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.

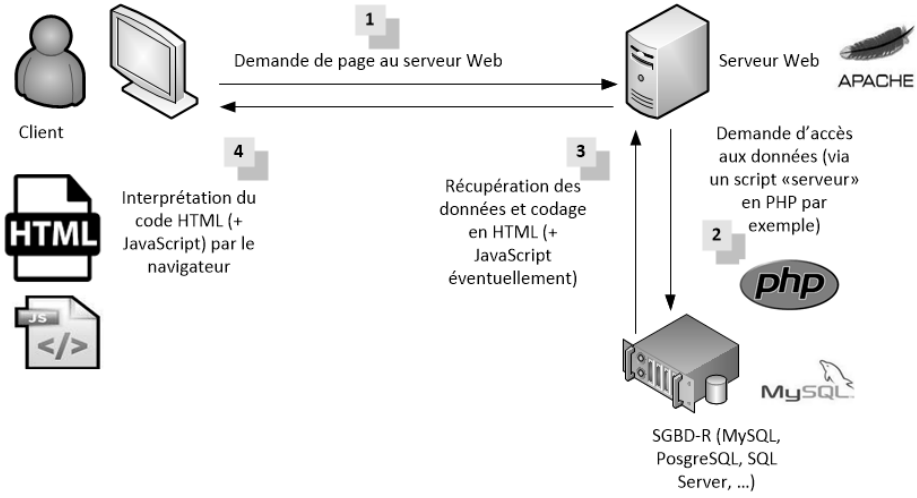
Le livre se terminera au chapitre Framework React Native avec un exposé consacré à React Native, la version du framework React permettant le développement d'applications pour mobiles.

Chapitre 7-2
Installation de Node.js

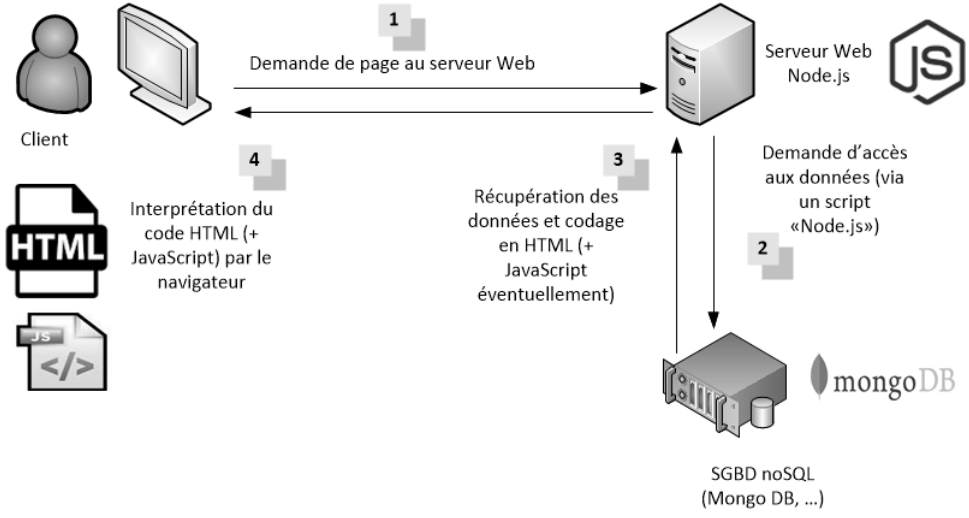
1. Présentation du framework Node.js

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
Adopter les bonnes pratiques

1. Espace de noms

1.1 Principe

Lorsque nous développons, nous éviterons toujours d'exposer un trop grand nombre de fonctions ou de variables dans l'espace de noms global afin d'éviter des conflits de nom. Cela signifie de ne donner l'accès qu'à ce qui est utile et masquer tout le reste. C'est un peu le principe d'une boîte noire avec des entrées et sorties bien définies mais la mécanique interne reste cachée.

C'est d'autant plus important si le code risque d'être utilisé dans d'autres contextes. Le problème vient que nous avons l'habitude d'ajouter des fonctions dans nos fichiers sans prendre en compte la réutilisation. Notre projet devenant plus conséquent, il devient de plus en plus difficile à gérer. Associer deux codes peut alors provoquer des conflits qui ne sont pas forcément visibles au premier coup d'œil et le résultat devient incertain.

Pour limiter les collisions de nom, nous ajouterons un contexte supplémentaire (une sorte de super contexte) qui garantira que nos fonctions et variables ne pourront être altérées/utilisées accidentellement. Cet espace de noms est une sorte de conteneur de noms. Un nom n'a alors de sens que par rapport à son espace de noms. Invoquer une fonction qui n'est pas dans l'espace de noms attendu devient donc impossible.

