web Components . . . 54
8. La philosophie d'Angular . . . 56
9. Commencer de zéro . . . 60
9.1. Node.js et NPM . . . 60
9.2. Angular CLI . . . 60
9.3. Structure de l'application . . . 61
9.4. Notre premier composant . . . 62
9.5. Notre premier module Angular . . . 64
9.6. Démarrer l'application . . . 66
10. Fin de l'extrait gratuit . . . 68
Annexe A: Historique des versions . . . 69
A.1. Changes since last release - 2019-08-30 . . . 69
A.2. v8.2.0 - 2019-08-01 . . . 69
A.3. v8.1.0 - 2019-07-02 . . . 70
A.4. v8.0.0 - 2019-05-29 . . . 70
A.5. v7.2.0 - 2019-01-09 . . . 71
A.6. v7.1.0 - 2018-11-27 . . . 71
A.7. v7.0.0 - 2018-10-25 . . . 71
A.8. v6.1.0 - 2018-07-26 . . . 72
A.9. v6.0.0 - 2018-05-04 . . . 73
A.10. v5.2.0 - 2018-01-10 . . . 74
A.11. v5.0.0 - 2017-11-02 . . . 74
A.12. v4.3.0 - 2017-07-16 . . . 75
A.13. v4.2.0 - 2017-06-09 . . . 76
A.14. v4.0.0 - 2017-03-24 . . . 76
A.15. v2.4.4 - 2017-01-25 . . . 77
A.16. v2.2.0 - 2016-11-18 . . . 78
A.17. v2.0.0 - 2016-09-15 . . . 78
A.18. v2.0.0-rc.5 - 2016-08-25 . . . 79
A.19. v2.0.0-rc.0 - 2016-05-06 . . . 80
A.20. v2.0.0-alpha.47 - 2016-01-15 . . . 82

Chapitre 1. Extrait gratuit

Ce que tu vas lire ici est un extrait gratuit de notre ebook sur Angular : c'est le début du livre, qui explique son but et son contenu, donne un aperçu d'ECMAScript 6, TypeScript, et des Web Components, décrit la philosophie d'Angular, puis te propose de construire ta première application.

Cet extrait ne demande aucune connaissance préliminaire.

Chapitre 2. Introduction

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Cool, non ?

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

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

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

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

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

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

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

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

NOTE	Cet ebook utilise Angular version 8.2.3 dans les exemples.

NOTE

Angular et versions

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

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

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

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

Chapitre 3. Une rapide introduction à ECMAScript 6

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

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

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

3.1. Transpileur

ES6 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 ES6 ont hâte de l'utiliser dans leurs applications, la communauté a trouvé une solution : un transpileur.

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

- Traceur, un projet Google, 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, on aurait donc tendance à te le conseiller. Le projet est maintenant le standard de-facto.

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

3.2. `let`

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

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

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

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

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

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

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

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

3.3. Constantes

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

```
const poniesInRace = 6;
```

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

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

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

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

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

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

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

Même chose avec les tableaux :

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

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

3.4. Raccourcis pour la création d'objets

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

Exemple :

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

peut être simplifié en :

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

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!');
    }
  };
}
```

3.5. Affectations déstructurées

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

En ES5 :

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

Maintenant, en ES6, tu peux écrire :

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

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

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

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

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

Et la même chose est possible avec des tableaux :

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

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

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

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

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

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

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

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

Et tu auras seulement une variable `pony`.

3.6. Paramètres optionnels et valeurs par défaut

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

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

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

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

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

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

Cela fonctionnait, mais ce n'était pas évident de savoir que les paramètres étaient optionnels, sauf à lire le corps de la fonction. ES6 offre désormais une façon plus formelle de déclarer des paramètres optionnels, dès la déclaration de la fonction :

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

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

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

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

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

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

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

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
```

3.7. *Rest operator*

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

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

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

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

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

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

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

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

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

Le *rest operator* ne doit pas être confondu avec le *spread operator* ("opérateur d'étalement"), même si, on te l'accorde, ils se ressemblent dangereusement ! Le *spread operator* est son opposé : il prend un tableau, et l'étale en arguments variables. Le seul cas d'utilisation qui me vient à l'esprit serait pour les fonctions comme `min` ou `max`, qui peuvent recevoir des arguments variables, et que tu voudrais appeler avec un tableau :

