Outils, méthodes et environnement de développement

Olivier Dommange
2018

Sommaire

Créer une page Web n'est pas très compliqué. Il suffit de connaître HTML. En le combinant avec d'autres langages comme CSS et Javascript, cela permet d'appliquer un formatage aux composants, de les animer et les rendre interactifs. La pratique demeure la même quel que soit la complexité du développement.

Le développeur a toutefois intérêt à augmenter sa productivité en limitant notamment les opérations répétitives ou encore la réécriture de codes au fil des projets. Il est également préoccupé de développer des applications performantes, sécurisées et adaptées à toute une gamme de navigateurs. Cette lourde responsabilité implique de réserver un temps considérable et précieux aux tests et débogage.

Des méthodes de travail et plus particulièrement certains outils permettent d'accompagner le processus de réalisation en aidant le développeur à augmenter l'efficacité de son travail. Ces outils peuvent être classés selon deux catégories en fonction de leur utilité dans la production :

Orienté code :

- Ceux qui **facilitent le développement de fonctionnalités** qui nécessitent la rédaction de nombreuses lignes de codes. Il s'agit de librairies prêtes à l'emploi. La librairie, jQuery codé en Javascript qui propose notamment des fonctions d'animation ou des requêtes au serveur en mode asynchrone compatibles avec tous les navigateurs récents en est un exemple.
- Il peut également s'agir d'un ensemble d'outils **réunis pour faciliter l'intégration et rendre plus performantes certaines opérations**. Ils sont également appelés « Frameworks ». C'est le cas de Bootstrap qui offre un formatage CSS prêt à l'emploi utile aux interfaces tout en donnant la possibilité d'intégrer l'animation de composants grâce à jQuery.

Orienté production :

- Il peut s'agir d'**augmenter la performance et optimiser certaines tâches utiles au développement**. Node.js propose la création d'un serveur web dédié et offre l'accès à des librairies permettant d'augmenter la performance du développeur, notamment avec gulp, yeoman ou grunt.
- Il peut également s'agir de **langages préformatées générant du code Javascript ou CSS** propre et compatibles sur les principaux navigateurs, performant et facile à maintenir. Les langages TypeScript ou JSX s'insèrent dans cette catégorie en générant du Javascript. LESS ou SASS génèrent eux du CSS.
- Il peut encore s'agir d'outils qui **augmentent la modularité du code** en optimisant la génération d'interfaces et l'intégration dynamique des contenus. React.js et Angular.js offre une solution dans ce sens.

Cette documentation présente des outils qui optimisent la production en augmentant en efficacité tout en améliorant le code et sa modularité. Bien que JSON et Javascript soient présentés, une bonne connaissance des principes du développement est utile pour aborder et pleinement profiter des explications qui suivent.

Les outils et concepts

De nouveaux outils, langages et librairies ont vu leur apparition depuis quelques années et tendent à transformer les méthodes de production. Nombre d'entre eux sont déclinés du langage Javascript et du format de transfert de données JSON. Il est donc très utile, sinon indispensable, de connaître ce langage et ce format pour envisager utiliser ces outils dérivés.

Node.js, par exemple, est un langage qui permet notamment de concevoir un serveur Web. Il est également très utile pour gérer des librairies de codes dans un projet. Node.js utilise Javascript et implique l'utilisation de JSON pour le paramétrage des librairies affectées à un projet.

S'ajoute à cela, l'utilisation d'outils, tels que l'invite de commandes. Cette méthode d'appel d'instructions est très utilisée avec Linux et offre une solution optimale et pratique dans plusieurs conditions. C'est le cas notamment pour le téléchargement de paquets de fichiers destinés à un projet ou pour l'exécution de commandes simples.

Ces outils augmentent la productivité en simplifiant les opérations effectuées de manière répétitives ou qui s'avèrent autrement quasi impossible, comme la minification (suppression des espaces et commentaires afin de réduire le poids du fichier) du code ou la transformation (compilation) d'une librairie telle que JSX ou TypeScript en fichier Javascript ou encore SASS ou LESS en fichier CSS.

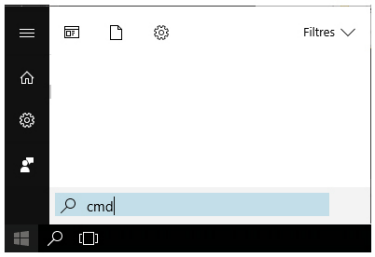
Ces méthodes et pratiques font dorénavant partis de la palette d'outils de développement et de production du développeur.

L'invite de commandes

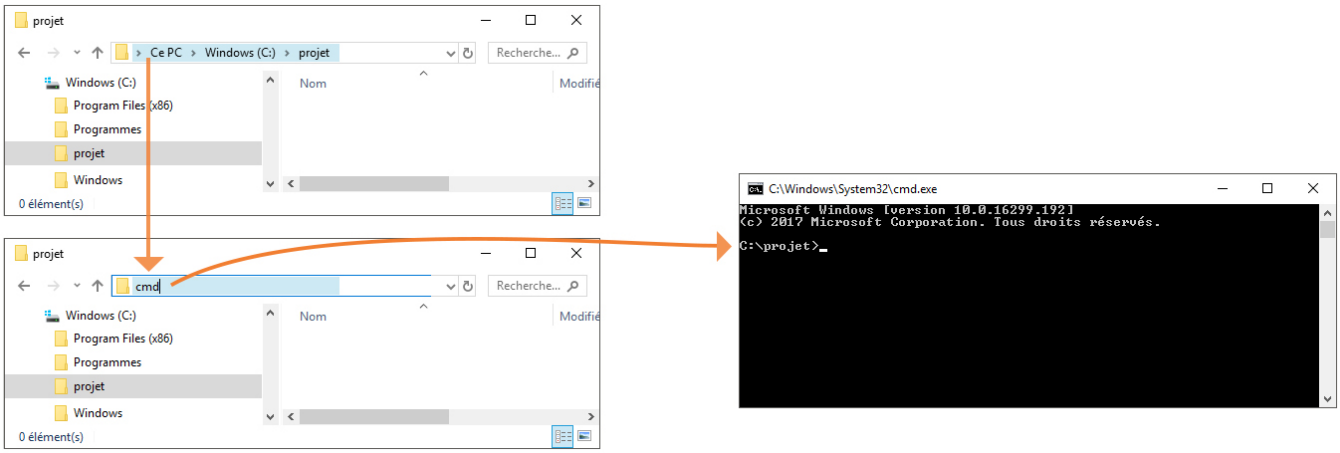
L’invite de commandes est très utilisée et pratique dans le cadre des déploiements et mise à jour des outils et librairies disposés dans des projets Web. Node.js, et les plateformes de développement qui s'y réfèrent, utilisent l'invite de commandes pour l'exécution d'opérations.

Accéder à l'invite de commandes de Windows

L'invite de commandes est un programme qui peut être lancé en indiquant « cmd » après avoir cliqué sur le bouton *« Démarrer »* de Windows (en bas à gauche de l'écran ou au clavier).



Une autre méthode, plus pratique, est d'indiquer « cmd » depuis une fenêtre de l'explorateur Windows. Cette méthode à l'avantage d'indiquer le chemin du répertoire dans l'invite de commandes et ainsi d'exécuter des opérations depuis cet emplacement.



Node.js

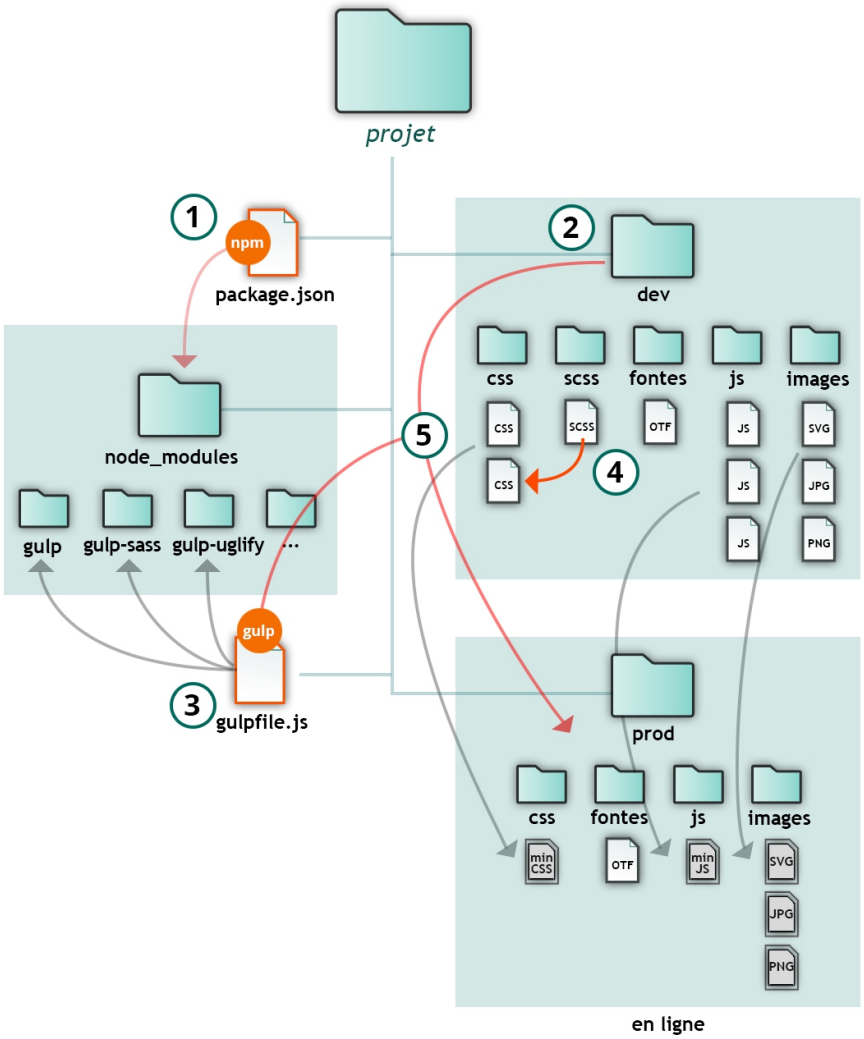
Node.js met à disposition un environnement de développement. Il contient notamment une bibliothèque de **serveur http intégrée** permettant de faire fonctionner un serveur Web comme le fait Apache. Il permet d’exécuter du Javascript sur ce serveur et d'actualiser le chargement des pages au fil des changements opérés.

Node.js s’installe comme un logiciel en téléchargeant l’outil depuis le site nodejs.org. Une fois installée, les outils de la librairie node.js sont accessibles depuis l’invite de commandes. Il est notamment possible d'intégrer à un projet des fonctionnalités aidant à la production en utilisant le gestionnaire de paquets (« Node Package Manager » - NPM) qu'intègre node.js (pour découvrir les commandes npm consulter npmjs.com). NPM per-

met de « lancer » des opérations au sein même du système d'exploitation et permet de gérer des opérations telles que l'installation (téléchargement et déploiement) de libraires Javascript telles ou administrer une série d'opérations sur des fichiers et leur contenu depuis l'invite de commandes.

Le projet

Pour une gestion efficace du projet, il est utile d'isoler l'espace de développement des fichiers qui seront utiles à la production (mise en ligne). L'organisation d'un projet sépare donc les fichiers utiles au développement du site de ceux qui sont optimisés par les outils (packages) de node.js avant d'être publiés. A ce titre, la librairie gulp est utilisé pour la mise en fonction de cette gestion de fichiers. Il dispose notamment des commandes destinées à l'optimisation des fichiers et de la fusion des contenus de ceux-ci.



[package.json] Gestionnaire des paquets archivés dans le dossier node_modules et utiles à la gestion et l'organisation du développement.

[dev] Espace de développement. En opposition au répertoire **[prod]** qui contient les fichiers optimisés destinés au produit publié.

[gulpfile.js] Scripts d'opérations destinés à l'optimisation du projet et de ses composants.

Processus que peut effectuer gulp par l'entremise de **[gulpfile.js]**. En occurrence, transformer le SASS en CSS.

