Testing Basics

We discuss here some basic ideas around testing, that you will further work on in the lab and in future labs and projects.

Testing Basics

When we refer to tests, we typically mean automated tests. Those tests fall into various categories:

Unit Tests Unit tests test a single tiny individual component of your application. You typically want to have unit tests for every bit of your code that is part of your application's *interface*.

You should avoid testing for things that are too implementation-dependent.

Unit tests are a crucial part of the refactoring process, whose goal is to rearrange and rewrite sections of your code. A solid suite of unit tests can allow you to do this freely without worrying about breaking code. "Your tests will catch that". And revision control allows you to recover if you've messed things up too much.

Integration Tests Integration tests test bigger parts of your application, making sure that different parts come together naturally.

Time permitting, we will talk more about integration tests later in the term.

Timing Tests Timing tests are used in algorithm implementations to assess the efficiency of the algorithms.

They can also be used to try to find bottlenecks in your application, though some of the browser profiling tools might be better.

Deployment Tests Deployment tests are meant to ensure that your application performs well on various browsers / deployment environments. Hard to do.

We will focus on unit tests for now.

Test-Driven Development

In Test-Driven-Development, you typically would follow these steps:

- Decide on a small piece of functionality you want to add.
- Make a GitHub issue about it.
  - If you prefer, you can create one bigger more "logical" issue, and create a "task list" in it, following the example at this blog post[1]. Then check those items off as you implement them.

[1] https://github.com/blog/1375%0A-task-lists-in-gfm-issues-pulls-comments

- Write a test for the code you want to introduce.
- Run your tests, and watch this new test fail. This makes us more certain that the test does indeed detect the feature we want to add.
- Optionally, make a git commit of the test, using "#..." to reference the issue you created.
  - This is a bit of a style decision, whether to commit the tests separately or whether to do one commit containing both test and new code.
- Write a minimal set of code that would make the test pass.
- Check that all your tests pass.
- Make a commit, using "#..." to reference the issue you created. Say "close #..." if it was a "single-problem issue".
  - This gives you a safe backup point to revert to.
- Consider any refactoring that you might want to have take place.
- Do the refactoring, and make sure your tests still all pass.
- Commit (optionally creating an issue first to explain what the refactoring was about).

This is some of the general theory behind testing. The lab readme will have more specific instructions.

Testing in Javascript

We will be using mocha[2] in combination with chai[3] for our unit tests. Here is a basic html file:

```
<!DOCTYPE html>
<html>
<head>
   <link rel="stylesheet" href="mocha.css" />
</head>
<body>
   <div id="mocha"></div>
   <script src="chai.js" type="text/javascript"></script>
   <script src="mocha.js"></script>
   <script>
      mocha.setup('bdd');
      var expect = chai.expect;
   </script>
   <script src="yourCode.js"></script>
   <script src="tests.spec.js"></script>
   <script>
      mocha.checkLeaks();
      mocha.globals(['chai']);
      mocha.run();
  </script>
</body>
</html>
```

[2] http://mochajs.org/
[3] http://chaijs.com/

All the tests go into tests.spec.js (or a similarly named file). You can also use multiple test files, just put them after each other. Use the rest as a template on every project.

mocha and chai together offer all the primitives we need for testing. We will describe them briefly in the following section. Refer to their homepages for more info.

Mocha and Chai

Mocha provides a couple of basic functions for us. We use them to structure our test suites.

Starts a new test suite. First argument is a string "title". Second is an anonymous argumentless function that contains the tests.

The string titles together with the titles from the "it" sections should form sentences asserting what the test does.

You can have nested "describe"s.

Starts a new unit test. Should only appear within the function in a "describe". First argument is a string "test title". Together with the "describe"'s string, it should form a sentence.

Second argument is an anonymous argumentless function. It will contain the statements that comprise the test.

They go in a "describe" and they take as argument an anonymous argumentless function. They will run this function before starting (respectively after completing) the tests. Look for "hooks" in mocha doc.

Similar, except that they run before/after *each* test in the describe, rather than only once. Look for "hooks" in mocha doc.

You can force the running of a single test, or test suite, by using this. Look for "exclusive tests" in mocha doc.

Can be used to tell it to skip the specific test. Look for "inclusive tests" in mocha doc.

Chai is used to write tests. We will be using its "expect" interface, whose API is here[4].

The start point. expect(something).to.. is the basic structure. Everything else is "chained" at the end of the expect.

be a series of "words" that have no meaning but are used to form something that looks more like a sentence: "to", "be", "is", "that", "have", "has", some more.

negates the test.

[4] http://chaijs.com/api/bdd/

for array comparisons, it will test each matching element. So two arrays would be equal if they have equal elements, even if they are different objects.
checks for typeof.
checks that target includes the value.
tests that the target equals the value.
[illegible]. Look in the Chai API[5].

[5] http://chaijs.com/api/bdd/