Gulp

LoftSchool
от мыслителя к создателю

Оглавление

Что такое Gulp

Gulp – это инструмент сборки веб-приложения, позволяющий автоматизировать рутинные, повторяющиеся задачи, такие как сборка и минификация CSS- и JS-файлов, запуск тестов, перезагрузка браузера и т.д. При помощи Gulp можно написать любой другой конфиг (конфигурационный файл, в котором прописаны настройки), поэтому справедливо будет сказать, что Gulp - это система для описания произвольного вида задач.

Задачи, которые помогает решить Gulp

- **Создание веб-сервера и автоматическая перезагрузка страницы** в браузере при сохранении кода, **слежение** за изменениями в файлах проекта;
- Использование различных JavaScript, CSS и HTML **препроцессоров** (CoffeeScript, Less, Sass, Stylus, Jade и т.д.);
- **Минификация** CSS и JS кода, а также, **оптимизация и конкатенация** отдельных файлов проекта в один;
- Автоматическое создание **вендорных префиксов** (приставок к названию CSS свойства, которые добавляют производители браузеров для нестандартных свойств) для CSS.
- **Управление файлами и папками в рамках проекта** - создание, удаление, переименование;
- **Работа с изображениями** - оптимизация, создание спрайтов;
- Создание различных карт проекта и автоматизация другого **ручного труда**.
- **Деплой** - отправка на внешний сервер.

Gulp имеет большое количество плагинов, которые можно найти на странице со списком плагинов, или через поиск по пакетам npm.

Также есть черный список плагинов, не рекомендованных к использованию по тем или иным причинам.

Gulp или Grunt

Grunt - это уже устаревший инструмент для описания различных задач, который использовался до Gulp. Но стоит отметить, что его еще продолжают использовать на многих проектах в силу тех или иных причин.

Для того чтобы увидеть разницу, давайте рассмотрим два конфигурационных файла, написанных на Grunt и Gulp. Задача, которую мы создадим, реализует компиляцию Sass-файлов, добавляет вендорные префиксы при помощи инструмента Autoprefixer. Также установим вотчер для слежки за изменениями файлов.

gruntfile.js
gulpfile.js

Внимательно изучив **gruntfile.js,** становится понятно, как Grunt реализует цепочку управления:

- Первый плагин берет файлы, что-то с ними делает и записывает результат выполнения во временную директорию.
- Следующий плагин читает файлы из временной директории, работает с ними, записывает результат. И так далее пока последний плагин не запишет файлы в итоговое место.

То есть Grunt выполняет лишние операции с постоянным чтением и записью файлов во временную директорию.

Gulp поступает иным образом. Он использует виртуальную файловую систему vinyl-fs.
С ее помощью можно единожды прочитать файл, создать из него объект с необходимыми данными и передавать его от плагина к плагину.

Вывод: одна и та же задача на Gulp будет выполняться быстрее за счет экономии времени на дисковых операциях.

Установка

Для работы Gulp, в первую очередь, необходимо установить node.js. Затем давайте перейдем в заранее созданную директорию с проектом и создадим файл package.json при помощи команды npm init, которую нужно выполнить в терминале, находясь в директории с проектом.

Мы будем использовать **Gulp4**. На данный момент он еще не вошел в релиз, но уже достаточно стабилен и удобен. Для того, чтобы была доступна сама команда gulp нам нужно установить **gulp-cli** (интерфейс командной строки для Gulp) глобально.

```
npm i –g gulp–cli
```

Также gulp нужно установить локально в проект, т.к. мы будем использовать его и подключать в конфигурационный файл через require.

```
npm install gulpjs/gulp#4.0 ––save–dev
```

Для пользователей Mac и Linux, возможно, придется выполнять команды с правами суперпользователя, **sudo**.

Задачи в Gulp

Если сейчас мы запустим команду **`gulp`** в терминале, то увидим ошибку, так как мы еще не создали файл **gulpfile.js**. Это конфигурационный файл, в котором собраны описания различных задач проекта.
Создадим в корневой директории проекта **gulpfile.js** и опишем первую задачу.

