Automated Testing

With a focus on the front end

Demo Repo:
https://github.com/tbenyon/automated-testing-talk

Tom Benyon
Head of Development
tom.benyon@mypthub.net

What we're going to discuss

- What and Why?
- Types of tests
- Vocabulary (Test doubles, Unit, Assertion)
- Approaches (TDD / BDD / Mockist vs Classical)
- What to avoid
- How to get started in your team
- Code Demo - (Front End - Javascript / Jest / Vue)

What is Automated testing?

- Traditionally - manual human based testing
- Automated tests
  - Can be code testing code
  - Can be UI level

Why Test?

- Adds confidence in code
- Catches bugs before they reach production
- Documents code with expected behaviour
- Encourages you to write better code
- Progress towards Continuous Delivery - the holy grail :)
  - This requires a lot more than just unit tests
  - A good aspiration

Example of my mistake

- Full Testing for this bug required:
  1. Resetting the database to trial user
  2. Visiting the purchase custom app page
  3. Resetting the database to expired trial
  4. Visiting the purchase custom app page
  5. Upgrading the account to standard plan
  6. Visiting the purchase custom app page
  7. Upgrading the account to premium plan
  8. Visiting the purchase custom app page

```
computed: {
    canPurchase() {
        return this.isSubscribed && !this.isTrial && currentUser.version === 3;
    },
    ...
}
```

this.

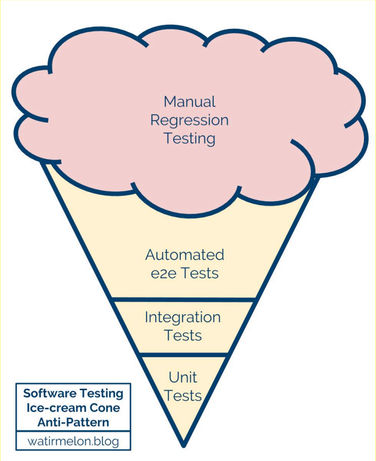
- My refactor moved logic from template to computed
- I only re-tested step one (the focus of the bug) after the refactor

