Partie 7
Frameworks JavaScript

Chapitre 7-1
Positionnement des frameworks JavaScript

1. Présentation générale des frameworks JavaScript

De très nombreux frameworks JavaScript existent, avec des positionnements fonctionnels différents.

Il ne peut être question, dans le cadre de ce livre réservé à des débutants en JavaScript, d'en faire une revue exhaustive.

Ils ont tous les points communs suivants : masquer la complexité du langage JavaScript, apporter de la robustesse dans les développements et aussi permettre, pour certains d'entre eux, d'interagir avec des bases de données.

1.1 Frameworks « front-end »

Les plus populaires des frameworks dits « front-end », c'est-à-dire gérant le côté interface utilisateur des applications web ou mobiles (téléphones mobiles, tablettes…) sont :

– Angular, framework développé par Google (la première version était connue sous l'appellation AngularJS)
– React JS (ou React), framework développé par Facebook
– Vue.js
– Svelte

1.2 Frameworks « back-end »

Pour les interactions avec les systèmes de gestion de bases de données, des frameworks dits « back-end » existent. Ils sont souvent eux-mêmes basés sur Node.js, qui est un environnement d'exécution multiplateforme Open Source exécutant du code JavaScript en dehors d'un navigateur (dans un runtime).

Node.js permet de concevoir des services d'accès à des Bases De Données et à des ressources disponibles sur Internet. Il fonctionne parfaitement sur Windows, Linux ou encore macOS.

Nous verrons par exemple que l'accès aux données pour le framework Svelte peut être assuré par les frameworks Express ou Sapper, tous deux basés sur Node.js.

1.3 Solutions de développement « hybride »

Pour terminer ce rapide panorama concernant les frameworks JavaScript, n'oublions pas les solutions dédiées aux développements pour périphériques mobiles (smartphones et tablettes). Dans un précédent livre, publié en 2020 aux Éditions ENI (Java et Ionic - Développement mobile pour Android : natif vs hybride), le framework Ionic a été présenté. Ce framework est lui-même basé sur d'autres frameworks, dont le très réputé Angular (soutenu par Google) et Apache Cordova.

Un chapitre du présent livre sera aussi consacré au Framework React Native (assez proche de React qui, lui, est réservé aux applications web). React Native permet très facilement à des développeurs ayant déjà une réelle expérience en React de concevoir des applications pour mobiles. Ce framework extrêmement utilisé et supporté par Facebook est une alternative plus que crédible à Ionic. Les applications React Native sont facilement déployables sur les mobiles fonctionnant sous systèmes d'exploitation Android et iOS (iPhone, iPad).

2. Les frameworks Node.js, Svelte, React et React Native

Comme indiqué précédemment, un court chapitre (Installation de Node.js) sera consacré à l'installation du framework Node.js, socle sur lequel fonctionnent les frameworks Svelte, React et React Native.

Le framework Svelte, relativement récent, est un challenger crédible pour React (React JS) et Vue.js. Svelte, présenté dans le chapitre Framework Svelte, possède de nombreux atouts techniques. Il bénéficie par contre pour l'instant d'une communauté de développeurs plus restreinte que celles des deux acteurs principaux (React et Vue.js).

Dans le cadre de ce livre, un choix éditorial a été fait de ne pas évoquer Vue.js. Vous trouvez, toujours aux Éditions ENI, des ouvrages dédiés exclusivement à Vue.js, notamment le livre Vue.js - Développez des applications web modernes en JavaScript de Yoann GAUCHARD.

Le chapitre Framework React sera celui dédié à React. Comme dans le chapitre consacré à Svelte, après une rapide présentation des concepts de base, de nombreux exemples seront proposés et largement commentés.

