JavaScript(https://www.sitepoint.com/javascript/) - June 27, 2017 - By Graham Cox (https://www.sitepoint.com/author/gcox/)

BDD in JavaScript: Getting Started with Cucumber and Gherkin

By now, everyone has heard about Test Driven Development (https://www.sitepoint.com/learning-javascript-test-driven-development-by-example/) (TDD), and the benefits that this can have on your product and your development lifecycle. It's a no-brainer really. Every time you write a test for a piece of code, you know that code works. And, what's more, you will know in the future if that code breaks.

Behavior Driven Development (BDD) is an extension to this concept, but instead of testing your *code* you are testing your *product*, and specifically that your product behaves as you desire it to.

In this article, I'm going to show you how to get up and running with Cucumber, a framework that runs automated acceptance tests written in a BDD style. The advantage of these tests is that they can written in plain English and consequently understood by non-technical people involved in a project. After reading, you will be in a position to decide if Cucumber is a good fit for you and your team and to start writing acceptance tests of your own.

Ready? Then let's dive in.

BDD vs TDD — so, What's the Difference?

Primarially in the way that tests are structured and written.

In a TDD setting, the tests are written, maintained and understood by the developers who wrote the code they are testing. It might well be that nobody else ever needs to read the tests at all, and that's fine.

In a BDD setting, the tests need to be understood by a lot more than just the developer writing the functionality. There are many more stakeholders who have an interest that the product behaves as it should.
These might include QA people, product analysts, sales, even upper management.

This means that, in an ideal world, BDD tests need to be written in a way that anyone who understands the product will be able to pick up the tests and understand them as well.

It's the difference between:

```
const assert = require('assert');
const webdriver = require('selenium-webdriver');
const browser = new webdriver.Builder()
  .usingServer()
  .withCapabilities({'browserName': 'chrome' })
  .build();

browser.get('http://en.wikipedia.org/wiki/Wiki');
browser.findElements(webdriver.By.css('[href^="/wiki/"]'))
.then(function(links){
  assert.equal(19, links.length); // Made up number
  browser.quit();
});
```

And:

```
Given I have opened a Web Browser
When I load the Wikipedia article on "Wiki"
Then I have "19" Wiki Links
```

The two tests do exactly the same thing, but one is actually human readable and the other is only readable by someone who knows both JavaScript and Selenium.

This article will show you how you can implement BDD Tests in your JavaScript project, using the Cucumber.js framework, allowing you to benefit from this level of testing for your product.

What Is Cucumber / Gherkin?

Cucumber is a testing framework (https://cucumber.io/) for behavior driven development. It works by allowing you to define your tests in Gherkin (https://github.com/cucumber/cucumber/wiki/Gherkin) form, and makes these gherkins executable by tying them to code.

Gherkin is the Domain Specific Language (DSL) that is used for writing Cucumber tests. It allows for test scripts to be written in a human readable format, which can then be shared between all of the stakeholders in the product development.

Gherkin files are files that contain tests, written in the Gherkin language. These files typically have a `.feature` file extension. The contents of these Gherkin files are often referred to as simply “gherkins”.

Gherkins

In a Gherkin defined test, you have the concept of *features* and *scenarios*. These are analogous to test suites and test cases in other testing frameworks, allowing for a clean way to structure your tests.

A scenario is literally just a single test. It should test exactly one thing in your application.

A feature is a group of related scenarios. As such, it will test many related things in your application. Ideally the features in the Gherkin files will closely map on to the Features in the application — hence the name.

Every Gherkin file contains exactly one feature, and every feature contains one or more scenarios.

Scenarios are then comprised of steps, which are ordered in a specific manner:

Given – These steps are used to set up the initial state before you do your test
When – These steps are the actual test that is to be executed
Then – These steps are used to assert on the outcome of the test

Ideally each scenario should be a single test case, so the number of When steps should be kept very small.

Steps are entirely optional. If you have no need to set anything up at all, you might have no Given steps, for example.

Gherkin files are designed to be human readable, and to give benefit to anyone involved in the product development. This includes non-technical people, so the Gherkin files should always be written in business language and not technical language. This means, for example, that you do not refer to individual UI components, but instead describe the product concepts that you are wanting to test.

