Better $http Mock DSL

*Status: Draft*

*Authors: pete@bacondarwin.com*

# Objective

The current manner of mocking the $http service is clunky and inflexible. The aim here is to design a new DSL for creating mock responses to $http requests.

# Background

Code that makes async HTTP requests is inherently tricky to test. The key is to make the async code become synchronous in the tests. The current ngMock $httpBackend does this but it relies on a record and replay style of coding, which means that your expectations must be defined away from the actual flow of the test. The interface for this mock code could be more fluent and flexible.

# Prior Art

## ngMock in Angular 1.x

The current Angular 1.x method for writing unit tests that involve $http server requests is to mock out the requests and responses using the ngMock $httpBackend service. This service overwrites the production $httpBackend service and allows the unit test developer to provide mock responses to requests, trigger these responses synchronously and check that the expected (and only expected) requests were made.

The ngMock $httpBackend lets you define mock responses that should be returned when a service requests a particular URL and HTTP method:

$httpBackend.when(‘GET’,‘some/url’).respond({ some: ‘data’ });

See <https://docs.angularjs.org/api/ngMock/service/$httpBackend#when> for more detail.

The code under test can then make GET requests to ‘some/url’ which are tracked. In the unit test you would then call $httpBackend.flush() to trigger all pending requests to be fulfilled, if they have a mock backend response defined.

Mock responses created using the “when” method above do not provide enough control for some unit tests, where you’d want to be able to assert that requests were definitely made and made in a specified order.

The ngMock $httpBackend supports this by allowing you to define mock responses that are expected:

$httpBackend.expect(‘GET’, ‘some/url’).respond({ some: ‘data’ });

In this case the expected request must and can only be made once per definition and requests must be made in order.

Also the $httpBackend.flush() method provides limited control over which requests are fulfilled and in what order.

## ngHttp Angular 2.x

There is some work in the Angular 2.x development to provide a similar improvement to what is being discussed here. There is a new project called ngHttp. Its implementation is still under development and will not be available directly in Angular 1.x. See <https://docs.google.com/document/d/1DMacL7iwjSMPP0ytZfugpU4v0PWUK0BT6lhyaVEmlBQ/edit?usp=sharing>

We should ensure that any changes to 1.x resulting from this doc are in parity with ngHttp mocks.

# Detailed Design

It is suggested defining and checking mock requests and responses should be done inline with the code that drives the unit under test. This will make it easier to follow the intention of the tests.

Rather than defining up front what the expected requests and responses should be, the new interface would allow the code under test to make requests and then the test code would explicitly check what requests have been made and trigger responses synchronously.

The $httpBackend would expose the following properties and methods:

* pendingRequests - a PendingRequestsCollection. This collection holds all the requests that have been made and not yet resolved.

The PendingRequestsCollection has the following properties and methods:

* count - the number of requests that have not yet been resolved.
* get(requestProps) - returns the first PendingRequest that matches the passed parameter. If there is E.g. $httpBackend.pendingRequests.get({url:’/some/url’, method:’POST’});
* all(requestProps) - returns all the requests that match the passed parameter.

PendingRequest objects have the following properties and methods:

* respondWith(data, headers, status) - a helper method that will synchronously resolve the response.
* config - the original config of the request
* response - the deferred object that can be resolved with the response to the request.

Consider the example in the docs. The current implementation looks like this:

// testing controller

describe('MyController', function() {

var $httpBackend, $rootScope, createController;

beforeEach(inject(function(\_$httpBackend\_, \_$rootScope\_) {

$httpBackend = \_$httpBackend\_;

$rootScope = \_$rootScope\_;

// backend definition common for all tests

$httpBackend.when('GET', '/auth.py').respond({userId: 'userX'}, {'A-Token': 'xxx'});

createController = function() {

return $controller('MyController', {'$scope' : $rootScope });

};

}));

afterEach(function() {

$httpBackend.verifyNoOutstandingExpectation();

$httpBackend.verifyNoOutstandingRequest();

});

it('should fetch authentication token', function() {

$httpBackend.expectGET('/auth.py');

var controller = createController();

$httpBackend.flush();

});

it('should send msg to server', function() {

// In this test you don’t care about the authentication, but

// the controller will still send the request and

// $httpBackend will respond without you having to

// specify the expectation and response for this request

// because of the `when()` definition in the `beforeEach` block

var controller = createController();

$httpBackend.flush();

$httpBackend.expectPOST('/add-msg.py', 'message content').respond(201, '');

$rootScope.saveMessage('message content');

expect($rootScope.status).toBe('Saving...');

$httpBackend.flush();

expect($rootScope.status).toBe('');

});

it('should send auth header', function() {

var controller = createController();

$httpBackend.flush();

$httpBackend.expectPOST('/add-msg.py', undefined, function(headers) {

// check if the header was send, if it wasn't the expectation won't

// match the request and the test will fail

return headers['Authorization'] == 'xxx';

}).respond(201, '');

$rootScope.saveMessage('whatever');

$httpBackend.flush();

});

});

The new interface would look like this:

// testing controller

describe('MyController', function() {

var $httpBackend, $rootScope, createController;

beforeEach(inject(function(\_$httpBackend\_, \_$rootScope\_) {

$httpBackend = \_$httpBackend\_;

$rootScope = \_$rootScope\_;

createController = function() {

return $controller('MyController', {'$scope' : $rootScope });

};

}));

afterEach(function() {

expect($httpBackend.pendingRequests.count).toEqual(0);

});

describe('pre-authentication', function() {

it('should fetch authentication token', function() {

var controller = createController();

$httpBackend.pendingRequests.get('GET', '/auth.py')

.respondWith({userId: 'userX'}, {'A-Token': 'xxx'});

});

});

describe('post-authentication', function() {

var controller;

beforeEach(function() {

// In this test you don’t care about the authentication, but

// the controller will still send the request and

// $httpBackend should respond

controller = createController();

$httpBackend.pendingRequests.get('GET', '/auth.py')

.respondWith({userId: 'userX'}, {'A-Token': 'xxx'});

};

it('should send msg to server', function() {

$rootScope.saveMessage('message content');

expect($rootScope.status).toBe('Saving...');

$httpBackend.pendingRequests.get({

method: 'POST',

url: '/add-msg.py',

data: 'message content'

})

.respondWith({ status: 201, data: { 'message response' });

expect($rootScope.status).toBe('message response');

});

it('should send auth header', function() {

$rootScope.saveMessage('whatever');

$httpBackend.pendingRequests.get({

method: 'POST',

url: '/add-msg.py',

headers: {'Authorization': 'xxx'})

.respondWith({ data: 'POST', status: 201, data: '');

});

});

# Caveats

This change in design is not compatible with the way that the current ngMock $httpBackend works. We can provide an opt-in or opt-out strategy by simply creating a new module (ngHttpMock, say) and then the unit test developer would explicitly choose which version of the mock they want be loading or not loading this new module.

The suggested API above tends to be quite verbose for situations where you want to provide numerous similar or identical responses to numerous requests. But one could argue that this implies a poor isolation of the unit under test?

# Security Considerations

This is unit testing and so there are limited if any security requirements.

# Performance Considerations / Test Strategy

The pending requests will be stored in hash maps so looking up requests for checking expectations and providing responses will be fast.

# Work Breakdown

The new API needs to be agreed.

The API can be implemented in a single