Component testing using WCT

What is it?

It's a browser-based testing environment with:

- text-fixture
- Mocha
- Chai
- Sinon

How to set it up?

You need to load `browser.js` before running your tests, so you need to install it in your project:

```
bower install Polymer/web-component-tester --save
```

And then load it in your .html files:

```
<script src="../../web-component-tester/browser.js"></script>
```

How to run the tests?

You can run them with the `wct` tool which you can install:

```
npm install -g web-component-tester
```

And then run it:

```
wct
```

It will run all the tests in all the browsers installed in your machine.

How to run the tests?

You can also run the tests using a `web server` like the one that comes with Polymer CLI:

```
polymer serve -o
```

Advantages of running them with a web server:

- You don't need to install `selenium webdrivers` every time you run `wct`.
- You can easily debug your tests.

Adding Suites to WCT

You can load more than one suite of tests to WCT using `WCT.loadSuites()`

```
WCT.loadSuites([
    'test-training_test.html',
    'test-training_test.html?dom=shadow',
    'fire-event_test.html',
    'fire-event_test.html?dom=shadow'
]);
```

Test Fixtures

It helps us resetting the test Suite's DOM.

```
<test-fixture id="BasicTestFixture">
  <template>
    <test-training></test-training>
  </template>
</test-fixture>
```

And then before each test we reset it:

```
beforeEach(() => {
        element = fixture('BasicTestFixture');
});
```

Mocha

We have the following functions to create Suites:

- `describe(string, function)`
- `context(string, function)`

This functions contains one or more Specs which can be defined with:

- `it(string, function)`
- `specify(string, function)`

Mocha

> *context* is an alias of *describe* and *specify* is and alias of *it* so we strongly recommend to only use *describe* and *it*.

```
describe('button', function(){
    context('when it is focused', function() {
        it('should have the class focused', function() {
                ...
        });
    });
    describe('when it is not focused', function() {
        specify('should not have the class focused', function()
                ...
        });
    });
});
```

Skipping tests in Mocha

If you need to skip some test to debug a specific one you can do it like this:

- Suites:

```
describe.skip('button', function() {...});
xdescribe('button', function() {...});
```

- Specs:

```
it.skip('should have the class focused', function() {...});
xit('should have the class focused', function() {...});
```

> This syntax also works with *context* and *specify*

Mocha Hooks Lifecycle

We can use the following functions to make things before or after each test:

- `before(function)`, only runs one time before a suite or spec starts.
- `beforeEach(function)`, runs everytime before a suite or spec starts.
- `after(function)`, runs one time after a suite or spec ends.
- `afterEach(function)`, runs everytime after a suite or spec ends.

Chai

Give us expectations which evaluate to `true` or `false`:

```
expect(Array.from(button.classList)).to.include('focused');
```

Chai's API

Sinon's spies (API)

A test spy is a function that records arguments and returns values for all its calls.

```
it('should fire two events when the button is tapped twice', ()

    const spy = sinon.spy();

    element.addEventListener('my-fired-event', spy);

    MockInteractions.tap(button);
    MockInteractions.tap(button);

    expect(spy.calledTwice).to.be.true;
});
```

Sinon's spies (API)

Normally, you'll want to spy on a function that already exists.

```
sinon.spy(console, 'log');
```

It will behave as the original function but you will be able to know if it was called, with which arguments...

```
console.log.calledOnce
```

But you will have to restore it after the test ends.

```
afterEach(function() {
    console.log.restore();
});
```

Sinon's stubs (API)

They are similar to sinon's spies but with a custom behavior.

```
beforeEach(() => {
    const response = new Response(JSON.stringify(getPersons()),
    });

    sinon.stub(window, 'fetch');

    fetch.returns(Promise.resolve(response));
});
```

After using it you will have to restore it:

```
afterEach(() => {
    fetch.restore();
});
```

Sinon's sandbox (API)

With sandbox you will not have to restore every spy and stub manually instead you will be able to only restore the sandbox and it will restore every spy and stub.

```
let sandbox;

before(() => {
    sandbox = sinon.sandbox.create();
});
```

```
afterEach(() => {
    sandbox.restore();
});
```

```
sandbox.spy(element, '_showResultInConsole');
```

iron-test-helpers

It's a set of utility classes that help us to make tests.

1. Install it:

```
bower install --save-dev PolymerElements/iron-test-helpers
```

2. Import them using html import (`<link rel="import">`):

TestHelpers:

`../../iron-test-helpers/test-helpers.html`

MockInteractions:

`../../iron-test-helpers/mock-interactions.html`

Test DOM mutations

When your element mutate (it uses dom-repeat, dom-if or a slot) you will have to wrap your test using `flush` function and mark the spec as an asynchronous one.

```
it('should show the paragraph ...', (done) => {
    element.set('showProperty', true);

    flush(() => {
        // we get the paragraph
        const showParagraph = element.shadowRoot.querySelector('

        expect(element.showProperty).to.be.true;
        expect(showParagraph).to.not.be.null;

        done();
    });
});
```

Testing async calls

You can use `Polymer.Base.async` to wait a certain amount of time:

```
it('should have the expected number of ...', (done) => {
    Polymer.Base.async(() => {
        expect(fetch.calledOnce).to.be.true;
        expect(element.persons).to.not.be.empty;
        expect(element.persons).to.be.deep.equal(getPersons());
        done();
    }, 100);
});
```

Polymer.Base