An Example Gherkin Test

The following is an example Gherkin for searching Google for Cucumber.js

```
Given I have loaded Google
When I search for "cucumber.js"
Then the first result is "GitHub - cucumber/cucumber-js: Cucumber for JavaScript"
```

Straight away we can see that this test tells us *what* to do and not *how* to do it. It is written in a language that makes sense to anyone reading it, and — importantly — that will most likely be correct no matter how the end product might be tweaked. Google could decide to completely change their UI, but as long as the functionality is equivalent then the Gherkin is still accurate.

You can read more about Given When Then (https://github.com/cucumber/cucumber/wiki/Given-When-Then) on the Cucumber wiki.

Cucumber.js

Once you have written your test cases in Gherkin form, you need some way to execute them. In the JavaScript world, there is a module (https://www.npmjs.com/package/cucumber) called Cucumber.js that allows you to do this. It works by allowing you to define JavaScript code that it can connect to the various steps defined inside of your Gherkin files. It then runs the tests by loading the Gherkin files and executing the JavaScript code associated with each step in the correct order.

For example, in the above example you would have the following steps:

```
Given('I have loaded Google', function() {});
When('I search for {stringInDoubleQuotes}', function() {});
Then('the first result is {stringInDoubleQuotes}', function() {});
```

Don't worry too much about what all of this means – it will be explained in detail later. Essentially though, it defines some ways that the Cucumber.js framework can tie your code to the steps in your Gherkin files.

Including Cucumber.js in Your Build

Including Cucumber.js in your build is as simple as adding the `cucumber` module to your build, and then configuring it to run. The first part of this is as simple as:

```
$ npm install --save-dev cucumber
```

The second of these varies depending on how you are executing your build.

Running by Hand

Executing Cucumber by hand is relatively easy, and it's a good idea to make sure you can do this first because the following solutions are all just automated ways of doing the same.

Once installed, the executable will be `./node_modules/.bin/cucumber.js`. When you run it, it needs to know where on the file system it can find all of its required files. These are both the Gherkin files and the JavaScript code to be executed.

By convention, all of your Gherkin files will be kept in the `features` directory, and if you don't instruct it otherwise then Cucumber will look in the same directory for the JavaScript code to execute as well. Instructing it where to look for these files is a sensible practice however, so that you have better control over your build process.

For example, if you keep all of your Gherkin files in the directory `myFeatures` and all of your JavaScript code in `mySteps` then you could execute the following:

```
$ ./node_modules/.bin/cucumber.js ./myFeatures -r ./mySteps
```

The `-r` flag is a directory containing JavaScript files to automatically require for the tests. There are other flags that might be of interest as well — just read the help text to see how they all work: `$ ./node_modules/.bin/cucumber.js --help`.

These directories are scanned recursively so you can nest files as shallowly or deeply as makes sense for your specific situation.

npm scripts

Once you have got Cucumber running manually, adding it to the build as an npm Script is a trivial case. You simply need to add the following command — without the fully qualified path, since npm handles that for you — to your package.json as follows:

```
"scripts": {
  "cucumber": "cucumber.js ./myFeatures -r ./mySteps"
}
```

Once this is done, you can execute:

```
$ npm run cucumber
```

And it will execute your Cucumber tests exactly as you did before.

Grunt

There does exist a Grunt plugin for executing Cucumber.js tests. Unfortunately it is very out of date, and doesn't work with the more recent versions of Cucumber.js, which means that you will miss out on a lot of improvements if you use it.

Instead, my preferred way is to simply use the grunt-shell (https://www.npmjs.com/package/grunt-shell) plugin to execute the command in the exact same way as above.

Once installed, configuring this is simply a case of adding the following plugin configuration to your `Gruntfile.js`:

```
shell: {
  cucumber: {
    command: 'cucumber.js ./myFeatures -r ./mySteps'
  }
}
```

And now, as before, you can execute your tests by running `grunt shell:cucumber`.

Gulp

Gulp is in exactly the same situation as Grunt, in that the existing plugins are very out of date and will use an old version of the Cucumber tool. Again, here you can use the gulp-shell (https://www.npmjs.com/package/gulp-shell) module to execute the Cucumber.js command as in the other scenarios.

Setting this up is as simple as:

```
gulp.task('cucumber', shell.task([
  'cucumber.js ./myFeatures -r ./mySteps'
]));
```

And now, as before, you can execute your tests by running `gulp cucumber`.

Your First Cucumber Test

> Please note that all of the code samples in this article are available on GitHub (https://github.com/sitepoint-editors/intro-to-cucumber.js).

