**Problem Statement**

Create an application to enter comments. The specification of the application is as follows:

1. Given I am posting a new comment, when I click on the submit button, the comment should be added to the to-do list
2. Given a comment, when I click on the like button, the number of likes for the comment should be increased

Here is the UI spec:



**Setting up the directory**

The following instructions are specific to setting up the project directory:

1. Create a new project directory.

**npm init**

This will create package.json

2. Get angular into the project using Bower:

**bower install angular**

3. Get angular-mocks for testing using Bower:

**bower install angular-mocks**

4. Initialize the application’s source directory:

**md app**

5. Initialize the test directory:

**md spec**

6. Initialize the unit test directory:

**md spec\unit**

7. Initialize the end-to-end test directory:

**md spec/e2e**

Once the initialization is complete, your folder structure should look as follows:



**Setting up Protractor**

1. Install Protractor in the project:

**$ npm install protractor**

2. Update Selenium WebDriver:

**$ ./node\_modules/protractor/bin/webdriver-manager update**

Make sure that Selenium has been installed.

3. Copy the example chromeOnly configuration into the root of the project:

**$ cp ./node\_modules/protractor/example/chromeOnlyConf.js .**

4. Configure the Protractor configuration using the following steps:

1. Open the Protractor configuration.

2. Edit the Selenium WebDriver location to reflect the relative directory to

chromeDriver:

chromeDriver: './node\_modules/protractor/selenium/chromedriver',

3. Edit the files section to reflect the test directory:

specs: ['spec/e2e/\*\*/\*.js'],

5. Set the default base URL:

baseUrl: 'http://localhost:8080/',

Here is the complete configuration:

exports.config = {

chromeOnly: true,

chromeDriver: './node\_modules/protractor/selenium/chromedriver',

capabilities: {

'browserName': 'chrome'

},

baseUrl: 'http://localhost:8080/',

specs: ['spec/e2e/\*\*/\*.js'],

};

**Setting up Karma**

1. Install Karma using the following command:

**npm install karma -g**

2. Initialize the Karma configuration:

**karma init**

3. Update the Karma configuration:

files: [

'bower\_components/angular/angular.js',

'bower\_components/angular-mocks/angular-mocks.js',

'spec/unit/\*\*/\*.js'

],

Here is the complete karma.conf.js file:

module.exports = function(config) {

config.set({

basePath: '',

frameworks: ['jasmine'],

files: [

'bower\_components/angular/angular.js',

'bower\_components/angular-mocks/angular-mocks.js',

9993'spec/unit/\*\*/\*.js'

],

reporters: ['progress'],

port: 9876,

autoWatch: true,

browsers: ['Chrome'],

singleRun: false

});

};

**Setting up http-server**

1. To install http-server in your project, type the following command:

**$ npm install http-server**

1. Once http-server is installed, you can run the server by providing it with the root directory of the web page. Here is an example:

**$ ./node\_modules/http-server/bin/http-server**

**Top-down or bottom-up approach**

1. **The bottom-up approach**: With this approach, we think about the different components we will need (controller, service, module, and so on) and then pick the most logical one and start coding.
2. **The top-down approach**: With this approach, we work from the user scenario and UI. We then create the application around the components in the application.

**Testing a controller**

**A simple controller test setup**

When testing a controller, tests are centered on the controller’s scope. The tests confirm either the objects or methods in the scope. Angular mocks provide inject, which finds a particular reference and returns it for you to use. When inject is used for the controller, the controllers scope can be assigned to an outer reference for the entire test to use. Here is an example of what this would look like:

describe('',function(){

var scope = {};

beforeEach(function(){

module('anyModule');

inject(function($controller){

$controller('AnyController',{$scope:scope});

});

});

});

In the preceding case, the test’s scope object is assigned to the actual scope of the controller within the inject function. The scope object can now be used throughout the test, and is also reinitialized before each test.

