

# Проводим отладку JavaScript-приложений

Андрей Уваров

В последнее время при разработке веб-приложений всё больше внимания стали уделять клиентской части – программированию на JavaScript. Яркими примерами являются Gmail.com, Live.com и многие другие сервисы, имеющие огромную популярность. Несметное количество пользователей этих сервисов – подтверждение жизнеспособности идеи усложнения клиентской логики. Цель статьи – осветить процесс разработки веб-приложений на языке JavaScript, а точнее, его практическую сторону – написание и отладку кода.

Любая новая программа содержит ошибки. Это утверждение применимо к любому языку программирования, и JavaScript не является исключением. С целью уменьшения количества ошибок многие разработчики пользуются следующими принципами:

- Написанный код должен быть максимально простым и понятным, не стоит применять какие-то неочевидные другим программистам трюки или специальные возможности отдельных браузеров.
- Необходимо писать код так, чтобы при большинстве ошибок не происходил сбой всего приложения.
- Необходимо писать тесты.

Перечисленные правила не являются чем-то новым в разработке и применимы к большинству языков программирования. Более подробно стоит остановиться лишь на последнем пункте: написании тестов для JavaScript. Но об этом чуть позже.

Итак, даже придерживаясь упомянутых правил, вы не застрахованы от ошибок. Существует три их основных типа:

- синтаксические ошибки;
- ошибки времени исполнения;
- логические ошибки.

Большинство синтаксических ошибок обнаруживается при загрузке скрипта браузером. Исключением из этого правила является ситуация, когда код генерируется и затем выполняется во время исполнения. К синтаксическим ошибкам относятся ошибки, связанные с неверным использованием управляющих конструкций языка, ключевых слов и прочее. Такие ошибки часто обнаруживаются при первом запуске программы. В основном они вызваны невнимательностью программиста.

Ошибки времени исполнения являются более сложными для выявления; многие из них – следствие того, что язык JavaScript слаботипизирован.

```
<script language="javascript" type="text/javascript">

    alert(eval("document.getElementById('test').value"));

</script>
```

Так, мы поставили «запятую» вместо «точки». И ошибка обнаружится, только когда программа будет выполняться.

```
<script language="javascript"
type="text/javascript">

    alert(document.getElementById
'test').value);

</script>
```

А в данном случае мы увидим предупреждение об ошибке уже на этапе загрузки страницы.

Логические ошибки часто возникают из-за некорректной реализации логики приложения. Они являются наиболее коварными, так как зачастую не приводят к видимым сбоям во вре-

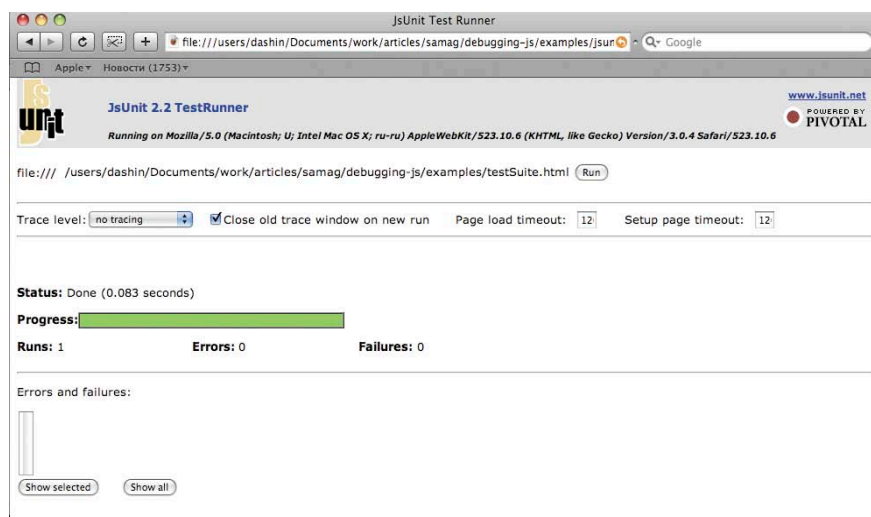


Рисунок 1. Результат выполнения теста

ма работы или загрузки программы. Но при этом программа может выдавать неверные результаты.

```
<script language="javascript" type="text/javascript">

    var myArray = new Array(1, 1, 1, 1);
    var sum = 0;
    // Попробуем найти сумму значений всех элементов массива
    for (var i = 1; i < myArray.length; i++) {
        sum += myArray[i];
    }

    document.write(sum);

</script>
```

Так как нумерация элементов в массиве начинается с нуля, то результат будет неверным. Подобного рода ошибки порой отнимают много времени.

