

# TESTING WEB APIS

Frank Walsh

# AGENDA

- Testing
- Test Driven Dev/Behaviour Driven Dev
- Automated Testing with Postman
  - Postman Collections
  - Postman Variables
  - Assertion framework: Chai

# TEST CATEGORIES

## Static testing

Find typos/basic syntax errors

## Unit Testing

Test one single unit in isolation

## Integration Testing

Separate units work together

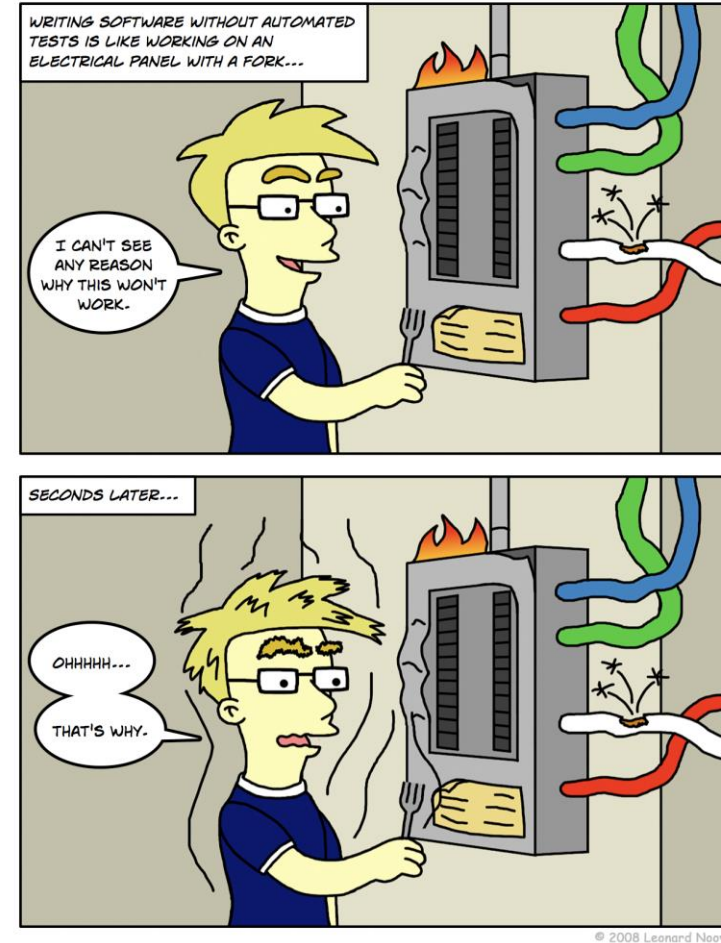
## End-to-End

Complete flow of project



# UNIT TESTING

- Code written by developer that exercises a small, specific area of functionality.
- “Program testing can be used to show the presence of bugs, but never to show their absence!” – Dijkstra
- Up to now – Manual tests with Postman
  - Not structured
  - Not repeatable
  - Not easy



# UNIT TESTS

- Unit Tests are specific pieces of code
- Tests are written by developers of the code, usually
  - Sometimes before the code is written
- Part of the code repository
  - They go where the code goes
- Use a framework
  - Junit, Jasmine, Chai, Mocha

UNIT  
TESTS



*API Testing*

# UNIT TEST CONVENTION

- All objects and methods
- Look for 100% coverage
  - Although property getters/setters are sometimes omitted
- All tests should pass before commits?

/

88.99% Statements 1948/2188 66.18% Branches 272/411 82.15% Functions 359/437 89.06% Lines 1937/2175

File		Statements		Branches		Functions		Lines
lib/	<div><div></div></div>	81.82%	72/88	25%	3/12	57.14%	8/14	81.82%
lib/agent/	<div><div></div></div>	84.16%	271/322	56.72%	38/67	69.09%	38/55	84.16%
lib/agent/api/	<div><div></div></div>	80.9%	144/178	45.45%	10/22	75%	27/36	80.9%
lib/agent/healthcheck/	<div><div></div></div>	100%	20/20	100%	0/0	100%	6/6	100%
lib/agent/metrics/	<div><div></div></div>	100%	4/4	100%	0/0	100%	0/0	100%
lib/agent/metrics/apm/	<div><div></div></div>	94.44%	85/90	55.56%	5/9	100%	20/20	94.44%
lib/agent/metrics/externalEdge/	<div><div></div></div>	100%	73/73	92.86%	13/14	100%	16/16	100%
lib/agent/metrics/incomingEdge/	<div><div></div></div>	100%	85/85	100%	14/14	100%	19/19	100%
lib/agent/metrics/rpm/	<div><div></div></div>	100%	56/56	75%	6/8	100%	10/10	100%
lib/instrumentations/	<div><div></div></div>	86.37%	393/455	56.86%	58/102	74.31%	81/109	86.37%
lib/instrumentations/core/http/	<div><div></div></div>	95.31%	305/320	84.62%	66/78	86.54%	45/52	95.31%

