Chapitre 6
Les décorateurs

1. Introduction

Un décorateur permet d'ajouter un comportement à un élément lors de l'exécution du code. Les décorateurs prennent la forme de fonctions qui peuvent être par la suite exécutées en utilisant une expression. Celle-ci commence avec le caractère @, suivi du nom du décorateur à exécuter.

Syntaxe :

```
@DecoratorName
```

Remarque

Il n'existe pas de convention de nommage actée pour les décorateurs. Toutefois, la convention de nommage la plus communément utilisée est celle dite "PascalCase". Dans ce chapitre, c'est la convention qui sera utilisée dans les exemples.

Les décorateurs sont souvent utilisés pour appliquer des aspects techniques sur les objets, comme par exemple :

– Logger l'exécution d'une méthode.
– Injecter des dépendances.
– Notifier le changement de la valeur d'une propriété.

L'autre intérêt des décorateurs est qu'ils permettent d'ajouter des métadonnées. Celles-ci permettent de préciser des informations supplémentaires sur un élément. Elles offrent aussi la possibilité aux développeurs d'écrire du code de manière plus déclarative. Cette utilisation des décorateurs est très courante et il existe de nombreux Frameworks et bibliothèques les mettant en œuvre pour différents besoins, comme par exemple :

– Définir le verbe HTTP lié à une action d'un contrôleur (Framework/Bibliothèque de type Web MVC).
– Définir la structure d'une table en base de données (Framework/Bibliothèque de type Object-relational Mapping, ou Mapping objet-relationnel en français).
– Définir les noms de propriétés lors de la conversion d'un objet en JSON (Framework/Bibliothèque de type Parser).

Remarque

La plupart des exemples cités ci-dessus seront mis en œuvre dans le code de ce chapitre et lors du TP (cf. chapitre Un premier projet avec Node.js).

L'implémentation des décorateurs dans JavaScript est une proposition de longue date qui n'est pas encore standardisée par ECMAScript. Le concept a été introduit dans la version 1.5 de TypeScript. Cependant, les décorateurs étant considérés comme expérimentaux, le langage ne permet pas de les utiliser par défaut. Il est donc nécessaire d'activer les options de compilation dédiées lors de l'initialisation du fichier `tsconfig.json`. Lors de l'utilisation de la commande `tsc --init`, il faut ajouter les arguments `--experimentalDecorators` (qui permet d'activer l'utilisation des décorateurs dans TypeScript) et `--emitDecoratorMetadata` (qui permet d'injecter dans le code, lors de la compilation, les métadonnées autour des types. cf. section Métadonnées) pour activer le support complet des décorateurs dans TypeScript.

Exemple :

```
tsc --init --experimentalDecorators --emitDecoratorMetadata
```

Les deux options de compilation portent le même nom que les arguments dans le fichier `tsconfig.json`.

© Editions ENI - All rights reserved

Exemple :

```
"experimentalDecorators": true,
"emitDecoratorMetadata": true
```

Remarque

Par défaut, lors de l'initialisation, ces deux options sont disponibles dans le fichier `tsconfig.json`, mais commentées.

Les décorateurs sont fortement liés à la programmation orientée objet et peuvent être appliqués aux :

– Classes
– Méthodes
– Propriétés
– Accesseurs
– Paramètres

Les décorateurs ont un ordre d'exécution qui s'applique de la dernière déclaration vers la première (on parle d'exécution *bottom to top*).

Remarque

Attention, un décorateur ne doit pas dépendre d'un ordre d'exécution. La dépendance d'un décorateur à l'exécution d'un autre est considérée comme une mauvaise pratique, car cela complexifie la maintenance d'un programme.

2. Décorateur de classe

Une fonction peut être utilisée en tant que décorateur de classe à partir du moment où son type correspond à celui de `ClassDecorator`. Ce type est défini dans le fichier de déclaration de base de TypeScript (`lib.d.ts`) :

```
declare type ClassDecorator = <TFunction extends Function>(
  target: Tfunction
) => TFunction | void;
```

Remarque

Le mot-clé `type` est utilisé en TypeScript pour définir des alias de type. Il sera abordé en détail dans la suite de cet ouvrage (cf. chapitre Système de types avancés). Le mot-clé `declare` est quant à lui utilisé dans les fichiers de définitions pour définir et typer un élément afin qu'il puisse être utilisé par la suite dans du code TypeScript.

Ce type définit que la fonction décoratrice accepte un paramètre dont le type est contraint, via l'utilisation d'un générique, à étendre celui de `Function`. Le paramètre `target` permet de récupérer le constructeur de la classe.

