Portfolio en ligne

Guillaume Pelletier-Auger

29 septembre 2016

Résumé

Ce portfolio d'art numérique est un site statique généré avec Node.js.

1 Génération d'un site Web statique avec Node.js

Je crée un portfolio en ligne pour mon travail en art numérique et je planifie le mettre à jour fréquemment. Le processus de mise à jour doit être aussi simple que possible. Mon but est de générer le contenu HTML en utilisant une série de modules Node.js. Chacun de ces modules sera traité en utilisant le module *fs* (pour *file system*) de Node.js.

Voici ce que j'ai en ce moment :

```
var fs = require("fs");

var header = `<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Guillaume Pelletier-Auger</title>
</head>
`;

var body = `
<body><h1>I'm writing my own HTML files with Node.js. It's pretty neat.</h1>
It's actually like... awesome! I can now think of a website as a data structure.
</body>
</html>
`;

fs.writeFile('index2.html', header + body, function(err) {
    if (err) {
        return console.error(err);
    } else {
        console.log("Data written successfully!");
    }
});
```

Je songeais précédemment à utiliser un générateur de sites Web statiques (*Static Site Engine*) comme Harp.js, mais il m'est clair maintenant que ce sera beaucoup plus simple de tout faire moi-même dans Node.js. Mon système aura ainsi une bien plus grande indépendance fonctionnelle.

2 Un site Web est une structure de données

Je dois penser à mon portfolio comme étant une structure de données. J'ai commencé avec ce fichier JSON :

```
{
    header : "<head></head><body><h1></h1>",
    items : {
        oscillators : {
            title_fr : "Les Oscillateurs",
            title_en : "Oscillators",
            content_fr : `

            `,
            content_en : `
            `,
        },
        joy_and_confusion : {

        },
        dunes : {
        }
    }
}
```

Je dois avoir une fonction qui ressemble à ça :

```
function buildWorks() {
    var content = `` ;
    for (var i = 0 ; i < portfolio.items.length ; i++)  {
        var title = portfolio.items[i].title ;
        content = content + `<img src="${title}.jpg"></a>` ;
    }
}
```

3 Liste de fichiers comme base de données

Mon système sera beaucoup plus efficace si chaque page de mon portfolio correspond à un fichier différent dans ma base de données. Il me serait en effet désagréable de travailler sur les diverses pages de mon portfolio si elles étaient toutes écrites dans le même fichier JavaScript ou JSON. Ça deviendrait rapidement lourd et pas clair.

En utilisant la méthode *readdirSync* du module *fs* de Node.js, j'obtiens la liste complète des fichiers qui sont dans le dossier *Pages*. Les fichiers qui seront dans le dossier *Pages* seront des modules Node.js (d'où l'utilisation de la méthode *require*) qui renvoient un objet JavaScript littéral. L'objet contient les propriétés fr, en et link.

```
var fs = require("fs");
var files = fs.readdirSync('pages');
var page = require('./pages/' + files[0]);
```

Voici à quoi ressemble une page définie comme module Node.js, pour l'instant :

```
exports.fr = {
    title : "Les Dunes",
    description : "Une série d'image générées par des fonctions itératives.",
    content : `<i>Les Dunes</i> est une série d'images générées par une variété de fonctions itératives.
    Elles ont été créées avec p5.js.`
};

exports.en = {
    title : "Dunes",
    description : "A series of images generated by iterated functions.",
    content : `<i>Dunes</i> is a series of images generated by various iterated functions.
    They were created in JavaScript with the p5.js library.`
}

exports.link = null;
```

La fonction qui va générer toutes mes pages doit donc traiter ces données. Elle doit :

1. Vérifier que toutes les données sont présentes.
2. Vérifier si la propriété *link* est *null* ou non.
   (a) Si *link* est *null*, ça veut dire qu'il s'agit d'une page qui sera hébergée directement sur mon site Web. La fonction doit donc créer un fichier HTML en utilisant les données contenues dans l'objet renvoyé par le module.
   (b) Si la propriété n'est pas *null*, ça veut dire que cet élément du portfolio est hébergé sur un site externe. La fonction ne crée donc aucun fichier HTML pour cet élément du portfolio.
3. La fonction doit utiliser chaque élément dans le dossier *Pages* pour générer les fichiers index.html, index-en.html et index-fr.html. Forcément, la fonction doit aussi lire dans un autre

dossier, nommé *Thumbnails*, qui contiendra une image pour chaque élément du portfolio.

