Урок 7

Полезные библиотеки, сборщики и шаблонизаторы

Знакомство с полезными расширениями, библиотеками, сборщиками пакетов скриптов.

Таск-менеджеры в JS

Вероятно, вы уже слышали, а может и сталкивались с Grunt. Но давайте проясним теоретическую часть.

Таск-менеджер — это микро-приложение, используемое для автоматизации важных рутинных заданий, выполняемых каждый раз при разработке и разворачивании проекта. Это могут быть такие задачи как: прогон модульных тестов, склейка файлов, минификация или препроцессинг CSS. Создав таск-файл, вы можете задать таск-менеджеру правила того, как следует проводить ту или иную процедуру. Вы освобождаете тем самым своё время и снижаете риск ошибок, обусловленных человеческим фактором.

Gulp – автоматизация процесса разработки

Gulp.js представляет собой потоковый сборщик проектов на JS. Это говорит о том, что инструмент базируется на идее потоковой передачи данных. Благодаря ей, вы имеете больше контроля над процессом, а для слияния вам не понадобятся промежуточные папки и файлы. Происходит просто передача файла в gulp, после чего идёт сохранение результата в выходной файл.

Что же может Gulp?

- Автоматически перезагружать браузер при любых изменениях в файлах проекта (более не надо затирать клавиши Ctrl/Cmd+R).
- Сжимать и склеивать скрипты и стили.
- Проверять скрипты и стили на ошибки.
- Использовать различные препроцессоры — SASS/LESS/Stylus, Jade и так далее.
- Оптимизировать изображения.

Конечно, возможностей больше. Здесь лишь перечислены основные из них.

Помимо базовой функциональности, gulp имеет возможность подключать к себе плагины. Каждый плагин выполняет одно простое действие. Сам же gulp только соединяет и организует эти действия в задачи.

Файл gulp — это прежде всего код, а не файл с настройками. Он соответствует спецификации CommonJS. Gulpfile - аккуратный и читаемый файл, ведь код уже структурирован.

Установка

Для работы Gulp вам потребуется Node.js. Если у вас его ещё нет, то скачайте пакет установки с официального сайта. Если у вас Unix-система, то рекомендуется собирать наиболее новую из исходников, а также NPM. Если используется ОС Windows, то посетите сайт http://nodejs.org или загрузите установочный пакет с расширением .msi из http://nodejs.org/dist/latest/.

В консоли выберите директорию, в которой будете решать задачи. Запустите в ней:

```
npm install node-static
```

Это установит в текущую директорию модуль node-static, который станет автоматически доступным для скриптов из поддиректорий.

Если у вас ОС Windows, и команда не сработала, то, скорее всего, дело в том, что не подключились новые пути. Перезапустите ваш файловый менеджер или консоль.

Проверим корректность установки. Создадим поддиректорию и положим в неё файл server.js с таким содержимым:

```
var http = require('http');
var static = require('node-static');
var file = new static.Server('.');
http.createServer(function(req, res) {
  file.serve(req, res);
}).listen(8080);
console.log('Server running on port 8080');
```

Запустим его:

```
node server.js.
```

```
Server running on port 8080
```

Заметьте, что нельзя запустить более одного сервера в один момент времени. При попытке запуска двух и более серверов произойдет конфликт портов.

Теперь откройте в браузере адрес http://127.0.0.1:8080/server.js. Результатом его работы должен быть код файла server.js.

Теперь можно приступать к установке Gulp.

```
npm install gulp -g
```

Команда npm install вызывает Node Package Manager. Флаг –g говорит о том, что Gulp установится для всех пользователей.

Создание проекта в gulp

Теперь создадим ещё одну папку – project. В ней будет лежать наш проект. Зайдя в неё, запустите следующую команду:

```
npm init
```

Создастся файл package.json. В нём хранится вся информация о нашем проекте. Далее можно установить в сам проект менеджер Gulp.

