Protractor for End to End test

- Protractor is an end-to-end testing framework for AngularJS applications and works as a solution integrator - combining powerful tools and technologies such as NodeJS, Selenium, webDriver, Jasmine, Cucumber and Mocha.

- It has a bunch of customizations from Selenium to easily create tests for AngularJS applications.

- Protractor also speeds up your testing as it avoids the need for a lot of “sleeps” and “waits” in your tests, as it optimizes sleep and wait times.

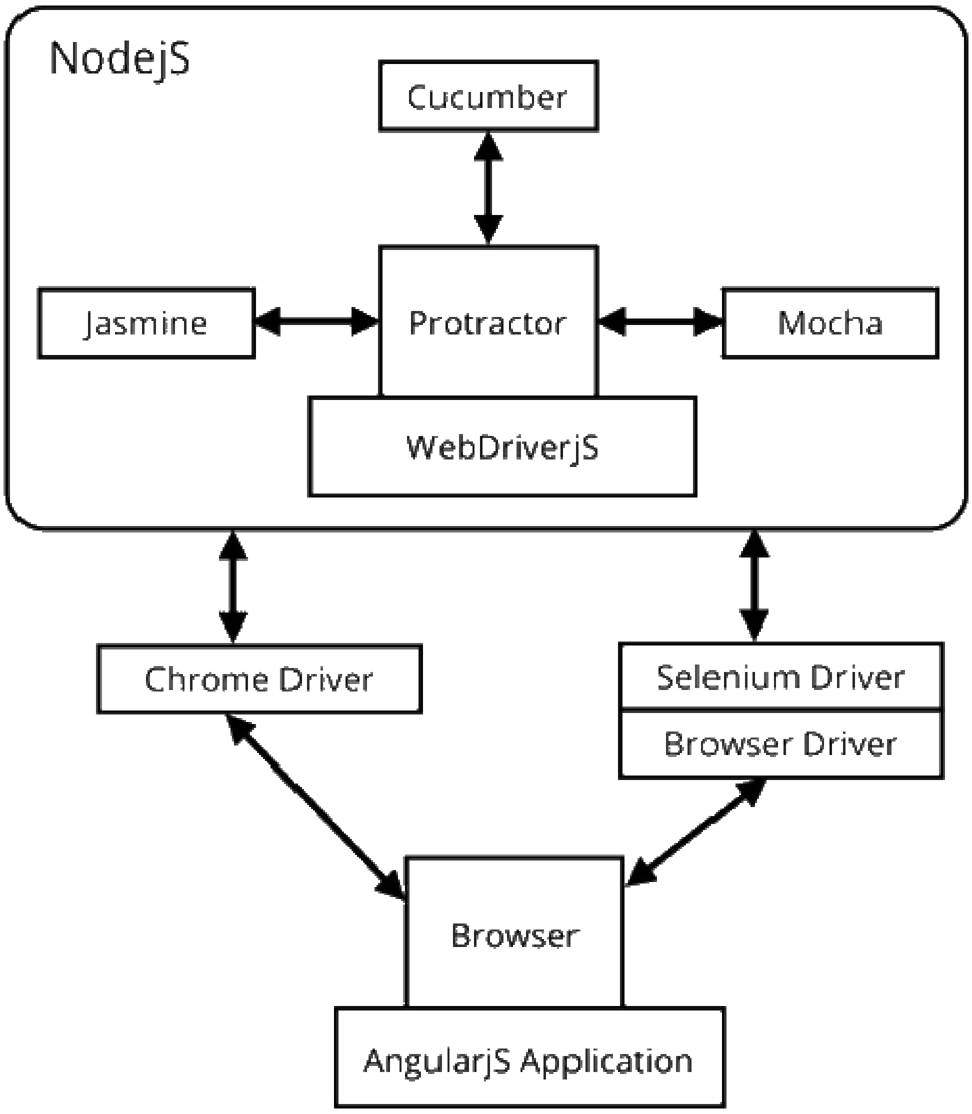
- As it is based on AngularJS concepts, that makes it easy to learn Protractor if you already know about AngularJS and vice versa.

- Protractor allows tests to be organized based on Jasmine, thus allowing you to write both unit and functional tests on Jasmine.
- It runs on real browsers and headless browsers.

What else you need from a framework to automate tests?

Deep-diving into Protractor

The first version of Protractor was released in July, 2013, when the framework was basically a prototype of a testing framework. However, Google, with the support of the testing community, is evolving the framework to follow the evolution of AngularJS and to meet the needs of the community that is using AngularJS.

