Testing Your Front-End:
Dividing your time and resources

Loren Klingman - Big Nerd Ranch

Sample Repo:
https://github.com/loren138/testing-demo

About Loren

I've been developing websites for 15 years. Working in PHP, Laravel, Angular, React, Spring, and currently Vue.js and Node.js

My Websites:
https://klingman.us
https://github.com/loren138

Where We're Going

- Big Questions
- Types of Testing
- Survey Testing Approaches
- Recommendations
- Tips and Tricks

Big Questions

Why are you testing?

- Because the boss said to
- Code coverage insurance policy for future issues
  - Caveat: You can write useless test cases that create coverage
- Feel good about deploys/code refactors
- Create dependable code

What do you want to know?

- The basic happy paths work
- Every nook and cranny works
- The parts work in isolation
- The parts work together (path explosion)
- The design looks right
- The validators/computed values work

What does the future look like?

- Lots of refactoring/pivots
- Keeping the functionality the same but planning to optimize code/handle tech debt
- Adding incremental functionality
- Bug Fixes

Types of Testing

Lint/Static Type Checks

First line of defense

Advantages

- Fast
- Give you a line of the failure
- Easy to write (just config for ESLint, nothing for static typing for Java)
- Unifies your code style

Disadvantages

- Not all rules auto-fix
- Rules are turned on for the entire project at once which makes it harder to add to an existing project slowly or just for new code

Unit Testing - Units of Code

A function works as expected

```
import Contact from "@/views/Contact.vue";

describe("Contact.vue", () => {
  it("validator works", () => {
    const localThis = { formData: {} };

    expect(Contact.methods.validate.call(localThis)).toBe(false);
    expect(localThis.nameError).toBe("Name is Required!");
    expect(localThis.emailError).toBe("Email is Required!");
    expect(localThis.messageError).toBe("Message is Required!");
  });
});
```

Unit Testing - Units of Code

A validator works as expected

Advantages

- Fast
- Very Isolated - Single Function
- Easy to find source of failure
- Easy to write

Disadvantages

- Reorganization of code requires reorganization of tests

Unit Testing - Shallow Mount

Closer to Integration Testing

```
import { shallowMount } from "@vue/test-utils";
import Contact from "@/views/Contact.vue";

describe("Contact.vue", () => {
  it("shows required errors", () => {
    const wrapper = shallowMount(Contact);

    wrapper.find("form").trigger("submit.prevent");

    expect(wrapper.find("#nameError").text()).toBe("Name is Required!");
    expect(wrapper.find("#emailError").text()).toBe("Email is Required!");
    expect(wrapper.find("#messageError").text()).toBe("Message is Required!");
  });
});
```

Unit Testing - Shallow Mount

Closer to Integration Testing

Advantages

- Fast
- Isolated
- Easy to find source of failure
- Easy to write

Disadvantages

- Reorganization of code requires reorganization of tests
- Refactoring to extract shared code results in a fairly large test case change
- Fake DOM
- Can’t test navigation between pages

Unit Testing - Mount (Full Render)

(Uses the Unit Testing Framework)
Closer to Integration Testing

```
import { mount } from "@vue/test-utils";
import Contact from "@/views/Contact.vue";

describe("Contact.vue", () => {
  it("shows required errors", () => {
    const wrapper = mount(Contact);

    wrapper.find("form").trigger("submit.prevent");

    expect(wrapper.find("#nameError").text()).toBe("Name is Required!");
    expect(wrapper.find("#emailError").text()).toBe("Email is Required!");
    expect(wrapper.find("#messageError").text()).toBe("Message is Required!");
  });
});
```

Unit Testing - Mount (Full Render)

(Uses the Unit Testing Framework)
Closer to Integration Testing

Advantages

- Fast
- Somewhat Isolated
- Easy to write
- Extracting code to a sub-component may not require a test change

Disadvantages

- Finding the source of a failure requires some knowledge of the codebase
- Fake DOM
- Can’t test navigation between pages