# INTEGRATION TESTING

- Combines several units into a test
- Exposes faults in interaction between integrated units
- Usually done after unit testing
- Performed by devs and independent testers



# TESTING OUR API

- Is this integration or unit testing?
  - Integration testing, because you have to run a web server (locally)
  - Your Web API is an “Application boundary”
    - Requires HTTP to interact with it
  - And you’ve a DB/3<sup>rd</sup> party APIs going
  - So you’re testing more than just your code...

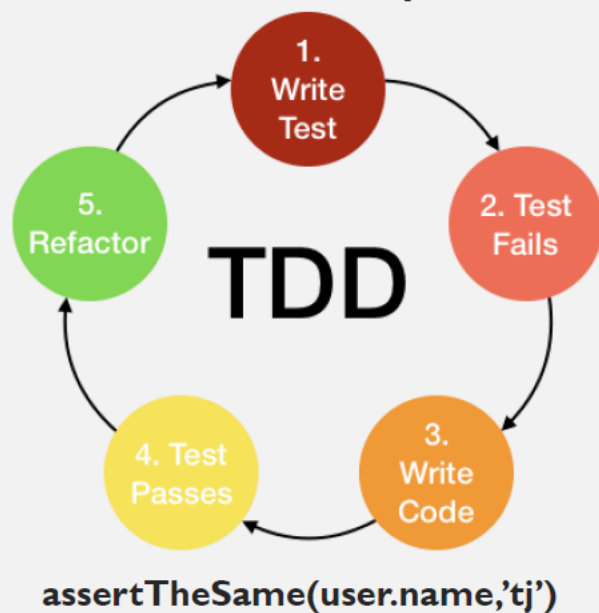


This Photo by Unknown Author is licensed under CC BY-SA-NC



## ASIDE – TDD AND BDD

- **Test Driven Development**



- **Behaviour Driven Development**

- Specify desired behaviour of the unit
- Based on requirements set by the business
- Behavioural specification from business and developer

```
expect(user).to.have.property('name').equal('tj')
```

```
user.should.have.property('name', 'tj');
```

# TDD

- Developers only
- Code
- Low level
- Build the thing right

# BDD

- Whole team
- Prose
- High level
- Build the right thing

- Test first
- Automation

# TESTING TOOLS

- **Test Frameworks**

- Makes it easier to write tests
- Provide hooks, test suites, test runners
- Examples Junit, VS Team Test, PHP Unit, Mocha

- **Assertion Frameworks**

- Perform checks and decisions
- Examples: assert, chai.js, should.js

- **Mocking Frameworks**

- Create mock dependencies, stubs, proxys
- Sinon, Jmock, Mockito, Mockgoose!



# AUTOMATED TESTING WITH POSTMAN

# POSTMAN TESTING

- Up to now, manual
- Fine for initial development cycle
- Better to have more structured method
  - Regression: check everything still works when you make a change and before committing
  - Use HTTP requests to test Express App