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

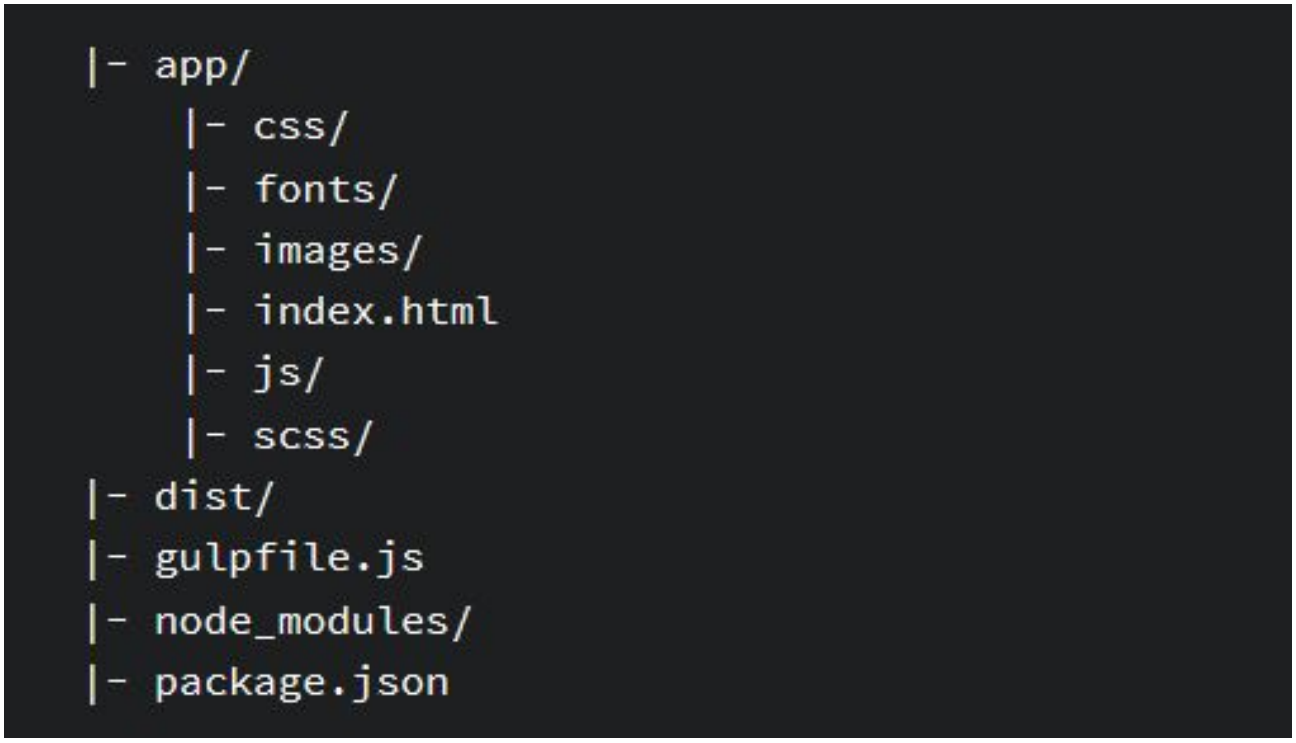
Мы произвели установку только в проект, но не глобально. Ключ —save-dev даёт команду добавить gulp в качестве зависимости в package.json.

В папке project создалась новая папка node_modules, внутри которой есть папка gulp.

Остаётся только определиться, как мы будет применять Gulp в проекте, и какова будет структура папок.

Определяем структуру папок

“Из коробки” Gulp умеет распознавать произвольную структуру папок. Мы применим следующую структуру:



Папка app применяется для разработки, папка dist содержит оптимизированные файлы. Весь программный код, таким образом, хранится в папке app. Начнём с задачи в gulpfile.js - в этом файле хранятся все настройки Gulp.

Пишем первую задачу

Сначала подключаем Gulp:

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

Команда require отдаётся Node.js, который проверяет папку node_modules в поисках папки gulp. При нахождении таковой её содержимое сохраняется в переменную gulp. Теперь написание задач для Gulp доступно через эту переменную.

```
gulp.task('task-name', function() {
  // код
});
```

Task-name – это имя задачи, которое используется в любой момент времени при запуске задач. Задачу можно запустить и через консоль при помощи команды gulp task-name. Давайте создадим задачу hello.

```
gulp.task('hello', function() {
  console.log('Hello, World!');
});
```

Запустим задачу из командной строки.

```
gulp hello
```

Увидим мы примерно следующее:

```
[11:14:36] Starting 'hello'…
Hello, World!
```