Unit Testing - Snapshot

Tests that elements render the same as before

```
import { shallowMount } from "@vue/test-utils";
import Contact from "@/views/Contact.vue";

describe("Contact.vue", () => {
  it("shows required errors", () => {
    const wrapper = shallowMount(Contact);

    wrapper.find("form").trigger("submit.prevent");

    expect(wrapper).toMatchSnapshot();
  });
});
```

Unit Testing - Snapshot

Tests that elements render the same as before

```
exports[`Contact.vue shows required errors 1`] = `
<div class="content-box contact">
 <div>
  <h2 class="contact__title">
   Contact Loren
  </h2>
  <!---->
  <form name="contact" method="post"><input type="hidden" name="form-name" value="contact">
   <div class="sender-info">
    <div><label for="name" class="error">Your name:</label> <input id="name" type="text" name="name">
     <div id="nameError" class="error">
      Name is Required!
     </div>
    </div>
    <div><label for="email" class="error">Your email:</label> <input id="email" type="email" name="email">
     <div id="emailError" class="error">
      Email is Required!
     </div>
    </div>
   </div>
   <div class="message-wrapper"><label for="message" class="error">Message:</label> <textarea id="message" name="message"></textarea>
    <div id="messageError" class="error">
     Message is Required!
    </div>
   </div> <button id="sendBtn" type="submit">
    Submit form
   </button>
  </form>
 </div>
</div>
`;
```

Unit Testing - Snapshot

Tests that elements render the same as before

Advantages

- Fast
- Isolated
- Really Easy to write

Disadvantages

- Any HTML change requires a change in the snapshot
- Changes are reviewed in the source and in the snapshot for PRs
- Can be hard to tell expected changes from unintentional ones

End-to-End Testing - Mocked Backend

Tests the fully rendered page

```
cy.server()
cy.route('activities/*', 'fixture:activities').as('getActivities')
cy.route('messages/*', 'fixture:messages').as('getMessages')
cy.visit('http://localhost:8888/dashboard')
cy.wait(['@getActivities', '@getMessages'])
cy.get('h1').should('contain', 'Dashboard')
```

https://docs.cypress.io/guides/guides/network-requests.html#Testing-Strategies

End-to-End Testing - Mocked Backend

Tests the fully rendered page

Advantages

- Refactored code should not require any test changes
- Can test navigation between pages
- Can test full workflows (ie add to cart, checkout, and view invoice)

Disadvantages

- Requires a real browser
- Requires a running web server
- Must write a mock backend or use a toolkit like Cypress which can intercept http calls
- Slow
- Tests must be carefully written to avoid being flaky
- Doesn’t ensure backend and mock backend have the same contract

End-to-End Testing - Live Backend

Tests the fully rendered page

```
describe("The Contact Page", function() {
  it("shows errors", function() {
    cy.visit("/#contact");
    cy.contains("h2", "Contact Loren");
    cy.get("#sendBtn").click();
    cy.contains("#nameError", "Required");
    cy.contains("#emailError", "Required");
    cy.contains("#messageError", "Required");
  });
});
```

End-to-End Testing - Live Backend

Tests the fully rendered page

Advantages

- No fake backend to maintain
- Ensures the contract between front and backend is maintained
- Doubles as testing your backend
- If a flow passes, you can be very confident it will work in production.

Disadvantages

- Requires a real browser
- Requires a running web server
- Slow backend computations result in slow tests
- Must set up test data/isolate tests
- Tests must be written carefully to avoid flaky tests

