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`...

Chapitre 4
La plateforme Node.js

1. Présentation de Node.js

Node.js est un environnement permettant d'exécuter du code JavaScript hors d'un navigateur. À l'heure de la rédaction de cet ouvrage, il repose sur le moteur JavaScript V8 développé par Google pour ses navigateurs Chrome et Chromium.

Son architecture est modulaire et événementielle. Il est fortement orienté réseau en possédant pour les principaux systèmes d'exploitation (Unix/Linux, Windows, Mac OS) de nombreux modules réseau (dont voici les principaux par ordre alphabétique : DNS, HTTP, TCP, TLS/SSL, UDP). De ce fait, il remplace avantageusement, dans le cadre qui nous intéresse ici (c'est-à-dire la création et la gestion d'applications web), un serveur web tel qu'Apache.

Créé par Ryan Lienhart Dahl en 2009, cet environnement est devenu rapidement très populaire pour ses deux qualités principales :

– Sa légèreté (en corollaire de sa modularité).
– Son efficacité induite par son architecture monothread (en corollaire de la gestion événementielle que propose nativement l'environnement JavaScript).

Intégrer Node.js dans le développement d'applications web participe donc à la logique actuelle de rendre les opérations d'accès aux données les moins bloquantes possible (pour dépasser la problématique dite du « bound I/O » selon laquelle, avant toute autre cause, la latence globale d'une application est due au temps de latence des accès aux données).

Node.js permet donc, pour les applications web, de créer des serveurs extrêmement réactifs.

Dans ce qui suit, vous allez :

– Installer et tester Node.js sous Linux, Windows ou macOS.
– Créer un serveur HTTP renvoyant une chaîne de caractères.
– Mettre en œuvre un module.
– Créer un serveur HTTP utilisant le module express invoqué sur une route REST et renvoyant des données formatées en JSON, d'abord en totalité, puis filtrées sur une propriété.

Dans ce chapitre, nous n'introduirons que quelques modules (et fonctions) de Node.js.

La documentation complète des modules est disponible à cette adresse : https://nodejs.org/api/

2. Installation et test de Node.js

2.1 Création du fichier de test

Pour tester Node.js, vous allez dans un premier temps créer un code JavaScript qui va être le plus simple possible, et le faire exécuter par Node.js.

Créez donc le fichier testDeNode.js qui ne comprend qu'une ligne de code :

```
console.log("Test de Node");
```

2.2 Installation et test de Node.js sous Ubuntu

- Pour installer Node.js sous Ubuntu, le plus simple est d'utiliser la commande `curl` et le gestionnaire de paquets en ligne de commande (`apt-get`) :

```
curl -sL https://deb.nodesource.com/setup_11.x | sudo -E bash -
sudo apt-get install nodejs
```

Il est à noter qu'au jour de l'écriture de cet ouvrage, la version courante de Node.js est la 11. Pour une version ultérieure, il suffit de changer son numéro de version dans la commande d'installation.

- Un lien symbolique nommé `node` peut être créé pour lancer plus naturellement vos serveurs :

```
sudo ln -s /usr/bin/nodejs /usr/bin/node
```

- Ouvrez un terminal (shell) et exécutez le fichier de test :

```
node testDeNode.js
```

La chaîne de caractères « Test de Node » s'affiche !

Une procédure complète est en ligne sur http://doc.ubuntu-fr.org/nodejs.

2.3 Installation et test de Node.js sous Windows

L'installation de Node.js et son test sous Windows vont se dérouler en quatre étapes :

- Téléchargez l'installateur Windows Installer en vous rendant sur le site officiel de Node.js : https://nodejs.org/en/download
- Exécutez l'installateur (le fichier .msi précédemment téléchargé) en acceptant les conditions d'utilisation et le paramétrage par défaut.
- Redémarrez votre ordinateur.
- Ouvrez l'invite de commandes et exécutez le fichier de test :

```
node testDeNode.js
```

La chaîne de caractères « Test de Node » s'affiche !