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

gulp.task('default', function(callback) {
  console.log('Hello world');
  callback();
});
```

Если теперь выполнить команду **`gulp`** - мы увидим в консоли сообщение **“`Hello world`”**.

При запуске команды **`gulp`** без параметров подразумевается, что выполнится задача со специализированным зарезервированным словом **default**.
Если запустить команду **`gulp`** и передать параметр, то этот параметр — имя задачи, которая описана в **gulpfile.js**.

Метод gulp.task принимает два параметра. Первый - имя задачи, второй - функцию, которая выполняет задачу.

Зачем в описанной таске используется callback? Для того, чтобы сигнализировать о завершении задачи. Есть четыре основных способа:

1. Вызов callback функции, как в примере выше
2. Возвратить промис
3. Возвратить поток
4. Возвратить дочерний процесс

Как правило, волноваться об этом не стоит, так как **в большинстве случаев вы всегда будете возвращать поток.**

Для выборки файлов используется метод gulp.src. Первым параметром он принимает строку или массив строк, которые - ни что иное, как пути до обрабатываемых в таске файлов. Вторым - объект настроек.

Описывая пути можно использовать специальные паттерны. За обработку которых отвечает модуль minimatch. Например:

```
gulp.src('dev/**/*.*');
gulp.src('dev/**/*.js');
gulp.src('dev/img/*.{png,jpg,gif}');
gulp.src('{dev1,dev2}/**/*.js');
gulp.src(['dev/style/*.sass', '!dev/style/_*.sass']);
```

Еще один метод, который нам сейчас понадобится - это **gulp.dest**, он будет принимать поток файлов и записывать их в указанную директорию. Подробнее о нем можно прочитать в документации.

Напишем простую задачу “copy”, которая будет копировать файлы js из одной директории в другую:

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

gulp.task('copy', function(){
  return gulp.src('dev/*.js')
      .pipe(gulp.dest('build/js'));
});
```

И выполним ее в терминале:

```
gulp copy
```

Вот что произойдет:

- **gulp.src** найдет все файлы в директории dev с расширением .js создаст из них специальные объекты vinyl и передаст дальше по потоку.
- метод **pipe** примет поток и передаст его в gulp.dest.

- **gulp.dest** запишет все файлы из потока в директорию build/js

Зачем нужен return? Как раз для того, чтобы сигнализировать о завершении задачи. Как мы говорили выше, один из способов сказать о том, что задача закончена - это вернуть поток. Если не поставить return, задача никогда не завершится.

Теперь мы можем описать задачу, которая будет не только копировать файлы, но и как-то их преобразовывать. Например, объединять все js-файлы в один, обфусцировать, минифицировать и переименовывать итоговый файл.

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');
var concat = require('gulp-concat');
var rename = require('gulp-rename');

gulp.task('copy', function(){
  return gulp.src('dev/*.js')
      .pipe(concat('app.js'))
      .pipe(uglify())
      .pipe(rename({
        suffix: '.min'
      }))
      .pipe(gulp.dest('build/js'));
});
```

Последовательное и параллельное выполнение задач

Нам нужно создать таск, который будет объединять в себе несколько заранее созданных задач.
Для реализации такого функционала в Gulp версии ниже четвертой используется модуль run-sequence.

Теперь же появились стандартные методы, такие как gulp.series и gulp.parallel.

Рассмотрим пример:

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

gulp.task('clean',  function() { /* just code */ });
gulp.task('sass',   function() { /* just code */ });
gulp.task('js',     function() { /* just code */ });
gulp.task('jade',   function() { /* just code */ });
gulp.task('watch',  function() { /* just code */ });

gulp.task('default', gulp.series(
    'clean',
    gulp.parallel(
        'sass',
        'js',
        'jade'
    ),
    'watch'
));
```

Вот что произойдет, после запуска команды **`gulp`**:

- Начнет свое выполнение задача **“clean”**.
- После выполнения задачи “clean”, одновременно запустятся задачи **“sass”, “js” и “jade”**.
- После того как предыдущие три задачи выполнились, запустится задача **“watch”**.

Как правило, задача “watch” не завершается. Gulp начнет следить за файлами в указанной директории. Поэтому важно помнить, что при последовательном выполнении задач, “watch“ нужно запускать **всегда последним**. Иначе следующие за ней таски никогда не выполнятся, т.к. никогда не закончится выполнении таски “watch”.

