Protractor Style Guide

Some opinionated guidelines for those out there looking for testing best practices with protractor

This style guide is a set of opinionated rules and best practices about writing e2e tests with Protractor. These are all learnings that my team and I came across while working with many other teams, on improving their testing related codebase and achieving some sort of code uniformity across teams. If you are familiar with John Papa's Angular Style Guide, which I highly recommend, think of this project as something similar, but then for Protractor.

The examples and use cases mentioned in this guide are all related to AngularJS applications, but most things are equally valid for non-Angular web applications and their e2e tests.

I wanted to share this project with the community in the hope that these learnings will make life easier for other developers, give them some sort of guidance and help them write better, cleaner Protractor tests, faster. If there is one person for whom this guide made a difference, I am very happy ツ

Examples

I always thought that an example is worth a thousand words, so I am trying to put together as many examples as I can, that can better showcase and clarify the rules explained here. There will be more examples added with time, but if you think you have one that I missed out on, head to the Contributing section.

Running the examples

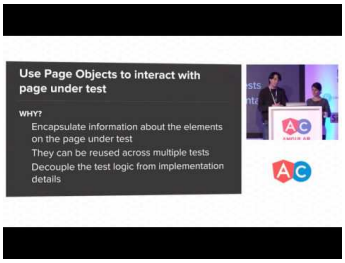
// TODO

Contributing

If you want to contribute to this project and make it better, send a pull request or submit an issue and you'll definitely get some extra ♥ for your help.

Video

Andres Dominguez and I gave a talk about this style guide at AngularConnect in London. Here's the video in case you want to watch it.



Credit

I want to thank Yun, Rene and my awesome team, who have encouraged and supported me in developing this guide as part of my work. Special thanks to Andres for sharing his knowledge and contributing to this project.

Last but not least, this project wouldn't mean anything without the support and help of the Angular community. You are awesome! -`ღ´-

And one last thing ...

#useYourSuperPowersToDoGood

Table of Contents

A word on testing

Before we get into the actual rules and best practices themselves, I would like to take a moment to talk about testing in general, what it means and why it is so important in the development process of an application.

If I look back at how I started with testing, I can remember very well that the beginning was anything but easy. At that time, I had a vague idea that it 'must be a good thing', but I didn't fully understand what it was all about. Everything seemed overly complicated, time consuming and just a big overhead all in all. This is a sentiment that I have heard many times from a lot of developers that start with testing, and it is also something that pushes many of them away from writing automated tests.

My **AHA!** moment was during the first major refactoring of our codebase, when I understood the true power of having automated tests. It was from that moment on that I stopped looking at writing tests as a chore but rather as something that is equally important as writing the application code itself.

Since then I have done a lot of reading about it and come to realise that testing is a whole world in itself, with its own discourse, theory and practices. And just like any other thing that we developers want to master, it takes time and reading and practice. I like to advocate for testing, because I truly believe in its benefits and in what it stands for. I could not imagine writing code without its accompanying tests anymore, and this is something that I like to encourage others to do every time I get the chance.

So why is testing so important ultimately?

Well first of all because tests will tell you if your application is working as it should. In my opinion this is the first and most important benefit of having automated tests. Successful tests will give you the confidence that everything in your application runs properly. They will give you the confidence that any changes in your codebase, no matter how small and insignificant they might seem at a first glance, don't break the behavior of your system. And all this at a very early stage of the development process. Just imagine introducing a set of changes to your codebase and having to manually test whether those changes didn't break any other parts of the application. Kind of makes you want to go to the corner and cry, right? This is precisely the point where automating testing becomes so powerful and important to have.

Automated tests also serve as documentation. I have often heard developers new to a project say that, having read the tests of the project, has helped them get insights of the application much easier and faster. And this is very true. If written properly, tests will express in clear and understandable terms what the expected behavior of the application is. This makes it very easy for new contributors to join a project, or why not be honest about it, for you to remember what that piece of code you wrote ages ago is supposed to do ;).

The last thing I want to mention here is my personal belief and experience that writing tests makes us better developers. Tests will force us to think twice about what our code really has to do, about corner cases and whether we handle all of them, they will teach us about our programming soft spots and make us learn from our mistakes. It's kind of like a "think before you speak" thing, if you will.