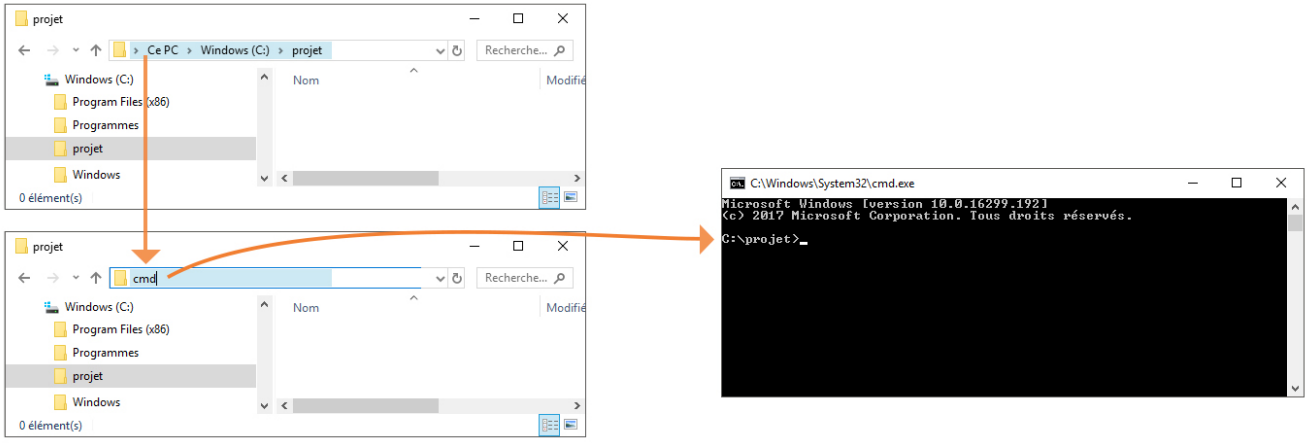
Optimisation (fusion des fichiers et minification du code) des fichiers effectué par **[gulpfile.js]** et transfert du répertoire **[dev]** au dossier **[prod]**.

Déploiement de l'environnement

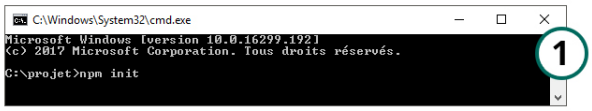
La mise en place de cet environnement se fait à l'aide de l'invite de commandes mis en fonction dans le dossier du projet.

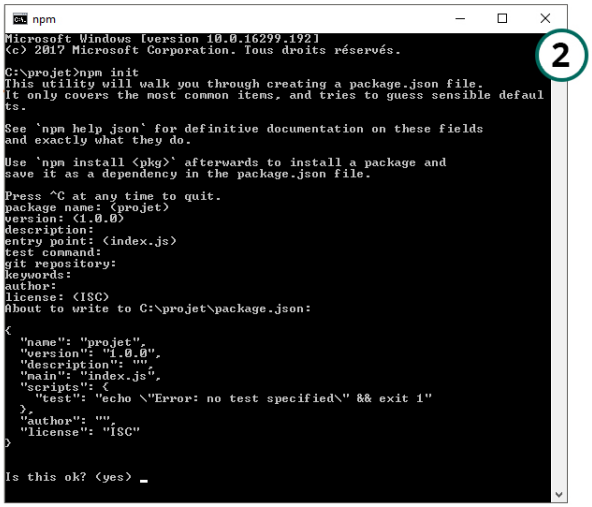
1. Initialisation

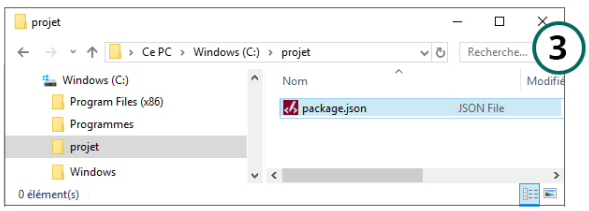
Dans le répertoire du projet, lancer l'invite de commandes.



2. Générer le gestionnaire de paquets (package.json)







1. Depuis l'invite de commandes engager l'initialisation du projet avec la commande npm init.

Invite de commandes

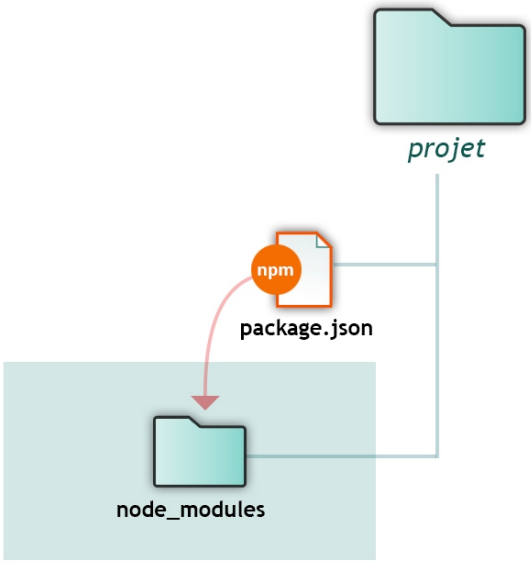
```
npm init
```

2. Pendant cette opération sont demandées les informations qui concernent le projet à initier (nom du paquet, description, version, auteur,...). Ces informations ne sont pas obligatoires et peuvent être rééditées plus en amont dans le fichier package.json qui sera généré.

3. Génération du fichier package.json.

3. **Installer un paquet**

Déployé depuis l'invite de commandes les ressources à utiliser. Exemple avec gulp.



Invite de commandes

```
npm install gulp -g
```

Cette commande installe l'environnement de gulp dans le système d'exploitation (ça n'ajoute pas de documents dans le répertoire du projet).

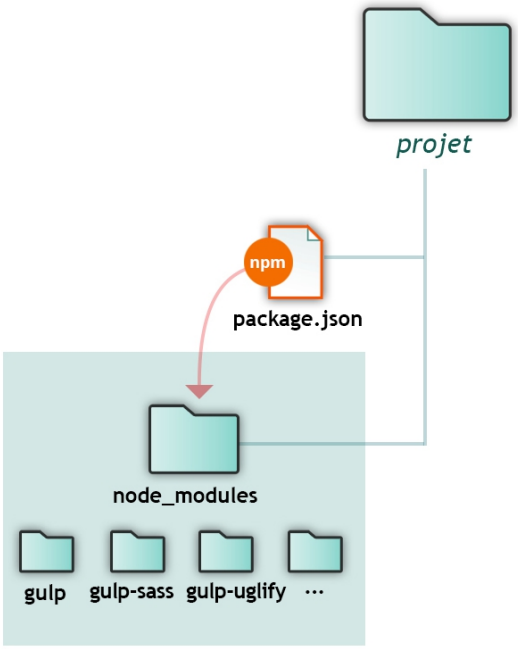
Invite de commandes

```
npm install gulp --save-dev
```

Cette commande crée le répertoire **node_modules** (parce qu'il n'existe pas encore) qui contiendra les ressources de gulp. La mention –dev inscrit dans le fichier package.json la dépendance de la ressource (gulp) installée. Ce qui permettra de le redéployer dans un nouvel environnement au besoin.

4. **Installer une librairie de ressources**

Des ressources utiles pour des tâches plus spécifiques.



Invite de commandes

```
npm install gulp-sass --save-dev
```

Cette commande installe la ressource supplémentaire. En occurrence, un processeur SASS provenant des ressources **gulp**.

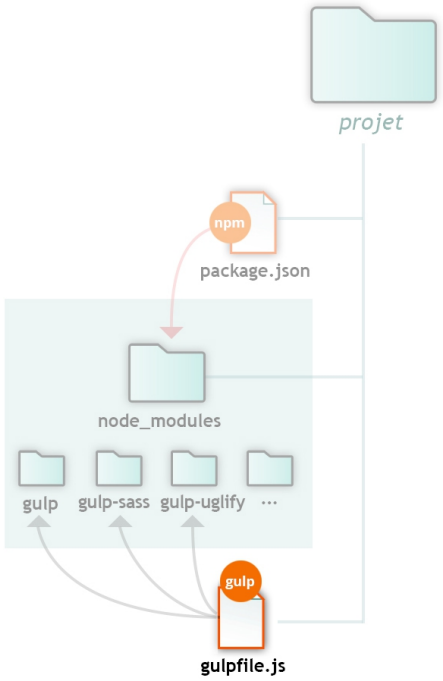
D'autres ressources sont disponibles et s'installent selon le même processus.Exemple avec gulp-uglify qui est utile à la minification du code Javascript.

Invite de commandes

```
npm install gulp-uglify --save-dev
```

Utiliser gulpfile.js

Gulp.js dispose de fonctions destinées à la gestion et l'optimisation de fichiers et leur contenu. Elles permettent par l'entremise de sous-librairies de mener certaines opérations ciblées du projet. La liste des sous-librairies est disponible sur le site https://gulpjs.com/plugins



Les composants de base de gulp se gèrent depuis le fichier **gulpfile.js** que doit disposer le projet pour mener ces opérations. Ce fichier est à disposer à la racine du projet aux côtés du **package.json** car, rappelons-le, ce dernier sert à l'installation des sous-librairies dont fera référence et usage gulp.

gulpfile.js

```
1  var gulp = require('gulp');
2  var sass = require('gulp-sass');
```

Fonctionnement de gulp

Gulp gère des tâches par la méthode **task()** qui permet de donner un nom à la tâche et d'exécuter une ou des actions provenant de sous-librairies gulp par l'entremise d'une fonction anonyme comme présenté dans l'exemple suivant. *Chaque tâche gulp s'établit sur ce modèle.*

gulpfile.js

```
..  […]
4   gulp.task('sass', function(){                    // Initialisation de la tâche
5         return gulp.src('dev/scss/styles.scss')    // sur un fichier ciblé
6                .pipe(sass())                       // à qui est imposé ce traitement
7                .pipe(gulp.dest('dev/css'))         // puis sauvegardé dans une
                                                        nouvelle version à cet endroit
8   });
```

Le ou les fichiers à transformer sont identifiés par l'entremise de la méthode **src()** en indiquant le chemin vers d'accès aux fichiers. La méthode src() peut accéder à plusieurs fichiers à l'aide du symbole astérisque (*). Par exemple, 'scss/*.scss'.

Les actions sont ensuite menées par la méthode **pipe()** et se concluent par la sauvegarde d'une nouvelle version de ces fichiers à un lieu établi par la méthode **dest()**.

L'exécution d'une tâche gulp

Les tâches définies avec gulp s'exécutent depuis l'invite de commandes en faisant appel au nom attribuée à cette tâche dans le fichier **gulpfile.js**.

Dans l'exemple suivant, sont liées les ressources (gulp et gulp-sass) et sert à convertir le code SASS en CSS. La tâche, nommée « sass » est définie pour mener à bien cette opération.

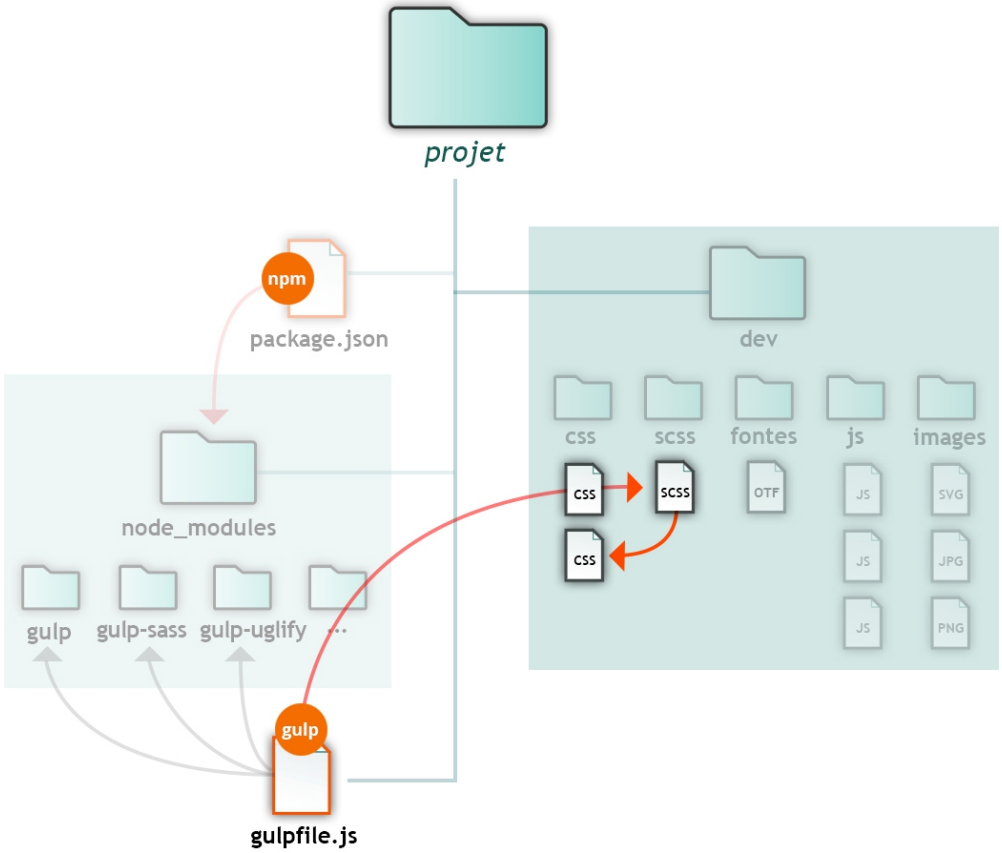
L'exécution de cette tâche se fait dans l'invite de commande en indiquant bien qu'il s'agisse d'une commande gulp.