**Initializing the scope**

To handle following inheritance:

<body ng-app='anyModule'>

<div ng-controller='ParentController'>

<div ng-controller='ChildController'>

</div>

</div>

</body>

As seen in the preceding code, we have this hierarchy of scopes that the ChildController function has access to. In order to test this, we have to initialize the scope object properly in the inject function. Here is how the preceding scope hierarchy can be recreated:

inject(function($controller,$rootScope){

var parentScope = $rootScope.$new();

$controller('ParentController',{$scope:parentScope});

var childScope = parentScope.$new();

$controller('AnyController',{$scope: childScope});

});

There are two main things that the preceding code does:

The $rootScope scope is injected into the test. The $rootScope scope is the highest level of scope that exists.

Each level of scope is created with the $new() method. This method creates the child scope.

**Add comments to the app**

Now that the setup and approach have been decided, we can start our first test. From a testing point of view, as we will be using a top-down approach, we will write our Protractor tests first and then build the application.

Test first, make it run, and make it better.

**Test first**

The scenario given is in a well-specified format already and fits our Protractor testing template:

describe('',function(){

beforeEach(function(){

});

it('',function(){

});

});

Placing the scenario in the template, we get the following code:

describe('Given I am posting a new comment',function(){

describe('When I push the submit button',function(){

beforeEach(function(){

});

it('Should then add the comment',function(){

});

});

});

Following the 3 A’s (Assemble, Act, Assert), we will fit the user scenario in the template.

**Assemble**

The browser will need to point to the first page of the application. As the base URL has already been defined, we can add the following to the test:

beforeEach(function(){

browser.get('/');

});

Now that the test is prepared, we can move on to the next step, Act.

**Act**

The next thing we need to do, based on the user specification, is add an actual comment. The easiest thing is to just put some text into an input box. The test for this, again without knowing what the element will be called or what it will do, is to write it based on what it should be.

Here is the code to add the comment section for the application:

beforeEach(function(){

...

var commentInput = $('input');

commentInput.sendKeys('a comment');

});

The last assemble component, as part of the test, is to push the **Submit** button. This can be easily achieved in Protractor using the click function. Even though we don’t have a page yet, or any attributes, we can still name the button that will be created:

beforeEach(function(){

...

var submitButton = element.all(by.buttonText('Submit')).click();

});

Finally, we will hit the crux of the test and assert the users’ expectations.

**Assert**

The user expectation is that once the **Submit** button is clicked, the comment is added. This is a little ambiguous, but we can determine that somehow the user needs to get notified that the comment was added.

The simplest approach is to display all comments on the page. In AngularJS, the easiest way to do this is to add an ng-repeat object that displays all comments. To test this, we will add the following:

it('Should then add the comment',function(){

var comments = element(by.repeater('comment in comments')).first();

expect(comment.getText()).toBe('a comment');

});

Now, the test has been constructed and meets the user specifications. It is small and concise. Here is the completed test:

describe('Given I am posting a new comment',function(){

describe('When I push the submit button',function(){

beforeEach(function(){

//Assemble

browser.get('/');

var commentInput = $('input');

commentInput.sendKeys('a comment');

//Act

//Act

var submitButton = element.all(by.buttonText('Submit')).

click();

});

//Assert

it('Should then add the comment',function(){

var comments = element(by.repeater('comment in

comments')).first();

expect(comment.getText()).toBe('a comment');

});

});

});

**Make it run**

Based on the errors and output of the test, we will build our application as we go.

1. The first step to make the code run is to identify the errors. Before starting off the site, let’s create a bare bones index.html page:

<!DOCTYPE html>

<html>

<head>

<title></title>

</head>

<body>

</body>

</html>

Already anticipating the first error, add AngularJS as a dependency in the page:

<script type='text/javascript'

src='bower\_components/angular/angular.js'></script>

</body>

2. Now, starting the web server using the following command:

**$ ./node\_modules/http-server/bin/http-server -p 8080**

3. Run Protractor to see the first error:

**$ ./node\_modules/.bin/protractor chromeOnlyConf.js**

4. Our first error states that AngularJS could not be found:

**Error: Angular could not be found on the page http://localhost:8080/ :**

**angular never provided resumeBootstrap**

This is because we need to add ng-app to the page. Let’s create a module and add it to the page.

