4 Системы построения фронтэнда

В современной веб-разработке Node все чаще используется для запуска инструментов и сервисов, от которых зависят фронтэнд-разработчики. Возможно, вам как Node-программисту придется отвечать за настройку и сопровождение этих инструментов. А как полностековому разработчику вам стоит использовать эти инструменты для создания более быстрых и надежных веб-приложений. В этой главе вы научитесь использовать сценарии npm, Gulp и webpack для построения проектов, удобных в сопровождении.

Преимущества от использования систем построения фронтэнда могут быть огромными. Такие системы помогают писать более понятный и устойчивый к будущим изменениям код. Нет необходимости беспокоиться о поддержке браузером ES2015, если приложение обрабатывается транспилятором Babel. Кроме того, благодаря возможности генерирования карт исходного кода, сохраняется возможность отладки на базе браузера.

В следующем разделе приводится краткое введение во фронтэнд-разработку с использованием Node. После этого будут рассмотрены примеры современных технологий — например, React, которые вы сможете использовать в своих проектах.

4.1. Фронтэнд-разработка с использованием Node

В последнее время как разработчики фронтэнда, так и разработчики кода на стороне сервера стали применять npm для передачи JavaScript. Это означает, что npm используется как для модулей фронтэнда (таких, как React), так и для серверного кода (например, Express). Однако некоторые модули трудно четко отнести к той или иной стороне: lodash — пример библиотеки общего назначения, которая может использоваться Node и браузерами. При тщательной упаковке lodash один модуль

может использоваться как Node, так и браузерами, а зависимостями в проекте можно будет управлять при помощи npm.

Возможно, вам уже встречались другие модульные системы, предназначенные для разработки на стороне клиента, — такие, как Bower (*http://bower.io/*). Никто не запрещает вам использовать их, но как разработчику Node вам стоит подумать об использовании npm.

Впрочем, распространение пакетов — не единственная область применения Node. Фронтэнд-разработчики все чаще полагаются на средства создания портируемого кода JavaScript с обратной совместимостью. Транспиляторы — такие, как Babel (*https://babeljs.io/*), — используются для преобразования современного кода ES2015 в более широко поддерживаемый код ES5. Среди других средств можно упомянуть минификаторы (например, UglifyJS; *https://github.com/mishoo/UglifyJS*) и инструменты статического анализа (например, ESLint; i) для проверки правильности кода перед его распространением.

Node также часто управляет системами запуска тестов. Вы можете запускать тесты для UI-кода в процессе Node или же использовать сценарий Node для управления тестами, выполняемым в браузере.

Также эти средства довольно часто используются вместе. Когда вы начинаете жонглировать траспилятором, минификатором, программой статического анализа и системой запуска тестов, вам нужно будет каким-то образом зафиксировать, как работает процесс построения. В одних проектах используются сценарии npm; в других — используют Gulp или webpack. В этой главе мы рассмотрим все эти подходы, а также некоторые практические приемы, связанные с ними.

4.2. Использование npm для запуска сценариев

Node поставляется с npm, а в npm существуют встроенные средства для запуска сценариев. Следовательно, вы можете рассчитывать на то, что коллеги или пользователи смогут выполнять такие команды, как `npm start` и `npm test`. Чтобы добавить собственную команду для `npm start`, включите ее в свойство `scripts` файла `package.json` своего проекта:

```
{
  ...
  "scripts": {
    "start": "node server.js"
  },
  ...
}
```

Даже если не определять `start`, значение `node server.js` используется по умолчанию; строго говоря, вы можете оставить его пустым — только не забудьте создать файл

с именем server.js. Также полезно определить свойство `test`, потому что вы можете включить свой тестовый фреймворк как зависимость и запустить его командой `npm test`. Предположим, вы используете для тестирования фреймворк Mocha (*www.npmjs.com/package/mocha*) и установили его командой `npm install --save-dev`. Чтобы вам не приходилось устанавливать Mocha глобально, вы можете добавить следующую команду в файл package.json:

```
{
  ...
  "scripts": {
    "test": "./node_modules/.bin/mocha test/*.js"
  },
  ...
}
```

Обратите внимание: в предыдущем примере Mocha передаются аргументы. Также при запуске сценариев npm можно передать аргументы через два дефиса:

```
npm test -- test/*.js
```

В табл. 4.1 приведена сводка некоторых команд npm.

Таблица 4.1. Команды npm

Команда	Свойство package.json	Примеры использования
start	scripts.start	Запуск сервера веб-приложения или приложения Electron
stop	scripts.stop	Остановка веб-сервера
restart		Последовательное выполнение команд stop и start
install, postinstall	scripts.install, scripts.postinstall	Выполнение собственных команд построения после установки пакета. Обратите внимание: postinstall может выполняться только в команде npm run postinstall

Поддерживаются многие команды, включая команды очистки пакетов перед публикацией и команды миграции между версиями пакетов. Впрочем, для большинства задач веб-разработки вам хватит команд `start` и `test`.