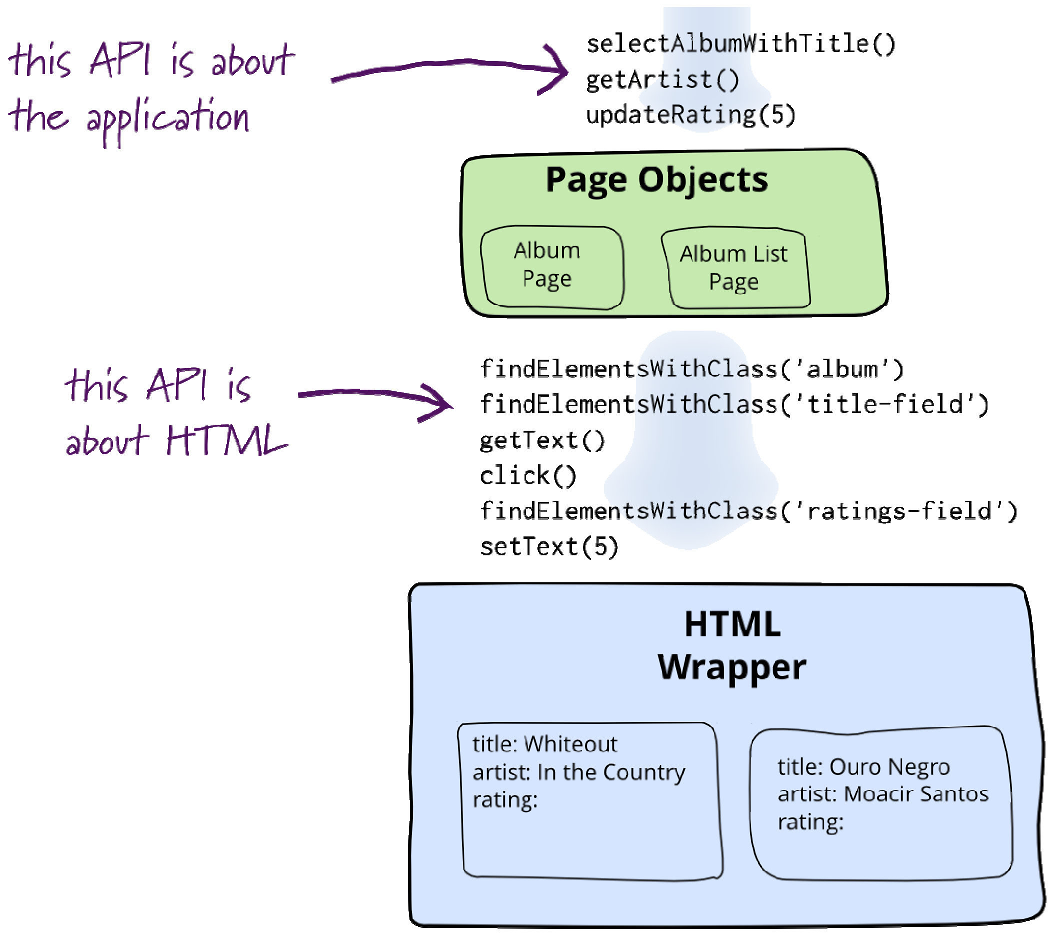


Protractor is a framework for automation of functional tests, so its intention isn't to be the only way to test an AngularJS application, but to cover the acceptance criteria required by the user. Even if there are tests that run in the UI level with Protractor, unit and integration tests are still needed. It runs on top of the Selenium, and thus provides all the benefits and advantages from Selenium. In addition, it provides customizable features to test AngularJS applications. It is also possible to use some drivers which implement WebDriver's wire protocol like ChromeDriver and GhostDriver, as Protractor runs on top of the Selenium. In the case of ChromeDriver it is possible to run tests without the Selenium Server. However to run tests using GhostDriver you need to use PhantomJS which uses GhostDriver to run tests in Headless mode.
The framework integrated with Jasmine can be used to create and organize tests and user expectations. Jasmine is compatible with Protractor due to which all resources that are extracted from browsers can be used to make tests as promises. Those promises

are resolved internally by using the “expect” command from Jasmine. That way the promises work smoothly while creating tests.

Page Object pattern

When you write tests against a web page, you need to refer to elements within that web page in order to click links and determine what's displayed. However, if you write tests that manipulate the HTML elements directly your tests will be brittle to changes in the UI. A page object wraps an HTML page, or fragment, with an application-specific API, allowing you to manipulate page elements without digging around in the HTML.



The basic rule of thumb for a page object is that it should allow a software client to do anything and see anything that a human can. It should also provide an interface that's easy to program to and hides the underlying widgetry in the window. So to access a text field you should have accessor methods that take and return a string, check boxes should use booleans, and buttons should be represented by action oriented method names. The page object should encapsulate the mechanics required to find and manipulate the data in the gui control itself. A good rule of thumb is to imagine changing the concrete control - in which case the page object interface shouldn't change.

Despite the term "page" object, these objects shouldn't usually be built for each page, but rather for the significant elements on a page [1]. So a page showing multiple albums would have an album list page object containing several album page objects. There would probably also be a header page object and a footer page object. That said, some of the hierarchy of a complex UI is only there in order to structure the UI - such composite structures shouldn't be revealed by the page objects. The rule of thumb is to model the structure in the page that makes sense to the user of the application.
Similarly if you navigate to another page, the initial page object should return another page object for the new page [2]. In general page object operations should return fundamental types (strings, dates) or other page objects.
There are differences of opinion on whether page objects should include assertions themselves, or just provide data for test scripts to do the assertions. Advocates of including assertions in page objects say that this helps avoid duplication of assertions in test scripts, makes it easier to provide better error messages, and supports a more TellDontAsk style API. Advocates of assertion-free page objects say that including assertions mixes the responsibilities of providing access to page data with assertion logic, and leads to a bloated page object.
I favor having no assertions in page objects. I think you can avoid duplication by providing assertion libraries for common assertions - which can also make it easier to provide good diagnostics. [3]

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.

I've described this pattern in terms of HTML, but the same pattern applies equally well to any UI technology. I've seen this pattern used effectively to hide the details of a Java swing UI and I've no doubt it's been widely used with just about every other UI framework out there too.

Concurrency issues are another topic that a page object can encapsulate. This may involve hiding the asynchrony in async operations that don't appear to the user as async. It may also involve encapsulating threading issues in UI frameworks where you have to worry about allocating behavior between UI and worker threads.

Page objects are most commonly used in testing, but can also be used to provide a scripting interface on top of an application. Usually it's best to put a scripting interface underneath the UI, that's usually less complicated and faster. However with an application that's put too much behavior into the UI then using page objects may make the best of a bad job. (But look to move that logic if you can, it will be better both for scripting and the long term health of the UI.)

It's common to write tests using some form of DomainSpecificLanguage, such as Cucumber or an internal DSL. If you do this it's best to layer the testing DSL over the page objects so that you have a parser that translates DSL statements into calls on the page object.

Patterns that aim to move logic out of UI elements (such as Presentation Model, Supervising Controller, and Passive View) make it less useful to test through the UI and thus reduce the need for page objects.

Page objects are a classic example of encapsulation - they hide the details of the UI structure and widgetry from other components (the tests). It's a good design principle to look for situations like this as you develop - ask yourself "how can I hide some details from the rest of the software?" As with any encapsulation this yields two benefits. I've already stressed that by confining logic that manipulates the UI to a single place you can modify it there without affecting other components in the system. A consequential benefit is that it makes the client (test) code easier to understand because the logic there is about the intention of the test and not cluttered by UI details.

Further Reading

I first described this pattern under the name Window Driver. However since then the term "page object" was popularized by the Selenium web testing framework and that's become the generally used name.
Selenium's wiki strongly encourages using page objects and provides advice on how they should be used. It also favors assertion-free page objects.
A team measured the times to update two versions of a suite of selenium tests after a software upgrade. They found the version with page object took a bit longer for the first test case, but much faster for the rest.

