ANGULAR

SUPPORT DE FORMATION "REALISEZ UNE APPLICATION WEB AVEC ANGULAR"

Découvrez les enjeux autour d'Angular

Les frameworks JS

Aux débuts du développement web, seul le HTML permet de créer des sites. Les sites web en 1990 ressemblent à des documents texte, car il n’existe pas vraiment d’autres possibilités. Ensuite, en 1998, le CSS arrive pour aider à créer des mises en page plus esthétiques. Finalement, à partir du début des années 2000, le JavaScript commence son règne sur le développement web, permettant des interactions entre l’utilisateur et la page.

Certains noteront que je ne parle pas de l’époque Flash ici : je pense qu’il est préférable, pour la santé mentale de tous, d'oublier totalement

l'existence de cette technologie qui n'était que très rarement bien exploitée, qui a créé bug sur bug sur bug et qui était remplie de problèmes de sécurité. Passons.

À partir de 2005, le système AJAX (Asynchronous Javascript And XML) permet des interactions entre l'utilisateur et des backend HTTP : il est enfin possible d'échanger des informations et de générer du contenu à partir de ces interactions.

En 2010, la première version d'AngularJS est lancée. Elle permet de créer plus facilement des Single Page Applications, des applications web qui imitent les applications natives : pas de rafraîchissement du navigateur, temps de chargement réduits, une UI beaucoup moins "internet" etc. Cette version permet déjà de faire énormément de choses, mais souffre d'une syntaxe plutôt complexe ainsi que des limitations du JavaScript. Voilà pourquoi Google choisit de complètement réécrire le framework pour sa version 2. Aujourd'hui, nous en sommes à Angular 5.x (maintenant appelé simplement "Angular") ; la version 3 ayant été sautée pour des raisons sémantiques tout simplement.

Pourquoi Angular ?

Il y a plusieurs frameworks JavaScript très populaires aujourd'hui : Angular, React, Ember, Vue… les autres frameworks marchent très bien, ont beaucoup de succès et sont utilisés sur des sites extrêmement bien fréquentés, React et Vue notamment. Angular présente également un niveau de difficulté légèrement supérieur, car on utilise le TypeScript plutôt que JavaScript pur ou le mélange JS/HTML de React. Ainsi, quels sont donc les avantages d'Angular ?

- Angular est géré par Google — il y a donc peu de chances qu'il disparaisse, et l'équipe de développement du framework est excellente.

- Le TypeScript — ce langage permet un développement beaucoup plus stable, rapide et facile.

- Le framework Ionic — le framework permettant le développement d'applications mobiles multi-plateformes à partir d'une seule base de code — utilise Angular.

Les autres frameworks ont leurs avantages également, mais Angular est un choix très pertinent pour le développement frontend.

Qu'est-ce que le TypeScript ? Pourquoi l'utiliser ?

Pour faire bref, le TypeScript est un sur-ensemble (un "superset") de JavaScript qui est justement transcompilé (transcompilation : "traduction" d'un langage de programmation vers un autre - différent de la compilation, qui transforme généralement le code vers un format exécutable) en JavaScript pour être compréhensible par les navigateurs. Il ajoute des fonctionnalités extrêmement utiles, comme, entre autres :

- le typage strict, qui permet de s'assurer qu'une variable ou une valeur passée vers ou retournée par une fonction soit du type prévu ;
- les fonctions dites lambda ou arrow, permettant un code plus lisible et donc plus simple à maintenir ;
- les classes et interfaces, permettant de coder de manière beaucoup plus modulaire et robuste.

Ceci n'est pas un cours sur le TypeScript ! Vous allez utiliser des fonctionnalités natives de ce langage tout au long de ce cours ; si vous souhaitez en savoir plus, ou s'il y a des commandes ou des façons de coder que vous ne comprenez pas, n'hésitez pas à consulter la documentation officielle.

Installez les outils et créez votre projet

Préparez l'environnement de développement

Qu’est-ce que le CLI ?

Le CLI, ou “Command Line Interface” (un outil permettant d'exécuter des commandes depuis la console), d’Angular est l’outil qui vous permet d’exécuter des scripts pour la création, la structuration et la production d’une application Angular.

Pour accéder à la ligne de commande :

- Sur Windows : si vous souhaitez utiliser l'invite de commande native de Windows, il vous suffit d'ouvrir la fenêtre Exécuter en faisant

Windows+R et en entrant cmd. Sinon, il existe des utilitaires natifs comme PowerShell ou des tiers comme PowerCmd ou ConEmu par exemple. Toutes les commandes citées dans le cours devraient fonctionner si vous êtes authentifié comme administrateur sur votre machine.

- Sur Mac : il suffit de lancer Terminal depuis Spotlight ou le Launchpad.

- Sur Linux : lancez le terminal de votre choix.

Installez les outils

Vous devez installer les outils suivants si vous ne les avez pas déjà sur votre machine :

NODE.JS
Téléchargez et installez la dernière version LTS de Node.js ici : https://nodejs.org/en/download/

NPM
NPM est un package manager qui permet l'installation d'énormément d'outils et de libraries dont vous aurez besoin pour tout type de développement. Pour l'installer, ouvrez une ligne de commande et tapez la commande suivante :

```
npm install -g npm@latest
```

Si la commande échoue pour un problème de permission sur Mac ou Linux, vous pouvez la lancer en mode super-utilisateur avec *sudo* ou consulter la documentation suivante.

ANGULAR/CLI
Vous allez maintenant installer le CLI d'Angular de manière globale sur votre machine avec la commande suivante (avec sudo si besoin) :

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

À partir de là, la commande `ng` est disponible depuis la ligne de commandes depuis n'importe quel dossier de l'ordinateur.

Pour plus d'informations, vous pourrez trouver la documentation de chaque élément sur le site correspondant :

- node.js : https://nodejs.org/en/docs/
- npm : https://docs.npmjs.com/
- ng : https://cli.angular.io/

Créez votre premier projet

Pour créer un nouveau projet Angular, naviguez vers le dossier souhaité depuis une ligne de commande et saisissez la commande suivante :

```
ng new mon-premier-projet
```

Ensuite, naviguez dans le dossier du projet et lancez le serveur de développement :

```
cd mon-premier-projet
ng serve --open
```

Si tout se passe bien, vous verrez les informations du serveur qui se lance à l'adresse *localhost:4200* et votre navigateur préféré se lancera automatiquement avec le message "Welcome to app!!" et le logo Angular.

Félicitations, votre environnement de développement est prêt !

Quelques conseils pour le développement

Pour développer, je vous conseille d'utiliser un éditeur comme Atom ou Sublime Text, ou un IDE comme VS Code ou WebStorm. Pour les screencast et screenshot, j'utiliserai WebStorm, qui comporte beaucoup de

fonctionnalités prévues spécifiquement pour le développement JavaScript/TypeScript, donc ne vous inquiétez pas si les couleurs ou les visuels dans votre éditeur ne ressemblent pas exactement à ceux que je vous montre.

Je vous conseille d'utiliser Chrome comme navigateur par défaut pour le développement, car les Developer Tools sont très complets, et vous avez également la possibilité d'installer Augury, un plugin Chrome spécifique pour le développement Angular.

Pour la suite du cours, j'utiliserai les styles en SCSS plutôt qu'en CSS simple : cela apporte pas mal d'avantages (création de variables, utilisations de mixins pour un code plus modulaire etc), et est intégré par le CLI. Ne vous inquiétez pas si vous ne maîtrisez pas la syntaxe SCSS : tout code CSS valable est aussi valable en SCSS.

Vous utiliserez également Bootstrap pour les styles de votre application pour faciliter la mise en page simple que vous effectuerez. Ce n'est bien sûr pas obligatoire, mais puisque le cours se concentre ici sur Angular, je ne parlerai pas de l'esthétique ou de la qualité générale de l'UI.

Dans la partie suivante, nous allons découvrir les éléments fondamentaux d'un projet Angular.

Structurez avec les components

Préparez le projet

Tout d'abord, créez un nouveau projet sur lequel vous travaillerez tout au long des chapitres suivants. Depuis une ligne de commandes, naviguez vers votre dossier cible et tapez la commande suivante :

```
ng new mon-projet-angular --style=scss --skip-tests=true.
```

Le premier flag (élément de la commande suivant un double tiret --) crée des fichiers `.scss` pour les styles plutôt que des fichiers `.css`. Le second flag annule la création des fichiers test.

Les unit tests sortent du périmètre de ce cours (on pourrait en faire un cours entier !), mais il y a énormément de ressources en ligne vous

permettant de comprendre le concept du unit testing si vous souhaitez en apprendre plus.

Maintenant vous pouvez ouvrir le dossier `mon-projet-angular` depuis votre éditeur.

Avant de plonger dans les différents dossiers, vous allez exécuter une commande pour installer Bootstrap dans votre projet. Depuis le dossier `mon-projet-angular` , avec une ligne de commande, tapez :

```
npm install bootstrap@3.3.7 --save.
```

Cette commande téléchargera Bootstrap et l'intégrera dans le `package.json` du projet. Il vous reste une dernière étape pour l'intégrer à votre projet. Ouvrez le fichier `.angular-cli.json` du dossier source de votre projet. Dans "apps", modifiez l'array `styles` comme suit :

```
"styles": [
    "../node_modules/bootstrap/dist/css/bootstrap.css",
    "styles.scss"
]
```

Maintenant vous pouvez lancer le serveur de développement local (il faudra le couper avec Ctrl-C et le relancer s'il tournait déjà pour que les changements prennent effet) :

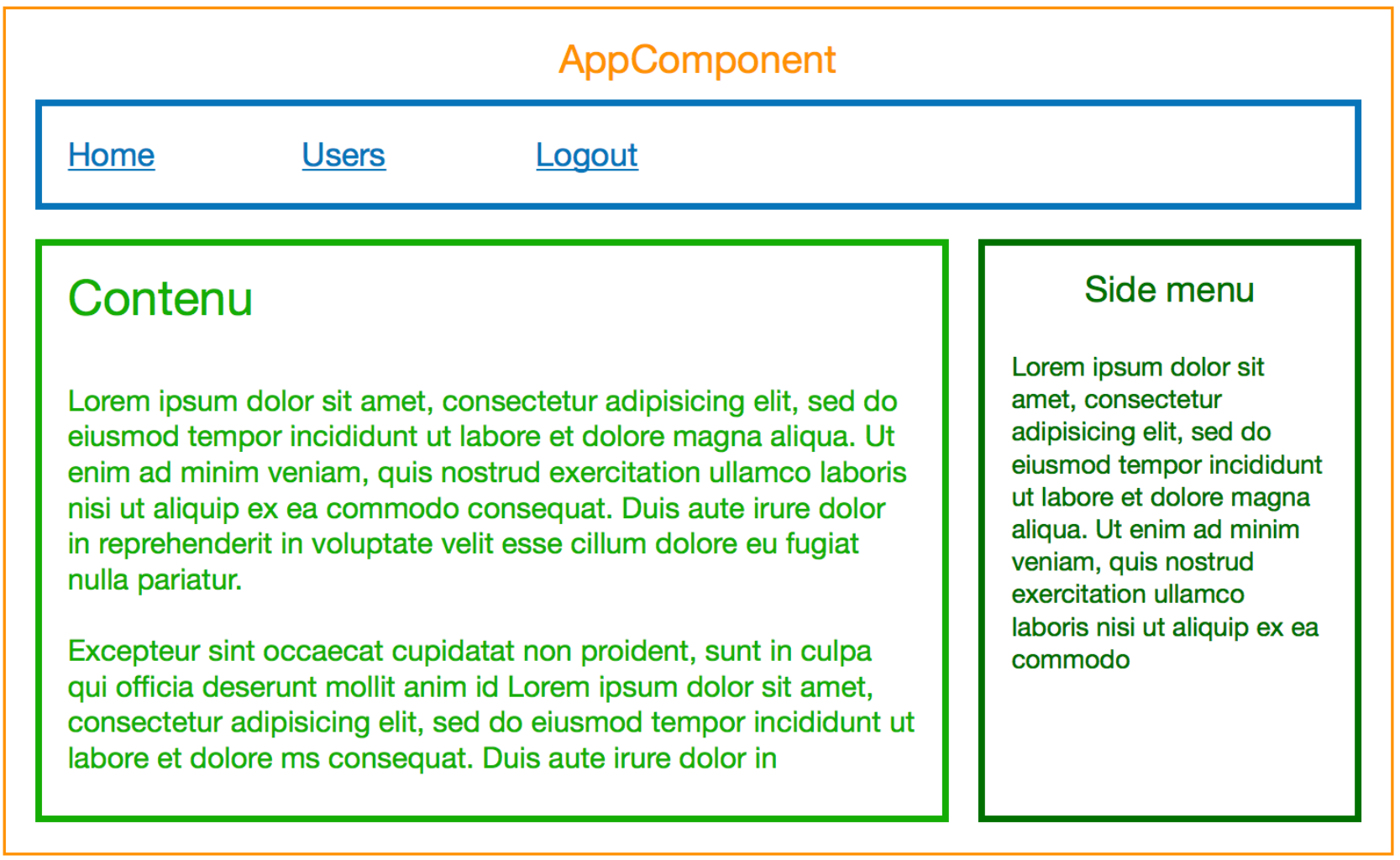
```
ng serve
```

Une fois le projet compilé, ouvrez votre navigateur à `localhost:4200` et vous y verrez l'application telle qu'elle a été créée par le CLI, avec les styles Bootstrap appliqués. Vous pouvez laisser tourner le serveur de développement en fond. À chaque fois que vous enregistrez des modifications au projet, le CLI met à jour le serveur : nul besoin de le redémarrer vous-même à chaque changement.

La structure des components d'une application Angular

Les components sont les composantes de base d'une application Angular : une application est une arborescence de plusieurs components.

Imaginez la page web suivante :



Tout d'abord, notre `AppComponent` est notre component principal : tous les autres components de notre application seront emboîtés ou "nested" dans celui-ci.

Pour cette structure, on peut imaginer un component pour la barre de menu, un autre pour la partie contenu et un dernier pour le menu à droite. Il n'y a pas de règle d'or : je vous montrerai au cours des chapitres suivants comment réfléchir à votre structure afin de trouver la séparation des components la plus pertinente.

Découvrez la structure du code

Le CLI crée énormément de fichiers au moment de la création d'une

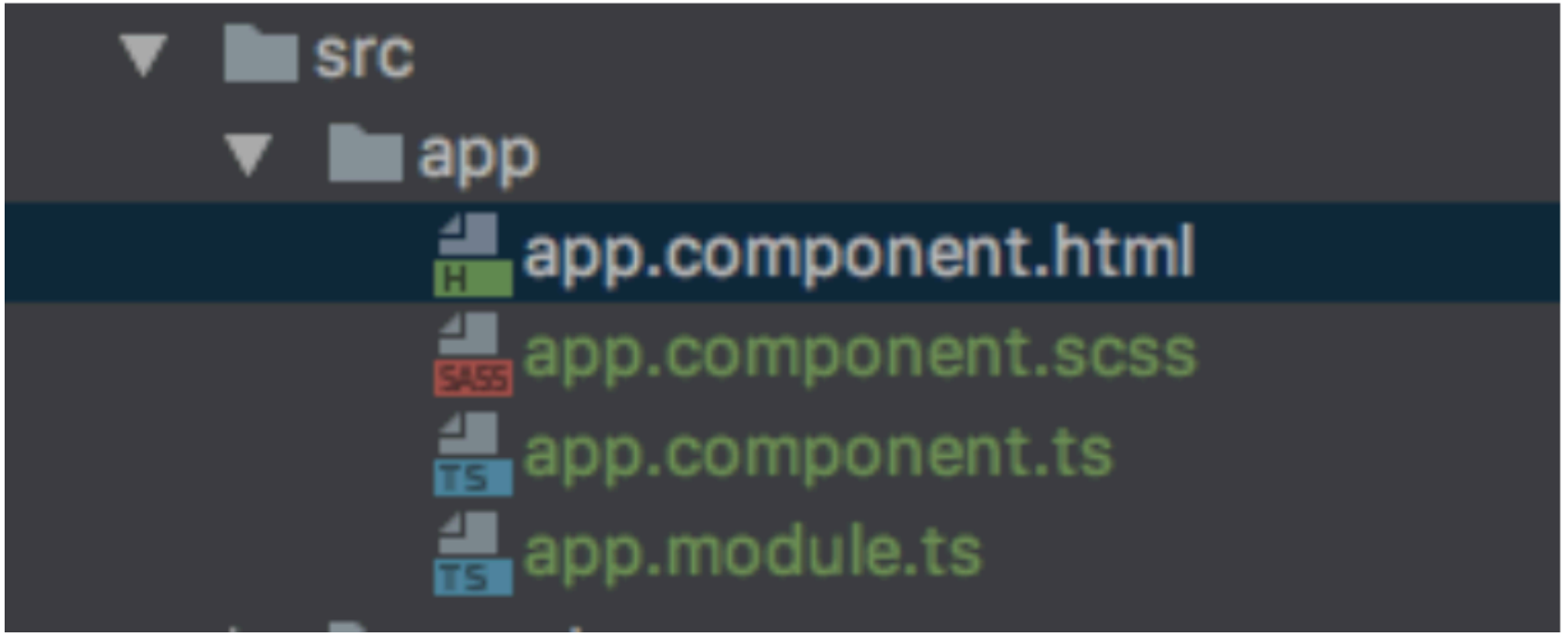
nouvelle application. Le dossier `e2e` est généré pour les tests end-to-end, un sujet que je n'aborderai pas dans ce cours mais pour lequel vous trouverez facilement des ressources en ligne. Ensuite, le dossier `node_modules` contient toutes les dépendances pour votre application : les fichiers source Angular et TypeScript, par exemple.

Le dossier qui vous intéressera principalement est le dossier `src` , où vous trouverez tous les fichiers sources pour votre application.

Pour commencer à comprendre la structure d'une application Angular, ouvrez `index.html` dans votre éditeur :

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>MonProjetAngular</title>
    <base href="/">

    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
    <app-root></app-root>
</body>
</html>
```

Comme vous pouvez le constater, au lieu d'y voir tout le contenu que nous voyons dans le navigateur, il n'y a que cette balise vide `<app-root>` : il s'agit d'une balise Angular. Pour en savoir plus, ouvrez le dossier `app` :



Ce dossier contient le module principal de l'application et les trois fichiers du component principal `AppComponent` : son template en HTML, sa feuille de styles en SCSS, et son fichier TypeScript, qui contiendra sa logique.

Ouvrez d'abord le fichier `app.component.html` :

```
<!--The content below is only a placeholder and can be replaced.-->
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <img width="300" alt="Angular Logo" src="
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/tutorial
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://github.com/angular/
  </li>
  <li>
```

```
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">A
  </li>
</ul>
```

Ici, vous voyez le code HTML correspondant à ce que vous voyez dans votre navigateur.

Donc comment fait Angular pour injecter ce code dans la balise <app-root> ?

Regardez maintenant dans le fichier `app.component.ts` :

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


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

Ici, à l'intérieur du décorateur `@Component()` , vous trouvez un objet qui contient les éléments suivants :

- `selector` : il s'agit du nom qu'on utilisera comme balise HTML pour afficher ce component, comme vous l'avez vu avec `<app-root>` . Ce nom doit être unique et ne doit pas être un nom réservé HTML de type `<div>` , `<body>` etc. On utilisera donc très souvent un préfixe comme `app` , par exemple ;

- `templateUrl` : le chemin vers le code HTML à injecter ;

- `styleUrls` : un array contenant un ou plusieurs chemins vers les feuilles de styles qui concernent ce component ;

Quand Angular rencontre la balise `<app-root>` dans le document HTML, il sait qu'il doit en remplacer le contenu par celui du template `app.component.html` , en appliquant les styles `app.component.scss` , le tout géré par la logique du fichier `app.component.ts` . Nous verrons ces interactions plus en détail dans les chapitres suivants. Pour faire un premier test, je vous propose de modifier la variable `title` dans `app.component.ts` , d'enregistrer le fichier, et de regarder le résultat dans votre navigateur.

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


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

Créez un component

Vous allez maintenant créer un nouveau component à l'aide du CLI d'Angular. Depuis le dossier principal de votre projet, tapez la commande suivante :

```
ng generate component mon-premier
```

```
installing component
  create src/app/mon-premier/mon-premier.component.scss
  create src/app/mon-premier/mon-premier.component.html
  create src/app/mon-premier/mon-premier.component.spec.ts
  create src/app/mon-premier/mon-premier.component.ts
  update src/app/app.module.ts
```

Comme vous le constaterez, le CLI a créé un nouveau sous-dossier `mon-premier` et y a créé un fichier template, une feuille de styles, un fichier component et un fichier spec : il s'agit d'un fichier de test que vous pouvez supprimer, car vous ne vous en servirez pas dans le cadre de ce cours.

Le CLI nous prévient également qu'il a mis à jour le fichier `app.module.ts` : ouvrez-le maintenant pour voir de quoi il s'agit :

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

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


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

import { MonPremierComponent } from './mon-premier/mon-premier.component';



@NgModule({

  declarations: [

    AppComponent,

    MonPremierComponent

  ],

  imports: [

    BrowserModule

  ],

  providers: [],
```

```
  bootstrap: [AppComponent]

})

export class AppModule { }
```

Le CLI a ajouté `MonPremierComponent` à l'array `declarations` de votre module. Il a également ajouté le statement import en haut du fichier. Ce sont des étapes **nécessaires** pour que vous puissiez utiliser votre component au sein de votre application Angular.

Regardez maintenant le fichier `mon-premier.component.ts` :

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


@Component({

  selector: 'app-mon-premier',

  templateUrl: './mon-premier.component.html',

  styleUrls: ['./mon-premier.component.scss']

})

export class MonPremierComponent implements OnInit {


  constructor() { }


  ngOnInit() {

  }


}
```

Vous constaterez que le CLI a créé un sélecteur : `app-mon-premier` . Nous pouvons donc utiliser ce sélecteur dans notre code pour y insérer ce component.

Revenez dans `app.component.html` et modifiez-le comme suit :

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
</div>
<app-mon-premier></app-mon-premier>
```

Dans votre navigateur, vous verrez le même titre qu'avant et, sur une deuxième ligne, le texte "mon-premier works". Il s'agit du texte par défaut créé par le CLI que vous trouverez dans `mon-premier.component.html` :

```
<p>
  mon-premier works!
</p>
```

Félicitations, vous avez créé votre premier component Angular !

Gérez des données dynamiques

L'intérêt d'utiliser Angular est de pouvoir gérer le DOM (Document Object Model : les éléments HTML affichés par le navigateur) de manière dynamique, et pour cela, il faut utiliser la liaison de données, ou "databinding".

Le databinding, c'est la communication entre votre code TypeScript et le template HTML qui est montré à l'utilisateur. Cette communication est divisée en deux directions :

- les informations venant de votre code qui doivent être affichées dans le navigateur, comme par exemple des informations que votre code a calculé ou récupéré sur un serveur. Les deux principales méthodes pour cela sont le "string interpolation" et le "property binding" ;

- les informations venant du template qui doivent être gérées par le code : l'utilisateur a rempli un formulaire ou cliqué sur un bouton, et il faut réagir et gérer ces événements. On parlera de "event binding" pour cela.