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

3.8. Classes

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

n'était pas une tâche aisée, surtout pour les débutants…

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

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

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

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

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

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

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

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

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

```
class Pony {
  get color() {
    console.log('get color');
    return this._color;
  }

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

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

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

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

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

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

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

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

3.9. *Promises*

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

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

Avec des *callbacks* :

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

Avec des *promises* :

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

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

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

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

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

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

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

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

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

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

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

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

ou plus élégamment :

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

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

Une gestion d'erreur par *promise* :

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

Une gestion d'erreur globale pour toute la chaîne :

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

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

3.10. (*arrow functions*)

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

Prenons notre exemple précédent avec des *promises* :

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

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

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

N'est-ce pas super cool ?!

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

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

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

En ES5 :

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

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

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

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

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

ou en *bindant* le `this` :

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

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

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

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

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

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

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

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

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

3.11. 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 retournée 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.

3.12. *Set* et *Map*

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

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

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

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

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

Tu peux aussi itérer sur une collection, avec la nouvelle syntaxe `for ⋯ of` :

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

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

3.13. 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 ', '! How are you', '?']
  // `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.

3.14. 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.

ES6 a pour objectif de créer une syntaxe avec le meilleur des deux mondes, sans se préoccuper de l'implémentation utilisée. Le comité Ecma TC39 (qui est responsable des évolutions d'ES6 et auteur de la spécification du langage) voulait une syntaxe simple (c'est indéniablement l'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.

3.15. Conclusion

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

Chapitre 4. Un peu plus loin qu'ES6

4.1. Types dynamiques, statiques et optionnels

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

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

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

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

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

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

Cela nous amène au sujet de la maintenabilité. Le code JS peut être difficile à maintenir, malgré les tests et la documentation. Refactoriser une grosse application JS n'est pas chose aisée, comparativement à ce qui peut être fait dans des langages statiquement typés. La maintenabilité est un sujet important, et les types aident les outils, ainsi que les développeurs, à éviter les erreurs lors de l'écriture et la modification de code. Google a toujours été enclin à proposer des solutions dans cette direction : c'est compréhensible, étant donné qu'ils gèrent des applications parmi les plus grosses du monde, avec GMail, Google apps, Maps… Alors ils ont essayé plusieurs approches pour améliorer la maintenablité 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'ES6 avec des annotations (des annotations de type et d'autres, dont je parlerai plus tard). Ils annoncèrent ensuite le support de TypeScript, le langage de Microsoft, avec des annotations de type additionnelles. Et enfin, quelques mois plus tard, l'équipe TypeScript annonçait, après un travail étroit avec l'équipe de Google, que la nouvelle version du langage (1.5) aurait toutes les nouvelles fonctionnalités d'AtScript. L'équipe Angular déclara alors qu'AtScript était officiellement abandonné, et que TypeScript était désormais la meilleure façon d'écrire des applications Angular !

4.2. Hello TypeScript

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

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

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

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

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

Il est temps désormais de se lancer dans TypeScript !

Chapitre 5. Découvrir TypeScript

TypeScript, qui existe depuis 2012, est un sur-ensemble de JavaScript, ajoutant quelques trucs à ES5. Le plus important étant le système de type, lui donnant même son nom. Depuis la version 1.5, sortie en 2015, cette bibliothèque essaie d'être un sur-ensemble d'ES6, incluant toutes les fonctionnalités vues précédemment, et quelques nouveautés, comme les décorateurs. Ecrire du TypeScript ressemble à écrire du JavaScript. Par convention les fichiers sources TypeScript ont l'extension `.ts`, et seront compilés en JavaScript standard, en général lors du build, avec le compilateur TypeScript. Le code généré reste très lisible.

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

Mais commençons par le début.

5.1. Les types de TypeScript

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

```
let variable: type;
```

Les différents types sont simples à retenir :

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2. 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';
```

5.3. 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;
}
```

5.4. 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 évidement 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 };
```

5.5. 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;
}
```

5.6. Des fonctions en propriété

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

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

La définition de cette interface serait :

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

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

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

5.7. Classes

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

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

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

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

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

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

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

Et une interface peut en étendre une ou plusieurs autres :

