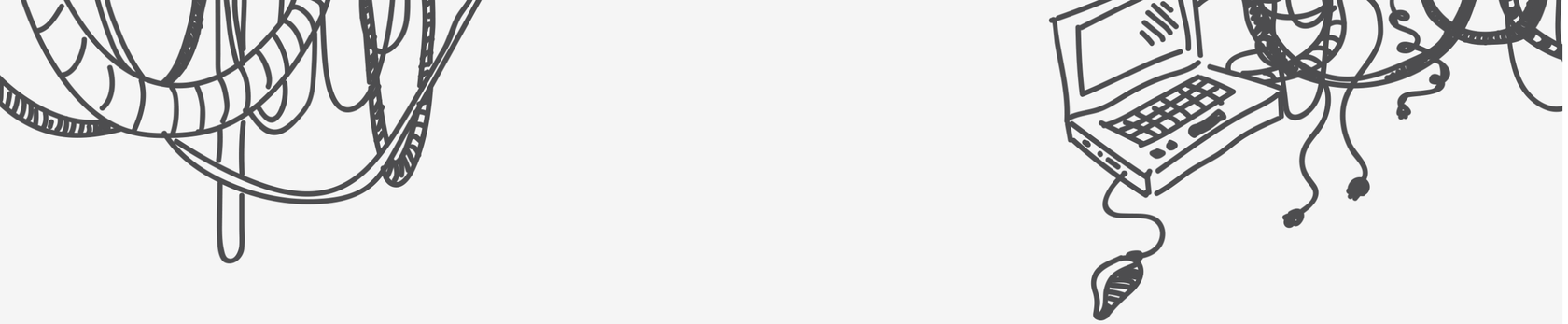
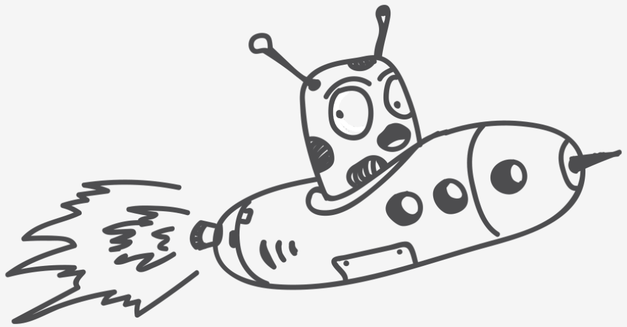
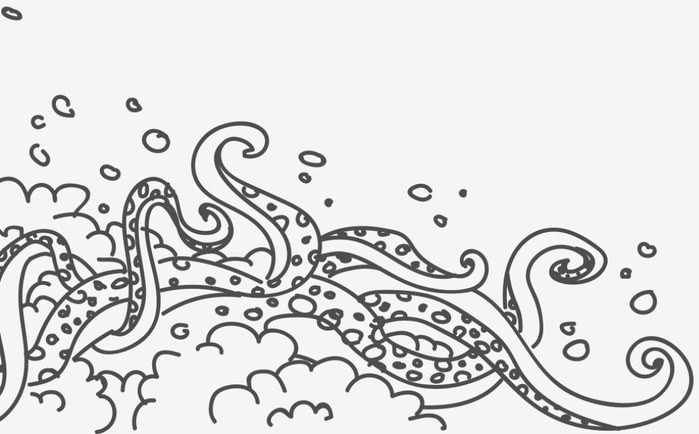






Konfiguracja zadań

Konfiguracja zadań

Plik `gulpfile.js`

W naszym repozytorium stwórz plik `gulpfile.js` a następnie wpisz do niego wstępną konfigurację:

```
const gulp = require('gulp');

function firstTask(done) {
  console.log('Pierwsze zadanie');
  done();
}

exports.firstTask = firstTask;
exports.default = gulp.parallel(firstTask);
```

Omówmy co zadziało się w naszym przykładzie.

Konfiguracja zadań

Plik `gulpfile.js`

W naszym repozytorium stwórz plik `gulpfile.js` a następnie wpisz do niego wstępną konfigurację:

```
const gulp = require('gulp');

function firstTask(done) {
  console.log('Pierwsze zadanie');
  done();
}

exports.firstTask = firstTask;
exports.default = gulp.parallel(firstTask);
```

Omówmy co zadziało się w naszym przykładzie.

Korzystamy z paczki `gulp` i zapisujemy ją do zmiennej.

Konfiguracja zadań

Plik `gulpfile.js`

W naszym repozytorium stwórz plik `gulpfile.js` a następnie wpisz do niego wstępną konfigurację:

```
const gulp = require('gulp');

function firstTask(done) {
  console.log('Pierwsze zadanie');
  done();
}

exports.firstTask = firstTask;
exports.default = gulp.parallel(firstTask);
```

Omówmy co zadziało się w naszym przykładzie.