The complete HTML page now looks as follows:

<!DOCTYPE html>

<html>

<head>

<title></title>

</head>

<body>

<script src="bower\_components/angular/angular.js"></script>

</body>

</html>

**Adding the module**

The first component that you need to define is an ng-app attribute in the index.html page.

Use the following steps to add the module:

1. Add ng-app as an attribute to the body tag:

<body ng-app='comments'>

2. Now, we can go ahead and create a simple comments module and add it to a file named comments.js:

angular.module('comments',[]);

3. Add this new file to index.html:

<script src='app/commentController.js'></script>

4. Rerun the Protractor test to get the next error:

**$ Error: No element found using locator: By.cssSelector('input')**

The test couldn’t find our input locator. You need to add the input to the page.

**Adding the input**

Here are the steps you need to follow to add the input to the page:

1. All we have to do is add a simple input tag to the page:

<input type='text' />

2. Run the test and see what the new output is:

**$ Error: No element found using locator: by.buttonText('Submit')**

3. Just like the previous error, we need to add a button with the appropriate text:

<button type='button'>Submit</button>

4. Run the test again and the next error is as follows:

**$ Error: No element found using locator: by.repeater('comment in comments')**

This appears to be from our expectation that a submitted comment will be available on the page through ng-repeat. To add this to the page, we will use a controller to provide the data for the repeater.

**Controller**

As we mentioned in the preceding section, the error is because there is no comments object. In order to add the comments object, we will use a controller that has an array of comments in its scope. Use the following steps to add a comments object in the scope:

1. Create a new file in the app directory named commentController.js:

angular.module('comments')

.controller('CommentController',['$scope', function($scope){

$scope.comments = [];

}])

2. Add it to the web page after the AngularJS script:

<script src='app/commentController.js'></script>

3. Now, we can add commentController to the page:

<div ng-controller='CommentController'>

4. Then, add a repeater for the comments as follows:

<ul ng-repeat='comment in comments'>

<li>{{comment}}</li>

</ul>

5. Run the Protractor test and let’s see where we are:

**$ Error: No element found using locator: by.repeater('comment in comments')**

Hmmm! We get the same error.

6. Let’s look at the actual page that gets rendered and see what’s going on. In Chrome, go to http://localhost:8080 and open the console to see the page source (*Ctrl* +*Shift* + *J*). You should see something like what’s shown in the following screenshot:



Notice that the repeater and controller are both there; however, the repeater is commented out. Since Protractor is only looking at visible elements, it won’t find the repeater.

7. Now we know why the repeater isn’t visible, but we have to fix it. In order for a comment to show up, it has to exist on the controller’s comments scope. The smallest change is to add something to the array to initialize it as shown in the following code snippet:

.controller('CommentController',['$scope',function($scope){

$scope.comments = ['anything'];

}]);

8. Now run the test and we get the following:

**$ Expected 'anything' to be 'a comment'.**

Here is what the HTML code looks like so far:

<!DOCTYPE html>

<html>

<head>

<title></title>

</head>

<body ng-app='comments'>

<div ng-controller='CommentController'>

<input type='text' />

<ul>

<li ng-repeat='comment in comments'>

{{comment.value}}

</li>

</ul>

</div>

<script src='bower\_components/angular/angular.js'></script>

<script src='app/comments.js'></script>

<script src='app/commentController.js'></script>

</body>

</html>

The comments.js module looks as follows:

angular.module('comments',[]);

Here is commentController.js:

angular.module('comments')

.controller('CommentController',['$scope', function($scope){

$scope.comments = [];

}])