Exemple :

```
const LogClassName: ClassDecorator = target => {
  console.log(target.name);
};

// Log: Person
@LogClassName
class Person {
  constructor(
    public readonly firstName: string,
    public readonly lastName: string
  ) {}
}
```

Un décorateur étant une fonction, il est possible d'exécuter celle-ci directement avec la classe en paramètre.

Exemple :

```
const LogClassName: ClassDecorator = target => {
  console.log(target.name);
};

// Log: Person
LogClassName(
  class Person {
    constructor(
      public readonly firstName: string,
      public readonly lastName: string
    ) {}
  }
);
```

© Editions ENI - All rights reserved

Remarque

Tous les types décorateurs peuvent être exécutés de cette manière. Dans la suite de ce chapitre, seule la syntaxe avec expression sera utilisée.

Lorsqu'une classe est décorée, il faut ensuite l'importer pour que le décorateur s'exécute (cf. chapitre Les modules). Si un élément est décoré, mais qu'il n'est importé dans aucun module, il est tout de même possible d'exécuter le décorateur en effectuant un import direct du module. Ce type d'import est utilisé pour déclencher un effet de bord dans le programme sans pour autant récupérer les éléments exportés par le module.

Exemple (person.ts) :

```
import "./person";
```

3. Décorateur de méthode

Une fonction peut être utilisée en tant que décorateur de méthode à partir du moment où son type correspond à celui de `MethodDecorator`. Ce type est défini dans le fichier de déclaration de base de TypeScript (`lib.d.ts`) :

```
declare type MethodDecorator = <T>(
  target: Object,
  propertyKey: string | symbol,
  descriptor: TypedPropertyDescriptor<T>
) => TypedPropertyDescriptor<T> | void;
```

Ce type définit que le décorateur accepte trois paramètres en entrée :

– `target` : l'objet dans lequel la propriété est contenue.

– `propertyKey` : la méthode qui a été décorée.

– `descriptor` : le descripteur de propriété sous la forme d'une instance du type `TypedPropertyDescriptor<T>`.

Remarque

Les descripteurs de propriété ont déjà été abordés lors de l'explication sur les boucles (cf. chapitre Types et instructions basiques).

Le type `TypedPropertyDescriptor<T>` est défini dans le fichier de déclaration de base de TypeScript (`lib.d.ts`) :

```
interface TypedPropertyDescriptor<T> {
  enumerable?: boolean;
  configurable?: boolean;
  writable?: boolean;
  value?: T;
  get?: () => T;
  set?: (value: T) => void;
}
```

Il existe une version simplifiée de ce type qui utilise `any` à la place du type générique `T : PropertyDescriptor`. Ce type est lui aussi défini dans le fichier de déclaration de base de TypeScript (`lib.d.ts`) :

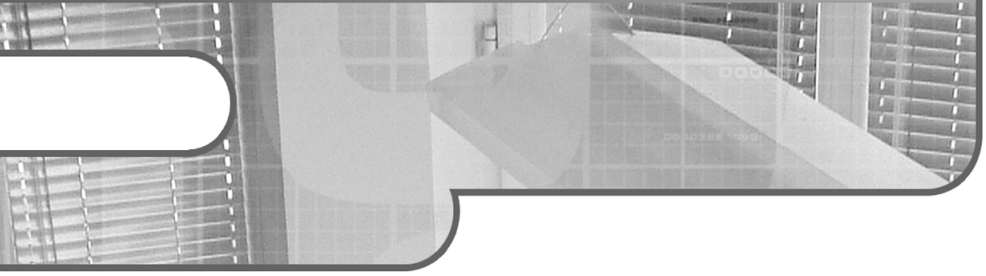
```
interface PropertyDescriptor {
    configurable?: boolean;
    enumerable?: boolean;
    value?: any;
    writable?: boolean;
    get?(): any;
    set?(v: any): void;
}
```

Remarque

Dans la section sur les boucles du chapitre Types et instructions basiques, le premier exemple montre comment définir une propriété sur un objet via l'utilisation de : `Object.defineProperty`. Le troisième paramètre attendu par la méthode `defineProperty` est typé avec l'interface `PropertyDescriptor`. Via la propriété `value` définie par cette interface, il est aussi possible de définir une méthode. C'est pour cela que les décorateurs de méthode s'appuient sur les descripteurs de propriété. De par leur nature, les décorateurs de méthode peuvent donc aussi être appliqués aux accesseurs.

Les décorateurs de méthode sont utiles pour encapsuler l'exécution d'une méthode afin de lui ajouter d'autres comportements (pouvant être appliqués avant ou après l'exécution de la méthode d'origine). Pour cela, il est nécessaire de redéfinir la méthode d'origine contenue dans la propriété `value` du paramètre `descriptor` avec une nouvelle fonction.