Now that we know how to execute Cucumber, lets actually write a test. For this example, we're going to do something fairly contrived just to show the system in action. In reality you would do something much more involved, for example directly calling the code you are testing, making HTTP API calls to a running service or controlling Selenium to drive a web browser to test your application.

Our simple example is going to prove that mathematics still works. We're going to have a two features — Addition and Multiplication.

Firstly, let's get set up.

```
$ npm init
$ npm install --save-dev cucumber
$ mkdir features steps
```

How you execute your tests is entirely up to you. For this example, I'm going to be doing it manually just for simplicity sake. In a real project you would integrate this into your build using one of the above options.

```
$ ./node_modules/.bin/cucumber.js features/ -r steps/
0 scenarios
0 steps
0m00.000s
$
```

Now, let's write our first actual feature. This will go in `features/addition.feature`:

```
Feature: Addition
  Scenario: 1 + 0
    Given I start with 1
    When I add 0
    Then I end up with 1

  Scenario: 1 + 1
    Given I start with 1
    When I add 1
    Then I end up with 2
```

Very simple, very easy to read. Tells us exactly what we are doing, and nothing about how we are doing it. Let's try it out:

```
$ ./node_modules/.bin/cucumber.js features/ -r steps/
Feature: Addition

  Scenario: 1 + 0
  ? Given I start with 1
  ? When I add 0
  ? Then I end up with 1

  Scenario: 1 + 1
  ? Given I start with 1
  ? When I add 1
  ? Then I end up with 2

Warnings:

1) Scenario: 1 + 0 - features/addition.feature:3
   Step: Given I start with 1 - features/addition.feature:4
   Message:
     Undefined. Implement with the following snippet:

       Given('I start with {int}', function (int, callback) {
         // Write code here that turns the phrase above into concrete actions
         callback(null, 'pending');
       });

2) Scenario: 1 + 0 - features/addition.feature:3
   Step: When I add 0 - features/addition.feature:5
   Message:
     Undefined. Implement with the following snippet:

       When('I add {int}', function (int, callback) {
         // Write code here that turns the phrase above into concrete actions
         callback(null, 'pending');
       });

3) Scenario: 1 + 0 - features/addition.feature:3
   Step: Then I end up with 1 - features/addition.feature:6
   Message:
     Undefined. Implement with the following snippet:

       Then('I end up with {int}', function (int, callback) {
         // Write code here that turns the phrase above into concrete actions
         callback(null, 'pending');
       });

4) Scenario: 1 + 1 - features/addition.feature:8
   Step: Given I start with 1 - features/addition.feature:9
   Message:
     Undefined. Implement with the following snippet:

       Given('I start with {int}', function (int, callback) {
         // Write code here that turns the phrase above into concrete actions
         callback(null, 'pending');
       });

5) Scenario: 1 + 1 - features/addition.feature:8
   Step: When I add 1 - features/addition.feature:10
   Message:
     Undefined. Implement with the following snippet:

       When('I add {int}', function (int, callback) {
         // Write code here that turns the phrase above into concrete actions
         callback(null, 'pending');
```

```
      });

6) Scenario: 1 + 1 - features/addition.feature:8
   Step: Then I end up with 2 - features/addition.feature:11
   Message:
     Undefined. Implement with the following snippet:

       Then('I end up with {int}', function (int, callback) {
         // Write code here that turns the phrase above into concrete actions
         callback(null, 'pending');
       });

2 scenarios (2 undefined)
6 steps (6 undefined)
0m00.000s
$
```

Wow. We've just written our Gherkin, and everything executes. It doesn't work, because we don't know yet what to do with any of those steps, but Cucumber tells us this very clearly.

Let's write our first step file then. This will simply implement the steps in the way that the Cucumber output tells us to, which doesn't do anything useful but tidies up the output.

This goes in `steps/maths.js`:

```
const defineSupportCode = require('cucumber').defineSupportCode;

defineSupportCode(function({ Given, Then, When }) {
  Given('I start with {int}', function (int, callback) {
    // Write code here that turns the phrase above into concrete actions
    callback(null, 'pending');
  });
  When('I add {int}', function (int, callback) {
    // Write code here that turns the phrase above into concrete actions
    callback(null, 'pending');
  });
  Then('I end up with {int}', function (int, callback) {
    // Write code here that turns the phrase above into concrete actions
    callback(null, 'pending');
  });
});
```

The `defineSupportCode` hook is Cucumber.js's way of allowing you to provide code that it will use for a variety of different situations. These will all be covered, but essentially any time you want to write code that Cucumber will call directly then it needs to be inside one of these blocks.

