Урок 8

Тестирование и сборка

Тестирование отдельных частей приложения. Сборка модулей с помощью Webpack.

Тестирование – обязательная часть создания приложения. Благодаря тестам программист понимает, работает ли та или иная часть программы так, как он запланировал, или нет.

Часто программисты тестируют вручную, то есть проводят манипуляции с кодом и интерфейсом, убеждаясь, что всё работает как задумано. Но при таком подходе можно пропустить много дефектов ПО. К тому же с увеличением объёма кодовой базы растёт и время, которое программист тратит на ручное тестирование.

Для экономии времени и упрощения масштабирования приложения используются автоматические тесты. Это небольшие подпрограммы, которые проверяют функции и методы, написанные программистом. Существует много разных библиотек для тестирования JavaScript-кода. Мы будем использовать одну из самых популярных – Jasmine.

Установка и запуск

Есть несколько способов работы с Jasmine. Мы будем запускать его через NPM. Для начала создадим файл **package.json**:

```
$ npm init
```

Теперь установим Jasmine глобально. Так он будет доступен для вызова из командной строки с помощью команды **jasmine**:

```
$ npm install -g jasmine
```

Установим его ещё и локально, чтобы тесты можно было запускать через NPM, даже если на компьютере нет глобального Jasmine:

```
$ npm install --save jasmine
```

Установленный локально пакет Jasmine записался в поле **dependencies** файла **package.json**:

```
{
  "name": "tests",
  "version": "1.0.0",
  "description": "",
  "author": "Me",
  "license": "ISC",
  "dependencies": {
    "jasmine": "^3.3.0"
  }
}
```

Теперь нужно инициализировать Jasmine, чтобы создать нужные для работы файлы и папки:

```
$ jasmine init
```

Итак, Jasmine создал папку **/spec**, в которой будут лежать все написанные нами тесты. Сейчас в ней есть только вложенная папка **support**. В ней находятся служебные файлы, и сейчас такой файл один

– **jasmine.json**. Это файл с настройками, управляющий работой Jasmine. Менять в нём без особой необходимости ничего не нужно.

```
{
  "spec_dir": "spec",
  "spec_files": [
    "**/*[sS]pec.js"
  ],
  "helpers": [
    "helpers/**/*.js"
  ],
  "stopSpecOnExpectationFailure": false,
  "random": true
}
```

Для запуска тестов используется npm-скрипт: это команда, которая заставляет **npm** выполнять определённую последовательность действий. Добавим скрипт в **package.json**:

```
{
  "name": "tests",
  "version": "1.0.0",
  "description": "",
  "scripts": {
    "test": "jasmine"
  },
  "author": "Me",
  "license": "ISC",
  "dependencies": {
    "jasmine": "^3.3.0"
  }
}
```

Теперь при вызове команды **npm test** будет вызвана команда **jasmine**. Причём **npm** будет использовать версию из папки **node_modules**, т. е. всё сработает, даже если Jasmine не установлен глобально:

```
$ npm test
```

Команда **jasmine** запускает все тесты, лежащие в папке **spec**, но там пока ничего нет. Давайте их напишем.

Разработка через поведение

Jasmine реализует подход BDD – Behavior-driven development (разработка через поведение): сначала продумывается и описывается поведение функции или метода, а уже потом пишется код.

Посмотрим, как это работает на практике, а заодно познакомимся с синтаксисом Jasmine. Создадим в рабочей папке файл **script.js**, а в папке **/spec** – файл **script.spec.js**. В первом мы будем писать функцию, а во втором – тесты. Предположим, мы хотим написать функцию, которая будет возводить

число в степень. Напишем заготовку в файл **script.js**:

```
//script.js
const pow = (a, n) => {
  // a – число
  // n – основание степени
}
```

Функция готова для вызова, но сейчас она видна только в пределах файла. Нужно её экспортировать, чтобы она была видна извне. Воспользуемся для этого модулями. Подробнее о модулях мы поговорим чуть позже, а пока просто напишем так:

```
//script.js
const pow = (a, n) => {
  // a –число
  // n – основание степени
}

module.exports = {
  pow: pow
}
```