© Editions ENI - All rights reserved



Chapitre 5
Développer pour le mobile avec React Native

1. Présentation de React Native

À présent que nous avons vu les bases de React et comment concevoir une application grâce à Redux, vous disposez des connaissances nécessaires pour écrire des applications relativement complexes. Mais jusque-là, nous sommes restés dans le monde du Web, c'est-à-dire que les applications que nous avons écrites étaient exécutées dans un navigateur. Dans ce chapitre nous allons voir comment aller un peu plus loin en appliquant ce que nous avons vu pour créer des applications mobiles.

1.1 Un peu d'historique

Jusqu'à récemment, on pouvait dire que le développement pouvait être compartimenté en plusieurs types :

– le développement d'applications « bureau » (desktop) ;

– le développement d'applications web ;

– le développement d'applications mobiles.

En tant que développeur, au moment de concevoir une application, la première question à se poser était de savoir quelle était la plateforme cible. Est-ce que je souhaite créer une application web accessible de n'importe quel navigateur, en faisant un compromis sur la réactivité de l'interface, notamment sur mobile ? Ou bien une application mobile, même si cela implique de développer une application par OS mobile : iOS, Android, Windows… ?

En supposant que l'on souhaitait développer pour le mobile, cette question était d'autant plus importante étant donné l'essor incontestable des smartphones. En 2009, un outil a été créé promettant de faire disparaître ce problème. Phonegap, devenu depuis Apache Cordova (https://cordova.apache.org/), permettait de créer des applications mobiles universelles à l'aide de technologies web. Pour faire simple, l'idée était de créer une application web, puis l'outil intégrait cette application dans un mini-navigateur au sein d'une application native. En 2013, le framework Ionic (https://ionicframework.com/) a ajouté une surcouche à Cordova permettant de créer facilement des applications mobiles à l'aide du framework Angular.

Si Phonegap/Cordova a eu beaucoup de succès et est toujours très utilisé, force est de constater que certaines applications mobiles nécessitaient malgré tout une réactivité au niveau de l'interface qu'une application web « cachée » dans une application native n'était pas en mesure d'offrir.

En 2015, un nouvel acteur entre en scène. Dans la période de succès de React, Facebook annonce React Native, une surcouche à React permettant de développer des applications mobiles natives. Par native, j'entends que les éléments rendus par l'application (textes, boutons, etc.) sont bien des éléments graphiques spécifiques à la plateforme mobile, et non pas des éléments web dans un navigateur intégré.

Le succès a une fois de plus été au rendez-vous. Il n'était plus nécessaire de choisir entre du code JavaScript universel et une application mobile native à l'interface fluide. Il était désormais possible de réutiliser la même technologie pour développer sur le Web et sur mobile !

Notez que depuis son annonce, React Native n'a jamais eu pour ambition de permettre la création d'applications universelles pouvant être exécutées sur navigateur et sur mobile. Le but était plutôt d'unifier les technologies utilisées.

© Editions ENI - All rights reserved

Autrement dit, plutôt que « une seule application pour toutes les plateformes », React Native a préféré se positionner sur la philosophie « une seule technologie pour toutes les plateformes ».

Terminons notre rapide historique par l'entrée en scène d'un outil formidable que nous allons utiliser dans ce chapitre : Expo (https://expo.io/). Sorti en 2015 sous le nom d'Exponent, Expo facilite grandement le développement d'applications avec React Native car il gère tout le processus de génération de l'application mobile (en développement comme pour la production), pour iOS et Android. Depuis l'été 2019, il est même possible de générer une application web, grâce au projet React Native Web (https://github.com/necolas/react-native-web).

Nous ne nous occupons que du code JavaScript, et Expo nous permet de lancer l'application sur mobile ou dans le simulateur de notre choix. Il génère même les binaires de l'application qu'il n'y a plus qu'à soumettre à l'AppStore et Google Play. Autant dire que nous aurions tort de ne pas profiter de tous ces avantages pour notre apprentissage de React Native !

1.2 Outils utilisés

Pour résumer, voici les outils que nous utiliserons dans ce chapitre :

– React, bien entendu !
– React Native, qui est au mobile ce que React DOM (la bibliothèque que nous avions en dépendance dans nos applications jusqu'à maintenant) est au Web.
– Expo, pour rendre le développement plus facile en nous permettant de nous concentrer sur l'application elle-même et non sur les problématiques natives inhérentes à chaque plateforme (génération des binaires dans XCode ou Android Studio…).

Remarque

L'utilisation de React Native et Expo ne nécessite pas de compte développeur Apple ou Google pour développer et tester l'application. En revanche pour ce qui est de la distribution, les règles sont les mêmes que pour les applications mobiles traditionnelles. Pour diffuser une application iOS, vous aurez besoin d'un compte développeur Apple (100 $US par an), et il en sera de même pour une application Android si vous souhaitez voir votre application sur Google Play (25 $US).