You'll notice that the example code here defines three different steps – one each for Given, When and Then. Each of these blocks is given a string — or a regex if you desire — that matches a step in a feature file, and a function that is executed when that step matches. Placeholders can be placed into the step string — or if you are using a Regex then you use capturing expressions instead — and these placeholders will be extracted out and made available as parameters to your function.

Executing this gives a much more succinct output, whilst still not actually doing anything:

```
$ ./node_modules/.bin/cucumber.js features/ -r steps/
Feature: Addition

  Scenario: 1 + 0
  ? Given I start with 1
  - When I add 0
  - Then I end up with 1

  Scenario: 1 + 1
  ? Given I start with 1
  - When I add 1
  - Then I end up with 2

Warnings:

1) Scenario: 1 + 0 - features/addition.feature:3
   Step: Given I start with 1 - features/addition.feature:4
   Step Definition: steps/maths.js:4
   Message:
     Pending

2) Scenario: 1 + 1 - features/addition.feature:8
   Step: Given I start with 1 - features/addition.feature:9
   Step Definition: steps/maths.js:4
   Message:
     Pending

2 scenarios (2 pending)
6 steps (2 pending, 4 skipped)
0m00.002s
```

Now to make it all work. All we need to do is implement the code in our step definitions. We're going to tidy up a little bit as well, to make things easier to read. This is essentially removing the need for the `callback` parameter since we're not doing anything asynchronous.

After this, our "steps/maths.js" will look like this:

```
const defineSupportCode = require('cucumber').defineSupportCode;
const assert = require('assert');

defineSupportCode(function({ Given, Then, When }) {
  let answer = 0;

  Given('I start with {int}', function (input) {
    answer = input;
  });
  When('I add {int}', function (input) {
    answer = answer + input;
  });
  Then('I end up with {int}', function (input) {
    assert.equal(answer, input);
  });
});
```

And executing it looks like this:

```
$ ./node_modules/.bin/cucumber.js features/ -r steps/
Feature: Addition

  Scenario: 1 + 0
   Given I start with 1
   When I add 0
   Then I end up with 1

  Scenario: 1 + 1
   Given I start with 1
   When I add 1
   Then I end up with 2

2 scenarios (2 passed)
6 steps (6 passed)
0m00.001s
```

Everything passes. We now know that addition works correctly.

Note that we only had to write a very little bit of code, and the Cucumber system glues it all together.
We got automatic parameterized tests by simply specifying how the step code is executed from the Gherkin files. This means that adding many more scenarios is really easy.

Next, let's prove that multiplication works as well. For this, we'll write the following Gherkin in `features/multiplication.feature`:

```
Feature: Multiplication

  Scenario: 1 * 0
    Given I start with 1
    When I multiply by 0
    Then I end up with 0

  Scenario: 1 * 1
    Given I start with 1
    When I multiply by 1
    Then I end up with 1

  Scenario: 2 + 2
    Given I start with 2
    When I multiply by 2
    Then I end up with 4
```

And then let's implement the new Step in our `steps/maths.js`. To do this we simply need to add the following block inside of the `defineSupportCode` method:

```
When('I multiply by {int}', function (input) {
  answer = answer * input;
});
```

That's it. Running this will give the following results:

```
$ ./node_modules/.bin/cucumber.js features/ -r steps/
Feature: Addition

  Scenario: 1 + 0
   Given I start with 1
   When I add 0
   Then I end up with 1

  Scenario: 1 + 1
   Given I start with 1
   When I add 1
   Then I end up with 2

Feature: Multiplication

  Scenario: 1 * 0
   Given I start with 1
   When I multiply by 0
   Then I end up with 0

  Scenario: 1 * 1
   Given I start with 1
   When I multiply by 1
   Then I end up with 1

  Scenario: 2 + 2
   Given I start with 2
   When I multiply by 2
   Then I end up with 4

5 scenarios (5 passed)
15 steps (15 passed)
0m00.003s
$
```

As simple as that, we've got a very easily extensible test suite that proves that maths works. As an exercise, why not try extending it to support subtraction as well? You can ask for help in the comments if you get stuck.