Korzystamy z paczki `gulp` i zapisujemy ją do zmiennej.

Tworzymy funkcję odpowiedzialną za wykonywanie różnych czynności.

Konfiguracja zadań

Plik `gulpfile.js`

W naszym repozytorium stwórz plik `gulpfile.js` a następnie wpisz do niego wstępną konfigurację:

```
const gulp = require('gulp');

function firstTask(done) {
  console.log('Pierwsze zadanie');
  done();
}

exports.firstTask = firstTask;
exports.default = gulp.parallel(firstTask);
```

Omówmy co zadziało się w naszym przykładzie.

Korzystamy z paczki `gulp` i zapisujemy ją do zmiennej.

Tworzymy funkcję odpowiedzialną za wykonywanie różnych czynności.

Eksportujemy naszą funkcję pod taką samą nazwą.

Konfiguracja zadań

Plik `gulpfile.js`

W naszym repozytorium stwórz plik `gulpfile.js` a następnie wpisz do niego wstępną konfigurację:

```
const gulp = require('gulp');

function firstTask(done) {
  console.log('Pierwsze zadanie');
  done();
}

exports.firstTask = firstTask;
exports.default = gulp.parallel(firstTask);
```

Omówmy co zadziało się w naszym przykładzie.

Korzystamy z paczki `gulp` i zapisujemy ją do zmiennej.

Tworzymy funkcję odpowiedzialną za wykonywanie różnych czynności.

Eksportujemy naszą funkcję pod taką samą nazwą.

Mówimy Gulpowi co jest **domyślnym zadaniem**.

Konfiguracja zadań - omówienie

```
const gulp = require('gulp');

function firstTask(done) {
  console.log('Pierwsze zadanie');
  done();
}

exports.firstTask = firstTask;
exports.default = gulp.parallel(firstTask);
```

Sprawdźmy czy wszystko poszło poprawnie.

W terminalu wpiszmy polecenie `gulp -T` co powinno pokazać nam listę tasków (w naszym przypadku tylko jeden). Jeżeli nie pamiętamy tego polecenia, wpiszmy `gulp -h`, co pokaże nam wszystkie możliwe polecenia dla gulp-cli.

Sprawdźmy czy możemy uruchomić nasze zadanie poleceniem `gulp firstTask`. W terminalu powinien pojawić się tekst "Pierwsze zadanie". Dodatkowo sprawdźmy, czy działa nasze domyślne zadanie, używając komendy `gulp` (bez podawania nazwy tasku)

Konfiguracja zadań - parallel i series

Jak pewnie zauważyliście, przy definiowaniu domyślnego zadania użyliśmy metody `parallel`. Służy do odpalania zadań równocześnie. Stosuje się kiedy chcemy odpalić kilka zadań równocześnie, a nie ma dla nas znaczenia, które zadanie zakończy się jako pierwsze.

```
gulp.parallel(zadanie1, zadanie2, zadanie3)
```

Przykład użycia `parallel`

```
const gulp = require('gulp');

function compileSass() {
}

function compileJS() {
}

function optimizeImages() {
}

// zadania wykonywane równocześnie,
// nie ma dla nas znaczenia które się
// zakończy jako pierwsze
exports.default = gulp.parallel(
  compileSass,
  compileJS,
  optimizeImages
);
```

Konfiguracja zadań - parallel i series

Przykład użycia `series`

```
const gulp = require('gulp');

function compileSass(done) {
  done();
}

function compileJS(done) {
  done();
}

function reloadBrowser(done) {
  done();
}

// przeładowanie przeglądarki dopiero po
// wykonaniu poprzednich zadań
// dlatego wykonują się jedno po drugim
exports.default = gulp.series(
  compileSass,
  compileJS,
  reloadBrowser
);
```

Gulp w wersji czwartej wprowadza jeszcze jedną metodę: **`series`**. Służy do odpalania kolejnych zadań jedno po drugim.

```
gulp.series(zadanie1, zadanie2, zadanie3)
```

Obie metody możemy wykorzystać też do odpalania zadań z wnętrza innych zadań:

```
exports.default = function() {
  console.log('Starujemy pracę');
  gulp.parallel(
    compileSass,
    compileJS,
    optimizeImages
  );
}
```

Konfiguracja zadań

Wróćmy jednak do skonfigurowania konkretnego zadania. Jak to wygląda w Gulp?

