Code : Bonnes pratiques et Outils

Sommaire

1. Dépendances du Code

1.1. Gestion des dépendances

Qu'est-ce qu'une dépendance ?

Un projet informatiques repose sur un ensemble de composants logiciels. Ces composants logiciels peuvent être :

- des composants **créés par le(s) programmeur(s)** qui travaillent sur le projet
- des composants logiciels qui ont été **créés par des tiers**

La combinaison de ces composants avec les autres éléments constitutifs d'un projet informatique contribuent à la création d'un **système modulaire**.

Un **système modulaire** est donc un système qui repose sur la combinaison de composants logiciels.

Le potentiel de réutilisabilité de certains de ces composants est faible et, pour d'autres, **le potentiel de réutilisabilité est fort**.

Les **composants logiciels réutilisables** :

- Peuvent être utilisés au sein d'autres projets informatiques dont la **nature** et les **objectifs** diffèrent;
- Peuvent prendre la forme de **librairies de composants** ou de **composants individuels**.

Un **composant logiciel réutilisable** dont dépend un projet peut être qualifié de **dépendance du projet**. Le projet ne peut pas fonctionner sans ce composant.

Qu'est-ce que la gestion des dépendances ?

La **gestion des dépendances** est une **technique** pour :

- **Déclarer**;
- **Retrouver**;
- et **Utiliser**

de façon **automatique** les **dépendances** nécessaires au bon fonctionnement d'un projet.

1.2. Gestionnaires de dépendances

Qu'est-ce qu'un gestionnaire de dépendances ?

Pour de nombreux langages de programmation, des outils ont été créés par des organisations et/ou des communautés open-source pour faciliter la gestion des dépendances, les **gestionnaires de dépendances**.

Les **gestionnaires de dépendances** peuvent également être appelés **gestionnaires de paquets**.

Quels principes fondamentaux ?

Quel que soit le langage, les **gestionnaires de dépendances** fonctionnent tous sur les mêmes principes fondamentaux :

- Pour **Déclarer** les dépendances, ils proposent de créer un **fichier de configuration** contenant la **liste des dépendances** et leur **version**. Cette version respecte les règles sémantiques de version **semver** (détaillées ici);
- Pour **Retrouver** les dépendances, il mettent à disposition de leurs utilisateurs un **site internet - un registre -** qui est fréquemment mis à jour et qui met à disposition du gestionnaire de dépendances la **liste des dépendances** disponibles ainsi que leur **localisation** :
    - Généralement, l'**adresse d'un dépôt GIT** pour télécharger la dépendance;
    - L'**adresse d'un dépôt lié à un autre système de gestion de code source** (SVN, Mercurial, ...)
    - Ou, plus simplement, un **URL de téléchargement pour un format d'archive**.

- Enfin, pour **Utiliser** les dépendances, les gestionnaires de dépendances installent les dépendances au sein du projet selon des **règles d'installation prédictibles**, généralement un dossier dont le nom est fixé par défaut ou par la configuration, et des **règles d'utilisation fixées** par convention ou par la configuration.

Quels outils pour JavaScript/ECMAScript ?

Dans l'univers JavaScript/ECMAScript, 3 outils se sont imposés pour gérer les dépendances (du plus populaire au moins populaire) :

- **npm**, le gestionnaire de dépendances open source officiel de Node.js qui existe depuis 2010 (Site Officiel);
- **yarn**, un gestionnaire de dépendances open source développé à l'origine par Facebook qui existe depuis 2016 (Site Officiel);
- **bower** (progressivement deprécié au profit des précedents), un gestionnaire de dépendances open source développé à l'origine par Twitter qui existe depuis 2012 (Site Officiel).

Nous intéresserons principalement à **npm** et **yarn**.

Quelles sont les caractéristiques clés de npm ?