More Advanced Cucumber.js Tricks

This is all very good, but there are a number of more advanced things that Cucumber can do that will make our lives easier.

Asynchronous Step Definitions

So far we've only ever written synchronous step definitions. In the JavaScript world, this is often not good enough though. So much in JavaScript needs to be asynchronous, so we need some way to handle it.

Thankfully, Cucumber.js has a couple of built-in ways of handling this, depending on what you prefer.

The way that was hinted at above, which is the more traditional JavaScript way of handling asynchronous steps, is to use a callback function. If you specify that the step definition should take a callback function as its last parameter, then the step is not considered to be finished until this callback is triggered. In this case, if the callback is triggered with any parameters then this is considered to be an error and the step will fail. If it is triggered without any parameters then the step is considered to have succeeded. If, however, the callback is not triggered at all then the framework will eventually time out and fail the step anyway. The moral of the story? If you accept a callback parameter then make sure you call it.

For example, a step definition to make an HTTP API call using callbacks might look like the following. This is written using Request (https://github.com/request/request) since that uses callbacks on response.

```
When('I make an API call using callbacks', function(callbacks) {
  request('http://localhost:3000/api/endpoint', (err, response, body) => {
    if (err) {
      callback(err);
    } else {
      doSomethingWithResponse(body);
      callback();
    }
  });
});
```

The alternative, and preferred way is by return type. If you return a Promise from your step then the step will only be considered to have finished when the Promise is settled. If the Promise is rejected then the step will have failed, and if the Promise is fulfilled then the step will have succeeded.

Alternatively though, if you return something that is not a Promise, then the Step will immediately be considered to have succeeded. This includes returning `undefined` or `null`. This means that you can choose during the execution of the step whether you need to return a Promise or not, and the framework will adapt as needed.

For example, a step definition to make an HTTP API call using Promises might look like the following. This is written using the Fetch (https://www.npmjs.com/package/node-fetch) API since that returns a Promise on response.

```
When('I make an API call using promises', function() {
  return fetch('http://localhost:3000/api/endpoint')
    .then(res => res.json())
    .then(body => doSomethingWithResponse(body));
});
```

Feature Background

A feature background is a Gherkin snippet that is prepended to the beginning of every scenario in the file. This allows for common setup steps to be easily shared between every scenario without needing to repeat them.

Backgrounds are written by using the `Background` keyword instead of the `Scenario` keyword. Ideally there should only be Given steps included, since it makes no sense to include When or Then steps that are shared between every test. However, the framework will not restrict you in this, so be careful in how you structure your tests.

Using this, we can re-write our Addition feature as follows:

```
Feature: Addition

  Background:
    Given I start with 1

  Scenario: 1 + 0
    When I add 0
    Then I end up with 1

  Scenario: 1 + 1
    When I add 1
    Then I end up with 2
```

This is actually exactly the same as before, but it is slightly shorter since we have factored out the common setup step.

Scenario Outlines

Scenario outlines are a way of generating scenarios from a table of test data. This allows for parameterised testing in an even more efficient way than before, since we can have the exact same test script repeated many times with different values inserted.

Scenario outlines are written by using the `Scenario Outline` keyword instead of the `Scenario` keyword, and then providing one or more `Examples` tables. The parameters from the `Examples` tables are then substituted into the `Scenario Outline` to produce scenarios that are run.

Using this, we can re-write our Multiplication feature as follows:

```
Feature: Multiplication

  Scenario Outline: <a> * <b>
    Given I start with <a>
    When I multiply by <b>
    Then I end up with <answer>

  Examples:
    | a | b | answer |
    | 1 | 0 | 0      |
    | 1 | 1 | 1      |
    | 2 | 2 | 4      |
```

Again, this is exactly the same as before but it has significantly less repetition. You'll actually see if you run this that it generates the exact same Scenarios as before in the output:

```
Feature: Multiplication

  Scenario: 1 * 0
   Given I start with 1
   When I multiply by 0
   Then I end up with 0

  Scenario: 1 * 1
   Given I start with 1
   When I multiply by 1
   Then I end up with 1

  Scenario: 2 * 2
   Given I start with 2
   When I multiply by 2
   Then I end up with 4
```

Data Tables

We've just seen a table used in a scenario outline, to generate the data from which we can generate scenarios.
However, we can use data tables inside of scenarios as well. These can be used as a way of providing tables of data, or structured input, or many other things.