Visual Regression Testing

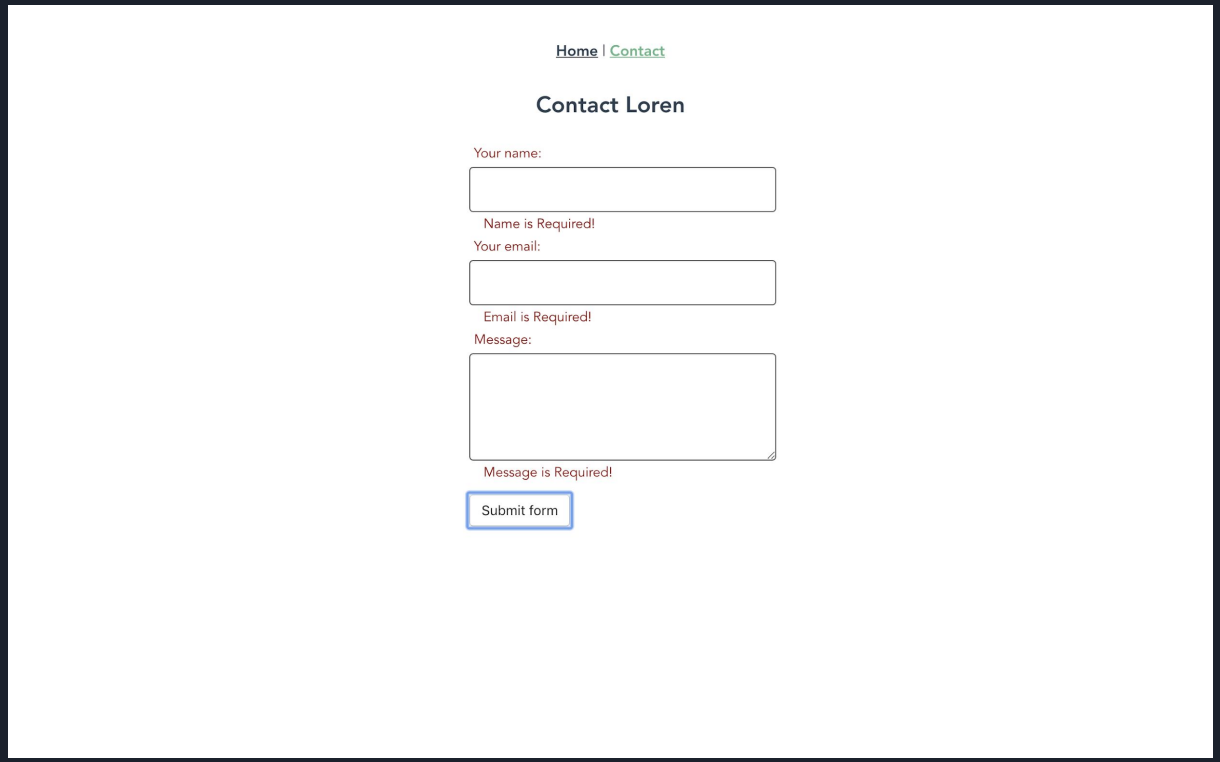
Tests the looks of the fully rendered page

(See video from my other talk for an overview)

```
Scenario("Empty Form Errors", (I) => {
  const imageName = subfolder + "contact-form-invalid.png";
  I.amOnPage("#/contact");
  // Make sure the page has loaded
  I.waitForElement("#sendBtn");
  I.click("#sendBtn");
  // Make sure error message has loaded
  I.waitForElement("#nameError");
  I.waitForText("Name is Required!");
  I.saveScreenshot(outputFolder + imageName);
  I.seeVisualDiff(imageName, { tolerance: 2, prepareBaseImage: false });
});
```

Visual Regression Testing

Tests the looks of the fully rendered page



Visual Regression Testing

Tests the looks of the fully rendered page

(When videos are posted see video from my other talk)