```
interface Animal extends CanRun, CanEat {}

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

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

```
class SpeedyPony {
  speed = 10;

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

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

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

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

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

Ce qui est l'équivalent du plus verbeux :

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

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

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

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

5.8. Utiliser d'autres bibliothèques

Mais si on travaille avec des bibliothèques externes écrites en JS, comment savoir les types des

paramètres attendus par telle fonction de telle bibliothèque ? La communauté TypeScript est tellement cool que ses membres ont défini des interfaces pour les types et les fonctions exposés par les bibliothèques JavaScript les plus populaires.

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

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

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

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

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

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

5.9. 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 méta-programmation. Ils ressemblent beaucoup aux annotations, qui sont principalement utilisées en Java, C#, et Python, et peut-être d'autres langages que je ne connais pas. Suivant le langage, tu peux ajouter une annotation sur une méthode, un attribut, ou une classe. Généralement, les annotations ne sont pas vraiment utiles au langage lui-même, mais plutôt aux frameworks et aux bibliothèques.

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'existait 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 (possiblement avec ES7/ES2016).

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

En TypeScript, les annotations sont préfixées par `@`, et peuvent être appliquées sur une classe, une propriété de classe, une fonction, ou un paramètre de fonction. Pas sur un constructeur en revanche, mais sur ses paramètres oui.

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

Il s'utilisera comme ça :

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

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

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

```
const Log = () => {
  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 "ES6+", mais tu ne pourras pas bénéficier des types, et ils sont sacrément utiles pour l'injection de dépendances. C'est possible, mais tu devras ajouter d'autres décorateurs pour remplacer les informations de type.

Ainsi mon conseil est d'essayer TypeScript. Tous mes exemples dans ce livre l'utiliseront à partir de maintenant, car Angular, et tout l'outillage autour, est vraiment conçu pour en tirer parti.

Chapitre 6. 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 que si le langage t'est déjà familier.

6.1. 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
```

6.2. 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 en 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) {
  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.

6.3. 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.

6.3.1. 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`
```

6.3.2. 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
```

6.3.3. 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.

6.4. 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 :

```ts
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.

```ts
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.