Это означает, что при подключении данного модуля, мы получим объект, в поле **pow** которого будет лежать наша функция **pow**. Теперь подключим **script.js** в файле **script.spec.js**:

```
//script.spec.js
const script = require('../script');
const pow = script.pow;
```

Теперь можно приступать к написанию спецификаций. Вкратце BDD подход на практике выглядит так:
Пишем тест. Придумываем поведение и описываем в спецификации.
Тест падает. Нужное поведение ещё не реализовано, поэтому тест не пройден.
Пишем код. Описываем в коде нужное поведение.
Отладка. Обновляем код до тех пор, пока он не начнёт проходить тест.
Тест пройден. Нужное поведение реализовано.
Пишем тест. Придумываем следующий вариант поведения.
Тест падает.
Пишем код.
…

Может показаться, что это слишком долго, но на деле тестирование позволяет сэкономить очень много времени на поддержание приложения. Другой плюс – более высокое качество кода и лучшие архитектурные решения.

Продумываем поведение нашей функции. В Jasmine тесты группируются по какому-либо признаку, например по функции или тестируемому модулю. Группа тестов описывается методом **describe**:

```
describe('Функция pow()', () => {
  ...
});
```

Каждый отдельный тест описывается методом **it**:

```
describe('Функция pow()', () => {
  it('должна возвращать 9 при аргументах (3, 2)', () => {
    ...
  })
});
```

Ожидаемое поведение описывается с помощью специальных методов **expect** и **toBe**:

```
describe('Функция pow()', () => {
  it('должна возвращать 9 при аргументах (3, 2)', () => {
    expect(pow(3, 2)).toBe(9);
  })
});
```

Такая запись означает, что мы ожидаем, что результат вызова pow(3, 2) будет 9. Запустим тест и проверим:

```
$ npm test
```

В консоли появляется результат тестирования:

```
Started
F

Failures:
1) Функция pow() должна возвращать 9 при аргументах (3, 2)
  Message:
    Expected undefined to be 9.
  Stack:
    Error: Expected undefined to be 9.
        at <Jasmine>
        at UserContext.it (...)
        at <Jasmine>

1 spec, 1 failure
```

Логично, ведь сейчас функция ничего не возвращает. Допишем её так, чтобы тест выполнялся. Мы не будем использовать объект **Math**, сделаем это вручную:

```
const pow = (a, n) => {
  let result = 1;
  for (let i = 0; i < n; i++) {
    result *= a;
  }

  return result;
```

```
}
```

Запустим тест снова и проверим результат:

```
Started
.


1 spec, 0 failures
```

Всё работает. Допишем ещё один вариант поведения:

```
describe('Функция pow()', () => {
  it('должна возвращать 9 при аргументах (3, 2)', () => {
    expect(pow(3, 2)).toBe(9);
  });

  it('должна возвращать null при аргументах (null, 2)', () => {
    expect(pow(null, 2)).toBeNull();
  })
});
```

Снова запускаем тесты:

```
Started
F.

Failures:
1) Функция pow() должна возвращать null при аргументах (null, 2)
  Message:
    Expected 0 to be null.
  Stack:
    Error: Expected 0 to be null.
        at <Jasmine>
        at UserContext.it (...)
        at <Jasmine>

2 specs, 1 failure
```

У нас в функции нет проверки на null, поэтому тест не пройден. Добавим её:

```
const pow = (a, n) => {
  if (a == null || n == null) {
    return null;
  }

  let result = 1;
  for (let i = 0; i < n; i++) {
    result *= a;
```

```
  }

  return result;
}
```

Проверяем:

```
Started
..


2 specs, 0 failures
```

Тест пройден. Продолжаем придумывать варианты ожидаемого поведения до тех пор, пока не кончатся идеи. В будущем, если найдётся какой-то неучтенный вариант поведения, мы добавим его в тест и только после этого будем писать код.

Webpack

Webpack – гибкая система сборки, которая позволяет объединять, преобразовывать и оптимизировать большое количество файлов. Благодаря Webpack можно использовать модули, препроцессоры и ещё много мощных инструментов для создания приложений. В рамках этого курса мы не сможем рассмотреть все возможности Webpack, но в общих чертах с ним познакомимся.

