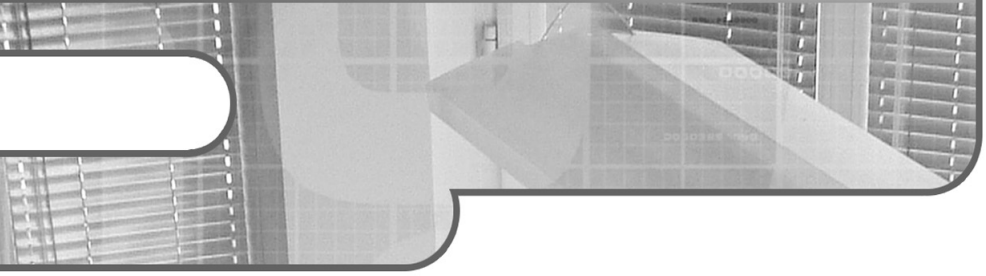


Chapitre 3
Adopter les bonnes pratiques

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 conflits 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.

Cette technique n'est pas adaptée si vous avez plusieurs fonctions à rendre accessibles.

Chapitre 3
Node.js

1. Présentation

Node.js est une plateforme bien adaptée pour les développeurs qui souhaitent utiliser JavaScript côté serveur. Elle permet de créer des applications web et autres types d'applications réseau de manière assez efficace comme des API, des services de streaming de données en temps réel, des serveurs de jeux, des applications IoT (internet des objets). En utilisant le moteur JavaScript V8 de Google Chrome, Node.js apporte JavaScript dans l'environnement serveur, ce qui facilite le partage de code entre le client et le serveur et simplifie le processus de développement.

L'un des aspects intéressants de Node.js est sa manière de gérer les opérations d'entrée/sortie (I/O). Au lieu d'utiliser plusieurs threads pour chaque requête, Node.js fonctionne sur un modèle événementiel et asynchrone. Cela signifie que lorsqu'une tâche I/O est en cours, Node.js peut passer à une autre sans attendre que la première soit terminée. Ce modèle est particulièrement utile pour les applications qui doivent gérer de nombreuses connexions simultanément.

Il bénéficie également d'un écosystème très actif, grâce à son gestionnaire de paquets npm. Les développeurs ont accès à une multitude de modules open source qu'ils peuvent intégrer dans leurs projets, ce qui permet de gagner du temps et de faciliter le développement d'applications. Que vous ayez besoin d'un framework pour construire votre application, d'une bibliothèque pour vous connecter à une base de données ou d'un outil pour automatiser certaines tâches, il y a de fortes chances que vous trouviez ce qu'il vous faut dans l'écosystème de Node.js.

Node.js est reconnue pour sa simplicité d'utilisation. Il convient à une large gamme de projets, des petits sites web aux applications d'entreprise plus complexes. La communauté autour de cette plateforme est également un grand atout, car elle est très active et offre de nombreuses ressources comme des tutoriels et de la documentation, pour aider les développeurs à progresser et à résoudre les problèmes qu'ils pourraient rencontrer.

2. L'installation de Node.js 18

2.1 Sous Ubuntu

Vous pouvez installer Node.js sous Ubuntu en utilisant le gestionnaire de paquets apt. C'est la méthode la plus basique et la plus rapide. Voici les étapes à suivre :

- Ouvrez un terminal sur votre système Ubuntu.
- Mettez à jour la liste des paquets disponibles :

```
sudo apt update
```

- Installez Node.js et npm (le gestionnaire de paquets de Node.js) :

```
sudo apt install nodejs npm
```

- Assurez-vous que l'installation s'est déroulée correctement en vérifiant les versions de Node.js de npm :

```
node -v
npm -v
```

Ces commandes affichent les versions respectives de Node.js et de npm installées sur votre système. Cette méthode installe Node.js et npm à partir des dépôts officiels d'Ubuntu, ce qui facilite les mises à jour et la gestion des packages.

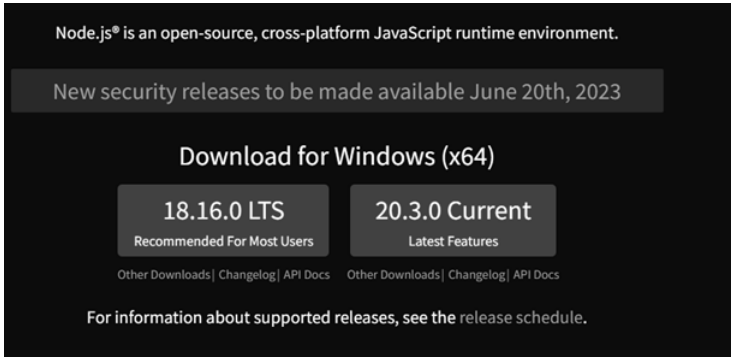
Remarque

La version de Node.js fournie avec apt peut être légèrement plus ancienne que la version LTS (Long Term Support) la plus récente. Si vous souhaitez installer une version plus récente de Node.js, il suffit de la télécharger et de l'installer manuellement en utilisant le site officiel de Node.js.