```ts
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 !

```ts
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.

Chapitre 7. 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.

7.1. Le nouveau Monde

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

Mes amis, cette heure est venue.

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

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

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

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

Ce standard émergent est défini dans 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.

7.2. Custom elements

Les éléments customs sont un nouveau standard qui permet au développeur de créer ses propres éléments du DOM, faisant de `<ns-pony></ns-pony>` un élément HTML parfaitement valide. La spécification définit comment déclarer de tels éléments, comment tu peux les faire étendre des éléments existants, comment tu peux définir ton API, etc.

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

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

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

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

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

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

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

7.3. Shadow DOM

Avec un nom qui claque comme celui-là, on s'attend à un truc très puissant. Et il l'est. Le Shadow DOM est une façon d'encapsuler le DOM de ton composant. Cette encapsulation signifie que la feuille de style et la logique JavaScript de ton application ne vont pas s'appliquer sur le composant et le ruiner accidentellement. Cela en fait l'outil idéal pour dissimuler le fonctionnement interne de ton composant, et s'assurer que rien n'en fuit à l'extérieur.

Si on retourne à notre exemple précédent :

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

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

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

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

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

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

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

7.4. Template

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

Pour utiliser un template, il doit être cloné :

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

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

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

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

7.5. 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 webcomponent.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/.

Polymer (et les autres) permet aussi de créer tes propres composants :

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

et de les utiliser :

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

Tu peux faire plein de trucs cools avec Polymer 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.

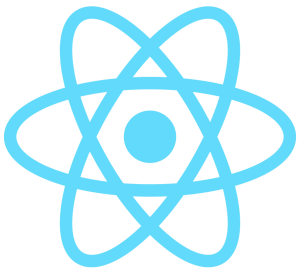
C'est un chapitre trop court pour te montrer sérieusement tout ce que l'on peut faire avec les Web Components, mais tu verras que certains de leurs concepts vont émerger dans la lecture à venir. Et tu verras sans aucun doute que l'équipe Google a conçu Angular pour rendre facile l'utilisation des Web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

Chapitre 8. La philosophie d'Angular

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

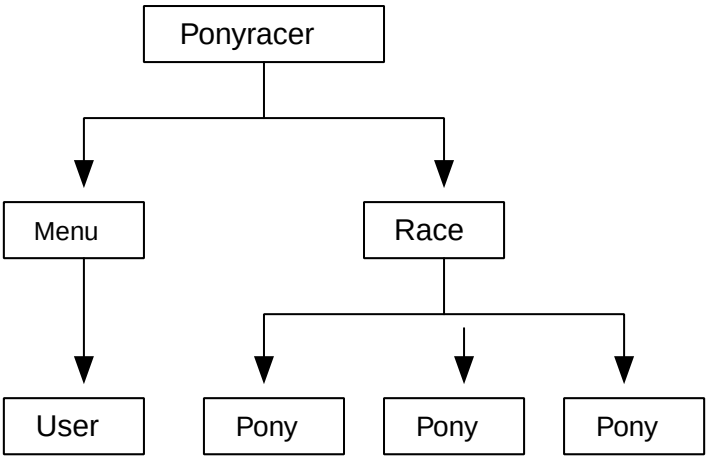
Avant tout, Angular est un framework orienté composant. Tu vas écrire de petits composants, et assemblés, ils vont constituer une application complète. Un composant est un groupe d'éléments HTML, dans un template, dédiés à une tâche particulière. Pour cela, tu auras probablement besoin d'un peu de logique 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 / controlleur", ou une directive.

Il faut aussi dire qu'un standard a été défini autour de ces composants : le standard *Web Component* ("composant web"). Même s'il n'est pas encore complètement supporté dans les navigateurs, tu peux déjà construire des petits composants isolés, réutilisables dans différentes applications (ce vieux rêve de programmeur). Cette orientation composant est largement partagée par de nombreux frameworks front-end : c'est le cas depuis le début de 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 : ES6 (ou ES2015, si tu préfères). Ainsi tu peux écrire tes composants en ES5 (pas cool !) ou en ES6 (super cool !). Mais cela ne leur suffisait pas, ils voulaient utiliser une

fonctionnalité qui n'est pas encore standard : les décorateurs. Alors ils ont travaillé étroitement avec les équipes de transpileurs (Traceur et Babel) et l'équipe Microsoft du projet TypeScript, pour nous permettre d'utiliser des décorateurs dans nos applications Angular. Quelques décorateurs sont disponibles, permettant de déclarer facilement un composant et sa vue. J'espère que tu es au courant, parce que je viens de consacrer deux chapitres à ces sujets !

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

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

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

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

Et le template pourrait ressembler à ça :

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

Si tu connais déjà AngularJS 1.x, le template doit t'être familier, avec les 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 ES6).

Tu pourras aussi prendre des modules sur étagère fournis par la communauté, et les ajouter simplement dans ton application pour bénéficier de leurs fonctionnalités.

De tels modules 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, ton premier module, et la syntaxe des templates.

Il y a un autre concept au cœur d'Angular : l'injection de dépendance (*Dependency Injection*, DI). C'est un pattern très puissant, et tu seras très rapidement séduit après la lecture du chapitre qui lui sera consacré. C'est particulièrement utile pour tester ton application, et j'adore faire des tests, et voir la barre de progression devenir entièrement verte dans mon IDE. Ça me donne l'impression de faire du bon boulot. Il y aura ainsi un chapitre entier consacré à tout tester : tes composants, tes services, ton interface…

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

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

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

Chapitre 9. Commencer de zéro

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

9.1. Node.js et NPM

Aujourd'hui, pratiquement tous les outils JavaScript moderne 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`).

9.2. 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 teeeeeeellement 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 !

MISE EN PRATIQUE	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 générons une nouvelle application avec la commande `ng new`. Si tu veux utiliser exactement la même version de la CLI que nous (`8.3.2`), tu peux utiliser `npm install -g @angular/cli@8.3.2` à la place.

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

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

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 rechargement 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 ! 🎉

9.3. 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, Atom, VisualStudio Code… Ils ont tous un bon support de TypeScript.

NOTE	Si ton IDE le supporte, la complétion de code devrait fonctionner car la dépendance Angular a ses propres fichiers `d.ts` dans le répertoire `node_modules`, et TypeScript est capable de le détecter. Tu peux même naviguer vers les définitions de type si tu le souhaites. TypeScript apporte sa vérification de types, donc tu verras les erreurs dès que tu les tapes. Comme nous utilisons des *source maps*, tu peux voir le code TypeScript directement dans ton navigateur, et même débugger ton application en positionnant des points d'arrêt directement dedans.

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 dans `src` 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*), et on veut que notre code soit transpilé en ECMAScript 5, lui permettant d'être exécuté par tout navigateur. L'option `sourceMap` permet de générer les *source maps* ("dictionnaires de code source"), c'est-à-dire des fichiers assurant le lien entre le code ES5 généré et le code TypeScript originel. Ces *source maps* sont utilisés par le navigateur pour te permettre de débugger le code ES5 qu'il exécute en parcourant le code TypeScript originel que tu as écrit.

Les projets TypeScript sont fréquemment utilisés avec TSLint, un *linter*, pratique pour vérifier si ton code suit un ensemble de bonnes pratiques. TSLint a ses propres options, stockées dans `tslint.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.

Le dernier, `karma.conf.js`, est utilisé pour configurer les tests (on en reparlera dans un chapitre consacré aux tests).

NOTE

L'ebook utilise Angular version `8.2.3` 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 `8.2.3` 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.

9.4. Notre premier composant

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 une seule propriété est requise : `selector`. Elle indiquera à Angular ce qu'il faudra chercher dans nos pages HTML. A 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`.

NOTE

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 **N**inja **S**quad, 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.

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>
```

9.5. Notre premier module Angular

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

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

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

Bien sûr, la CLI a généré un module également, appelé `app.module.ts`.