W wersji czwartej funkcja z zadaniem może wyglądać następująco:

```
function taskName(done) {
  gulp.src('./scss/**/*.scss') // pobieramy pliki do pamięci
    .pipe(operacja1) // wykonujemy na nich kilka operacji - np. minimalizację kodu
    .pipe(operacja2)
    .pipe(operacja3)
    .pipe(operacja4)
    .pipe(operacja5)
    .pipe(gulp.dest('...')) // zapisujemy na dysku w określonym miejscu

    done();
}

exports.taskName = taskName; // jeżeli chcemy je wykorzystywać w terminalu
```

Konfiguracja zadań

gulp.src()

Funkcja `gulp.src()` służy do pobrania plików, na których w pamięci będziemy przeprowadzać różne operacje - np. minimalizację kodu, translację, łączenie kilku plików itp. W ścieżce do plików możemy spotkać 2 zapisy:

- `./scss/*/*.scss` - oznacza wszystkie pliki w danym katalogu
- `./scss/**/*.scss` - oznacza wszystkie pliki w danym katalogu i podkatalogach

pipe()

Za pomocą funkcji `pipe()` wykonujemy kolejne czynności na plikach - np. kompilację scss do css:

```
const gulp  = require("gulp");
const sass  = require("gulp-sass");

function taskName(done) {
  //pobieramy pliki do pamięci
  gulp.src('./scss/**/*.scss')
    .pipe(sass())
    ...

  done();
}
```

Konfiguracja zadań

gulp.src()

Po wykonaniu wszystkich operacji wynik końcowy musimy zapisać na dysku. Wykonujemy to za pomocą `gulp.dest()`:

```
//dołączamy paczki do pliku
const gulp = require('gulp');
const sass = require("gulp-sass");

function taskName(done) {
  gulp.src('./scss/**/*.scss') //pobieramy pliki do pamięci
    .pipe(sass())
    .pipe() //inne operacje za pomocą pipe()
    .pipe() //inne operacje za pomocą pipe()
    .pipe(gulp.dest('./css')); //zapisujemy do katalogu css

    done();
}

exports.taskName = taskName;
```

Konfiguracja Gulp

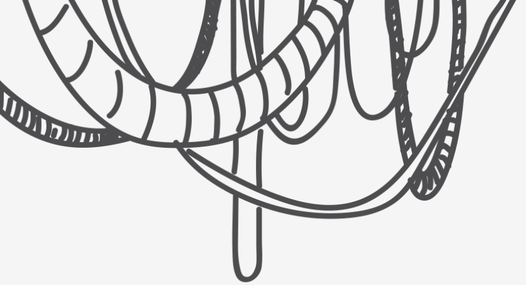
Aby zamienić SCSS na CSS, stworzymy w pliku konfiguracji nowe zadanie. W tym celu musimy zainstalować odpowiednią paczkę.

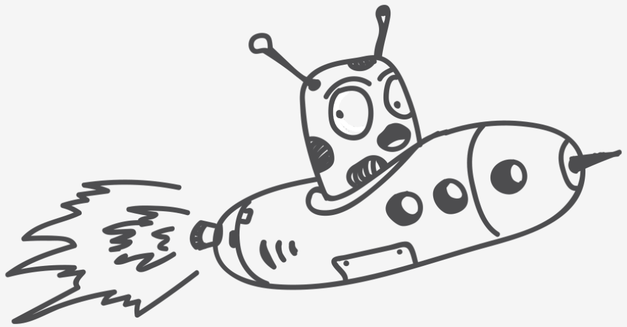
```
npm install sass gulp-sass --save-dev
```

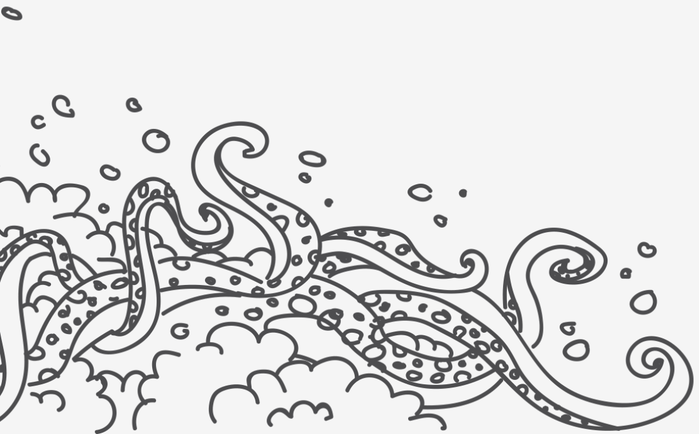
Spójrzmy na kod obok. Pobiera on plik `scss/main.scss` i za pomocą modułu `sass` zamienia go na `css` i zapisuje w katalogu `css`. Aby nasze zadanie działało poprawnie, w głównym katalogu projektu powinniśmy mieć katalog `scss` w którym znajduje się plik `main.scss`.

Nasz kod w pliku `gulpfile.js` będzie wyglądał następująco:

```
const gulp = require('gulp');
const sass = require('gulp-sass');
sass.compiler = require('sass');

function compileSass(done) {
    gulp.src('./scss/main.scss')
    .pipe(sass().on('error', sass.logError))
    .pipe(gulp.dest('./css'));
    done();
}

exports.default = gulp.parallel(compileSass);
```







Formatowanie stylów wyjściowych

Konfiguracja Gulp - Formatowanie

Jak kod będzie wyglądał po kompilacji?