Advantages

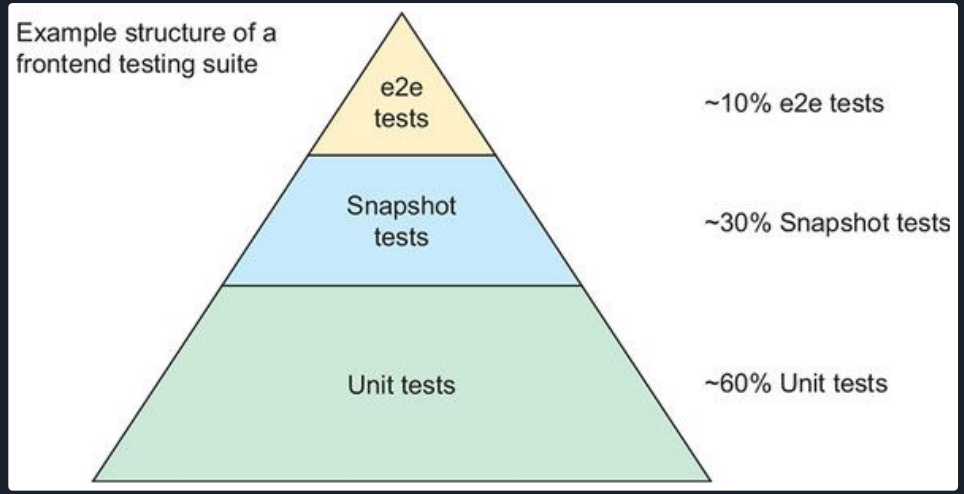
- Prevents design regressions
- Catches side effects from shared CSS changes
- Catches elements covering other elements which would still pass E2E and unit testing

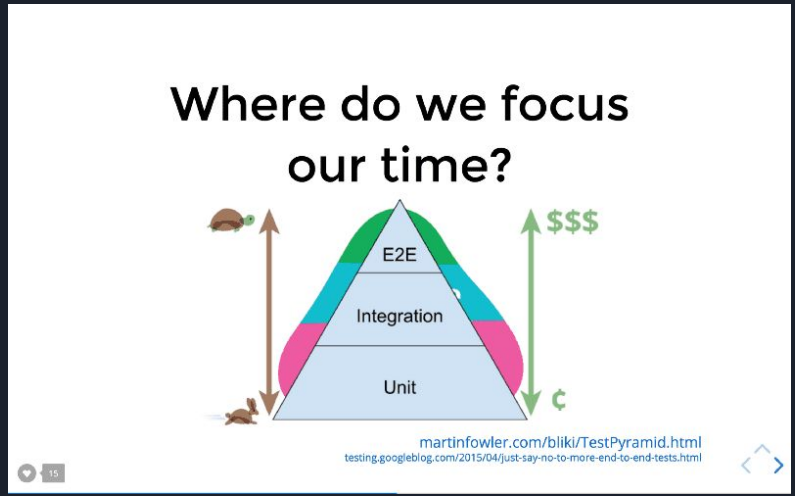
Disadvantages

- Requires a real browser
- Requires a running web server
- Slow
- Too much difference allowance to prevent flaky tests can cause tests to pass that should have failed
- 3rd party services to avoid flaky tests are expensive

Testing Approaches

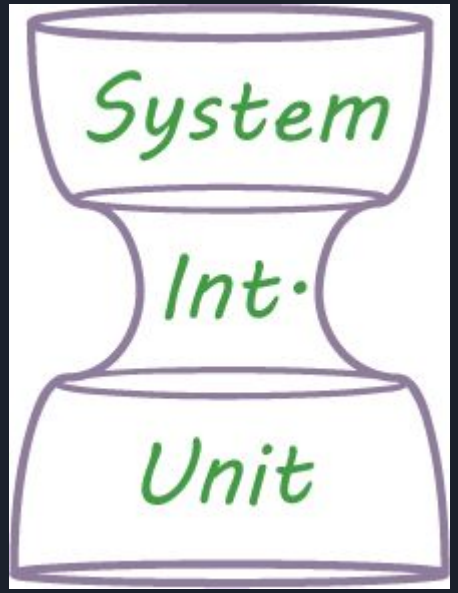
Pyramid





Testing Vue.js Applications by Edd Yerburgh
https://livebook.manning.com/#!/book/testing-vue-js-applications/chapter-1/93