Cette classe est décorée avec `@NgModule`.

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

@NgModule({
})
export class AppModule { }
```

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

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

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

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

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

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

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

import { AppComponent } from './app.component';

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

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

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

import { AppComponent } from './app.component';

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

Notre module est prêt, démarrons l'application !

9.6. Démarrer l'application

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

pour le moment.

Angular CLI génère par défaut un fichier séparé contenant cette logique de démarrage : `main.ts`.

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

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 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.

Chapitre 10. Fin de l’extrait gratuit

Voilà, j’espère que ce que tu as lu t’aura comblé. Si tu rêves désormais de lire la suite (tu devrais !), va l’acheter sur le site du livre ! :)

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: `8.2.3`
- Angular CLI: `8.3.2`

A.1. Changes since last release - 2019-08-30

A gentle introduction to ECMAScript 6

- 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 `8.3.2` (2019-08-30)
- Bump to cli `8.3.0` (2019-08-22)

A.2. 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.3. 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.4. v8.0.0 - 2019-05-29

Global

- Bump to ng `8.0.0` (2019-05-29)

A gentle introduction to ECMAScript 6

- 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.5. 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.6. 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.7. 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.8. 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.9. 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.10. 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.11. 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.12. 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.13. 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.14. 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 mulitple async pipe subscriptions problem (2017-03-23)

Internationalization

- Add a new chapter on internationalization (i18n) (2017-03-23)

A.15. 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.16. 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.17. 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.18. 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 HttpModule import (2016-08-21)
- `http.post()` now autodetects the body type, removing the need of using `JSON.stringify` and setting the `ContentType` (2016-06-16)

Router

- Introduce RouterModule (2016-08-21)
- Update the router to the API v3! (2016-07-08)
- Warn about router module being rewritten (and deprecated) (2016-06-16)

Changelog

- Mention free updates and web page for obtaining latest version (2016-07-25)

A.19. 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 uses `let` instead of `#` to declare a variable `*ngFor="let pony of ponies" [small]`(2016-05-03)#
- `*ngFor` now also exports a `first` variable (2016-04-16)

Dependency Injection

- Better explanation of hierarchical injectors (2016-03-04)

Pipes

- A `replace` pipe has been introduced (2016-04-16)

Reactive Programming

- Observables are not scheduled for ES7 anymore (2016-03-04)

Building components and directives

- Explain how to remove the compilation warning when using `@Input` and a setter at the same time (2016-03-04)
- Add an explanation on `isFirstChange` for `ngOnChanges` (2016-03-04)

Testing your app

- `injectAsync` is now deprecated and replaced by `async` (2016-05-03)
- Add an example on how to test an event emitter (2016-03-04)

Forms

- A pattern validator has been introduced to make sure that the input matches a regexp (2016-04-16)
- Add a mnemonic tip to rememeber the `[()]` syntax: the banana box! (2016-03-04)
- Examples use `module.id` to have a relative `templateUrl` (2016-03-04)
- Fix error `ng-no-form` → `ngNoForm` (2016-03-04)
- Fix errors `(ngModel)` → `(ngModelChange)`, `is-old-enough` → `isOldEnough` (2016-03-04)

Send and receive data with Http

- Use `JSON.stringify` before sending data with a POST (2016-03-04)
- Add a mention to `JSONP_PROVIDERS` (2016-03-04)

Router

- Introduce the new router (previous one is deprecated), and how to use parameters in URLs! (2016-05-06)
- `RouterOutlet` inserts the template of the component just after itself and not inside itself (2016-03-04)

Zones and the Angular magic

- New chapter! Let's talk about how Angular 2 works under the hood! First part is about how AngularJS 1.x used to work, and then we'll see how Angular 2 differs, and uses a new concept called zones. (2016-05-03)

A.20. v2.0.0-alpha.47 - 2016-01-15

Global

- First public release of the ebook! (2016-01-15)