Invite de commandes

```
gulp sass
```

gulpfile.js

```
var gulp = require('gulp');
var sass = require('gulp-sass');

gulp.task('sass', function(){
      return
gulp.src('dev/scss/styles.scss')
            .pipe(sass())
            .pipe(gulp.dest('dev/css'));
});
```



Le déploiement en production

Les tâches gulp permettent de préparer et optimiser les fichiers pour une mise en ligne.

Dans l'exemple suivant, est conçu une tâche « prod » qui utilise la ressource gulp-uglify (qui optimisera les fichiers Javascript) et fait appel à la tâche « sass » créée précédemment par la fonction *start()*.

Sont rapatriés dans un dossier « prod » les fichiers devant être mis en ligne. A travers cette démarche gulp crée les dossiers et insère les fichiers. Lors qu'un fichier existe, il le met à jour.

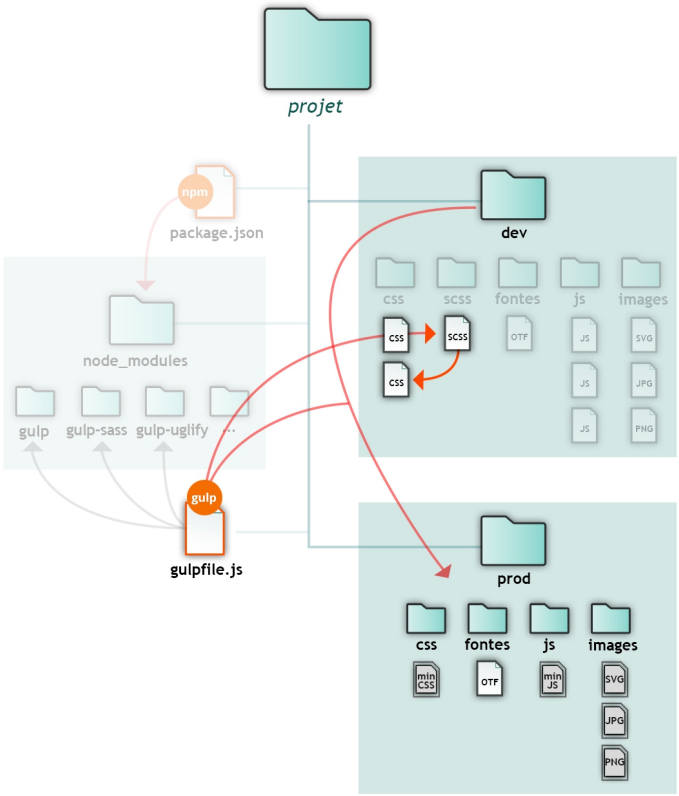
L'exécution de cette tâche se fait dans l'invite de commande.

gulpfile.js

```
..  [...]
3   var uglify = require('gulp-uglify');
..
16  gulp.task('prod', function(){
17          gulp.src('dev/js/*.js')
18              .pipe(uglify())
19              .pipe(gulp.dest('prod/js'));
20          gulp.start('sass')
21              .pipe(gulp.dest('prod/css'));
22          gulp.src('dev/scss/*.css')
23              .pipe(gulp.dest('prod/css'));
24          gulp.src('dev/images/*.css')
25              .pipe(gulp.dest('prod/images'));
26          gulp.src('dev/fonts/*.css')
27              .pipe(gulp.dest('prod/fonts'));
28          gulp.src('dev/*.html')
29              .pipe(gulp.dest('prod'));
30  });
```

Invite de commandes

```
gulp prod
```



package.json

Le fichier **package.json**, est au cœur de la synchronisation des modules à télécharger via les commandes **npm** dans l'invite de commandes. Ce fichier dispose des informations concernant les ressources, telles que leur version ou dépendances (autres modules nécessaires pour qu'ils fonctionnent). La gestion des ressources d'un projet se fait par le fichier **package.json**. Il permet la mise en place de tout un projet par l'exécution d'une seule commande (*init*).

Invite de commandes

```
npm init
```

Les composants du fichier package.json

Le fichier package.json comporte des informations générales, telles que le nom du projet, la version du package, une description, le nom de l'auteur ou encore la licence.

package.json

```
{
  "name": "projet ",
  "version": "1.0.0",
  "description": "Ce package utilise : gulp et jQuery",
  "scripts": {
  },
  "license": "MIT",
  "dependencies": {
    "gulp": "^3.9.1",
    "gulp-sass": "^3.1.0",
    "jquery": "^3.2.1"
  }
}
```

Des dépendances (*'dependencies'*) indiquent les ressources à installer dans le projet dans le dossier node_modules et les éventuelles versions de mise à jour. Les symboles suivants sont utiles pour cette spécification :

- **version** : Cette version uniquement
- **>version** : Cette version ou supérieur
- **>=version** : Cette version égale ou supérieur
- **<version** : Cette version ou intérieur
- **<=version** : Cette version égale ou supérieur
- **~version** : Version à peu près équivalente
- **^version** : Compatible avec cette version
- **1.2.x** : Version 1.2 et ses déclinaisons
- ***** : Quelque version que ce soit

Les scripts du package.json

Le fichier **package.json**, permet d'intégrer des scripts prêts à être exécuter dans l'invite de commande et ainsi mener une série d'opérations sur un projet.

package.json

```
1   {
2     "name": "projet web",
3     "version": "1.0.0",
4     "description": "Ce package gère un projet web",
5     "scripts": {
6         "test": "echo 'Ceci est un test'"
7     },
8     "license": "MIT",
9     "dependencies": {
..        [...]
10    }
11  }
```

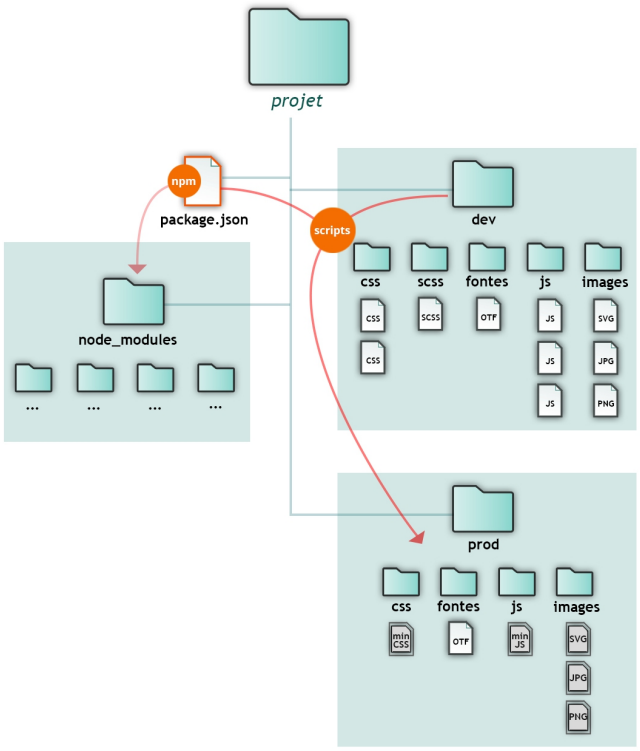
La fonction *run* permet d'exécuter un script depuis l'invite de commandes. Comme l'illustre l'exemple suivant :

Invite de commandes

```
npm run test
>echo 'Ceci est un test'
Ceci est un test
```

Il est ainsi possible d'initier des opérations en série à partir de l'intitulé d'un script. Ces opérations peuvent mener les mêmes opérations qui s'établissent avec les librairies gulp ou grunt et leurs sous-librairies.

L'avantage d'une librairie tels que gulp ou grunt est qu'il permet de gérer plus librement les tâches à mener par l'entremise d'un code Javascript gérant ces opérations.



Exécuter plusieurs scripts simultanément

Chaque script indiqué dans le fichier **package.json** peut être exécuté l'un après l'autre. Cela peut s'avérer plus ou moins long à exécuter sans compter qu'il y a un risque d'oublier l'exécution d'un des scripts.

La librairie « *npm-run-all* » permet d'exécuter une série d'opérations en les réunissant dans un groupe et en leur attribuant une tâche chacune.

package.json

```
..  [...]
5     "scripts": {
6         "start": "npm-run-all dev:*",
7         "dev:sass-dev": "...",
8         "dev:js-minified": "..."
9     },
10    "dependencies": {
11      "npm-run-all": "^2.3.0",
12      "node-sass": "^3.8.0",
13      "uglifyjs": "^2.4.10"
14    }
```

Dans le **package.json** qui précède la librairie ***npm-run-all*** permet d'exécuter plusieurs scripts en même temps. Ces scripts sont associés par la relation qui existent entre eux grâce au préfixe ***dev:*** qui leur a été attribué. Ainsi, les scripts sass-dev et js-minified seront exécutés en même temps. Ceux-ci convertirons le code SASS et minifier le code Javsacript de certains fichiers.

Les librairies indiquées dans les scripts doivent être installées en indiquant leur dépendance (*dependencies*).

La fonction *run* permet d'exécuter un script en rappelant le nom du script associé. En occurrence, *start* dans l'exemple qui précède :

Invite de commandes

```
npm run start
```

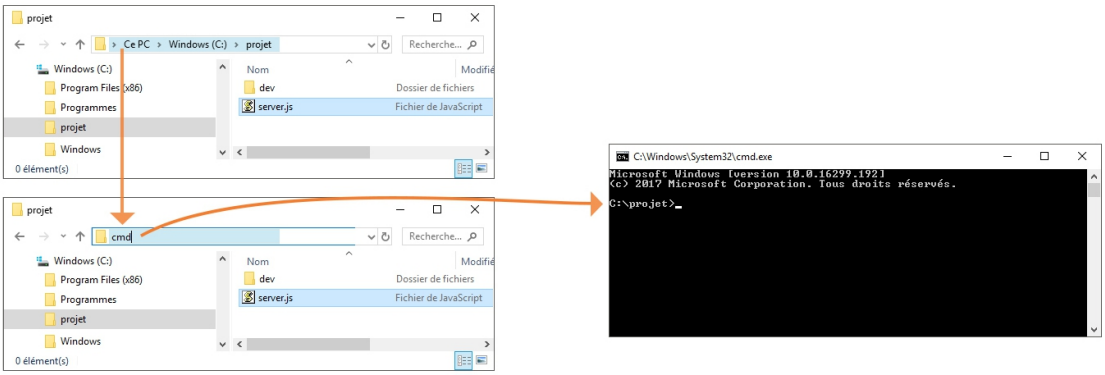
Serveur : Node.js

Une fois installée sur un système, l'application node.js est en mesure d'interpréter les codes d'une application pour lui permettre d'exécuter des opérations ciblées grâce notamment aux paquets NPM. Node.js offre également la possibilité de concevoir un serveur Web qui exécute des opérations, telles que le rafraîchissement du navigateur en temps réel.

Un des avantages qu'offre la gestion d'un serveur est qu'on puisse gérer les opérations effectuées par celui-ci. Ces opérations sont initiées par des événements qui peuvent être déclenchés à la chaîne, en parallèle ou en dépendance.

Démarrer le serveur

Le lancement d'un serveur node.js se fait par l'invite de commandes Windows. L'invite de commandes peut s'ouvrir sur Windows depuis la fenêtre d'un dossier en cliquant dans la barre de navigation et en y indiquant **'cmd'**.



Un fichier **server.js** sert à la génération et au lancement du serveur en référence à la librairie ***http*** de node.js. Cette librairie gère les interactions et échanges de données qui s'engagent entre le navigateur et le serveur par l'entremise du protocole http.