I have often seen teams trying to 'cheat' on their tests by writing dummy specs that always complete successfully. I have a hard time understanding why anyone would go down that path, but the reason I hear most frequently is 'code coverage'. Now, I don't believe in writing tests just for the sake of reaching some coverage thresholds, that most likely you didn't have any say in to begin with, no more than I believe that you can have the cookie and eat it too. If you ever find yourself drifting that path, just remember that cheating on your tests is, at the end of the day, just cheating on yourself. Tests are there for a reason!

If you venture out into the testing literature, you might be overwhelmed by the many different types of tests authors refer to. This style guide is all about e2e tests (with Protractor), but because I think it is important to make the distinction, I want to say a few words about unit testing as well.

Unit Testing

Unit tests are the first line of defence against bugs and errors in your code. They ensure that the smallest parts of an application, the units, work as intended, by testing them in complete isolation from the rest of the application.

Let's take a car as an example. Unit testing a car would be taking each of its parts, like the wheels, or the engine, separately, and checking whether they work properly, independently from each other.

It is important to remember the "in complete isolation" aspect of unit testing. Instantiating services, making API calls, or even worse, instantiating the entire system (all very expensive operations), in order to test just one single unit, is unnecessary and definitely not a good practice. The general rule of thumb is to mock any external dependencies of the unit under test.

E2E Testing

E2E tests are the next common sense step after unit tests, that will allow you to trace bugs and errors in the behavior of your system. They are a means of testing if all units of an application interact as expected with each other and if the system as a whole works as intended.

Coming back to our previous car example, e2e testing the car would mean checking if all components of the car integrate well with each other and have the expected behavior as an overall. For instance you would want to know if the break lights turn on when the driver hits the break or if the wheels start turning when the driver steps on the acceleration pedal, and so on.

There is a lot of discussion around what parts of an application should or shouldn't be e2e tested and about how a proper system under test should be set up. Many developers who start with e2e testing are usually confused about what the 'right' way to do things is. Julie Ralph wrote a very good article about this, that I will link here once it is officially published, in which she explains very nicely the pros and cons of the options out there.

I for one like to think of e2e testing as testing from the perspective of the end user. Consider an email application for example. What is the functionality that this application provides to the end user? Can a user view their inbox? Can they write emails or delete them? Can they save drafts? How does the user interact with the application in order to perform these actions and what are the expected outcomes of each one of them? These are all questions that e2e tests should answer by simulating all these user interaction flows and making assertions about their outcomes and about what the user experiences throughout the process.

Protractor

If you've been in the Angular world for long enough, you'll probably remember about the [Angular Scenario Runner](https://code.angularjs.org/1.2.16/docs/guide/e2e-testing). The Scenario Runner was originally shipped with Angular as a tool to help developers e2e test their application. However, due to some design and maintenance issues, the team decided to not continue with its development and provide a better solution on the long term. Currently, the Angular Scenario Runner is deprecated and in maintenance mode (last time I checked ;)), so in case you are using it in your application, you might want to reconsider.

The "new" tool for e2e testing your AngularJS applications is Protractor. Protractor is an e2e test framework built on top of WebDriverJS, that comes with some Angular specific locator strategies on top of those offered by webdriverjs. You can read all about the project and the API specification at http://www.protractortest.org

Page Objects

Let's assume we have a very simple application that can answer any given question. Let's call it the "Grandfather of All Knowledge" app. The idea of the app is pretty straightforward: the user enters a question in an input field, presses a button and gets the answer to the question. Easy! ... and clever if you get the right answer ;)

The HTML markup for the application is as follows:

```
<!-- grandfather.html -->

<body ng-app="GrandfatherOfAllKnowledgeApp">
  <div class="question">
    <input class="question__field" ng-model="question.text"
           placeholder="What would you like to ask Grandfather of all knowledge?">
    <button class="question__button" ng-click="answerQuestion()"
            ng-disabled="!question.text">?</button>
  </div>
  <div class="answer">{{answer}}</div>
</body>
```