Il existe également des situations comme les formulaires, par exemple, où l'on voudra une communication à double sens : on parle donc de "two-way binding".

String interpolation

L'interpolation est la manière la plus basique d'émettre des données issues de votre code TypeScript.

Imaginez une application qui vérifie l'état de vos appareils électriques à la maison pour voir s'ils sont allumés ou non. Créez maintenant un nouveau component `AppareilComponent` avec la commande suivante :

```
ng generate component appareil
```

Ouvrez ensuite `appareil.component.html` (dans le nouveau dossier `appareil` créé par le CLI), supprimez le contenu, et entrez le code ci-dessous :

```
<li class="list-group-item">
  <h4>Ceci est dans AppareilComponent</h4>
</li>
```

Ensuite, ouvrez `app.component.html` , et remplacez tout le contenu comme suit :

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
```

```
      <h2>Mes appareils</h2>
      <ul class="list-group">
        <app-appareil></app-appareil>
        <app-appareil></app-appareil>
        <app-appareil></app-appareil>
      </ul>
    </div>
  </div>
</div>
```

Les classes CSS utilisées ici sont des classes issues de Bootstrap pour simplifier une mise en page propre. Vous trouverez plus d'informations sur bootstrapdocs.com. Pour ce cours, j'ai utilisé la version 3.3.7.

Maintenant votre navigateur devrait vous montrer quelque chose comme cela :

Pour l'instant, rien de bien spectaculaire. Vous allez maintenant utiliser l'interpolation pour commencer à dynamiser vos données. Modifiez `appareil.component.html` ainsi :

```
<li class="list-group-item">
  <h4>Appareil : {{ appareilName }}</h4>
</li>
```

Ici, vous trouvez la syntaxe pour l'interpolation : les doubles accolades `{{ }}`. Ce qui se trouve entre les doubles accolades correspond à l'expression

TypeScript que nous voulons afficher, l'expression la plus simple étant une variable. D'ailleurs, puisque la variable `appareilName` n'existe pas encore, votre navigateur n'affiche rien à cet endroit pour l'instant. Ouvrez maintenant `appareil.component.ts` :

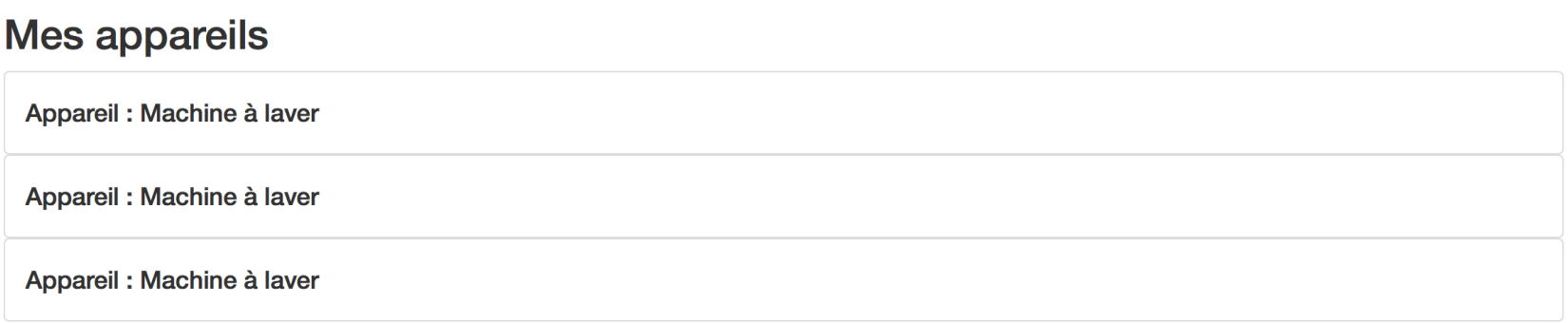
```
import { Component, OnInit } from '@angular/core';


@Component({
  selector: 'app-appareil',
  templateUrl: './appareil.component.html',
  styleUrls: ['./appareil.component.scss']
})
export class AppareilComponent implements OnInit {


  constructor() { }

  ngOnInit() {
  }


}
```

Ajoutez maintenant la ligne de code suivante en haut de la déclaration de class :

```
export class AppareilComponent implements OnInit {


  appareilName: string = 'Machine à laver';


  constructor() { }
```

La déclaration de type ici (les deux-points suivis du type `string` n'est pas obligatoire, car TypeScript déduit automatiquement le type d'une variable lorsque vous l'instanciez avec une valeur. J'ai simplement inclus la déclaration de type pour montrer la syntaxe TypeScript (vous en aurez besoin dans des chapitres ultérieurs).

Une fois le fichier enregistré, votre navigateur affiche maintenant :



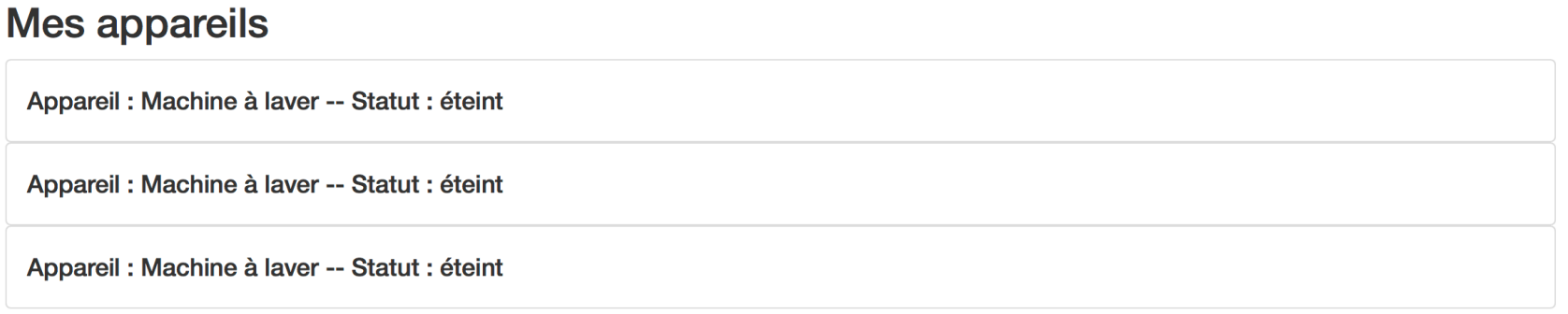
Voilà ! Vous avez maintenant une communication entre votre code TypeScript et votre template HTML. Pour l'instant, les valeurs sont codées "en dur", mais à terme, ces valeurs peuvent être calculées par votre code ou récupérées sur un serveur, par exemple. Ajoutez maintenant une nouvelle variable dans votre `AppareilComponent` :

```
appareilName: string = 'Machine à laver';

appareilStatus: string = 'éteint';
```

Puis intégrez cette variable dans le template :

```
<li class="list-group-item">
  <h4>Appareil : {{ appareilName }} -- Statut : {{ appareilStatus }}</h4>
</li>
```

Votre navigateur montre ceci :



On peut utiliser toute expression TypeScript valable pour l'interpolation. Pour démontrer cela, ajouter une méthode au fichier `AppareilComponent` :

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


@Component({
  selector: 'app-appareil',
  templateUrl: './appareil.component.html',
  styleUrls: ['./appareil.component.scss']
})
export class AppareilComponent implements OnInit {

  appareilName: string = 'Machine à laver';
  appareilStatus: string = 'éteint';

  constructor() { }

  ngOnInit() {
  }

  getStatus() {
    return this.appareilStatus;
  }
```

```
}
```

Alors que cette méthode ne fait que retourner la même valeur qu'avant, on peut imaginer une situation où elle ferait un appel API, par exemple, qui retournerait le statut de l'appareil électrique.

Modifiez maintenant le template pour prendre en compte ce changement :

```
<li class="list-group-item">
  <h4>Appareil : {{ appareilName }} -- Statut : {{ getStatus() }}</h4>
</li>
```

Vous devez avoir le même résultat visuel dans le navigateur.

Property binding

La liaison par propriété ou "property binding" est une autre façon de créer de la communication dynamique entre votre TypeScript et votre template : plutôt qu'afficher simplement le contenu d'une variable, vous pouvez modifier dynamiquement les propriétés d'un élément du DOM en fonction de données dans votre TypeScript.

Pour votre application des appareils électriques, imaginez que si l'utilisateur est authentifié, on lui laisse la possibilité d'allumer tous les appareils de la maison. Puisque l'authentification est une valeur globale, ajoutez une variable boolean dans `AppComponent`, votre component de base (vous pouvez supprimer la variable `title` puisque vous ne l'utilisez plus) :

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


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
```

```
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  isAuth = false;
}
```

Ajoutez maintenant un bouton au template global `app.component.html`, en dessous de la liste d'appareils :

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <h2>Mes appareils</h2>
      <ul class="list-group">
        <app-appareil></app-appareil>
        <app-appareil></app-appareil>
      </ul>
      <button class="btn btn-success" disabled>Tout allumer</button>
    </div>
  </div>
</div>
```

La propriété `disabled` permet de désactiver le bouton. Afin de lier cette propriété au TypeScript, il faut le mettre entre crochets `[]` et l'associer à la variable ainsi :

```
<button class="btn btn-success" [disabled]="!isAuth">Tout allume</button>
```

Le point d'exclamation fait que le bouton est désactivé lorsque `isAuth === false`. Pour montrer que cette liaison est dynamique, créez une méthode

`constructor` dans `AppComponent`, dans laquelle vous créerez une timeout qui associe la valeur `true` à `isAuth` après 4 secondes (pour simuler, par exemple, le temps d'un appel API) :

```
export class AppComponent {

  isAuth = false;


  constructor() {

    setTimeout(

      () => {

        this.isAuth = true;

      }, 4000

    );

  }

}
```

Pour en voir l'effet, rechargez la page dans votre navigateur et observez comment le bouton s'active au bout de quatre secondes. Pour l'instant le bouton ne fait rien : vous découvrirez comment exécuter du code lorsque l'utilisateur cliquera dessus avec la liaison des événements, ou "event binding".

Event binding

Avec le string interpolation et le property binding, vous savez communiquer depuis votre code TypeScript vers le template HTML. Maintenant, je vais vous montrer comment réagir dans votre code TypeScript aux événements venant du template HTML.

Actuellement, vous avez un bouton sur votre template qui s'active au bout de 4 secondes. Vous allez maintenant lui ajouter une fonctionnalité liée à l'événement "click" (déclenché quand l'utilisateur clique dessus).

Ajoutez la liaison suivante à votre bouton dans le template HTML :

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <h2>Mes appareils</h2>
      <ul class="list-group">
        <app-appareil></app-appareil>
        <app-appareil></app-appareil>
        <app-appareil></app-appareil>
      </ul>
      <button class="btn btn-success"
              [disabled]="!isAuth"
              (click)="onAllumer()">Tout allumer</button>
    </div>
  </div>
</div>
```

Ici, j'ai choisi de répartir le code sur plusieurs lignes pour le rendre plus lisible. Ce n'est bien sûr pas obligatoire, mais je vous conseille d'y penser quand un objet possède trois propriétés ou plus.

Comme vous pouvez le constater, on utilise les parenthèses `()` pour créer une liaison à un événement. Pour l'instant, la méthode `onAllumer()` n'existe pas, donc je vous propose de la créer maintenant dans `app.component.ts` , en dessous du constructeur.

Il existe une convention de nomenclature pour les méthodes liées aux événements que j'ai employée ici : "on" + le nom de l'événement. Cela permet, entre autres, de suivre plus facilement l'exécution des méthodes lorsque l'application devient plus complexe.

La méthode affichera simplement un message dans la console dans un premier temps :

```
onAllumer() {
    console.log('On allume tout !');
}
```

Enregistrez le fichier, et ouvrez la console dans votre navigateur. Lorsque le bouton s'active, cliquez dessus, et vous verrez votre message apparaître dans la console.

Même si cela reste une fonction très simple pour l'instant, cela vous montre comment lier une fonction TypeScript à un événement venant du template. De manière générale, vous pouvez lier du code à n'importe quelle propriété ou événement des éléments du DOM. Pour plus d'informations, vous pouvez consulter le Mozilla Developer Network ou W3Schools, par exemple.

Two-way binding

La liaison à double sens (ou two-way binding) utilise la liaison par propriété et la liaison par événement en même temps ; on l'utilise, par exemple, pour les formulaires, afin de pouvoir déclarer et de récupérer le contenu des champs, entre autres.

Pour pouvoir utiliser le two-way binding, il vous faut importer `FormsModule` depuis `@angular/forms` dans votre application. Vous pouvez accomplir cela en l'ajoutant à l'array `imports` de votre `AppModule` (sans oublier d'ajouter le statement `import` correspondant en haut du fichier) :

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

```
import { AppComponent } from './app.component';
import { MonPremierComponent } from './mon-premier/mon-premier.component';
import { AppareilComponent } from './appareil/appareil.component';
import { FormsModule } from '@angular/forms';


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

Le two-way binding emploie le mélange des syntaxes de property binding et d'event binding : des crochets et des parenthèses `[()]` . Pour une première démonstration, ajoutez un `<input>` dans votre template `appareil.component.html` et liez-le à la variable `appareilName` en utilisant la directive `ngModel` :

```
<li class="list-group-item">
  <h4>Appareil : {{ appareilName }} -- Statut : {{ getStatus() }}</h4>
```

```
  <input type="text" class="form-control" [(ngModel)]="appareilName">
</li>
```

Dans votre template, vous verrez un `<input>` par appareil. Le nom de l'appareil est déjà indiqué dedans, et si vous le modifiez, le contenu du `<h4>` est modifié avec. Ainsi vous voyez également que chaque instance du component `AppareilComponent` est entièrement indépendante une fois créée : le fait d'en modifier une ne change rien aux autres. Ce concept est très important, et il s'agit de l'une des plus grandes utilités d'Angular.

Propriétés personnalisées

Il est possible de créer des propriétés personnalisées dans un component afin de pouvoir lui transmettre des données depuis l'extérieur.

Il est également possible de créer des événements personnalisés, mais ce sujet va au-delà du périmètre de ce cours. Vous trouverez plus d'informations dans la documentation d'Angular.

Pour l'application des appareils électriques, il serait intéressant de faire en sorte que chaque instance d' `AppareilComponent` ait un nom différent qu'on puisse régler depuis l'extérieur du code. Pour ce faire, il faut utiliser le décorateur `@Input()` en remplaçant la déclaration de la variable `appareilName` :

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


@Component({
  selector: 'app-appareil',
  templateUrl: './appareil.component.html',
  styleUrls: ['./appareil.component.scss']
})
export class AppareilComponent implements OnInit {
```

```
  @Input() appareilName: string;

  appareilStatus: string = 'éteint';

  constructor() { }

  ngOnInit() {
  }

  getStatus() {
    return this.appareilStatus;
  }

}
```

N'oubliez pas d'importer `Input` depuis `@angular/core` en haut du fichier !

Ce décorateur, en effet, crée une propriété `appareilName` qu'on peut fixer depuis la balise `<app-appareil>` :

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <h2>Mes appareils</h2>
      <ul class="list-group">
        <app-appareil appareilName="Machine à laver"></app-appareil>
        <app-appareil appareilName="Frigo"></app-appareil>
        <app-appareil appareilName="Ordinateur"></app-appareil>
```

```
      </ul>

      <button class="btn btn-success"

              [disabled]="!isAuth"

              (click)="onAllumer()">Tout allumer</button>

    </div>

  </div>

</div>
```

C'est une première étape intéressante, mais ce serait encore plus dynamique de pouvoir passer des variables depuis `AppComponent` pour nommer les appareils (on peut imaginer une autre partie de l'application qui récupérerait ces noms depuis un serveur, par exemple).
Heureusement, vous savez déjà utiliser le property binding !

Créez d'abord vos trois variables dans `AppComponent` :

```
export class AppComponent {

  isAuth = false;


  appareilOne = 'Machine à laver';

  appareilTwo = 'Frigo';

  appareilThree = 'Ordinateur';


  constructor() {
```

Maintenant, utilisez les crochets `[]` pour lier le contenu de ces variables à la propriété du component :

```
<ul class="list-group">

    <app-appareil [appareilName]="appareilOne"></app-appareil>

    <app-appareil [appareilName]="appareilTwo"></app-appareil>
```

```
    <app-appareil [appareilName]="appareilThree"></app-appareil>
</ul>
```

Vous pouvez également créer une propriété pour régler l'état de l'appareil :

```
export class AppareilComponent implements OnInit {

  @Input() appareilName: string;
  @Input() appareilStatus: string;

  constructor() {
```

```
<ul class="list-group">
    <app-appareil [appareilName]="appareilOne" [appareilStatus]="'éteint'">
    <app-appareil [appareilName]="appareilTwo" [appareilStatus]="'allumé'">
    <app-appareil [appareilName]="appareilThree" [appareilStatus]="'éteint'
</ul>
```

Notez bien que si vous employez les crochets pour le property binding et que vous souhaitez y passer un string directement, il faut le mettre entre apostrophes, car entre les guillemets, il doit y avoir un statement de TypeScript valable. Si vous omettez les apostrophes, vous essayez d'y passer une variable nommée `allumé` ou `éteint` et l'application ne compilera pas.

Structurez le document avec des Directives

Les directives sont des instructions intégrées dans le DOM que vous utiliserez presque systématiquement quand vous créerez des applications Angular. Quand Angular lit votre template et rencontre une directive qu'il reconnait, il suit les instructions correspondantes. Vous pouvez créer vos propres directives, mais dans le cadre de ce cours, nous allons uniquement aborder certaines directives qui sont fournies avec Angular et qui sont extrêmement utiles.

Il existe deux types principaux de directive : les directives structurelles et les directives par attribut.

Les directives structurelles

Ce sont des directives qui, comme leur nom l'indique, modifient la structure du document. Dans ce chapitre, vous allez en découvrir deux (il en existe d'autres) : `*ngIf` , pour afficher des données de façon conditionnelle, et `*ngFor` , pour itérer des données dans un array, par exemple.

*ngIf

Un component auquel on ajoute la directive `*ngIf="condition"` ne s'affichera que si la condition est "truthy" (elle retourne la valeur `true` où la variable mentionnée est définie et non-nulle), comme un statement `if` classique.

Pour une démonstration simple, ajoutez une `<div>` rouge qui ne s'affichera que si l'appareil est éteint :

```
<li class="list-group-item">
  <div style="width:20px;height:20px;background-color:red;"
       *ngIf="appareilStatus === 'éteint'"></div>
  <h4>Appareil : {{ appareilName }} -- Statut : {{ getStatus() }}</h4>
  <input type="text" class="form-control" [(ngModel)]="appareilName">
</li>
```

*ngFor

Lorsque l'on ajoute la directive `*ngFor="let obj of myArray"` à un component, Angular itérera l'array `myArray` et affichera un component par objet `obj`. Pour en comprendre l'utilisation, je vous propose de modifier la façon dont votre application génère des appareils électriques.

On peut imaginer que votre application récupère, depuis un serveur, un array contenant tous les appareils et leurs états. Pour l'instant, créez cet array directement dans `AppComponent` :

```
export class AppComponent {
  isAuth = false;

  appareils = [
    {
      name: 'Machine à laver',
      status: 'éteint'
    },
    {
      name: 'Frigo',
      status: 'allumé'
    },
    {
      name: 'Ordinateur',
      status: 'éteint'
    }
  ];

  constructor() {
```

Vous avez un array avec trois objets, chaque objet ayant une propriété `name` et une propriété `status`. Vous pourriez même créer une interface

ou une class TypeScript `Appareil`, mais dans ce cas simple ce n'est pas nécessaire.