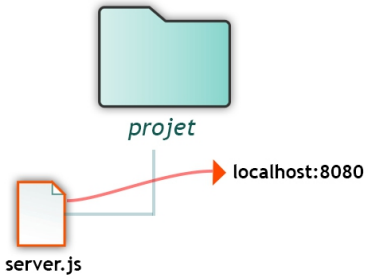
server.js

```
1  var http = require('http');
2
3  var server = http.createServer(function(req, res) {
4    res.writeHead(200, { 'Content-Type' : 'text/plain' });
5    res.end('Je suis un serveur node.js');
6  });
7  server.listen(8080, 'localhost');
```

Comme l'illustre l'exemple précédent, l'objet http permet de générer le serveur par la méthode createServer(). Celle-ci contient les informations que générera la réponse à transmettre au navigateur par le protocole http, qui est composé d'une entête indiquant le statut (200) ainsi que le type de contenu (content-type) par l'entremise de la méthode writeHead().

Le contenu transmis au navigateur lest quant à lui par l'entremise de la méthode end().

Le lancement du serveur (et donc l'appel à la méthode createServer() se fera au moment où le port 8080 est accédé via le serveur local (locahost) comme précisé dans la méthode listen(). Le port peut être différent de celui-ci, du moment où il n'est pas utilisé à d'autres fins par un logiciel ou d'autres outils du système.

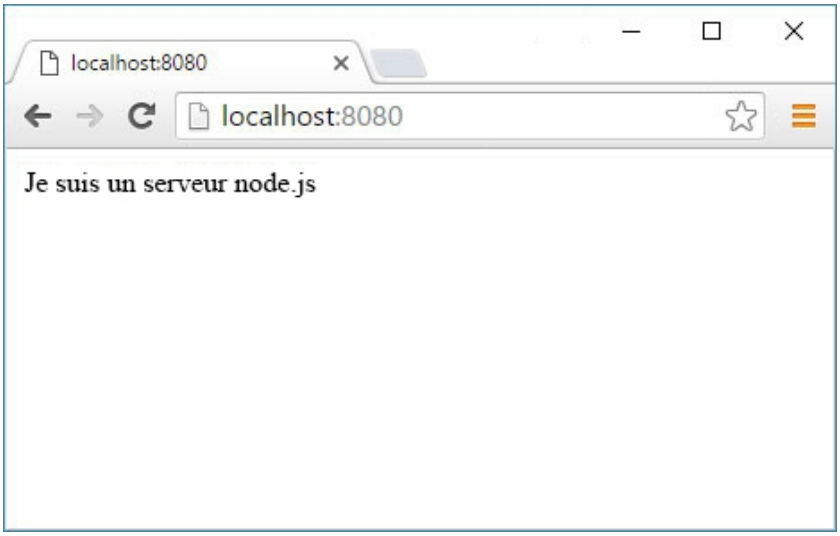


L'exécution des opérations indiquées dans le fichier **server.js** se fait depuis l'invite de commandes avec l'instruction « node » et le nom du fichier à exécuter. Ce qui donne **node server**.

Invite de commandes

```
node server
```

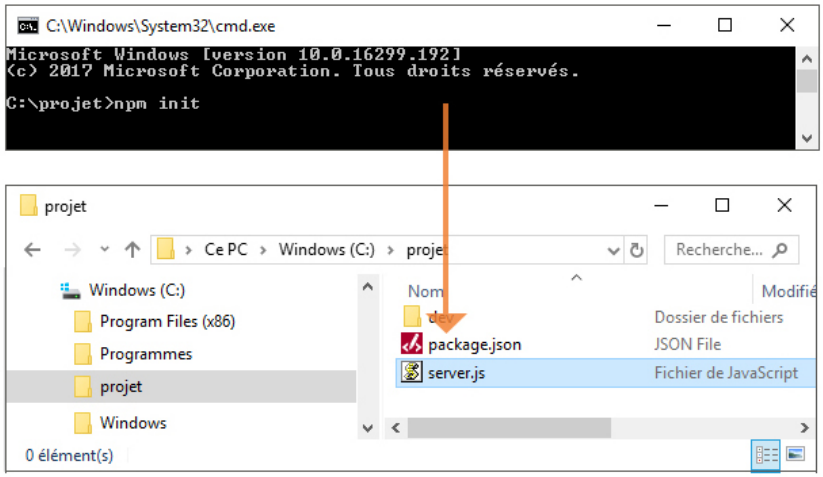
Résultat : Dans le navigateur à l'adresse localhost:8080.



La librairie express

Node.js dispose de fonctionnalités de mise en marche d'un serveur mais il s'avère quelque peu laborieux et le sera davantage lorsqu'il s'agira d'utiliser des outils de gestion données avec des outils tels que MongoDB ou la gestion de « sockets ». La librairie Express.js propose une solution dynamique et performante en synchronisme avec ces outils et dans les échanges d'information entre le navigateur et le serveur.

Le déploiement de la librairie *Express* implique l'utilisation un installation de cette librairie ce qui nécessite la création du fichier package.json qui sera généré par l'instruction npm init.



L'installation de express se fait ensuite par l'appel de l'instruction **npm install express --save-dev** dans l'invite de commandes.

Invite de commandes

```
npm install express --save-dev
```

Le fichier « server.js » utilisera cette librairie en y faisant référence par son inclusion dans le fichier avec la fonction require(). La méthode get() récupère les valeurs transmises par le navigateur et retourne un contenu par la méthode send(). Enfin la connexion entre le serveur et le navigateur sur un port 8080 est établi par la méthode listen().

server.js

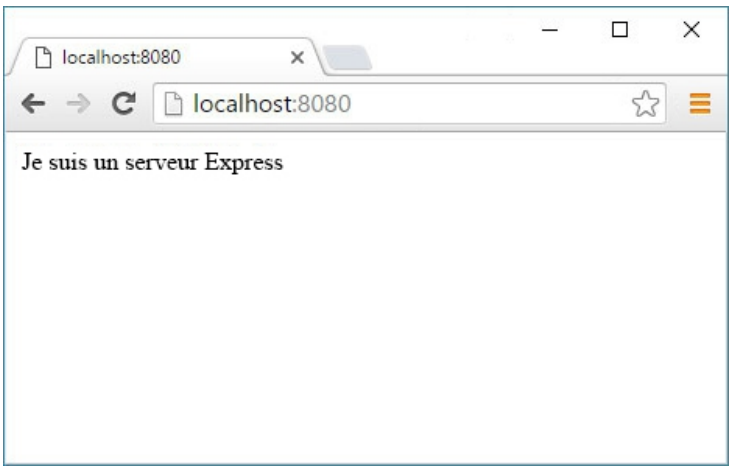
```
1  var express = require('express');
2  var app = express();
3
4  app.get('/*', function(req, res) {
5    res.send('Je suis un serveur Express');
6  });
7
8  app.listen(8080), function () {
9    console.log('Serveur démarré au port : 8080');
10 });
```

L'exécution des opérations indiquées dans le fichier **server.js** se fait depuis l'invite de commandes avec l'instruction **node server**.

Invite de commandes

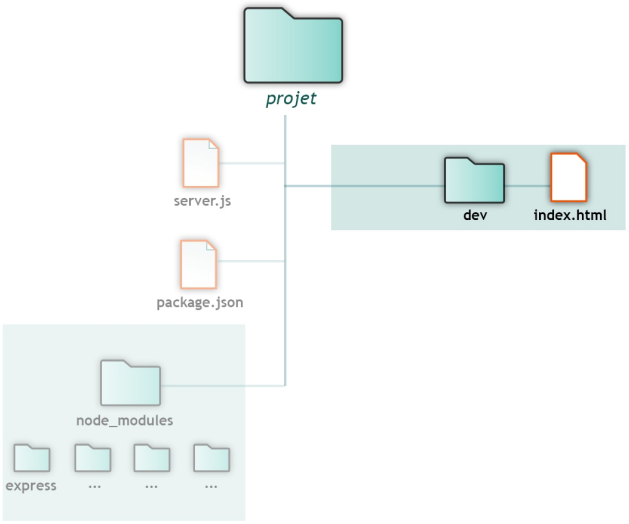
```
node server
```

Résultat : Dans le navigateur à l'adresse localhost:8080.



Définir un fichier racine

Faire appel au serveur depuis le navigateur à l'adresse localhost:8080 est dorénavant possible. Il est toutefois utile de prévoir un fichier qui sera celui consulté lors de l'accès du serveur. Dans l'exemple qui suit, il s'agira d'un fichier « index.html » qui se trouvera dans le répertoire « dev ».



Créer le fichier « index.html » dans le répertoire « dev » de base de l'application.

index.html

```
<!doctype html>
<html>
  <head>

  </head>
  <body>
    Serveur Web node.js disposant de
contenus HTML.
  </body>
</html>
```

Dans le fichier « server.js » s'ajoute les indications qui permettent de localiser le répertoire racine avec la méthode use() puis l'envoi du fichier « index.html » au serveur par la méthode sendFile(). Celui-ci s'insère dans une fonction anonyme qui transmet les valeurs récupérées en get par la méthode get() au fichier « index.html ». Celle-ci est transmise à la méthode listen() et s'exécute au lancement du serveur.

server.js

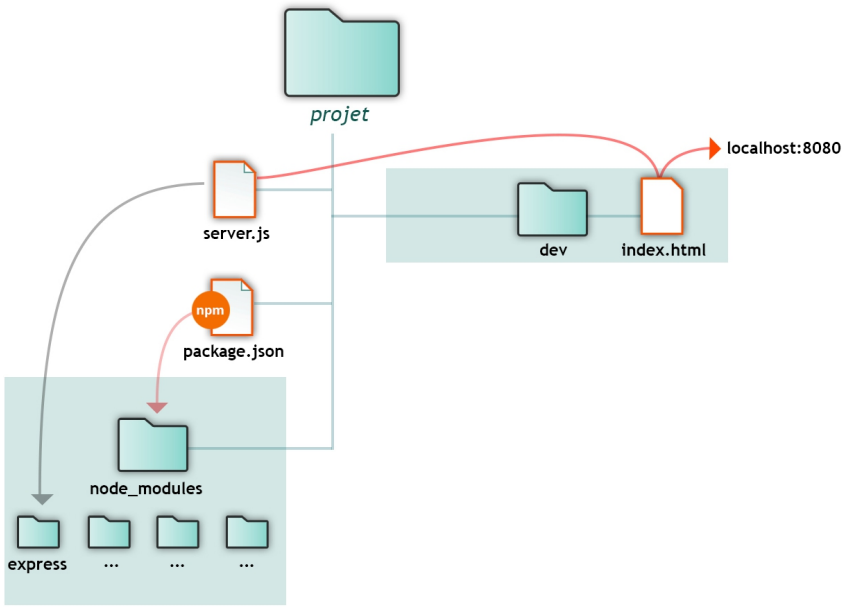
```
..  ...
6   app.use('/dev', express.static(__dirname + '/dev'));
7
8   app.get('/*', function(req, res) {
9     res.sendFile(__dirname + '/dev/index.html');
10  });
11
12  app.listen(8080, function () {
13    console.log('Serveur démarré au port : 8080);
14  });
```

L'exécution des opérations indiquées dans le fichier **server.js** se fait depuis l'invite de commandes avec l'instruction **node server**.

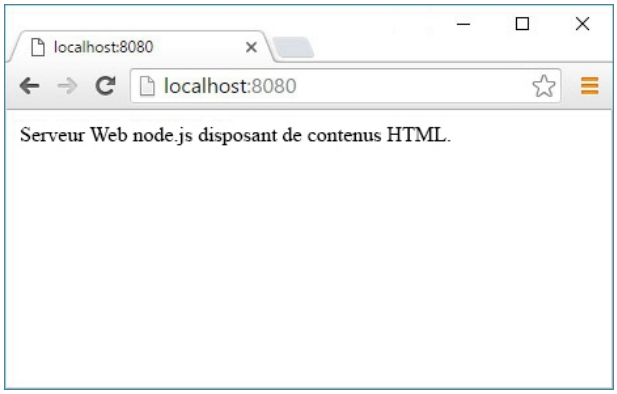
Invite de commandes

```
node server
```

Dorénavant sera transmis au navigateur le fichier « index.html » à l'appel de l'adresse localhost:8080.