The e2e tests in this case should cover exactly the interaction we described earlier: user enters question, presses button and gets an answer. They look something like this:

```
/* grandfather.spec.js */

describe("The grandfather of all knowledge", function() {

    beforeEach(function() {
        browser.get('/#/grandfather-of-all-knowledge');
    });

    it('should answer any question', function() {
        var question = element(by.model('question.text'));
        var answer = element(by.binding('answer'));
        var button = element(by.className('question__button'));

        question.sendKeys("What is the purpose of meaning?");
        button.click();
        expect(answer.getText()).toEqual("Chocolate!");
    });

    it('should not allow empty questions', function() {
        var question = element(by.model('question.text'));
        var answer = element(by.binding('answer'));
        var button = element(by.className('question__button'));

        question.sendKeys("    ");
        expect(button.isEnabled()).toBeFalsy();
    });
});
```

Now, the fact that we are declaring the question/answer/button elements in each spec, adds a lot of duplicate code to our tests, and as we all know, duplicate code is generally not a good practice. Let's fix that!

```
/* grandfather.spec.js */

describe("The grandfather of all knowledge", function() {

    var question = element(by.model('question.text'));
    var answer = element(by.binding('answer'));
    var button = element(by.className('question__button'));

    beforeEach(function() {
        browser.get('/#/grandfather-of-all-knowledge');
    });

    it('should answer any question', function() {
```

```
        question.sendKeys("What is the purpose of meaning?");
        button.click();
        expect(answer.getText()).toEqual("Chocolate!");
    });

    it('should not allow empty questions', function() {
        question.sendKeys("    ");
        expect(button.isEnabled()).toBeFalsy();
    });
});
```

This is already better, but we can do even better than that. If you look at the tests we wrote, notice that they are dealing with multiple concerns at the same time: keeping track of the page under test, selecting elements from that page, defining the interaction with these elements, making the assertions and so on. Furthermore, imagine that the markup of the tested page changes, so for instance the class of our button would change from "question__button" to just simply "button". This means that we would have to revisit all the code in our tests where we declared the button and make sure the selector is updated to the correct class name. For large code bases, but not only, this is simply not maintainable.

This is precisely the point where **Page Objects** will come to the rescue. Page Object is a design pattern that is largely used in test automation for enhancing test maintenance and reducing code duplication. Page Objects provide an API to the page under test and are responsible of abstracting away its implementation details from the tests themselves.

There is one explanation of Page Objects by Simon Stewart that I particularly like and that has helped me a lot in understanding and working with the concept better. It goes like this:

> I think on Page Objects being like Janus-like objects (Janus is the Roman God with two heads that looks both ways). Looking one way, he introspects deeply into the HTML, he knows the page and how everything is hooked together and understands how to find a particular element. Looking the other way, he presents the services that the page offers to you.

Coming back to our Grandfather of all Knowledge application, let's see how a Page Object would look like:

```
/* grandfather.pageObject.js */

var GrandfatherOfAllKnowledge = function() {
    this.question = element(by.model('question.text'));
    this.answer = element(by.binding('answer'));
    this.button = element(by.className('question__button'));

    this.askQuestion = function(question) {
        this.question.sendKeys(question);
        this.button.click();
    };
};
module.exports = GrandfatherOfAllKnowledge;


/* grandfather.spec.js */

var GrandfatherOfAllKnowledge = require('./grandfatherPageObject');

describe("The grandfather of all knowledge", function() {

    var grandfatherOfAllKnowledge = new GrandfatherOfAllKnowledge();

    beforeEach(function() {
        browser.get('/#/grandfather-of-all-knowledge');
    });

    it('should answer any question', function() {
        grandfatherOfAllKnowledge.askQuestion("What is the purpose of meaning?");
        expect(grandfatherOfAllKnowledge.answer.getText()).toEqual("Chocolate!");
    });

    it('should not allow empty questions', function() {
        grandfatherOfAllKnowledge.askQuestion("    ");
        expect(grandfatherOfAllKnowledge.button.isEnabled()).toBeFalsy();
    });
});
```

Much cleaner right? Notice how nicely our concerns are separated now and how much more readable the tests have become. You will of course still have to make sure that your Page Objects are up to date with any changes in the markup they serve, but since that's all encapsulated in one place, the task is much more manageable.

Style Guide Rules

Generic Rules

[Rule-01: Do not cheat on your tests]

If you're skipping tests or writing dummy ones just to pass that good old coverage check, you're on the road to perdition. I'm not necessarily advocating for 100% code coverage, though that would be nice, but at least make sure that all your use cases have a minimum amount of tests and that the major possible breaking points are covered.

Why?