2.4 Installation et test de Node.js sous Mac OS

L'installation de Node.js et son test sous Mac Os vont se faire en trois étapes :

- Téléchargez le package d'installation pour Mac Os (Macintosh Installer) en vous rendant sur le site officiel de Node.js : https://nodejs.org/en/download/
- Ouvrez un terminal et installez le package :

```
pkg install nomPackage.pkg
```

- Exécutez le fichier de test :

```
node testDeNode.js
```

La chaîne de caractères « Test de Node » s'affiche !

3. La modularité de Node.js

3.1 Les modules et les packages

Une des principales forces de Node.js est d'être modulaire (et notamment de proposer de nombreux modules réseau). Si certains de ces modules sont installés directement en même temps que Node.js, la plupart doivent être installés à la demande.

Lors de la création d'une application qui exige l'installation de modules, deux méthodes sont possibles pour effectuer celle-ci :

– Directement avec le gestionnaire de modules npm (et son option `install`).
– Indirectement (mais toujours avec npm) via la spécification des dépendances de l'application (c'est-à-dire des modules nécessaires à celle-ci) dans un fichier nommé package.json.

Un module est utilisé dans une application avec la fonction `require()` :

```
var moduleDansVotreApplication = require('<nomDuModule>');
```

Votre application Node.js peut être elle-même réutilisée comme module sous certaines conditions qui seront présentées ultérieurement.

Attardons-nous sur un point un peu subtil : la distinction entre modules et packages.

Les modules sont les briques conceptuelles d'une application Node.js. Un module peut être organisé en plusieurs codes JavaScript et dépendre d'autres modules. Ainsi, toutes les ressources nécessaires à un module (les codes, le fichier package.json spécifiant ses dépendances...) sont regroupées dans un package qui, de fait, est un dossier.

Donc, si les deux termes sont quasiment interchangeables, le terme « module » renvoie plus à la fonctionnalité globale, et « package » au dossier et à l'organisation des fichiers de code qui se trouvent dans celui-ci.

3.1.1 Le gestionnaire de modules de Node.js : npm

npm (*Node.js Package Manager*) est le gestionnaire de modules de Node.js (il est installé avec celui-ci).

Les modules sont installés globalement dans le dossier node_modules, situé au niveau des répertoires système si l'option `-g` est utilisée :

```
npm install -g <module>
```

ou sinon (sans l'option `g`) dans le répertoire courant (mais également dans un dossier nommé node_modules).

3.1.2 Spécification des dépendances : le fichier package.json

Pour spécifier les dépendances nécessaires à la création d'une application Node.js (c'est-à-dire les modules associés aux packages nécessaires à celle-ci), il est recommandé de créer un fichier nommé package.json.

Dans le contexte d'un fichier package.json, nous ne parlerons plus que de packages (et non de modules).

Voici un schéma minimal de ce fichier :

```
{
   "name":        "<nom de l'application>",
   "version":     "<version de l'application>",
   "description": "<description de l'application>",
   "author":      "<nom de l'auteur de l'application>",
   "main":        "<code à exécuter comme point d'entrée>",
```

```
    "scripts": {
        "start": "node <code à exécuter>"
    },
    "dependencies": {
      "<nom du package>": "<version minimale du package à installer>",
      ...
    }
}
```

Pour installer les modules nécessaires, la commande suivante doit être exécutée :

```
npm install
```

Et voici celle qui va lancer le serveur :

```
npm start
```

Pour créer un squelette de fichier *package.json*, utilisez la commande suivante :

```
npm init --yes
```

Dans ce cas, la valeur de la propriété `main` est initialisée à index.js. Expliquons un peu l'intérêt de cette propriété :

Si votre application devient un package (comprenant un ou plusieurs codes réutilisables), la propriété `main` désigne le code qui est le point d'entrée dans le package lors de l'exécution de l'instruction `require()`.

3.2 Création d'un premier serveur Node.js de test

Vous allez écrire votre premier serveur (le bien classique « Hello World ») en utilisant le module HTTP qu'offre Node.js.

▶ Saisissez le code de ce serveur dans le fichier helloAvecNode.js :

```
var http = require('http');

var server = http.createServer((request, => response){
    response.end('Hello World de Node.js');
});

server.listen(8888);
```