- **Installation :** npm est installé en même temps que Node.js;
- **Utilisation :** s'utilise à partir de l'invite de commande.

```
# Afficher la version de npm
npm --version
```

- **Documentation :** la documentation officielle de npm est docs.npmjs.com;
- **Configuration :** le fichier de configuration de npm est un fichier intitulé `package.json` situé à la racine d'un projet. Ce fichier peut être initialisé par npm comme suit à partir de la racine d'un projet :

```
npm init
```

- **Registre :** le registre de npm est www.npmjs.com;
- **Utilisation :** lorsqu'une installation de dépendances est effectuée à la racine d'un projet, la dépendance est installée par convention dans un sous-dossier du dossier `node_modules` du projet.

```
# Installer une dépendance avec npm
# à partir de la racine d'un projet
npm i [nom-de-la-dépendance]
```

Pour démarrer un projet basé sur Node.js et qui utilise des dépendances gérées par npm :

```
node [fichier-principal-du-projet]
```

Quelles sont les caractèristiques clés de yarn ?

- **Installation :** yarn peut être installé en suivant la procédure proposée ici.

```
# A partir d'une invite de commande
# ouverte en tant qu'ADMINISTRATEUR
# du système :
corepack enable
```

- **Utilisation :** s'utilise à partir de l'invite de commande.

```
# Afficher la version de yarn
yarn --version
```

- **Documentation :** la documentation officielle de yarn est yarnpkg.com/getting-started;
- **Configuration :** les fichiers de configuration de yarn sont 2 fichiers intitulés `package.json` (pour les dépendances) et `.yarnrc.yml` (pour la configuration de yarn lui-même) situés à la racine d'un projet. Ces fichiers peuvent être initialisés par yarn comme suit à partir de la racine d'un projet :

```
yarn init -2
```

- **Registre :** le registre de yarn est yarnpkg.com;
- **Utilisation :** lorsqu'une installation de dépendances est effectuée à la racine d'un projet, la dépendance est installée par convention dans un sous-dossier du dossier `.yarn` du projet.

```
# Installer une dépendance avec yarn
# à partir de la racine d'un projet
yarn add [nom-de-la-dépendance]
yarn install
```

Pour démarrer un projet basé sur Node.js et qui utilise des dépendances gérées par yarn :

```
yarn node [fichier-principal-du-projet]
```

2. Documentation du Code

1. Normes pour l'écriture des commentaires
2. Générer une documentation technique

2.1. Normes pour l'écriture des commentaires

Au sein d'un programme informatique, on trouve du **code exécutable** écrit par des programmeurs.

On peut également y placer des **commentaires** qui n'ont pas vocation à être exécutés. Les **commentaires** sont des **descriptions qui peuvent être lues par un humain** et qui **expliquent ce que fait le code**.

Les commentaires :

- Facilitent la **maintenance** du code;
- Aident à la **détection des erreurs** lors de la relecture du code;
- Facilitent la **transmission de la connaissance** au sein des équipes;

Un **code bien documenté** est aussi important qu'un **code bien écrit**.

La plupart des languages proposent des **normes** pour l'écriture des commentaires.

Ces normes ont pour objectif de permettre à des programmeurs venant d'horizons différents de pouvoir lire et comprendre facilement les commentaires écrits par leurs pairs.

La normalisation des commentaires offre, certes, des avantages en termes de lisibilité mais permet également de créer des outils qui sont en mesure de **générer une documentation technique à partir des commentaires**.

En JavaScript/ECMAScript, plusieurs solutions sont à la fois :

- un **ensemble de normes** pour l'écriture des commentaires;
- un **outil** pour générer une documentation technique à partir des commentaires.

On en citera 3 par ordre de popularité (de la plus populaire à la moins populaire) :

- JSDoc;
- ESDoc;
- et Documentation;

Nous nous attarderons particulièrement sur JSDoc dont :

- Les règles peuvent être consultées sur le site officiel;
- Et la procédure d'installation et d'utilisation sur le Depôt GIT de JSDoc.

Pour faciliter la création de commentaires conforme à la spécification JSDoc, on peut utiliser l'extension Document This de Visual Studio Code.

ATTENTION, l'extension de Visual Studio Code ne fonctionne que si JSDoc est installé sur le projet.

Lorsque l'extension est installée, on peut utiliser le raccourci - à taper 2 fois d'affilée - : `Ctrl+Alt+D` sur un code selectionné pour générer une partie des commentaires.