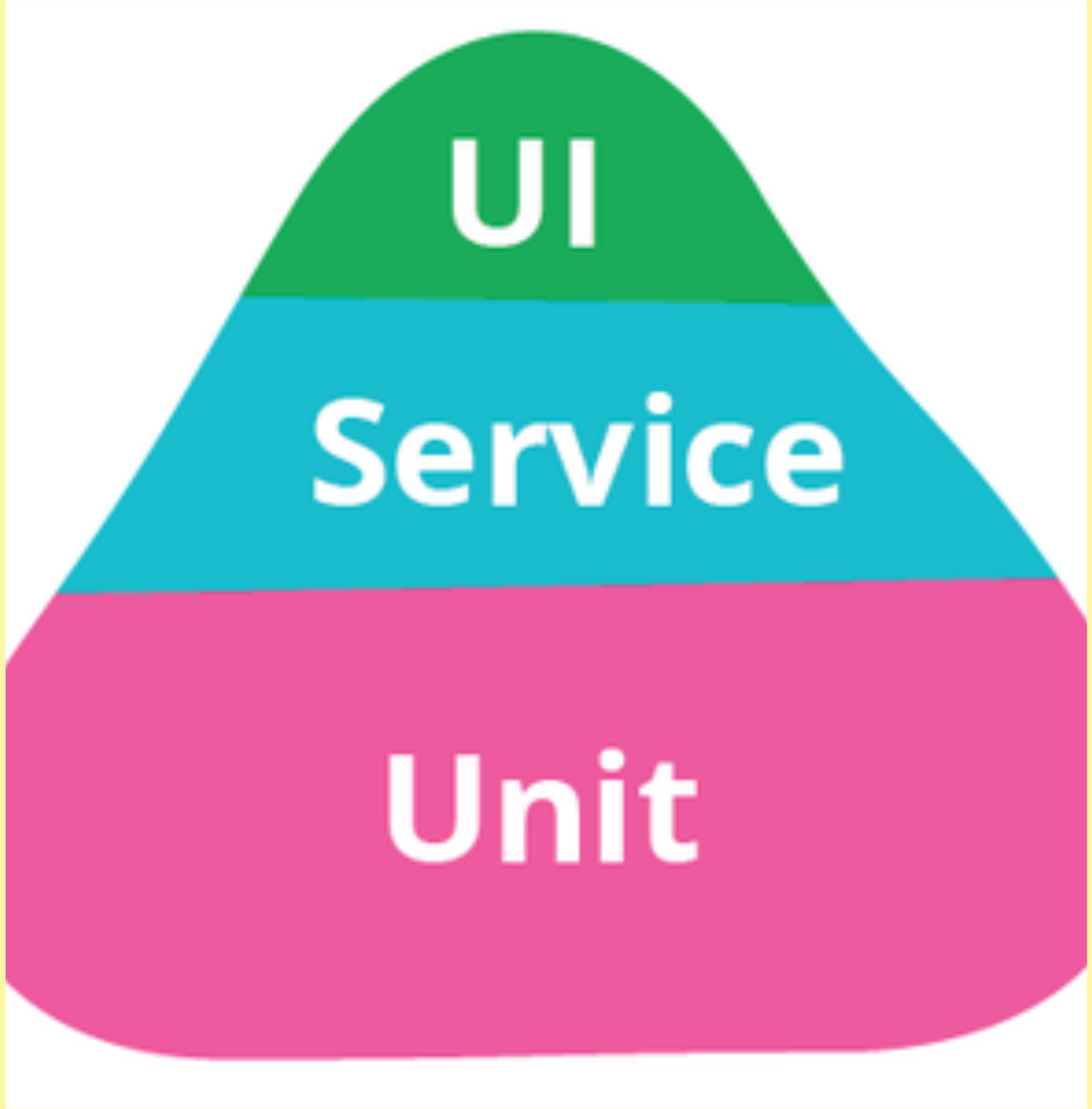
Types Of Automated Test

- Unit Tests
- Integration tests / Service level tests
- End to end

What Type Of Testing Should we Prioritise?

- Where does manual testing fit in this pyramid?
- Avoid the ice cream cone





https://martinfowler.com/bliki/TestPyramid.html

Test Doubles - Mocking VS STUBS

> “
> Test Double is a generic term for any case where you replace a production object for testing purposes.
>
> https://martinfowler.com/bliki/TestDouble.html

- There is wide range of definitions for ‘test doubles’
  - stub, dummy, fake
  - Mocked code that is just there to let the test run
- mock, spy
  - Allows assertions against behaviour
  - Mocks some or all of the behaviour of code

Key Vocabulary

Unit - a single responsibility / unit of functionality

Assertion - the part of the test after setup and logic impl
validates if the test passed

Methods

- `expect(value)`
- `expect.extend(matchers)`
- `expect.anything()`
- `expect.any(constructor)`
- `expect.arrayContaining(array)`
- `expect.assertions(number)`
- `expect.hasAssertions()`
- `expect.not.arrayContaining(array)`
- `expect.not.objectContaining(object)`
- `expect.not.stringContaining(string)`
- `expect.not.stringMatching(string | regexp)`
- `expect.objectContaining(object)`
- `expect.stringContaining(string)`
- `expect.stringMatching(string | regexp)`
- `expect.addSnapshotSerializer(serializer)`
- `.not`
- `.resolves`
- `.rejects`
- `.toBe(value)`
- `.toHaveBeenCalled()`
- `.toHaveBeenCalledTimes(number)`
- `.toHaveBeenCalledWith(arg1, arg2, ...)`
- `.toHaveBeenLastCalledWith(arg1, arg2, ...)`
- `.toHaveBeenNthCalledWith(nthCall, arg1, arg2, ....)`
- `.toHaveReturned()`
- `.toHaveReturnedTimes(number)`
- `.toHaveReturnedWith(value)`
- `.toHaveLastReturnedWith(value)`

Example Test

```
it( name: 'should return an object with the date formatted correctly and add a life',  fn: async () => {
  const testObject = {
    lives: 4,
    timeStamp: 1620079226000
  }
  const result = formatObject(testObject);
  expect(result).toBe( expected: {
    lives: 5,
    date: "2021-05-03T22:00:26.000Z"
  });
});
```

What is Test Driven Development (TDD)?

1. Decide on the smallest additional piece of functionality you can add
2. Write a test for that functionality (it should fail)
3. Write logic that fixes that test
4. Run all the tests

5. Break the logic
6. Ensure the tests fail

What is BDD?

- Behaviour Driven Development
- A principle that we use with TDD
- Write tests:
  - based on behaviours
  - in human readable language

> Given some initial context,
> When an event occurs,
> then ensure some outcomes.

```
describe('When a value is passed in', () => {
  it('should render the correct value', async () => {
    const valueOutputElement = wrapper.get('span');
    wrapper.setProps({ value: 1000000 });
    await waitRaf(wrapper);
    expect(valueOutputElement.text()).toBe('£1,000,00');
  });
  it('should render the correct currency', async () => {
    // ... logic here :)
  });
```

```
FAIL  tests/unit/components/AnimatedCount.spec.js
 ● AnimatedCount.vue › When a value is passed in › should render the correct value

   expect(received).toBe(expected) // Object.is equality

   Expected: "£1,000,00"
   Received: "£1,000,000"

     24 |       wrapper.setProps({ value: 1000000 });
     25 |       await waitRaf(wrapper);
   > 26 |       expect(valueOutputElement.text()).toBe('£1,000,00');
        |                                         ^
     27 |     });
     28 |   });
     29 | });
```

