Elixir

Laravel comporte un certain nombre d'outils intéressants. Parmi eux il en est un que j'ai toujours négligé et je m'y suis intéressé récemment en me rendant compte qu'il est vraiment très utile, il s'agit d'Elixir.

Pour comprendre Elixir il faut déjà connaître Gulp. C'est un gestionnaire de tâches (task runner). En gros on lui dit de faire une série d'actions (minifier du css ou du Javascript, compresser une image, concaténer des fichiers, copier des fichiers, compiler du LESS ou du SASS, créer un serveur…). Il sait pratiquement tout faire grâce à la montagne de plugins qui existent.

Gulp

Pour faire fonctionner Gulp il faut déjà installer Node.js. C'est un véritable écosystème en JavaScript dont je ne vais pas parler dans cet article parce que ce n'est pas le sujet. Il faut juste l'installer en récupérant la bonne version sur cette page.

Une fois que Node est installé il faut installer Gulp avec npm qui est automatiquement installé en même temps que Node. C'est un gestionnaire de packages JavaScript, un peu comme Composer est un gestionnaire de packages pour PHP. Dans la console tapez ça pour l'installer de façon globale (donc disponible dans n'importe quel dossier) :

```
npm install gulp -g
```

Pour fonctionner Gulp a besoin de deux fichiers :

- **package.json** : contient la liste des plugins dont on a besoin (c'est l'équivalent de composer.json),
- **gulpfile.js** : contient la liste des tâches.

On peut créer le fichier **package.json** avec cette commande :

```
npm init
```

Il faut répondre aux questions posées (ou utiliser **Entrée** pour accepter la valeur par défaut), et on se retrouve avec ce code :

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Pour le moment on a rien à installer, alors on va installer Gulp localement pour le projet :

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

Vous vous retrouvez avec un dossier **node_modules** bien rempli et le fichier **package.json** a été mis à jour :

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "gulp": "^3.9.1"
  }
}
```

Mais Gulp tout seul ne sait pas faire grand chose… On va installer le plugin pour compiler du LESS :

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

On va créer un fichier **less/styles.less** avec de la syntaxe LESS :

```
@base: #f938ab;

.box-shadow(@style, @c) when (iscolor(@c)) {
  -webkit-box-shadow: @style @c;
  box-shadow:         @style @c;
}
.box-shadow(@style, @alpha: 50%) when (isnumber(@alpha)) {
  .box-shadow(@style, rgba(0, 0, 0, @alpha));
}
.box {
  color: saturate(@base, 5%);
  border-color: lighten(@base, 30%);
  div { .box-shadow(0 0 5px, 30%) }
}
```

On va maintenant créer le fichier des tâches **gulpfile.js** :

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

gulp.task('default', function () {
  return gulp.src('./less/styles.less')
    .pipe(less())
    .pipe(gulp.dest('./css'));
});
```

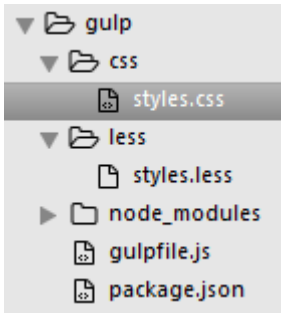
On définit qu’on utilise **gulp** et le plugin **gulp-less** en affectant deux variables.

Ensuite on définit la tâche par défaut (**default**). La syntaxe est simple et efficace.

On lance la tâche simplement en entrant dans la console :

```
gulp
```

Si ce n’était pas la tâche par défaut il faudrait entrer à la suite le nom de la tâche. On obtient un nouveau fichier **css/styles.css** :



Avec ce code :

```
.box {
  color: #fe33ac;
  border-color: #fdcdea;
}
.box div {
  -webkit-box-shadow: 0 0 5px rgba(0, 0, 0, 0.3);
  box-shadow: 0 0 5px rgba(0, 0, 0, 0.3);
}
```

C’est à dire exactement ce qu’on voulait !

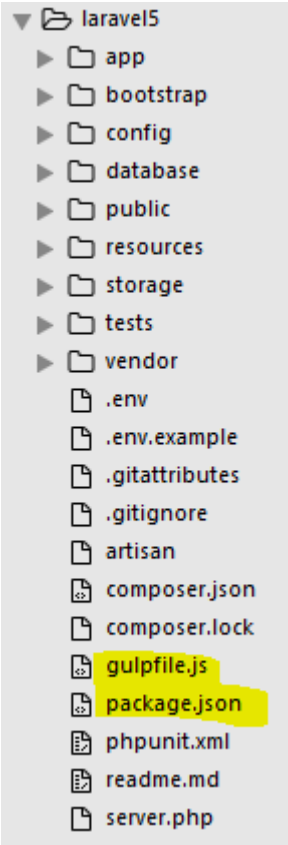
Si vous voulez en savoir plus sur Gulp je vous conseille la documentation officielle.