Задачи, разумеется, всегда сложнее, чем вышеобозначенная. Как правило, в самой задаче содержится два дополнительных метода Gulp и разные плагины.

```
gulp.task('task-name', function () {
  return gulp.src('source-files')   // получаем источники с помощью gulp.src
    .pipe(aGulpPlugin())            // прогоняем их через плагин
    .pipe(gulp.dest('destination')) // выходные файлы в папке destination
})
```

В примере настоящая задача вызывает два дополнительных метода – gulp.src и gulp.dest. Gulp.src указывает на файл, который будет использоваться для задачи, а gulp.dest указывает на папку, в которую будут складываться файлы по завершению задачи.

Препроцессинг в Gulp

Плагин gulp-sass позволяет компилировать Sass в CSS. Установка плагина очень похожа на установку самого gulp. Следует использовать флаг —save-dev, дабы добавлять плагин в devDependencies в файле package.json.

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

Разумеется, сам плагин нужно подключить в коде с помощью require.

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

Назовём задачу sass-compile, ведь она выполняет компиляцию Sass в CSS.

```
gulp.task('sass-compile', function(){
  return gulp.src('source-files')
    .pipe(sass()) // используем gulp-sass
    .pipe(gulp.dest('destination'))
});
```

Обязательно нужно задать файл источник и целевую папку для результатов. Мы создадим styles.scss в папке app/scss, который добавится в задачу в методе gulp.src. Результатом должен быть styles.css в папке app/css – этот файл мы укажем в параметре gulp.dest.

```
gulp.task('sass-compile', function(){
  return gulp.src('app/scss/styles.scss')
    .pipe(sass()) // Конвертируем Sass в CSS с помощью gulp-sass
    .pipe(gulp.dest('app/css'))
});
```

В качестве теста добавим Sass функцию в наш файл styles.scss.

```
// styles.scss
.testing {
  width: percentage(5/7);
}
```

При запуске в командной строке мы увидим styles.css в папке app/css. При этом функция percentage(5/7) превратится в 71.42857%.

```
/* styles.css */
.testing {
  width: 71.42857%;
}
```

Это указывает на то, что задача работает корректно. Часто требуется компиляция больше чем одного файла .scss в CSS. Здесь уже помогут подстановки Node.

Подстановки в Node

Подстановки – это механизм, который позволяет добавить более одного файла в gulp.src. Он схож с регулярными выражениями. Таким образом, сервер проверяет имена файлов и путей на сходство с шаблоном. Для большинства задач вполне хватает 4 различных моделей:

- *.scss: Символ * совпадает с любым шаблоном в текущей директории. Т.е. мы ищем все файлы с расширением .scss в корневой папке проекта.
- **/*.scss: Этот шаблон ищет файлы с окончанием .scss в корне и всех вложенных папках.
- !not-me.scss: Символ ! указывает, что Gulp не будет включать в результат поиска определённый файл. В нашем примере мы отклоняем файл not-me.scss.
- *.+(scss|sass): Знак + и круглые скобки () создают множественные шаблоны. Сами шаблоны разделяются символом |. В примере мы ищем все файлы с окончанием .scss или .sass в корневой папке проекта.

Теперь можно поменять app/scss/styles.scss на шаблон scss/ **/*.scss. Этот шаблон совпадает с любым файлом .scss в папке app/scss или дочерней директории.

```
gulp.task('sass-compile', function() {
  return gulp.src('app/scss/**/*.scss') // Получаем все файлы с окончанием .scss в папке app/scss и дочерних директориях
    .pipe(sass())
    .pipe(gulp.dest('app/css'))
})
```

Все подходящие Sass файлы автоматически подключаются к задаче. Если мы добавим в проект новый файл print.scss, то в папке app/css по окончанию работы появится файл print.css.

Итак, у нас получилось скомпилировать все файлы одной командой, но ведь мы до сих пор запускаем компиляцию руками каждый раз, когда это необходимо. Это неудобно, но мы можем установить автоматический запуск нашей задачи при обновлении файла при помощи метода «watching».