Notes

1: There's an argument here that the name "page object" is misleading because it makes you think you should have just one page object per page. Something like "panel object" would be better - but the term "page object" is what's become accepted. Another illustration of why naming is one of the TwoHardThings.
2: Having page objects be responsible for creating other page objects in response to things like navigation is common advice. However some practitioners prefer that page objects return some generic browser context, and the tests control which page objects to build on top of that context based on the flow of the test (particularly conditional flows). Their preference is based on the fact that the test script knows what pages are expected next and this knowledge doesn't need to be duplicated in the page objects themselves. They increase their preference when using statically typed languages which usually reveal page navigations in type signatures.
3: One form of assertions is fine even for people like me who generally favor a no-assertion style. These assertions are those that check the invariants of a page or the application at this point, rather than specific things that a test is probing.

Page Objects

Within your web app's UI there are areas that your tests interact with. A Page Object simply models these as objects within the test code. This reduces the amount of duplicated code and means that if the UI changes, the fix need only be applied in one place.

Implementation Notes

PageObjects can be thought of as facing in two directions simultaneously. Facing towards the developer of a test, they represent the **services** offered by a particular page. Facing away from the developer, they should be the only thing that has a deep knowledge of the structure of the HTML of a page (or part of a page) It's simplest to think of the methods on a Page Object as offering the "services" that a page offers rather than exposing the details and mechanics of the page. As an example, think of the inbox of any web-based email system. Amongst the services that it offers are typically the ability to compose a new email, to choose to read a single email, and to list the subject lines of the emails in the inbox. How these are implemented shouldn't matter to the test.

Because we're encouraging the developer of a test to try and think about the services that they're interacting with rather than the implementation, PageObjects should seldom expose the underlying WebDriver instance. To facilitate this, methods on the PageObject should return other PageObjects. This means that we can effectively model the user's journey through our application. It also means that should the way that pages relate to one another change (like when the login page asks the user to change their password the first time they log into a service, when it previously didn't do that) simply changing the appropriate method's signature will cause the tests to fail to compile. Put another way, we can tell which tests would fail without needing to run them when we change the relationship between pages and reflect this in the PageObjects.

One consequence of this approach is that it may be necessary to model (for example) both a successful and unsuccessful login, or a click could have a different result depending on the state of the app. When this happens, it is common to have multiple methods on the PageObject.

Page Objects to Overcome Protractor's Shortcomings

Protractor has Shortcomings?!

Let us go through this example spec from the Protractor GitHub page:

```
describe('angularjs homepage', function() {
  it('should greet the named user', function() {
```

```
  browser.get('http://www.angularjs.org');
  element(by.model('yourName')).sendKeys('Julie');
  var greeting = element(by.binding('yourName'));
  expect(greeting.getText()).toEqual('Hello Julie!');
});

describe('todo list', function() {
  var todoList;
  beforeEach(function() {
  browser.get('http://www.angularjs.org');
  todoList = element.all(by.repeater('todo in todos'));
});

it('should list todos', function() {
  expect(todoList.count()).toEqual(2);
  expect(todoList.get(1).getText()).toEqual('build an angular app');
});

it('should add a todo', function() {
  var addTodo = element(by.model('todoText'));
  var addButton = element(by.css('[value="add"]'));

  addTodo.sendKeys('write a protractor test');
  addButton.click();

  expect(todoList.count()).toEqual(3);
  expect(todoList.get(2).getText()).toEqual('write a protractor test');
});
```

Although this spec is syntactically perfect and does the right thing, it can cause a few problems:

1. Lack of Domain Specific Language (DSL) Support[1]: It is hard to understand what is being tested, because the Protractor specific vocabulary (element, by.binding, by.model, etc.) is not related to the business and features. Having test specifications that speak the jargon of the domain goes a long way in helping business understand the rationale behind the tests
2. Code Duplication[2]: Following are the steps to add a new Todo function:

1. Get the addTodo input by model
2. Get the addButton button by css
3. Write something on the input using sendKeys
4. Click on the button to confirm.

Although it is just four lines long, it is likely that we will need to add todos in many of our tests, so these four lines will be copied and pasted along many files, leading to code duplication. For example, when you are testing the amount of todos that are pending, you will need to simulate adding todos before check the count.

3. High Coupling[3]: Just as an exercise, ask yourself, what would happen if the input field where you type a new todo has to be changed? What if it was required to click a button before being able to enter a new todo? Such changes (which may seem unlikely at the start) end up happening in every project. It is essential to not be highly coupled, but instead be flexible enough to be ready for change.
4. Tough Maintenance: All these issues above will make your awesome Protractor tests a huge pain to maintain in a medium/long term. As the project grows and change creeps in, nobody will be keen on changing the numerous 'Adding todo logic' spread across your code.
As you can see, even a very small application example like the "Todo" one above is likely to suffer from issues. These problems get magnified as the complexity of your application increases.

How Page Objects can help

Firstly, what are Page Objects? Here is a definition from the Selenium team:

Page Object is a Design Pattern which has become popular in test automation for enhancing test maintenance and reducing code duplication. A page object is an object-oriented class that serves as an interface to a page of your AUT. The tests then use the methods of this page object class whenever they need to interact with that page of the UI. The benefit is that if the UI changes for the page, the tests themselves don't need to change, only the code within the page object needs to change. Subsequently all changes to support that new UI are located in one place.

How can PageObjects help? Let's take a look at how we can refactor the same example above using Page Objects:

Refactored Spec -

```
'use strict';

  var AngularPage = require('../pages/angular.page.js');

  describe('angularjs homepage', function () {
  var page;

beforeEach(function () {
  page = new AngularPage();
});

it('should greet the named user', function () {
  page.typeName('Julie');
  expect(page.greeting).toEqual('Hello Julie!');
});

describe('todo list', function () {
  it('should list todos', function () {
  expect(page.todoList.count()).toEqual(2);
  expect(page.todoAt(1)).toEqual('build an angular app');
});

it('should add a todo', function () {
  page.addTodo('write a protractor test');
  expect(page.todoList.count()).toEqual(3);
  expect(page.todoAt(2)).toEqual('write a protractor test');
});
});
```

Notice that the much of the code is contained inside the AngularPage. The idea is to move all the logic required to interact with the page inside this class.