Maintenant la magie `*ngFor` :

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <h2>Mes appareils</h2>
      <ul class="list-group">
        <app-appareil  *ngFor="let appareil of appareils"
                       [appareilName]="appareil.name"
                       [appareilStatus]="appareil.status"></app-appareil>
      </ul>
      <button class="btn btn-success"
                [disabled]="!isAuth"
                (click)="onAllumer()">Tout allumer</button>
    </div>
  </div>
</div>
```

Le statement `let appareil of appareils`, comme dans une for loop classique, itère pour chaque élément `appareil` (nom arbitraire) de l'array `appareils`. Après cette directive, vous pouvez maintenant utiliser l'objet `appareil`, dont vous connaissez la forme, à l'intérieur de cette balise HTML. Vous pouvez donc utiliser le property binding, et y passer les propriétés `name` et `status` de cet objet.

N'oubliez pas l'astérisque devant ces directives, qui signifie à Angular de les traiter comme directives structurelles !

Les directives par attribut

À la différence des directives structurelles, les directives par attribut modifient le comportement d'un objet déjà existant. Vous avez déjà utilisé une directive de ce type sans le savoir : la directive `ngModel` que vous avez employée pour le two-way binding, qui modifie la valeur du `<input>` et répond à tout changement qu'on lui apporte. Je vais vous montrer deux autres exemples très utiles : `ngStyle` et `ngClass` , qui permettent d'attribuer des styles ou des classes de manière dynamique.

ngStyle

Cette directive permet d'appliquer des styles à un objet du DOM de manière dynamique. Imaginez que, pour l'application des appareils électriques, vous souhaitiez modifier la couleur du texte selon si l'appareil est allumé ou non, disons vert pour allumé, rouge pour éteint. `ngStyle` vous permet de faire cela :

```
<h4 [ngStyle]="{color: getColor()}">Appareil : {{ appareilName }} -- Statut
```

`ngStyle` prend un objet JS de type clé-valeur, avec comme clé le style à modifier, et comme valeur la valeur souhaitée pour ce style. Ici, vous faites appel à une fonction `getColor()` dans `AppareilComponent` que vous allez maintenant créer :

```
getColor() {
    if(this.appareilStatus === 'allumé') {
      return 'green';
    } else if(this.appareilStatus === 'éteint') {
      return 'red';
    }
}
```

Cette fonction retourne la valeur `'green'` si l'appareil est allumé, et `'red'` s'il est éteint, modifiant ainsi la couleur du texte dans le template.

ngClass

Au-delà de modifier des styles directement, il peut être très utile d'ajouter des classes CSS à un élément de manière dynamique. Comme `ngStyle`, `ngClass` prend un objet clé-valeur, mais cette fois avec la classe à appliquer en clé, et la condition en valeur.

Pour cet exemple, je vous propose d'appliquer des classes Bootstrap à la balise `<li>` en fonction du statut de l'appareil :

```
<li [ngClass]="{'list-group-item': true,
                'list-group-item-success': appareilStatus === 'allumé',
                'list-group-item-danger': appareilStatus === 'éteint'}">
  <div style="width:20px;height:20px;background-color:red;"
       *ngIf="appareilStatus === 'éteint'"></div>
  <h4 [ngStyle]="{color: getColor()}">Appareil : {{ appareilName }} -- Stat
  <input type="text" class="form-control" [(ngModel)]="appareilName">
</li>
```

Angular appliquera donc systématiquement la classe `list-group-item`, et selon le contenu de la variable `appareilStatus`, appliquera l'une ou l'autre des deux autres classes. Vous pouvez bien évidemment créer vos propres classes et les utiliser ; j'ai simplement choisi des classes Bootstrap pour simplifier l'explication.

Que ce soit pour `ngStyle` ou pour `ngClass`, les objets JS peuvent être des variables valables dans votre TypeScript qui seront ensuite référencées par la directive, par exemple : `[ngClass]="myClassObject"`.

Modifiez les données en temps réel avec les Pipes

Les pipes (/pʌɪp/) prennent des données en input, les transforment, et puis affichent les données modifiées dans le DOM. Il y a des pipes fournis avec Angular, et vous pouvez également créer vos propres pipes si vous en avez besoin. Je vous propose de commencer avec les pipes fournis avec Angular.

Utilisez et paramétrez les Pipes

Un pipe que l'on utilise très souvent est DatePipe , qui analyse des objets JS de type Date et qui les affiche d'une manière plus lisible que leur encodage de base. Par exemple, imaginez que vous vouliez ajouter la date de la dernière mise à jour dans votre application des appareils électriques.

Commencez par créer cet objet dans `AppComponent` et par l'afficher directement dans le template :

```
export class AppComponent {

  isAuth = false;

  lastUpdate = new Date();
```

```
<h2>Mes appareils</h2>

<p>Mis à jour : {{ lastUpdate }}</p>
```

L'objet Date a bien été créé, mais sous sa forme actuelle, il n'est pas très utile. L'avantage d'un pipe est de pouvoir modifier l'affichage de cet objet sans en modifier la nature. Ajoutons le DatePipe dans le template grâce au caractère | :

```
<p>Mis à jour : {{ lastUpdate | date }}</p>
```

La date s'affiche maintenant sous la forme "Oct 06, 2017" dans le DOM ; c'est déjà beaucoup plus lisible, mais on peut faire mieux. Angular permet de paramétrer DatePipe en lui passant un argument de formatage, par exemple :

```
<p>Mis à jour : {{ lastUpdate | date: 'short' }}</p>
```


```
<p>Mis à jour : {{ lastUpdate | date: 'yMMMMEEEEd' }}</p>
```

Il y a beaucoup de possibilités de formatage de DatePipe : vous trouverez plus d'informations dans la documentation d'Angular.

Utilisez une chaîne de Pipes

Vous pouvez avoir besoin de plusieurs pipes pour un seul élément du DOM. Imaginez, par exemple, que vous souhaitiez afficher la date de l'exemple précédent en majuscules. Vous aurez simplement à faire comme cela :

```
<p>Mis à jour : {{ lastUpdate | date: 'yMMMMEEEEd' | uppercase }}</p>
```

L'ordre des pipes est important. Si vous les inversez, UpperCasePipe ne fonctionnera pas et votre application n'affichera rien. Pensez à l'ordre dans lequel les modifications doivent être exécutées et mettez les pipes dans cet ordre.

async

Le pipe `async` est un cas particulier mais extrêmement utile dans les applications Web, car il permet de gérer des données asynchrones, par exemple des données que l'application doit récupérer sur un serveur. Dans les chapitres suivants, vous apprendrez à communiquer avec un serveur extérieur, mais pour l'instant, vous allez simuler ce comportement en créant une Promise qui va se résoudre au bout de quelques secondes.
Modifiez `lastUpdate` comme suit :

```
lastUpdate = new Promise((resolve, reject) => {

    const date = new Date();
```

```
    setTimeout(
      () => {
        resolve(date);
      }, 2000
    );
  });
```

Si vous enregistrez le fichier, l'application vous créera une erreur :

```
Error: InvalidPipeArgument: '[object Promise]' for pipe
'DatePipe'.
```

En effet, au moment de générer le DOM, `lastUpdate` est encore une Promise et n'a pas de valeur modifiable par les pipes.

Ce genre d'erreur n'est pas limitée à l'utilisation de pipe. Si vous essayez d'afficher une Promise sans pipe, elle s'affichera "[object Promise]".

Il nous faut donc ajouter AsyncPipe en début de chaîne pour dire à Angular d'attendre l'arrivée des données avant d'exécuter les autres pipes :

```
<p>Mis à jour : {{ lastUpdate | async | date: 'yMMMMEEEEd' | uppercase }}</
```

Maintenant, quand votre page recharge, le champ "Mis à jour" est vide et puis, au bout de deux secondes, les données retournées par la Promise sont reçues, modifiées par les pipes suivants, et affichées.

Félicitations ! À ce stade, vous savez :

- comment créer les composantes d'une application Angular : les components. Vous savez leur passer des données, réagir aux événements qu'ils déclenchent et même faire les deux en même temps !

- utiliser des directives pour structurer votre application et en modifier

le contenu de manière dynamique ;

- profiter des pipes pour modifier l'affichage des données sans en changer la nature.

Avec ces éléments, vous avez déjà de quoi créer des applications basiques et fonctionnelles, mais Angular est extrêmement riche et permet d'intégrer facilement d'autres fonctionnalités : c'est précisément le sujet de la prochaine partie de ce cours.

Améliorez la structure du code avec les Services

Qu'est-ce qu'un service ?

Dit très simplement, un service permet de centraliser des parties de votre code et des données qui sont utilisées par plusieurs parties de votre application ou de manière globale par l'application entière. Les services permettent donc :

- de ne pas avoir le même code doublé ou triplé à différents niveaux de l'application - ça facilite donc la maintenance, la lisibilité et la stabilité du code ;
- de ne pas copier inutilement des données - si tout est centralisé,

chaque partie de l'application aura accès aux mêmes informations, évitant beaucoup d'erreurs potentielles.

Dans le cas de l'application que vous avez créée lors des derniers chapitres, on pourrait imaginer un service `AppareilService` qui contiendrait les données des appareils électriques, et également des fonctions globales liées aux appareils, comme "tout allumer" ou "tout éteindre" que vous pourrez enfin intégrer. L'authentification reste simulée pour l'instant, mais quand vous l'intégrerez, on pourrait imaginer un deuxième service `AuthService` qui s'occuperait de vérifier l'authentification de l'utilisateur, et qui pourrait également stocker des informations sur l'utilisateur actif comme son adresse mail et son pseudo.

Injection et instances

Pour être utilisé dans l'application, un service doit être injecté, et le niveau choisi pour l'injection est très important. Il y a trois niveaux possibles pour cette injection :

- dans `AppModule` : ainsi, la même instance du service sera utilisée par tous les components de l'application *et* par les autres services ;
- dans `AppComponent` : comme ci-dessus, tous les components auront accès à la même instance du service mais non les autres services ;
- dans un autre component : le component lui-même et tous ses enfants (c'est-à-dire tous les components qu'il englobe) auront accès à la même instance du service, mais le reste de l'application n'y aura pas accès.

Pour les exemples de ce cours, vous injecterez systématiquement les services dans `AppModule` pour rendre disponible une seule instance par service à toutes les autres parties de votre application.

Créez maintenant un sous-dossier `services` dans `app` , et créez-y un nouveau fichier appelé `appareil.service.ts` :

```
export class AppareilService {


}
```

Vous y intégrerez bientôt des données et des fonctions, mais pour l'instant, vous allez injecter ce service dans `AppModule` en l'ajoutant à l'array `providers` (n'oubliez pas d'ajouter l'import correspondant en haut du fichier :

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

import { AppComponent } from './app.component';
import { MonPremierComponent } from './mon-premier/mon-premier.component';
import { AppareilComponent } from './appareil/appareil.component';
import { FormsModule } from '@angular/forms';
import { AppareilService } from './services/appareil.service';



@NgModule({
  declarations: [
    AppComponent,
    MonPremierComponent,
    AppareilComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [
```

```
    AppareilService

  ],

  bootstrap: [AppComponent]

})

export class AppModule { }
```

Angular crée maintenant une instance du service `AppareilService` pour l'application entière. Pour l'intégrer dans un component, on le déclare comme argument dans son constructeur. Intégrez-le dans `AppComponent` (sans oublier d'ajouter l'import en haut) :

```
constructor(private appareilService: AppareilService) {

    setTimeout(

      () => {

        this.isAuth = true;

      }, 4000

    );

  }
```

Maintenant, dans `AppComponent` , vous avez un membre appelé `appareilService` qui correspond à l'instance de ce service que vous avez créé dans `AppModule` . Vous y ajouterez de la fonctionnalité dans le chapitre suivant.

Utilisez les services

Le premier élément qu'il serait logique de déporter dans le service serait l'array appareils. Copiez-le depuis `AppComponent` , collez-le dans `AppareilService` et, de nouveau dans `AppComponent` , déclarez `appareils` simplement comme un array de type `any` :

```
export class AppareilService {

  appareils = [

    {

      name: 'Machine à laver',

      status: 'éteint'

    },

    {

      name: 'Frigo',

      status: 'allumé'

    },

    {

      name: 'Ordinateur',

      status: 'éteint'

    }

  ];

}
```

```
export class AppComponent {


  isAuth = false;


  appareils: any[];
```

Il faut maintenant que `AppComponent` puisse récupérer les informations stockées dans `AppareilService`. Pour cela, vous allez implémenter la méthode `ngOnInit()`.

`ngOnInit()` correspond à une "lifecycle hook". Le détail de ces hooks va au-delà du cadre de ce cours, mais pour l'instant, tout ce que vous avez

besoin de savoir, c'est que la méthode `ngOnInit()` d'un component est exécutée une fois par instance au moment de la création du component par Angular, et après son constructeur. On l'utilise très souvent pour initialiser des données une fois le component créé. Plus tard dans cette partie du cours, vous découvrirez également `ngOnDestroy()`. Pour plus d'informations, référez-vous à la documentation d'Angular.

Pour ce faire, vous allez d'abord créer la fonction `ngOnInit()` - généralement on la place après le constructeur et avant les autres méthodes du component :

```
constructor(private appareilService: AppareilService) {
    setTimeout(
      () => {
        this.isAuth = true;
      }, 4000
    );
  }

ngOnInit() {

}
```

Ensuite, dans la déclaration de classe `AppComponent`, vous allez implémenter l'interface `OnInit` (en l'important depuis `@angular/core` en haut) :

```
import { Component, OnInit } from '@angular/core';
import { AppareilService } from './services/appareil.service';

@Component({
  selector: 'app-root',
```

```
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit {
```

Vous pouvez maintenant récupérer les informations depuis `AppareilService` dans la méthode `ngOnInit()` :

```
ngOnInit() {
    this.appareils = this.appareilService.appareils;
}
```

La liaison directe à un array comme ici n'est généralement pas un best practice. J'ai choisi d'employer cette méthode ici pour montrer plus simplement l'intégration des services, mais ne vous en faites pas : nous verrons les meilleures méthodes plus tard dans ce cours !

Votre application devrait fonctionner à nouveau, avec la liste des appareils électriques qui s'affiche comme avant. Il n'y a aucune différence visuelle, mais votre code est maintenant plus modulaire, et ce sera plus facile d'ajouter des fonctionnalités. Par exemple, vous allez pouvoir créer deux nouvelles méthodes : `switchOnAll()` et `switchOffAll()` pour allumer ou éteindre tous les appareils d'un coup.

Commencez par préparer ces méthodes dans `AppareilService` :

```
switchOnAll() {
    for(let appareil of this.appareils) {
      appareil.status = 'allumé';
    }
}

switchOffAll() {
```

```
    for(let appareil of this.appareils) {
      appareil.status = 'éteint';
    }
}
```

Puis ajoutez un deuxième bouton dans le template de `AppComponent` :

```
<button class="btn btn-success"
              [disabled]="!isAuth"
              (click)="onAllumer()">Tout allumer</button>
<button class="btn btn-danger"
              [disabled]="!isAuth"
              (click)="onEteindre()">Tout éteindre</button>
```

Enfin, il ne vous reste plus qu'à capturer les événements `click` dans `AppComponent` pour ensuite déclencher les méthodes dans `AppareilService`. Commencez déjà par `onAllumer()` :

```
onAllumer() {
    this.appareilService.switchOnAll();
}
```

Ensuite, pour `onEteindre()`, vous allez d'abord afficher un message de confirmation pour vous assurer que l'utilisateur est certain de vouloir tout éteindre :

```
onEteindre() {
    if(confirm('Etes-vous sûr de vouloir éteindre tous vos appareils ?')) {
      this.appareilService.switchOffAll();
    } else {
      return null;
```

```
    }
}
```

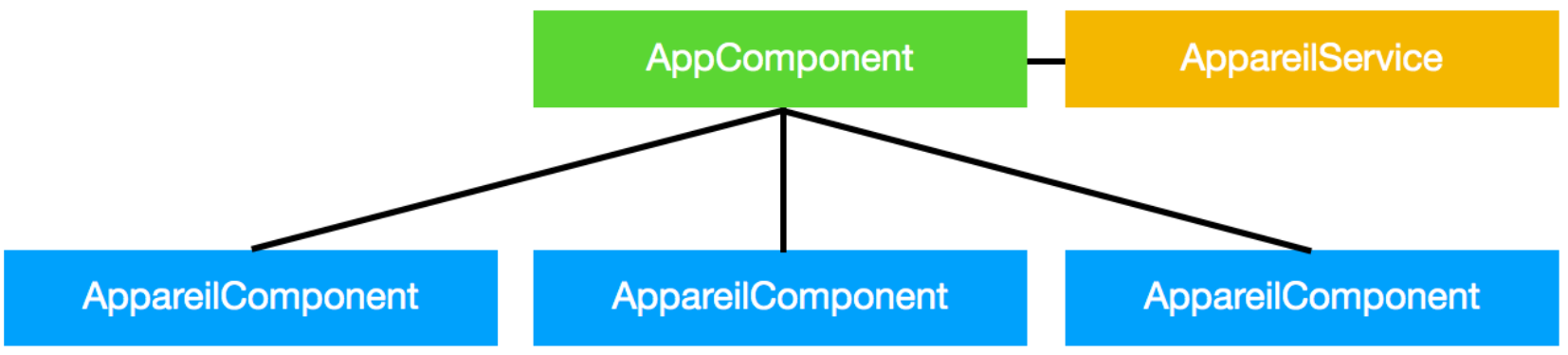
Vos boutons allument et éteignent tous les appareils grâce à la communication entre votre `AppComponent` et votre `AppareilService`.

Mais j'aurais pu faire tout ça à l'intérieur du component - quel est l'intérêt d'avoir tout mis dans un service ?

Effectivement, les fonctionnalités que vous avez ajoutées pour l'instant auraient pu rester dans `AppComponent`, mais dans le chapitre suivant, vous allez profiter du service pour créer de la communication entre vos components, notamment des components enfants vers leur parent.

Faites communiquer vos components

Pour l'instant, votre utilisateur ne peut qu'allumer ou éteindre tous les appareils à la fois. Ce qui pourrait être très intéressant, ce serait qu'il puisse en allumer ou éteindre un à la fois. Actuellement, le plan de l'application ressemble à ça :



`AppareilService` fournit les données sur les appareils à `AppComponent`. Ensuite, `AppComponent` génère trois instances de `AppareilComponent` selon ces données. Il n'y a actuellement aucune communication entre les components enfants et leur parent. Vous pouvez modifier cela en intégrant `AppareilService` dans les `AppareilComponent` et en créant des méthodes qui permettent de modifier un appareil à la fois. Procédez étape par étape.

Dans un premier temps, il faudra que chaque instance de `AppareilComponent` puisse dire à `AppareilService` à quel membre de l'array `appareils` elle correspond. Heureusement, Angular nous permet de faire ça facilement. Dans la directive `*ngFor`, ajoutez :

```
<ul class="list-group">
    <app-appareil  *ngFor="let appareil of appareils; let i = index"
                   [appareilName]="appareil.name"
                   [appareilStatus]="appareil.status"></app-appareil>
</ul>
```

Cette commande rend disponible l'index de l'objet `appareil` dans l'array `appareils`. Ensuite, il faut pouvoir capturer et travailler avec cette variable : vous pouvez utiliser le property binding. Pour cela, ajoutez un membre `index` au component en tant que `@Input()` :

```
@Input() appareilName: string;
@Input() appareilStatus: string;
@Input() index: number;
```

Puis liez-y l'index `i` depuis le template :

```
<ul class="list-group">
    <app-appareil  *ngFor="let appareil of appareils; let i = index"
                   [appareilName]="appareil.name"
                   [appareilStatus]="appareil.status"
                   [index]="i"></app-appareil>
</ul>
```

À partir de là, vous avez une variable `index` disponible à l'intérieur du component qui correspond à l'index de l'appareil dans l'array de

`AppareilService`. Vous verrez dans quelques instants pourquoi vous en avez besoin.

Dans `AppareilService`, vous allez maintenant créer les méthodes permettant d'allumer ou d'éteindre un seul appareil en fonction de son index dans l'array `appareils` :

```
switchOnOne(i: number) {
    this.appareils[i].status = 'allumé';
}

switchOffOne(i: number) {
    this.appareils[i].status = 'éteint';
}
```

Ensuite, dans `AppareilComponent`, vous allez d'abord intégrer le service `AppareilService`, en l'important en haut du fichier comme toujours :

```
constructor(private appareilService: AppareilService) { }
```

Puis vous allez préparer la méthode qui, en fonction du statut actuel de l'appareil, l'allumera ou l'éteindra :

```
onSwitch() {
    if(this.appareilStatus === 'allumé') {
      this.appareilService.switchOffOne(this.index);
    } else if(this.appareilStatus === 'éteint') {
      this.appareilService.switchOnOne(this.index);
    }
}
```

Le nom `onSwitch()` ici est choisi pour respecter la norme d'employer "on"

pour la capture d'un événement, et non pour dire "switch on" comme "allumer".

Enfin, vous allez créer le bouton dans le template qui déclenchera cette méthode. Il serait intéressant que ce bouton soit contextuel : si l'appareil est allumé, il affichera "Éteindre" et inversement. Pour cela, le plus simple est de créer deux boutons dotés de la directive `*ngIf` :

```
<li [ngClass]="{'list-group-item': true,
                'list-group-item-success': appareilStatus === 'allumé',
                'list-group-item-danger': appareilStatus === 'éteint'}">

  <h4 [ngStyle]="{color: getColor()}">Appareil : {{ appareilName }} -- Stat
  <input type="text" class="form-control" [(ngModel)]="appareilName">

  <button class="btn btn-sm btn-success"
          *ngIf="appareilStatus === 'éteint'"
          (click)="onSwitch()">Allumer</button>
  <button class="btn btn-sm btn-danger"
          *ngIf="appareilStatus === 'allumé'"
          (click)="onSwitch()">Eteindre</button>

</li>
```

Vous pouvez supprimer la `<div>` conditionnelle rouge, car elle ne sert plus vraiment, avec tous les styles que vous avez ajoutés pour signaler l'état d'un appareil.

Et voilà ! Vos components communiquent entre eux à l'aide du service, qui centralise les données et certaines fonctionnalités. Même si les effets ne sont que visuels pour l'instant, vous pouvez très bien imaginer qu'à l'intérieur des méthodes du service `AppareilService` , il y ait des appels

API permettant de vraiment allumer ou éteindre les appareils et d'en vérifier le fonctionnement.

Gérez la navigation avec le Routing

L'un des énormes avantages d'utiliser Angular est de pouvoir créer des "single page application" (SPA). Sur le Web, ces applications sont rapides et lisses : il n'y a qu'un seul chargement de page au début, et même si les données mettent parfois du temps à arriver, la sensation pour l'utilisateur est celle d'une application native. Au lieu de charger une nouvelle page à chaque clic ou à chaque changement d'URL, on remplace le contenu ou une partie du contenu de la page : on modifie les components qui y sont affichés, ou le contenu de ces components. On accomplit tout cela avec le "routing", où l'application lit le contenu de l'URL pour afficher le ou les components requis.

L'application des appareils électriques n'a que la view des appareils à

afficher pour le moment ; je vous propose de créer un component pour l'authentification (qui restera simulée pour l'instant) et vous créerez un menu permettant de naviguer entre les views.

Tout d'abord, créez le component avec le CLI :

```
ng g c auth
```

Vous allez également devoir modifier un peu l'organisation actuelle afin d'intégrer plus facilement le routing : vous allez créer un component qui contiendra toute la view actuelle et qui s'appellera `AppareilViewComponent` :

```
ng g c appareil-view
```

Ensuite, coupez tout le contenu de la colonne dans `app.component.html`, enregistrez-le dans `appareil-view.component.html`, et remplacez-le par la nouvelle balise `<app-appareil-view>` :

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <app-appareil-view></app-appareil-view>
    </div>
  </div>
</div>
```

Il faudra également déménager la logique de cette view pour que tout remarche : injectez `AppareilService`, créez l'array `appareils`, intégrez la logique `ngOnInit` et déplacez les fonctions `onAllumer()` et `onEteindre()` :

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

```
import { AppareilService } from '../services/appareil.service';


@Component({
  selector: 'app-appareil-view',
  templateUrl: './appareil-view.component.html',
  styleUrls: ['./appareil-view.component.scss']
})
export class AppareilViewComponent implements OnInit {


  appareils: any[];


  lastUpdate = new Promise((resolve, reject) => {
    const date = new Date();
    setTimeout(
      () => {
        resolve(date);
      }, 2000
    );
});


constructor(private appareilService: AppareilService) { }


ngOnInit() {
  this.appareils = this.appareilService.appareils;
}


onAllumer() {
  this.appareilService.switchOnAll();
}
```

```
  onEteindre() {
    if(confirm('Etes-vous sûr de vouloir éteindre tous vos appareils ?')) {
      this.appareilService.switchOffAll();
    } else {
      return null;
    }
  }

}
```

Vous pouvez faire le ménage dans `AppComponent` , en retirant tout ce qui n'y sert plus. Créez également une boolean `isAuth` dans `AppareilViewComponent` , et déclarez-la comme `false` , car vous allez intégrer un service d'authentification pour la suite.

Ajoutez la barre de navigation suivante à `AppComponent` :

```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-collapse">
      <ul class="nav navbar-nav">
        <li><a href="#">Authentification</a></li>
        <li class="active"><a href="#">Appareils</a></li>
      </ul>
    </div>
  </div>
</nav>
```

Maintenant, tout est prêt pour créer le routing de l'application.

Créez des routes

Tout d'abord, qu'est-ce qu'une route dans une application Angular ?

Il s'agit des instructions d'affichage à suivre pour chaque URL, c'est-à-dire quel(s) component(s) il faut afficher à quel(s) endroit(s) pour un URL donné.

Puisque le routing d'une application est fondamentale pour son fonctionnement, on déclare les routes dans `app.module.ts` .

Il est possible d'avoir un fichier séparé pour le routing, mais en termes de fonctionnalité, cela ne change rien : c'est juste une question d'organisation du code.

On crée une constante de type `Routes` (qu'on importe depuis `@angular/router`) qui est un array d'objets JS qui prennent une certaine forme :

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


import { AppComponent } from './app.component';
import { MonPremierComponent } from './mon-premier/mon-premier.component';
import { AppareilComponent } from './appareil/appareil.component';
import { FormsModule } from '@angular/forms';
import { AppareilService } from './services/appareil.service';
import { AuthComponent } from './auth/auth.component';
import { AppareilViewComponent } from './appareil-view/appareil-view.compon
import { Routes } from '@angular/router';


const appRoutes: Routes = [
```

```
  { path: 'appareils', component: AppareilViewComponent },
  { path: 'auth', component: AuthComponent },
  { path: '', component: AppareilViewComponent }
];
```

Le path correspond au string qui viendra après le `/` dans l'URL : sur votre serveur local, le premier path ici correspond donc à `localhost:4200/appareils`.