Kod CSS może być wygenerowany w jednym z czterech stylów:

- nested
- expanded
- compact
- compressed

Konfiguracja Gulp - Formatowanie

`nested` - Domyślny

Jest to domyślny styl. Kolejne stylowania są odpowiednio wcięte, co przypomina nieco choinkę. Charakteryzuje się też tym, że komentarze css /* */ nie są usuwane.

```
#main {
   color: #fff;
   background-color: #000; }
   #main p {
      width: 10em; }
.huge {
   font-size: 10em;
   font-weight: bold;
   text-decoration: underline; }
```

Konfiguracja Gulp - Formatowanie

`expanded` - Najbardziej przyjazny

→ Ten styl jest dla nas najbardziej czytelny.

→ Wcięcia są stosowane tylko dla właściwości. Każda z nich wstawiana jest do osobnej linii.

```
#main {
    color: #fff;
    background-color: #000;
}
#main p {
    width: 10em;
}
.huge {
    font-size: 10em;
    font-weight: bold;
    text-decoration: underline;
}
```

Konfiguracja Gulp - Formatowanie

`compact` - Reguły css w jednej linii

- Zajmuje mniej miejsca na ekranie i pozwala skupić większą uwagę na selektorach niż na ich właściwościach.
- Każda deklaracja zajmuje tylko jedną linię.

```
#main { color: #fff; background-color: #000; }
#main p { width: 10em; }
.huge { font-size: 10em; font-weight: bold;
        text-decoration: underline; }
```

Konfiguracja Gulp - Formatowanie

`compressed` - Skompresowany

→ Zajmuje najmniej miejsca, likwiduje możliwe spacje.

→ Styl ma kilka metod kompresji.

→ W tej wersji rozmiar pliku CSS jest najmniejszy.

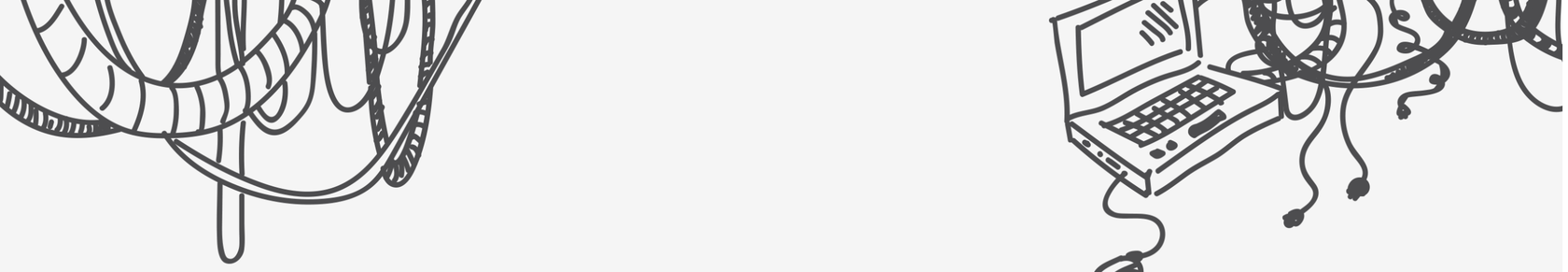
```
#main{color:#fff;background-color:#000}#main p{width:10em}
.huge{font-size:10em;font-weight:bold;text-decoration:underline}
```