Контроль изменений

Gulp имеет метод watch – он проверяет файлы на изменения.

```
// Gulp watch
gulp.watch('files-to-watch', ['tasks', 'to', 'run']);
```

При необходимости слежения за всеми Sass файлами и запуске задачи при любом изменении в методе watch нужно заменить files-to-watch на app/scss/**/*.scss, ['tasks', 'to', 'run'] на [‘sass’]:

```
// Gulp watch
gulp.watch('app/scss/**/*.scss', ['sass-compile']);
```

Но зачастую нам требуется следить лишь за несколькими типами файлов. Тогда можно объединить их в методе watch:

```
gulp.task('watch', function(){
  gulp.watch('app/scss/**/*.scss', ['sass-compile']);
// другие ресурсы
})
```

При запуске gulp watch будет видно, что проверка началась сразу же, а задача по компиляции Sass будет запускаться при любых изменениях в файлах .scss.

Можно заставить Gulp обновлять страницу в браузере при любом изменении .scss. В это поможет Browser Sync.

Живая перезагрузка с помощью Browser Sync

Browser Sync помогает поднять сервер, но также он имеет и другие полезные функции - синхронизация действий на множестве устройств, но для начала нужно Browser Sync:

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

Здесь мы не использовали префикс gulp-, т.к. Browser Sync уже работает с Gulp. Нет необходимости в этом префиксе.

Далее подключаем плагин.

```
var browserSync = require('browser-sync');
```

Стартуем сервер в Gulp, выполняя задачу browserSync. После запуска нужно указать корневую папку.

```
gulp.task('browserSync', function() {
  browserSync({
    server: {
      baseDir: 'app'
    },
  })
})
```

Теперь изменим задачу 'sass-compile', чтобы Browser Sync смог добавлять новые стили к странице.

```
gulp.task('sass', function() {
  return gulp.src('app/scss/**/*.scss')
    .pipe(sass())
    .pipe(gulp.dest('app/css'))
    .pipe(browserSync.reload({
      stream: true
    }))
});
```

Теперь Browser Sync настроен. Перезагрузку налету нужно запускать, стартуя одновременно две задачи: watch и browserSync. При этом не нужно запускать две консоли! Нужно просто сказать Gulp, что перед запуском watch должна быть выполнена задача browserSync.

```
gulp.task('watch', ['array', 'of', 'tasks', 'to', 'complete','before', 'watch'], function (){
  // ...
})
```

```
gulp.task('watch', ['browserSync'], function (){
  gulp.watch('app/scss/**/*.scss', ['sass-compile']);
// другие ресурсы
})
```

Теперь обновить версию CSS-файла. Для этого необходимо, чтобы sass-compile запускался перед watch.

```
gulp.task('watch', ['browserSync', 'sass'], function (){
  gulp.watch('app/scss/**/*.scss', ['sass']);
// другие ресурсы
});
```

Теперь при запуске gulp watch sass-compile и browserSync запустятся одновременно. По окончанию выполнения обеих задач запустится watch.

Одновременно с этим откроется окно браузера, а именно страничка app/index.html. Если изменить styles.scss, то страница браузера автоматически обновится.

Есть ещё кое-что, о чём стоит упомянуть, если мы уже обновляем страницу браузера при любых изменениях в .scss, почему бы не делать это при изменениях во всех HTML или JavaScript файлах? Сделать это можно, добавив дополнительные процессы и вызывая функцию browserSync.reload при сохранении файла:

```
gulp.task('watch', ['browserSync', 'sass'], function (){
  gulp.watch('app/scss/**/*.scss', ['sass']);
// Обновляем браузер при любых изменениях в HTML или JS
  gulp.watch('app/*.html', browserSync.reload);
  gulp.watch('app/js/**/*.js', browserSync.reload);
});
```

Итак, мы уже умеем делать следующее:

- поднимать сервер;
- пользоваться препроцессором Sass;
- обновлять браузер при любых изменениях в файлах.

Теперь научимся оптимизировать файлы.

Оптимизация CSS и JavaScript-файлов

Для оптимизации CSS и JavaScript нужно выполнить две задачи: минификацию и конкатенацию. Проблема в том, что конкатенация скрипта должны проходить в правильном порядке следования скриптов,т.е. если в index.html подключены три скрипта:

```
<body>
  <!—- другие ресурсы -->
  <script src="js/lib/a-library.js"></script>
  <script src="js/lib/another-library.js"></script>
  <script src="js/main.js"></script>
</body>
```

То и конкатенировать они должны в том же порядке. Иначе может возникнуть ошибка ненайденных методов, функций и переменных.