Code Coverage as a Tool

- Don’t get hung up on 100% coverage
- Use tools to prevent you missing key logic that should be tested
- The more strictly you adhere to the TDD process, the less these tools are required

```
 70  1x  import apiRequest from '@/mixins/apiRequest';
 71  1x  import EntityImage from '@/components/EntityImage';
 72  1x  import PackagePreview from '@/pages/Marketplace/PackagePreview';
 73  1x  import LayoutAlpha from '@/layouts/LayoutAlpha';
 74  1x  import StickyWrapper from '@/components/StickyWrapper';
 75  1x  import moment from 'moment';
 76  1x  import currentUser from '@/mixins/currentUser';
 77
 78 56x  export default {
 79        name: 'MarketplacePackageView',
 80        components: { StickyWrapper, LayoutAlpha, PackagePreview, EntityImage },
 81        mixins: [apiRequest, currentUser],
 82        computed: {
 83          navigatorButtons() {
 84 10x       return [
 85             { key: 'item', name: this.pkg && this.packageAvailable ? this.pkg.title : 'Packages' },
 86           ];
 87          },
 88          bannerImage() {
 89            if (this.loading) return this.$firestore('packages/m');
 90            if (!this.pkg) return this.$firestore('packages/m');
 91            if (!this.pkg.hasOwnProperty('images')) return this.$firestore('packages/m', null, this.pkg.id);
 92            if (this.pkg.images === null) return this.$firestore('packages/m', null, this.pkg.id);
 93            return this.$firestore('packages/m', this.pkg.images.filename, this.pkg.id);
 94          },
 95          packageAvailable() {
 96 17x       if (this.pkg.available_from && this.pkg.available_until) {
 97 12x         return moment().isBetween(this.pkg.available_from, this.pkg.available_until);
 98           }
 99
100  5x       if (this.pkg.available_from) {
101  2x         return moment().isAfter(this.pkg.available_from);
102           }
103
104  3x       if (this.pkg.available_until) {
105  2x         return moment().isBefore(this.pkg.available_until);
106           }
107
108  1x       return true;
109          },
110        },
111        data() {
112  6x      return {
113            loading: true,
114            pkg: false,
115          };
116        },
117        async created() {
118  6x      this.loading = true;
119 18x      const apiRequest = this.$actions.fetchPackageById(this.$route.params.id, async (pkg) => {
120            this.pkg = pkg;
121          });
122
123          await this.handleApi(apiRequest, () => {
124            this.$notify.error('Unable to load package.');
125          });
```

Code Walkthrough

Mockist or Classical

- London style === ‘Mockist’
  - Always try to mock every object
  - Try to keep all units isolated

- Detroit style === ‘Classical’
  - Use real objects where possible
  - Mock when things get complicated

- Everything is tested in isolation and less tests break with code changes
  - Tests can cover some integration checks where it make sense to do so

What to Avoid

- Coded Delays
- Testing too widely
- Focus on code coverage over what should be tested
- Using logic built values in assertions

AUTOMATED TESTING

Using logic built values in assertions

```
it( name: 'should return an object with the date formatted correctly and add a life', fn: async () => {
  const testObject = {
    lives: 4,
    timeStamp: 1620079226000
  }
  const result = formatObject(testObject);
  expect(result).toBe( expected: {
    lives: testObject.lives + 1,
    date: JSON.stringify(new Date( value: 1620079226000))
  });
});
```

```
it( name: 'should return an object with the date formatted correctly and add a life', fn: async () => {
  const testObject = {
    lives: 4,
    timeStamp: 1620079226000
  }
  const result = formatObject(testObject);
  expect(result).toBe( expected: {
    lives: 5,
    date: "2021-05-03T22:00:26.000Z"
  });
});
```

Useful Links

- Vue Test Utils - Documentation
- Jest - Documentation
- Mocks and Stubs - Martin Fowler
- BDD - Dan North

Benefits we're seeing

1. More confidence in releases
2. Better code
3. Pride in the work
4. Caught a couple of bugs
5. Prevented bugs from going out

How to start unit testing in your team

1. Implement a single test
2. Get it running on your CI pipeline

- Take ownership
- Choose a file that requires little mocking
- Don’t care about code coverage
- Don’t strive for TDD straight away
- Gradually increase expectation