- Tests are there for a reason. They are there to help you track errors and bugs in your code
- One test less can possibly mean one more bug shipped to production
- The 'I will test it manually later' thing doesn't work
- The 'I wrote the code myself, I'm sure it works' thing doesn't work either

[Rule-02: Prefer unit over e2e tests when possible]

Try to cover as much as possible of your logic with unit tests rather than with e2e tests. If you have doubts whether a certain part of your code should be unit or e2e tested, if it makes sense to unit test it, always prefer that over e2e tests.

Why?

- Unit tests are much faster than e2e tests

[Rule-03: Don't e2e test what has already been unit tested]

If you already have unit tests for a particular behavior, there is no need to write more e2e tests for that exact same part of the code. It might be that the same behavior will be hit as part of the flow under test in one of your e2e tests, but having two dedicated tests for the same piece of code is not necessary

Why?

- Avoid duplicate tests

[Rule-04: Use one configuration file]

Any protractor test suite starts out with a protractor.conf.js file, which basically is, as the name suggests, configuration code for Protractor written in javascript. Most of the times you will want to run your tests against multiple environments, not just a single one. In order to do that, it's a better practice to declare the different targets you want to test against via your task runner and keep the shared configuration in one Protractor config file, rather than have a configuration file for each of your environments. Just because a file contains configuration, doesn't mean that it isn't code and shouldn't be treated as such.

Why?

- Avoid code duplication
- You can create spin-offs using a task runner

```
/* avoid */
protractor.conf.local.js
protractor.conf.dev.js
protractor.conf.test.js



/* recommended */

/* protractor.conf.js */
exclude: [],
multiCapabilities: [{
  browserName: 'chrome',
  shardTestFiles: true,
  maxInstances: 3
}],
```

```
  allScriptsTimeout: 11000,
  getPageTimeout: 10000,
  framework: 'jasmine',
  jasmineNodeOpts: {
    isVerbose: false,
    showColors: true,
    includeStackTrace: false,
    defaultTimeoutInterval: 40000
  }



  /* recommended */

  /* Gruntfile.js */
  // grunt-protractor-coverage config

  e2e.local: {
    options: {
      args: {
        baseUrl: local_base_url,
        seleniumAddress: local_Se,
        specs: ['..../**/*.spec.js']
        }
      }
    },
    e2e.dev: {
      options: {
        args: {
          baseUrl: dev_base_url,
          seleniumAddress: dev_Se,
          specs: ['..../**/*.spec.js']
        }
      }
    }
```

Project Structure

[Rule-05: Group your e2e tests in a structure that makes sense to the structure of your project]

Why?
* Finding your e2e related files should be intuitive and easy
* Makes the folder structure more readable
* Clearly separates e2e tests from unit tests

Small scale Angular apps

```
/* avoid */
|-- project-folder
  |-- app
      |-- css
      |-- img
      |-- partials
          home.html
          profile.html
          contacts.html
      |-- js
          |-- controllers
          |-- directives
          |-- services
          app.js
          ...
      index.html
  |-- test
      |-- unit
      |-- e2e
          home.pageObject.js
          home.spec.js
          profile.pageObject.js
          profile.spec.js
          contacts.pageObject.js
          contacts.spec.js
```

```
/* recommended */
|-- project-folder
```

```
    |-- app
        |-- css
        |-- img
        |-- partials
              home.html
              profile.html
              contacts.html
        |-- js
            |-- controllers
            |-- directives
            |-- services
            app.js
            ...
        index.html
    |-- test
        |-- unit
        |-- e2e
            |-- page-objects
                  home.pageObject.js
                  profile.pageObject.js
                  contacts.pageObject.js
            home.spec.js
            profile.spec.js
            contacts.spec.js
```

Large scale Angular apps

```
/* avoid */
|-- project-folder
  |-- app
      |-- home
            home.html
            home.module.js
            home.controller.js
      |-- profile
            profile.html
            profile.module.js
            profile.controller.js
      |-- contacts
            contacts.html
            contacts.module.js
            contacts.controller.js
      app.module.js
      app.controller.js
      app.css
      index.html
  |-- test
      |-- unit
      |-- e2e
          home.pageObject.js
          home.spec.js
          profile.pageObject.js
          profile.spec.js
          contacts.pageObject.js
          contacts.spec.js
```

```
/* recommended */
|-- project-folder
  |-- app
      |-- home
            home.html
            home.module.js
            home.controller.js
      |-- profile
            profile.html
            profile.module.js
            profile.controller.js
      |-- contacts
            contacts.html
            contacts.module.js
            contacts.controller.js
      app.js
      app.module.js
      app.controller.js
```

```
        app.css
        index.html
    |-- test
        |-- unit
        |-- e2e
            |-- home
                  home.pageObject.js
                  home.spec.js
            |-- profile
                  profile.pageObject.js
                  profile.spec.js
            |-- contacts
                  contacts.pageObject.js
                  contacts.spec.js
```