## Модульное тестирование

Для выявления логических ошибок и ошибок времени исполнения хорошо себя зарекомендовало модульное тестирование – это процесс верификации отдельных частей программы, – для каждого метода или функции (кроме самых тривиальных) пишется свой тест. В случае изменения каких-либо частей программы написанные тесты помогут избежать появления новых ошибок.

Перечислю основные преимущества модульного тестирования:

- Благодаря тестам существенно повышается количество ошибок, найденных на этапе разработки, и соответственно повышается качество продукта.
- Разработчик вынужден писать код, приспособленный для тестирования. Следствием из этого является более чёткий и структурированный код.
- Тесты экономят время разработчика, которое он вынужден был бы тратить на тестирование кода.

Существует несколько средств для модульного тестирования JavaScript-кода. Наиболее популярными являются реализации JUnit (<http://jsunit.net> и <http://jsunit.berlios.de>) (так как обе реализации имеют название JsUnit, то во избежание путаницы будем обозначать их доменными именами).

Суть модульного тестирования заключается в том, что для каждого тестируемого метода пишется специальный тестовый метод. При помощи тестового метода проверяется корректность реализованной логики. Тестовые методы объединяются в группы (test suites), а затем выполняются при помощи UnitRunner – специального компонента фреймворка (JsUnit).

Познакомимся чуть ближе с модульным тестированием на примере [jsunit.net](http://jsunit.net).

Итак, в качестве метода, корректность работы которого нам необходимо проверить, мы выберем `isCreditCardNumberValid()`, которая проверяет валидность номера кредитной карты

(реализацию опустим), и обусловимся, что данная функция находится в файле `commons.js`.

Для проверки работоспособности данного метода создадим тестовый метод и поместим его в файл `tests/testCommons.html`:

```
...
<script language="javascript" type="text/javascript" 1
src="../commons.js"></script>
<script language="javascript" type="text/javascript" 1
src="./jsunit/app/jsUnitCore.js"></script>
<script language="javascript" type="text/javascript">
function testIsCreditCardNumberValid() {

    // assert - специальная функция JsUnit, которая в случае
    // если ей на вход подаётся значение false, сигнализирует
    // о том, что тест провалился, и предоставляет информацию
    // о конкретном месте, где это произошло
    assert(!isCreditCardNumberValid("foo"));
    assert(!isCreditCardNumberValid(null));
    assert(isCreditCardNumberValid("446667651"));

}
</script>
...
```

Здесь и далее содержимое файла опускается, так как не имеет для нас особого значения.

В качестве следующего шага создадим группу для нашего теста – файл `tests/testSuite.html`:

```
...
<script language="javascript" type="text/javascript" 1
src="./jsunit/app/jsUnitCore.js"></script>
<script language="javascript" type="text/javascript">

function coreSuite() {
    var result = new top.jsUnitTestSuite();
    result.addTestPage("./myTest.html");
    return result;
}

function suite() {
    var newsuite = new top.jsUnitTestSuite();
    newsuite.addTestSuite(coreSuite());
    return newsuite;
}

</script>
...
```

На этом написание тестов завершено, и можно приступить к тестированию. Для этого нам необходимо открыть в браузере следующий адрес:

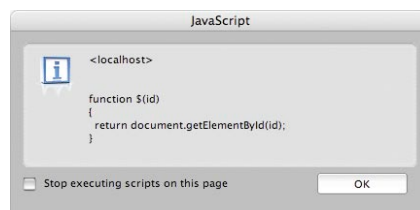


Рисунок 2. Вывод тела функции при помощи `alert()`

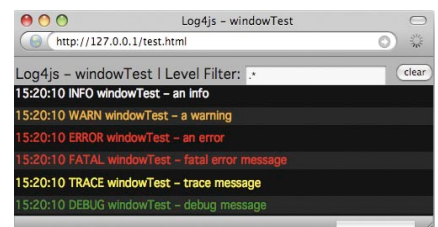


Рисунок 3. Логгер в работе

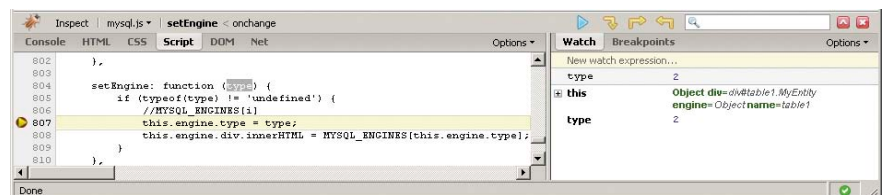


Рисунок 4. Firebug на службе отладки

```
<полный путь>testRunner.html?testpage=<путь>testSuite.html
```

В моём случае это:

```
file:///users/dashin/Documents/work/articles/samag ↵
/debugging-js/examples/jsunit ↵
/testRunner.html?testpage=/users/dashin/Documents/work ↵
/articles/samag/debugging-js/examples/testSuite.html
```

Как видно из **рис. 1** – наш тест успешно пройден, и мы можем считать, что функция `isCreditCardNumberValid()` нормально справляется со своей задачей.