Our suite now is more focused on the behaviour of the page, than on how select this or that element.

Now let's take a look at how the page object looks like.

The Page Object -

```
'use strict';

var AngularPage = function () {
  browser.get('http://www.angularjs.org');
};

 AngularPage.prototype = Object.create({}, {
    todoText: { get: function () { return element(by.model('todoText')); }},
    addButton: { get: function () { return element(by.css('[value="add"]')); }},
    yourName: { get: function () { return element(by.model('yourName')); }},
    greeting: { get: function () { return element(by.binding('yourName')).getText(); }},
    todoList: { get: function () { return element.all(by.repeater('todo in todos')); }},
    typeName: { value: function (keys) { return this.yourName.sendKeys(keys); }},
    todoAt: { value: function (idx) { return this.todoList.get(idx).getText(); }},
    addTodo: { value: function (todo) {
```

```
    this.todoText.sendKeys(todo);
    this.addButton.click();
  }}
});

module.exports = AngularPage;
```

We’ll now revisit the “shortcomings” I had listed earlier and see how the refactored Page Objects version helped overcome them:

1. **Lack of Domain Language Support:** Notice that all the Protractor specific words are gone in the refactored version and you have more business-like expressions instead. That is much easier to read, especially by business folks, thus helping them better understand the test specifications.
2. **Code Duplication:** The lines of code that were likely to be copied and pasted are now inside the Page Object and we just call this Page Object if we need to execute them.
3. **High Coupling:** Since the tests only talk to the Page Object, it is totally decoupled from the elements inside each page. This means that if we need to change something on the UI the tests will stay pretty much the same. With Page Object we have the right abstraction of the page elements and their actions.
4. **Hard Maintenance:** As a consequence of the points above, it is much easier now to maintain these tests in the medium/long term.

Well done! You just learned how to write better Protractor tests.

Protractor and Page Objects

In a previous post, we looked at using the Protractor framework with AngularJS apps. While Protractor is a great testing tool out of the box, it can benefit from some best practices to make a testing suite and its code more manageable. One practice worth considering for organizing test code is the use of a Page Object design pattern.

A page object is a class that simply stores your page elements. Elements and methods are housed in the page object file. This pattern allows testers to write clean code, avoid duplication, and better maintain their suites.

Let’s take a look at our previous Protractor example:

```
describe('Protractor Test', function() {
  var addField = element(by.css('[placeholder="add new todo here"]'));
  var checkedBox = element(by.model('todo.done'));
  var addButton = element(by.css('[value="add"]'));

  it('should navigate to the AngularJS homepage', function() {
    browser.get('https://angularjs.org/'); //overrides baseURL
  });

  it('should show a search field', function() {
    browser.sleep(5000); //used just to give you enough time to scroll to todo section
    addField.isDisplayed();
  });

  it('should let you add a new task ', function() {
    addField.sendKeys('New Task');
    addButton.click();
    browser.sleep(5000); //used just to see the task list update
  });
});
```

As you can see, each element is defined within the spec. This pattern can be ok for a small suite, but it isn't able to support larger, more complex automation. With this pattern, each spec has its' own set of elements to maintain. If multiple specs use the same elements then code duplication will quickly become a problem. And, if the UI of your application changes (that rarely happens, right?), you will need to update your elements within each, individual test.

All of these complications can be fixed with the use of page objects. Here is what a page object file for the spec could look like:

```

'use strict';

module.exports = {
    toDo: {
        addField: element(by.css('[placeholder="add new todo here"]')),
        checkedBox: element(by.model('todo.done')),
        addButton: element(by.css('[value="add"]'))
    },

    go: function() {
        browser.get('https://angularjs.org/'); //overrides baseURL
        browser.waitForAngular();
    },

    addItem: function(item) {
        var todo = this.toDo;

        todo.addField.isDisplayed();
        todo.addField.sendKeys(item);
        todo.addButton.click();
    }
};
```

The majority of our logic needed to run the tests is now housed in a file called toDoPage.js. As your application grows, you may have multiple page object files that correspond to individual views. For example, a login view and home view may have their own, dedicated page objects. Organization of your tests and files may differ based on preference, but it’s important to decide on that organization structure early on.

Now that the page object has been defined, here is the cleaned up test spec:

```
var toDoPage = require('../pages/toDoPage.js');

describe('Protractor Test', function() {

  it('should navigate to the AngularJS homepage', function() {
    toDoPage.go();
  });

  it('should let you add a new task ', function() {
   toDoPage.addItem('New Task Item')
  });
});
```

This cleans up the spec significantly and enhances maintainability. With the Page Object design pattern, test workflows are in the specs while logic and UI elements are in the page objects. That is a good way to look at the separation between these two components.

Using Page Objects to Organize Tests

When writing end-to-end tests, a common pattern is to use Page Objects. Page Objects help you write cleaner tests by encapsulating information about the elements on your application page. A Page Object can be reused across multiple tests, and if the template of your application changes, you only need to update the Page Object.

Without Page Objects

Here's a simple test script (example_spec.js) for 'The Basics' example on the angularjs.org homepage.

```
describe('angularjs homepage', function() {
  it('should greet the named user', function() {
    browser.get('http://www.angularjs.org');
    element(by.model('yourName')).sendKeys('Julie');
    var greeting = element(by.binding('yourName'));
    expect(greeting.getText()).toEqual('Hello Julie!');
  });
});
```

With PageObjects

To switch to Page Objects, the first thing you need to do is create a Page Object. A Page Object for 'The Basics' example on the angularjs.org homepage could look like this:

```
var AngularHomepage = function() {
  var nameInput = element(by.model('yourName'));
  var greeting = element(by.binding('yourName'));

  this.get = function() {
    browser.get('http://www.angularjs.org');
  };

  this.setName = function(name) {
    nameInput.sendKeys(name);
  };

  this.getGreeting = function() {
    return greeting.getText();
  };
};
```

The next thing you need to do is modify the test script to use the PageObject and its properties. Note that the *functionality* of the test script itself does not change (nothing is added or deleted).

```
describe('angularjs homepage', function() {
  it('should greet the named user', function() {
    var angularHomepage = new AngularHomepage();
    angularHomepage.get();

    angularHomepage.setName('Julie');

    expect(angularHomepage.getGreeting()).toEqual('Hello Julie!');
  });
});
```