Автоматическая пересборка. Watch

Мы хотим запускать задачу каждый раз, когда в отслеживаемом файле или директории что-то изменилось. Для этого используется метод gulp.watch

Опишем задачу, которая будет запускать слежение:

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

gulp.task('watch', function(){
  gulp.watch('dev/style/**.*.sass', gulp.series('template'));
  gulp.watch('dev/js/**.*.js', gulp.series('js'));
  gulp.watch('dev/template/**.*.jade', gulp.series('template'));
});
```

После запуска этой задачи Gulp начнет следить за файлами, переданными первым параметром. Если в них произойдет изменение, то Gulp автоматически запустит соответствующий таск.

В **gulp.series** через запятую можно передать несколько тасков. Можно использовать **gulp.parallel.** В этом плане ограничений нет.

Обработка ошибок. Нотификация

В процессе выполнения задачи, плагины могут генерировать ошибки. Например, когда gulp-sass или gulp-jade получают невалидный код. И если в этот момент работает запущенный вотчер, то из-за ошибки он может остановить свою работу.

Потоки node.js устроены таким образом, что если генерируется ошибка на которую нет обработчика, то node.js-процесс полностью останавливается.

Gulp сам умеет обрабатывать ошибки. Он использует встроенную библиотеку async-done, которая смотрит за разными сигналами завершения задачи. Как мы говорили выше, это могут быть callback, промис, поток и т.д. Эта библиотека так же оборачивает вызовы в специальные конструкции, благодаря которым и ловятся ошибки. Но бывают ситуации, когда у них не получается поймать ошибку и, в этом случае, процесс, конечно, упадет.

Далее мы разберем два момента:

1. Как поставить обработчик ошибок, чтобы Gulp не останавливал свою работу в любых ситуациях.
2. Как сделать уведомления об ошибках более удобными.

Рассмотрим следующий код:

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

gulp.task('sass', function(){
  return gulp.src('style/*.sass')
  .pipe(sass())
  .pipe(gulp.dest('build/style/'));
});

gulp.task('watch', function(){
  gulp.watch('style/*.sass', gulp.series('sass'));
});

gulp.task('default', gulp.series('sass', 'watch'));
```

У нас есть задача с именем **“sass”**, которая берет все sass-файлы из директории style, компилирует их при помощи плагина gulp-sass и кладет в директорию build/style.
Есть задача **“watch”**. Она следит за sass-файлами в директории style и при каждом изменении файла из этой директории запускает таску “sass”.
В конце описана дефолтная задача, которая последовательно запускает сначала “sass” и после того как она отработает запускает “watch”.

Запустим задачу default при помощи команды **`gulp`** - все будет работать корректно. Каждый раз, когда мы сохраняем очередные изменения в sass-файлах, “watch” перезапустит таску с именем “sass”.
Но если мы сохраним файл с ошибкой, которую Gulp не сможет обработать, работа Gulp завершится.

Далее изменяться будет только код задачи **“sass”**, поэтому показан будет только он.

```
gulp.task('sass', function () {
    return gulp.src('style/*.sass')
    .pipe(sass())
    .on('error', function (error) {
        console.log(error);
        this.end();
        })
    .pipe(gulp.dest('build/style/'));
});
```

При помощи метода **.on** мы повесили обработчик, теперь если в предыдущем звене цепочки появится ошибка, она будет поймана и процесс работы **не прервется**, а в консоль выведется объект ошибки.
Обратите внимание, что обработчик нужно поставить **после звена**, в котором предполагается ошибка.

Есть еще один нюанс. Потоки node.js устроены так, что при ошибке, pipe перестает передавать данные и процесс повисает. Таким образом, последний поток **.pipe(gulp.dest)** никогда не завершится, т.к. вся обработка остановилась.

Когда обработчика нет, то ошибка ловится библиотекой async-done и Gulp понимает, что задача завершилась. Но если ошибку мы ловим сами, при помощи метода **.on**, то хорошим тоном является просигнализировать об окончании обработки при помощи **this.end()**

Но как нам сделать сообщения об ошибках более информативными и удобными? Можно использовать плагины нотификации. Мы будем работать с gulp-notify.

Посмотрим на следующий код:

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

gulp.task('sass', function(){
    return gulp.src('style/*.sass')
    .pipe(sass())
    .on('error', notify.onError())
    .pipe(gulp.dest('build/style/'));
});
```

Теперь ошибка обрабатывается плагином **gulp-notify**. В консоль будет выводиться аккуратное сообщение об ошибке, а также всплывать системное окошко уведомлений.

Для того, чтобы системное уведомление работало в Windows, в документации есть специальное примечание.

Метод **notify.onError** принимает функцию для форматирования. С ее помощью мы можем сделать уведомление еще более информативным.

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

gulp.task('sass', function(){
    return gulp.src('style/*.sass')
    .pipe(sass())
    .on('error', notify.onError(function(error) {
        return {
            title: 'Styles',
            message: error.message
            };
        }))
    .pipe(gulp.dest('build/style/'));
});
```

С остальными возможностями плагина gulp-notify можно ознакомиться в документации.