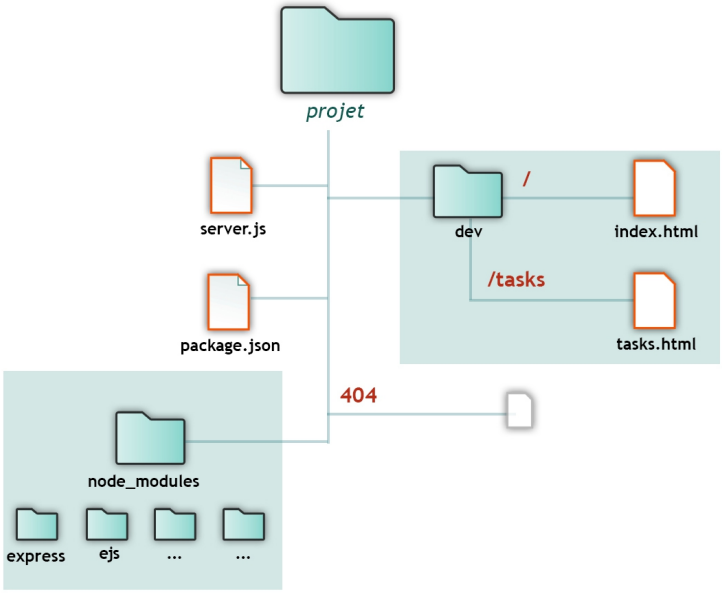


Résultat : Dans le navigateur à l'adresse localhost:8080.



Les routeurs

Circuler dans les pages d'une application peut s'établir par l'entremise de la définition de lieux d'accès prévus par le serveur. Dans cette optique, sont prévus une URL pour chaque page scénario envisagé. Il s'agit de routeurs qui sont repérés dans le code par les valeurs transmises par la méthode get du protocole http.



dev/index.html

Dans le cas de cet exemple, le fichier « index.html » dispose d'un lien vers le routeur tasks.

```
1   <!doctype html>
2   <html>
3     <head>
4           <title>App node et Express</title>
5     </head>
6     <body>
7           <h1>Gestionnaire de tâches</h1>
8           <ul>
9               <li><a href="tasks">Tâches</a></li>
10          </ul>
11    </body>
12  </html>
```

dev/tasks.html

Le fichier « tasks.html » indique dans une liste statique deux tâches.

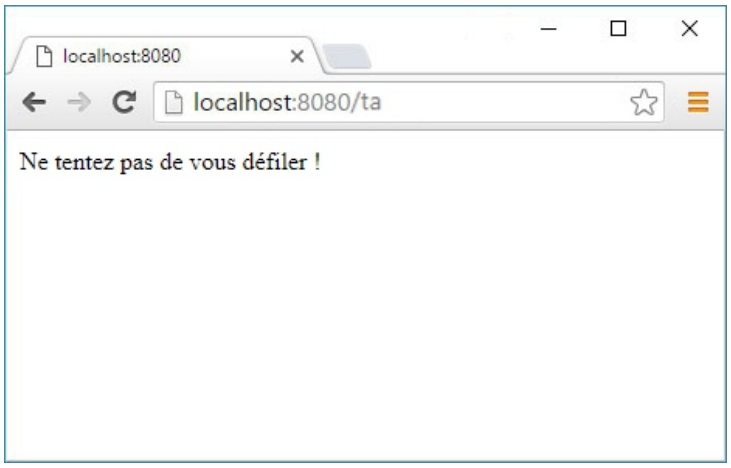
```
<!doctype html>
<html>
  <head>
        <title>App node et Express</title>
  </head>
  <body>
        <h1>2 tâche(s) prévue(s)</h1>
        <ul>
            <li>Tâche n°1</li>
            <li>Tâche n°2</li>
        </ul>
  </body>
</html>
```

Dans le fichier « server.js » sont organisés les routeurs par la méthode get pour chacune des pages. Dans l'évetualité où un utilisateur accède à une page qui n'existe pas, une réponse http disposant du statut 404 sera transmis par la méthode use(). La vérification de cet accès n'est à prévoir qu'à la fin du processus de vérification des routeurs.

server.js

```
var express = require('express');
var app = express();

app.use('/dev', express.static(__dirname + '/dev'));

app.get('/*', function(req, res) {
  res.sendFile(__dirname + '/dev/index.html');
});

app.get('/tasks', function(req, res) {
  res.sendFile(__dirname + '/dev/tasks.html');
});

app.use(function(req, res, next) {
  res.status(404).send('Ne tentez pas de vous défiler !');;
});

app.listen(8080, function () {
  console.log('Serveur démarré au port : 8080);
});
```

Dans le cas présenté la page 404 sera visible dans le cas où l'utilisateur n'accède pas à la page d'accueil ou la page tasks.



Utiliser un moteur de template

La gestion des routeurs implique de traiter des données et de les transmettre dans les documents HTML. Node.js est un serveur, il peut traiter du contenu, l'insérer dans un document HTML avant de le transmettre au navigateur comme le fait Apache avec PHP.

Ce traitement peut être fait par de nombreuses libraires de code. Certaines d'entre elles comme React.js, Angular.js ou Vue.js ont pris de l'importance et se livrent une forte concurrence.

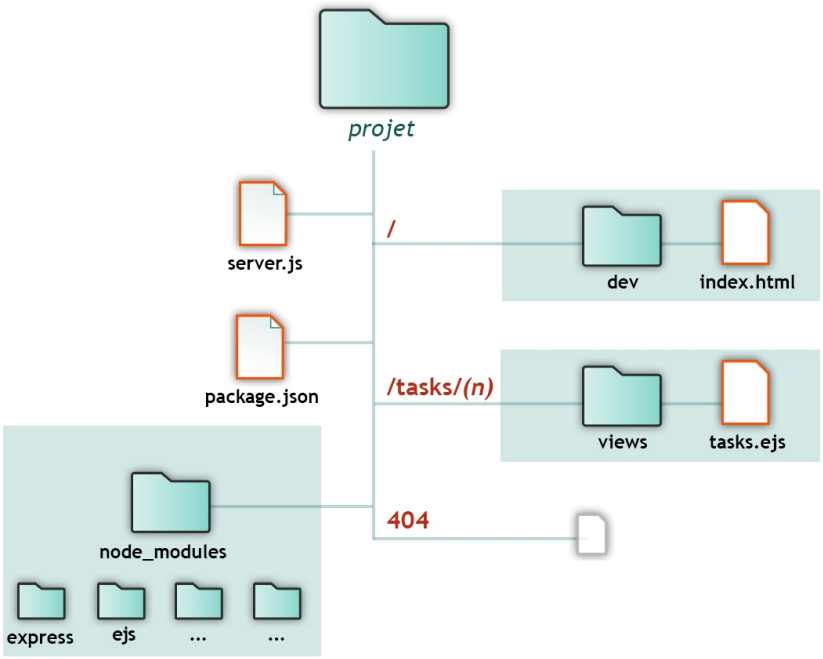
Dans le cas présent, sera utilisé la librairie ejs (Embeded Javascript) qui gère les contenus transmis par variables contenant des chaînes de caractère, entiers ou tableaux de données (array).

L'installation de ejs se fait ensuite par l'appel de l'instruction **npm install ejs --save-dev** dans l'invite de commandes.

Invite de commandes

```
npm install ejs --save-dev
```

Le traitement des templates par ejs implique que celles-ci soit dans un dossier « views ». Il est donc utile de revoir l'accès de la page « tasks.html » et en la transformant en page « page.ejs » placée dans un répertoire « views » a ajouté dans le projet.



Du précédent exemple, sera dirigé dorénavant vers le routeur vers le fichier « tasks.ejs » par la même méthode sendFile().

server.js

```
..  [...]
10  app.get('/tasks', function(req, res) {
11    res.sendFile('tasks.ejs');
12  });
..  [...]
```

Transfert de données au template

La librairie express.js prévoit la récupération de données depuis l'URL et la conversion de celles-ci en variable au format JSON. La récupération se fait par la déclaration d'une variable avec le symbole deux points [:] qui la précède à l'emplacement de la donnée transmise dans l'URL. Dans l'exemple qui suit, il s'agirait d'un entier récupéré dans l'URL par la variable [:numstasks].

server.js

```
..  [...]
10  app.get('/tasks/:numtasks', function(req, res) {
11    res.sendFile('tasks.ejs' + {tasks: req.params.numtasks});
12  });
..  [...]
```

Il est ensuite possible de transmettre la valeur récupérée au template « tasks.ejs » en exploitant la variable req initiée lors de la déclaration de la fonction anonyme comme l'illustre l'exemple précédent (req.params.numtasks).

L'exploitation de la valeur dans le template est possible grâce à la librairie ejs qui récupère la valeur transmise et l'utilise à l'aide de la syntaxe définit par la librairie. Les opérations sont directement intégrées au HTML à l'aide de délimiteurs <% et %>. Dans le cas de l'affichage de la valeur d'une variable est ajouté le symbole égal (=) au délimiteur. Comme ceci <%= tasks %>.

views/tasks.ejs

```
<!doctype html>
<html>
  <head>
        <title>App node et Express</title>
  </head>
  <body>
        <h1><%= tasks %> tâche(s) prévue(s)</h1>
        <ul>
            <% for(var i = 1 ; i <= tasks ; i++) { %>

                <li>Tâche n°<%= i %> <span></span></li>

            <% } %>
        </ul>
  </body>
</html>
```

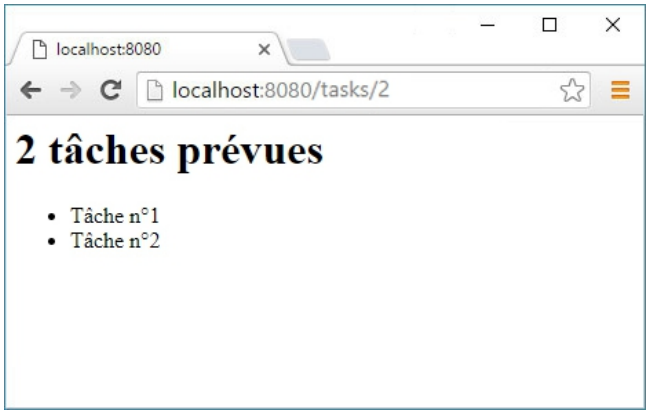
Le fichier « index.html » propose des liens vers différents nombre de tâches à accomplir en indiquant des entiers différents dans l'URL.

dev/index.html

```
<!doctype html>
<html>
  <head>
        <title>App node et Express</title>
  </head>
  <body>
        <h1>Gestionnaire de tâches</h1>
        <ul>
            <li><a href="tasks/1">1 tâche</a></li>
            <li><a href="tasks/2">2 tâches</a></li>
            <li><a href="tasks/3">3 tâches</a></li>
            <li><a href="tasks/4">4 tâches</a></li>
        </ul>
  </body>
</html>
```

Résultat : index.html sert de page d'accueil du serveur. Il propose d'accéder à différentes listes de tâches qui sont dirigées vers le fichier « tasks.ejs ».

L'URL indique le nombre de tâches à afficher et reproduit la liste des tâches correspondantes par l'exploitation des données transmises au template. En occurrence, l'entier « 2 » qui est dans l'URL.



Virtualiser les librairies

L'utilisation de ressources provenant de différentes librairies ayant une structure spécifique entraîne une organisation éclatée de ses ressources. Le répertoire « node_modules » dispose de nombreuses ressources mis à jour. Modifier son organisation en pensant faciliter l'accès à certains fichiers aurait comme conséquence que les fichiers ne soient plus en relation avec node.js et les instructions du package.json qui tient le rôle de chef d'orchestre.