# CHAI

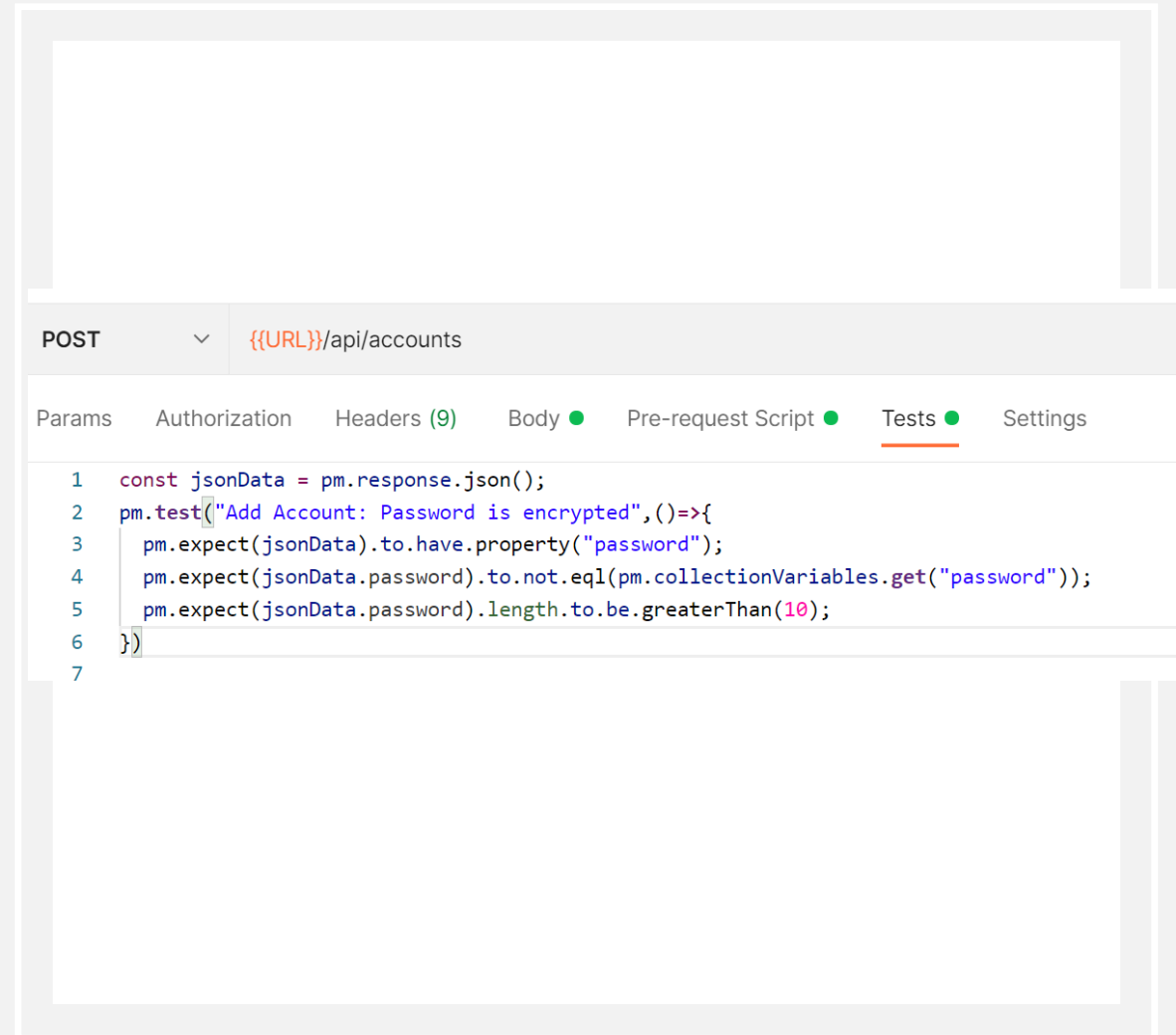
- BDD / TDD assertion library
  - Run in browser and server-side (e.g. node)
- Features
  - Expressive syntax
  - Can test Async code (Promises)
  - Pluggable
    - Compatible with test runners such as Karma



Chai Assertion Library

# TESTING OVER HTTP WITH **POSTMAN**

- Postman includes the **Chai** assertion Library by default
- Provide a high-level abstraction for testing HTTP
- Can specify pre-request and test scripts as part of Postman Request
- Scripts are run when request is sent
  - Pre-request script can be used to set up scenario(fixture)
  - Tests script can be used to check request and response is as expected.



# ASSERTIONS WITH CHAI **EXPECT**

- Chai has several interfaces.
  - Should, Expect, Assert
- Expect allows you to chain together

Readable assertions

- Write tests that are closer to natural language.
  - Suitable for BDD
- Chai plugin for postman uses Expect interface

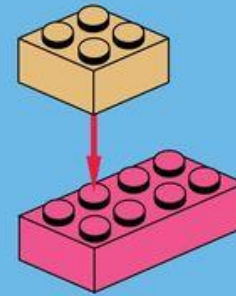
```
var expect = require('chai').expect
, foo = 'bar'
, beverages = { tea: [ 'chai', 'matcha', 'oolong' ] };

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(beverages).to.have.property('tea').with.lengthOf(3);
```



# HOW CHAI WORKS WITH POSTMAN...

- Define Request in Postman as before
- Use the Tests tab to define your test in Javascript
- The **pm** object provides functionality for testing your request and response data.
  - provides access to request and response data, and variables.
- Provide description of unit test using “**test**”
- Use “**expect()**” to define several test cases into it.