Давайте пойдем еще дальше. Текущий обработчик будет ловить ошибку, сгенерированную плагином gulp-sass. Но ошибки могут появиться в любом звене цепочки пайпов. Для решения этой задачи можно на каждом потоке повесить свой обработчик **.on('error')**. Но это не самое красивое решение и так обычно не поступают.

Хорошим решением будет использование плагина gulp-plumber

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

gulp.task('sass', function(){
    return gulp.src('style/*.sass')
    .pipe(plumber())
    .pipe(sass())
    .pipe(gulp.dest('build/style/'));
});
```

gulp-plumber модифицирует поток, встраивая в него специальный обработчик ошибок. Благодаря этому можно один раз повесить функцию обработки, которая сработает на ошибке, сгенерированной **в любом из пайпов ниже**. Таким образом gulp-plumber должен располагаться **в самой вершине** цепочки пайпов.

По умолчанию ошибка будет выводится в консоль. Но можно объединить оба рассмотренных метода и получить следующий код:

```
var plumber = require('gulp-plumber');
var notify = require('gulp-notify');

gulp.task('sass', function(){
    return gulp.src('style/*.sass')
    .pipe(plumber({
        errorHandler: notify.onError(function(error) {
            return {
                title: 'Styles',
                message: error.message
            };
        }))
    .pipe(sass())
    .pipe(gulp.dest('build/style/'));
});
```

Теперь обрабатываются ошибки из любого пайпа, а нотификация реализована красиво и удобно.

Работа с browser-sync

Browser-sync - это инструмент, который позволяет производить автоматическое отображение приложения (проекта) в браузере и перезагрузку при внесении изменений в код. Причем эти изменения могут вносится и отслеживаться синхронно в нескольких браузерах или устройствах.

Пример использования :

```
var gulp = require('gulp');
var browserSync = require('browser-sync').create();
var sass = require('gulp-sass');

// Запускаем сервер
gulp.task('serve', gulp.series('sass'), function () {
    browserSync.init({
        server: "./app"   // Базовая директория
    });
browserSync.watch('./app/ * */*.*').on('change',
browserSync.reload); // Отслеживаем изменения и передаем на клиент
 });

 // Компилируем sass в css
 gulp.task('sass', function() {
 return gulp.src("app/scss/main.scss")
 .pipe(sass())
 .pipe(gulp.dest("app/css"));
 });

 gulp.task('watch', function(){ // Отслеживаем изменения
 gulp.watch('app/scss/app/scss/*.scss', gulp.series('sass'));
 });

 gulp.task('default', gulp.series('sass'), gulp.parallel('watch',
'serve')); // Задача по умолчанию
```

Загрузка gulp-плагинов

Давайте посмотрим, как обычно мы экспортируем gulp-плагины для использования их в описании задачи. Как пример - таска, которая компилирует файлы с последующей минификацией при помощи плагина gulp-csso.

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

gulp.task('sass', function(){
    return gulp.src('style/*.sass')
        .pipe(sass())
        .pipe(csso())
        .pipe(gulp.dest('build/style/'));
});
```

Для работы этой задачи нам нужно подключить три модуля. Сам Gulp, плагин компиляции и плагин минификации. Это происходит в первых трех строчках примера, при помощи метода **require**.

Таким образом, чтобы реализовать таску, нам нужно получить все модули, от которых она зависит.

Теперь представим, что задача зависит от множества gulp-плагинов. В таком случае, нам придется получить каждый модуль при помощи метода require. Это не самое красивое решение и организовать подключение плагинов можно гораздо удобнее. В этом нам поможет gulp-load-plugins.

Посмотрим на следующий пример:

```
var gulpLoadPlugins = require('gulp-load-plugins');
var gulpPlugins = gulpLoadPlugins();
```

Первой строкой мы экспортируем gulp-load-plugins, как любой другой модуль. На следующей - вызываем сохраненную в переменную gulpLoadPlugins, функцию, которая, в свою очередь, возвращает объект со всеми gulp-плагинами, добавленными в package.json.