Chaque code ayant son espace de noms, il n'y a plus de risque de collisions associé à un code appartenant à un autre espace de noms.

Un espace de noms est une possibilité associée à de nombreux langages. Dans le langage Java, par exemple, le mot-clé `package` sert à effectuer la déclaration, de même en C# avec `namespace`. En JavaScript, nous n'avons malheureusement pas de mot-clé pour cet usage mais nous avons d'autres astuces tout aussi puissantes pour l'obtenir.

Ce principe est important pour la qualité de vos programmes. Une fois que l'aurez compris, il deviendra naturel et votre développement s'améliorera d'autant plus.

Plus votre programme est important et plus vous avez besoin d'espaces de noms. C'est la dimension qui en impose l'usage. Prendre en compte l'espace de noms sera donc le gage d'un code de meilleure qualité, qui pourra évoluer plus simplement et avec plus de sécurité.

1.2 Fonction

1.2.1 Fonction interne

Toutes les déclarations faites dans une fonction deviennent locales à cette fonction. En effet, pour rappel, lorsqu'une variable est déclarée dans une fonction via le mot-clé `var`, son contexte d'exécution est lié à celui de la fonction et en dehors de cette dernière, elle n'existe plus. On peut donc percevoir que la fonction est un puissant moyen pour limiter la portée des déclarations et donc semble un candidat idéal pour l'espace de noms.

Exemple

```
function aireRectangle( longueur, largeur ) {
  var aire = longueur * largeur;
  alert( aire + " cm2" );
}
aireRectangle( 10, 5 );
alert( aire );
```

Dans cet exemple, la variable `aire` est déclarée à l'intérieur de la fonction `aireRectangle`, elle n'existe donc pas en dehors de ce périmètre, c'est pourquoi la dernière instruction `alert(aire)` échoue. Notre fonction `aireRectangle` agit comme un espace de noms.

Cette particularité n'est pas limitée qu'aux déclarations de variables mais est également vraie pour les déclarations de fonctions.

Exemple

```
function aire( type, longueur, largeur ) {

  var aire = 0;

  function aireRectangle( longueur, largeur ) {
    var aire = ( longueur * largeur );
    return aire;
  }

  function aireCarre( cote ) {
    return aireRectangle( cote, cote );
  }

  if ( type == "rectangle" ) {
    aire = aireRectangle( longueur, largeur );
  } else
  if ( type == "carre" ) {
    aire = aireCarre( longueur );
  }

  return aire;

}

alert( aire( "rectangle", 10, 20 ) );
alert( aire( "carre", 10 ) );
alert( aireCarre( 20 ) );        // Erreur !
```

Les fonctions `aireRectangle` et `aireCarre` n'existent pas en dehors de la fonction `aire`, preuve en est que notre dernière instruction consistant à utiliser la fonction `aireCarre` a provoqué une erreur d'exécution.

En combinant ces fonctions internes avec des variables locales, nous obtenons en réalité un espace de noms pour `aireRectangle` et `aireCarre` qui n'appartiennent qu'à la fonction `aire`. Personne ne peut donc altérer ou manipuler ces fonctions en dehors de votre fonction. Il s'agit donc d'une technique simple pour éliminer les accès indésirables.

1.2.2 Fonction anonyme

Si nous voulons protéger notre exécution, il est également possible de passer par une fonction anonyme comme nous l'avons vu succinctement dans le premier chapitre. Cette fonction sans nom garantira un contexte indépendant.

Exemple

```
(function () { // Notre fonction anonyme

  function aireRectangle( longueur, largeur ) {
    var aire = ( longueur * largeur );
    return aire;
  }

  function aireCarre( cote ) {
    return aireRectangle( cote, cote );
  }

  alert( aireRectangle( 10, 20 ) );
  alert( aireCarre( 20 ) );
} )();

alert( aireRectangle( 10, 20 ) ); // Erreur
```

La fonction anonyme est exécutée dès qu'elle est déclarée puisque, n'ayant pas de nom, elle ne peut être invoquée ultérieurement. Son rôle n'est pas de fournir un service réutilisable mais de protéger le contenu de tout accès ultérieur.

La fonction anonyme sert d'espace de noms à notre code. Toutes les déclarations internes n'existeront plus après son exécution, c'est pourquoi l'invocation ultérieure de la méthode `airRectangle` échoue.

L'exécution est saine car nous ne pouvons pas créer de collision sur nos noms de fonctions. Si nous associons notre code à un autre code qui lui-même a défini la fonction `aireRectangle`, notre code continuera de fonctionner normalement.

1.2.3 Fonction anonyme avec paramètres

Le cas précédent est simple mais insuffisant dans la pratique car nous aimerions malgré tout disposer d'une fonction `aire` accessible partout par exemple mais sans rendre visibles les fonctions annexes `aireCarre` et `aireRectangle`.