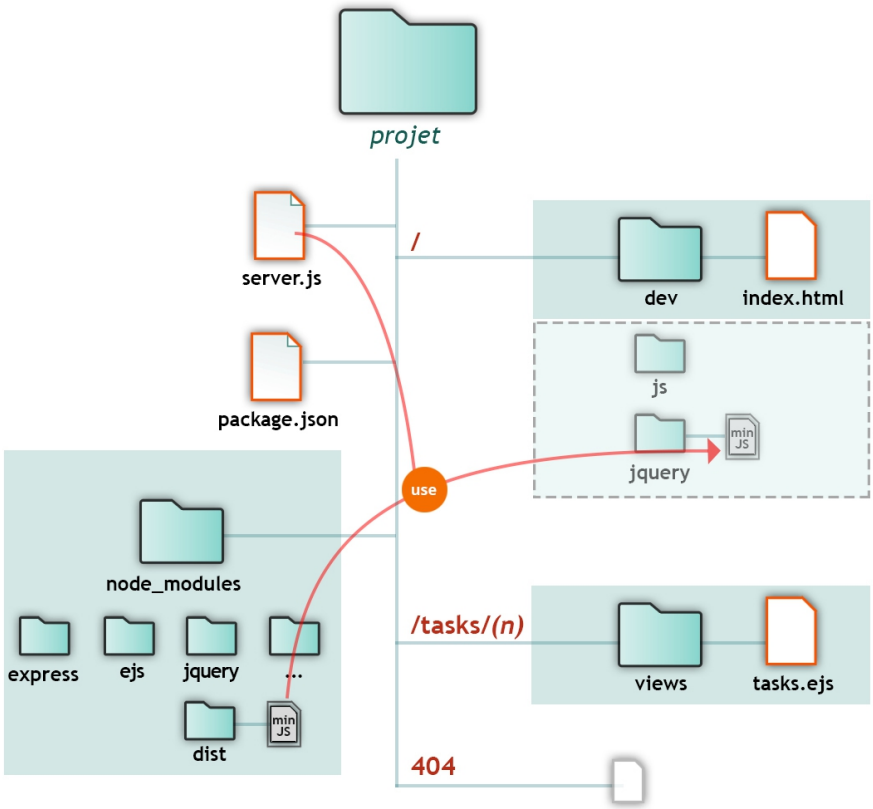
Il est donc utile de virtualiser certains fichiers et dossiers pour que les ressources d'origine restent bien à leur place mais qu'elles soient aisément disponibles par les fichiers utiles au développement qui en font référence. La librairie Express.js a prévu ce cas de figure et sait virtualiser des ressources pour un projet.

Afin d'illustrer son fonctionnement, la librairie jQuery sera installée par l'appel de l'instruction **npm install jquery --save-dev** dans l'invite de commandes.

Invite de commandes

```
npm install jquery --save-dev
```

La virtualisation sers possible en spécifiant dans le fichier du serveur **« server.js »** l'emplacement de la ressource d'origine, dans le cas présent jQuery qui est logé dans le répertoire **« node_modules/jquery/dist »**, et de lui indiquer l'emplacement à virtualiser de cette ressources. Compte tenu de la structure du projet, il serait pratique que ce soit dans le répertoire **« dev/js/jquery »**. Dans la cas présent, bien que les répertoires « js » et « jquery » n'existent pas, il seront virtuellement existant.



La méthode **use()** de Express.js permet cette opération.

server.js

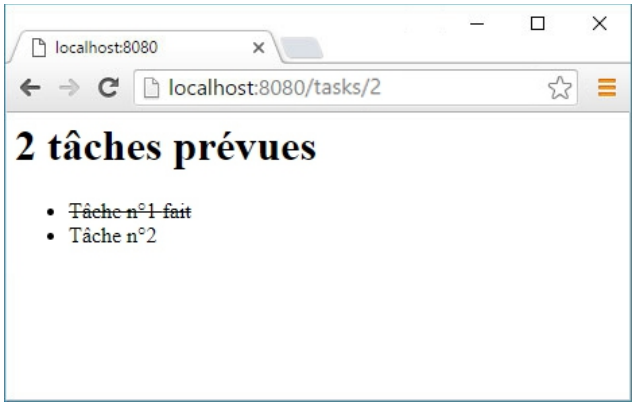
```
..  [...]
6   app.use('/dev/js/jquery', express.static(__dirname + '/node_modules/jquery/dist'));
..  [...]
```

Le fichier « tasks.ejs » peut dorénavant utiliser la libraire jQuery en localisant celle-ci selon sont emplacement virtuel.

views/tasks.ejs

```
1   <!doctype html>
2   <html>
3     <head>
4           <title>App node et Express</title>
5           <script src="../dev/js/jquery/jquery.min.js"></script>
6           <script>
7           $(window).on('load', function(){
8               $('li').on('click', function(){
9                   $(this) .css({textDecoration:'line-through'})
10                          .find('span')
11                          .html('fait');
12              });
13          })
14          </script>
..  [...]
```

Résultat : la librairie jQuery est associée à la page générée par le template et est en interaction avec le contenu de la page.



Git

Git est un outil d'archivage (versioning) qui gère les versions des fichiers d'un projet. Les outils qu'il propose facilite par le travail collaboratif et s'intègre totalement dans une production dite Agile par la création de branches de développement qui s'assimilent aux pratiques, notamment celles du sprint, de la méthode SCRUM.

Git permet de :

- gérer les versions des fichiers ;
- comparer les modifications entre ces versions ;
- récupérer une version antérieure ;
- accompagner la correction d'éventuels conflits provenant du travail en parallèle de plusieurs développeurs sur un même fichier.

Environnement et outils

Pour travailler avec Git il faut disposer d'un compte **github** (github.com). Les projets, appelés dépôts (repository) se gèrent depuis cette plateforme. Elle permet notamment de gérer le travail collaboratif en associant d'autres personnes à un dépôt et de consulter les fichiers et leurs versions. Github propose de consulter les projets d'autres développeurs, de suivre leurs travaux ou encore télécharger leurs projets.

Travailler avec Git signifie de disposer des fichiers sur son poste de travail (en local) et de les synchroniser avec le dépôt en ligne. En local, des commandes sont opérées pour mettre à jour les fichiers et gérer le projet à distance. Afin que le système d'exploitation puisse supporter ces commandes, il fait **installer la librairie Git** sur son poste. Il est téléchargeable à cette adresse https://git-scm.com/download.

L'édition des commandes peut ensuite se faire selon l'une de ces deux méthodes :

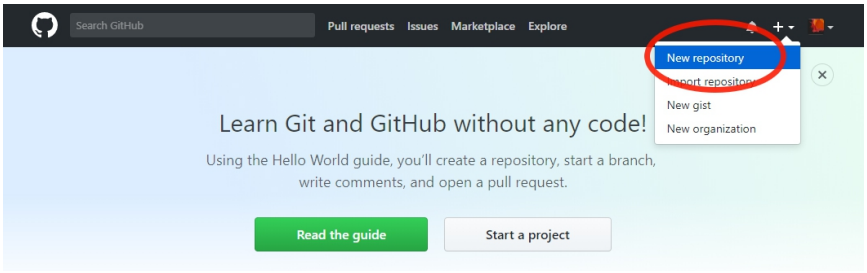
1. **En ligne de commande** dans l'invite de commandes. Git est à l'origine disposer à effectuer les opérations par l'entremise de commandes. Cette pratique est très répandue (plus que la pratique suivante – et parfois même nécessaire pour certains cas de figure).

2. **En utilisant un logiciel** proposant une interface rendant les opérations plus aisées (visuels) à mener. Git liste les logiciels du genre (https://git-scm.com/downloads/guis). Dans les exemples, qui suivront, a été utilisé le logiciel GitHub Desktop (https://desktop.github.com).

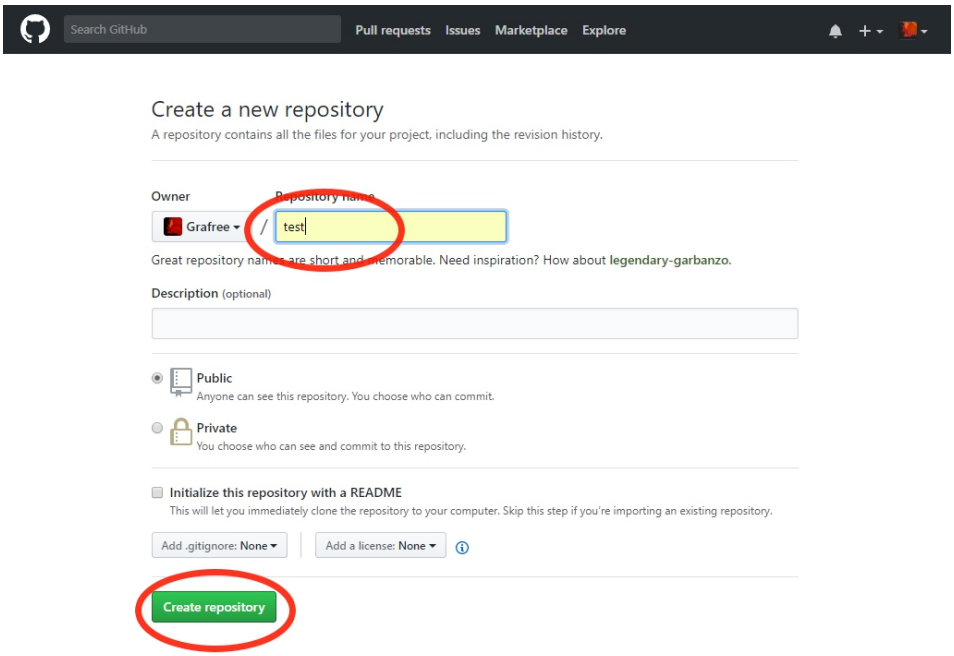
Créer un dépôt (*repository*)

Chaque projet archivé avec Git dispose d'un dépôt (repository) créé dans GitHub. Il s'agit du lieu de partage que les développeurs qui collaboreront sur le projet clonerons sur leur poste pour travailler.

Depuis le site de GitHub, créer un nouveau dépôt s'initie depuis la menu des actions (il est également possible de l'initier en local par ligne de commande ou un logiciel comme GitHub Desktop).



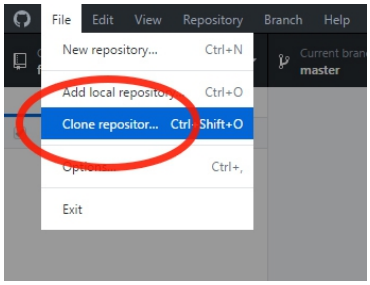
Puis configurer le dépôt en précisant le nom de celui-ci et validant avec le bouton « Create repository ».



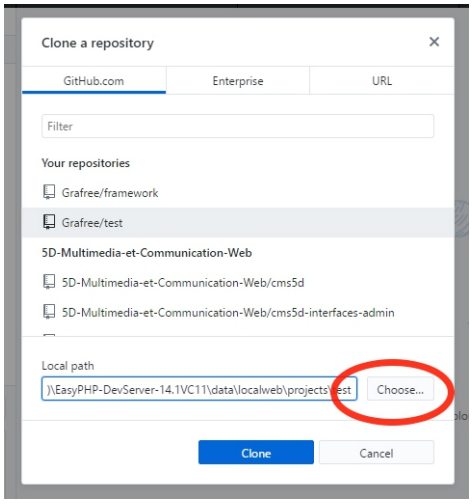
Synchroniser un dépôt en local

Une fois le dépôt créé, il est nécessaire de le synchroniser en local avec le lieu où les fichiers seront disposés sur le poste. Il est nécessaire de se connecter via le logiciel à son compte GitHub.

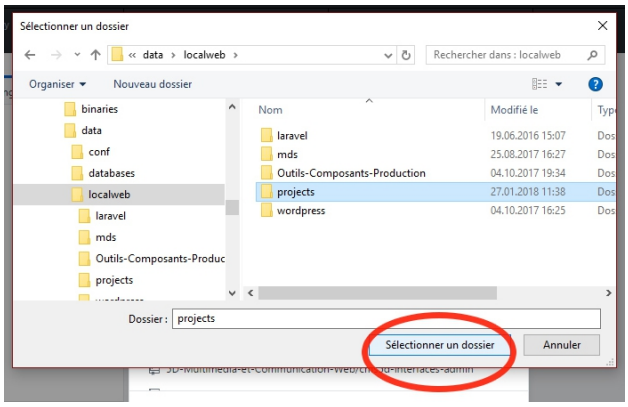
1, Depuis le logiciel GitHub Desktop, choisir dans le menu File, la commande « Clone repository ».

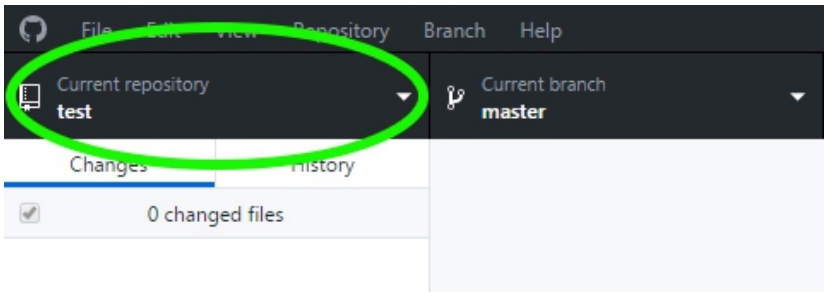


2, Identifier le dépôt du projet en le sélectionnant à partir de la liste proposant les dépôts des différents projets.