Locator Strategies

[Rule-06: Never use xpath]

Why?
* It's the slowest and most brittle locator strategy of all
* Markup is very easily subject to change and therefore xpath locators require a lot of maintenance
* xpath expressions are unreadable and very hard to debug

```javascript
 /* avoid */
 var elem = element(by.xpath('/*/p[2]/b[2]/following-sibling::node()' +
  '[count(./*/p[2]/b[2]/following-sibling::br[1]/preceding-sibling::node())' +
  '=' +
  ' count((/*/p[2]/b[2]/following-sibling::br[1]/preceding-sibling::node()))' +
  ']'));
```

[Rule-07: Prefer Protractor locators when possible]

Why?

- They are very specific locators
- Access elements easier
- Expressions are less likely to change than markup
- Simple and readable locators

```
<ul class="red">
  <li>{{color.name}}</li>
  <li>{{color.shade}}</li>
  <li>{{color.code}}</li>
</ul>
```

```
/* avoid */
var nameElem = element.all(by.css('.red li')).get(0);
```

```
/* recommended */
var nameElem = element(by.binding('color.name'));
```

[Rule-08: Prefer by.id and by.css when no Protractor locators are available]

Why?

- Both are very performant and readable locators
- Access elements easier

[Rule-09: Avoid text locators for text that changes frequently]

Why?

- Text for buttons, links, and labels tends to change over time

- Your tests should not break when you make minor text changes

Page Objects

[Rule-10: Use Page Objects to interact with page under test]

Why?

- They encapsulate information about the elements on the page under test
- Can be reused across multiple tests
- Decouple the test logic from implementation details

```
/* avoid */

/* question.spec.js */
describe('Question page', function() {
  it('should answer any question', function() {
    var question = element(by.model('question.text'));
    var answer = element(by.binding('answer'));
    var button = element(by.css('.question-button'));

    question.sendKeys('What is the purpose of life?');
    button.click();
    expect(answer.getText()).toEqual("Chocolate!");
  });
});



/* recommended */

/* question.spec.js */
var QuestionPage = require('./question.page');

describe('Question page', function() {
  var question = new QuestionPage();

  it('should ask any question', function() {
    question.ask('What is the purpose of meaning?');
    expect(question.answer.getText()).toEqual('Chocolate');
  });
});



/* recommended */

/* question.page.js */
var QuestionPage = function() {
  this.question = element(by.model('question.text'));
  this.answer = element(by.binding('answer'));
  this.button = element(by.className('question-button'));

  this.ask = function(question) {
    this.question.sendKeys(question);
    this.button.click();
  };
};
module.exports = QuestionPage;
```

[Rule-11: UpperCamelCase the names of your Page Objects]

Why?

- By definition, a Page Object is an object-oriented class and therefore all class naming conventions apply to it

```
/* avoid */
var grandfatherOfAllKnowledge = function() {};
var grandfather-of-all-knowledge = function() {};
var grandfather_of_all_knowledge = function() {};



/* recommended */
var GrandfatherOfAllKnowledge = function() {
```

```
  /*...*/
};
```

[Rule-12: Pick a descriptive file naming convention for your Page Object files]

I've seen both 'page' or 'pageObject' being used and as far as I am concerned, they are equally fine

Why?

- This will ensure that your Page Object related files are easily recognisable and distinguishable from all the other test files

```
/* avoid */
  |-- test
      |-- unit
      |-- e2e
          |-- home
              |-- home.js
              |-- home.spec.js
          |-- profile
              |-- profile.js
              |-- profile.spec.js
          |-- contacts
              |-- contacts.js
              |-- contacts.spec.js
          |-- archive
              |-- archive.js
              |-- archive.spec.js
```

```
/* recommended */
  |-- test
      |-- unit
      |-- e2e
          |-- home
              |-- home.pageObject.js
              |-- home.spec.js
          |-- profile
              |-- profile.pageObject.js
              |-- profile.spec.js
          |-- contacts
              |-- contacts.pageObject.js
              |-- contacts.spec.js
          |-- archive
              |-- archive.pageObject.js
              |-- archive.spec.js
```

[Rule-13: Declare one Page Object per file]

Why?