Webpack использует Node.js, поэтому нужно убедиться, что он установлен. После этого создадим файл **package.json** и установим пакеты **webpack** и **webpack-cli**. Второй нужен для того, чтобы запускать сборщик командой **webpack**.

```
$ npm init
$ npm install --save webpack
$ npm install --save webpack-cli
```

Теперь настроим npm-скрипт **build**, который будет запускать сборщик командой **webpack**. Добавим его в **package.json**:

```
...
"scripts": {
  "build": "webpack"
},
...
```

Теперь мы можем запускать сборку командой **npm run build**.

Давайте посмотрим, как работать с модулями через Webpack. В JavaScript есть несколько подходов к реализации модулей. С одним из них, CommonJS, мы познакомились выше. Это модульная система, которая используется в Node.js. В ней содержимое модуля экспортируется с помощью **module.exports**, а подключается через **require**. ES2015 предлагает более современный вариант использования модулей. Мы будем использовать его. Создадим два файла – **script.js** и **module.js**. Первый будет основным файлом, а второй – подключаемым модулем.

Теперь создадим самый главный файл для нашей сборки – **webpack.config.js**. Это конфигурационный файл, в котором описано как именно webpack будет собирать наш проект. Пока просто экспортируем из него объект с настройками:

```
module.exports = {
  ...
}
```

В файле **module.js** опишем какую-нибудь функцию:

```
//module.js
const calc = (a, b) => {
  return a + b;
}
```

Экспортируем содержимое. В ES2015 модули экспортируются с помощью команды **export**:

```
//module.js
const calc = (a, b) => {
  return a + b;
}

export default {
  calc: calc
};
```

Теперь подключим этот модуль в **script.js**. Для подключения модулей в ES2015 используется команда import:

```
//module.js
import module from './module.js';

const calc = module.calc;
console.log(calc(2, 3));
```

Осталось настроить Webpack. Нам нужно, чтобы Webpack взял файл **script.js**, подтянул связанный с ним файл **module.js** и объединил их в один файл, например **build.js**. Прежде всего укажем параметр **entry**. Он содержит путь к главному файлу, т. е. к тому, в котором подключаются модули:

```
module.exports = {
  entry: './script',
  output: {
    filename: './build.js'
  }
}
```

Запускаем сборку:

```
$ npm run build
```

Webpack создал папку **/dis**t, а в ней файл **build.js.** Он включает в себя содержимое и **script.js**, и **module.js**. Вдобавок файл уже оптимизирован и готов к подключению на html-странице.

Webpack умеет работать и со вложенными модулями. Создадим файл **submodule.js**:

```
const subcalc = (a) => {
  return a * 10;
}

export default {
  subcalc: subcalc
};
```

Подключим и используем его в **module.js**:

```
import submodule from './submodule.js';
const subcalc = submodule.subcalc;

const calc = (a, b) => {
  return subcalc(a) + subcalc(b);
}

export default {
  calc: calc
};
```

Теперь, если запустить сборку, содержимое **submodule.js** тоже попадёт в **build.js**.

Используя такой подход, можно дробить приложение на отдельные небольшие модули, что облегчает поддержку и масштабирование.

Практическое задание

1. Вынести компоненты интернет-магазина в отдельные модули и настроить сборку.
2. Найти в официальной документации способ автоматически перезапускать **webpack** при изменении файла. Изменить скрипт **build**, добавив туда этот способ. Подсказка: при запуске нужно использовать определённый флаг.
3. * Написать приложение-калькулятор, используя подход BDD. Приложение должно состоять из четырёх методов для сложения, вычитания, умножения и деления. Каждый метод принимает на вход два аргумента и выполняет действие. При написании тестов учесть случаи, когда на вход подаются не числа, а строки, null или undefined.

Дополнительные материалы

1. Руководство по Jasmine.
2. Скринкаст по Webpack.

Используемая литература

1. Официальная документация Jasmine.
2. Официальная документация Webpack.