Урок 8

Тестирование в JavaScript

Знакомство с методиками автоматизированного тестирования в JavaScript.

Не так давно язык JavaScript применялся исключительно для достаточно простого оживления статичных страниц – применения анимации, реакции на клики, создания всплывающих окон, но JS бурно развивался и продолжает развитие. Достаточно большая часть логики плавно переходит с BackEnd в клиентскую часть веб-приложений. Она же, в свою очередь, зачастую уже сравнима по сложности с серверной логикой.

Современные компании развиваются также быстро, и в подобных условиях расширение JS-логики приносит всё больше риска сломать существующий функционал при добавлении новых модулей. И так оно и оставалось, если бы на помощь не пришли автоматические тесты. Этой методике посвящено огромное количество статей, создано множество библиотек и даже фреймворков.

На этом уроке мы рассмотрим создание площадки для проведения автоматического тестирования вашего кода, научимся принципам написания тестов, подготовим код к модульным тестам и напишем сами тесты. Чтобы не рассматривать оторванные от реальности примеры, мы будем писать тесты на базе уже написанного нами модуля корзины.

Для автоматизации тестирования мы будем использовать одну из популярных JS-библиотек для тестирования - Jasmine.

Необходимость и применимость тестов

Итак, у нас уже есть реализованный модуль корзины. В нём есть набор методов по добавлению товаров, получению содержимого т.д. На момент релиза мы уже протестировали функционал – он корректно работает и радует покупателей и нас.

Проходит время, и бизнес-департамент решает ввести возможность получения скидки по некой акции с промо-кодами. Тогда в корзине потребуется вводить поля для кодов и функционал обновления общей стоимости корзины. И вероятность того, что просьба появится не через месяц после релиза, а через полгода-год, весьма велика. Таким образом, в вашей оперативной памяти (в голове, то есть) код уже давно удалён за ненадобностью и отсутствием обращений. Так что для реализации вам потребуется не только реализовать новый функционал, но и вспомнить старый код, чтобы не нарушить его работу.
По прошествии n часов модуль скидок готов, а вы переходите к тестированию всех вариантов работы корзины, чтобы быть на 100% уверенным в том, что существующий функционал работает стабильно и корректно. Это занимает много времени, скорее всего, больше, чем написание нового кода, а особенно сильно настроение портится тогда, когда вы выясняете, что старый модуль при определённом стечении условий перестаёт работать. И хорошо, если это вы находите на тестировании, а не на production-сервере. Это приводит к самой страшной проблеме в современной интернет-торговле – потенциальный покупатель теряет возможность купить необходимый ему товар и уходит к конкуренту.
Вы срочно исправляете проблему, возвращая клиентов и их заказы, но тут бизнес просит вас ввести функционал сложного расчёта стоимости доставки. Получается, что снова надо повторять все процедуры написания и проверки кода с учётом прошлых рисков. Раз за разом вы всё глубже погружаетесь в этот замкнутый цикл. И как из него выйти?
Для этого вернёмся к началу – времени, когда функционал корзины только-только создавался. Мы уже знаем, как разбить логику по слоям и модулям. Теперь нам нужно подключить библиотеку для тестов и написать скрипты, которые будут эти тесты выполнять. Например, при вызове функции удаления товара из корзины финальная сумма заказа должна уменьшиться на стоимость этого товара. Таким образом, автоматические тесты делают те же самые шаги, что и вы, тестируя код вручную, но не требуя вашего ресурса каждый раз, когда надо проверить работоспособность системы.
Множество тестов сформируется в пакет проверки, запуская который вы сразу увидите, что пошло не так, и какие модули нужно исправлять.
Таким образом:

1. Только ознакомившись с описаниями тестов, вы тут же поймёте, за какие действия отвечает участок кода, и какие комбинация входных данных могут подаваться.
2. Описывая кодом новый функционал, вы можете не бояться сломать существующую логику, т.к. будете постоянно запускать тесты, которые сразу покажут неисправность.

Разумеется, при написании нового функционала его нужно будет покрывать тестами, чтобы вся логика продолжаться оставаться контролируемой.