Ne pas ajouter de slash au début de la propriété `path`.

Ensuite, le `component` correspond au component que l'on veut afficher lorsque l'utilisateur navigue au `path` choisi.

J'ai ajouté un `path` vide, qui correspond tout simplement à `localhost:4200` (ou à la racine de l'application seule), car si on ne traite pas le `path` vide, chaque refresh de l'application la fera planter. Je vous montrerai d'autres façons de gérer cela dans les chapitres suivants.

Les routes sont maintenant créées, mais il faut les enregistrer dans votre application. Pour cela, vous allez importer `RouterModule` depuis `@angular/router` et vous allez l'ajouter à l'array imports de votre `AppModule`, tout en lui appelant la méthode `forRoot()` en lui passant l'array de routes que vous venez de créer :

```
imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(appRoutes)
],
```

Maintenant que les routes sont enregistrées, il ne reste plus qu'à dire à Angular où vous souhaitez afficher les components dans le template lorsque l'utilisateur navigue vers la route en question. On utilise la balise

`<router-outlet>` :

```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
```

Lorsque vous changez de route (pour l'instant, en modifiant l'URL directement dans la barre d'adresse du navigateur), la page n'est pas rechargée, mais le contenu sous la barre de navigation change. Dans le chapitre suivant, vous allez intégrer les liens de la barre de navigation afin que l'utilisateur puisse naviguer facilement.

Naviguez avec les routerLink

Afin que l'utilisateur puisse naviguer à l'intérieur de votre application, il est nécessaire de créer des liens ou des boutons qui naviguent vers les routes que vous avez créées. Dans le chapitre précédent, vous avez créé des liens typiques dans la barre de navigation, mais qui ne font rien pour l'instant.

Vous pourriez vous dire qu'il suffirait de marquer le `path` de vos routes directement dans l'attribut `href` , et techniquement, cela permet d'atteindre les routes que vous avez créées.

Alors pourquoi on ne fait pas comme ça ?

Tout simplement parce que, si vous regardez bien, en employant cette technique, la page est rechargée à chaque clic ! On perd totalement l'intérêt d'une Single Page App !

Du coup, on retire l'attribut `href` et on le remplace par l'attribut

routerLink :

```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-collapse">
      <ul class="nav navbar-nav">
        <li><a routerLink="auth">Authentification</a></li>
        <li class="active"><a routerLink="appareils">Appareils</a></li>
      </ul>
    </div>
  </div>
</nav>
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
```

Ainsi, les routes sont chargées instantanément : on conserve l'ergonomie d'une SPA.

Pour finaliser cette étape, il serait intéressant que la classe `active` ne s'applique qu'au lien du component réellement actif. Heureusement, Angular fournit un attribut pour cela qui peut être ajouté au lien directement ou à son élément parent :

```
<ul class="nav navbar-nav">
    <li routerLinkActive="active"><a routerLink="auth">Authentification</a>
    <li routerLinkActive="active"><a routerLink="appareils">Appareils</a></
```

```
</ul>
```

Maintenant, les liens s'activent visuellement.

Naviguez avec le Router

Il peut y avoir des cas où vous aurez besoin d'exécuter du code avant une navigation. Par exemple, on peut avoir besoin d'authentifier un utilisateur et, si l'authentification fonctionne, de naviguer vers la page que l'utilisateur souhaite voir. Je vous propose d'intégrer cette fonctionnalité à l'application des appareils électriques (l'authentification elle-même restera simulée pour l'instant).

Tout d'abord, créez un nouveau fichier `auth.service.ts` dans le dossier services pour gérer l'authentification (n'oubliez pas de l'ajouter également dans l'array `providers` dans `AppModule`) :

```
export class AuthService {

  isAuth = false;


  signIn() {
    return new Promise(
      (resolve, reject) => {
        setTimeout(
          () => {
            this.isAuth = true;
            resolve(true);
          }, 2000
        );
      }
    );
```

```
  }


  signOut() {
    this.isAuth = false;
  }
}
```

La variable `isAuth` donne l'état d'authentification de l'utilisateur. La méthode `signOut()` "déconnecte" l'utilisateur, et la méthode `signIn()` authentifie automatiquement l'utilisateur au bout de 2 secondes, simulant le délai de communication avec un serveur.

Dans le component `AuthComponent`, vous allez simplement créer deux boutons et les méthodes correspondantes pour se connecter et se déconnecter (qui s'afficheront de manière contextuelle : le bouton "se connecter" ne s'affichera que si l'utilisateur est déconnecté et vice versa) :

```
import { Component, OnInit } from '@angular/core';
import { AuthService } from '../services/auth.service';

@Component({
  selector: 'app-auth',
  templateUrl: './auth.component.html',
  styleUrls: ['./auth.component.scss']
})
export class AuthComponent implements OnInit {

  authStatus: boolean;

  constructor(private authService: AuthService) { }
```

```
ngOnInit() {
  this.authStatus = this.authService.isAuth;
}

onSignIn() {
  this.authService.signIn().then(
    () => {
      console.log('Sign in successful!');
      this.authStatus = this.authService.isAuth;
    }
  );
}

onSignOut() {
  this.authService.signOut();
    this.authStatus = this.authService.isAuth;
  }

}
```

Puisque la méthode `signIn()` du service retourne une Promise, on peut employer une fonction callback asynchrone avec `.then()` pour exécuter du code une fois la Promise résolue. Ajoutez simplement les boutons, et tout sera prêt pour intégrer la navigation :

```
<h2>Authentification</h2>
<button class="btn btn-success" *ngIf="!authStatus" (click)="onSignIn()">Se
<button class="btn btn-danger" *ngIf="authStatus" (click)="onSignOut()">Se
```

Le comportement recherché serait qu'une fois l'utilisateur authentifié, l'application navigue automatiquement vers la view des appareils. Pour cela, il faut injecter le `Router` (importé depuis `@angular/router`) pour accéder à la méthode `navigate()` :

```
constructor(private authService: AuthService, private router: Router) { }

onSignIn() {
    this.authService.signIn().then(
      () => {
        console.log('Sign in successful!');
        this.authStatus = this.authService.isAuth;
        this.router.navigate(['appareils']);
      }
    );
}
```

La fonction navigate prend comme argument un array d'éléments (ce qui permet de créer des chemins à partir de variables, par exemple) qui, dans ce cas, n'a qu'un seul membre : le `path` souhaité.

Le chemin `appareils` est toujours accessible actuellement, même sans authentification : dans un chapitre ultérieur, vous apprendrez à le sécuriser totalement. Avant cela, vous allez apprendre à ajouter des paramètres à vos routes.

Paramètres de routes

Imaginez qu'on souhaite pouvoir cliquer sur un appareil dans la liste d'appareils afin d'afficher une page avec plus d'informations sur cet appareil : on peut imaginer un système de routing de type `appareils/nom-de-l'appareil`, par exemple. Si on n'avait que deux ou trois appareils, on pourrait être tenté de créer une route par appareil, mais imaginez un cas de

figure où l'on aurait 30 appareils, ou 300. Imaginez qu'on laisse l'utilisateur créer de nouveaux appareils ; l'approche de créer une route par appareil n'est pas adaptée. Dans ce genre de cas, on choisira plutôt de créer une route avec paramètre.

Tout d'abord, vous allez créer la route dans `AppModule` :

```
const appRoutes: Routes = [
  { path: 'appareils', component: AppareilViewComponent },
  { path: 'appareils/:id', component: SingleAppareilComponent },
  { path: 'auth', component: AuthComponent },
  { path: '', component: AppareilViewComponent }
];
```

L'utilisation des deux-points `:` avant un fragment de route déclare ce fragment comme étant un paramètre : tous les chemins de type `appareils/*` seront renvoyés vers `SingleAppareilComponent`, que vous allez maintenant créer :

```
import { Component, OnInit } from '@angular/core';
import { AppareilService } from '../services/appareil.service';

@Component({
  selector: 'app-single-appareil',
  templateUrl: './single-appareil.component.html',
  styleUrls: ['./single-appareil.component.scss']
})
export class SingleAppareilComponent implements OnInit {

  name: string = 'Appareil';
  status: string = 'Statut';
```

```
  constructor(private appareilService: AppareilService) { }

  ngOnInit() {
  }

}
```

```
<h2>{{ name }}</h2>
<p>Statut : {{ status }}</p>
<a routerLink="/appareils">Retour à la liste</a>
```

Pour l'instant, si vous naviguez vers `/appareils/nom`, peu importe le nom que vous choisissez, vous avez accès à `SingleAppareilComponent`.

Maintenant, vous allez y injecter `ActivatedRoute`, importé depuis `@angular/router`, afin de récupérer le fragment `id` de l'URL :

```
constructor(private appareilService: AppareilService,
            private route: ActivatedRoute) { }
```

Puis, dans `ngOnInit()`, vous allez utiliser l'objet `snapshot` qui contient les paramètres de l'URL et, pour l'instant, attribuer le paramètre `id` à la variable `name` :

```
ngOnInit() {
    this.name = this.route.snapshot.params['id'];
}
```

Ainsi, le fragment que vous tapez dans la barre d'adresse après `appareils/` s'affichera dans le template, mais ce n'est pas le

comportement recherché. Pour atteindre l'objectif souhaité, commencez par ajouter, dans `AppareilService` , un identifiant unique pour chaque appareil et une méthode qui rendra l'appareil correspondant à un identifiant :

```
appareils = [
    {
      id: 1,
      name: 'Machine à laver',
      status: 'éteint'
    },
    {
      id: 2,
      name: 'Frigo',
      status: 'allumé'
    },
    {
      id: 3,
      name: 'Ordinateur',
      status: 'éteint'
    }
];

getAppareilById(id: number) {
    const appareil = this.appareils.find(
      (s) => {
        return s.id === id;
      }
    );
    return appareil;
```

```
}
```

Maintenant, dans `SingleAppareilComponent` , vous allez récupérer l'identifiant de l'URL et l'utiliser pour récupérer l'appareil correspondant :

```
ngOnInit() {
    const id = this.route.snapshot.params['id'];
    this.name = this.appareilService.getAppareilById(+id).name;
    this.status = this.appareilService.getAppareilById(+id).status;
}
```

Puisqu'un fragment d'URL est forcément de type `string` , et que la méthode `getAppareilById()` prend un nombre comme argument, il ne faut pas oublier d'utiliser `+` avant `id` dans l'appel pour caster la variable comme nombre.

Vous pouvez naviguer manuellement vers `/appareils/2` , par exemple, mais cela recharge encore la page, et vous perdez l'état des appareils (si vous en allumez ou éteignez par exemple). Pour finaliser cette fonctionnalité, intégrez l'identifiant unique dans `AppareilComponent` et dans `AppareilViewComponent` , puis créez un `routerLink` pour chaque appareil qui permet d'en regarder le détail :

```
@Input() appareilName: string;
@Input() appareilStatus: string;
@Input() index: number;
@Input() id: number;
```

```
<ul class="list-group">
  <app-appareil  *ngFor="let appareil of appareils; let i = index"
                 [appareilName]="appareil.name"
                 [appareilStatus]="appareil.status"
```

```
                [index]="i"

                [id]="appareil.id"></app-appareil>

</ul>

<h4 [ngStyle]="{color: getColor()}">Appareil : {{ appareilName }} -- Statut

<a [routerLink]="[id]">Détail</a>
```

Ici, vous utilisez le format array pour `routerLink` en property binding afin d'accéder à la variable `id` .

Ça y est ! Vous pouvez maintenant accéder à la page Détail pour chaque appareil, et les informations de statut qui s'y trouvent sont automatiquement à jour grâce à l'utilisation du service.

Redirection

Il peut y avoir des cas de figure où l'on souhaiterait rediriger un utilisateur, par exemple pour afficher une page 404 lorsqu'il entre une URL qui n'existe pas.

Pour l'application des appareils électriques, commencez par créer un component 404 très simple, appelé `four-oh-four.component.ts` :

```
<h2>Erreur 404</h2>

<p>La page que vous cherchez n'existe pas !</p>
```

Ensuite, vous allez ajouter la route "directe" vers cette page, ainsi qu'une route "wildcard", qui redirigera toute route inconnue vers la page d'erreur :

```
const appRoutes: Routes = [
  { path: 'appareils', component: AppareilViewComponent },
  { path: 'appareils/:id', component: SingleAppareilComponent },
  { path: 'auth', component: AuthComponent },
```

```
  { path: '', component: AppareilViewComponent },
  { path: 'not-found', component: FourOhFourComponent },
  { path: '**', redirectTo: 'not-found' }
];
```

Ainsi, quand vous entrez un chemin dans la barre de navigation qui n'est pas directement pris en charge par votre application, vous êtes redirigé vers `/not-found` et donc le component 404.

Guards

Il peut y avoir des cas de figure où vous souhaiterez exécuter du code avant qu'un utilisateur puisse accéder à une route ; par exemple, vous pouvez souhaiter vérifier son authentification ou son identité. Techniquement, ce serait possible en insérant du code dans la méthode `ngOnInit()` de chaque component, mais cela deviendrait lourd, avec du code répété et une multiplication des erreurs potentielles. Ce serait donc mieux d'avoir une façon de centraliser ce genre de fonctionnalité. Pour cela, il existe la guard `canActivate`.

Une guard est en effet un service qu'Angular exécutera au moment où l'utilisateur essaye de naviguer vers la route sélectionnée. Ce service implémente l'interface `canActivate`, et donc doit contenir une méthode du même nom qui prend les arguments `ActivatedRouteSnapshot` et `RouterStateSnapshot` (qui lui seront fournis par Angular au moment de l'exécution) et retourne une valeur booléenne, soit de manière synchrone (boolean), soit de manière asynchrone (sous forme de Promise ou d'Observable) :

```
import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot } from '@
import { Observable } from 'rxjs/Observable';


export class AuthGuard implements CanActivate {
```

```
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | b


  }
}
```

N'oubliez pas d'importer les différents éléments en haut du fichier.

Si vous ne connaissez pas encore les Observables, ne vous inquiétez pas, vous les découvrirez en détail dans le chapitre suivant.

Sauvegardez ce fichier dans le dossier `services` sous le nom `auth-guard.service.ts`.

Ensuite, il faut injecter le service `AuthService` dans ce nouveau service. Pour injecter un service dans un autre service, il faut que le service dans lequel on injecte un autre ait le décorateur `@Injectable`, à importer depuis `@angular/core` :

```
import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot } from '@
import { Observable } from 'rxjs/Observable';
import { AuthService } from './auth.service';
import { Injectable } from '@angular/core';


@Injectable()
export class AuthGuard implements CanActivate {


  constructor(private authService: AuthService) { }


  canActivate(
    route: ActivatedRouteSnapshot,
```

```
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | b


  }
}
```

À l'intérieur de la méthode `canActivate()`, vous allez vérifier l'état de l'authentification dans `AuthService`. Si l'utilisateur est authentifié, la méthode renverra `true`, permettant l'accès à la route protégée. Sinon, vous pourriez retourner `false`, mais cela empêchera simplement l'accès sans autre fonctionnalité. Il serait intéressant de rediriger l'utilisateur vers la page d'authentification, le poussant à s'identifier :

```
import { ActivatedRouteSnapshot, CanActivate, Router, RouterStateSnapshot }
import { Observable } from 'rxjs/Observable';
import { AuthService } from './auth.service';
import { Injectable } from '@angular/core';


@Injectable()
export class AuthGuard implements CanActivate {


  constructor(private authService: AuthService,
              private router: Router) { }


  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | b
    if(this.authService.isAuth) {
      return true;
    } else {
      this.router.navigate(['/auth']);
```

```
    }
  }
}
```

Pour appliquer cette garde à la route `/appareils` et à toutes ses routes enfants, il faut l'ajouter dans `AppModule`. N'oubliez pas d'ajouter `AuthGuard` à l'array `providers`, puisqu'il s'agit d'un service :

```
const appRoutes: Routes = [
  { path: 'appareils', canActivate: [AuthGuard], component: AppareilViewCom
  { path: 'appareils/:id', canActivate: [AuthGuard], component: SingleAppar
  { path: 'auth', component: AuthComponent },
  { path: '', component: AppareilViewComponent },
  { path: 'not-found', component: FourOhFourComponent },
  { path: '**', redirectTo: 'not-found' }
];
```

Maintenant, si vous essayez d'accéder à `/appareils` sans être authentifié, vous êtes automatiquement redirigé vers `/auth`. Si vous cliquez sur "Se connecter", vous pouvez accéder à la liste d'appareils sans problème.

La route `/appareils` étant protégée, vous pouvez retirer les liaisons `disabled` des boutons "Tout allumer" et "Tout éteindre", car vous pouvez être certain que tout utilisateur accédant à cette route sera authentifié.

Dans ce chapitre, vous avez appris à gérer le routing de votre application avec des `routerLink` et de manière programmatique (avec `router.navigate()`). Vous avez également vu comment rediriger un utilisateur, et comment protéger des routes de votre application avec les guards.

Observez les données avec RxJS

Pour réagir à des événements ou à des données de manière asynchrone (c'est-à-dire ne pas devoir attendre qu'une tâche, par exemple un appel HTTP, soit terminée avant de passer à la ligne de code suivante), il y a eu plusieurs méthodes depuis quelques années. Il y a le système de callback, par exemple, ou encore les Promise. Avec l'API RxJS, fourni et très intégré dans Angular, la méthode proposée est celle des Observables.

Observables

Mais qu'est-ce qu'un Observable ?

Très simplement, un Observable est un objet qui émet des informations auxquelles on souhaite réagir. Ces informations peuvent venir d'un champ de texte dans lequel l'utilisateur rentre des données, ou de la progression

d'un chargement de fichier, par exemple. Elles peuvent également venir de la communication avec un serveur : le client HTTP, que vous verrez dans un chapitre ultérieur, emploie les Observables.

Les Observables sont mis à disposition par RxJS, un package tiers qui est fourni avec Angular. Il s'agit d'un sujet très vaste, donc je ne parlerai que de quelques éléments importants dans ce cours. Si vous souhaitez en savoir plus, vous trouverez toute la documentation sur le site ReactiveX.

À cet Observable, on associe un Observer — un bloc de code qui sera exécuté à chaque fois que l'Observable émet une information.
 L'Observable émet trois types d'information : des données, une erreur, ou un message complete. Du coup, tout Observer peut avoir trois fonctions : une pour réagir à chaque type d'information.

Pour créer un cas concret simple, vous allez créer un Observable dans `AppComponent` qui enverra un nouveau chiffre toutes les secondes. Vous allez ensuite observer cet Observable et l'afficher dans le DOM : de cette manière, vous pourrez dire à l'utilisateur depuis combien de temps il visualise l'application.

Pour avoir accès aux Observables et aux méthodes que vous allez utiliser, il faut ajouter deux imports :

```
import { Observable } from 'rxjs/Observable';

import 'rxjs/add/observable/interval';
```

Le premier import sert à rendre disponible le type Observable, et le deuxième vous donne accès à la méthode que vous allez utiliser : la méthode `interval()`, qui crée un Observable qui émet un chiffre croissant à intervalles réguliers et qui prend le nombre de millisecondes souhaité pour l'intervalle comme argument.

Implémentez `OnInit` et créez l'Observable dans `ngOnInit()` :

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

```
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/interval';


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit {


  ngOnInit() {
    const counter = Observable.interval(1000);
  }
}
```

Maintenant que vous avez un Observable, il faut l'observer ! Pour cela, vous utiliserez la fonction `subscribe()`, qui prendra comme arguments entre une et trois fonctions anonymes pour gérer les trois types d'informations que cet Observable peut envoyer : des données, une erreur ou un message `complete`.

Créez une variable `secondes` dans `AppComponent` et affichez-la dans le template :

```
export class AppComponent implements OnInit {


  secondes: number;


  ngOnInit() {
    const counter = Observable.interval(1000);
```

```
  }
}

<ul class="nav navbar-nav">
    <li routerLinkActive="active"><a routerLink="auth">Authentification</a>
    <li routerLinkActive="active"><a routerLink="appareils">Appareils</a></
</ul>
<div class="navbar-right">
    <p>Vous êtes connecté depuis {{ secondes }} secondes !</p>
</div>
```

Maintenant vous allez souscrire à l'Observable et créer trois fonctions : la première va se déclencher à chaque émission de données par l'Observable et va attribuer cette valeur à la variable "secondes" ; la deuxième gèrera toute erreur éventuelle ; et la troisième se déclenchera si l'Observable s'achève :

```
ngOnInit() {
    const counter = Observable.interval(1000);
    counter.subscribe(
      (value) => {
        this.secondes = value;
      },
      (error) => {
        console.log('Uh-oh, an error occurred! : ' + error);
      },
      () => {
        console.log('Observable complete!');
      }
    );
}
```

Dans le cas actuel, l'Observable que vous avez créé ne rendra pas d'erreur et ne se complétera pas. Je veux simplement vous montrer l'intégration complète de la fonction `subscribe()`. Il est possible de souscrire à un Observable en intégrant uniquement la première fonction.

Le compteur dans le DOM fonctionne, et continuera à compter à l'infini ! Dans le chapitre suivant, vous verrez comment éviter les erreurs potentielles liées à ce genre de comportement infini.

Subscriptions

Dans le chapitre précédent, vous avez appris à créer un Observable et à vous y souscrire. Pour rappel, la fonction `subscribe()` prend comme arguments trois fonctions anonymes :

- la première se déclenche à chaque fois que l'Observable émet de nouvelles données, et reçoit ces données comme argument ;
- la deuxième se déclenche si l'Observable émet une erreur, et reçoit cette erreur comme argument ;
- la troisième se déclenche si l'Observable s'achève, et ne reçoit pas d'argument.

Pour l'instant, cette souscription n'est pas stockée dans une variable : on ne peut donc plus y toucher une fois qu'elle est lancée, et ça peut vous causer des bugs ! En effet, une souscription à un Observable qui continue à l'infini continuera à recevoir les données, que l'on s'en serve ou non, et vous pouvez en subir des comportements inattendus.

Ce n'est pas le cas pour tous les Observables. Généralement, les souscriptions aux Observables fournis par Angular se suppriment toutes seules lors de la destruction du component.