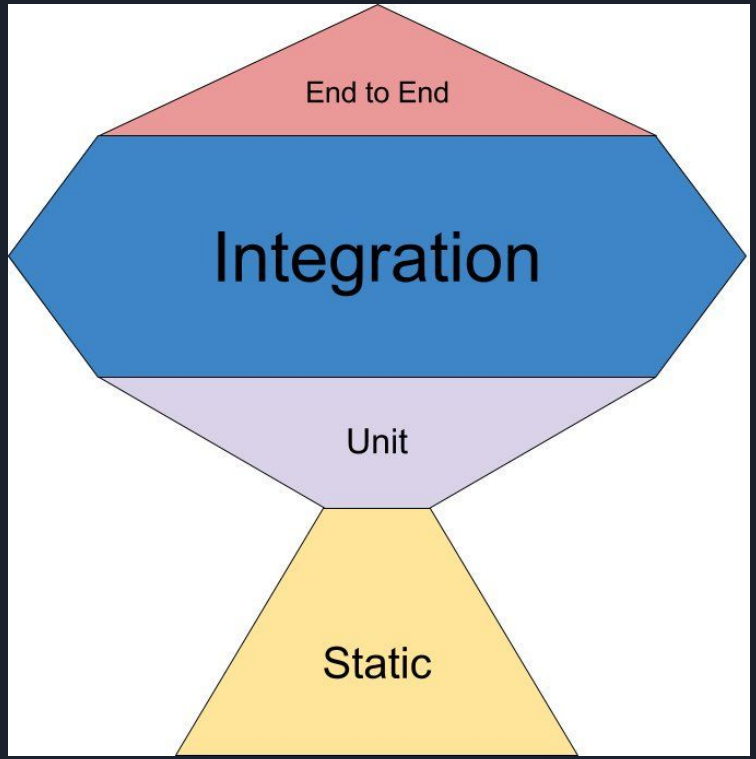
Hourglass



38% system tests
16% integration tests
45% unit tests

https://www.getautoma.com/blog/the-test-hourglass

Testing Trophy

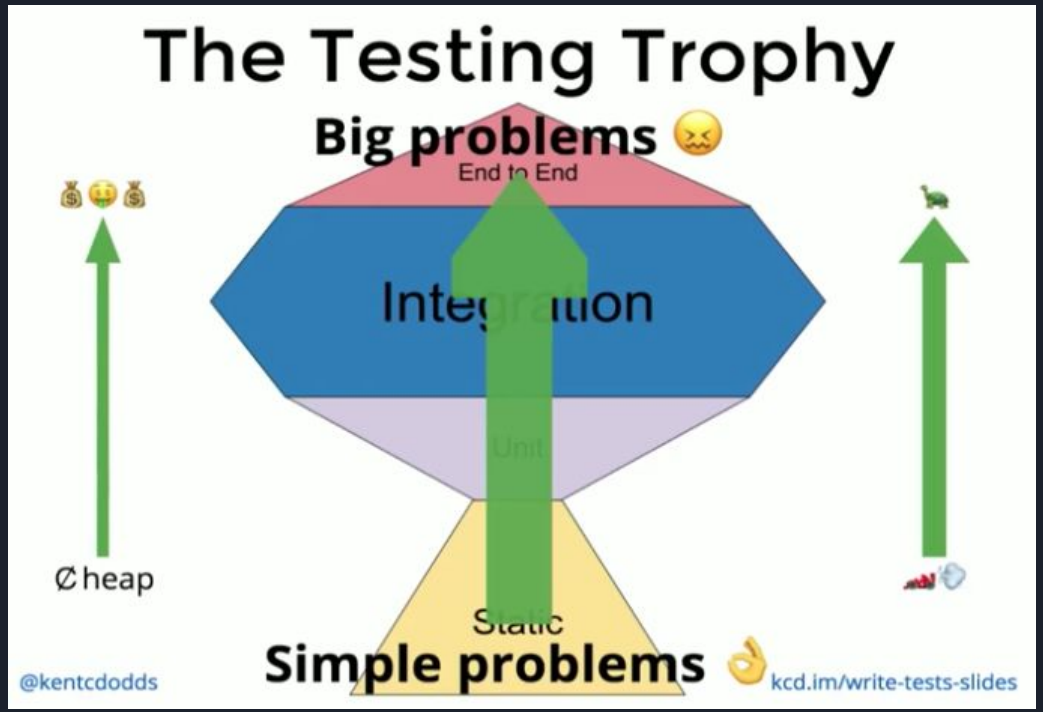


Kent C. Dodds

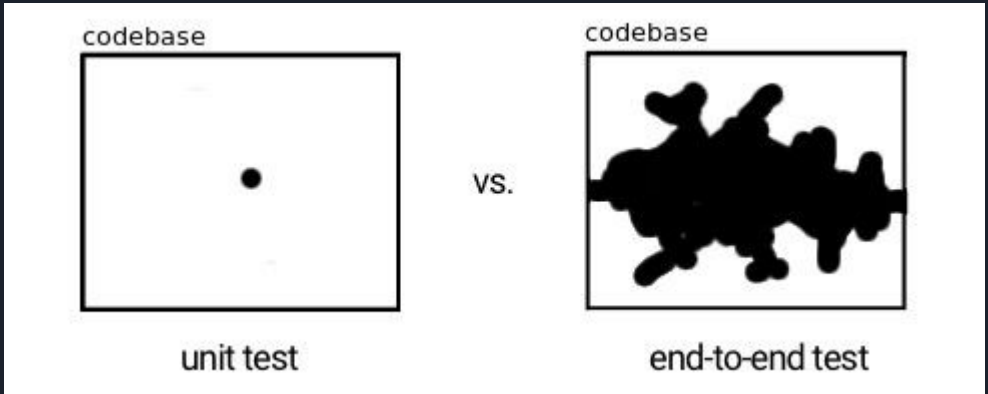
https://kentcdodds.com/blog/write-tests

(Static is help from static typing/lint rules to catch issues.)

Testing Trophy



Lean Testing:
What's the return on investment of a unit test?



Eugen Kiss:
https://blog.usejournal.com/lean-testing-or-why-unit-tests-are-worse-than-you-think-b6500139a009

Martin Sústrik:
http://250bpm.com/blog:40

My Recommendations

Early/New Project

- Lint/static type checks in a pre-commit hook and your CI pipeline
- End-to-end test only risky/business critical flows
- Manual Test Pre-Release
  - Helps you realize what you want to end-to-end test
- Unit test only complicated/critical units of code
  - Custom validators
  - Client Side Data Restructuring
  - Pricing Calculators

Growing Project

- End-to-End test anything you were manually testing
- End-to-End test full happy path flows
- End-to-End test likely error flows
- Add Visual Regression Tests as mid-term/final UI is implemented
    - UX is ready to validate screenshots and screenshot differences

Stable Project

- Add Unit/Integration (Full Render) Tests to cover flows inside each component
  - These tests would be hard to cover with end-to-end due to path explosion
- Consider lowering the threshold for functions to unit test
- Trying to achieve nearly full coverage between end-to-end tests (~70%) and other tests (~30%)

Shared Libraries