Многие задачи, которые вы будете определять, не укладываются в поддерживаемые имена команд. Например, предположим, что вы работаете над простым проектом, написанным на ES2015, и хотите транспилировать его на ES5. Это можно сделать командой `npm run`. В следующем разделе рассматривается учебный пример, в котором будет создан новый проект для построения файлов ES2015.

4.2.1. Создание специализированных сценариев npm

Команда `npm run` (синоним для `npm run-script`) используется для определения произвольных сценариев, которые запускаются командой `npm run` *имя-сценария*. Посмотрим, как создать такой сценарий для построения сценария на стороне клиента с использованием Babel.

Создайте новый проект и установите необходимые зависимости:

```
mkdir es2015-example
cd es2015-example
npm init -y
npm install --save-dev babel-cli babel-preset-es2015
echo '{ "presets": ["es2015"] }' > .babelrc
```

В результате выполнения этих команд должен быть создан новый проект Node с базовыми инструментами и плагинами Babel для ES2015. Затем откройте файл package.json и добавьте в раздел `scripts` свойство `babel`.

Оно должно выполнять сценарий, установленный в папке node_modules/.bin проекта:

```
"babel": "./node_modules/.bin/babel browser.js -d build/"
```

Ниже приведен пример файла в синтаксисе ES2015, который вы можете использовать; сохраните его в файле browser.js:

```
class Example {
  render() {
    return '<h1>Example</h1>';
  }
}
const example = new Example();
console.log(example.render());
```

Чтобы протестировать этот пример, введите команду `npm run babel`. Если все было настроено правильно, должна появиться папка построения с файлом browser.js. Откройте browser.js и убедитесь в том, что это действительно файл ES5 (поищите конструкцию вида `var _createClass` в начале файла.

Если ваш проект при построении ничего более не делает, ему можно присвоить имя `build` вместо `babel` в файле package.json. Но можно пойти еще дальше и добавить UglifyJS:

```
npm i --save-dev uglify-es
```

UglifyJS вызывается командой `node_modules/.bin/uglifyjs`; добавьте эту команду в package.json из раздела `scripts` с именем `uglify`:

```
./node_modules/.bin/uglifyjs build/browser.js -o build/browser.min.js
```

Теперь вы сможете выполнить команду `npm run uglify`. Всю функциональность можно связать воедино, объединив оба сценария. Добавьте еще одно свойство `script` с именем `build` для вызова обеих задач:

```
"build": "npm run babel && npm run uglify"
```

Оба сценария запускаются командой `npm run build`. Участники вашей команды могут объединить несколько средств упаковки клиентской части вызовом этой простой команды. Такое решение работает, потому что Babel и UglifyJS могут выполняться как сценарии командной строки, они получают аргументы командной строки и легко добавляются как однострочные команды в файл package.json. Также Babel позволяет определить сложное поведение в файле `.babelrc`, что было сделано ранее в этой главе.

4.2.2. Настройка средств построения фронтэнда

В общем случае возможны три варианта настройки средств построения фронтэнда при использовании со сценариями npm:

- передача аргументов командной строки (например, `./node_modules/.bin/uglify --source-map`);
- создание конфигурационного файла для конкретного проекта. Часто применяется для Babel и ESLint;
- включение параметров конфигурации в файл package.json. Babel также поддерживает эту возможность.

Что, если ваши требования к построению содержат дополнительные этапы с такими операциями, как копирование, конкатенация или перемещение файлов? Можно создать сценарий командного интерпретатора и запустить его из сценария npm, но вашим коллегам, сведущим в JavaScript, будет удобнее, если вы будете использовать JavaScript. Многие системы построения предоставляют JavaScript API для автоматизации построения. В следующем разделе описано одно из таких решений: Gulp.

4.3. Автоматизация с использованием Gulp

Gulp (*http://gulpjs.com/*) — система построения, работающая на основе потоков (streams). Пользователь может сводить потоки воедино для создания процессов построения, не ограничивающихся простой транспиляцией или минификацией кода. Представьте, что у вас имеется проект с административной областью, построенной на базе Angular, и с общедоступной областью, построенной на базе React; оба подпроекта имеют некоторые общие требования к построению. С Gulp вы сможете повторно использовать части процесса построения для каждой стадии. На рис. 4.1 представлены примеры двух процессов построения с общей функциональностью.