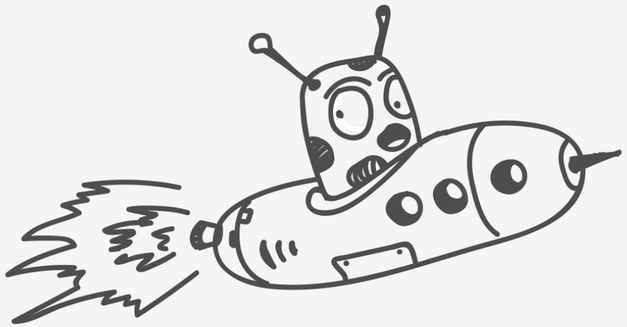
Konfiguracja Gulp - Formatowanie

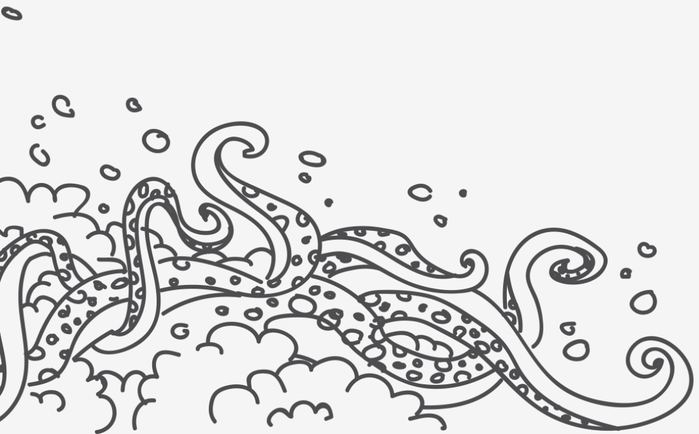
Aby zmienić styl wynikowy kompilacji scss, musimy podać odpowiedni styl w naszym zadaniu.

```
const gulp = require('gulp');
const sass = require('gulp-sass');
sass.compiler = require('sass');

function compileSass(done) {
  gulp.src('./scss/main.scss')
    //nested, expanded, compact, compressed
    .pipe(sass({outputStyle: "expanded"}).on('error', sass.logError))
    .pipe(gulp.dest('./css'));
  done();
}

exports.default = gulp.parallel(compileSass);
```

Jak widzimy, funkcja `sass()` pozwala jako parametr podać prosty obiekt (literał zapisany za pomocą {}), w którym możemy przekazać opcje konfiguracyjne (znajdziemy je na stronie danego pluginu). Opcje możemy przekazywać dla większości paczek, z których będziemy korzystać - oczywiście dla każdej inne.







Mapowanie kodu źródłowego

Konfiguracja Gulp - Mapowanie

- `Sourcemaps` czyli mapy kodu źródłowego informują przeglądarkę o tym, w której linii pliku **Sass** znajduje się źródło wygenerowanej deklaracji **CSS**.
- Znacznie ułatwia to debugowanie i edycję plików, gdyż struktura plików i kodu z reguły różni się między **Sass** a **CSS**.

Aby w gulpie użyć sourcemaps, musimy doinstalować odpowiednią paczkę. Szukamy w necie paczki "`gulp sourcemaps`". Pokaże nam się strona https://www.npmjs.com/package/gulp-sourcemaps na którą wchodzimy. Instalujemy paczkę poleceniem:

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

Konfiguracja Gulp - Mapowanie

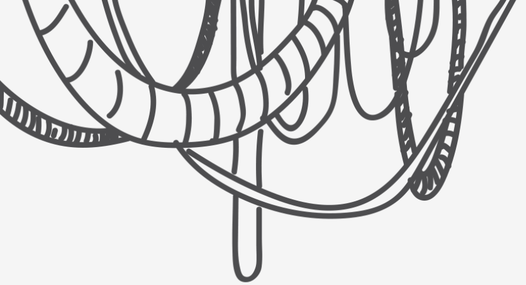
gulpfile.js

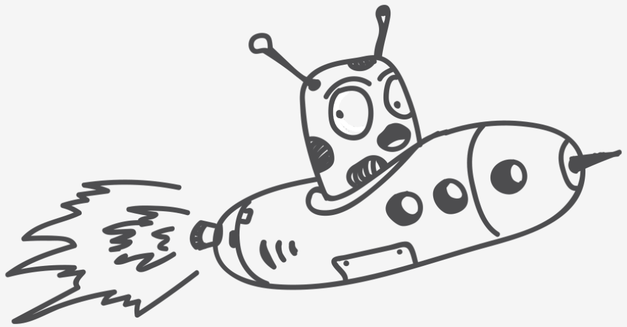
Według instrukcji dołączamy ją do naszego pliku:

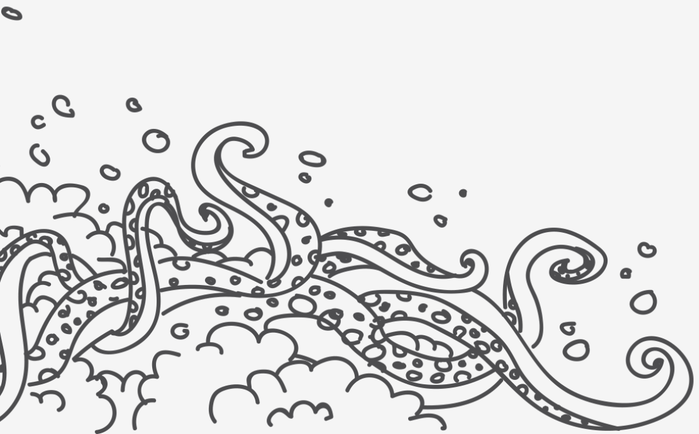
```
const gulp = require('gulp');
const sass = require('gulp-sass');
sass.compiler = require('sass');
const sourcemaps = require('gulp-sourcemaps');

function compileSass(done) {
  gulp.src('./scss/main.scss')
    .pipe(sourcemaps.init()) //włączamy mapę na początku wszystkich operacji
    .pipe(sass({outputStyle: "expanded"}).on('error', sass.logError))
    .pipe(sourcemaps.write()) //i zapisujemy tuż przed zapisem plików wynikowych
    .pipe(gulp.dest('./css'));
  done();
}

exports.default = gulp.parallel(compileSass);
```

Konfiguracja Gulp - Mapowanie

Jeżeli teraz odpalimy nasze zadanie poleceniem gulp, zobaczymy, że w kodzie pliku `css/main.css` pojawi się duży dziwny komentarz. To właśnie rzutowanie kodu źródłowego na wynikowy. Odświeżmy widok naszej strony i spróbujmy jeszcze raz zbadać nagłówek. Debugger tym razem powinien wskazać prawidłowy plik do edycji.