**Make it pass**

With TDD, you want to add the smallest possible component to make the test pass. Since we have hardcoded, for the moment, the comments to be initialized to anything, change anything to a comment; this should make the test pass. Here is the code to make the test pass:

angular.module('comments')

.controller('CommentController',['$scope', function($scope){

$scope.comments = ['a comment'];

}]);

…

Run the test! We get a passing test:

**$ 1 test, 1 assertion, 0 failures**

Wait a second! We still have some work to do. Although we got the test to pass, it is not done. We added some hacks just to get the test passing. The two things that stand out are:

Clicking on the **Submit** button, which really doesn’t have any functionality

Hardcoded initialization of the expected value for a comment

The preceding changes are critical steps we need to perform before we move forward.

They will be tackled in the next phase of the TDD life cycle, that is, make it better (refactor).

**Make it better**

The two components that need to be reworked are:

1. Adding behavior to the **Submit** button
2. Removing hardcoded value of the comments

**Implementing the Submit button**

The **Submit** button needs to actually do something. We were able to sidestep the implementation by just hardcoding the value. Using our tried and trusted TDD techniques, switch to an approach focused on unit testing.

So far, the focus has been on the UI and pushing changes to the code. We haven’t written a single unit test.

For this next bit of work, we will switch gears and focus on driving the development of the **Submit** button through tests. We will be following the TDD life cycle (test first, make it run, make it better).

**Configuring Karma**

Here are the steps you need to follow to configure Karma:

1. Update the files section with the added files:

files: [

...

'app/comments.js',

'app/commentController.js',

...

],

2. Start Karma:

**$ karma start**

3. Confirm that Karma is running:

**$ Chrome 36.0.1985 (Windows 7): Executed 1 of 1 SUCCESS (0.018 secs /0.015 secs)**

**Test first**

Let’s first start with a new file in the spec/unit folder called comments.js. We will use the base template:

describe('', function(){

beforeEach(function(){

});

it('', function(){

});

});

According to the specification, when the **Submit** button is clicked, it needs to add a comment. We will need to fill in the blanks of the three components of a test (Assemble, Act, Assert).

**Assemble**

The behavior will need to be part of a controller for the frontend to use it. The object under test in this case is the controller’s scope object; we will need to add this to assemble this test. To wire up the AngularJS controller we need to initialize the module and then inject the CommentController scope into the test.

var scope = {};

beforeEach(function(){

module('comments');

inject(function($controller){

$controller('CommentController',{$scope:scope});

});

...

})

Now, the controller’s scope object, which is under test, is available to the test.

**Act**

The specification determines that we need to call a add method in the scope object. Add the following code to the beforeEach section of the test:

beforeEach(function(){

…

scope.add('any Comment');

});

Now for the assertion:

**Assert**

Assert that the comment items in the scope object now contain any comment as the first element. Add the following code to the test:

it('',function(){

expect(scope.comments[0]).toBe('any comment');

});

Save the file and let’s move on to the next step of the life cycle and make it run (execute).

**Make it run**

Now that we have most of the test prepared, we need to make the test pass. Looking at the output of the console where Karma is running, we see the following:

**$ TypeError: undefined is not a function…unit/comments.js:4:9**

Looking at the line number, that is 4:9, of our unit test, we see that this is the add function.

Let’s go ahead and put in an add function into the controller’s scope object using the following steps:

1. Open the controller scope and create a function named add:

$scope.add = function(){}

2. Check Karma’s output and let’s see where we are:

**$ Expected 'a comment' to be 'any comment'.**

3. Now, we have hit the expectation. Remember to think of the smallest change to get this to work. Modify the add function to set the $scope.comments array to any comment when called:

$scope.add = function(){

$scope.comments.unshift('any comment');

};

Unshift is a standard JavaScript function that adds an item to the front of an array.

4. When we check Karma’s output, we see the following:

**$ Chrome 36.0.1985 (Windows 7): Executed 1 of 1 SUCCESS**

Success! The test passes, but again needs some work. Let’s move on to the next stage and make it better (refactor).