Afin d'éviter tout problème, quand vous utilisez des Observables personnalisés, il est vivement conseillé de stocker la souscription dans un

objet Subscription (à importer depuis rxjs/Subscription) :

```
export class AppComponent implements OnInit {


  secondes: number;
  counterSubscription: Subscription;


  ngOnInit() {
    const counter = Observable.interval(1000);
    this.counterSubscription = counter.subscribe(
      (value) => {
        this.secondes = value;
      },
      (error) => {
        console.log('Uh-oh, an error occurred! : ' + error);
      },
      () => {
        console.log('Observable complete!');
      }
    );
  }
}
```

Le code fonctionne de la même manière qu'avant, mais vous pouvez maintenant y ajouter le code qui évitera les bugs liés aux Observables.
Vous allez implémenter une nouvelle interface, `OnDestroy` (qui s'importe depuis `@angular/core`), avec sa fonction `ngOnDestroy()` qui se déclenche quand un component est détruit :

```
export class AppComponent implements OnInit, OnDestroy {
```

```
  secondes: number;

  counterSubscription: Subscription;


  ngOnInit() {
    const counter = Observable.interval(1000);
    this.counterSubscription = counter.subscribe(
      (value) => {
        this.secondes = value;
      },
      (error) => {
        console.log('Uh-oh, an error occurred! : ' + error);
      },
      () => {
        console.log('Observable complete!');
      }
    );
  }


  ngOnDestroy() {
    this.counterSubscription.unsubscribe();
  }
}
```

La fonction `unsubscribe()` détruit la souscription et empêche les comportements inattendus liés aux Observables infinis, donc n'oubliez pas de unsubscribe !

Subjects

Il existe un type d'Observable qui permet non seulement de réagir à de nouvelles informations, mais également d'en émettre. Imaginez une variable dans un service, par exemple, qui peut être modifié depuis plusieurs components ET qui fera réagir tous les components qui y sont liés en même temps. Voici l'intérêt des Subjects.

Pour l'application des appareils électriques, l'utilisation d'un Subject pour gérer la mise à jour des appareils électriques permettra de mettre en place un niveau d'abstraction afin d'éviter des bugs potentiels avec la manipulation de données. Pour l'instant, l'array dans `AppareilViewComponent` est une référence directe à l'array dans `AppareilService`. Dans ce cas précis, cela fonctionne, mais dans une application plus complexe, cela peut créer des problèmes de gestion de données. De plus, le service ne peut rien refuser : l'array peut être modifié directement depuis n'importe quel endroit du code. Pour corriger cela, il y a plusieurs étapes :

- rendre l'array des appareils `private` ;
- créer un Subject dans le service ;
- créer une méthode qui, quand le service reçoit de nouvelles données, fait émettre ces données par le Subject et appeler cette méthode dans toutes les méthodes qui en ont besoin ;
- souscrire à ce Subject depuis `AppareilViewComponent` pour recevoir les données émises, émettre les premières données, et implémenter `OnDestroy` pour détruire la souscription.

Première étape (dans `AppareilService`) :

```
private appareils = [
  {
    id: 1,
    name: 'Machine à laver',
```

```
      status: 'éteint'
    },
    {
      id: 2,
      name: 'Frigo',
      status: 'allumé'
    },
    {
      id: 3,
      name: 'Ordinateur',
      status: 'éteint'
    }
  ];
```

Deuxième étape (dans `AppareilService`) :

```
import { Subject } from 'rxjs/Subject';

export class AppareilService {

  appareilsSubject = new Subject<any[]>();

  private appareils = [
```

Quand vous déclarez un Subject, il faut dire quel type de données il gèrera. Puisque nous n'avons pas créé d'interface pour les appareils (vous pouvez le faire si vous le souhaitez), il gèrera des array de type `any[]`. N'oubliez pas l'import !

Troisième étape, toujours dans `AppareilService` :

```
emitAppareilSubject() {
    this.appareilsSubject.next(this.appareils.slice());
  }

switchOnAll() {
    for(let appareil of this.appareils) {
      appareil.status = 'allumé';
    }
    this.emitAppareilSubject();
}

switchOffAll() {
    for(let appareil of this.appareils) {
      appareil.status = 'éteint';
      this.emitAppareilSubject();
    }
}

switchOnOne(i: number) {
    this.appareils[i].status = 'allumé';
    this.emitAppareilSubject();
}

switchOffOne(i: number) {
    this.appareils[i].status = 'éteint';
    this.emitAppareilSubject();
}
```

Dernière étape, dans `AppareilViewComponent` :

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { AppareilService } from '../services/appareil.service';
import { Subscription } from 'rxjs/Subscription';

@Component({
  selector: 'app-appareil-view',
  templateUrl: './appareil-view.component.html',
  styleUrls: ['./appareil-view.component.scss']
})
export class AppareilViewComponent implements OnInit, OnDestroy {

  appareils: any[];
  appareilSubscription: Subscription;

lastUpdate = new Promise((resolve, reject) => {
  const date = new Date();
  setTimeout(
    () => {
      resolve(date);
    }, 2000
  );
});

constructor(private appareilService: AppareilService) { }

ngOnInit() {
  this.appareilSubscription = this.appareilService.appareilsSubject.subsc
```

```
    (appareils: any[]) => {
      this.appareils = appareils;
    }
  );
  this.appareilService.emitAppareilSubject();
}

onAllumer() {
  this.appareilService.switchOnAll();
}

onEteindre() {
  if(confirm('Etes-vous sûr de vouloir éteindre tous vos appareils ?')) {
    this.appareilService.switchOffAll();
  } else {
    return null;
  }
}

  ngOnDestroy() {
    this.appareilSubscription.unsubscribe();
  }

}
```

L'application refonctionne comme avant, mais avec une différence cruciale de méthodologie : il y a une abstraction entre le service et les components, où les données sont maintenues à jour grâce au Subject.

Pourquoi est-ce important ?

Dans ce cas précis, ce n'est pas fondamentalement nécessaire, mais imaginez qu'on intègre un système qui vérifie périodiquement le statut des appareils. Si les données sont mises à jour par une autre partie de l'application, il faut que l'utilisateur voie ce changement sans avoir à recharger la page. Il va de même dans l'autre sens : un changement au niveau du view doit pouvoir être reflété par le reste de l'application sans rechargement.

Opérateurs

L'API RxJS propose énormément de possibilités — beaucoup trop pour tout voir dans ce cours. Cependant, j'aimerais vous parler rapidement de l'existence des opérateurs.

Un opérateur est une fonction qui se place entre l'Observable et l'Observer (la Subscription, par exemple), et qui peut filtrer et/ou modifier les données reçues avant même qu'elles n'arrivent à la Subscription. Voici quelques exemples rapides :

- `map()` : modifie les valeurs reçues — peut effectuer des calculs sur des chiffres, transformer du texte, créer des objets…
- `filter()` : comme son nom l'indique, filtre les valeurs reçues selon la fonction qu'on lui passe en argument.
- `throttleTime()` : impose un délai minimum entre deux valeurs — par exemple, si un Observable émet cinq valeurs par seconde, mais ce sont uniquement les valeurs reçues toutes les secondes qui vous intéressent, vous pouvez passer `throttleTime(1000)` comme opérateur.
- `scan()` et `reduce()` : permettent d'exécuter une fonction qui réunit l'ensemble des valeurs reçues selon une fonction que vous lui passez — par exemple, vous pouvez faire la somme de toutes les valeurs reçues. La différence basique entre les deux opérateurs : `reduce()` vous retourne uniquement la valeur finale, alors que `scan()`

retourne chaque étape du calcul.

Ce chapitre n'est qu'une introduction au monde des Observables RxJS pour vous présenter les éléments qui peuvent vous être utiles très rapidement et ceux qui sont intégrés dans certains services Angular. Pour en savoir plus, n'hésitez pas à consulter la documentation sur le site ReactiveX.

Écoutez l'utilisateur avec les Forms - méthode template

Jusqu'ici, dans ce cours, vous avez appris à créer une application dynamique qui permet d'échanger des informations entre components à l'aide des services et des Observables, qui change l'affichage selon le routing et qui réagit à certains événements provenant de l'utilisateur, comme des clics sur un bouton, par exemple. Cependant, pour l'instant, toutes les données que vous avez utilisées ont été codées "en dur" dans l'application, c'est-à-dire fournies ni par l'utilisateur, ni par un serveur. Dans les chapitres suivants, vous allez apprendre à interagir avec l'utilisateur et avec les serveurs afin de créer une application totalement dynamique.

En Angular, il y a deux grandes méthodes pour créer des formulaires :

- la méthode **template** : vous créez votre formulaire dans le template, et Angular l'analyse pour comprendre les différents inputs et pour en mettre à disposition le contenu ;

- la méthode **réactive** : vous créez votre formulaire en TypeScript et dans le template, puis vous en faites la liaison manuellement — cette approche est plus complexe, mais elle permet beaucoup plus de contrôle et une approche dynamique.

N'oubliez pas d'importer FormsModule dans AppModule si ce n'est pas déjà fait !

Dans la suite de ce chapitre nous allons découvrir la méthode template.

Créez le formulaire

Pour comprendre et pratiquer cette méthode, vous allez créer un nouveau component `EditAppareilComponent` qui permettra à l'utilisateur d'enregistrer un nouvel appareil électrique :

```
<div class="row">
  <div class="col-sm-8 col-sm-offset-2">
    <form>
      <div class="form-group">
        <label for="name">
          Nom de l'appareil
        </label>
        <input type="text" id="name" class="form-control">
      </div>
      <div class="form-group">
        <label for="status">
          État de l'appareil
        </label>
```

```
        <select id="status" class="form-control">
          <option value="allumé">Allumé</option>
          <option value="éteint">Éteint</option>
        </select>
      </div>
      <button class="btn btn-primary">Enregistrer</button>
    </form>
  </div>
</div>
```

Angular parcourt votre template et trouve la balise `<form>`, créant ainsi un objet qui sera utilisable depuis votre code TypeScript. Avant de passer à la soumission du formulaire, il faut signaler à Angular quels inputs correspondront à des `controls`, c'est-à-dire des champs dont le contenu est à soumettre. Pour cela, il suffit d'ajouter deux attributs aux inputs en question : un attribut name, qui correspondra à la clef de la paire clef-valeur qui sera rendu, et l'attribut `ngModel`, sans parenthèses ni crochets. Ce deuxième attribut signale à Angular que vous souhaitez enregistrer ce contrôle :

```
<div class="form-group">
    <label for="name">
      Nom de l'appareil
    </label>
    <input type="text" id="name" class="form-control" name="name" ngModel>
</div>
<div class="form-group">
    <label for="status">
      État de l'appareil
    </label>
    <select id="status" class="form-control" name="status" ngModel>
```

```
      <option value="allumé">Allumé</option>
      <option value="éteint">Éteint</option>
    </select>
</div>
```

Il faut maintenant préparer la gestion de la soumission de ce formulaire. Dans votre template, au lieu d'ajouter un événement sur le bouton, vous allez déclarer le bouton de type `submit`, et ajouter le code suivant dans la balise `<form>` :

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
<button class="btn btn-primary" type="submit">Enregistrer</button>
```

Déclarer le bouton de type submit à l'intérieur du `<form>` déclenche le comportement de soumission classique de HTML. En ajoutant l'attribut `(ngSubmit)`, vous recevez cette soumission et exécutez la méthode `onSubmit()` (que vous n'avez pas encore créée).

L'attribut `#f` est ce qu'on appelle une référence locale. Vous donnez simplement un nom à l'objet sur lequel vous ajoutez cet attribut ; ce nom sera ensuite utilisable par Angular. C'est d'ailleurs le cas ici : on appelle le formulaire `f` pour ensuite le passer comme argument à la méthode de soumission.

De manière générale, on ne donne pas de valeur à une référence locale : on écrit simplement `#f` ou `#my-name`. Dans ce cas précis d'un formulaire en méthode template, on y attribue la valeur `ngForm` pour avoir accès à l'objet créé par Angular.

Pour récapituler : quand l'utilisateur clique sur le bouton de type submit, la méthode que vous attribuez à `(ngSubmit)` est exécutée, et grâce à la référence locale `#f="ngForm"`, vous pouvez passer l'objet à la méthode (et donc la récupérer dans votre code TypeScript).

Créez maintenant la méthode `onSubmit()` afin de recevoir les informations venant du formulaire. Pour l'instant, vous allez simplement les afficher dans la console :

```
onSubmit(form: NgForm) {
    console.log(form.value);
}
```

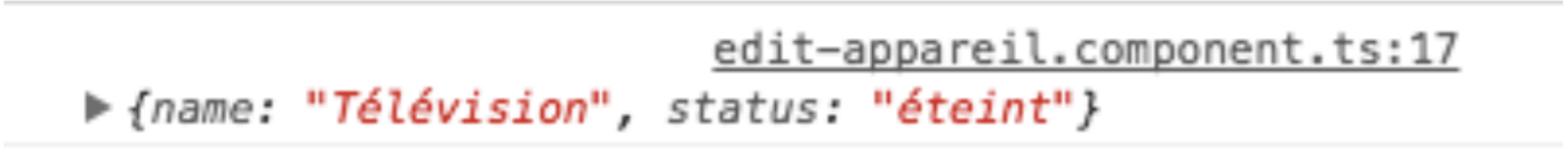
Ici vous utilisez la propriété `value` du `NgForm`. L'objet `NgForm` (à importer depuis `@angular/forms`) comporte beaucoup de propriétés très intéressantes ; je n'en expliquerai que certaines dans ce cours, mais vous pouvez en trouver la liste complète dans la documentation officielle.

Pour avoir accès au formulaire, créez une nouvelle route dans `AppModule` et un `routerLink` correspondant dans la barre de menu :

```
const appRoutes: Routes = [
  { path: 'appareils', canActivate: [AuthGuard], component: AppareilViewCom
  { path: 'appareils/:id', canActivate: [AuthGuard], component: SingleAppar
  { path: 'edit', canActivate: [AuthGuard], component: EditAppareilComponen
  { path: 'auth', component: AuthComponent },
  { path: '', component: AppareilViewComponent },
  { path: 'not-found', component: FourOhFourComponent },
  { path: '**', redirectTo: 'not-found' }
];
```

```
<ul class="nav navbar-nav">
    <li routerLinkActive="active"><a routerLink="auth">Authentification</a>
    <li routerLinkActive="active"><a routerLink="appareils">Appareils</a></
    <li routerLinkActive="active"><a routerLink="edit">Nouvel appareil</a><
</ul>
```

Naviguez vers cette nouvelle page. Si vous tapez "Télévision" comme nom et choisissez "Éteint" et vous cliquez sur "Enregistrer", dans la console vous avez :



Ça y est, vous avez accès aux informations du formulaire ! Avant de les traiter et d'en créer un nouvel appareil, vous allez d'abord ajouter une validation du formulaire, afin que l'utilisateur ne puisse pas le soumettre sans rentrer des données valables.

Validez les données

Pour le formulaire de cette application, on peut dire que le nom de l'appareil est un champ obligatoire. Pour ajouter cette obligation, ajoutez simplement la directive suivante :

```
<input type="text" id="name" class="form-control" name="name" ngModel requi
```

Cet attribut ressemble à un attribut HTML5 classique, mais sachez qu'Angular désactive par défaut le comportement de validation HTML5.

Le champ est obligatoire, mais le formulaire peut encore être soumis, car vous n'avez pas encore intégré l'état de validation du formulaire. Pour accomplir cela, vous allez lier la propriété `disabled` du bouton à la propriété `invalid` du formulaire, qui est mise à disposition par Angular :

```
<button class="btn btn-primary" type="submit" [disabled]="f.invalid">Enregi
```

Vous employez la référence locale `f` pour avoir accès à l'objet `NgForm` , et donc à sa propriété `invalid` , désactivant le bouton si elle retourne `true` .

Il serait intéressant de déclarer l'appareil éteint par défaut afin de ne pas

laisser ce champ vide. Pour déclarer une réponse par défaut, vous allez créer une variable TypeScript et la lier à la propriété `ngModel` du contrôle :

```
export class EditAppareilComponent implements OnInit {


  defaultOnOff = 'éteint';


  constructor() { }
```

```
<select id="status" class="form-control" name="status" [ngModel]="defaultOn
      <option value="allumé">Allumé</option>
      <option value="éteint">Éteint</option>
</select>
```

Ainsi, le formulaire ne pourra être soumis que si le champ "Nom de l'appareil" n'est pas vide, et l'option choisie par défaut empêche le deuxième champ d'être vide également.

Il ne reste plus qu'à exploiter ces données et en créer un nouvel appareil !

Exploitez les données

Dans la méthode `onSubmit()`, vous allez récupérer les données et les attribuer à deux constantes pour les envoyer à `AppareilService` :

```
onSubmit(form: NgForm) {
    const name = form.value['name'];
    const status = form.value['status'];
}
```

Puisque vous savez que le formulaire comportera forcément un champ `name` et un champ `status`, vous savez que vous pouvez utiliser cette syntaxe sans problème.

Créez maintenant la méthode dans `AppareilService` qui générera le nouvel appareil :

```
addAppareil(name: string, status: string) {
    const appareilObject = {
      id: 0,
      name: '',
      status: ''
    };
    appareilObject.name = name;
    appareilObject.status = status;
    appareilObject.id = this.appareils[(this.appareils.length - 1)].id + 1;
    this.appareils.push(appareilObject);
    this.emitAppareilSubject();
}
```

Cette méthode crée un objet du bon format, et attribue le nom et le statut qui lui sont passés comme arguments. La ligne pour l'id prend l'id du dernier élément actuel de l'array et ajoute 1. Ensuite, l'objet complété est ajouté à l'array et le Subject est déclenché pour tout garder à jour.

Il ne reste plus qu'à intégrer cette fonction dans `EditAppareilComponent`. Il serait intéressant qu'une fois l'appareil créé, l'utilisateur soit redirigé vers la liste des appareils. N'oubliez pas d'importer et d'injecter le router pour cela, ainsi qu' `AppareilService` :

```
export class EditAppareilComponent implements OnInit {

  defaultOnOff = 'éteint';

  constructor(private appareilService: AppareilService,
```

```
                private router: Router) { }


  ngOnInit() {
  }


  onSubmit(form: NgForm) {
    const name = form.value['name'];
    const status = form.value['status'];
    this.appareilService.addAppareil(name, status);
    this.router.navigate(['/appareils']);
  }


}
```

Et voilà ! Maintenant, grâce à un formulaire simple géré par la méthode template, l'utilisateur peut créer un nouvel appareil pour la liste.

Écoutez l'utilisateur avec les Forms - méthode réactive

Préparez le terrain

À la différence de la méthode template où Angular crée l'objet du formulaire, pour la méthode réactive, vous devez le créer vous-même et le relier à votre template. Même si cela a l'air plus complexe, cela vous permet de gérer votre formulaire en détail, notamment avec la création programmatique de contrôles (permettant, par exemple, à l'utilisateur d'ajouter des champs).

Pour illustrer la méthode réactive, vous allez créer une nouvelle section dans l'application des appareils électriques : vous allez permettre aux utilisateurs de créer un profil utilisateur simple. Cette démonstration

utilisera des compétences que vous avez apprises tout au long de ce cours, et vous allez également créer votre premier modèle de données sous forme d'une classe TypeScript.

Commencez par le modèle `User` ; créez un nouveau dossier `models` , et dedans un fichier `User.model.ts` :

```
export class User {
  constructor(
    public firstName: string,
    public lastName: string,
    public email: string,
    public drinkPreference: string,
    public hobbies?: string[]
  ) {}
}
```

Ce modèle pourra donc être utilisé dans le reste de l'application en l'important dans les components où vous en avez besoin. Cette syntaxe de constructeur permet l'utilisation du mot-clé `new` , et les arguments passés seront attribués à des variables qui portent les noms choisis ici, par exemple `const user = new User('James', 'Smith', 'james@james.com', 'jus d\'orange', ['football', 'lecture'])` créera un nouvel objet `User` avec ces valeurs attribuées aux variables `user.firstName` , `user.lastName` etc.

Ensuite, créez un `UserService` simple qui stockera la liste des objets `User` et qui comportera une méthode permettant d'ajouter un utilisateur à la liste :

```
import { User } from '../models/User.model';
import { Subject } from 'rxjs/Subject';
```

```
export class UserService {

  private users: User[];

  userSubject = new Subject<User[]>();


  emitUsers() {

    this.userSubject.next(this.users.slice());

  }


  addUser(user: User) {

    this.users.push(user);

    this.emitUsers();

  }

}
```

Ce service contient un array privé d'objets de type personnalisé `User` et un Subject pour émettre cet array. La méthode `emitUsers()` déclenche ce Subject et la méthode `addUser()` ajoute un objet `User` à l'array, puis déclenche le Subject.

N'oubliez pas d'ajouter ce nouveau service à l'array `providers` dans `AppModule` !