Ma boucle pour traiter chacun des fichiers dans le dossier *pages* ressemblera donc à ça :

```
//We get a reference to the Node.js filesystem module.
var fs = require("fs");

//We let the files variable be an array containing
//the full names of all the files inside the "pages" folder.
var files = fs.readdirSync('pages');

//We loop through the list of files.
//If the currently processed file doesn't have an external link, we run the generatePage() function.
for (var i = 0; i < files.length; i++) {
    var page = require('./pages/' + files[i]);
    if (!files[i].link){
        generatePage(files[i]);
    }
}

//We call the generateMosaic() function twice, sending it the list of files and language option.
generateMosaic(files, "fr");
generateMosaic(files, "en");
```

La fonction *generateMosaic()* ressemblerait à ça :

```
function generateMosaic(files, language) {
    //We start writing the HTML content.
    var mosaic = `<div class="mosaic">`;

    //We loop through all the files.
    for (var i = 0; i < files.length; i++) {
        //We let "page" be the file that is currently processed.
        var page = require('./pages/' + files[i]);

        if (language == "fr") {
            var pageLang = page.fr;
        } else if (language == "en") {
            var pageLang = page.en;
        }

        //I reformat the filename using a regular expression.
        //This is done because the filenames in the "pages" folder are .js,
        //and I need .html files for the output.

        var r = /([^ :\\/]*?)(?:\.([^  :\\/.]*))?$/;

        var formattedName = files[i].match(r)[1];

        //We form the full link, considering also the language argument.
        //If page.link exists, we pick it. Otherwise, we build the link.
        var link = (page.link || "./" + language + "/" + formattedName + ".html");
```

```
        //Next, we need to build the divs containing the title and description of the portfolio item.
        var title = pageLang.title;
        var description = pageLang.description;

        var itemDiv = `<div class = "portfolio-item"><div class = "thumbnail">
        <a href="${link}"><img src="../images/${formattedName}.jpg"></a></div>
        <div class = "portfolio-description">
        <h2>${title}</h2><p>${description}</p></div>`;

        mosaic = mosaic + itemDiv;
    }
    //Closing the div of the mosaic
    mosaic = mosaic + `</div>`;
}
```

Je n'aime pas vraiment cette immense fonction un peu désordonnée. Peut-être devrais-je apprendre des éléments de programmation fonctionnelle pour améliorer mon code.

Voici la fonction *generatePage()* :

```
function generatePage(file) {
    var header = generateHeader(file);

    var footer = ``;

}
```

4 Création des liens de navigation

La création des liens de navigation sera clairement la partie la plus complexe de mon système.

5 Questions ouvertes

Comment ferais-je pour trier les divers éléments du portfolio ? C'est une chose fondamentale. Pour l'instant, mon système ne fera que créer chaque élément du portfolio en ordre alphabétique. Il me faut absolument un système pour trier ces éléments à ma volonté.

Devrais-je avoir un fichier qui contient plusieurs pages de mon portfolio et optionnellement des liens vers des fichiers externes qui définiraient des pages de portfolio plus complexes ? Je songe au fait que de nombreux items de mon portfolio pourrait être hébergés sur d'autres serveurs, ce qui rendrait l'acte de créer un fichier JavaScript pour définir chacun de ces items un peu redondant.

Une idée de réponse à la première question : je pourrais utiliser des nombres au début de mes noms de fichiers, comme *0001-dunes.js* et ensuite mon programme utiliserait ces nombres pour classer les items du portfolio mais ignorerait ces nombres lors de la capture du nom de fichier. *0001-dunes.js* deviendrait *dunes.html*.

Ce n'est pas une bonne solution. Le mieux ce serait d'avoir une liste quelque part qui contiendrait les items que je veux afficher dans l'ordre. Ce système me permettrait d'ajouter aisément un item en début de liste sans modifier les items suivants. Il me permettrait aussi d'ignorer des items à volonté. Je pourrais créer un système qui prendrait cette liste et chercherait chacun des items tout d'abord dans un fichier commun comme *portfolio.json* et ensuite dans le dossier *pages*.