Configuring Test Suites

It is possible to separate your tests into various test suites. In your config file, you could setup the suites option as shown below.

```
exports.config = {
  // The address of a running selenium server.
  seleniumAddress: 'http://localhost:4444/wd/hub',

  // Capabilities to be passed to the webdriver instance.
  capabilities: {
    'browserName': 'chrome'
  },

  // Spec patterns are relative to the location of the spec file. They may
  // include glob patterns.
  suites: {
    homepage: 'tests/e2e/homepage/**/*Spec.js',
    search: ['tests/e2e/contact_search/**/*Spec.js',
      'tests/e2e/venue_search/**/*Spec.js']
  },

  // Options to be passed to Jasmine-node.
  jasmineNodeOpts: {
    showColors: true, // Use colors in the command line report.
  }
};
```

From the command line, you can then easily switch between running one or the other suite of tests. This command will run only the homepage section of the tests:

```
protractor protractor.conf.js --suite homepage
```

Additionally, you can run specific suites of tests with the command:

```
protractor protractor.conf.js --suite homepage,search
```

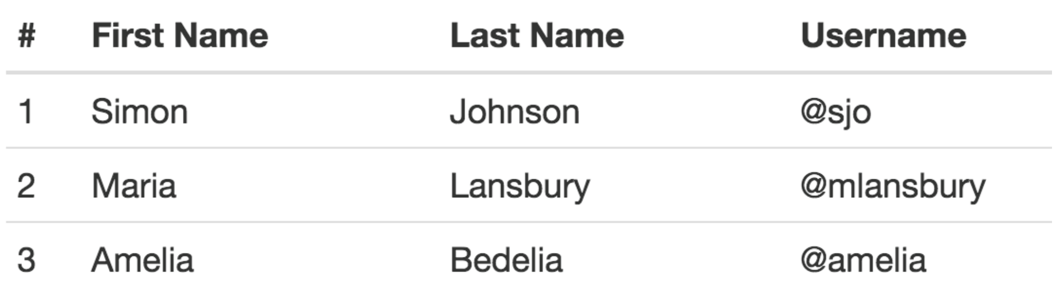
Protractor Page Objects

Protractor Page objects provide a helpful way to organize your end-to-end test code. While your actual tests take actions and make assertions about how your application should respond, page objects encapsulate the details of how to perform those actions on a page.

```
var LoginPage = function() {
  this.userInput = element(by.model('username'));
  this.passwordInput = element(by.model('password'));
  this.loginButton = element(by.css('.app-login'));

  this.get = function() {
    browser.get('#/login');
  };

  this.login = function(username, password) {
    this.userInput.sendKeys(username);
    this.passwordInput.sendKeys(password);
    this.loginButton.click();
  };
};

describe('login', function() {
  it('welcomes the user', function() {
    var loginPage = new LoginPage();
    loginPage.get();
    loginPage.login('jsmith', 'Pa$$word77')

    // The rest of the test
  });
});
```

The loginPage object provides an interface for tests to interact with the application’s login page. This makes it easy for different tests to interact with the page without duplicating code.

Partial Page Objects



#	First Name	Last Name	Username
1	Simon	Johnson	@sjo
2	Maria	Lansbury	@mlansbury
3	Amelia	Bedelia	@amelia

Tables are good candidates for partial page objects

As our applications grow complex, our pages tend to contain duplicated structure. For instance, the above table of users has the same first name, last name, and username structure on each of its rows. To exploit this duplicated structure, we'd like to write test code that looks like this:

expect(infoPage.getUser(2).lastName.getText()).toEqual('Bedelia');

The new part is the getUser method—instead of a protractor element, this method returns another page object. To achieve this, we need to pass a parent element into the UserPartial constructor that will "wrap" all of the partial page's elements:

```
function UserPartial(parent) {
  this.parent = parent;
  this.firstName = this.parent.element(By.css('.app-first-name'));
  this.lastName = this.parent.element(By.css('.app-last-name'));
  this.username = this.parent.element(By.css('.app-username'));
};

function InfoPage() {
  this.users = element.all(By.css('.app-user'));
};

InfoPage.prototype.getUser = function(i) {
  return new UserPartial(this.users.get(i));
};
```

Complications

This pattern would still work if we defined thirdLastName = infoPage.getUser(2).lastName before the page was even shown at all. Later on when

we checked thirdLastName.getText(), it would find the last name. This is because the promise that element() returns isn’t resolved until getText is called.
It’s easy to accidentally write code that resolves these promises too early. Consider this change to the UserPartial constructor:

```
function UserPartial(parent) {
  this.parent = parent;
  // . . .
  this.userId = this.parent.getAttribute(‘data-id’);
};
```

Getter syntax is not very browser-compatible. And normally that wouldn’t be much of an issue since this is test-only code, but it also means it’s unsupported by CoffeeScript. If a CoffeeScript solution is what you’re looking for, I recommend using Object.defineProperties as a workaround:

```
class UserPartial
  constructor: (@parent) ->

  Object.defineProperties @prototype,
    userId:
      get: -> @parent.getAttribute(‘data-id')
```

Protractor: Using the Page Object Model

What is Protractor?

Protractor
end to end testing for AngularJS

Protractor is an end-to-end (e2e) test automation framework for AngularJS application. It is an open source Node.js program built on top of WebDriverJS originally developed by a team at Google. Test cases written in Protractor run in the browser simulating the actions of a real user. An e2e test written in Protractor makes sure your application behaves as expected.

Challenge: Code Duplication

There is always duplication in test cases. For instance login, find, and logout are clearly duplicated in the following two test cases:

Test case 1: login to the website, find an item, add it to my wish list and logout.

Test case 2: login to the website, find an item, add it to cart, purchase and logout.

Duplicate test cases result in code duplication. An e2e test suite with code duplication is difficult to maintain and requires costly modifications. In this tutorial, we will implement a page object design best practice for Protractor to minimize code duplication, make tests more readable, reduce the cost of modification, and improve maintainability.

The most important concept here is to separate the abstraction of the test object (the page) and the test script (the spec). Hence, a single test object can be used multiple times by test scripts without rewriting it.