Jasmine

Jasmine - это BDD фреймворк (Behavior-Driven Development — разработка на основе поведений), созданный для тестирования JS-кода. Он обладает весьма достойной документацией, достаточно прост в подключении и настройке рабочей среды, имеет понятный синтаксис, разрешает запускать тесты как в браузере, так и в командной строке. Одним из очень интересных расширений является jasmine-jquery, который позволяет тестировать ещё и jQuery-части кода.
Необходимые файлы и документацию можно найти по ссылке http://jasmine.github.io/.
Для начала работы нам понадобятся исходники, которые лежат здесь https://github.com/pivotal/jasmine/downloads. Потребуются следующие файлы:

- lib/jasmine-*/jasmine.js — сам фреймворк;
- lib/jasmine-*/jasmine-html.js — оформление результатов в виде HTML;
- lib/jasmine-*/jasmine.css — внешний вид результата выполнения тестов;
- SpecRunner.html — файл, который следует открыть в браузере для запуска тестов.

Основными ключевыми словами при работе с Jasmine являются:

- describe — определение набора тестов, наборы могут быть вложенными;
- it — определение теста внутри любого набора тестов;
- expect — определяет ожидания, которые проверяются в тесте.

Ключевые слова describe и it – это просто вызовы функций, которым передаются два параметра. Первый — название группы или теста, второй — функция содержащая код. Простой пример для ясности:

```
describe "Набор тестов", ->
  it "проверка ожиданий", ->
    expect(1 + 2).toBe(3)
```

Для отключения выполнения набора тестов или одного теста, нужно применять ключевые слова xdescribe и xit соответственно.

```
describe "Отключение", ->
  xdescribe "отключенный набор тестов", ->
    it "тест не будет запущен, так как набор отключен", ->
      expect(true).toBe(true)
  xit "отключеный тест", ->
    expect(true).toBe(true)
```

Jasmine имеет стандартный набор ожиданий для проверки результатов:

```
it "сравнение с использованием ===", ->
    expect(1 + 2).toBe(3)
  it "сравнение переменных и объектов (включая содержимое)", ->
    a = {x: 8, y: 9}
    b = {x: 8, y: 9}
    expect(a).toEqual(b)
```

```
  expect(a).not.toBe(b) # отрицание - a не является b
it "значение должно быть определено", ->
  expect(window.document).toBeDefined()
it "значение должно быть не определено", ->
  expect(window.notExists).toBeUndefined()
it "значение должно быть null", ->
  a = null
  expect(a).toBeNull()
it "значение должно быть верно", ->
  expect(5 > 0).toBeTruthy()
it "значение должно быть не верно", ->
  expect(5 < 0).toBeFalsy()
it "значение должно быть меньше чем", ->
  expect(1 + 2).toBeLessThan(5)
it "значение должно быть больше чем", ->
  expect(1 + 2).toBeGreaterThan(0)
it "значение должно быть близко к числу", ->
  expect(1.2345).toBeCloseTo(1.2, 1)
it "значение должно соответствовать регулярному выражению", ->
  expect("some string").toMatch(/string/)
it "значение должно содержать", ->
  expect([1, 2, 3]).toContain(2)
  expect("some string").toContain("some")
it "должно быть вызвано исключение", ->
  func = -> window.notExists.value
  expect(func).toThrow()
```

Подготовка проекта и первые тесты

Jasmine для корректной работы потребуется определённая файловая структура. Создадим папку проекта и назовём её jasmine. В неё же добавим файл index.html и 3 поддиректории:

1. vendor – файлы для jasmine и внешние библиотеки для наших модулей.
2. modules - тестируемая js-логика.
3. specs - сами тесты.

Назовём файлы по имени модулей, чтобы не запутаться.
В папку vendor положим следующие файлы: jasmine.css, jasmine.js, jasmine-html.js, boot.js - jasmin-овские файлы.
В главном файле мы видим стандартную заготовку html-файла, в которую подключаем файлы скриптов jasmine, наш модуль корзины, модуль тестов корзины и файлы результатов тестов. Перед всем этим обязательно подключим jQuery – ведь мы его используем!

Теперь обозначим наш тест. Проверять будем получение содержимого корзины. В нашем случае оно тянется из статического json-файла, что позволяет написать несложный тест для понимания общей методологии тестирования.

```
describe('Корзина товаров', function() {
    var basket = Basket();
    basket.basket_items = [
                        {
              "id_product" : 123,
              "price" : 100
                        }
            ];
    // тесты...
});
```