Тестирование является лишь способом выявления ошибок, и в случае с модульным тестированием мы также можем определить фрагмент кода, содержащий ошибку. Но точное место ошибки не всегда представляется возможным распознать при помощи тестов. Вот тут и начинается настоящая отладка приложения. Самым простым и самым распространённым, но не самым удобным и изящным средством отладки является функция `alert()`. К ней прибегают многие разработчики, чтобы уже на этапе выполнения программы узнать значение какой-либо переменной или функции. Данная функция выведет тело интересующей функции или значение переменной:

```
<script type="text/javascript" >
function $(id) {
    return document.getElementById(id);
}
alert($);
</script>
```

## Логгирование

Более правильным и удобным решением является использование какого-либо логгера. Логгеры – это специальные библиотеки, позволяющие выводить отладочную информацию во время выполнения программы. На данный момент существует большое количество таких библиотек. Все они близки по функциональности, поэтому выбрать для себя можно любую, основываясь на своих «религиозных» убеждениях.

Принцип использования логгеров аналогичен отладке приложений при помощи `alert()`, но на практике логгеры оказываются существенно удобнее. Рассмотрим работу логгера на примере `Log4js` (<http://log4js.berlios.de>).

`Log4js` очень похож на `log4j` (<http://logging.apache.org/log4j>). `Log4j` является аналогичным проектом, появившимся ранее и написанным на Java. Суть заключается в том, что мы выводим интересующую нас информацию при помощи специальных функций: `fatal()`, `error()`, `warn()` и других, и можем в конфигурационном файле определить, каким образом эти сообщения будут выводиться. Так, например, мы можем использовать для этого дополнительное окно браузера или же при помощи AJAX незаметно отправлять сообщения на сервер, где они будут сохраняться.

```
<html>
<head>
<title>Log4js example</title>
<script src="log4js.js" type="text/javascript"></script>
</head>
<body>
```

```
<script type="text/javascript" >

    var myLogger = new Log4js.getLogger("windowTest");

    // Устанавливаем режим работы логгера
    myLogger.setLevel(Log4js.Level.ALL);
    // Задаём способ логгирования
    myLogger.addAppender(new Log4js.ConsoleAppender(false));

    // Тестовые сообщения
    myLogger.info('an info');
    myLogger.warn('a warning');
    myLogger.error('an error');
    myLogger.fatal('fatal error message');
    myLogger.trace('trace message');
    myLogger.debug('debug message');

</script>

</body>
</html>
```

В данном логгере существует несколько режимов работы: OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE, ALL. В режиме OFF не отображается ни один из типов сообщений, а в режиме ALL отображаются все типы сообщений. Каждый режим включает в себя предыдущий. Так, например, при включенном режиме WARN будут выводиться все сообщения `warn()`, `error()` и `fatal()`, а при включенном FATAL будут отображаться только сообщения `fatal()`.

Логгеры весьма хороши для использования на рабочих серверах. С их помощью можно узнать о клиентских ошибках и оказать своевременную помощь коду – устранить ошибки. Но злоупотреблять логгерами также не стоит, так как они могут привести к лишним затратам процессорного времени.

## Отладчики

Наряду с логгированием для отладки приложений применяют специальные программы – отладчики. Одними из самых популярных на данный момент являются Firebug (<http://www.getfirebug.com>) и Venkman (<http://www.mozilla.org/projects/venkman>). Firebug и Venkman представляют собой расширения для браузера Firefox и предоставляют разработчику возможность пошаговой отладки, установки условных и безусловных точек останова, просмотра значений переменных во время выполнения скриптов, профилирования, а также просмотра HTTP-заголовков (реализовано только в Firebug) и некоторые другие полезные вещи. На данный момент эти отладчики являются наиболее функциональными и удобными из существующих. И в использовании практически ничем не отличаются от отладчиков для других языков.

Рассмотрим вкратце процесс отладки с использованием Firebug. Прежде всего необходимо включить данное расширение, это делается на вкладке «Tools». Сам отладчик может работать как в отдельном окне, так и в фрейме браузера. Затем открываем в браузере страничку, содержащую интересующий нас JavaScript-код. Теперь в отладчике мы можем найти интересующий код, поставить точку останова и продолжить выполнять код уже пошагово, внимательно разыскивая ошибки.

Почти постоянный рост популярности JavaScript привёл к тому, что сейчас в нашем распоряжении находится достаточное количество средств, призванных упростить разработку и повысить качество результата. ●