For example, the Addition scenario could be re-written to add an arbitrary number of values as follows:

```
Scenario: Add numbers
  Given I start with 0
  When I add the following numbers:
    | 1 |
    | 2 |
    | 3 |
    | 4 |
  Then I end up with 10
```

For this simple example, the step will look something like this:

```
When('I add the following numbers:', function (table) {
  answer = table.raw()
    .map(row => row[0])
    .map(v => parseInt(v))
    .reduce((current, next) => current + next, answer);
});
```

The `table` parameter we are provided is a `DataTable` object, which has a `raw` method on it that you can call. This method returns a 2D array of all the values in the data table, such that each entry in the outer array is a row in the table, and each entry in the inner array is a cell from that Row — as a String.

A more complex example might be using a Data Table to populate a form. This could then use the table to provide all of the inputs, rather than having a very hard to read step definition. This could read something like:

```
Scenario: Create a new user
  When I create a new user with details:
    | Username | graham               |
    | Email    | my.email@example.com |
    | Password | mySecretPassword     |
  Then the user is created successfully
```

In this case, the data table class can give us easier access to the table by use of the `rowsHash` method.

Our step for this might look like:

```
When('I create a new user with details:', function (table) {
  const data = table.rowsHash();
  createUser(data);
});
```

In the case, the `data` object will have been parsed from the data table and will look like:

```
{
  "Username": "graham",
  "Email": "my.email@example.com",
  "Password": "mySecretPassword"
}
```

Making access to the fields very easy by the keys in the first column.

Hooks

Like most test frameworks, Cucumber.js has support for hooks (https://github.com/cucumber/cucumber-js/blob/master/docs/support_files/hooks.md) that are executed before and after a scenario is run.

These are set up in the same way that the step definitions are, and are simply called as the name describes — before or after the scenario runs, regardless of success or failure.

As a simple example, to make our maths features more reliable we can do the following:

```
defineSupportCode(function({ Before, Given, Then, When }) {
  let answer;

  Before(function() {
    answer = 0;
  });
});
```

Enhancing our mathematics steps file as above will guarantee that the `answer` variable is reset to 0 before each scenario is run, meaning that we don't need a Given step if we're starting from 0.

If you need it, the first parameter to these hooks is always the scenario result for the scenario that the hook is running before or after. This can be used to adapt the functionality to the scenarios that are being run.

Hooks can be made asynchronous in exactly the same way as step definitions, by accepting a callback function as a second parameter or by returning a Promise.

Events

If the simple before and after hooks aren't enough for you, for whatever reason, then there are many more events to explore (https://github.com/cucumber/cucumber-js/blob/master/docs/support_files/event_handlers.md). These give us the ability to handle:

- BeforeFeatures – called once before anything is run at all, provided with the list of features.
- BeforeFeature – called before each Feature file is run, provided with the Feature.
- BeforeScenario – called before each Scenario is run, provided with the Scenario. This is roughly analogous to the "Before" hook.
- BeforeStep – called before each Step is run, provided with the Step.
- StepResult – called after each Step is run, provided with the result of the step.
- AfterStep – called after each Step is run, provided with the Step.
- ScenarioResult – called after each Scenario is run, provided with the result of the scenario.
- AfterScenario – called after each Scenario is run, provided with the Scenario. This is roughly analogous to the "After" hook.
- AfterFeature – called after each Feature is run, provided with the Feature.
- FeaturesResult – called once after everything is run, provided with the result of running everything.
- AfterFeatures – called once after everything is run, provided with the list of features.

These give full interaction with the entire lifecycle of the test framework, and will be called in the order listed above.

Handling these events is done using the `registerHandler` method from the `defineSupportCode` method. This could look something like this:

```
defineSupportCode(function({ registerHandler }) {
  registerHandler('BeforeStep', function(step) {
    console.log('About to execute step:' + util.inspect(step));
  });
  registerHandler('ScenarioResult', function(scenario) {
    console.log('Result of Scenario:' + util.inspect(scenario));
  });
});
```

(https://www.sitepoint.com/)
Event Handlers can be made asynchronous in exactly the same way as step definitions, by accepting a callback function as a second parameter or else by returning a Promise.

World – Sharing Code and Data

Until now, we've had no way of sharing code between steps. We can quite easily have as many JavaScript files containing step definitions, hooks, events, etc as we want, but they are all independent of each other (not counting tricks with the Node Module system to store state).