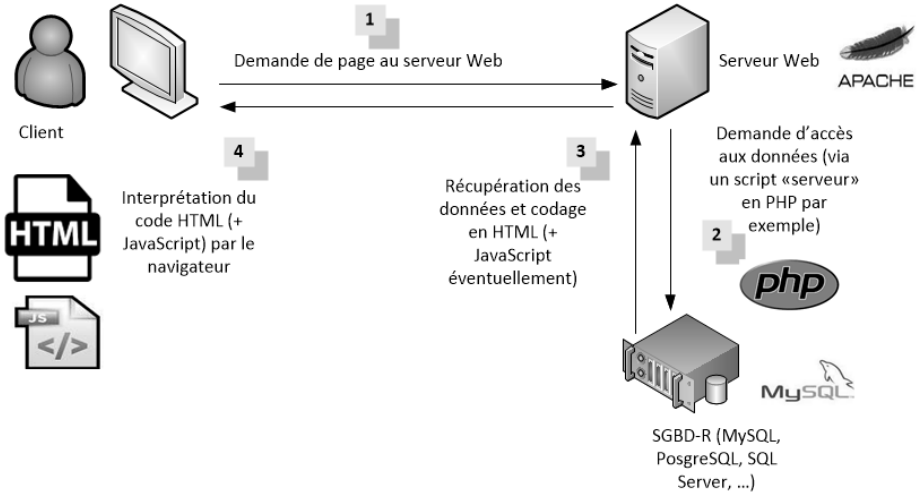
Le livre se terminera au chapitre Framework React Native avec un exposé consacré à React Native, la version du framework React permettant le développement d'applications pour mobiles.

Chapitre 7-2
Installation de Node.js

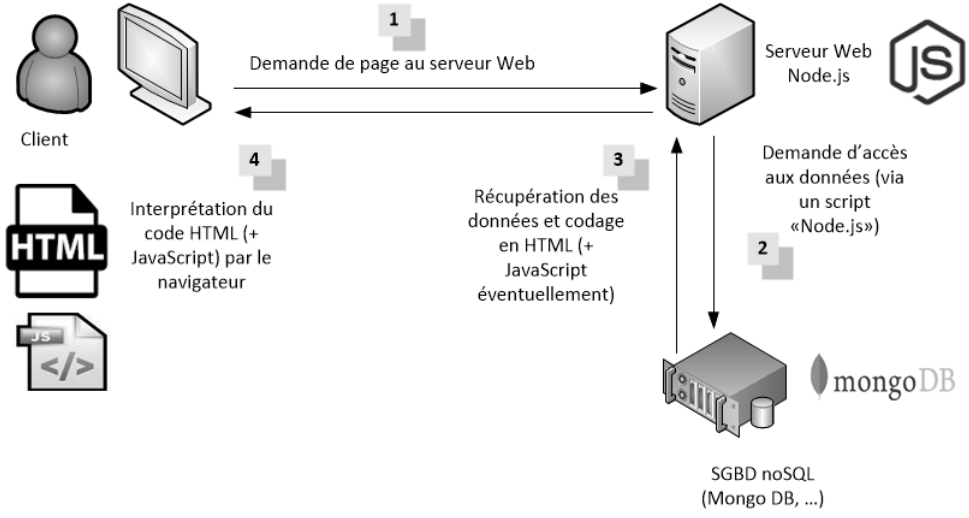
1. Présentation du framework Node.js

JavaScript a longtemps été cantonné à une utilisation côté client. On utilisait JavaScript seulement pour ajouter de l'interactivité dans les pages web (animations, contrôles de saisie...).

Pour les accès aux Bases De Données distantes, comme MySQL, les applications web intégraient par ailleurs des scripts orientés serveur, là aussi souvent développés en langage PHP.



Avec Node.js, il est bien sûr toujours possible d'utiliser JavaScript côté client pour manipuler les pages HTML. En plus, Node.js propose un environnement côté serveur qui permet aussi d'utiliser le langage JavaScript pour générer des pages web. En clair, il vient en remplacement de langages serveur comme PHP, Java EE, etc.