Notez aussi que pour utiliser le simulateur iOS vous devrez travailler sur Mac, mais il vous sera tout de même possible de tester votre application à l'aide d'un vrai iPhone ou iPad, sans que celui-ci ait besoin d'être enregistré sur un compte développeur Apple.

La bonne nouvelle étant que vous pouvez tout à fait apprendre à développer avec React Native sans compte développeur, et ainsi créer vos premières applications. Si l'une de vos créations vous rend suffisamment fier au point de vouloir la diffuser au monde, peut-être qu'investir dans un compte développeur semblera plus facile.

C'est parti, attaquons sans plus tarder la mise en place de notre première application mobile.

2. Une première application

2.1 Génération de l'application et premier lancement

Pour initialiser une application, le seul élément à installer est `expo-cli`, paquet NPM à installer de manière globale et contenant les outils en ligne de commande d'Expo :

```
$ npm i -g expo-cli
```

Une fois le module installé, rendez-vous dans le répertoire où vous souhaitez créer votre application afin d'initialiser le projet :

```
$ expo init ma-premiere-app-mobile
```

© Editions ENI - All rights reserved

Au jour où ces lignes sont écrites (cela change fréquemment), il est demandé si vous souhaitez utiliser le template « Managed » ou « Bare », si vous souhaitez utiliser TypeScript ou non, et si l'installation doit utiliser Yarn ou NPM. Dans tous les cas, prenez le choix par défaut.

Remarque

Depuis la sortie d'Expo 33 au printemps 2019, il est possible d'utiliser les fonctions d'Expo (notamment pour accéder à certaines fonctions natives de l'OS) même dans une application React Native ne reposant pas sur Expo. Expo distingue donc depuis les applications « managed », reposant sur Expo, des applications « bare » qui incluent Expo en tant que dépendance. Pour notre cas nous utiliserons le mode « managed ».

La commande crée le répertoire ma-premiere-app-mobile et l'initialise avec les fichiers nécessaires pour lancer une application minimaliste avec Expo. Les fichiers principaux sont les suivants :

– package.json : il est initialisé avec les dépendances nécessaires et les commandes qui nous faciliteront le développement.
– app.json : contient des métadonnées de l'application pour Expo, notamment la version d'Expo à utiliser. À terme, il contiendra aussi des données nécessaires pour générer le binaire de l'application.
– App.js : le fichier principal de l'application.

Allons jeter un œil au fichier App.js afin de voir de quoi il retourne. Notez que depuis que j'ai écrit ce chapitre il se peut qu'Expo ait été mis à jour au point de changer légèrement le contenu des fichiers générés. Je n'ai cependant aucun mal à croire que les principes décrits ici resteront les mêmes.

Tout d'abord sont importées les bibliothèques React, et sans surprise quelques composants de React Native :

```
import React from 'react'
import { StyleSheet, Text, View } from 'react-native'
```

Rien de vraiment nouveau ici. Le composant principal de l'application, `App` se présente comme une classe héritant de `Component` comme nous en avons déjà vues.

Cette classe ne contient qu'une méthode `render`, renvoyant trois éléments (composant `Text`), intégrés dans une `View` :

```
export default class App extends React.Component {
 render() {
   return (
     <View style={styles.container}>
       <Text>Open up App.js to start working on your app!</Text>
       <Text>Changes you make will automatically reload.</Text>
       <Text>Shake your phone to open the developer menu.</Text>
     </View>
   )
 }
}
```

Cette `View` définit un attribut `style`, qui fait référence à la déclaration de feuille de style déclarée en dessous à l'aide de `StyleSheet.create` :

```
const styles = StyleSheet.create({
 container: {
   flex: 1,
   backgroundColor: '#fff',
   alignItems: 'center',
   justifyContent: 'center'
 }
})
```

Nous verrons un peu plus loin ce que représentent les composants `View` et `Text` ainsi que les feuilles de style. Pour le moment, contentons-nous de lancer notre application. Mais d'abord, vous aurez besoin d'un simulateur iOS ou Android en cours d'exécution :

– Pour iOS vous devrez (sur Mac) d'abord vous assurer que Xcode est installé (gratuit sur l'AppStore), ainsi que les outils en ligne de commande associés (il suffit normalement de lancer Xcode pour ce soit suggéré et installé). Le simulateur sera ensuite disponible par le menu **Xcode - Open Developer Tool - Simulator**.

– Pour Android, il faudra installer Android Studio puis créer un projet (le projet de base proposé convient très bien), et enfin créer un simulateur grâce au menu **Tools - Android - AVD Manager** (il se peut que ce menu soit grisé pendant que le projet est totalement initialisé, soit quelques minutes).

© Editions ENI - All rights reserved