L'étape suivante est de créer `UserListComponent` — pour simplifier cet exemple, vous n'allez pas créer un component pour les utilisateurs individuels :

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

import { User } from '../models/User.model';

import { Subscription } from 'rxjs/Subscription';

import { UserService } from '../services/user.service';
```

```
@Component({
  selector: 'app-user-list',
  templateUrl: './user-list.component.html',
  styleUrls: ['./user-list.component.scss']
})
export class UserListComponent implements OnInit, OnDestroy {

  users: User[];
  userSubscription: Subscription;

  constructor(private userService: UserService) { }

  ngOnInit() {
    this.userSubscription = this.userService.userSubject.subscribe(
      (users: User[]) => {
        this.users = users;
      }
    );
    this.userService.emitUsers();
  }



  ngOnDestroy() {
    this.userSubscription.unsubscribe();
  }

}
```

Ce component très simple souscrit au Subject dans `UserService` et le déclenche pour en récupérer les informations et les rendre disponibles au template (que vous allez maintenant créer) :

```
<ul class="list-group">
  <li class="list-group-item" *ngFor="let user of users">
    <h3>{{ user.firstName }} {{ user.lastName }}</h3>
    <p>{{ user.email }}</p>
    <p>Cette persone préfère le {{ user.drinkPreference }}</p>
    <p *ngIf="user.hobbies && user.hobbies.length > 0">
      Cette personne a des hobbies !
      <span *ngFor="let hobby of user.hobbies">{{ hobby }} - </span>
    </p>
  </li>
</ul>
```

Ici, vous appliquez des directives `*ngFor` et `*ngIf` pour afficher la liste des utilisateurs et leurs hobbies, s'ils en ont.

Afin de pouvoir visualiser ce nouveau component, ajoutez une route `users` dans `AppModule` , et créez un `routerLink` . Ajoutez également un objet `User` codé en dur dans le service pour voir les résultats :

```
const appRoutes: Routes = [
  { path: 'appareils', canActivate: [AuthGuard], component: AppareilViewCom
  { path: 'appareils/:id', canActivate: [AuthGuard], component: SingleAppar
  { path: 'edit', canActivate: [AuthGuard], component: EditAppareilComponen
  { path: 'auth', component: AuthComponent },
  { path: 'users', component: UserListComponent },
  { path: '', component: AppareilViewComponent },
  { path: 'not-found', component: FourOhFourComponent },
```

```
  { path: '**', redirectTo: 'not-found' }
];

<ul class="nav navbar-nav">
    <li routerLinkActive="active"><a routerLink="auth">Authentification</a>
    <li routerLinkActive="active"><a routerLink="appareils">Appareils</a></
    <li routerLinkActive="active"><a routerLink="edit">Nouvel appareil</a><
    <li routerLinkActive="active"><a routerLink="users">Utilisateurs</a></l
</ul>

private users: User[] = [
    new User('Will', 'Alexander', 'will@will.com', 'jus d\'orange', ['coder
];
```

Notez que j'ai choisi de ne pas protéger la route avec `AuthGuard` afin d'accélérer le processus.

Dernière étape : il faut ajouter `ReactiveFormsModule`, importé depuis `@angular/forms`, à l'array `imports` de votre `AppModule` :

```
imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    RouterModule.forRoot(appRoutes)
],
```

Maintenant que tout est prêt, vous allez créer `NewUserComponent` qui contiendra votre formulaire réactif.

Construisez un formulaire avec FormBuilder

Dans la méthode template, l'objet formulaire mis à disposition par Angular

était de type `NgForm`, mais ce n'est pas le cas pour les formulaires réactifs. Un formulaire réactif est de type `FormGroup`, et il regroupe plusieurs `FormControl` (tous les deux importés depuis `@angular/forms`). Vous commencez d'abord, donc, par créer l'objet dans votre nouveau component `NewUserComponent` :

```
import { Component, OnInit } from '@angular/core';
import { FormGroup } from '@angular/forms';


@Component({
  selector: 'app-new-user',
  templateUrl: './new-user.component.html',
  styleUrls: ['./new-user.component.scss']
})
export class NewUserComponent implements OnInit {

  userForm: FormGroup;


  constructor() { }


  ngOnInit() {
  }


}
```

Ensuite, vous allez créer une méthode qui sera appelée dans le constructeur pour la population de cet objet, et vous allez également injecter `FormBuilder`, importé depuis `@angular/forms` :

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

```
import { FormBuilder, FormGroup } from '@angular/forms';


@Component({
  selector: 'app-new-user',
  templateUrl: './new-user.component.html',
  styleUrls: ['./new-user.component.scss']
})
export class NewUserComponent implements OnInit {

  userForm: FormGroup;

  constructor(private formBuilder: FormBuilder) { }

  ngOnInit() {
    this.initForm();
  }

  initForm() {

  }

}
```

`FormBuilder` est une classe qui vous met à disposition des méthodes facilitant la création d'objet `FormGroup`. Vous allez maintenant utiliser la méthode `group` à l'intérieur de votre méthode `initForm()` pour commencer à créer le formulaire :

```
initForm() {
    this.userForm = this.formBuilder.group({
```

```
      firstName: '',
      lastName: '',
      email: '',
      drinkPreference: ''
    });
}
```

La méthode `group` prend comme argument un objet où les clés correspondent aux noms des contrôles souhaités et les valeurs correspondent aux valeurs par défaut de ces champs. Puisque l'objectif est d'avoir des champs vides au départ, chaque valeur ici correspond au string vide.

Les contrôles correspondants aux hobbies seront ajoutés par la suite avec une autre méthode.

Il faut maintenant créer le template du formulaire et lier ce template à l'objet `userForm` que vous venez de créer :

```
<div class="col-sm-8 col-sm-offset-2">
  <form [formGroup]="userForm" (ngSubmit)="onSubmitForm()">
    <div class="form-group">
      <label for="firstName">Prénom</label>
      <input type="text" id="firstName" class="form-control" formControlNam
    </div>
    <div class="form-group">
      <label for="lastName">Nom</label>
      <input type="text" id="lastName" class="form-control" formControlName
    </div>
    <div class="form-group">
      <label for="email">Adresse e-mail</label>
```

```
      <input type="text" id="email" class="form-control" formControlName="e
    </div>
    <div class="form-group">
      <label for="drinkPreference">Quelle boisson préférez-vous ?</label>
      <select id="drinkPreference" class="form-control" formControlName="dr
        <option value="jus d\'orange">Jus d'orange</option>
        <option value="jus de mangue">Jus de mangue</option>
      </select>
    </div>
    <button type="submit" class="btn btn-primary">Soumettre</button>
  </form>
</div>
```

Analysez le template :

- Sur la balise `<form>` , vous utilisez le property binding pour lier l'objet `userForm` à l'attribut `formGroup` du formulaire, créant la liaison pour Angular entre le template et le TypeScript.
- Également dans la balise `<form>` , vous avez toujours une méthode `onSubmitForm()` liée à ngSubmit, mais vous n'avez plus besoin de passer le formulaire comme argument puisque vous y avez déjà accès par l'objet `userForm` que vous avez créé.
- Sur chaque `<input>` qui correspond à un `control` du formulaire, vous ajoutez l'attribut `formControlName` où vous passez un string correspondant au nom du `control` dans l'objet TypeScript.
- Le bouton de type `submit` déclenche l'événement `ngSubmit` , déclenchant ainsi la méthode `onSubmitForm()` , que vous allez créer dans votre TypeScript.

Pour tout mettre ensemble, injectez `UserService` et `Router` (sans oublier

de les importer) dans le constructeur du component, et créez la méthode `onSubmitForm()` :

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';
import { UserService } from '../services/user.service';
import { Router } from '@angular/router';
import { User } from '../models/User.model';

@Component({
  selector: 'app-new-user',
  templateUrl: './new-user.component.html',
  styleUrls: ['./new-user.component.scss']
})
export class NewUserComponent implements OnInit {

userForm: FormGroup;

constructor(private formBuilder: FormBuilder,
            private userService: UserService,
            private router: Router) { }

ngOnInit() {
  this.initForm();
}

initForm() {
  this.userForm = this.formBuilder.group({
    firstName: '',
```

```
    lastName: '',
    email: '',
    drinkPreference: ''
  });
}

onSubmitForm() {
  const formValue = this.userForm.value;
  const newUser = new User(
    formValue['firstName'],
    formValue['lastName'],
    formValue['email'],
    formValue['drinkPreference']
  );
  this.userService.addUser(newUser);
  this.router.navigate(['/users']);
  }

}
```

La méthode `onSubmitForm()` récupère la `value` du formulaire, et crée un nouvel objet `User` (à importer en haut) à partir de la valeur des `controls` du formulaire. Ensuite, elle ajoute le nouvel utilisateur au service et navigue vers `/users` pour en montrer le résultat.

Il ne reste plus qu'à ajouter un lien dans `UserListComponent` qui permet d'accéder à `NewUserComponent` et de créer la route correspondante `new-user` dans `AppModule` :

```
<ul class="list-group">
  <li class="list-group-item" *ngFor="let user of users">
```

```
    <h3>{{ user.firstName }} {{ user.lastName }}</h3>
    <p>{{ user.email }}</p>
    <p>Cette persone préfère le {{ user.drinkPreference }}</p>
    <p *ngIf="user.hobbies && user.hobbies.length > 0">
      Cette personne a des hobbies !
      <span *ngFor="let hobby of user.hobbies">{{ hobby }} - </span>
    </p>
  </li>
  <a routerLink="/new-user">Nouvel utilisateur</a>
</ul>
```

Puisque ce `routerLink` se trouve à l'intérieur du `router-outlet`, il faut ajouter un `/` au début de l'URL pour naviguer vers `localhost:4200/new-user`. Si vous ne mettez pas le `/`, ce lien naviguera vers `localhost:4200/users/new-user`

```
{ path: 'users', component: UserListComponent },
{ path: 'new-user', component: NewUserComponent },
```

Validators

Comme pour la méthode template, il existe un outil pour la validation de données dans la méthode réactive : les `Validators`. Pour ajouter la validation, vous allez modifier légèrement votre exécution de `FormBuilder.group` :

```
initForm() {
    this.userForm = this.formBuilder.group({
      firstName: ['', Validators.required],
      lastName: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]],
```

```
      drinkPreference: ['', Validators.required]
    });
}
```

Plutôt qu'un string simple, vous passez un array à chaque `control`, avec comme premier élément la valeur par défaut souhaitée, et comme deuxième élément le ou les `Validators` (dans un array s'il y en a plusieurs) souhaités. Il faut également importer `Validators` depuis `@angular/forms`. Dans ce cas de figure, tous les champs sont requis et la valeur du champ `email` doit être sous un format valable d'adresse mail (la validité de l'adresse elle-même n'est forcément pas évaluée).

Même si les `Validators` sont des fonctions, il ne faut pas ajouter les parenthèses `()` en les déclarant ici. Les déclarations de `Validators` dans `FormBuilder` informent Angular de la validation souhaitée : Angular s'occupe ensuite d'exécuter ces fonctions au bon moment.

En liant la validité de `userForm` à la propriété `disabled` du bouton `submit`, vous intégrez la validation de données :

```
<button type="submit" class="btn btn-primary" [disabled]="userForm.invalid"
```

Dans ce chapitre, vous avez vu les Validators `required` et `email` : il en existe d'autres, et vous avez également la possibilité de créer des Validators personnalisés. Pour plus d'informations, référez-vous à la section correspondante de la documentation Angular.

Ajoutez dynamiquement des FormControl

Pour l'instant, vous n'avez pas encore laissé la possibilité à l'utilisateur d'ajouter ses hobbies. Il serait intéressant de lui laisser la possibilité d'en ajouter autant qu'il veut, et pour cela, vous allez utiliser un `FormArray`. Un `FormArray` est un array de plusieurs `FormControl`, et permet, par exemple, d'ajouter des nouveaux `controls` à un formulaire. Vous allez

utiliser cette méthode pour permettre à l'utilisateur d'ajouter ses hobbies.

Modifiez d'abord `initForm()` pour ajouter un `FormArray` vide qui s'appellera hobbies avec la méthode `array` :

```
initForm() {
    this.userForm = this.formBuilder.group({
      firstName: ['', Validators.required],
      lastName: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]],
      drinkPreference: ['', Validators.required],
      hobbies: this.formBuilder.array([])
    });
}
```

Modifiez ensuite `onSubmitForm()` pour récupérer les valeurs, si elles existent (sinon, retournez un array vide) :

```
onSubmitForm() {
    const formValue = this.userForm.value;
    const newUser = new User(
      formValue['firstName'],
      formValue['lastName'],
      formValue['email'],
      formValue['drinkPreference'],
      formValue['hobbies'] ? formValue['hobbies'] : []
    );
    this.userService.addUser(newUser);
    this.router.navigate(['/users']);
}
```

Afin d'avoir accès aux `controls` à l'intérieur de l'array, pour des raisons de typage strict liées à TypeScript, il faut créer une méthode qui retourne `hobbies` par la méthode `get()` sous forme de `FormArray` (`FormArray` s'importe depuis `@angular/forms`) :

```
getHobbies(): FormArray {
    return this.userForm.get('hobbies') as FormArray;
}
```

Ensuite, vous allez créer la méthode qui permet d'ajouter un `FormControl` à `hobbies` , permettant ainsi à l'utilisateur d'en ajouter autant qu'il veut. Vous allez également rendre le nouveau champ requis, afin de ne pas avoir un array de `hobbies` avec des string vides :

```
onAddHobby() {
    const newHobbyControl = this.formBuilder.control(null, Validators.requi
    this.getHobbies().push(newHobbyControl);
}
```

Cette méthode crée un `control` avec la méthode `FormBuilder.control()`, et l'ajoute au `FormArray` rendu disponible par la méthode `getHobbies()`.

Enfin, il faut ajouter une section au template qui permet d'ajouter des hobbies en ajoutant des `<input>` :

```
<div formArrayName="hobbies">
      <h3>Vos hobbies</h3>
      <div class="form-group" *ngFor="let hobbyControl of getHobbies().cont
        <input type="text" class="form-control" [formControlName]="i">
      </div>
      <button type="button" class="btn btn-success" (click)="onAddHobby()">
</div>
```

```
<button type="submit" class="btn btn-primary" [disabled]="userForm.invalid"
```

Analysez cette :

- à la `<div>` qui englobe toute la partie `hobbies`, vous ajoutez l'attribut `formArrayName`, qui correspond au nom choisi dans votre TypeScript ;
- la `<div>` de class `form-group` est ensuite répété pour chaque `FormControl` dans le `FormArray` (retourné par `getHobbies()`, initialement vide, en notant l'index afin de créer un nom unique pour chaque `FormControl` ;
- dans cette `<div>`, vous avec une `<input>` qui prendra comme `formControlName` l'index du `FormControl` ;
- enfin, vous avez le bouton (de type `button` pour l'empêcher d'essayer de soumettre le formulaire) qui déclenche `onAddHobby()`, méthode qui, pour rappel, crée un nouveau `FormControl` (affichant une nouvelle instance de la `<div>` de class `form-group`, et donc créant une nouvelle `<input>`)

Félicitations ! Maintenant, vous savez créer des formulaires par deux méthodes différentes, et comment récupérer les données saisies par l'utilisateur. Pour l'instant, la capacité d'enregistrement et de gestion de ces données a été limitée au service et donc tout est systématiquement remis à zéro à chaque rechargement de l'app. Dans les chapitres suivants, vous allez apprendre à interagir avec un serveur (et puis, plus précisément, avec un backend Firebase) afin de rendre les données permanentes et que votre application soit totalement dynamique.

Interagissez avec un serveur avec HttpClient

Dans une application Angular, vous aurez très souvent besoin de faire des appels à un backend ou à un autre serveur — pour enregistrer ou charger des données, par exemple, ou pour effectuer des calculs ou des modifications de données que vous ne souhaitez pas faire faire par le frontend. Angular met à disposition un service appelé `HttpClient` qui permet de créer et d'exécuter des appels HTTP (fait par AJAX - Asynchronous JavaScript and XML) et de réagir aux informations retournées par le serveur.

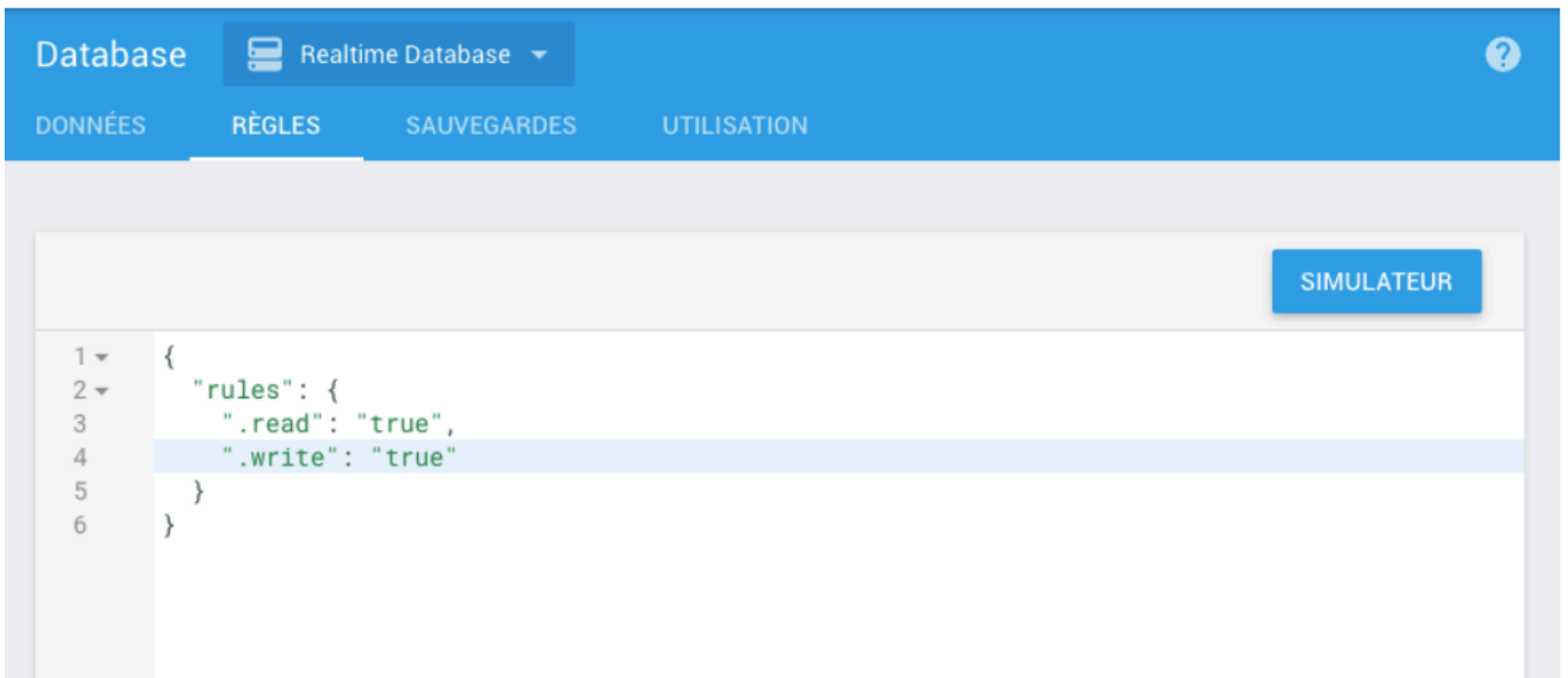
Dans ce chapitre, vous allez configurer un backend avec le service Firebase de Google. Ce service permet la création d'un backend complet sans coder, et node comprend énormément de services, dont l'authentification, une

base de données NoSQL et un stockage de fichiers. Dans un chapitre ultérieur, vous apprendrez à utiliser les fonctions mises à disposition par Firebase afin de mieux intégrer le service. Pour ce chapitre, vous allez simplement utiliser l'API HTTP afin de comprendre l'utilisation de `HttpClient`.

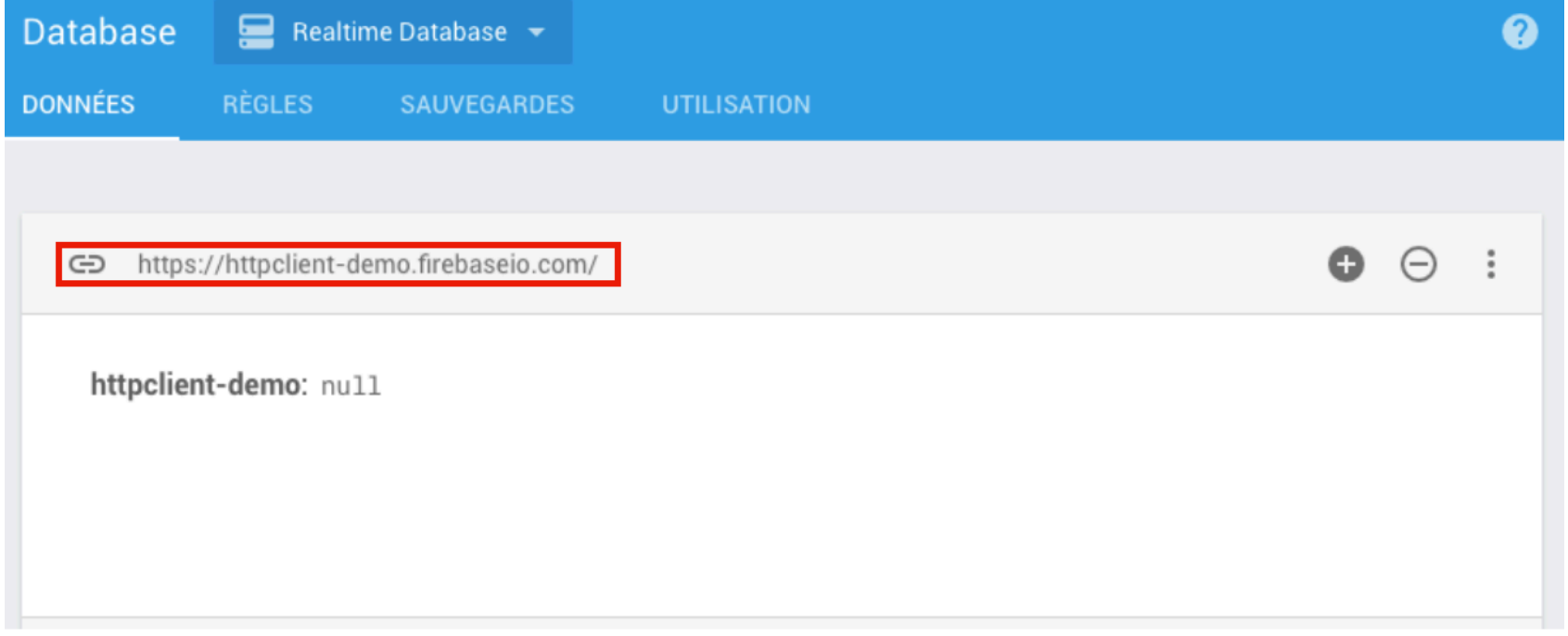
Préparez le backend

Allez à firebase.com, créez un compte Google ou authentifiez-vous si vous en avez déjà un, et créez un nouveau projet Firebase. Vous pouvez le domicilier dans votre pays de résidence et lui donner le nom que vous voulez.

Une fois arrivé sur la console, allez dans Database et choisissez le Realtime Database. Afin d'éviter tout problème d'authentification pour l'instant, allez dans la section "Règles" et définissez `read` et `write` en `true`, puis publiez les règles modifiées :



Revenez à la section Données et notez l'URL de votre base de données, vous allez en avoir besoin pour configurer les appels HTTP :



Le backend est maintenant prêt, et vous allez pouvoir intégrer `HttpClient` à votre application des appareils électriques.

Envoyez vers le backend

Pour avoir accès au service `HttpClient`, il faut tout d'abord ajouter `HttpClientModule`, importé depuis `@angular/common/http`, à votre `AppModule` :

```
imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule,
    RouterModule.forRoot(appRoutes)
],
```

Vous allez utiliser `HttpClient`, dans un premier temps, pour la gestion des données de la liste d'appareils. Vous allez donc l'injecter dans `AppareilService`, en y ayant auparavant ajouté le décorateur `@Injectable()` (importé depuis `@angular/core`) :

```
import { Subject } from 'rxjs/Subject';

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

import { HttpClient } from '@angular/common/http';


@Injectable()

export class AppareilService {


  appareilsSubject = new Subject<any[]>();


  private appareils = [

    {

      id: 1,

      name: 'Machine à laver',

      status: 'éteint'

    },

    {

      id: 2,

      name: 'Frigo',

      status: 'allumé'

    },

    {

      id: 3,

      name: 'Ordinateur',

      status: 'éteint'

    }

  ];


constructor(private httpClient: HttpClient) { }
```

Si votre service n'avait pas encore de constructeur, créez-le pour injecter `HttpClient`.