Chapitre 3
La programmation orientée objet

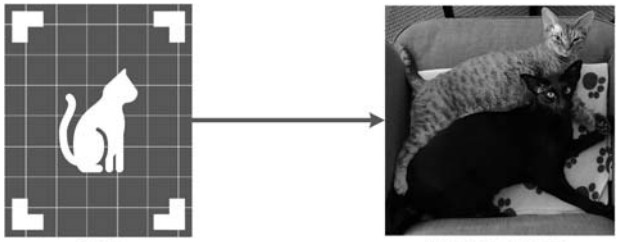
1. Introduction

La programmation orientée objet (aussi connu sous l'acronyme : POO) est un des paradigmes de développement les plus utilisés pour réaliser des applications. Dans les débuts de la programmation, les logiciels étaient réalisés à partir de séquences d'instructions s'exécutant les unes après les autres, c'est ce que l'on appelle la programmation impérative. Au fil du temps, les programmes informatiques sont devenus plus complexes, ce qui augmenta considérablement la difficulté à les maintenir et les faire évoluer. Dans les années 70, *Alan Kay* posa les bases de la programmation orientée objet. Ce paradigme est devenu rapidement populaire et est encore utilisé aujourd'hui comme fondation dans bon nombre de logiciels que nous utilisons au quotidien. Ce style de programmation permet d'organiser un logiciel en utilisant des sous-ensembles plus petits, communément appelés objets. Chacun de ces objets contient des données et de la logique qui lui sont propres. Les objets sont donc des parties autonomes d'un programme et ont des interactions les uns avec les autres pour le faire fonctionner.

TypeScript est un langage dit *multiparadigme*. Il permet donc de développer des applications en utilisant la programmation impérative, orientée objet ou encore fonctionnelle (cf. chapitre TypeScript et la programmation fonctionnelle). Les premières versions de TypeScript ont apporté de nombreuses capacités syntaxiques autour de la programmation orientée objet. C'est d'ailleurs l'une des raisons pour lesquelles le langage est devenu populaire auprès de certaines communautés de développeurs, notamment C# et Java. Cependant, il est important de garder en tête que même si TypeScript est syntaxiquement proche de Java et C#, il n'en est pas un équivalent. En réalité, les capacités objets de TypeScript suivent la norme ECMAScript, mais le langage dispose aussi de particularités qui lui sont propres. Ces différentes capacités seront décrites en détail dans ce chapitre.

2. Les classes

La première notion importante à apprendre lorsque l'on commence l'apprentissage de la programmation orientée objet est le concept de classe. Chaque objet doit être créé par une classe. Celle-ci peut être comparée à une notice de fabrication qui contient l'ensemble des informations nécessaires à la création d'un objet. Une fois définie, la classe est utilisée dans le programme pour créer un objet, on parle alors d'instance de classe. Il est possible de créer autant d'objets que l'on souhaite pour faire fonctionner un programme.



Les objets vont ensuite interagir les uns avec les autres pour faire fonctionner le programme.

Pour déclarer une classe en TypeScript, il faut utiliser le mot-clé `class`, lui donner un nom et ouvrir les accolades pour définir ses caractéristiques.

Remarque

Le concept de classe n'est pas nouveau en JavaScript. Le mot-clé `class` est un sucre syntaxique basé sur la notion de prototype dans le langage (cf. chapitre Types et instructions basiques). En ECMAScript 5, il est possible de définir une classe via une fonction constructrice. Celle-ci a la particularité de définir la structure des objets et de les créer. Depuis ECMAScript 2015, les classes sont privilégiées par rapport aux fonctions constructrices, car ces dernières ont une syntaxe peu intuitive.

Syntaxe :

```
class ClassName {
  // ...
}
```

Exemple :

```
class Employee {
  // ...
}
```

Remarque

Le nommage des classes en TypeScript suit la convention dite "PascalCase". Ce type de nommage précise que chaque mot composant le nom de la classe doit avoir sa première lettre en majuscule (exemple : `PaySplit`). Les classes et les interfaces sont les seuls éléments en TypeScript utilisant cette convention, cela permet de facilement les repérer lorsqu'on lit du code. Pour tous les autres éléments, c'est la convention de nommage "camelCase" qui est utilisée. Elle est similaire à la convention "PascalCase" à ceci près que la première lettre est en minuscule (exemple : `firstName`).

Une fois déclarée, la classe peut être utilisée pour créer de nouvelles instances. Chacune de ces instances sera alors considérée par TypeScript comme étant du type défini par la classe. Pour instancier une classe, il faut utiliser le mot-clé `new` et assigner l'instance dans une variable.

Syntaxe :

```
let/const variable: ClassName = new ClassName();
```

Exemple :

```
const employee = new Employee();
```

Plusieurs instances d'une même classe peuvent être créées si besoin. Chaque instance est alors complètement indépendante des autres.

Exemple :

```
const employee1 = new Employee();
const employee2 = new Employee();

// Log: false
console.log(employee1 === employee2);
```

Avec TypeScript, les classes sont dites "*first class citizen*" (Objet de premier ordre), il est donc possible de les assigner dans des variables.