2.2. Générer une documentation technique

Pour installer JSDoc, après avoir initialisé un projet avec npm :

```
npm i jsdoc
```

Pour utiliser JSDoc pour générer de la documentation à partir des commentaires :

- Pour un fichier de code source `a.js` :

  ```
  npx jsdoc a.js
  ```

- Pour un fichier de code source `a.js` et un fichier `b.js` :

  ```
  npx jsdoc a.js b.js
  ```

- Pour tous les fichiers se terminant par l'extension `.js` :

  ```
  npx jsdoc *.js
  ```

- Pour tous les fichiers dans un dossier `src` et tous les sous-dossiers du dossier `src` :

  ```
  npx jsdoc -r src
  ```

Par défaut la documentation est générée dans le dossier `out` à la racine du projet.

Le fichier `index.html` est le point d'entrée de la documentation.

3. Lisibilité et qualité du Code

1. Utiliser un outil de contrôle de la forme
2. Utiliser un outil de contrôle de la qualité

3.1. Utiliser un outil de contrôle de la forme

La **normalisation** de la présentation et de la mise en forme du code propose les même avantages que l'écriture de commentaires. Elle facilite :

- La **maintenance** du code;
- La **détection des erreurs** au sein du code;
- La **transmission de connaissances** au sein d'une équipe;

En JavaScript/ECMAScript, l'utilitaire Prettier est une solution reconnue pour la présentation et la mise en forme du code.

La documentation pour l'installation et l'utilisation de Prettier est disponible sur le site officiel.

Pour **installer** Prettier à partir de la racine d'un projet initialisé avec `npm` :

```
npm install --save-dev --save-exact prettier
```

Prettier nécessite un fichier de configuration `.prettierrc.json` qui peut être vide par défaut. Pour créer ce fichier :

```
echo {}> .prettierrc.json
```

Enfin pour formatter de façon uniforme tous les fichiers d'un dossier `src` avec Prettier :

```
npx prettier src
```

On peut également installer l'extension Prettier - Code Formatter de Visual Studio Code pour pouvoir lancer automatiquement Prettier à partir de l'éditeur sur les fichiers sur lesquels on travaille.

ATTENTION, l'extension de Visual Studio Code ne fonctionne que si Prettier est installé sur le projet.

Pour activer le formattage automatique du code des fichiers javascript, il faut ajouter la configuration suivante aux paramètres de Visual Studio Code :

```
// Paramètre par défaut
"editor.formatOnSave": false,
// Activer Prettier à la sauvegarde pour le langage JavaScript
"[javascript]": {
    "editor.formatOnSave": true
},
// Activer Prettier à la sauvegarde pour le langage TypeScript aussi par exemple
"[typescript]": {
    "editor.formatOnSave": true
}
```

3.2. Utiliser un outil de contrôle de la qualité

En plus de la présentation et la mise en forme du code, on peut utiliser un *Linter*. Un *Linter* est un outil qui permet d'**analyser du code source** pour détecter :

- Des erreurs de programmation ou des éventuels bugs;
- Des constructions syntaxiques qui peuvent engendrer des erreurs;
- Des problèmes de présentation du code (le *Linter* ne met pas en forme le code comme un outil de mise en forme du code);

En JavaScript/ECMAScript, l'utilitaire ESLint est une solution reconnue pour l'analyse du code.

La documentation pour l'installation et l'utilisation de ESLint est disponible sur le site officiel.

Pour **installer** ESLint à partir de la racine d'un projet initialisé avec `npm` :

```
npm install eslint --save-dev
```

ESLint nécessite un fichier de configuration `.eslintrc.js` ou `.eslintrc.json` ou `.eslintrc.yml` qui contient des couples clés/valeurs de configuration par défaut. Pour créer ce fichier :

```
npx eslint --init
# Puis répondre au questionnaire en fonction de la nature du projet...
```

Enfin pour *Linter* tous les fichiers d'un dossier `src` avec ESLint :

```
npx eslint src
```

On peut également installer l'extension ESLint de Visual Studio Code pour pouvoir lancer automatiquement ESLint à partir de l'éditeur sur les fichiers sur lesquels on travaille.

ATTENTION, l'extension de Visual Studio Code ne fonctionne que si ESLint est installé sur le projet.