Vous allez d'abord créer une méthode qui va enregistrer l'array `appareils` dans la base de données au endpoint `/appareils` par la méthode POST :

```
saveAppareilsToServer() {
    this.httpClient
      .post('https://httpclient-demo.firebaseio.com/appareils.json', this.a
      .subscribe(
        () => {
          console.log('Enregistrement terminé !');
        },
        (error) => {
          console.log('Erreur ! : ' + error);
        }
      );
}
```

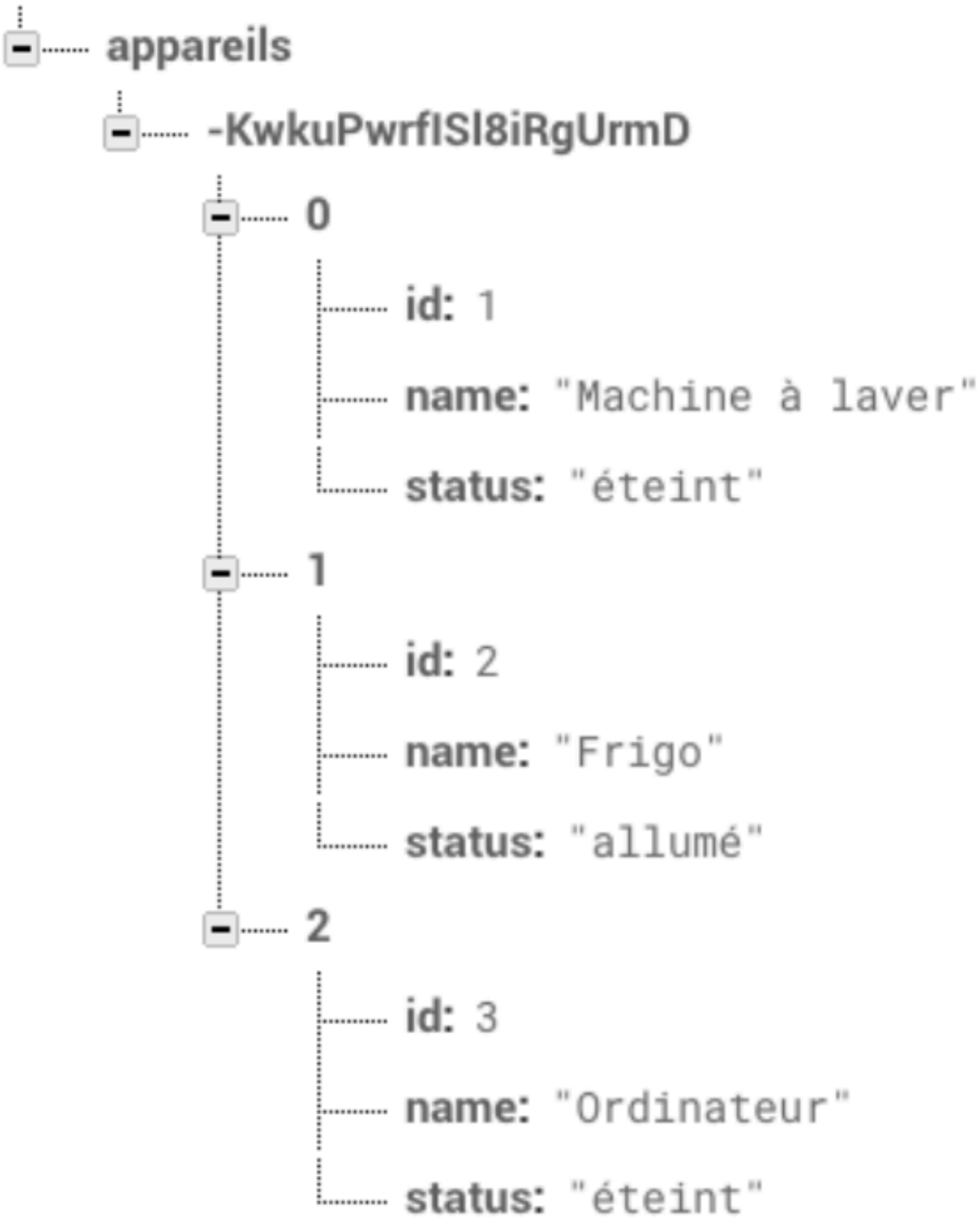
Analysez cette méthode :

- la méthode `post()`, qui permet de lancer un appel POST, prend comme premier argument l'URL visée, et comme deuxième argument le corps de l'appel, c'est-à-dire ce qu'il faut envoyer à l'URL ;
- l'extension `.json` de l'URL est une spécificité Firebase, pour lui dire que vous lui envoyez des données au format JSON ;
- la méthode `post()` retourne un Observable — elle ne fait pas d'appel à elle toute seule. C'est en y souscrivant que l'appel est lancé ;
- dans la méthode `subscribe()`, vous prévoyez le cas où tout fonctionne et le cas où le serveur vous renverrait une erreur.

Créez maintenant un bouton dans `AppareilViewComponent` qui déclenche cette sauvegarde (en passant, si vous ne l'avez pas encore fait, vous pouvez retirer l'activation conditionnelle des boutons "Tout allumer" et "Tout éteindre") :

```
<button class="btn btn-success"
        (click)="onAllumer()">Tout allumer</button>
<button class="btn btn-danger"
        (click)="onEteindre()">Tout éteindre</button>
<button class="btn btn-primary"
        (click)="onSave()">Enregistrer les appareils</button>

onSave() {
    this.appareilService.saveAppareilsToServer();
}
```

Enregistrez le tout, et cliquez sur le bouton que vous venez de créer : vous devez avoir votre message de réussite qui apparait dans la console. Si vous regardez maintenant la console Firebase :



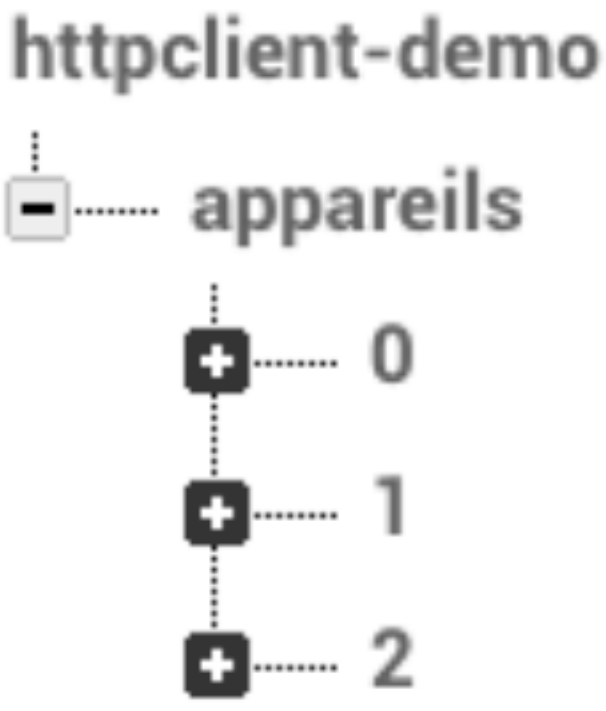
Firebase a créé un nouveau node sous `appareils` avec un identifiant unique, et y a enregistré votre array `appareils`.

Cependant, si vous cliquez plusieurs fois sur ce bouton, Firebase continuera à créer de nouveaux nodes, et dans ce cas de figure, ce n'est pas le comportement souhaité. Il faudrait que chaque enregistrement écrase le

précédent : pour cela, utilisez plutôt la méthode `put()` (il n'y a pas besoin de changer les arguments, car les méthodes `put()` et `post()` prennent les deux mêmes premiers arguments) :

```
saveAppareilsToServer() {

    this.httpClient

      .put('https://httpclient-demo.firebaseio.com/appareils.json', this.ap

      .subscribe(

        () => {

          console.log('Enregistrement terminé !');

        },

        (error) => {

          console.log('Erreur ! : ' + error);

        }

      );

}
```

Maintenant, quand vous enregistrez les données, votre console Firebase montre le node suivant :



Ainsi, vous savez envoyer des données vers un serveur avec les méthodes

POST et PUT. Pour la suite, vous allez intégrer la requête GET pour récupérer et traiter des données depuis le serveur.

Recevez depuis le backend

Afin de demander la liste des appareils (maintenant stocké au endpoint `/appareils`), vous allez créer une nouvelle méthode qui emploie la méthode `get()` dans `AppareilService` :

```
getAppareilsFromServer() {
    this.httpClient
      .get<any[]>('https://httpclient-demo.firebaseio.com/appareils.json')
      .subscribe(
        (response) => {
          this.appareils = response;
          this.emitAppareilSubject();
        },
        (error) => {
          console.log('Erreur ! : ' + error);
        }
      );
}
```

Comme pour `post()` et `put()` , la méthode get() retourne un Observable, mais puisqu'ici, vous allez recevoir des données, TypeScript a besoin de savoir de quel type elles seront (l'objet retourné est d'office considéré comme étant un Object). Vous devez donc, dans ce cas précis, ajouter `<any[]>` pour dire que vous allez recevoir un array de type `any` , et que donc TypeScript peut traiter cet objet comme un array : si vous ne le faites pas, TypeScript vous dira qu'un array ne peut pas être redéfini comme Object.

Vous pouvez maintenant vider l'array `appareils` du service et intégrer un bouton au component permettant de déclencher la méthode `getAppareilsFromServer()` :

```
<button class="btn btn-success"
        (click)="onAllumer()">Tout allumer</button>
<button class="btn btn-danger"
        (click)="onEteindre()">Tout éteindre</button>
<button class="btn btn-primary"
        (click)="onSave()">Enregistrer les appareils</button>
<button class="btn btn-primary"
        (click)="onFetch()">Récupérer les appareils</button>

onFetch() {
    this.appareilService.getAppareilsFromServer();
}
```

Maintenant, vous pouvez ajouter de nouveaux appareils, en modifier l'état et les sauvegarder, puis récupérer la liste sauvegardée.

Il serait également possible de rendre automatique le chargement et l'enregistrement des appareils (par exemple en appelant la méthode `getAppareilsFromServer()` dans `ngOnInit()`, et `saveAppareilsToServer()` après chaque modification), mais j'ai souhaité vous laisser la possibilité de les exécuter manuellement afin de voir le résultat de manière plus concrète.

Dans ce chapitre, vous avez appris à passer des appels à un serveur HTTP avec le service `HttpClient`. Vous avez utilisé un backend Firebase pour cette démonstration : en effet, Firebase propose une API beaucoup plus flexible que des appels HTTP simples afin de profiter pleinement des services proposés. Dans les prochains chapitres de ce cours, vous allez apprendre à intégrer certains des services Firebase dans votre application.

Créez une application complète avec Angular et Firebase

Pour cette section, vous allez créer une nouvelle application et appliquer des connaissances que vous avez apprises tout au long du cours Angular, ainsi que quelques fonctionnalités que vous n'avez pas encore rencontrées. Vous allez créer une application simple qui recense les livres que vous avez chez vous, dans votre bibliothèque. Vous pourrez ajouter une photo de chaque livre. L'utilisateur devra être authentifié pour utiliser l'application.

Malgré sa popularité, j'ai choisi de ne pas intégrer AngularFire dans ce cours. Si vous souhaitez en savoir plus, vous trouverez plus d'informations sur la page GitHub d'AngularFire. Vous emploierez l'API JavaScript mise à disposition directement par Firebase.

Pensez à la structure de l'application

Prenez le temps de réfléchir à la construction de l'application. Quels seront les components dont vous aurez besoin ? Les services ? Les modèles de données ?

L'application nécessite l'authentification. Il faudra donc un component pour la création d'un nouvel utilisateur, et un autre pour s'authentifier, avec un service gérant les interactions avec le backend.

Les livres pourront être consultés sous forme d'une liste complète, puis individuellement. Il faut également pouvoir ajouter et supprimer des livres. Il faudra donc un component pour la liste complète, un autre pour la vue individuelle et un dernier comportant un formulaire pour la création/modification. Il faudra un service pour gérer toutes les fonctionnalités liées à ces components, y compris les interactions avec le serveur.

Vous créerez également un component séparé pour la barre de navigation afin d'y intégrer une logique séparée.

Pour les modèles de données, il y aura un modèle pour les livres, comportant simplement le titre, le nom de l'auteur et la photo, qui sera facultative.

Il faudra également ajouter du routing à cette application, permettant l'accès aux différentes parties, avec une guard pour toutes les routes sauf l'authentification, empêchant les utilisateurs non authentifiés d'accéder à la bibliothèque.

Allez, c'est parti !

Structurez l'application

Pour cette application, je vous conseille d'utiliser le CLI pour la création des components. L'arborescence sera la suivante :

```
ng g c auth/signup
ng g c auth/signin
ng g c book-list
ng g c book-list/single-book
ng g c book-list/book-form
ng g c header
ng g s services/auth
ng g s services/books
ng g s services/auth-guard
```

Les services ainsi créés ne sont pas automatiquement mis dans l'array `providers` d'`AppModule`, donc ajoutez-les maintenant. Pendant que vous travaillez sur `AppModule`, ajoutez également `FormsModule`, `ReactiveFormsModule` et `HttpClientModule` :

```
imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule
  ],
providers: [AuthService, BooksService, AuthGuardService],
```

N'oubliez pas d'ajouter les imports en haut du fichier !

Intégrez dès maintenant le routing sans guard afin de pouvoir accéder à toutes les sections de l'application pendant le développement :

```
const appRoutes: Routes = [
  { path: 'auth/signup', component: SignupComponent },
  { path: 'auth/signin', component: SigninComponent },
  { path: 'books', component: BookListComponent },
  { path: 'books/new', component: BookFormComponent },
  { path: 'books/view/:id', component: SingleBookComponent }
```

```
];

imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule,
    RouterModule.forRoot(appRoutes)
],
```

Générez également un dossier appelé `models` et créez-y le fichier `book.model.ts` :

```
export class Book {
  photo: string;
  synopsis: string;
  constructor(public title: string, public author: string) {
  }
}
```

Avant de lancer `ng serve` , utilisez NPM pour ajouter Bootstrap à votre projet, et ajoutez-le à l'array styles de `.angular-cli.json` :

```
npm install bootstrap@3.3.7 --save

"styles": [
        "../node_modules/bootstrap/dist/css/bootstrap.css",
        "styles.css"
  ],
```

Enfin, préparez `HeaderComponent` avec un menu de navigation, avec les `routerLink` et AppComponent qui l'intègre avec le `router-outlet` :

```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <ul class="nav navbar-nav">
      <li routerLinkActive="active">
        <a routerLink="books">Livres</a>
      </li>
    </ul>
    <ul class="nav navbar-nav navbar-right">
      <li routerLinkActive="active">
        <a routerLink="auth/signup">Créer un compte</a>
      </li>
      <li routerLinkActive="active">
        <a routerLink="auth/signin">Connexion</a>
      </li>
    </ul>
  </div>
</nav>

<app-header></app-header>
<div class="container">
  <router-outlet></router-outlet>
</div>
```

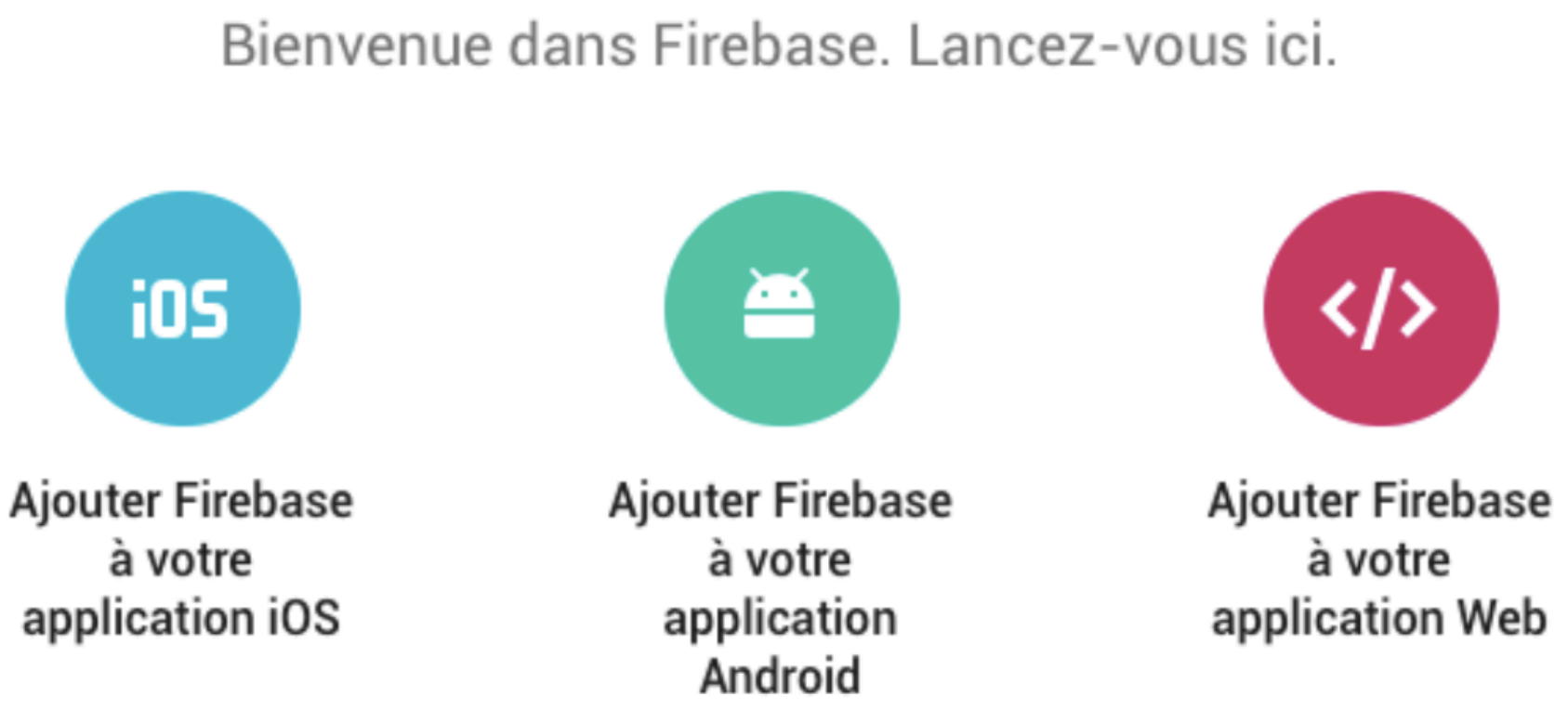
La structure globale de l'application est maintenant prête !

Intégrez Firebase à votre application

D'abord, installez Firebase avec NPM :

```
npm install firebase --save- console output -
```

Pour cette application, vous allez créer un nouveau projet sur Firebase. Une fois l'application créée, la console Firebase vous propose le choix suivant (sous la rubrique Overview) :



Choisissez "Ajouter Firebase à votre application Web" et copiez-collez la configuration dans le constructeur de votre `AppComponent` (en ajoutant `import * as firebase from 'firebase';` en haut du fichier, mettant à disposition la méthode `initializeApp()`) :

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor() {
    const config = {
```

```
      apiKey: 'AIzaSyCwfa_fKNCVrDMR1E88S79mpQP-6qertew4',
      authDomain: 'bookshelves-3d570.firebaseapp.com',
      databaseURL: 'https://bookshelves-3d570.firebaseio.com',
      projectId: 'bookshelves-3d570',
      storageBucket: 'bookshelves-3d570.appspot.com',
      messagingSenderId: '6634573823'
    };
    firebase.initializeApp(config);
  }
}
```

Votre application Angular est maintenant liée à votre projet Firebase, et vous pourrez maintenant intégrer tous les services dont vous aurez besoin.

Authentification

Votre application utilisera l'authentification par adresse mail et mot de passe proposée par Firebase. Pour cela, il faut d'abord l'activer dans la console Firebase :

L'authentification Firebase emploie un système de token : un jeton d'authentification est stocké dans le navigateur, et est envoyé avec chaque requête nécessitant l'authentification.

Dans `AuthService`, vous allez créer trois méthodes :

- une méthode permettant de créer un nouvel utilisateur ;
- une méthode permettant de connecter un utilisateur existant ;
- une méthode permettant la déconnexion de l'utilisateur.

Puisque les opérations de création, de connexion et de déconnexion sont asynchrones, c'est-à-dire qu'elles n'ont pas un résultat instantané, les méthodes que vous allez créer pour les gérer retourneront des Promise, ce qui permettra également de gérer les situations d'erreur.

Importez Firebase dans `AuthService` :

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


import * as firebase from 'firebase';


@Injectable()
export class AuthService {
```

Ensuite, créez la méthode `createNewUser()` pour créer un nouvel

utilisateur, qui prendra comme argument une adresse mail et un mot de passe, et qui retournera une Promise qui résoudra si la création réussit, et sera rejetée avec le message d'erreur si elle ne réussit pas :

```
createNewUser(email: string, password: string) {
    return new Promise(
      (resolve, reject) => {
        firebase.auth().createUserWithEmailAndPassword(email, password).the
          () => {
            resolve();
          },
          (error) => {
            reject(error);
          }
        );
      }
    );
}
```

Toutes les méthodes liées à l'authentification Firebase se trouvent dans `firebase.auth()`.

Créez également `signInUser()`, méthode très similaire, qui s'occupera de connecter un utilisateur déjà existant :

```
signInUser(email: string, password: string) {
    return new Promise(
      (resolve, reject) => {
        firebase.auth().signInWithEmailAndPassword(email, password).then(
          () => {
            resolve();
```

```
          },
          (error) => {
            reject(error);
          }
        );
      }
    );
}
```

Créez une méthode simple `signOutUser()` :

```
signOutUser() {
    firebase.auth().signOut();
}
```

Ainsi, vous avez les trois fonctions dont vous avez besoin pour intégrer l'authentification dans l'application !

Vous pouvez ainsi créer `SignupComponent` et `SigninComponent` , intégrer l'authentification dans `HeaderComponent` afin de montrer les bons liens, et implémenter `AuthGuard` pour protéger la route `/books` et toutes ses sous-routes.