6 La base de données

Ma base de données sera structurée de la façon suivante : tout est contenu dans le dossier *pages*. Le dossier contient 2 types de choses : un fichier *pages.js* qui définit plusieurs pages de mon site Web, et d'autres fichiers JavaScript qui définissent une seule page. J'utilise ce système pour pouvoir à la fois avoir de nombreuses pages simples (et dont les liens mènent à l'extérieur de mon site Web) et aussi plusieurs pages plus complexes (des articles complets, des pages contenant beaucoup d'éléments) qui seront plus agréables à maintenir dans des fichiers individuels. Par exemple, ma page *about.html* sera suffisamment volumineuse qu'il me sera beaucoup plus agréable de l'écrire dans un fichier distinct.

Tous les fichiers contenus dans le dossier *pages* sont des modules Node.js. Le fichier *pages.js* renvoie 2 objets : *list* et *pages*. L'objet *list* est une liste de toutes les pages que je souhaite afficher dans mon portfolio. Cette liste me sert à déterminer quelles pages apparaissent dans mon portfolio et dans quel ordre. Cette liste me permet de trier les pages et d'en exclure certaines, à ma volonté.

Voici à quoi ressemble le module *pages.js* pour l'instant :

```
exports.list = ["dunes", "about"];

exports.pages = {

    oscillators : {
        fr : {
            title : "Les Oscillateurs",
            description : "Un film d'animation en noir et blanc réalisé avec p5.js."
        },
        en : {
            title : "Oscillators",
            description : "A black and white animation film made with p5.js."
        }
        link : "http ://www.vimeo.com/pelletierauger"
    },

    joy_and_confusion : {
        fr : {
            title : "Les joies confuses",
            description : "Un film d'animation coloré réalisé avec p5.js."
        },
        en : {
            title : "Joy and Confusion",
            description : "A colorful animation film made with p5.js."
        },
        link : "http ://www.vimeo.com/pelletierauger"
    }
};
```

J'ai fait quelques expérimentations avec la fonction map, mais je laisse ça de côté pour l'instant. C'est un peu trop compliqué pour aujourd'hui.

```
var pages = require("./pages/pages.js");
//pages.map(makePage);

//pages.list.map(function(page){
//    makePage(page);
//});
```

Voici à quoi ressemble mon programme pour l'instant (1er octobre 2016) :

```
//First, we get the list of all the pages we need to make, in order, in pages.list, and
//we also get all the pages that are defined inside the pages.js module, in pages.pages.
var pages = require("./pages/pages.js");
makeIndexes(pages);

//For each element in the list of pages, the single page is either a property of pages.js module,
//or it's a property of a standalone module.
//Let page be the property pages.list[i] if it exists, otherwise load a standalone module.

for (var i = 0; i < pages.list.length; i++) {
    var page = (pages.pages[pages.list[i]] || require('./pages/' + pages.list[i]));
    console.log(page);
    makePage(page, "fr");
    makePage(page, "en");
}

function makePage(page, language) {
    var header = makeHeader(page, language);
    var content = makeContent(page, language);
    var footer = makeFooter(page, language);
    makeFile(header + content + footer);
}

function makeHeader(page, language) {
    return `<!DOCTYPE html>
    <html>
    <head>
        <meta charset="UTF-8">
        <title>Guillaume Pelletier-Auger - ${page[language].title}</title>
    </head>
    `;
}

function makeContent(page, language) {
    //h1 must be an anchor link to index.html in the chosen language.
    var nav1 = (language == "fr") ? "Travaux"  : "Works";
    var nav2 = (language == "fr") ? "À propos"  : "About";
    var nav3 = (language == "fr") ? "Français"  : "English";
    var oppositeLanguage = (language == "fr") ? "en"  : "fr";
    var content = `<body><h1>Guillaume Pelletier-Auger</h1>
    <div id="nav"><ul>
```

```
    <li>${nav1}</li>
    <li>${nav2}</li>
    <li>${nav3}</li>
    </ul>
    </div>
    <div id="main">${page[language].content}</div>
    `;
    return content;
}

function makeFooter(page, language) {
    return `<div id="footer"Guillaume Pelletier—Auger — 2016</div>
    </body>
    </html>
    `;
}

function makeFile(header, content, footer) {

}
```

7 Nouvelles choses à considérer

Mon système doit me permettre de créer des noms de fichiers ajustés à la langue. Sinon c'est très désagrable d'avoir un lien comme ça : *https ://www.pelletierauger.com/fr/about.html*.

C'est très laid ! Il faut donc que chaque item de ma base de données possède une nouvelle propriété : *filename*.

Alternativement, je pourrais simplement avoir une fonction qui prendrait la propriété *title* et la transformerait automatiquement en url valide, en transformant les espaces en traits d'union, et en transformant le reste en lettres minuscules. Je vais aussi devoir faire disparaître les accents français.