Obserwowanie plików

Konfiguracja Gulp – Watch

W tej chwili nawet najmniejsza zmiana w SCSS zmusza nas za każdym razem do powrotu do terminalu w celu odpalenia zadania kompilacji. Nie jest to wygodne.

Gulp udostępnia nam mechanizm watch (nie trzeba nic doinstalowywać), Po odpaleniu watch działa w tle (w terminalu) i obserwuje wskazane przez nas pliki. Gdy w którymkolwiek z nich zajdzie jakaś zmiana, gulp odpali odpowiednie zadania (w naszym przypadku nasze zadanie kompilacji).

Aby użyć watchera w gulpie, musimy skorzystać z funkcji `gulp.watch`.

```
gulp.watch('./scss/**/*.scss', gulp.series(sassTask));
gulp.watch('./*.html', gulp.series(htmlSuperTask));
```

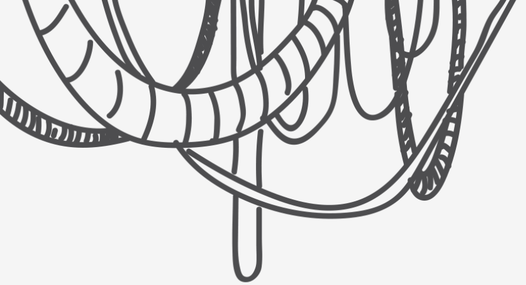
Konfiguracja Gulp – Watch

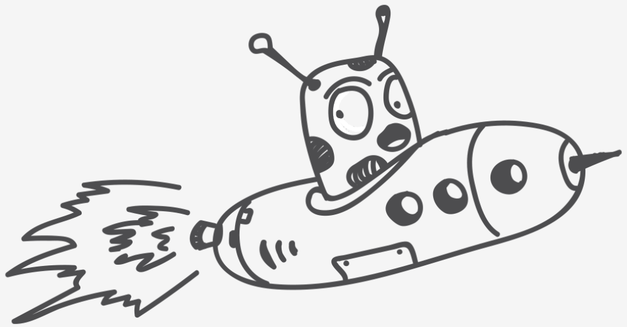
Nie zawsze jednak chcemy odpalać `watch`, dlatego najlepiej zrobić to w osobnym tasku. Dodajmy więc do naszej konfiguracji odpowiednie zadanie, które odpalimy też w domyślnym zadaniu.

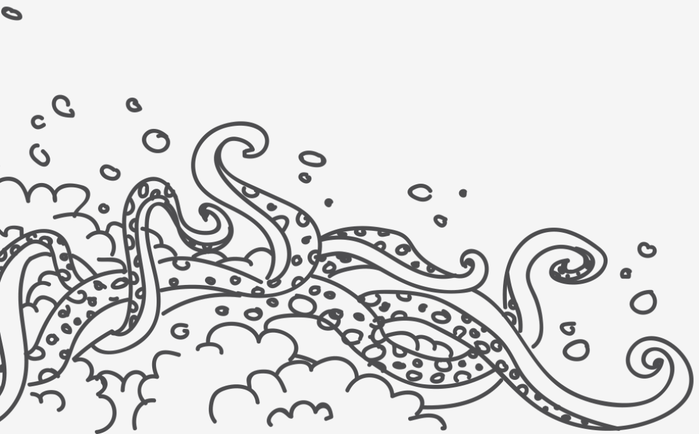
Od tej pory odpalenie domyślnego zadania (w terminalu `gulp`) spowoduje początkową kompilację SCSS, a następnie odpalenie nasłuchiwania na plikach. Jeżeli zajdzie w nich zmiana, zostanie odpalone ponownie zadanie kompilacji scss na css.

Aby przerwać działanie Gulpa, w terminalu naciskamy (najlepiej kilka razy pod rząd) klawisze **CTRL + C**

```
const gulp = require('gulp');
const sass = require('gulp-sass');
sass.compiler = require('sass');
const sourcemaps = require('gulp-sourcemaps');

function compileSass(done) {
  gulp.src('./scss/main.scss')
    .pipe(sourcemaps.init())
    .pipe(sass({outputStyle: "expanded"})
        .on('error', sass.logError))
    .pipe(sourcemaps.write('.'))
    .pipe(gulp.dest('./css'));
  done();
}

function watcher(done) {
  gulp.watch('./scss/**/*.scss', gulp.series(compileSass));
}

exports.sass = gulp.parallel(compileSass);
exports.default = gulp.parallel(compileSass, watcher);
```







Autoprefixer

Konfiguracja Gulp – Autoprefixer

Czym jest autoprefixer? Jest to narzędzie, które pozwala zapomnieć nam o prefiksach dla różnych przeglądarkach. My tylko podajemy jakie przeglądarki interesują nas w danym projekcie, a autoprefixer do wynikowego kodu dodaje nam odpowiednie dopiski.