Рассмотрим следующий package.json. Тут нас интересует объект devDependencies и его поля gulp-sass и gulp-csso.

```
{
  'name': 'ls-builder',
  'version': '0.0.1',
  'devDependencies': {
  'gulp': 'github:gulpjs/gulp#4.0',
      'gulp-sass': '2.3.1',
      'gulp-csso': '2.0.0'
  }
}
```

Модуль gulp-load-plugins будет просматривать в package.json все зависимости нашего проекта и искать gulp-плагины при помощи паттернов:

```
gulp-*
gulp.*
```

При совпадении, часть паттерна отбрасывается “gulp-” или “gulp.” и получается имя плагина, по которому мы будем к нему обращаться. Если название плагина состоит из нескольких слов, то итоговое имя будет в CamelCase нотации.

Примеры:

```
gulp-sass → sass
gulp-replace-task → replaceTask
gulp-some-very-long-name → someVeryLongName
```

Теперь когда мы выполним код:

```
var gulpLoadPlugins = require('gulp-load-plugins');
var gulpPlugins = gulpLoadPlugins();
```

То к плагинам gulp-sass и gulp-csso можно будет обращаться таким образом:

```
gulpPlugins.sass();
gulpPlugins.csso();
```

Вся задача целиком выглядит так:

```
var gulp = require('gulp-gulp');
var gulpLoadPlugins = require('gulp-load-plugins');
var gulpPlugins = gulpLoadPlugins();


gulp.task('sass', function(){
    return gulp.src('style/*.sass')
        .pipe(gulpPlugins.sass())
        .pipe(gulpPlugins.csso())
        .pipe(gulp.dest('build/style/'));
});
```

С полной функциональностью модуля gulp-load-plugins можно ознакомиться в документации.

Перечень рекомендуемых плагинов

gulp-inject	Вставляет указанные файлы или ссылки на них в пределах специальных указателей в целевых файлах. В базе поддерживает вставку в файлы html, jade, jsx , less, slm, haml, sass / scss
gulp-rename	Переименовывает файлы
gulp-filter	Фильтрует файлы по выбранному шаблону
gulp-if	Управление потоком с помощью условий
gulp-chmod	Изменяет права доступа для выбранных файлов
gulp-connect-php	Запускает php сервер
run-sequence	Запускает последовательность тасков в определенном порядке. Применяется тогда, когда нет возможности использовать зависимости.
gulp-autoprefixer	Добавляет css-префиксы
gulp-changed	Отслеживает, какие именно файлы в потоке были изменены
gulp-concat	Выполняет конкатенацию файлов
gulp-csso	Выполняет не только минификацию css, но и оптимизацию структуры файла
gulp-load-plugins	Автоматически загружает любой плагин в файл package.json, добавляя его в глобальную область видимости, как метод объекта и позволяет производит над ним определенные действия
gulp-sourcemaps	Поддержка работы с sourcemaps

gulp-uglify	Минификация js-файлов
gulp-plumber	Предотвращает остановку потока при возникновении ошибок, генерируемых плагинами
gulp-iconfont	Создает иконочный шрифт из нескольких SVG файлов
gulp-responsive	Генерирует адаптивные изображения под требуемые разрешения устройств с указанием соответствующих префиксов в наименовании.
gulp-sharp	Работа с JPEG, PNG, WebP и TIFF изображениями. Плагин умеет изменять размер, ориентацию, фон, альфа-канал и многое другое.
gulp-imagemin	Сжатие изображений
spritesmith	Утилита для создания спрайтов
gulp-replace	Потоковая замена строк по шаблону
browser-sync	Потоковое отображение проекта в браузере при изменении исходных файлов
gulp-sass	Работа с sass
gulp-jade	Работа с шаблонизатором jade \ pug
gulp-notify	Вывод ошибок при выполнении тасков
del	Удаляет файлы и папки
gulp-add-src	Добавляет больше ‘src’ путей к файлам в любом месте pipe
gulp-eslint	Работа с eslint
gulp-ssh	Работа с ssh
gulp-zip	Архивация файлов
gulp-exec	Позволяет запускать Shell команды

gulp-flatten	При копировании вендорных файлов в папку проекта Удаляет относительные пути
gulp-jshint	Проверка правильности js кода
gulp-compass	Компилирование SCSS в CSS с использованием Compass
gulp-size	Отображает рамеры проекта
gulp-useref	Парсит специальные блоки в html и подставляет в указанное место файлы скриптов или стилей
gulp-babel	Пакет для компиляции ES2015 в ES5