8 Nouvelles fonctionnalités

Je travaille présentement sur de nouvelles fonctionnalités, dont certaines utiliseront les modules *Highlight.js* et *MathJax-node*.

Choses à faire :

1. Créer un système qui me permet d'avoir une *url* différente pour les versions anglaises et françaises de mon portfolio. Comme ça je pourrais, par exemple, avoir des liens comme : https ://pelletierauger.github.io/growing-wreath.html ?lang=fr qui indiqueraient à mes programmes JavaScript quelle langue afficher.
2. Créer un système qui vérifie si une page contient du code (en utilisant une expression rationnelle) et, si c'est le cas, utilise le module *highlight.js* pour transformer ce code en colorant sa syntaxe.
3. Créer un système qui vérifie si une page contient des formules mathématiques LaTeX (en utilisant une expression rationnelle), et si c'est le cas, utilise le module *MathJax-node* pour transformer ces formules LaTeX en code HTML avec une feuille de style CSS.

9 Génération d'un blog

Je vais me créer un système qui génère un blog.

Choses à faire :

1. Lire le fichier *posts.js* et en extraire une liste de posts. Lire ensuite le dossier *posts* et en extraire les posts eux-mêmes. Bâtir une liste de contenu. Diviser le contenu selon la valeur *itemsPerPage*. Créer les pages.
2. Un système pour écrire et formater les dates. En français et en anglais.
3. Un fil RSS (pas du tout pressant).
4. Une pagination.
5. Formater le code, formater les mathématiques LaTeX.

Autres choses à faire :

1. Un générateur de *site map* et d'archives du blog.

10 Nouvelle refonte du site Web

Il y a beaucoup de nouvelles choses à accomplir pour faire de mon site Web un endroit grâce auquel je pourrais partager mon travail de manière optimale.

1. Refaire la page d'ouverture, la présentation des projets principaux, en utilisant le nouveau design réalisé récemment.
2. Écrire le contenu des projets.
3. Le plus gros casse-tête : pouvoir inclure des sketchs *p5.js* dans les pages de mes projets et dans les messages de mon blog.
4. Pouvoir inclure des mathématiques LaTeX avec *MathJax-node* dans les messages de mon blog.
5. Pouvoir inclure du code formatté avec *Highlight.js* dans les messages de mon blog.
6. Écrire une meilleure présentation dans la page *À propos*.
7. Créer une section *Archive* qui présenterait aussi une mappemonde de mes projets, une représentation visuelle de tous les projets que j'ai créés et des connexions entre eux. Il pourrait peut-être même y avoir une place pour les projets non réalisés, pour les *rêves*. Je pourrais définir chacun de mes projets selon certains mots-clés, et selon certaines relations qu'il entretient avec les autres projets, et créer un système où les projets s'organiseraient spatialement eux-mêmes. Ensuite, je dessinerais la position de chaque projet avec une ellipse, et ses connexions avec les autres projets par des lignes. La page *Archive* inclurait la mappemonde, une auto-documentation expliquant comment cette mappemonde a été réalisée, et finalement, au bas de la page, une liste visuellement très sobre de tous les projets. En plaçant le curseur au-dessus d'un point, on obtiendrait un *DIV* superposé à la mappemonde, qui présenterait sommairement le projet et donnerait un hyperlien. Les points devraient décider eux-mêmes de quelles couleurs ils veulent colorer leur propre partie de la mappemonde. Ainsi, la mappemonde pourrait changer complètement à chaque nouveau projet que j'y ajouterais. La visibilité des projets non réalisés pourrait être optionnelle. Les projets pourraient avoir des points plus petits ou plus gros selon la quantité de lignes de code qu'ils contiennent. Et une masse plus grande, aussi ! Si je calcule la masse dans mon système de particules, les gros points massifs se déplaceront plus lentement que les petits points agiles et légers. Ce sera beau.

11 Intégration de sketchs *p5.js* sur le site Web

C'est un bon casse-tête. L'élément le plus compliqué du casse-tête est la nécessité de pouvoir inclure des sketchs *p5.js* dans des notes de blogs, donc possiblement plusieurs sketchs dans le même fichier HTML. Évidemment que, conséquamment, tous les sketchs *p5.js* que je vais inclure sur mon site Web devront être programmés en *instance mode*. Ce n'est pas un très gros problème en soi. Le plus gros problème est de créer un environnement dans lequel je peux créer aisément mes sketchs et ensuite les déployer aussi aisément.