Elixir

Installation et fichiers CSS

Si je vous ai présenté Gulp c’est parce qu’Elixir se contente de l’utiliser de façon très intelligente.

Dans une installation toute fraîche de Laravel vous trouvez deux fichiers qui maintenant devraient vous parler :



Regardons ce qu'on a par défaut. Voici le contenu de **package.json** :

```
{
  "private": true,
  "scripts": {
    "prod": "gulp --production",
    "dev": "gulp watch"
  },
  "devDependencies": {
    "gulp": "^3.9.1",
    "laravel-elixir": "^5.0.0",
    "bootstrap-sass": "^3.0.0",
  }
}
```

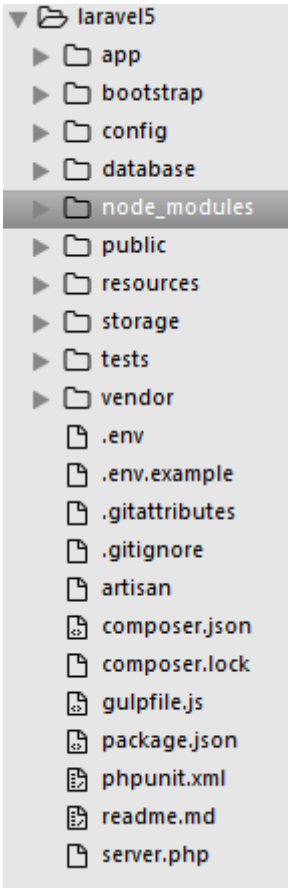
On voit qu'on a Gulp déclaré ainsi que le plugin **laravel-elixir**. On trouve aussi **bootstrap-sass**. Pour installer tout ça il faut utiliser la commande qu'on a déjà vue ci-dessus :

```
npm install
```

L'installation prend un certain temps parce qu'il y a du monde !

Si tout se passe bien vous devriez avoir un dossier **node_modules**

avec toutes les dépendances :

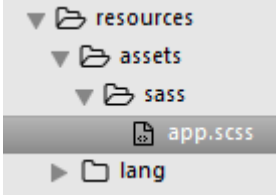


Voici maintenant ce qu'on trouve dans **gulpfile.js** :

```
var elixir = require('laravel-elixir');

/*
 |----------------------------------------------------------------
----------
 | Elixir Asset Management
 |----------------------------------------------------------------
----------
 |
 | Elixir provides a clean, fluent API for defining some basic
Gulp tasks
 | for your Laravel application. By default, we are compiling the
Sass
 | file for our application, as well as publishing vendor
resources.
 |
 */

elixir(function(mix) {
    mix.sass('app.scss');
```

```
});
```

On n’a pas directement des tâches Gulp mais des méthodes d’Elixir qui sert d’interface en simplifiant la syntaxe. Ici par exemple on demande de compiler le code SASS qui se trouve dans le fichier **app.scss**. Où se trouve ce fichier ? Ici :



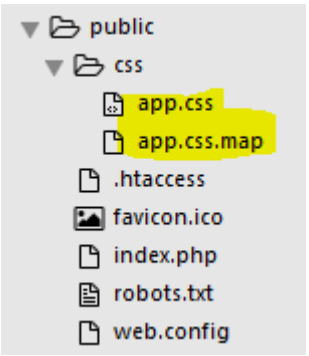
Très logiquement rangé dans les ressources. Par défaut ce fichier est vide. On va y ranger un peu de code :

```
$font-stack:    Helvetica, sans-serif;
$primary-color: #333;

body {
  color: $primary-color;
}
```

Et on lance Elixir (donc gulp) :

```
gulp
```

Et on se retrouve avec le fichier compilé accompagné du fichier map bien rangés :



Avec ce code CSS :

```
body {
  color: #333; }
```

Tout simple !

Si vous n’avez pas besoin du fichier map il suffit de préciser cette option :

```
elixir.config.sourcemaps = false;
```

Si vous lancez la commande :

```
gulp --production
```

Le résultat est que vous obtenez automatiquement le code minifié :

```
body{color:#333}
```

Elixir peut vraiment nous simplifier la vie !

On peut évidemment compiler plusieurs fichiers à la fois, prévoir une autre destination.

On procède pareil pour des fichiers LESS :

```
elixir(function(mix) {
    mix.less('app.less');
});
```

Et la cerise sur le gâteau vous pouvez surveiller les modifications des fichiers sources pour avoir une compilation à la volée avec cette commande :

```
gulp watch
```

Copie de fichiers ou dossiers

Une action fréquente est la copie de fichiers ou de dossiers. là aussi Elixir nous propose une syntaxe élégante :

```
elixir(function(mix) {
    mix.copy('dossier/truc.css', 'public/css/truc.css');
    mix.copy('dossier/actions', 'resources/actions');
});
```

Dans la première action on copie un fichier, dans la seconde un dossier complet.