As it happens, this is not true. Cucumber.js has a concept of the "World", which is all of the state that a scenario is running with. All of the step definitions, hooks and event handlers have access to this by accessing the `this` parameter, regardless of the file that the step definition is defined in. This is why all of the examples are written using the traditional `function` keyword, instead arrow functions (https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions). Arrow functions in JavaScript rebind the `this` variable for you, which means that you lose access to the World state that you might need in your tests.

This works as-is with no extra handling needed, so you can just make use of it straight away. Immediately this means that we can have our code much cleaner, by splitting the Cucumber code logically between multiple files and have it all work as expected, whilst still having access to a shared state.

Summary

Behavior Driven Development is a fantastic way of ensuring that your product has the correct behavior, and Cucumber as a tool is a very powerful way of implementing this such that every stakeholder in the product can read, understand and maybe even write behavior tests.

This article is only scratching the surface of what what Cucumber is capable of so I encourage you to try it yourself to get a feel for its power. Cucumber also has a very active community and their mailing list (https://groups.google.com/forum/#!forum/cukes) and Gitter channel (https://gitter.im/cucumber/) are great ways to seek help, should you need it.

Are you already using Cucumber? Has this article encouraged to you to give it a try? Either way, I'd love to hear from you in the comments below.

This article was peer reviewed by Jani Hartikainen (https://www.sitepoint.com/author/jhartikainen/). Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

Meet the author

Graham Cox (https://www.sitepoint.com/author/gcox/) (https://twitter.com/grahamcox82) (https://www.reddit.com/user/sazzer) (https://github.com/sazzer)

Graham Cox is a Software Developer from the UK who has been writing software for almost 15 years, predominantly using Java and Javascript but also covering a wide variety of other languages.

Login or Create Account to Comment

(https://www.sitepoint.com/)

Login (/Premium/Sign-In?Ref_source=Sitepoint&Ref_medium=Comments&Redirect_path=%2Fbdd-Javascript-Cucumber-Gherkin%2F%23commentsSection)

Create Account (/Premium/Sign-Up?Ref_source=Sitepoint&Ref_medium=Comments&Redirect_path=%2Fbdd-Javascript-Cucumber-Gherkin%2F%23commentsSection)

Popular In the Community

A GUIDE TO SETTING UP LET'S ENCRYPT SSL ON...
Chirag Bhansali
6 Sep
What is a zone editor? I couldn't find one in...

THE ULTIMATE BEGINNER'S GUIDE TO...
downingjohnw
30 Dec
Intrusive ads are one thing. Intrusive ads...

UX LESSONS FROM AMAZON: 4 HACKS...
squibs
31 Aug
There's no doubt that Amazon leads the...

USING PREACT AS A REACT ALTERNATIVE —...
Phil Mander
12 Dec
Thanks for the article Ahmed, I was lookin...

REACTJS IN PHP: WRITING COMPILERS IS EASY AN...
Christopher Pitt
4 Sep
Writing a compiler isn't the same as...

A COMPARISON OF SHARED AND CLOUD...
Surja
23 Aug
Thank you for writing this helpful article. G...

HOW TO BUILD A REAC APP THAT WORKS WITH
Hugo Hyz
7 Dec
Awesome content! I'm looking to transfert..

Conversation (1)

Sort by **Oldest**

Add a comment...

zack 赵
https://github.com/cucumber/cucumber-js/blob/master/docs/support_files/event_handlers.md
while cucumber upgread 3.x.x, has disable the event_handlers ,so we can get the runtime of scenario Just use {Before or After or BeforeAll or AfterAll}

Reply · Share ·

Terms · Privacy

Add Spot.IM to your site ·

Stuff We Do

- Premium (/premium/)
- Versioning (/versioning/)
- Themes (/themes/)
- Forums (/community/)
- References (/html-css/css/)

About

- Our Story (/about-us/)
- Press Room (/press/)

Contact

- Contact Us (/contact-us/)
- FAQ (https://sitepoint.zendesk.com/hc/en-us)
- Write for Us (/write-for-us/)
- Advertise (/advertise/)

Legals

- Terms of Use (/legals/)
- Privacy Policy (/legals/#privacy)

Connect

(https://www.facebook.com/sitepoint) (http://twitter.com/sitepointdotcom) (/versioning/) (https://www.sitepoint.com/feed/) (https://plus.google.com/+sitepoint)

(https://www.sitepoint.com/)