**Make it better**

The main point that needs to be refactored is the add function. It doesn’t take any arguments! This should be straightforward to add, and simply confirm that the test still runs. Update the add function of CommentController.js to take an argument and use that argument to add to the comments array:

$scope.add = function(commentToAdd){

$scope.comments.unshift(commentToAdd);

};

Check the output window of Karma and ensure that the test still passes.

The complete unit test looks as follows:

describe('',function(){

var scope = {};

beforeEach(function(){

module('comments');

inject(function($controller){

$controller('CommentController', {$scope:scope});

});

scope.add('any comment');

});

it('',function(){

expect(scope.comments[0]).toBe('any comment');

})

});

The CommentController file now looks as follows:

angular.module('comments')

.controller('CommentController', ['$scope', function($scope){

$scope.comments = [];

$scope.add = function(commentToAdd){

$scope.comments.unshift(newComment);

};

}]);

**Back up the test chain**

We completed the unit test and addition of the add function. Now we can add the function to specify the behavior of the **Submit** button. The way to link the add method to the button is to to use the ng-click attribute. The steps to add behavior to the **Submit** button are as follows:

1. Open the index.html page and link it as follows:

<button type="button" ng-click="add('a comment')">Submit</button>

Is the value hardcoded? Well, again, we want to do the smallest change and ensure that the test still passes. We will work through our refactors until the code is how we want it, but instead of a big bang approach, we want to make small incremental changes.

2. Now let’s rerun the Protractor test and ensure that it still passes. The output says it passes, and we are okay. The hardcoded value wasn’t removed from the comments.

Let’s go ahead and remove that now. The CommentsController file should now look as follows:

$scope.comments = [];

3. Run the test and see that we still get a passing test.

We need to remove the hardcoded value in ng-click. The comment being added should be determined by the input in the comment input text.

**Bind the input**

Here are the steps you need to follow to bind the input:

1. To be able to bind the input into something meaningful, add an ng-model attribute to the input tag:

<input type='text' ng-model='newComment'/>

2. Then, in the ng-click attribute, simply use the newComment model as the input:

<button type='button' ng-click='add(newComment)'>Submit</button>

Run the Protractor test and confirm that everything has passed and is good to go.

**Next Specification**

Now that we have the first specification working end-to-end and unit tested, we can start the next specification. The next specification states that the users want the ability to like a comment.

We will use the same top-down approach and start our test from a Protractor test. We will continue to follow the TDD life cycle, that is, test first, make it run, make it better.

**Test first**

Following the pattern, we will start with a basic Protractor test template:

describe('',function(){

beforeEach(function(){

});

it('', function(){

});

});

When we fill in the specification, we get the following:

describe('When I like a comment',function(){

beforeEach(function(){

});

it('should then be liked', function(){

});

});

With the template in place, we are ready to construct the test.

**Assemble**

The assembly of this test will require a comment to exist. Place the comment within the existing posted comment test. It should look similar to this:

describe(''Given I am posting a new comment', function(){

describe('When I like a comment',function(){

…

});

});

**Act**

The user specification we test is that the like button performs an action for a specific comment. Here are the steps that will be required and the code required to do them (note that the following steps will be added to the beforeEach text):

1. Store the first comment so that it can be used in the test:

var firstComment = null;

beforeEach(function(){

…

2. Find the first comment’s like button:

var firstComment = element.all(by.repeater('comment in

comments').first();

var likeButton = firstComment.element(by.buttonText('like'));

3. The code for the like button when it is clicked is as follows:

likeButton.click();

**Assert**

The specification expectation is that once the comment has been liked, it is liked. This is best done by putting an indicator of the number of likes, and ensuring the count is 1. The code will then be as follows:

it('Should increase the number of likes to one',function(){

var commentLikes = firstComment.element(by.binding('likes'));

expect(commentLikes.getText()).toBe(1);

});