Syntaxe :

```
let/const ClassName = class {
 // ...
};
```

Lorsqu'une classe est contenue dans une variable, il faut alors utiliser le mot-clé new sur celle-ci afin de créer une instance de la classe.

Exemple :

```
const Employee = class {
  // ...
};

const employee = new Employee();
```

Remarque

Cette syntaxe est plus particulière et parfois utilisée dans certaines implémentations plus complexes. Dans la suite de ce chapitre, l'ensemble des exemples de code ne s'appuieront pas sur cette syntaxe.

3. Les propriétés

Étant donné qu'un objet doit fonctionner de manière autonome, il est nécessaire qu'il conserve un état au cours de son utilisation. Cet état est contenu dans l'objet sous la forme de donnée. L'état d'un programme en programmation orientée objet correspondra donc à l'ensemble des états de chaque instance utilisée pour le faire fonctionner.

Pour assigner une donnée dans un objet en TypeScript, il faut utiliser une propriété. Celles-ci sont définies au niveau de la classe et doivent être typées. Pour déclarer une propriété, il suffit de l'ajouter entre les accolades de la classe en lui donnant un nom et en précisant ensuite son type.

Syntaxe :

```
class ClassName {
  propertyName: type;
}
```

Exemple :

```
class Employee {
  firstName: string;
  lastName: string;
}
```

Une fois l'instance d'une classe créée, il est alors possible d'assigner des valeurs aux différentes propriétés, mais aussi de les récupérer. Les propriétés sont disponibles en utilisant un "**.**" après le nom de la variable contenant l'objet.

Exemple :

```
let employee = new Employee();
employee.firstName = "Evelyn";
employee.lastName = "Miller";

// Log: Evelyn
console.log(employee.firstName);

// Log: Miller
console.log(employee.lastName);

employee = new Employee();
```

```
// Log: undefined
console.log(employee.firstName);

// Log: undefined
console.log(employee.lastName);
```

Remarque

Par défaut, les propriétés seront initialisées avec la valeur `undefined`.

Ce premier exemple pose un problème de compilation, lorsque l'on démarre un projet en TypeScript avec la configuration de base du compilateur (obtenue en lançant la commande `tsc --init`). Par défaut, le compilateur initialise le projet avec l'option `--strict` activée dans le fichier `tsconfig.json`. Elle active des sous-options dont l'option `--strictPropertyInitialization`. Cette dernière déclenche une erreur lorsque les valeurs des propriétés ne sont pas initialisées à l'instanciation d'une classe. Dans le précédent exemple, les propriétés ne sont pas initialisées lors de l'utilisation du mot-clé `new`, elles contiennent donc la valeur `undefined`. Cette situation peut être due à une erreur d'inattention, c'est pourquoi TypeScript ne l'autorise pas par défaut.

Il est possible de forcer le compilateur à ne pas remonter cette erreur en utilisant le caractère "!" à la fin du nom d'une propriété (aussi appelé *definite assignment assertion operator*).

Exemple :

```
class Employee {
  firstName!: string;
  lastName!: string;
}

const employee = new Employee();

// Log: undefined
console.log(employee.firstName);

// Log: undefined
console.log(employee.lastName);
```

Remarque

Attention à ne pas forcer le compilateur de manière systématique. La plupart des erreurs remontées par le mode strict de TypeScript sont importantes. Forcer l'initialisation des propriétés permet, par exemple, de ne pas obtenir d'erreur si elles sont utilisées sans valeur. C'est une erreur courante dans le développement web, surtout lorsque la propriété est censée contenir une fonction (ce qui déclenche l'erreur : undefined is not a function).

Assigner une valeur aux propriétés permet aussi de contourner l'erreur de compilation liée au mode strict.

Exemple :

```
class Employee {
  firstName: string = "Evelyn";
  lastName: string = "Miller";
}

const employee = new Employee();

// Log: Evelyn
console.log(employee.firstName);

// Log: Miller
console.log(employee.lastName);
```

Remarque

Dans la suite de ce chapitre, l'option `--strict` sera toujours considérée comme active dans les exemples de code.

Une fois définies, les propriétés peuvent être retrouvées dans l'environnement de développement via l'autocomplétion.

Exemple (Visual Studio Code) :