Using the PhoneCat application

We will use the popular AngularJS PhoneCat application to demonstrate how Protractor tests could make use of the page object design pattern to create simple and maintainable e2e test automation.

A concise instruction set, on how to setup the PhoneCat application in your local machine, is at the end of this post.

Abstraction: Separation of Test Object from Test Script

The PhoneCat app has the '*phones list view*' page where all available phones are listed. A user can search or change the order of the listed phones on the page. When selecting a phone from the list, a user navigates to the '*phone details view*' page, where more details about the selected phone are included.

In line with the page object design pattern best practice: the PhoneCat application has two test objects, the*phones list view* page and the *phone details view* page. Each of the pages should be self-contained, meaning they should provide all the locators and functions required to interact with each page. For example, the*phones list view* page should have a locator for the search input box and a function to search.

The image below shows the separation of the test object (page object files) from the test script (spec files). The spec files under the spec folder contain only test scripts. The page object files under the page object folder contain page specific locators and functions.

Figure 1: Separation of page object from test specification

Test Object (Page Object)

The PhoneCat application have the *phones list page* and the *phone details page*. The following two page object files provide the locators and functions required to interact with these pages.

```
Phones = {

    elements: {

        _search: function () {

            return element(by.model('query'));

        },



        _sort: function(){

            return element(by.model('orderProp'));

        },



        _phoneList: function(){

            return element.all(by.repeater('phone in phones'));

        }

        _phoneNameColumn: function(){

            returnelement.all(by.repeater('phone in
phones').column('phone.name'));

        }

    },

    _phonesCount: function(){

        return this.elements._phoneList().count();

    },
    searchFor: function(word){

        this.elements._search().sendKeys(word);

    },

    clearSearch: function(){

        this.elements._search().clear();

    },

    _getNames: function(){

        return this.elements._phoneNameColumn().map(function(elem){

            return elem.getText();

        });

    },

    sortItBy: function(type){

        this.elements._sort().element(by.css('option[value="' + type +
'"]')).click();

    },

    selectFirstPhone : function(){

        element.all(by.css('.phones li a')).first().click();

        return require('./phone.details.page.js');

    }

};

module.exports = Phones;
```

Listing 1: phones.page.js

```
PhoneDetails = {

    elements:{
        _name: function(){
            return element(by.binding('phone.name'));
        },
        _image: function(){
            return element(by.css('img.phone.active'));
        },
        _thumbnail: function(index){
            return element(by.css('.phone-thumbs li:nth-child(' + index +') img'));
        }
    },
    _getName: function(){
        return this.elements._name().getText();
    },
    _getImage: function(){
        return this.elements._image().getAttribute('src');
    },
    clickThumbnail: function(index){
        this.elements._thumbnail(index).click();
    }
};
module.exports = PhoneDetails;
```

Listing 2: phone.details.page

Test Script (Spec)

The test script can now make use of the page object files. All the functions required to interact with the page (the test object) are encapsulated in the page object and the test scripts are more readable and concise.

```
describe('Phone list view', function(){

   var phones = require('../page_objects/phones.page.js');

   beforeEach(function() {

      browser.get('app/index.html#/phones');

   })

   it('should filter the phone list as a user types into the search box',
function() {

      expect(phones._phonesCount()).toBe(20);

      phones.searchFor('nexus');

      expect(phones._phonesCount()).toBe(1);

      phones.clearSearch();

      phones.searchFor('motorola');

      expect(phones._phonesCount()).toBe(8);

   });

   it('should be possible to control phone order via the drop down select box',
function() {

      phones.clearSearch();

      phones.searchFor('tablet'); //let's narrow the dataset to make the test
assertions shorter

      expect(phones._getNames()).toEqual([

         "Motorola XOOM\u2122 with Wi-Fi",

         "MOTOROLA XOOM\u2122"

      ]);
```

```
        phones.sortItBy('name');
        expect(phones._getNames()).toEqual([
            "MOTOROLA XOOM\u2122",
            "Motorola XOOM\u2122 with Wi-Fi"
        ]);
    });
    it('should render phone specific links', function() {
        phones.clearSearch();
        phones.searchFor('nexus');
        phones.selectFirstPhone();
        browser.getLocationAbsUrl().then(function(url) {
            expect(url.split('#')[1]).toBe('/phones/nexus-s');
        });
    });
});
```

Listing 3: phones.spec.js

```
describe('Phone detail view', function(){
    var phones = require('../page_objects/phones.page.js'),
        phoneDetails;
    beforeEach(function() {
        browser.get('app/index.html#/phnes');
        phones.searchFor('nexus');
        phoneDetails = phones.selectFirstPhone();
    });

    it('should display nexus-s page', function() {

        expect(phoneDetails._getName()).toBe('Nexus S');

    });

    it('should display the first phone image as the main phone image',
function() {

        expect(phoneDetails._getImage()).toMatch(/img\/phones\/nexus-
s.0.jpg/);

    });

    it('should swap main image if a thumbnail image is clicked on', function() {

        phoneDetails.clickThumbnail(3);

        expect(phoneDetails._getImage()).toMatch(/img\/phones\/nexus-
s.2.jpg/);

        phoneDetails.clickThumbnail(1);

        expect(phoneDetails._getImage()).toMatch(/img\/phones\/nexus-
s.0.jpg/);

    });

});
```

Listing 4: phone.details.spec.js

In conclusion, when a page object design pattern is properly used in Protractor test automation, it will make an e2e test easy to maintain and reduce code duplication.