Lorsque l'extension est activée, les erreurs détectées apparaîtront dans l'onglet *Problèmes* de Visual Studio Code lors de l'édition d'un fichier.

4. Modularisation du Code

4.1. Principe de la modularisation

Les principes de la modularisation sont détaillés sur la page du site officiel du MDN concernant les modules.

En résumé, la modularisation consiste à proposer une approche pour **diviser un programme** en plusieurs modules indépendants les uns des autres.

Plusieurs approches ont ainsi été développées au fil du temps sous la forme de *librairies* ou de *framework* ou *nativement* prise en charge par les systèmes.

A ce titre, on citera principalement les approches :

- `CommonJS`, abrégée `CJS`, nativement disponible sur Node.js à l'aide de la méthode `require()` pour faire appel à un module et `exports` pour déclarer la valeur exportée par le module. Cette approche s'emploie généralement en *back-end*.
- `Asynchronous Module Definition`, abrégée `AMD`, implémentée à l'aide de *librairies* comme Require.JS qui propose une fonction `requirejs()` pour faire appel à un module. Cette approche s'emploie généralement en *front-end*.
- `ECMAScript Modules`, abrégée `ESM`, nativement disponible sur les systèmes prenant en charge les dernières versions du JavaScript/ECMAScript et qui s'emploie à l'aide des **mots-clés** `import` pour faire appel à un module et `export` pour déclarer une valeur exportée par le module. Cette approche s'emploie généralement en *front-end* ou *back-end*.

4.2. Techniques de modularisation

Approche CJS

Cette approche est principalement employée en *back-end*.

L'approche CJS est nativement disponible sur Node.JS et fait l'objet d'une Page de la documentation officielle.

Cette méthode est implémentée à l'aide de :

- La méthode `require()` est exécutée au sein du *module principal* qui nécessite le résultat de l'exécution d'un *module secondaire*;
- La valeur de la propriété `exports` est définie à l'issue de l'exécution du code du *module secondaire* et sa valeur sera la valeur retournée par la méthode `require()` du *module principal*.

C'est une approche **synchrone**. Tant que la totalité du code du *module secondaire* n'est pas exécutée, la méthode `require()` **bloque** l'exécution du code du *module principal*.

On appelle cela l'**import statique** - *static import*.

Exemple sur Node.JS :

***module principal* : `principal.js`**

```
// On utilise require pour demander l'exécution
// du code du module secondaire :
const valeurFournie = require('./secondaire.js');
// On affiche la valeur retournée
// par require dans la console :
console.log(valeurFournie)
```

Affiche dans la console :

```
{
    fonctionFournie: function(){},
    variableFournie: "Ceci est un texte",
    operationFournie: 6
}
```

***module secondaire* : `secondaire.js`**

```
// On définit des variables/valeurs
const uneFonction = function(){};
const uneVariable = "Ceci est un texte";
const uneOperation = 3 + 3;
// On assigne à exports la valeur (ici un objet)
// qui sera retournée par l'exécution de require() :
exports = {
    fonctionFournie: uneFonction,
    variableFournie: uneVariable,
    operationFournie: uneOperation
};
```

Cette approche permet de modulariser un programme. **Seul le premier appel** à `require()` entraîne l'execution du code du *module secondaire*.

Les appels suivants retournent la valeur créée par le *module secondaire* lors de la première exécution. Cette méthode dispose donc d'un mécanisme de ***cache***.

Approche AMD

Cette approche est principalement employée en *front-end*.

L'approche AMD est implémentée à l'aide de *frameworks* ou de *librairies* comme *Require.JS*, *WebPack* ou *Babel* après *transpilation*. Ici nous l'introduirons à l'aide de la librairie Require.JS.

Cette méthode est implémentée à l'aide de :

- Une balise `<script>` pour charger la *librairie* Require.JS avec un attribut `data-main` dont la valeur est l'URL relative du *module principal*;
- La méthode `requirejs()` qui est exécutée au sein du *module principal* qui nécessite l'exécution d'un *module secondaire*;

C'est une approche **asynchrone**. L'exécution du code du *module secondaire* **ne bloque pas** l'exécution du code du *module principal*.