Les scripts

JavaScript

Quand on utilise pas mal de code JavaScript on sépare les fonctionnalités en plusieurs fichiers, mais il arrive un moment où on veut tout rassembler dans un seul fichier, on veut finalement minifier le résultat.

Pour le rassemblement la syntaxe est celle-ci :

```
elixir(function(mix) {
    mix.scripts([
        'main.js',
        'module.js'
    ]);
});
```

Par défaut les fichiers doivent se trouver dans **resource/assets/js**, sinon il faut spécifier le chemin.

Par défaut le résultat sera dans **public/js/all.js**. Pour spécifier une autre destination il faut le faire avec un deuxième argument.

Et évidemment le résultat sera minifié en utilisant :

```
gulp --production
```

On peut rassembler tous les scripts présents dans un dossier avec cette syntaxe :

```
elixir(function(mix) {
    mix.scriptsIn('resource/js/mon_directory');
});
```

Autres scripts

Il arrive de plus en plus souvent qu’on utilise des langages plus évolués que le JavaScript en version ES5 et évidemment on a envie au final de tout ramener à du JavaScript compréhensible par tous les navigateurs (donc actuellement du ES5).

CoffeeScript

Par exemple on peut utiliser CoffeeScript. Je ne l'ai personnellement jamais utilisé. Il existe une méthode dans Elixir pour le compiler :

```
elixir(function(mix) {
    mix.coffee(['app.coffee', 'controllers.coffee']);
});
```

Browserify

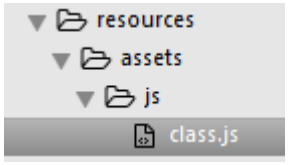
Elixir prend également en charge Browserify. Pour ceux qui ne connaissent pas, Browserify prend du code conforme à la syntaxe CommonJS pour le transformer en quelque chose d'interprétable par les navigateurs. La syntaxe est encore la même :

```
elixir(function(mix) {
    mix.browserify('main.js');
});
```

Babel

Plus intéressant : Elixir prend aussi en charge Babel. Donc si on utilise la nouvelle syntaxe ES2015 on peut facilement utiliser Babel pour transformer le code en ES5.

Prenons par exemple un fichier en ES2015 placé ici :



Avec ce code :

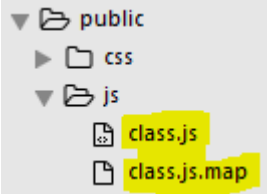
```
class SkinnedMesh extends THREE.Mesh {
  constructor(geometry, materials) {
    super(geometry, materials);
    this.idMatrix = SkinnedMesh.defaultMatrix();
    this.bones = [];
    this.boneMatrices = [];
  }
  update(camera) {
    super.update();
```

```
  }
  static defaultMatrix() {
    return new THREE.Matrix4();
  }
}
```

On prévoit cette tâche :

```
elixir(function(mix) {
    mix.babel('class.js');
});
```

Lorsqu'on l'a exécutée on se retrouve avec deux fichiers résultants (le fichier javaScript et le map) ici :



Et on peut vérifier le code résultant dans **class.js** :

```
"use strict";

var _createClass = function () { function defineProperties(target, props) { for (var i = 0; i < props.length; i++) { var descriptor = props[i]; descriptor.enumerable = descriptor.enumerable || false; descriptor.configurable = true; if ("value" in descriptor) descriptor.writable = true; Object.defineProperty(target, descriptor.key, descriptor); } } return function (Constructor, protoProps, staticProps) { if (protoProps) defineProperties(Constructor.prototype, protoProps); if (staticProps) defineProperties(Constructor, staticProps); return Constructor; }; }();

var _get = function get(object, property, receiver) { if (object === null) object = Function.prototype; var desc = Object.getOwnPropertyDescriptor(object, property); if (desc === undefined) { var parent = Object.getPrototypeOf(object); if (parent === null) { return undefined; } else { return get(parent, property, receiver); } } else if ("value" in desc) { return desc.value; } else { var getter = desc.get; if (getter === undefined) { return undefined; } return getter.call(receiver); } };
```

```
function _classCallCheck(instance, Constructor) { if (!(instance instanceof Constructor)) { throw new TypeError("Cannot call a class as a function"); } }

function _possibleConstructorReturn(self, call) { if (!self) { throw new ReferenceError("this hasn't been initialised - super() hasn't been called"); } return call && (typeof call === "object" || typeof call === "function") ? call : self; }

function _inherits(subClass, superClass) { if (typeof superClass !== "function" && superClass !== null) { throw new TypeError("Super expression must either be null or a function, not " + typeof superClass); } subClass.prototype = Object.create(superClass && superClass.prototype, { constructor: { value: subClass, enumerable: false, writable: true, configurable: true } }); if (superClass) Object.setPrototypeOf ? Object.setPrototypeOf(subClass, superClass) : subClass.__proto__ = superClass; }

var SkinnedMesh = function (_THREE$Mesh) {
  _inherits(SkinnedMesh, _THREE$Mesh);

  function SkinnedMesh(geometry, materials) {
    _classCallCheck(this, SkinnedMesh);

        var _this = _possibleConstructorReturn(this, Object.getPrototypeOf(SkinnedMesh).call(this, geometry, materials));

    _this.idMatrix = SkinnedMesh.defaultMatrix();
    _this.bones = [];
    _this.boneMatrices = [];
    return _this;
  }

  _createClass(SkinnedMesh, [{
    key: "update",
    value: function update(camera) {
      _get(Object.getPrototypeOf(SkinnedMesh.prototype), "update", this).call(this);
    }
  }], [{
    key: "defaultMatrix",
```