Конструкция describe указывает название тестируемого модуля.
Далее создаём переменную basket, которая будет являться самой корзиной. Пока мы зададим содержимое корзины непосредственно в describe.
У нас есть основные возможности модуля корзины: считывание данных, добавление товаров в корзину, удаление.
Резонно будет заметить, что каждый тест может работать вне зависимости от предыдущих ему тестов, а сами тесты есть возможность запускать в разной последовательности, но в нашем случае такой необходимости нет. Мы должны обеспечить достаточно базовый функционал тестов для корзины. У нас тесты будут производить операции с одной и той же корзиной, получая для неё данные и т.д. Этим мы моделируем все варианты работы методов, проверяем их корректность.

```
it('тест сборки содержимого корзины 1', function() {
            expect(basket.basket_items.length).toBe(1);
});
it('тест сборки содержимого корзины 2', function() {
            expect(basket.basket_items[0]).toEqual({
        "id_product" : 123,
        "price" : 100
            });
});
```

В этих двух простых тестах мы проверяем, что данные в корзину добавились корректно. Первый тест использует строгое сравнение toBe, второй уже проверяет более сложное соответствие – сравнение с объектом.

Теперь мы можем открыть наш файл index.html и проверить работу всех тестов.

Асинхронные вызовы

Но не стоит забывать, что наша корзина, как и многие другие элементы нашего проекта, активно использует AJAX. Это говорит о том, что простое сравнение может не отработать, т.к. на момент сравнения ответа от сервера попросту не будет существовать.
Именно поэтому стоит стараться оборачивать подобные вызовы в отдельные методы ещё на этапе создания функционала. При тестировании вам нужно будет переопределить их, т.к. тестирование JS-кода не включает в себя тестирование обращений к серверной части – эта часть тестирования производится на стороне back-end разработки.

Итоги

Итак, мы научились основам тестирования JS-кода. Это повысит стабильность наших продуктов и улучшит их качество.

Что дальше

В настоящее время сайты всё больше превращаются в совершенно иной тип сетевых ресурсов – RIA.
Насыщенное интернет-приложение (англ. rich internet application, RIA) — это веб-приложение, загружаемое пользователем через интернет, предназначенное для выполнения функций традиционных настольных приложений и работающее на устройстве пользователя (не на сервере).
Язык разметки HTML5 и язык программирования JavaScript являются одной из фундаментальных частей RIA-приложений.
Само RIA всё же традиционно состоит из двух частей: клиентской и серверной. Серверная часть может хранить информацию, необходимую для работы приложения, заниматься обслуживанием запросов, поступающих от клиентской части. Клиентская часть выполняется на компьютере пользователя, занимается рисованием интерфейса пользователя, выполняет запросы пользователя, при необходимости может отправлять запросы серверной части. Клиентская часть RIA выполняется в безопасной среде, т.н. «песочнице», и не требует установки дополнительного ПО.
В архитектуре RIA часть работы или вся работа может выполняться клиентом. Развитие стандартов сети Интернет привело к возможности реализовать RIA. При этом сложно провести чёткую границу между тем, какие именно технологии включают в себя такие приложения, а какие — нет. Но все RIA имеют одну особенность: на устройстве пользователя перед началом работы RIA загружается так называемый «движок клиента»; в дальнейшем движок может догружаться по ходу работы приложения.
Для реализации подобных приложений очень часто используются JS-фреймворки, которые снимают с разработчиков необходимость тратить время на реализацию низкоуровневой архитектуры, позволяя сосредоточиться на реализации клиентской логики приложения.
Одним из наиболее мощных о популярных фреймворков является AngularJS. Это фреймворк корпоративного уровня, который используется для создания и обслуживания сложных веб-приложений. Angular используют такие компании, как Domino's Pizza, Ryanair, iTunes Connect, PayPal Checkout и Google. Это фреймворк с открытым исходным кодом, который поддерживается компанией Google. Angular позиционирует себя как «расширение HTML», созданное для разработки комплексных веб-приложений.
Angular — MVC-фреймворк. В нём реализовано двустороннее сопоставление и обмен данными между моделями и представлениями. Этот подход даёт возможность автоматически обновлять хранилища с обеих сторон при любом изменении данных. В Angular можно создавать многократно используемые компоненты представлений (View Component). А благодаря имеющейся в нём

структуре сервисов (service framework) можно легко построить взаимодействие между бэкендом и фронтендом. Ну и, наконец, Angular — это чистый JavaScript.

Домашнее задание

1. Покрыть тестами оставшиеся методы в прототипах.
2. Доделать оставшиеся ДЗ.
3. * Подумать над вопросом : “Как поддерживать свои тесты актуальными?”

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. “jQuery для профессионалов” - Адам Фримен