L'exécution de la méthode `requirejs()` **ne retourne pas** de valeur.

On appelle cela l'**import dynamique** - *dynamic import*.

Exemple sur un navigateur moderne quelconque :

***Page HTML* : `index.html`**

```
<!DOCTYPE html>
<html>
    <head>
        <title>Page HTML</title>
        <!-- l'attribut data-main attribute
                permet à Require.JS de charger
                scripts/principal.js après le
                chargement de require.js -->
        <script
            data-main="scripts/principal"
            src="scripts/require.js"
        ></script>
    </head>
    <body>
        <h1>Page HTML</h1>
    </body>
</html>
```

***module principal* dans : `scripts/principal.js`**

```
// On utilise requirejs pour demander l'exécution
// du code du module secondaire :
requirejs(['scripts/secondaire'], function(valeurFournie){
    // Ce callback est exécuté après l'exécution
    // du code du module secondaire. l'argument
    // valeurFournie reçoit la valeur retournée
    // par le code du module secondaire.
    // On affiche la valeur reçue par
    // par le callback dans la console :
    console.log(valeurFournie)
});
// La fonction requirejs NE RETOURNE RIEN
```

Affiche dans la console :

```
{
    fonctionFournie: function(){},
    variableFournie: "Ceci est un texte",
    operationFournie: 6
}
```

***module secondaire* dans : `scripts/secondaire.js`**

```
// On définit des variables/valeurs
const uneFonction = function(){};
const uneVariable = "Ceci est un texte";
const uneOperation = 3 + 3;
// On assigne à exports la valeur (ici un objet)
// qui sera retournée par l'exécution de require() :
return {
    fonctionFournie: uneFonction,
    variableFournie: uneVariable,
    operationFournie: uneOperation
};
```

L'exécution de la méthode `requirejs()` entraîne le **téléchargement** et l'**exécution** du code du *module secondaire*.

Les exécution suivantes n'entraînent pas le **téléchargement** et l'**exécution** du code du *module secondaire*. La valeur fournie est la même que celle résultant de la première exécution du code du *module secondaire*.

En *front-end*, en plus du mécanisme de cache, la modularisation AMD permet de moins solliciter le réseau et d'améliorer les performances applicatives.

Approche ESM

Cette approche peut être employée en *back-end* ou en *front-end* si le système est suffisamment récent (version récente de node.js ou version récente de navigateur Internet).

L'approche ESM est nativement disponible sur Node.JS. Elle fait l'objet d'une Page de la documentation officielle de Node.JS.

L'approche ESM est nativement disponible sur les navigateurs récents. Elle fait l'objet d'une Page de la documentation officielle du MDN.

Cette méthode est implémentée à l'aide de :

- En *front-end* uniquement, des balises `<script>` avec l'attribut `type="module"` obligatoire pour tous les scripts;
- En *front-end* et *en back-end*, du mot-clé `import` pour les **imports statiques** ou de la méthode `import()` pour les **imports dynamiques**. Le mot-clé ou la méthode sont utilisés au sein du *module principal* qui nécessite le résultat de l'exécution d'un *module secondaire*;
- En *front-end* et *en back-end*, du mot-clé `export` qui permet de définir chaque valeur qui sera fournie au *module principal*.

C'est une approche **synchrone** si on utilise les **imports statiques**. C'est une approche **asynchrone** si on utilise les **imports dynamiques**.

ATTENTION, en *front-end*, ESM ne fonctionne que si le document HTML initial ainsi que les scripts sont fournis par un serveur.

Exemple d'**import statique - synchrone** sur un navigateur moderne quelconque :

***Page HTML* : `index.html`** ***module principal* dans : `scripts/principal.js`** ***module secondaire* dans : `scripts/secondaire.js`**

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Page HTML</title>
        <script
            type="module"
            src="scripts/principal.js"
        ></script>
        <script
            type="module"
            src="scripts/secondaire.js"
        ></script>
    </head>
    <body>
        <h1>Page HTML</h1>
    </body>
</html>
```

```
// On utilise le mot clé import pour demander
// l'exécution du code du module secondaire :
import * as valeurFournie from "scripts/secondaire.js"
// On affiche la valeur retournée
// par require dans la console :
console.log(valeurFournie)
```

Affiche dans la console :