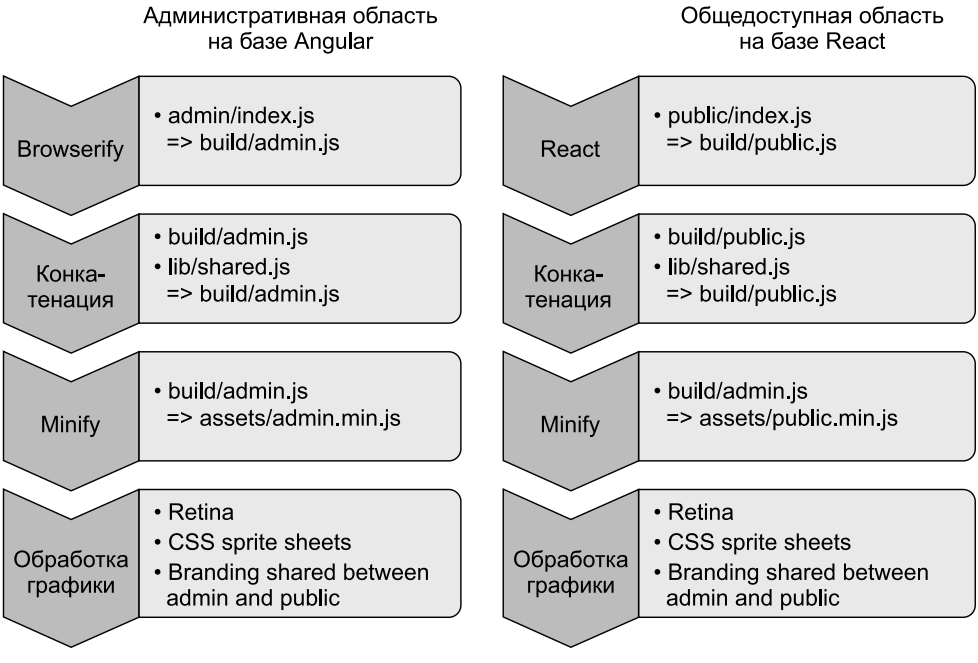

Рис. 4.1. Два процесса построения с общей функциональностью

Gulp позволяет обеспечить высокую степень повторного использования кода с помощью двух приемов: применения плагинов и определения ваших собственных задач построения. Как видно из иллюстрации, процесс построения представляет собой поток, поэтому задачи и плагины можно объединять в цепочку. Например, React-часть приведенного примера можно обработать с использованием Gulp Babel (*www.npmjs.com/package/gulp-babel/*) и встроенных средств gulp.src:

```
gulp.src('public/index.jsx')
  .pipe(babel({
    presets: ['es2015', 'react']
  }))
  .pipe(minify())
  .pipe(gulp.dest('build/public.js'));
```

В эту цепочку можно даже достаточно легко добавить стадию конкатенации. Но, прежде чем описывать синтаксис более подробно, посмотрим, как создать небольшой проект Gulp.

4.3.1. Добавление Gulp в проект

Чтобы добавить Gulp в проект, необходимо установить пакеты gulp-cli и gulp из npm. Многие пользователи устанавливают gulp-cli глобально, поэтому примеры

Gulp можно запустить простым вводом команды `gulp`. Учтите, что, если пакет gulp был ранее установлен глобально, следует выполнить команду `npm rm --global gulp`. Выполните следующий фрагмент, чтобы установить gulp-cli глобально и создать новый проект Node с зависимостью от Gulp:

```
npm i --global gulp-cli
mkdir gulp-example
cd gulp-example
npm init -y
npm i –save-dev gulp
```

Затем создайте файл gulpfile.js:

```
touch gulpfile.js
```

Откройте файл. Далее мы воспользуемся Gulp для построения небольшого проекта React, в котором используются пакеты gulp-babel (*www.npmjs.com/package/gulp-babel*), gulp-sourcemaps и gulp-concat:

```
npm i --save-dev gulp-sourcemaps gulp-babel babel-preset-es2015
npm i --save-dev gulp-concat react react-dom babel-preset-react
```

Не забудьте использовать npm с ключом `--save-dev`, когда вы хотите добавить плагины Gulp в проект. Если вы экспериментируете с новыми плагинами и позднее решите удалить их, используйте команду `npm uninstall --save-dev`, чтобы удалить их из ./node_modules и обновить файл package.json проекта.

4.3.2. Создание и выполнение задач Gulp

Создание задач Gulp требует написания кода Node с Gulp API в файле с именем gulpfile.js. Gulp API содержит методы для таких операций, как поиск файлов и передача их через плагины, которые каким-то образом их преобразовывают.

Попробуйте сами: откройте файл gulpfile.js и создайте задачу построения, которая использует gulp.src для поиска файлов JSX, Babel для обработки ES2015 и React, а затем выполняет конкатенацию для слияния файлов, как показано в листинге 4.1.