- Much more unit test focus
- Possibly no end-to-end tests if you are developing a library of functions (moment.js)
- Maybe lots of end-to-end tests testing interactions with a date picker or drag and drop
- Run the end-to-end tests on multiple browsers

Tips and Tricks

Reducing Flaky E2E Tests

- Use (short, 1s) waits anywhere you can over immediate checks
- Use longer waits when waiting for an API call to return
- Don’t use long waits everywhere as it causes failing test runs to take an extremely long time
- Be sure your backend is consistent
- Setup clean test data before each test
- Rerun failing tests once or twice before failing the build
- Rerun new/changed tests twice and require them to pass every time

Speeding up E2E Tests

- Test multiple scenarios in one block (ie full flows) to avoid browser reboots
- Be sure scenarios and/or files can be run in parallel (no dependencies on previous files/scenarios)
- Run the tests in parallel
- Speed up your backend

In Summary

- How you test now, depends on where your project is going
- End-to-end tests give you the most confidence in your project per test case and are the least likely to break due to refactors
- Unit tests are the easiest to fix but require more maintenance during code refactoring
- Lint rules/static code checks do the most to prevent bugs from ever being compiled/committed (if you check that pre-commit)

Big Nerd Ranch

We design and build exceptional apps and teach others to do the same.

https://www.bignerdranch.com/

Questions?

https://klingman.us
https://www.bignerdranch.com

Slides:
http://files.klingman.us/testing-front-end.pdf