## EXAMPLE – GET AUTHENTICATION TOKEN

**Scenario:** Test the authentication endpoint returns a Token

- Build request in Postman.
- Define test in Tests tab
- **pm.response.json()** returns response body json object
- **pm.test(..)** takes test name and runs test function
- The **test function** specifies the test that uses the pm object to define what's expected (e.g. content type, status)
- Use **pm.expect(..)** to check response object

POST

localhost:3000/api/accounts/security/token

Params

Authorization

Headers (9)

Body ●

Pre-request Script

Tests ●

Settings

```
1  const jsonData = pm.response.json();
2
3  pm.test("Authenticate Account: Successful Response", ()=>pm.response.to.have.status(200))
4
5  pm.test("Authenticate Account: Response Object contains right properties", ()=>{
6    pm.expect(jsonData).to.be.an("object");
7    pm.expect(jsonData.token).to.be.a("string");
8  })
```

# POSTMAN COLLECTIONS

## Account Registration and Movies Access

- Collection of related requests to test an API
- Can structure a collection run order to test process flow in API

Create Account



Get Authentication  
Token



Get Movies

### RUN ORDER

- ✓ POST Add Account
- ✓ POST Get Token
- ✓ GET Get Movies

# POSTMAN VARIABLES

- *Variables* enable you to store and reuse values in Postman
  - Handy for repeatable testing
- Can store the URL in a variable URL and reference it in your requests using `{{URL}}`

The screenshot displays the Postman interface. At the top, the 'Movies' environment is selected, with a 'Fork' button on the right. Below this is a table of environment variables:

	VARIABLE	TYPE ⓘ	INITIAL VALUE ⓘ	CURRENT VALUE ⓘ
<input checked="" type="checkbox"/>	URL	default ▾	http://localhost:8080	http://localhost:8080
<input checked="" type="checkbox"/>	movieId	default ▾	500	500
	Add a new variable			

Below the table, the breadcrumb 'Accounts2022 Test Right / Add Account' is visible. The main area shows a 'POST' request to the endpoint `{{URL}}/api/accounts`. At the bottom, a tabbed interface includes 'Params', 'Authorization', 'Headers (9)', 'Body', 'Pre-request Script', 'Tests' (which is the active tab, indicated by a red underline), and 'Settings'.

# VARIABLE SCOPES

- Global
  - access data between collections
- Collection
  - available throughout the requests in a collection
- Environment
  - scope your work to different environments, for example local development versus testing or production.