Je définis en ce moment une note de blog avec un paramètre nommé *date*. Je devrais aussi pouvoir donner à une note de blog ou à un projet un paramètre optionnel nommé *p5sketch*, qui serait une liste de fichiers à inclure sur le site Web. Ces fichiers doivent être inclus dans un dossier qui porte le même nom que la page en question, ou la note de blog en question. *maker.js* doit ensuite copier chacun de ces fichiers dans un dossier nommé de la même façon, mais dans l'arborescence du site Web.

```
exports.date = {
    year : 2017,
    month : 5,
    day : 23
};

// Important : Dans la liste ci—dessous, les fichiers doivent apparaître
// dans l'ordre dans lequel ils doivent être chargés,
// et sketch.js doit toujours être le dernier fichier chargé.

exports.p5sketch = {
    p5DOM : false,
    files : ["particle—prototype.js", "particles.js", "sketch.js"]
};
```

Ainsi, lorsque *maker.js* détecte qu'une page qu'il essaie de créer contient un sketch *p5.js*, il doit :

1. Lire la propriété p5sketch de cette page ou de cette note de blog.
2. Lire le dossier qui porte le même nom que l'identifiant de la page ou de la note de blog.
3. Créer un dossier qui porte le même nom aux bons endroits de l'arborescence du site, en français et en anglais.
4. À l'extérieur de ce nouveau dossier, donc au même endroit où le fichier HTML de la page ou de la note de blog sera situé, créer un fichier vide nommé *sketch.js*.
5. Copier le contenu du fichier original sketch.js *à la suite du nouveau fichier sketch.js*.
6. Copier les fichiers du premier dossier vers le deuxième dossier, excepté pour sketch.js.
7. Créer la section *head* du fichier HTML avec des liens appropriés vers tous les fichiers JavaScript, sketch.js et les autres.

12 Mise en place des sketchs *p5.js* dans les pages qui les contiennent

Je dois aussi trouver la meilleure façon de mettre en place les sketchs à l'intérieur des pages et des notes de blog. Je crois que je vais faire ça en créant des *DIV* bien nommés dans mes pages et notes elles-mêmes, et ensuite en utilisant la fonction *.parent* de *p5.DOM* pour envoyer mon canevas HTML5 dans le bon *DIV*. Si j'utilise cette méthode, ça veut donc dire que tous les sketchs *p5.js* auront besoin de la bibliothèque *p5.DOM*.

13 Idée pour une grande simplification

Minute... je n'ai pas besoin de copier des fichiers et des dossiers... tout peut se passer dans le dossier nommé *sketches* qui contient tous les sketchs, et même, chacun des sketchs peut avoir son propre fichier *sketch.js*, puisque chaque sketch sera dans son propre dossier. Et le blog n'aura qu'à charger tous les fichiers dans tous les bons sous-dossiers du dossier *sketches*. La seule chose importante sera qu'effectivement, tous les sketchs devront être programmés en *instance mode*.

Je n'ai qu'à envoyer un *array* à ma fonction *makeHeader()* :

```
function makeHeader(page, language, stepsFromRoot, sketches) {

    var prefix = "";
    if (stepsFromRoot == 0) {
        prefix = "./";
    } else if (stepsFromRoot > 0) {
        for (var i = 0; i < stepsFromRoot; i++) {
            prefix += "../";
        }
    }

    var scripts = ``;
    if (sketches) {
        scripts += `<script src="${prefix}libraries/p5.js" type="text/javascript"></script>
        `;
        scripts += `<script src="${prefix}libraries/p5.dom.js" type="text/javascript"></script>
        `;
        for (let i = 0; i < sketches.length; i++) {
            scripts += `<script src="${prefix}`;
            scripts += sketches[i];
            scripts +=`" type="text/javascript"></script>
            `;
        }
    }

    //...

    return `<!DOCTYPE html>
    <html>
    <head>
        <meta charset="UTF-8">
        <title>Guillaume Pelletier-Auger${title}</title>
        <meta name="viewport"
        content="width=device-width, initial-scale=1,
        maximum-scale=4, user-scalable=yes" />
        ${scripts}
        <link href="${prefix}style.css" rel="stylesheet" type="text/css">
        <link href="https ://fonts.googleapis.com/css ?family=${codeFont}Sorts+Mill+Goudy :400,400i"
         rel="stylesheet">
        ${codeCSS}
    </head>`;
}
```