Commencez par `SignupComponent` afin de pouvoir enregistrer un utilisateur :

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { AuthService } from '../../services/auth.service';
import { Router } from '@angular/router';

@Component({
```

```
  selector: 'app-signup',
  templateUrl: './signup.component.html',
  styleUrls: ['./signup.component.css']
})
export class SignupComponent implements OnInit {

  signupForm: FormGroup;
  errorMessage: string;

  constructor(private formBuilder: FormBuilder,
              private authService: AuthService,
              private router: Router) { }

  ngOnInit() {
    this.initForm();
  }

initForm() {
  this.signupForm = this.formBuilder.group({
    email: ['', [Validators.required, Validators.email]],
    password: ['', [Validators.required, Validators.pattern(/[0-9a-zA-Z]{
  });
}

onSubmit() {
  const email = this.signupForm.get('email').value;
  const password = this.signupForm.get('password').value;

  this.authService.createNewUser(email, password).then(
```

```
      () => {
        this.router.navigate(['/books']);
      },
      (error) => {
        this.errorMessage = error;
      }
    );
  }
}
```

Dans ce component :

- vous générez le formulaire selon la méthode réactive
    - les deux champs, `email` et `password` , sont requis — le champ `email` utilise `Validators.email` pour obliger un string sous format d'adresse email ; le champ `password` emploie `Validators.pattern` pour obliger au moins 6 caractères alphanumériques, ce qui correspond au minimum requis par Firebase ;
- vous gérez la soumission du formulaire, envoyant les valeurs rentrées par l'utilisateur à la méthode `createNewUser()`
    - si la création fonctionne, vous redirigez l'utilisateur vers `/books` ;
    - si elle ne fonctionne pas, vous affichez le message d'erreur renvoyé par Firebase.

Ci-dessous, vous trouverez le template correspondant :

```
<div class="row">
  <div class="col-sm-8 col-sm-offset-2">
    <h2>Créer un compte</h2>
```

```
<form [formGroup]="signupForm" (ngSubmit)="onSubmit()">
  <div class="form-group">
    <label for="email">Adresse mail</label>
    <input type="text"
           id="email"
           class="form-control"
           formControlName="email">
  </div>
  <div class="form-group">
    <label for="password">Mot de passe</label>
    <input type="password"
           id="password"
           class="form-control"
           formControlName="password">
  </div>
    <button class="btn btn-primary"
            type="submit"
            [disabled]="signupForm.invalid">Créer mon compte</button>
  </form>
  <p class="text-danger">{{ errorMessage }}</p>
 </div>
</div>
```

Il s'agit d'un formulaire selon la méthode réactive comme vous l'avez vu dans le chapitre correspondant. Il y a, en supplément, le paragraphe contenant l'éventuel message d'erreur rendu par Firebase.

Vous pouvez créer un template presque identique pour `SignInComponent` pour la connexion d'un utilisateur déjà existant. Il vous suffit de renommer `signupForm` en `signinForm` et d'appeler la méthode

`signInUser()` plutôt que `createNewUser()`.

Ensuite, vous allez modifier `HeaderComponent` pour afficher de manière contextuelle les liens de connexion, de création d'utilisateur et de déconnexion :

```
import { Component, OnInit } from '@angular/core';
import { AuthService } from '../services/auth.service';
import * as firebase from 'firebase';

@Component({
  selector: 'app-header',
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.css']
})
export class HeaderComponent implements OnInit {

isAuth: boolean;

constructor(private authService: AuthService) { }

ngOnInit() {
  firebase.auth().onAuthStateChanged(
    (user) => {
      if(user) {
        this.isAuth = true;
      } else {
        this.isAuth = false;
      }
    }
```

```
    );
  }

  onSignOut() {
    this.authService.signOutUser();
  }

}
```

```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <ul class="nav navbar-nav">
      <li routerLinkActive="active">
        <a routerLink="books">Livres</a>
      </li>
    </ul>
    <ul class="nav navbar-nav navbar-right">
      <li routerLinkActive="active" *ngIf="!isAuth">
        <a routerLink="auth/signup">Créer un compte</a>
      </li>
      <li routerLinkActive="active" *ngIf="!isAuth">
        <a routerLink="auth/signin">Connexion</a>
      </li>
      <li>
        <a (click)="onSignOut()"
           style="cursor:pointer"
           *ngIf="isAuth">Déconnexion</a>
      </li>
    </ul>
```

```
  </div>

</nav>
```

Ici, vous utilisez `onAuthStateChanged()`, qui permet d'observer l'état de l'authentification de l'utilisateur : à chaque changement d'état, la fonction que vous passez en argument est exécutée. Si l'utilisateur est bien authentifié, `onAuthStateChanged()` reçoit l'objet de type `firebase.User` correspondant à l'utilisateur. Vous pouvez ainsi baser la valeur de la variable locale `isAuth` selon l'état d'authentification de l'utilisateur, et afficher les liens correspondant à cet état.

Il ne vous reste plus qu'à créer `AuthGuardService` et l'appliquer aux routes concernées. Puisque la vérification de l'authentification est asynchrone, votre service retournera une Promise :

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

import { CanActivate, Router } from '@angular/router';

import { Observable } from 'rxjs/Observable';

import * as firebase from 'firebase';


@Injectable()

export class AuthGuardService implements CanActivate {


  constructor(private router: Router) { }


  canActivate(): Observable<boolean> | Promise<boolean> | boolean {

    return new Promise(

      (resolve, reject) => {

        firebase.auth().onAuthStateChanged(

          (user) => {

            if(user) {
```

```
            resolve(true);
          } else {
            this.router.navigate(['/auth', 'signin']);
            resolve(false);
          }
        }
      );
    }
  );
  }
}

const appRoutes: Routes = [
  { path: 'auth/signup', component: SignupComponent },
  { path: 'auth/signin', component: SigninComponent },
  { path: 'books', canActivate: [AuthGuardService], component: BookListComp
  { path: 'books/new', canActivate: [AuthGuardService], component: BookForm
  { path: 'books/view/:id', canActivate: [AuthGuardService], component: Sin
];
```

Ah, mais qu'a-t-on oublié ? Le routing ne prend en compte ni le path vide, ni le path wildcard ! Ajoutez ces routes dès maintenant pour éviter toute erreur :

```
const appRoutes: Routes = [
  { path: 'auth/signup', component: SignupComponent },
  { path: 'auth/signin', component: SigninComponent },
  { path: 'books', canActivate: [AuthGuardService], component: BookListComp
  { path: 'books/new', canActivate: [AuthGuardService], component: BookForm
  { path: 'books/view/:id', canActivate: [AuthGuardService], component: Sin
```

```
  { path: '', redirectTo: 'books', pathMatch: 'full' },
  { path: '**', redirectTo: 'books' }
];
```

Ainsi, votre application comporte un système d'authentification complet, permettant l'inscription et la connexion/déconnexion des utilisateurs, et qui protège les routes concernées. Vous pouvez maintenant ajouter les fonctionnalités à votre application en sachant que les accès à la base de données et au stockage, qui nécessitent l'authentification, fonctionneront correctement.

Base de données

Dans ce chapitre, vous allez créer les fonctionnalités de l'application : la création, la visualisation et la suppression des livres, le tout lié directement à la base de données Firebase.

Pour créer `BooksService` :

- vous aurez un array local `books` et un Subject pour l'émettre ;
- vous aurez des méthodes :
    - pour enregistrer la liste des livres sur le serveur,
    - pour récupérer la liste des livres depuis le serveur,
    - pour récupérer un seul livre,
    - pour créer un nouveau livre,
    - pour supprimer un livre existant.

Pour la première étape, rien de nouveau (sans oublier d'importer Book et Subject) :

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

```
import { Subject } from 'rxjs/Subject';

import { Book } from '../models/book.model';


@Injectable()

export class BooksService {


  books: Book[] = [];

  booksSubject = new Subject<Book[]>();


  emitBooks() {

    this.booksSubject.next(this.books);

  }

}
```

Ensuite, vous allez utiliser une méthode mise à disposition par Firebase pour enregistrer la liste sur un node de la base de données — la méthode `set()` :

```
saveBooks() {

    firebase.database().ref('/books').set(this.books);

}
```

La méthode `ref()` retourne une référence au node demandé de la base de données, et `set()` fonctionne plus ou moins comme `put()` pour le HTTP : il écrit et remplace les données au node donné.

Maintenant que vous pouvez enregistrer la liste, vous allez créer les méthodes pour récupérer la liste entière des livres et pour récupérer un seul livre, en employant les deux fonctions proposées par Firebase :

```
getBooks() {
```

```
  firebase.database().ref('/books')
    .on('value', (data: DataSnapshot) => {
        this.books = data.val() ? data.val() : [];
        this.emitBooks();
      }
    );
}


getSingleBook(id: number) {
  return new Promise(
    (resolve, reject) => {
      firebase.database().ref('/books/' + id).once('value').then(
        (data: DataSnapshot) => {
          resolve(data.val());
        }, (error) => {
          reject(error);
        }
      );
    }
  );
}
```

Pour `getBooks()`, vous utilisez la méthode `on()`. Le premier argument `'value'` demande à Firebase d'exécuter le callback à chaque modification de valeur enregistrée au endpoint choisi : cela veut dire que si vous modifiez quelque chose depuis un appareil, la liste sera automatiquement mise à jour sur tous les appareils connectés. Ajoutez un constructor au service pour appeler `getBooks()` au démarrage de l'application :

```
constructor() {
```

```
    this.getBooks();
}
```

Le deuxième argument est la fonction callback, qui reçoit ici une `DataSnapshot` : un objet correspondant au node demandé, comportant plusieurs membres et méthodes (il faut importer `DataSnapshot` depuis `firebase.database.DataSnapshot`). La méthode qui vous intéresse ici est `val()`, qui retourne la valeur des données, tout simplement. Votre callback prend également en compte le cas où le serveur ne retourne rien pour éviter les bugs potentiels.

La fonction `getSingleBook()` récupère un livre selon son id, qui est simplement ici son index dans l'array enregistré. Vous utilisez `once()`, qui ne fait qu'une seule requête de données. Du coup, elle ne prend pas une fonction callback en argument mais retourne une Promise, permettant l'utilisation de `.then()` pour retourner les données reçues.

Pour BooksService, il ne reste plus qu'à créer les méthodes pour la création d'un nouveau livre et la suppression d'un livre existant :

```
createNewBook(newBook: Book) {
    this.books.push(newBook);
    this.saveBooks();
    this.emitBooks();
  }


  removeBook(book: Book) {
    const bookIndexToRemove = this.books.findIndex(
      (bookEl) => {
        if(bookEl === book) {
          return true;
        }
```

```
    }
  );
  this.books.splice(bookIndexToRemove, 1);
  this.saveBooks();
  this.emitBooks();
}
```

Ensuite, vous allez créer `BookListComponent`, qui :

- souscrit au Subject du service et déclenche sa première émission ;
- affiche la liste des livres, où chaque livre peut être cliqué pour en voir la page `SingleBookComponent` ;
- permet de supprimer chaque livre en utilisant `removeBook()` ;
- permet de naviguer vers `BookFormComponent` pour la création d'un nouveau livre.

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { BooksService } from '../services/books.service';
import { Book } from '../models/book.model';
import { Subscription } from 'rxjs/Subscription';
import { Router } from '@angular/router';

@Component({
  selector: 'app-book-list',
  templateUrl: './book-list.component.html',
  styleUrls: ['./book-list.component.css']
})
export class BookListComponent implements OnInit, OnDestroy {
```

```
books: Book[];

booksSubscription: Subscription;


constructor(private booksService: BooksService, private router: Router) {


ngOnInit() {
  this.booksSubscription = this.booksService.booksSubject.subscribe(
    (books: Book[]) => {
      this.books = books;
    }
  );
  this.booksService.emitBooks();
}


onNewBook() {
  this.router.navigate(['/books', 'new']);
}


onDeleteBook(book: Book) {
  this.booksService.removeBook(book);
}


onViewBook(id: number) {
  this.router.navigate(['/books', 'view', id]);
}


ngOnDestroy() {
  this.booksSubscription.unsubscribe();
}
```

```
}

<div class="row">
  <div class="col-xs-12">
    <h2>Vos livres</h2>
    <div class="list-group">
      <button
        class="list-group-item"
        *ngFor="let book of books; let i = index"
        (click)="onViewBook(i)">
        <h3 class="list-group-item-heading">
          {{ book.title }}
          <button class="btn btn-default pull-right" (click)="onDeleteBook(
            <span class="glyphicon glyphicon-minus"></span>
          </button>
        </h3>
        <p class="list-group-item-text">{{ book.author }}</p>
      </button>
    </div>
    <button class="btn btn-primary" (click)="onNewBook()">Nouveau livre</bu
  </div>
</div>
```

Il n'y a rien de nouveau ici, donc passez rapidement à `SingleBookComponent` :

```
import { Component, OnInit } from '@angular/core';
import { Book } from '../../models/book.model';
import { ActivatedRoute, Router } from '@angular/router';
import { BooksService } from '../../services/books.service';
```

```
@Component({
  selector: 'app-single-book',
  templateUrl: './single-book.component.html',
  styleUrls: ['./single-book.component.css']
})
export class SingleBookComponent implements OnInit {

  book: Book;

  constructor(private route: ActivatedRoute, private booksService: BooksSer
              private router: Router) {}

  ngOnInit() {
    this.book = new Book('', '');
    const id = this.route.snapshot.params['id'];
    this.booksService.getSingleBook(+id).then(
      (book: Book) => {
        this.book = book;
      }
    );
  }

  onBack() {
    this.router.navigate(['/books']);
  }
}
```

Le component récupère le livre demandé par son id grâce à

`getSingleBook()` , et l'affiche dans le template suivant :

```
<div class="row">
  <div class="col-xs-12">
    <h1>{{ book.title }}</h1>
    <h3>{{ book.author }}</h3>
    <p>{{ book.synopsis }}</p>
    <button class="btn btn-default" (click)="onBack()">Retour</button>
  </div>
</div>
```

Il ne reste plus qu'à créer `BookFormComponent` , qui comprend un formulaire selon la méthode réactive et qui enregistre les données reçues grâce à `createNewBook()` :

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Book } from '../../models/book.model';
import { BooksService } from '../../services/books.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-book-form',
  templateUrl: './book-form.component.html',
  styleUrls: ['./book-form.component.css']
})
export class BookFormComponent implements OnInit {

  bookForm: FormGroup;
```

```
constructor(private formBuilder: FormBuilder, private booksService: Books
            private router: Router) { }


ngOnInit() {
  this.initForm();
}


initForm() {
  this.bookForm = this.formBuilder.group({
    title: ['', Validators.required],
    author: ['', Validators.required],
    synopsis: ''
  });
}
```

```
  onSaveBook() {
    const title = this.bookForm.get('title').value;
    const author = this.bookForm.get('author').value;
    const synopsis = this.bookForm.get('synopsis').value;
    const newBook = new Book(title, author);
    newBook.synopsis = synopsis;
    this.booksService.createNewBook(newBook);
    this.router.navigate(['/books']);
  }
}
```

```
<div class="row">
```

```
<div class="col-sm-8 col-sm-offset-2">
  <h2>Enregistrer un nouveau livre</h2>
  <form [formGroup]="bookForm" (ngSubmit)="onSaveBook()">
    <div class="form-group">
      <label for="title">Titre</label>
      <input type="text" id="title"
             class="form-control" formControlName="title">
    </div>
    <div class="form-group">
      <label for="author">Auteur</label>
      <input type="text" id="author"
             class="form-control" formControlName="author">
    </div>
    <div class="form-group">
      <label for="synopsis">Synopsis</label>
      <textarea id="synopsis"
                class="form-control" formControlName="synopsis">
      </textarea>
    </div>
    <button class="btn btn-success" [disabled]="bookForm.invalid"
            type="submit">Enregistrer
    </button>

  </form>
 </div>
</div>
```

Et ça y est, votre application fonctionne ! Elle enregistre et lit votre liste de livres sur votre backend Firebase, rendant ainsi son fonctionnement

totalement dynamique !

Si vous souhaitez voir le fonctionnement en temps réel de la base de données, ouvrez une deuxième fenêtre et ajoutez-y un nouveau livre : vous le verrez apparaître dans la première fenêtre presque instantanément !

Pour compléter cette application, vous allez ajouter la fonctionnalité qui permet d'enregistrer une photo de chaque livre grâce à l'API Firebase Storage.

Storage

Dans ce dernier chapitre, vous allez apprendre à utiliser l'API Firebase Storage afin de permettre à l'utilisateur d'ajouter une photo du livre, de l'afficher dans `SingleBookComponent` et de la supprimer si on supprime le livre, afin de ne pas laisser des photos inutilisées sur le serveur.

Tout d'abord, vous allez ajouter une méthode dans `BooksService` qui permet d'uploader une photo :

```
uploadFile(file: File) {
    return new Promise(
      (resolve, reject) => {
        const almostUniqueFileName = Date.now().toString();
        const upload = firebase.storage().ref()
          .child('images/' + almostUniqueFileName + file.name).put(file);
        upload.on(firebase.storage.TaskEvent.STATE_CHANGED,
          () => {
            console.log('Chargement…');
          },
          (error) => {
            console.log('Erreur de chargement ! : ' + error);
            reject();
```

```
          },
          () => {
            resolve(upload.snapshot.downloadURL);
          }
        );
      }
    );
}
```

Analysez cette méthode :

- l'action de télécharger un fichier prend du temps, donc vous créez une méthode asynchrone qui retourne une Promise ;
- la méthode prend comme argument un fichier de type File ;
- afin de créer un nom unique pour le fichier (évitant ainsi d'écraser un fichier qui porterait le même nom que celui que l'utilisateur essaye de charger), vous créez un string à partir de `Date.now()`, qui donne le nombre de millisecondes passées depuis le 1er janvier 1970 ;
- vous créez ensuite une tâche de chargement `upload` :
  - `firebase.storage().ref()` vous retourne une référence à la racine de votre bucket Firebase,
  - la méthode `child()` retourne une référence au sous-dossier `images` et à un nouveau fichier dont le nom est l'identifiant unique + le nom original du fichier (permettant de garder le format d'origine également),
- vous utilisez ensuite la méthode `on()` de la tâche `upload` pour en suivre l'état, en y passant trois fonctions :
  - la première est déclenchée à chaque fois que des données sont

envoyées vers le serveur,

- la deuxième est déclenchée si le serveur renvoie une erreur,
- la troisième est déclenchée lorsque le chargement est terminé et permet de retourner l'URL unique du fichier chargé.

Pour des applications à très grande échelle, la méthode `Date.now()` ne garantit pas à 100% un nom de fichier unique, mais pour une application de cette échelle, cette méthode suffit largement.

Maintenant que le service est prêt, vous allez ajouter les fonctionnalités nécessaires à `BookFormComponent`.

Commencez par ajouter quelques membres supplémentaires au component :

```
bookForm: FormGroup;

fileIsUploading = false;

fileUrl: string;

fileUploaded = false;
```

Ensuite, créez la méthode qui déclenchera `uploadFile()` et qui en récupérera l'URL retourné :

```
onUploadFile(file: File) {
    this.fileIsUploading = true;
    this.booksService.uploadFile(file).then(
      (url: string) => {
        this.fileUrl = url;
        this.fileIsUploading = false;
        this.fileUploaded = true;
      }
    );
```

```
}
```

Vous utiliserez `fileIsUploading` pour désactiver le bouton `submit` du template pendant le chargement du fichier afin d'éviter toute erreur — une fois l'upload terminé, le component enregistre l'URL retournée dans `fileUrl` et modifie l'état du component pour dire que le chargement est terminé.

Il faut modifier légèrement `onSaveBook()` pour prendre en compte l'URL de la photo si elle existe :

```
onSaveBook() {
    const title = this.bookForm.get('title').value;
    const author = this.bookForm.get('author').value;
    const synopsis = this.bookForm.get('synopsis').value;
    const newBook = new Book(title, author);
    newBook.synopsis = synopsis;
    if(this.fileUrl && this.fileUrl !== '') {
      newBook.photo = this.fileUrl;
    }
    this.booksService.createNewBook(newBook);
    this.router.navigate(['/books']);
}
```

Vous allez créer une méthode qui permettra de lier le `<input type="file">` (que vous créerez par la suite) à la méthode `onUploadFile()` :

```
detectFiles(event) {
    this.onUploadFile(event.target.files[0]);
}
```

L'événement est envoyé à cette méthode depuis cette nouvelle section du template :

```
<div class="form-group">
    <h4>Ajouter une photo</h4>
    <input type="file" (change)="detectFiles($event)"
           class="form-control" accept="image/*">
    <p class="text-success" *ngIf="fileUploaded">Fichier chargé !</p>
</div>
<button class="btn btn-success" [disabled]="bookForm.invalid || fileIsUploa
      type="submit">Enregistrer
</button>
```

Dès que l'utilisateur choisit un fichier, l'événement est déclenché et le fichier est uploadé. Le texte "Fichier chargé !" est affiché lorsque `fileUploaded` est `true` , et le bouton est désactivé quand le formulaire n'est pas valable ou quand `fileIsUploading` est `true` .

Il ne reste plus qu'à afficher l'image, si elle existe, dans `SingleBookComponent` :

```
<div class="row">
  <div class="col-xs-12">
    <img style="max-width:400px;" *ngIf="book.photo" [src]="book.photo">
    <h1>{{ book.title }}</h1>
    <h3>{{ book.author }}</h3>
    <p>{{ book.synopsis }}</p>
    <button class="btn btn-default" (click)="onBack()">Retour</button>
  </div>
</div>
```

Il faut également prendre en compte que si un livre est supprimé, il faut également en supprimer la photo. La nouvelle méthode `removeBook()` est la suivante :

```
removeBook(book: Book) {
    if(book.photo) {
      const storageRef = firebase.storage().refFromURL(book.photo);
      storageRef.delete().then(
        () => {
          console.log('Photo removed!');
        },
        (error) => {
          console.log('Could not remove photo! : ' + error);
        }
      );
    }
    const bookIndexToRemove = this.books.findIndex(
      (bookEl) => {
        if(bookEl === book) {
          return true;
        }
      }
    );
    this.books.splice(bookIndexToRemove, 1);
    this.saveBooks();
    this.emitBooks();
}
```

Puisqu'il faut une référence pour supprimer un fichier avec la méthode

`delete()`, vous passez l'URL du fichier à `refFromUrl()` pour en récupérer la référence.

Déployez votre application

Ça y est ! L'application est prête à être déployée. Si votre serveur de développement tourne encore, arrêtez-le et exécutez la commande suivante :

```
ng build --prod
```

Vous utilisez le CLI pour générer le package final de production de votre application dans le dossier `dist`.

Le build est un moment où plusieurs erreurs peuvent arriver, et ce ne sera pas forcément à cause de votre code. De mon côté, j'ai dû faire une mise à jour du CLI et modifier la version dans les devDependencies dans package.json. Malheureusement, on ne peut pas prévoir quelles erreurs peuvent arriver à ce moment-là, mais vous ne serez jamais les seuls à rencontrer votre problème : copiez l'erreur dans votre moteur de recherche favori et vous trouverez certainement une réponse.

Le dossier `dist` contient tous les fichiers à charger sur votre serveur de déploiement pour votre application.

Conclusion

Félicitations ! Vous avez utilisé vos nouvelles compétences Angular pour créer une application dynamique, comportant plusieurs components qui :

- affichent des données dans le template avec le data binding ;
- sont construits de manière dynamique avec les directives ;
- communiquent ensemble grâce aux services ;
- sont accessibles par un routing personnalisé ;

- emploient des Observables pour gérer des flux de données ;
- utilisent des formulaires pour exploiter des données fournies par l'utilisateur ;
- fonctionnent avec un backend Firebase pour la gestion de l'authentification, des données et des fichiers.

Vous avez maintenant les connaissances nécessaires pour créer votre propre application Angular et pour trouver les informations dont vous avez besoin pour intégrer les fonctionnalités que vous ne connaissez pas encore. N'hésitez pas à aller plus loin, à chercher de nouvelles idées et de nouvelles méthodes, et vous créerez certainement des applications fabuleuses !