Une fois l'installation terminée, vous pouvez utiliser Node.js pour développer des applications JavaScript côté serveur.

2.2 Sous Windows

- Rendez-vous sur le site officiel de Node.js à l'adresse : https://nodejs.org



Boutons de téléchargement sur le site nodejs.org

- La page d'accueil propose deux versions de Node.js : LTS et Current. Cliquez sur le bouton **18.16.0 LTS** pour télécharger la dernière version LTS de Node.js (à l'heure de la rédaction de cet ouvrage).

Remarque

La version LTS est recommandée pour la plupart des utilisateurs, car elle est plus stable et bénéficie d'un support à long terme. La version Current est la version la plus à jour. Elle possède des fonctionnalités possiblement non encore stables. Vous pouvez suivre le cycle de développement de Node.js via le lien GitHub : https://github.com/nodejs/Release.

- Une fois le téléchargement terminé, ouvrez le fichier d'installation (.msi) que vous avez téléchargé.
- L'assistant d'installation de Node.js démarre. Suivez les instructions de l'assistant pour configurer l'installation. Vous pouvez généralement accepter les paramètres par défaut, sauf si vous avez des besoins spécifiques.
- Vous pouvez sélectionner les composants supplémentaires à installer. La plupart du temps, il est recommandé de cocher les cases **Automatically install the necessary tools for Node.js** et **Add to PATH** pour que Node.js soit accessible depuis n'importe quel emplacement dans le terminal.
- Cliquez sur le bouton **Next** et suivez les instructions de l'assistant pour terminer l'installation.
- Pour vérifier que Node.js est installé correctement, ouvrez une fenêtre de terminal (invite de commandes ou PowerShell) et exécutez les commandes suivantes :

```
node -v
npm -v
```

Ces commandes affichent respectivement la version de Node.js et la version de npm installées sur votre système.

2.3 Sous Mac

- Rendez-vous sur le site officiel de Node.js à l'adresse : https://nodejs.org/en/download

Les différents packages selon l'environnement

- La page d'accueil propose deux versions de Node.js : LTS et Current. Cliquez sur le bouton **LTS** pour télécharger la dernière version LTS de Node.js.
- Une fois le téléchargement terminé, ouvrez le fichier d'installation (.pkg) que vous avez téléchargé.
- L'assistant d'installation de Node.js démarre. Suivez les instructions de l'assistant pour configurer l'installation. Vous pouvez généralement accepter les paramètres par défaut, sauf si vous avez des besoins spécifiques.
- Vous pouvez sélectionner les composants supplémentaires à installer. La plupart du temps, il est recommandé de cocher les cases **Automatically install the necessary tools for Node.js** et **Install npm package manager** pour installer npm (le gestionnaire de paquets de Node.js).
- Cliquez sur le bouton **Install** et suivez les instructions de l'assistant pour terminer l'installation.
- Pour vérifier que Node.js est installé correctement, ouvrez une fenêtre de terminal (invite de commandes ou PowerShell) et exécutez les commandes suivantes :

```
node -v
npm -v
```

Ces commandes affichent respectivement la version de Node.js et la version de npm installées sur votre système.

3. Les modules

3.1 La notion de module

En Node.js, un module est une unité de code réutilisable qui encapsule un ensemble de fonctionnalités spécifiques. Il permet d'organiser et de structurer le code en le séparant en différentes parties, ce qui facilite la maintenance, la collaboration et la réutilisation du code. Les modules sont basés sur le système de modules CommonJS, qui est une spécification pour les modules JavaScript. Chaque fichier JavaScript dans Node.js est considéré comme un module, et les variables, les fonctions et les objets définis dans ce fichier ne sont pas accessibles depuis d'autres modules par défaut.

Pour utiliser un module dans Node.js, vous devez l'importer à l'aide de l'instruction `require()`. L'instruction `require()` prend en paramètre le chemin du module que vous souhaitez importer, qu'il s'agisse d'un module natif de Node.js, d'un module installé via npm ou d'un module personnalisé que vous avez créé.