The created test now looks as follows:

describe('When I like a comment',function(){

var firstComment = null;

beforeEach(function(){

//Assemble

firstComment = element.all(by.repeater('comment in comments').first();

var likeButton = firstComment.element(by.buttonText('like'));

//Act

likeButton.click();

});

//Assert

it('Should increase the number of likes to one', function(){

var commentLikes = firstComment.element(by.binding('likes'));

expect(commentLikes.getText()).toBe(1);

});});

**Make it run**

The test has been prepared and is itching to run. We will now run the test and fix the code until the test passes. The following steps will detail the error and the fix cycle required to make the test path:

1. Run Protractor.

2. View the error message in the command line:

**$ Error: No element found using locator: by.buttonText("like")**

3. As the error states, there is no like button. Go ahead and add the button:

<li ng-repeat='comment in comments'>

{{comment}}

<button type="button">like</button>

</li>

4. Run Protractor.

5. View the next error message:

**$ Expected 'a comment like' to be 'a comment'.**

6. By adding the like button, we caused our other test to fail. The reason is our use of the getText() method. Protractor’s getText() method gets the inner text including inner elements. To fix this, we will need to update the previous test to include like as part of the test:

it('Should then add the comment',function(){

var comments = element.all(by.repeater('comment in comments')).first();

expect(comments.getText()).toBe('a comment like');

});

7. Run Protractor.

8. View the next error message:

**$ Error: No element found using locator: by.binding("likes")**

9. Time to add a likes binding. This one is a little more involved. Likes needs to be bound to a comment. We need to change the way the comments are held in the controller. Comments need to hold the comment value and the number of likes. A comment should be an object like this: {value:'',likes:0}. Again, the focus of this step is just to get the test to pass. The next step is to update the controller’s add function to create comments based on the object we described in the preceding steps.

Open commentController.js and edit the add function as follows:

$scope.add = function(commentToAdd){

var newComment = {value:commentToAdd,likes:0};

$scope.comments.unshift(newComment);

};

10. Update the page to use the value for the comment:

<li ng-repeat='comment in comments'>

{{comment.value}}

11. Before rerunning the Protractor test, we need to add the new comment.likes binding to the HTML page:

<li ng-repeat='comment in comments'>

… {{comment.likes}}

12. Now rerun the Protractor tests and let’s see where the errors are:

**$ Expected 'a comment like 0' to be 'a comment like'**

13. Because the inner text of the comment has changed, we need to change the expectation of the test:

it('Should then add the comment', function(){

…

expect(comments.getText()).toBe('a comment like 0');

});

14. Run Protractor:

**$ Expected '0' to be '1'.**

15. Now, we are finally down to the expectation of the test. In order to make this test pass, the smallest change will be to make the like button update the likes on the comment array. The first step is to add a like method on the controller, which will update the number of likes:

$scope.like = function(comment){

comment.likes++;

};

16. Link the like method to the HTML page using an ng-click attribute on the button as follows:

<button type="button" ng-click='like(comment)'>like</button>

17. Run Protractor and confirm that the tests pass!

The page now looks as follows:



Compared to the drawing at the beginning, all the features have been created. Now that we made the test pass in Protractor, we need to check the unit tests to ensure that our changes didn’t break the unit tests.

**Fixing the unit tests**

One of the primary changes required was to make the comment an object, consisting of a value and number of likes. Before thinking too much about how the unit tests could have been affected, let’s kick them off. Execute the following command:

**$ karma start**

As expected, the error is related to the new comment object:

**$ Expected { value : 'any comment', likes : 0 } to be 'any comment'.**

Reviewing the expectation, it seems like the only thing required is for comment.value to be used in the expectation as opposed to the comment object itself. Change the expectation as follows:

it('',function(){

var firstComment = scope.comments[0];

expect(firstComment.value).toBe('any comment');

})

Save the file and check the Karma output. Confirm that the test passes.

Both the Karma and Protractor tests pass and we have completed the primary user behaviors of adding a comment and liking it. You are free now to move on to the next step and make things better.