Mechanizm ten istnieje też dla gulpa. Na stronie https://www.npmjs.com/package/gulp-autoprefixer mamy dokładną instrukcję instalacji. Ponownie - podobnie do poprzednich paczek - musimy wykonać kilka kroków.

Instalujemy autoprefixer poleceniem:

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

Konfiguracja Gulp – Autoprefixer

Oraz dodajemy konfiguracje do naszego pliku `gulpfile.js`.

Autoprefixer dodajemy dopiero po kompilacji, czyli odpaleniu funkcji `sass()`.

Możemy również skonfigurować działanie autoprefixer. Robimy to poprzez odpowiedni obiekt w pliku `package.json`.

Klucz `browserslist` pozwala wskazać jakie przeglądarki nas interesują. Listę przeglądarek możemy znaleźć na stronie: https://github.com/browserslist/browserslist#full-list.

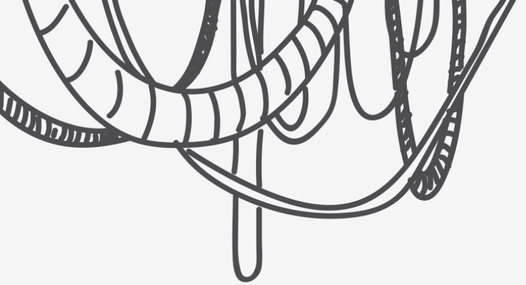
package.json

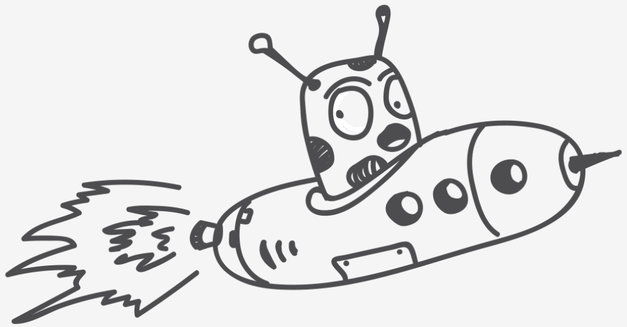
Dodajemy tą parę klucz-wartość na koniec obiektu.

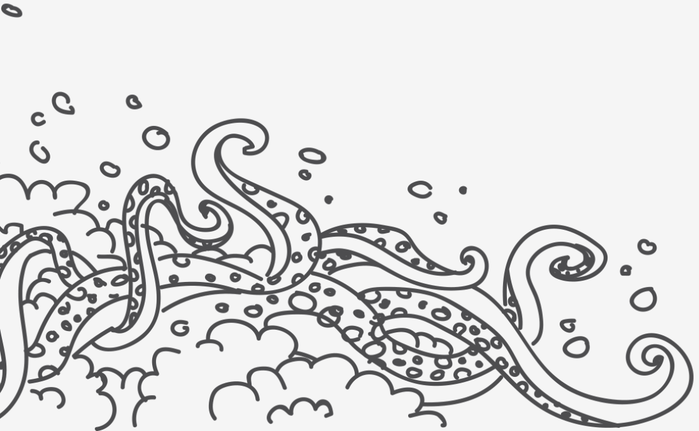
```
"browserslist": [
  "defaults"
]
```

gulpfile.js

```
// Pozostałe biblioteki
const autoprefixer = require('gulp-autoprefixer');

function compileSass(done) {
  gulp.src('./scss/main.scss')
    .pipe(sourcemaps.init())
    .pipe(sass({outputStyle: "expanded"})
        .on('error', sass.logError))
    .pipe(autoprefixer())
    .pipe(sourcemaps.write('.'))
    .pipe(gulp.dest('./css'));
  done();
}

// Reszta konfiguracji
```







BrowserSync

Konfiguracja Gulp - BrowserSync

BrowserSync

Paczka `browser-sync` pozwala nam na automatyczne przeładowywanie strony po wykryciu zmian w różnych plikach.

Dzięki niej, nie musimy ręcznie odświeżać strony nad którą pracujemy. Wtyczka ta tworzy mały lokalny serwer we wskazanej lokalizacji i otwiera kartę przeglądarki dokładne w tym miejscu.