Pour cela, nous pouvons créer un objet vide qui va nous servir de conteneur pour les fonctions accessibles. Il jouera alors le rôle d'espace de noms pour ces dernières.

Exemple

```
var monAire = {};

(function ( ns ) {
  function aireRectangle( longueur, largeur ) {
    var aire = ( longueur * largeur );
    return aire;
  }
  function aireCarre( cote ) {
    return aireRectangle( cote, cote );
  }
  function aire() {
    if ( arguments.length == 2 ) {
      return aireRectangle( arguments[ 0 ], arguments[ 1 ] );
    } else
    if ( arguments.length == 1 ) {
      return aireCarre( arguments[ 0 ] );
    } else
    return "calcul d'aire impossible";
  }

  ns.aire = aire; // Contenu accessible
} )( monAire );

alert( monAire.aire( 10,20 ) );
```

Dans cet exemple, nous avons créé une fonction `aire` interne à une fonction anonyme. Celle-ci utilise les fonctions `aireCarre` et `aireRectangle`. Pour que la fonction `aire` puisse être utilisable partout, nous avons fait en sorte que la fonction anonyme exécutée puisse avoir un paramètre objet stockant la référence à la méthode `aire`.

C'est cet objet qui va faire office d'espace de noms pour la méthode `aire`. Les autres méthodes `aireCarre` et `aireRectangle` restent définitivement cachées de l'utilisateur.

Nous aurions pu également faire un test de la propriété `aire` de l'objet en fin de fonction anonyme pour ne pas écraser une précédente déclaration, par exemple avec :

```
if ( !ns.aire )
   ns.aire = aire;
```

Ainsi, nous retrouvons par ce système quelque chose de similaire aux méthodes privées et publiques en programmation objet. La fonction anonyme joue alors le rôle de classe.

Le fait que la méthode `aire` puisse continuer à fonctionner en dehors de la fonction anonyme est également un principe de fermeture. La fermeture est la faculté pour une fonction qui est bien accessible d'utiliser un contexte qui ne plus l'être (il est donc privé).

La variable `monAire` est dans l'espace de noms global. Il est cependant possible de réduire légèrement les conflits de nom en utilisant l'objet prédéfini `window`. Cet objet a la particularité d'avoir ses propriétés visibles dans l'espace de noms global. En remplaçant `monAire` par `window.monAire` on réduit la visibilité de notre variable `monAire` qui devient alors une propriété de `window`. À l'usage il n'y a pas vraiment de différence.

D'une manière générale, l'association de nouvelles facultés à `window` est plutôt dépendante du contexte web, on cherchera à apporter des facultés supplémentaires à notre fenêtre.

Pour remarque, les collisions de noms sont limitées par les espaces de noms mais à condition que ceux-ci soient aussi distingués. Sinon on reporte le problème à un niveau supérieur. Généralement, on emploiera une convention pour le choix d'un nom d'espace de noms de type URI (*Uniform Resource Identifier*) pour être sûr d'en être les seuls détenteurs. Par exemple, si on travaille dans une société ABC, on pourrait faire en sorte que l'objet `monAir` s'appelle `abc`, voire `com_abc`...

1.3 Fermeture

Nous avons déjà utilisé une fermeture pour notre fonction `aire`, mais pour être plus explicite, nous allons ici l'utiliser d'une autre façon, en ne gardant au final qu'une fonction d'accès.

```
var monAire = ( function () {
  function aireRectangle( longueur, largeur ) {
    var aire = ( longueur * largeur );
    return aire;
  }
  function aireCarre( cote ) {
    return aireRectangle( cote, cote );
  }
  function aire() {
    if ( arguments.length == 2 ) {
      return aireRectangle( arguments[ 0 ], arguments[ 1 ] );
    } else
    if ( arguments.length == 1 ) {
      return aireCarre( arguments[ 0 ] );
    } else
    return "calcul d'aire impossible";
  }
  return function() {
    return aire.apply(this,arguments);
  };
} )();

alert( monAire( 20 ) );
alert( monAire( 10,20 ) );
```

Dans cet exemple, notre fonction anonyme retourne une nouvelle fonction. C'est elle qui conserve la logique de fonctionnement que nous souhaitons exposer. Ici, nous faisons en sorte que la fonction `aire` invisible puisse être utilisée avec n'importe quel nombre d'arguments, c'est pour cette raison que nous employons `apply`.

La fermeture est bien respectée car la fonction retournée continue à accéder aux fonctions `aire`, `aireCarre` et `aireRectangle` alors que ces dernières sont inaccessibles dans le contexte global.