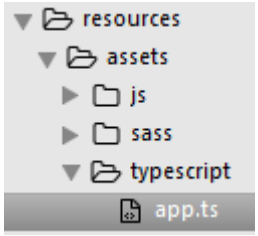
```
    value: function defaultMatrix() {
      return new THREE.Matrix4();
    }
  }]);

  return SkinnedMesh;
}(THREE.Mesh);
```

TypeScript

Personnellement j'aime bien TypeScript qui va plus loin que ES2015 et supporte par exemple les décorateurs. C'est d'ailleurs le langage qui a été choisi pour AngularJS 2. Elixir ne le supporte pas par défaut et il faut donc charger un nouveau plugin :

```
npm install elixir-typescript --save-dev
```

On va créer un petit fichier TypeScript dans un dossier spécifique :



Avec ce simple code :

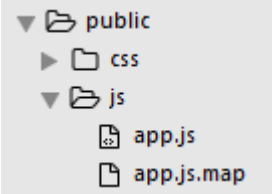
```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
```

Et cette tâche :

```
var elixir = require('laravel-elixir');
var elixirTypscript = require('elixir-typescript');

elixir(function(mix) {
```

```
  mix.typescript('app.ts');
});
```

Après exécution on a le résultat ici :



Avec ce JavaScript résultant :

```
var Greeter = (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
}());
```

Si on veut utiliser les décorateurs il faut le préciser dans les options :

```
var elixir = require('laravel-elixir');
var elixirTypscript = require('elixir-typescript');

elixir(function(mix) {
  mix.typescript('app.ts', 'public/js', {
    "target": "es5",
    "experimentalDecorators": true
  });
});
```

Si on a par exemple ce code pour **app.ts** :

```
function f() {
    console.log("f(): evaluated");
     return function(target, propertyKey: string, descriptor:
PropertyDescriptor) {
        console.log("f(): called");
    }
}
```

```
function g() {
    console.log("g(): evaluated");
     return function(target, propertyKey: string, descriptor:
PropertyDescriptor) {
        console.log("g(): called");
    }
}

class C {
    @f()
    @g()
    method() { }
}
```

Après compilation on obtient :

```
var __decorate = (this && this.__decorate) || function
(decorators, target, key, desc) {
    var c = arguments.length, r = c < 3 ? target : desc === null ?
desc = Object.getOwnPropertyDescriptor(target, key) : desc, d;
    if (typeof Reflect === "object" && typeof Reflect.decorate ===
"function") r = Reflect.decorate(decorators, target, key, desc);
    else for (var i = decorators.length - 1; i >= 0; i--) if (d =
decorators[i]) r = (c < 3 ? d(r) : c > 3 ? d(target, key, r) :
d(target, key)) || r;
    return c > 3 && r && Object.defineProperty(target, key, r), r;
};
function f() {
    console.log("f(): evaluated");
    return function (target, propertyKey, descriptor) {
        console.log("f(): called");
    };
}
function g() {
    console.log("g(): evaluated");
    return function (target, propertyKey, descriptor) {
        console.log("g(): called");
    };
}
var C = (function () {
    function C() {
    }
    C.prototype.method = function () { };
    __decorate([
```

```
        f(),
        g()
    ], C.prototype, "method", null);
    return C;
}());
```

Les options peuvent se trouver dans un fichier à part **tsconfig.json** :

```
{
  "compilerOptions": {
    "target": "es5",
    "experimentalDecorators": true,
  }
}
```

Il sera automatiquement pris en compte.

Vous pouvez trouver toutes les options disponibles ici.

On peut encore faire d’autres choses intéressantes avec Elixir, allez jeter un coup d’oeil à la documentation.