Aby z niej skorzystać, musimy ją zainstalować:

```
npm install browser-sync --save-dev
```

Konfiguracja Gulp - BrowserSync

Następnie musimy zmienić konfigurację naszego pliku `gulpfile.js`. Pierwszą rzeczą jest zaimportowanie biblioteki.

```
const browserSync = require("browser-sync").create();
```

Dodajmy teraz funkcję `reload` która będzie miała za zadanie przeładowywać kartę przeglądarki z naszym uruchomionym zadaniem.

```
function reload(done) {
  browserSync.reload();
  done();
}
```

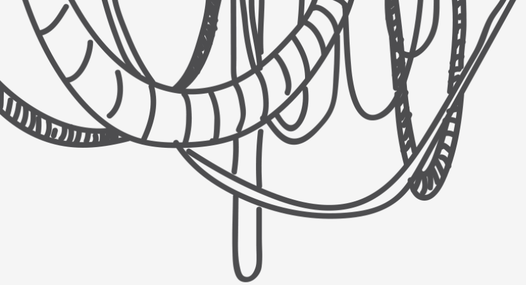
Konfiguracja Gulp - BrowserSync

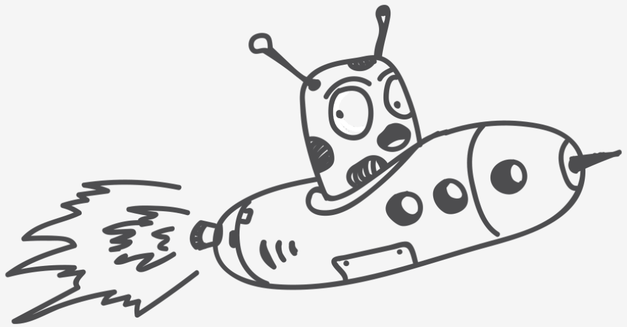
Potrzebujemy również zmodyfikować funkcję `watcher`. Musi ona stworzyć serwer a także przeładowywać stronę kiedy wykryje zmiany w plikach. Dodajmy na początku tej funkcji metodę `browserSync.init()`.

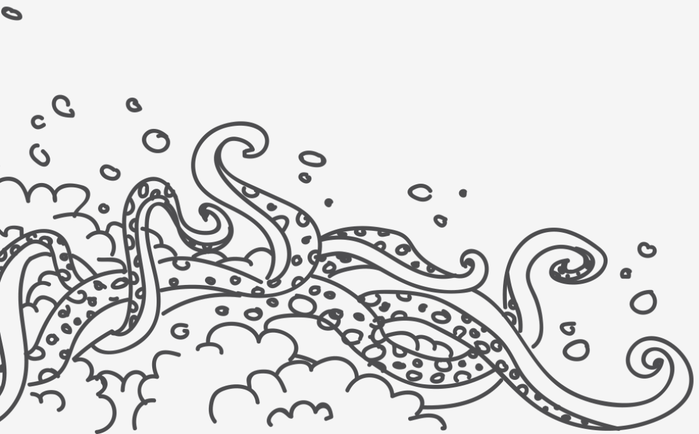
```
browserSync.init({
  server: "./"
});
```

Do serii wykonywanych zadań dodajemy funkcję `reload`. Dodaliśmy również obserwowanie plików `html` aby one również wyzwalały przeładowanie strony w przeglądarce.

```
gulp.watch("./scss/**/*.scss", gulp.series(compileSass, reload));
gulp.watch("./*.html", gulp.series(reload));
```







Konfiguracja podczas zajęć

Konfiguracja podczas zajęć

W naszym repozytorium do tego modułu, zadania są umieszczone w różnych folderach. Każde zadanie posiada własny zestaw plików typu: HTML, SCSS itd.

Nie chcemy instalować dla każdego zadania całego środowiska do kompilacji SCSS -> CSS, dlatego zrobimy to tylko **raz** w głównym katalogu repozytorium.

Wystarczy wpisać `npm install`. Wszystkie pliki konfiguracyjne są już przygotowane.

Jednak można zauważyć, że w pliku `gulpfile.js` pojawiła się zmienna `entryPath`. Jej zadaniem jest wskazanie na **aktualnie wykonywane zadanie** (katalog z tym zadaniem).


```
const entryPath = "Dzień 1/Podstawy/1_Pierwsze kroki z SCSS";
```

Ta wartość zmiennej `entryPath` kieruje nas do folderu: Dzień 1/Podstawy/**1_Pierwsze kroki z SCSS**.

Konfiguracja podczas zajęć

Bardzo ważne jest, że kiedy przechodzisz do wykonywania kolejnego zadania **zmienić wartość zmiennej `entryPath`!**

Wartość tej zmiennej zawsze musi reprezentować ścieżkę do aktualnie wykonywanego zadania.

Po zmianie wartości, należy przerwać działanie Gulpa `CTRL + C` i włączyć go ponownie `gulp`.