Appendix

GitHub Repo: For This Tutorial

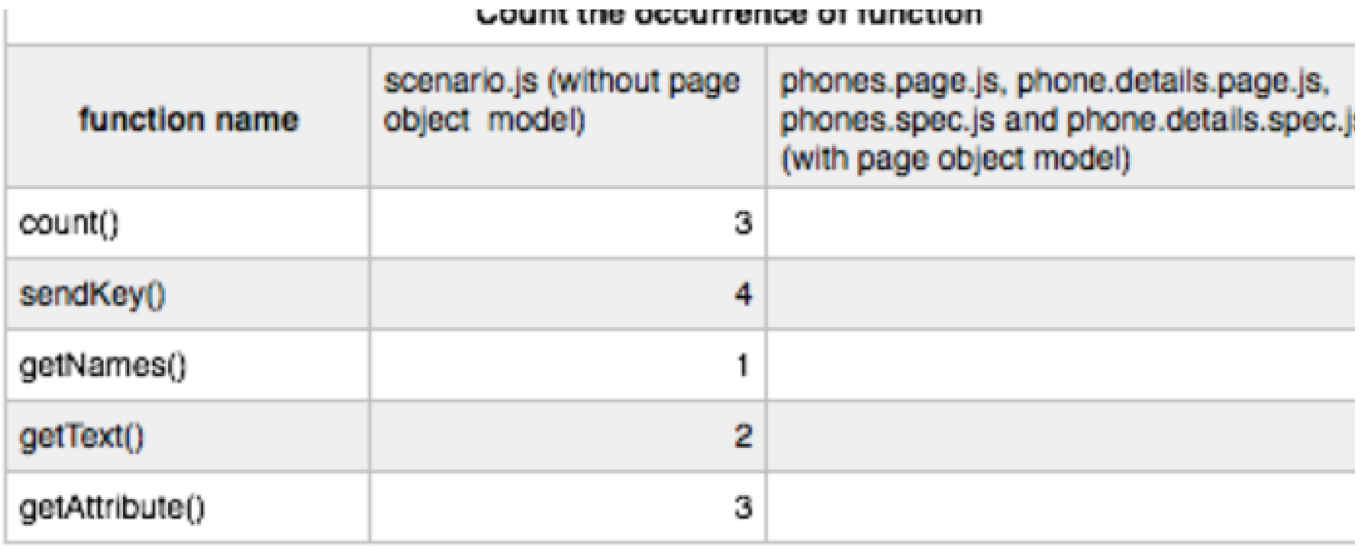
The following gitHub link for the PhoneCat tutorial and adopt it to this tutorial. It is basically a sample Protractor test (scenarios.js) of the PhoneCat app rewritten in a page object model.

https://github.com/xgirma/angular-phonecat.git

This could be a good starting point for discussion on the application of the page object model to improve the maintainability of Protractor tests.

Comparison

The following table shows the main use of the page object model, which is minimizing code duplication. The table compares a Protractor test included in the PhoneCat (scenario.js) and a Protractor test (phones.page.js, phone.details.page.js, phones.spec.js and phone.details.spec.js) which implements the same test case with the page object model. As shown in the table, even in this simple test, code duplication is enormous when implemented without the page object model. In contrast, code duplication when implemented with the page object model is very minimal.



Count the occurrence of function

function name	scenario.js (without page object model)	phones.page.js, phone.details.page.js, phones.spec.js and phone.details.spec.j (with page object model)
count()	3	
sendKey()	4	
getNames()	1	
getText()	2	
getAttribute()	3	

Table 1: Comparison of code duplication with and with out page object model

PhoneCat app: the Setup

1. Install Git and Node.js.

2. Clone the angular-phonecat repository. ($ git clone --depth=14 https://github.com/angular/angular-phonecat.git)

3. Change your current directory to *angular-phonecat ($ cd angular-phonecat).* Download the tool dependencies by running (*$ npm install*).

4. Use npm helper scripts to start a local development web-server(*$ npm start*). This will create a local webserver in your machine, listening to port 8000. Browse the application at http://localhost:8000/app/index.html

5. To install the drivers needed by Protractor (*$ npm run update-webdriver*) and to run the Protractor end to end tests (*$ npm run protractor*).

Refer to the AngularJS site for complete instructions.

https://docs.angularjs.org/tutorial/step_00

Final note: If you want to try the code samples given in this tutorial, besides creating folders, the page object files, and the spec files, you need to change the path to the the new spec files in *protractor-cof.js* file. Simply change **spec: [‘e2e/*.js’]** to **spec:[‘e2e/spec/*.spec.js’]** or to a path where you put the spec files.

Protractor and Page Objects for AngularJS E2E Testing

When it was announced that Protractor would be replacing the karma runner as *THE* end to end test framework for AngularJS, I’ll admit, I was a bit sad. While I didn’t have a deep love for the karma runner, it was crazy simple to setup and run, and my initial perusal of the Protractor docs was intimidating. After spending some time with Protractor though, I have come to enjoy using it in the same manner in which I would test a Rails application.

Going through the Protractor docs and getting started can still be a daunting task. My goal is to alleviate some of that by sharing my configuration and setup, as well as using the page object pattern for creating an API through which we’ll interact with our web pages. We’ll also be using the jasmine-given library to help clean up our specs.

The Setup

The first thing we will need to do is make sure we have Protractor and jasmine-given available to our application.

```
$ npm install protractor jasmine-given --save-dev
```

Please note the **--save-dev**. This is so that node will add these as dev dependencies to **package.json**. It is also important to note that Protractor can be installed globally with the **-g** switch. Personally, I do not like to install application dependencies globally, and try to avoid them if at all possible.

Protractor has two dependencie of it’s own, selenium server and a browser driver. There exists pages of documentation for running the selenium server stand alone and managing browser drivers, but Protractor gives us a script to manage both.

In your project directory, run the following command:

```
$ node_modules/protractor/bin/webdriver-manager update
```

This will install the selenium server jar file and the chrome driver executable locally to your project. Now to start the selenium server, simply run the following command in a separate terminal window.

```
$ node_modules/protractor/bin/webdriver-manager start
```

The last piece of our setup puzzle is the config file that we’ll pass to Protractor when running our specs.

In spec-e2e.conf.js

```
exports.config = {

 seleniumAddress: "http://127.0.0.1:4444/wd/hub",

 seleniumPort: null,

 seleniumArgs: [],

 specs: [

   '../spec-e2e/**/*spec.{js,coffee}'

 ],

 capabilities: {

   'browserName': 'chrome'

 },

 baseUrl: 'http://localhost:8000',

 jasmineNodeOpts: {

   onComplete: null,
   isVerbose: false,

   showColors: true,

   includeStackTrace: false

 }

};
```

With that, we tell protractor where to find our selenium server, which browser(s) to test against, and any jasmine options we choose to pass along. You may reference the example config in the Protractor repo for additional config options.