3, Sélectionner le répertoire dans lequel sera télécharger le dépôt et les fichiers qu'il contient. Cette opération créera un répertoire pour le projet.





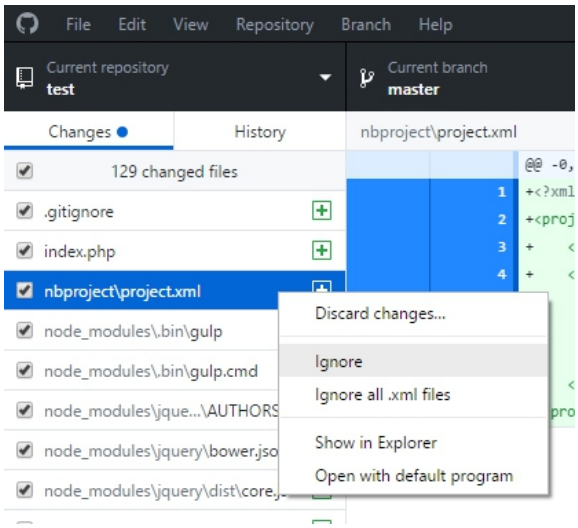
Résultat ; le dépôt synchronisé est identifié comme « Current repository ».

Les fichiers créés dans ce répertoire sont dorénavant susceptibles d'être ajoutés au dépôt du projet.

Ignorer des fichiers

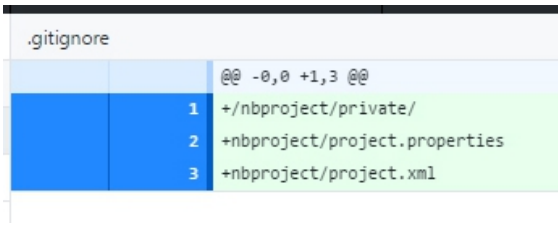
Certains fichiers ne sont pas utiles au fonctionnement du projet. Ils sont parfois générés par un logiciel d'édition de codes qui dispose de paramètres de synchronisation des fichiers par exemple. Ces fichiers n'ont pas à être présent dans le projet. On cherchera donc à les ignorer lors des synchronisations de fichiers est les introduisant dans une liste de fichiers .

Les projets Git dispose d'un fichier *.gitignore* qui sert à lister les fichiers à ne pas synchroniser. Les fichiers peuvent être ajoutés à la liste en ouvrant le fichier dans un éditeur de code. Un logiciel comme GitHub Desktop permet de le faire en localisant le fichier et en indiquant qu'il ne doit pas être archivé. Cette manipulation ajoutera le fichier à la liste des fichiers à exclure.



En accédant au menu contextuel (clic sur le bouton droit de la souris) sur le fichier, il est possible de le retirer en choisissant l'option « Ignore ».

Résultat : Le fichier se trouve dans la liste des fichiers exclus et ignorer du fichier *.gitignore*.



Aller un peu plus loin

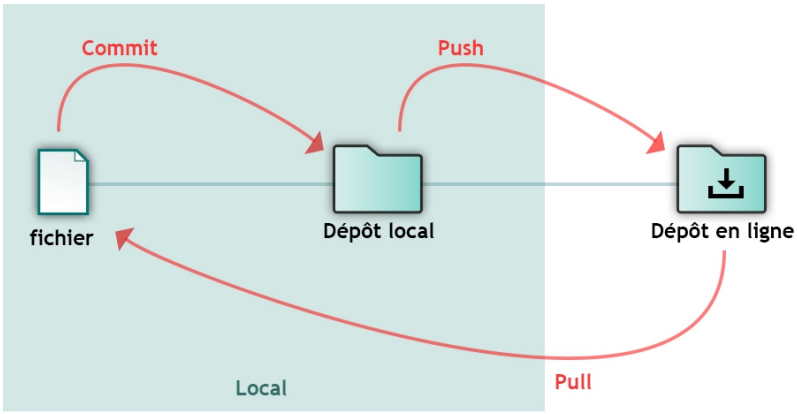
Il arrive qu'il soit plus pratique de retirer de la synchronisation tous les fichiers et/ou dossiers d'un répertoire. Indiquer un à un les fichiers à ignorer peut s'avérer long et fastidieux. Il peut dans ce cas être intéressant d'ajouter manuellement l'une des instructions suivantes dans le fichier.

Forme syntaxique	***Description***
`dossier/**`	Ignorer tous les dossiers
`Dossier/*.ext`	Ignorer tous les fichiers disposant de l'extension *ext*
`Dossier/*.*`	Ignorer tous les fichiers
`Dossier/!fichier.text`	Conserver le fichier fichier.text dans la synchronisation (dans le cas où le répertoire dans lequel il est contenu est ignoré.

Les actions d'archivage

Archiver un fichier se fait selon un processus que permet de présenter les étapes de sa production et ne prévoir un transfert définitif vers le dépôt en ligne qu'une fois l'ensemble des fonctionnalités sont bels et bien opérationnelles.

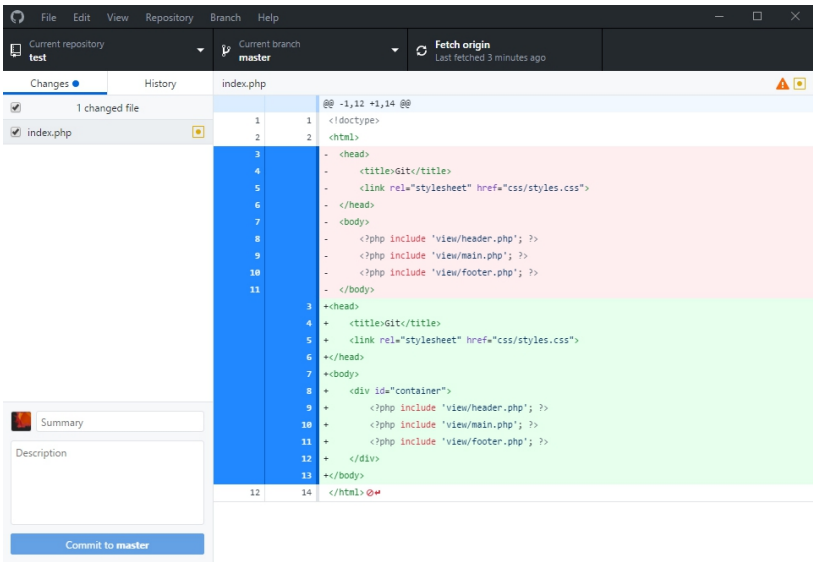
L'opération d'archivage des fichiers se fait donc en 2 temps. Il y a dans un premier temps le **« commit »** (engagement) qui conserve les fichiers en local mais les prépare au transfert définitive. Puis le **« push »** (pousser au dépôt distant) qui transmet tous les fichiers précédemment *commités* au dépôt en ligne.



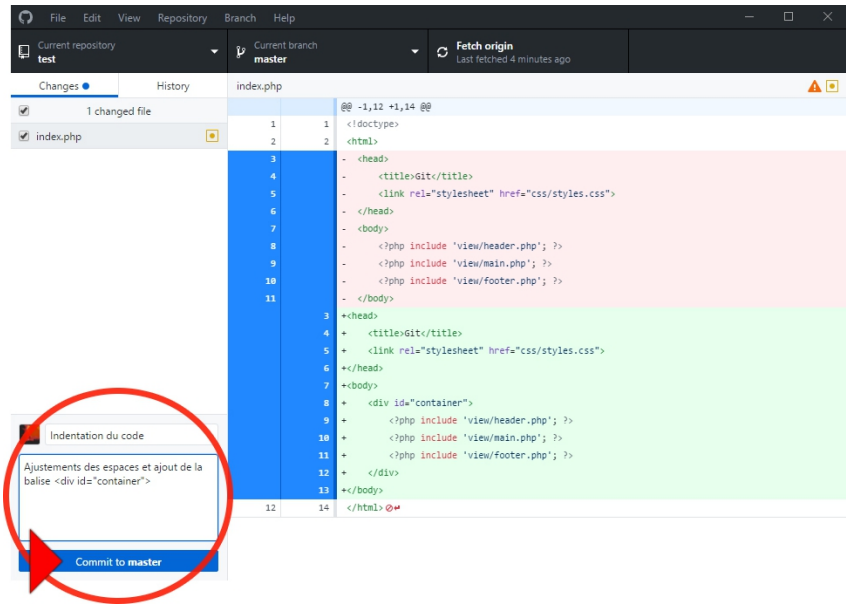
La récupération des fichiers du dépôt en ligne de fichiers ajoutés par d'autres développeurs se fait par un **« pull »** (tirer les fichiers dans notre dépôt local).

Commit et push

Le logiciel GitHub Desktop repère automatiquement les fichiers qui ont connus un changement et les liste dans la colonne de gauche. En cliquant, sur le fichier, le logiciel montre les changements apportés en indiquant en rouge la précédente version des lignes de code et en vert le code modifié.

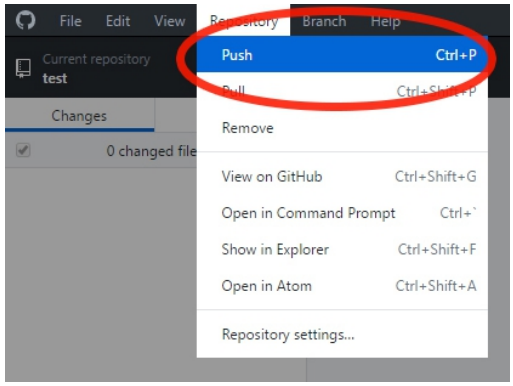


Faire un **« commit »** implique d'indiquer ce qui change. Git ne permet de commit s'il n'y a pas d'indication sur le changement qu'apporte ces fichiers au projet. Dans le logiciel GitHub Desktop, l'ajout d'une description est possible au-dessus du bouton permettant le « commit ».

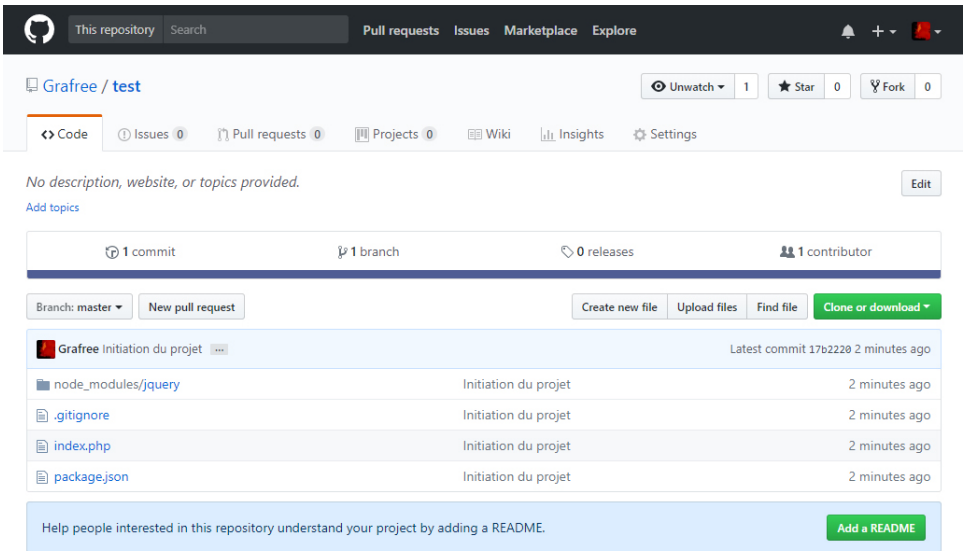


Le « push » conclue un développement prêt à être partagé et groupe très souvent plusieurs « commit ». Il peut se comparer à une publication en ligne du code.

Dans le logiciel GitHub Desktop, cette opération se fait depuis la rubrique « Repository » et par la commande « Push ».