# POSTMAN TESTING EXAMPLE

1

## Create Collection

- Collection of related API requests

2

## Create Environment

- Define commonly used variables

3

## Create Tests

- Check response is as expected

4

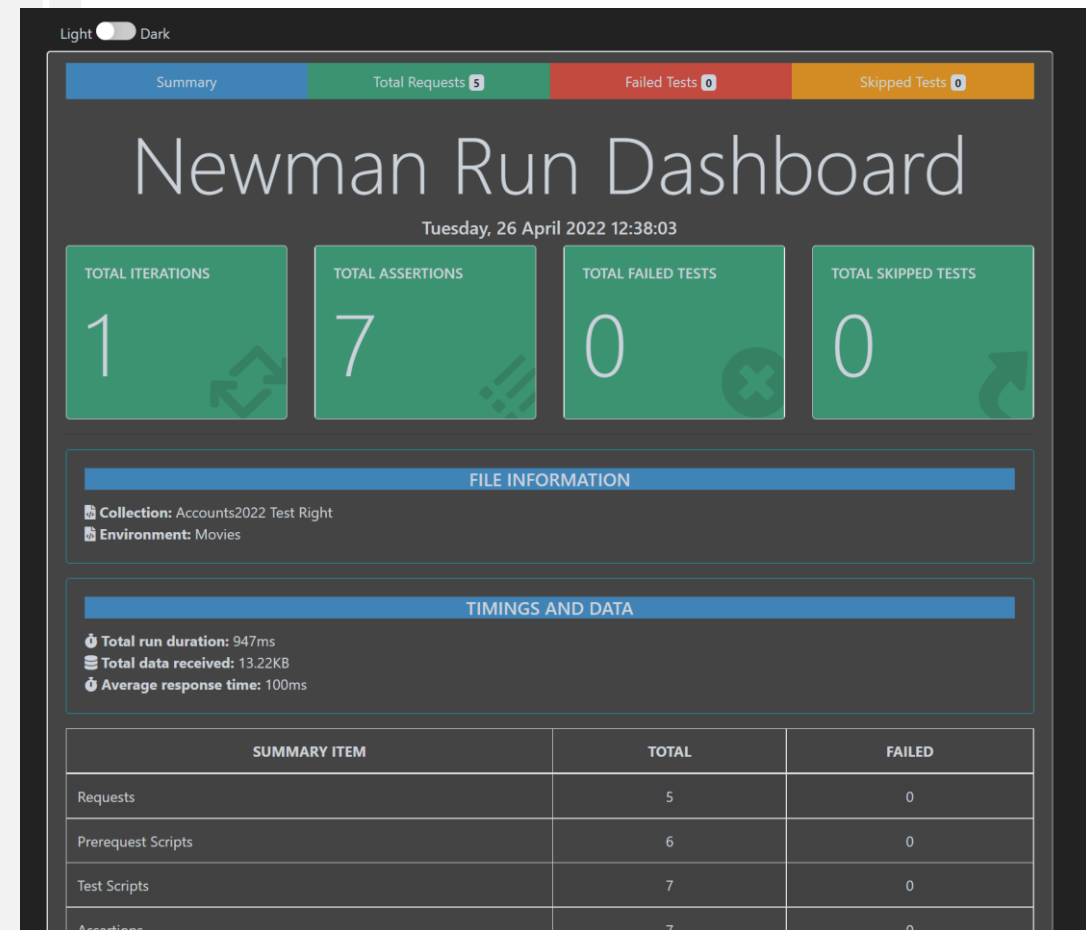
## Run Tests!

# RUNNING THE TEST EXTERNALLY USING NEWMAN

Newman is a command-line collection runner for Postman. Can use it to execute your tests from command line and integrate into Continuous Integration/Continuous Delivery pipeline.

- Export Collection as JSON file
- Export Environment as JSON file
- Install Newman and Newman-html-extre and run on command line
- Add test script to your package.json file

```
"scripts": {  
  "start": "nodemon --exec babel-node index.js",  
  "test": "newman run ./tests/collection1.json -e ./tests/env1.json --reporters htmlextra"  
},  
"author": "fxwalsh",
```



# TESTING STRATEGIES

- Right-BICEP
  - Right – are results CORRECT
  - B – are boundary conditions correct
  - I – check inverse relationship
  - C – Cross check result using other means
  - E – Force error conditions
  - P – Performance characteristics



# MOCKING/STUBBING

FYI....

# MOCKING FRAMEWORK

- What if your code has methods that use/integrate a DB?
- What if your code uses an API that's not ready
- Can use mocking and stubs to override/replace/mutate aspects of the code to allow you to test various scenarios in an isolated fashion
- Examples: Proxyquire, Sinon

**REAL SYSTEM**



**CLASS IN UNIT TEST**



# IMPROVEMENTS - MOCKING

- Unit testing should only concern the unit you're testing
  - Should be independent of servers/db dependencies
- Tests should just test the unit in question
- Unit under test may have dependencies on other (complex) units, e.g. database
- To isolate the behaviour of a unit, replace dependencies by “mocks” that simulate the behaviour
- DBs are impractical to incorporate into the unit test.
- In short, mocking is creating objects that simulate the behaviour of real objects.



# MOCKING MONGODB

- Several mocking frameworks out there
  - Mockery, PowerMockito
- We use Mongoose
  - How about “Mockgoose”?!
  - Turns out it exists!
- NPM install `–save-dev Mockgoose`



# MOCKGOOSE

- Mockgoose spins up **mongod** when `mongoose.connect` call is made.
- Just uses memory store with no persistence.
- Can take a while on first test, after which it's fast
  - Tests may time out
  - You can increase mocha wait time

```
describe (...){  
    this.timeout(10000);
```

```
15 // Connect to database
16 if (nodeEnv == 'test'){
17     var mockgoose = new Mockgoose(mongoose);
18     mockgoose.prepareStorage().then(function() {
19         mongoose.connect(config.mongodb);
20     });
21 }
22 else
23 {
24     mongoose.connect(config.mongodb);
25 }
```