Файлы расположены в разных директориях, а потому стандартный плагин gulp-concatenate не подойдёт, поэтому нужен плагин gulp-useref, который решает эту проблему. Gulp-useref объединяет любое количество CSS и JavaScript-файлов в один, используя комментарии: начальный «<!—build:» и конечный «<!—endbuild—>».

```
<!-- build:<type> <path> -->
... HTML разметка, скрипты/ ссылки.
<!-- endbuild -->
```

<type> задаёт тип файла: значения js, css или remove. Предпочтительно устанавливать типы в соответствии с файлами, которые хотите объединить, если установить тип remove, то Gulp удалит весь блок между комментариев.

<path> — путь к результативному файлу. Финальный файл JavaScript должен лежать в папке js с именем main.min.js.

```
<!--build:js js/main.min.js -->
<script src="js/lib/a-library.js"></script>
<script src="js/lib/another-library.js"></script>
<script src="js/main.js"></script>
<!-- endbuild -->
```

Настроим плагин gulp-useref в gulp-файле.

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

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

Создание задачи useref похоже на предыдущие наши задачи. Различие в том, что функция useref.assets() должна вызываться перед gulp.src. Ниже код:

```
gulp.task('useref', function(){
  var assets = useref.assets();
  return gulp.src('app/*.html')
    .pipe(assets)
    .pipe(assets.restore())
    .pipe(useref())
    .pipe(gulp.dest('dist'))
});
```

При запуске useref Gulp пройдётся по трём скриптам и сольёт их в один dist/js/main.min.js. Но сам выходной файл ещё не минифицирован. В этом нам поможет плагин gulp-uglify, который применяется для минификации JS файлов.

```
$ npm install gulp-uglify --save-dev
// другие подключения...
var uglify = require('gulp-uglify');
gulp.task('useref', function(){
  var assets = useref.assets();
  return gulp.src('app/*.html')
    .pipe(assets)
    .pipe(uglify())
    .pipe(assets.restore())
    .pipe(useref())
    .pipe(gulp.dest('dist'))
});
```

При запуске задачи useref Gulp автоматически минифицирует файл `main.min.js`. Плагин gulp-useref объединит скрипты между комментариев «<!—build:» и «<!—endbuild—>» в один файл и поместит весь код в `js/main.min.js`.

Таким же способом соединяются и CSS-файлы.

```
<!--build:css css/styles.min.css-->
<link rel="stylesheet" href="css/styles.css">
<link rel="stylesheet" href="css/another-stylesheet.css">
<!--endbuild-->
```

Задача useref слегка изменится. Теперь функция uglify() нам не нужна. Ставим gulp-if, чтобы создавать вилки логики.

```
$ npm install gulp-if --save-dev
// другие подключения...
var gulpIf = require('gulp-if');
gulp.task('useref', function(){
  var assets = useref.assets();
  return gulp.src('app/*.html')
    .pipe(assets)
// Если JS то запускаем uglify()
    .pipe(gulpIf('*.js', uglify()))
```

```
    .pipe(assets.restore())
    .pipe(useref())
    .pipe(gulp.dest('dist'))
});
```

Для минификации CSS файлов потребуется плагин gulp-minify-css.

```
npm install gulp-minify-css
```

```
var minifyCSS = require('gulp-minify-css');
gulp.task('useref', function(){
  var assets = useref.assets();
  return gulp.src('app/*.html')
    .pipe(assets)
// Минифицируем только CSS файлы
    .pipe(gulpIf('*.css', minifyCSS()))
// Uglifies only if it's a Javascript file
    .pipe(gulpIf('*.js', uglify()))
    .pipe(assets.restore())
    .pipe(useref())
    .pipe(gulp.dest('dist'))
});
```

На выходе мы получаем по одному оптимизированному CSS и JavaScript-файлу при любом запуске задачи useref.

Домашнее задание

1. Написать задачу gulp для сборки проекта. Ведь мы уже используем как минимум несколько файлов JS.
2. Составить скрипт подготовки сайта к релизу.
3. * Научиться создавать CSS-сборки наподобие JS.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. “jQuery для профессионалов” - Адам Фримен