Листинг 4.1. Gulp-файл для ES2015 и React с использованием Babel

```
const gulp = require('gulp');
const sourcemaps = require('gulp-sourcemaps');
const babel = require('gulp-babel');
const concat = require('gulp-concat');

gulp.task('default', () => {
  return gulp.src('app/*.jsx')
    .pipe(sourcemaps.init())
    .pipe(babel({
```

Плагины Gulp загружаются так же, как и стандартные модули Node.

Встроенные средства gulp.src используются для поиска всех файлов React с расширением JSX.

Запускает анализ файлов для построения отладочных карт исходного кода.

```
      presets: ['es2015', 'react'] ←— Настраивает gulp-babel для использования ES2015 и React (jsx).
    }))
    .pipe(concat('all.js'))    ←——— Объединяет все файлы с исходным кодом в all.js.
    .pipe(sourcemaps.write('.'))  ←——— Записывает файлы с картами отдельно.
    .pipe(gulp.dest('dist'));    ←——— Перенаправляет все файлы в dist/folder.
});
```

В листинге 4.1 используются плагины Gulp для получения, обработки и записи файлов. Сначала все входные файлы находятся по шаблону, после чего плагин gulp-sourcemaps используется для сбора метрик карты исходного кода для отладки на стороне клиента. Обратите внимание на то, что для работы с картами исходного кода нужны две фазы: в одной вы указываете, что хотите использовать карты, а в другой записываете файлы карт. Также gulp-babel настраивается для обработки файлов с использованием ES2015 и React.

Эта задача Gulp может быть выполнена командой gulp в терминале.

В этом примере все файлы преобразуются с использованием одного плагина. Так уж вышло, что Babel и транспилирует код React JSX, и преобразует ES2015 и ES5. Когда это будет сделано, выполняется конкатенация файлов с использованием плагина gulp-concat. После завершения транспиляции происходит безопасная запись карт исходного кода, и итоговая сборка помещается в папку dist.

Чтобы опробовать этот gulp-файл, создайте файл JSX с именем app/index.jsx. Простой файл JSX, который может использоваться для тестирования Gulp, может выглядеть так:

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('example')
);
```

Gulp позволяет легко описать стадии построения на JavaScript, а при помощи метода `gulp.task()` вы сможете добавить в этот файл собственные задачи. Задачи обычно строятся по одной схеме:

- ввод — получение исходных файлов;
- траспиляция — прохождение через плагин, который преобразует их;
- конкатенация — объединение файлов для создания монолитной сборки;
- вывод — назначение местонахождения или перемещение выходных файлов.

В предыдущем примере `sourcemaps` является особым случаем, потому что требует двух конвейеров: для конфигурации и для вывода файлов. Это логично, потому

что карты исходного кода зависят от соответствия исходной нумерации строк и нумерации строк в транспилированной сборке.

4.3.3. Отслеживание изменений

Последнее, что потребуется фронтэнд-разработчику, — цикл «построение/обновление». Для упрощения процесса легче всего воспользоваться плагином Gulp для отслеживания изменений в файловой системе. Впрочем, существуют и альтернативные решения. Некоторые библиотеки хорошо работают с «горячей» перезагрузкой, более общие проекты на базе DOM и CSS хорошо работают с проектом LiveReload (*http://livereload.com/*).

Например, к проекту из листинга 4.1 можно добавить gulp-watch (*www.npmjs.com/package/gulp-watch*). Добавьте пакет в проект:

```
npm i --save-dev gulp-watch
```

Не забудьте загрузить пакет в gulpfile.js:

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

А теперь добавьте задачу `watch`, которая вызывает задачу по умолчанию из предыдущего примера:

```
gulp.task('watch', () => {
  watch('app/**.jsx', () => gulp.start('default'));
});
```

Этот фрагмент определяет задачу с именем `watch`, а затем использует вызов `watch()` для отслеживания изменений в файлах React JSX. При каждом изменении файла выполняется задача построения по умолчанию. С небольшими изменениями этот рецепт может использоваться для построения файлов SASS (Syntactically Awesome Style Sheets), оптимизации графики и вообще практически всего, что только может понадобиться в проектах клиентской части.

4.3.4. Использование отдельных файлов в больших проектах

По мере роста проекта обычно приходится добавлять новые задачи Gulp. Рано или поздно образуется большой файл, в котором трудно разобраться. Впрочем, проблема решается: разбейте свой код на модули.

Как вы уже видели, Gulp использует систему модулей Node для загрузки плагинов. Специальной системы загрузки плагинов не существует; используются стандартные модули. Также система модулей Node может использоваться для разбивки длинных gulp-файлов с целью упрощения сопровождения. Чтобы использовать отдельные файлы, выполните следующие действия.