```
{
    fonctionFournie: function(){},
    variableFournie: "Ceci est un texte",
    operationFournie: 6
}
```

```
// On définit des variables/valeurs
const uneFonction = function(){};
const uneVariable = "Ceci est un texte";
const uneOperation = 3 + 3;
// On utilise le mot-clé export pour définir
// chaque variable dont la valeur sera fournie
// et sous quel nom de propriété :
export uneFonction as fonctionFournie;
export uneVariable as variableFournie;
export uneOperation as operationFournie;
```

Exemple d'**import dynamique - asynchrone** sur un navigateur moderne quelconque :

***Page HTML* : `index.html`**

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Page HTML</title>
        <script
            type="module"
            src="scripts/principal.js"
        ></script>
        <script
            type="module"
            src="scripts/secondaire.js"
        ></script>
    </head>
    <body>
        <h1>Page HTML</h1>
    </body>
</html>
```

***module principal* dans : `scripts/principal.js`**

```
// On utilise la fonction import() pour demander
// l'exécution du code du module secondaire :
import("scripts/secondaire.js").then(function(valeurFournie){
    // Ce callback est exécuté après l'exécution
    // du code du module secondaire. l'argument
    // valeurFournie reçoit la valeur exportée
    // par le code du module secondaire.
    // On affiche la valeur reçue par
    // par le callback dans la console :
    console.log(valeurFournie)
})
// La fonction import() NE RETOURNE RIEN
```

Affiche dans la console :

```
{
    fonctionFournie: function(){},
    variableFournie: "Ceci est un texte",
    operationFournie: 6
}
```

***module secondaire* dans : `scripts/secondaire.js`**

```
// On définit des variables/valeurs
const uneFonction = function(){};
const uneVariable = "Ceci est un texte";
const uneOperation = 3 + 3;
// On utilise le mot-clé export pour définir
// chaque variable dont la valeur sera fournie
// et sous quel nom de propriété :
export uneFonction as fonctionFournie;
export uneVariable as variableFournie;
export uneOperation as operationFournie;
```

Dans le cas d'un programme en *back-end*, on préféra les **imports statiques - synchrones** qui sont simples à implémenter et qui n'ont pas un impact important sur les performances applicatives.

Dans le cas d'un programme en *front-end*, on préféra les **imports dynamiques - asynchrones** qui permettent d'améliorer significativement les performances applicatives.

5. Design patterns de Code

5.1. Objectifs d'un design pattern

En génie logiciel, un *design pattern* logiciel est une solution réutilisable pour résoudre un problème classique rencontré dans un contexte précis.

Les *design patterns* font partie de ce qu'on appelle les bonnes pratiques du développement logiciel.

Les *design patterns* ne se préoccupent pas de la nature des fonctionnalités qui résultent de leur implémentation. Il visent à décrire une façon de programmer pour atteindre les objectifs suivants :

- Faciliter la collaboration au sein d'une équipe;
- Faciliter la réutilisabilité du code produit;
- Faciliter la maintenance du code produit.

Dans cette optique, les *design patterns* peuvent être classés en plusieurs catégories :

- Les *design patterns* de **création** qui s'intéressent à la façon de créer des modules;
- Les *design patterns* de **structure** qui s'intéressent à l'organisation des différents modules;
- Les *design patterns* de **comportement** qui s'intéressent à la façon dont les différents modules communiquent entre eux.

L'implémentation des *design patterns* est indépendante du langage utilisé. En d'autres termes, on retrouvent les mêmes *design patterns* dans tous les langages de programmation.

Leur implémentation diffère, bien sûr, en fonction de la syntaxe et des API proposées par le langage.

5.2. Découverte des design pattern

Cette partie sera illustrée par les exemples de cours.

Cependant, et pour aller plus loin sur les *design patterns* en JavaScript/ECMAScript, je vous recommande l'excellent livre de Addy Osmani - Learning JavaScript Design Patterns (*gratuit en version numérique*) qui est une référence en la matière.

Tous les exemples proposés sont illustrés et peuvent être testés à la lecture. Je vous invite vivement à vous y intéresser.