- Keeps code clean and makes things easy to find

[Rule-14: Use a single module.exports at the end of the Page Object file]

Why?

- One Page Object per file means there's only one class to export

```
/* avoid */

/* user-profile.page.js */
var UserProfilePage = function() {};
var UserSettingsPage = function() {};

module.exports = UserPropertiesPage;
module.exports = UserSettingsPage;
```

```
/* recommended */

/* user-profile.pageObject.js */
var UserProfilePage = function() {};
module.exports = UserPropertiesPage;
```

```
/* user-settings.pageObject.js */
var UserSettingsPage = function() {};
module.exports = UserSettingsPage;
```

[Rule-15: Require and instantiate all the modules at the top]

Why?

- The module dependencies should be clear and easy to find
- Separates dependencies from the test code
- Makes the dependencies available to all specs of the suite

```
/* avoid */

/* user-properties.spec.js */
var UserPage = require('./user-properties.page');
var MenuPage = require('./menu.page');
var FooterPage = require('./footer.page');

describe('User properties page', function() {
  var user = new UserPage();
  var menu = new MenuPage();
  var footer = new FooterPage();
  // specs
});
```

[Rule-16: Declare all public elements in the constructor]

Why?

- The consumer of the Page Object should have quick access to the available elements on a page

```
<form>
   Name: <input type="text" ng-model="ctrl.user.name">
   E-mail: <input type="text" ng-model="ctrl.user.email">
   <button id="save-button">Save</button>
</form>
```

```
/* recommended */
var UserPropertiesPage = function() {
  this.name = element(by.model('ctrl.user.name'));
  this.email = element(by.model('ctrl.user.email'));
  this.saveButton = element(by.id('save-button'));
};
```

[Rule-17: Declare functions for operations that require more that one step]

Why?

- Most elements are exposed by the Page Object and can be used directly in the test
- Doing otherwise adds unnecessary complexity

```
/* avoid */

/* user-properties.page.js */
var UserPropertiesPage = function() {
  this.name = element(by.model('ctrl.user.name'));
  this.saveButton = element(by.id('save-button'));

  this.enterName = function(name) {
    this.name.sendKeys(name);
  };
};


/* user-properties.spec.js */
var UserPage = require('./user-properties.page');
```

```
    describe('User properties page', function() {
      var user = new UserPage();

      it('should enable save button when a username is entered', function() {
         user.enterName('TeddyB');
         expect(user.saveButton.isEnabled()).toBe(true);
       });
    });
```

```javascript
    /* recommended */

    /* user-properties.page.js */
    var UserPropertiesPage = function() {
      this.name = element(by.model('ctrl.user.name'));
      this.saveButton = element(by.id('save-button'));
    };


    /* user-properties.spec.js */
    var UserPage = require('./user-properties.page');

    describe('User properties page', function() {
      var user = new UserPage();

      it('should enable save button when a username is entered', function() {
         user.name.sendKeys('TeddyB');
         expect(user.saveButton.isEnabled()).toBe(true);
       });
    });
```

[Rule-18: Do not add any assertions in your Page Object definitions]

Martin Fowler has a very good thought on this one:

> Page Objects are commonly used for testing, but should not make assertions themselves. Their responsibility is to
> provide access to the state of the underlying page. It's up to test clients to carry out the assertion logic.

Why?
* It is the responsibility of the test to do all the assertions
* Reader of the test should be able to understand the behavior of the application by looking at the test only

[Rule-19: Add wrappers for directives, dialogs, and common elements]

Why?
* You can use them in multiple tests
* Avoid code duplication
* When the directive changes you only need to change the wrapper once

```javascript
    /* recommended */
    describe('protractor website', function() {
       var menu = new Menu();
       it('should navigate to API view', function() {
         browser.get('http://www.protractortest.org/#/');

         menu.dropdown('Reference')
             .option('Protractor API')
             .click();
         expect(browser.getCurrentUrl())
             .toBe('http://www.protractortest.org/#/api');
       });
    });
```

Test Suites

[Rule-20: Don't mock unless you need to]

This rule is a bit controversial, in the sense that opinions are very divided when it comes to what the best practice is. Some developers argue that e2e tests should use mocks for everything in order to avoid external network calls and have a second set of integration tests to test the APIs and database. Other developers argue that e2e tests should operate on the entire system and be as close to the 'real deal' as possible.