Voici un exemple d'utilisation d'un module dans Node.js (cet exemple suppose d'avoir un fichier appelé math.js qui exporte des fonctions mathématiques) :

```
// math.js
exports.add = function(a, b) {
  return a + b;
};

exports.multiply = function(a, b) {
  return a * b;
};
```

Dans un autre fichier JavaScript, vous pouvez importer le module math.js et utiliser ses fonctions :

```
// app.js
const math = require('./math.js');

console.log(math.add(2, 3)); // Affiche 5
console.log(math.multiply(4, 5)); // Affiche 20
```

Dans cet exemple, nous avons utilisé `require()` pour importer le module math.js en spécifiant le chemin relatif vers le fichier. Ensuite, nous avons accédé aux fonctions `add` et `multiply` exportées par le module et nous les avons utilisées dans notre fichier app.js.

C'est ainsi que les modules sont utilisés et importés en Node.js. Ils permettent de découper le code en modules distincts pour une meilleure organisation et une réutilisation facile.

3.2 Le gestionnaire de module npm

Le gestionnaire de module npm (*Node Package Manager*) est un outil qui permet aux développeurs de gérer les packages et les dépendances de leurs projets Node.js. Il est livré avec Node.js et est utilisé pour installer, mettre à jour et supprimer des packages, ainsi que pour gérer les versions des packages installés. Il est également utile pour partager des modules entre les projets et pour travailler avec des bibliothèques tierces. Lorsque vous créez un nouveau projet, vous pouvez initialiser npm en exécutant la commande `npm init`. Cela crée un fichier package.json qui stocke les informations sur votre projet, y compris les dépendances et les scripts. Lorsque vous avez besoin d'ajouter une nouvelle dépendance à votre projet, vous pouvez l'installer en utilisant la commande `npm install`. Il est aussi possible d'installer des packages pour les utiliser dans n'importe quel projet. Enfin, npm dispose d'un grand nombre de packages gratuits et open source. Il est donc courant de trouver une solution à un problème en cherchant une bibliothèque npm. Cela permet aux développeurs de gagner du temps et de se concentrer sur la logique métier.

Le dossier node_modules est un répertoire utilisé par Node.js pour stocker les modules installés via npm. Lorsque vous installez un module à l'aide de la commande `npm install`, les fichiers du module sont téléchargés depuis le registre npm et placés dans le dossier node_modules de votre projet. Chaque module installé via npm a son propre dossier dans node_modules, et ce dossier porte le nom du module en question. À l'intérieur de ce dossier, vous trouverez les fichiers JavaScript, les dépendances et autres ressources spécifiques au module.

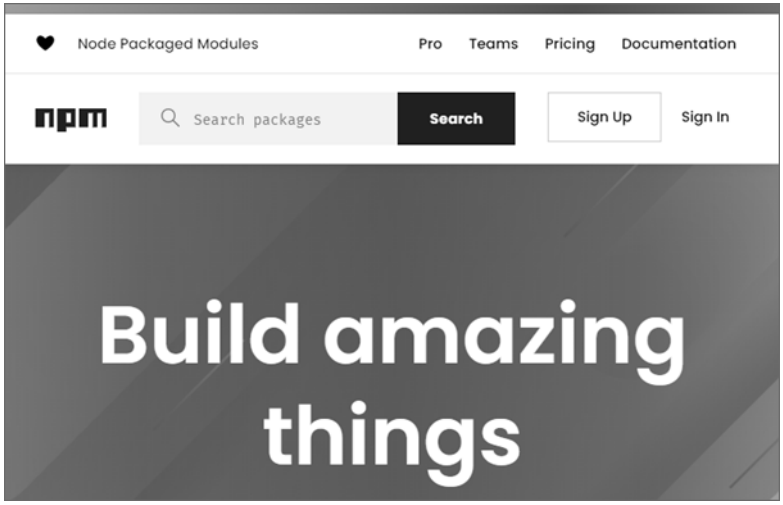
Par exemple, si vous installez le module Express, un dossier express est créé dans node_modules. Ce dossier express contient tous les fichiers nécessaires à l'utilisation du module Express dans votre projet.

Lorsque vous utilisez des modules dans votre code, Node.js recherche automatiquement ces modules dans le dossier node_modules. Vous pouvez les importer à l'aide de l'instruction `require()` sans spécifier le chemin complet du module.

Remarque

Les modules installés occupent de la place en stockage. Il est courant de supprimer le dossier node_modules ou alors de l'ignorer lorsque l'on versionne un projet. Le dossier node_module n'est pas nécessaire pour le stockage, car toutes les informations utiles sont présentes dans packages.json. En revanche, le dossier node_modules est indispensable pour le lancement de l'application (il suffit de le générer de nouveau via la commande `npm install`).

3.3 npmjs.com



Présentation du site npmjs.com