Our first test

For a simple example of using Protractor to test drive an application, we are going to use logging into a web application. In **spec/e2e/login_spec.js** we setup our test using jasmine-given and page objects.

```
require("protractor/jasminewd");

require('jasmine-given');



var LoginPage = require("./pages/login_page");



describe("app", function() {

  var page = new LoginPage();

  describe("visiting the login page", function() {

    Given(function() {

      page.visitPage();

    });

    describe("when a user logs in", function() {

      Given(function() {
        page.fillEmail("testy@example.com");

      });

      Given(function() {

        page.fillPassword();

      });

      When(function() {

        page.login();

      });

      Then(function() {

        page.getCurrentUser().then(function(text) {

       expect(text).toEqual("Randy Savage");

        });

      });

    });

  });

});
```

First we require our protractor jasmine web driver, and also jasmine-given. Next we require our not yet existing login page object. Last, we walk through the steps of navigating to the login page, filling in and submitting the form. Last, we assert that our page knows about our **currentUser** and that it's text is equal to Randy Savage.

We can run this spec with the following command:

```
$ node_modules/protractor/bin/protractor path/to/spec-e2e.conf.js
```

The output from this run should throw an error that node cannot find the module **pages/login_page**, so let's go ahead and add that.

```
var LoginPage = (function () {

    function LoginPage() {

        this.emailField = element(By.input("user.email"));

        this.passwordField = element(By.input("user.password"));

        this.loginButton = element(By.id("log-in"));

        this.currentUser = element(By.binding("{{currentUser.name}}"));

    }


    LoginPage.prototype.visitPage = function () {

        browser.get("/");

    };


    LoginPage.prototype.fillEmail = function (email) {

        this.emailField.sendKeys(email);

    };


    LoginPage.prototype.fillPassword = function (password) {

        if (password == null) {

            password = "password";

        }

        this.passwordField.sendKeys(password);

    };
```

```
  LoginPage.prototype.login = function () {

    this.loginButton.click();

  };



  LoginPage.prototype.getCurrentUser = function () {

    return this.currentUser.getText();

  };



  return LoginPage;



})();



module.exports = LoginPage;
```

In the constructor function of our **LoginPage** class, we setup instances of the things on the page we wish to interact with using Protractor's locators. We use **By.input** to find an input with a binding to **ng-model="user.email"** and again for **ng-model="user.password"**. We find our submit button by id, and lastly we find the current users name by searching the page for the **currentUser.name** binding. We then setup the functions on our page object that we are calling from our spec, using the elements we setup in our constructor. It is important to note the implicit return of **this.currentUser.getText()**. This is because the **getText()** method returns a promise, which we return to and resolve in our spec. Finally we ensure that our page object is available to the spec by exporting it.

Now, when we run our spec again, we should get an actionable failure **Error: No element found using locator: by.model("user.email")** allowing us to implement our markup and logic to get our spec passing. With just a few simple steps, we were able to get a selenium server running and Protractor configured to use it. As well as creating a pleasant testing DSL by abstracting our page specific login into a reuseable page object.

Astrolabe

Astrolabe is an extension for protractor that adds page objects to your functional/e2e tests.

Installation

via npm (node package manager)

```
$ npm install astrolabe
```

Usage

Example signInPage.js

```
var Page = require('astrolabe').Page;

module.exports = Page.create({

    url: { value: 'http://mysite.com/signin' },

    username: { get: function() { return this.findElement(this.by.input('username')); } }, // finds an input element with the name 'username'
    submit:   { get: function() { return this.findElement(this.by.id('submit')); } }       // finds an element with the id 'submit'
});
```

adding to tests:

```
var signInPage = require('./path/to/signInPage');

...
```

navigating:

```
signInPage.go(); // will send browser to 'http://mysite.com/signin'

signInPage.go('some', 'path'); // will send browser to 'http://mysite.com/signin/some/path'
signInPage.go('some/path');    // will send browser to 'http://mysite.com/signin/some/path'

signInPage.go({ some: 'query' }); // will send browser to 'http://mysite.com/signin?some=query'
```

interacting: (See Protractor API Docs for more info on available api methods)

```
signInPage.username.sendKeys('a username'); // will fill the username input with the text 'a username'

signInPage.submit.click(); // will click on the submit element
signInPage.username.getAttribute('value'); // will return a promise that is resolved with
the value of the text field, in this case 'a username'

// this can be used within an expectation
expect(signInPage.username.getAttribute('value')).toBe('a username');
```

It is possible to create convenience methods to wrap up common logic.

Example signInPage.js

```
var Page = require('astrolabe').Page;

module.exports = Page.create({

    url: { value: 'http://mysite.com/signin' },

    username: { get: function() { return this.findElement(this.by.input('username')); } },
    password: { get: function() { return this.findElement(this.by.input('password')); } },
    submit:   { get: function() { return this.findElement(this.by.id('submit')); } },
    invalid:  { get: function() { return this.findElement(this.by.id('incorrectLogin')); } },

    InvalidLoginException: { get: function() { return this.exception('Invalid Login'); } },

    // Adds a signIn method to the page object.
    signIn:   { value: function(username, password) {

        var page = this;

        page.go();

        page.username.sendKeys(username);
        page.password.sendKeys(password);

        page.submit.click();

        return this.invalid.isDisplayed().then(function (wrongLogin) {
            if (wrongLogin) {
                page.InvalidLoginException.thro(username + ', ' + password + ' is not valid');
            }
        });
    } }
});
```

can be used in your tests:

```
var signInPage = require('./path/to/signInPage');

...
```

```
signInPage.signIn('test user', 'testpassword'); // will navigate to sign in page, enter username and password then click submit.

...
```

Cloning and running Astrolabe's tests

Clone the github repository.

```
git clone https://github.com/stuplum/astrolabe.git
cd astrolabe
npm install

npm test
```

Running Astrolabe's example protractor test

Install protractor with.

```
npm install protractor
```

Start up a selenium server (See the appendix below for help with this). By default, the tests expect the selenium server to be running at http://localhost:4444/wd/hub.
The example folder contains a simple test suite which runs against angularjs.org. It is a port of the simple test suite included with protractor.

Currently only the protractor runner is supported. The runner accepts a configuration file, which runs the tests at example/onProtractorRunner.js.

```
node_modules/.bin/protractor examples/protractor.conf.js
```

****