I tend to agree with Julie Ralph on this one, when she says that

> which option is better depends on how many backends you have, how expensive they are to start up, and the needs of
> your project.

If you can test everything together, then my advice is to do that, because that's the closest to the real app context
you can get. This will also give you much more confidence in your tests and that your app is production ready. If tha
not possible Protractor has some mocking strategies that you can use.

Why?
* Using the real application with all the dependencies gives you high confidence
* Helps you spot some corner cases you might have overlooked

[Rule-21: Use Jasmine2]

Why?
* Well [documented](http://jasmine.github.io/2.0/introduction.html)
* Supported by Protractor out of the box
* You can use **beforeAll** and **afterAll**

[Rule-22: Make your tests independent at file level]

Why?
* You can run tests in parallel with sharding
* The execution order is not guaranteed
* You can run suites in isolation

[Rule-23: Make your tests independent from each other]

This rule holds true unless the operations performed to initialize the state of the tests are too expensive. For exam
if your e2e tests would require that you create a new user before each spec is executed, you might end up with too hi
test run times. However, this does not mean you should make tests directly depend on one another.
So, instead of creating a user in one of your tests and expect that record to be there for all
other subsequent tests, you could harvest the power of jasmine's [beforeAll](http://jasmine.github.io/edge/introducti
to create the user.

```javascript
  /* avoid */
  it('should create user', function() {
     browser.get('#/user-list');
     userList.newButton.click();

     userProperties.name.sendKeys('Teddy B');
     userProperties.saveButton.click();

     browser.get('#/user-list');
     userList.search('Teddy B');
     expect(userList.getNames()).toEqual(['Teddy B']);
  });

  it('should update user', function() {
     browser.get('#/user-list');
     userList.clickOn('Teddy B');

     userProperties.name.clear().sendKeys('Teddy C');
     userProperties.saveButton.click();

     browser.get('#/user-list');
     userList.search('Teddy C');
     expect(userList.getNames()).toEqual(['Teddy C']);
  });
```

```javascript
  /* recommended */
 describe('when the user Teddy B is created', function(){

     beforeAll(function(){
        browser.get('#/user-list');
        userList.newButton.click();

        userProperties.name.sendKeys('Teddy B');
        userProperties.saveButton.click();
        browser.get('#/user-list');
     });

     it('should exist', function(){
```

```
            userList.search('Teddy B');
            expect(userList.getNames()).toEqual(['Teddy B']);
            userList.clear();
        });

        describe('and gets updated to Teddy C', function(){

            beforeAll(function(){
                userList.clickOn('Teddy B');

                userProperties.name.clear().sendKeys('Teddy C');
                userProperties.saveButton.click();

                browser.get('#/user-list');
            });

            it('should be Teddy C', function(){
                userList.search('Teddy C');
                expect(userList.getNames()).toEqual(['Teddy C']);
                userList.clear();
            });
        });
    });
```

Why?

- You can run each single test in isolation
- You can debug your tests (ddescribe/fdescribe/xdescribe/iit/fit/xit)

[Rule-24: Navigate to the page under test before each test]

Why?

- Assures you that the page under test is in a clean state

[Rule-25: Have a suite that navigates through the major routes of the app]

Why?

- Makes sure the major parts of the application are correctly connected
- Users usually don't navigate by manually entering urls
- Gives confidence about permissions

Helper Classes

Useful Links

Karma - http://karma-runner.github.io/

Karma design docs - https://github.com/karma-runner/karma/blob/master/thesis.pdf

Protractor - http://angular.github.io/protractor

GTAC 2010: The Future of Front-End Testing - https://www.youtube.com/watch?v=oX-0Mt5zju0

Selenium - http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-pattern

Google Selenium pages - https://github.com/SeleniumHQ/selenium/wiki/PageObjects

Martin Fowler article on Page Objects - http://martinfowler.com/bliki/PageObject.html