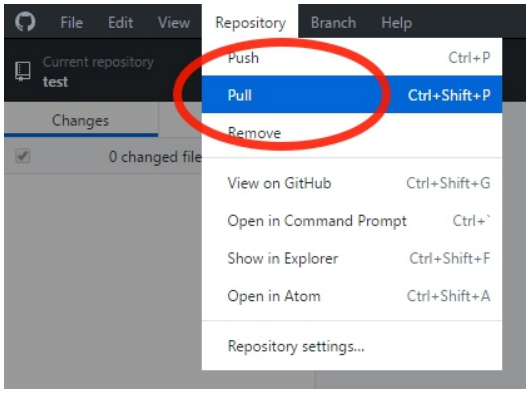


Résultat : Ces fichiers sont visibles et accessibles depuis le site de Github. On constate notamment, que le titre donné au commentaire des derniers « commits » pour chaque fichier est indiqué.



Pull

Il s'agit de l'opération de récupération des fichiers d'un projet suite à des modifications de fichiers dans le dépôt en ligne. A tout moment, un « pull » peut être effectué pour récupérer les dernières versions des fichiers. Effectuer des « pulls » fréquents est une bonne habitude à prendre lors d'un développement en équipe. Cela ne remplacera toutefois pas une bonne communication qui limite le travail par plusieurs développeurs sur un même fichier en même temps.

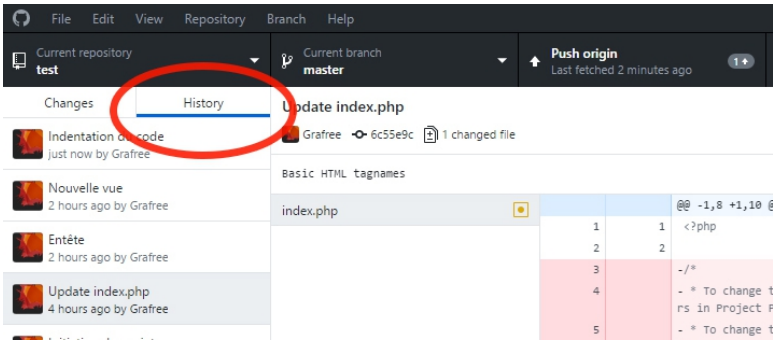


Dans le logiciel GitHub Desktop, cette opération se fait depuis la rubrique « Repository » et par la commande « Pull ».

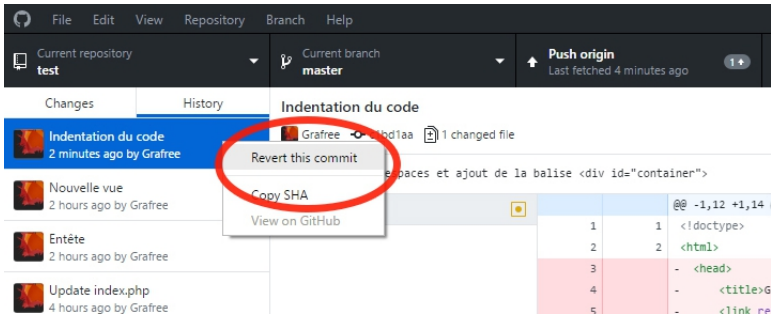
L'historique des « commits »

L'un des avantages qu'offre Git est de permettre à un développeur de faire marche arrière et de récupérer une version antérieure d'un ou plusieurs fichiers. Il crée un historique des fichiers à travers les « commits » qui permet ce retour en arrière.

Dans le logiciel GitHub Desktop, l'onglet « History » permet d'accéder à la liste des « commits » effectués et de consulter les fichiers afin de constater des modifications réalisées.



Il est ensuite possible de récupérer les fichiers d'un précédent « commit » en accédant au menu contextuel (clic sur le bouton droit de la souris) et en sélectionnant la commande « Revert this commit ».

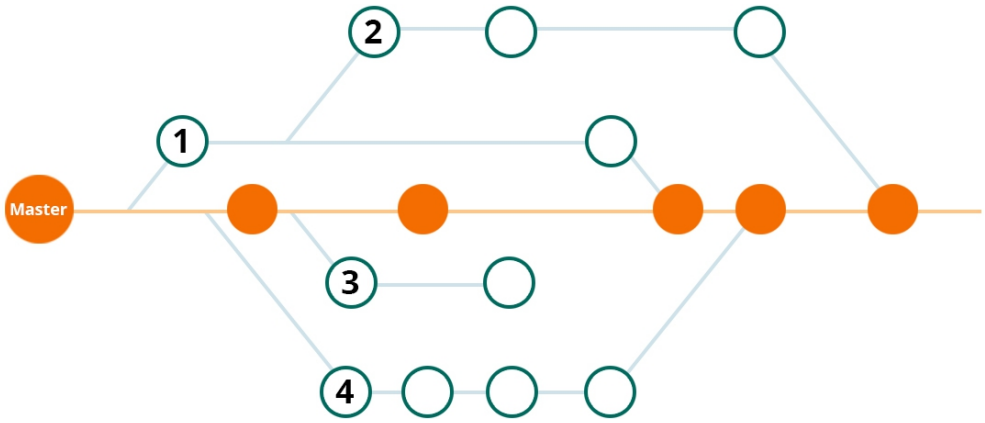


Les branches

Les ajouts des fichiers et de leurs versions se greffent par défaut à un tronc commun qui est appelé « master ». Ce tronc est considéré comme la version fonctionnelle du projet. Les améliorations, futurs développements ou simples expériences devraient donner lieu à un développement parallèle qui n'affecte pas le tronc principal « master » et n'être fusionné avec ce dernier que lorsqu'il sera jugé prêt à l'emploi.

Il s'agit-là de la représentation de ce qu'est une branche. Elle peut être créer à tout moment, avoir une durée de vie plus ou moins longue et se greffer à nouveau au projet du tronc principal lorsque le développement est terminé. Certaines branches seront abandonnées parce que le développement qu'elles comportent n'a pas donné le résultat attendu ou simplement parce qu'il s'agissait d'un test qui une fois abouti ne sert plus.

Le schéma suivant illustre différents scénarios correspondant à des développements dans un même projet.



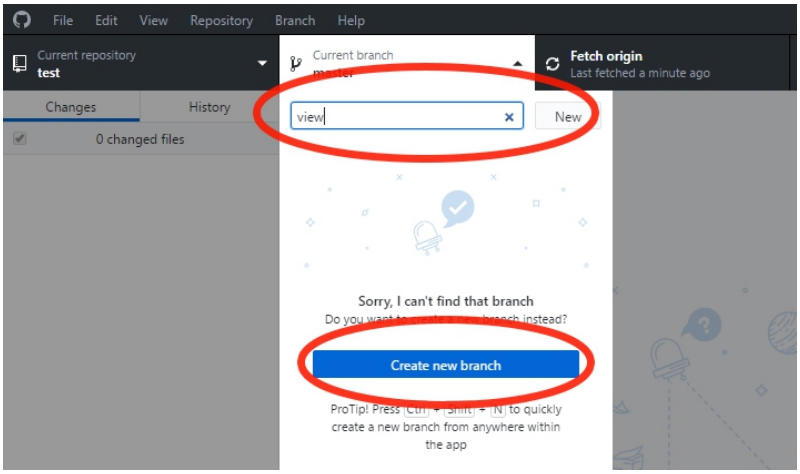
Branches (scénarios)
1. Développement d'un module par un collaborateur au projet. Au terme de ce développement, le module a été ajouté au tronc principal.
2. Développement d'une nouvelle fonctionnalité qui s'avère utile au projet. La nécessité de cette fonctionnalité a été définie pendant le développement du module à la branche no°1. Les développements initiés au démarrage du module ont été récupérés dans la branche no°2. Une fusion a été engagée au tronc principal, une fois le développement terminé.
3. Développement test d'une fonctionnalité qui a aidé au projet mais n'a pas d'utilité définitive.
4. Développement d'une nouvelle fonctionnalité prévue au projet. Au terme de son développement, elle a été ajoutée au tronc principal.

Créer une branche

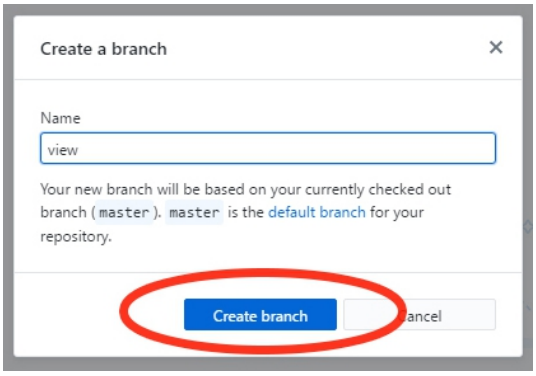
Git permet la création d'une branche à tout moment.

Dans le logiciel GitHub Desktop, l'onglet **« Current branch »** donne accès à un champ texte qui permet d'indiquer un nom à cette future branche.

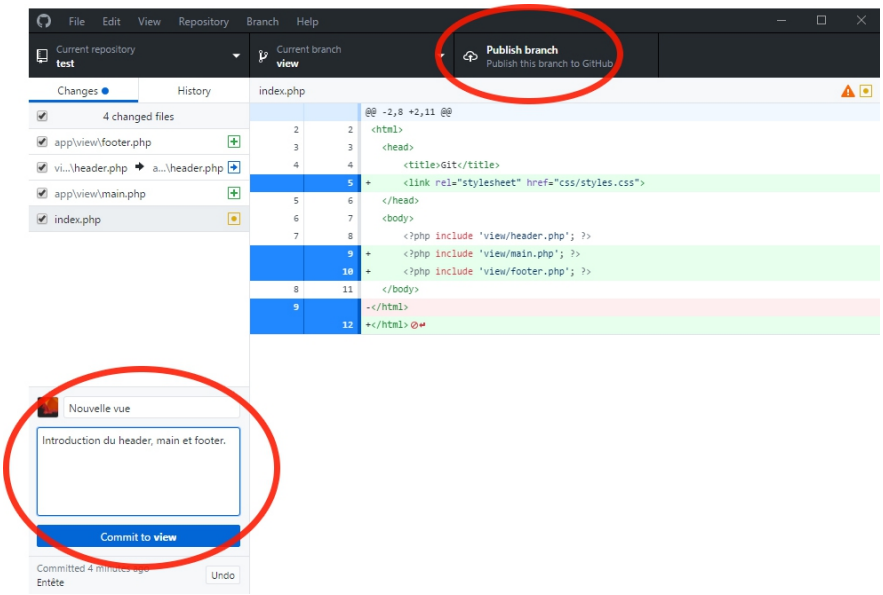
L'initiation de cette branche se fait en cliquant sur le bouton **« Create new branch »**.



La confirmation de la création de la branche se fait ensuite par la validation du nom choisi.



Résultat : A l'avenir les « commits » et « push » se feront sur cette branche. L'onglet **« Current branch »** donne accès aux différentes branches ainsi qu'au tronc principal (master), ce qui permet de poursuivre les développements en parallèle.

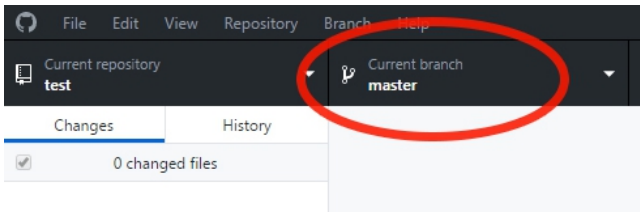


Fusionner une branche

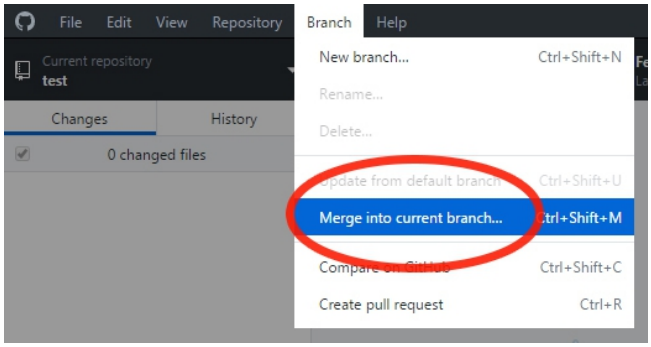
Une fois le développement sur une branche terminé est à prévoir la fusion de cette branche au tronc principal.

Pour cela, il faut initier la démarche en ce situant sur le tronc principal.

Dans le logiciel GitHub Desktop, l'onglet **« Current branch »** donne accès aux différentes branches et au tronc principal.



Accéder depuis la rubrique « Branch » à la commande **« Merge into current branch... »**.



Une fenêtre permet de définir la branche à fusionner. Une l'avoir sélectionner, valider le choix en cliquant sur le bouton **« Merge into master »**.