Cheatsheet - My Rails Testing Roadmap

When to TDD

1. If you're **unsure how to implement something**, first develop working code without tests. Then throw this away, and restart development with TDD.
2. If you're **changing the behavior of an existing feature**, then TDD right away (assuming the code has tests to begin with)
3. If the **code doesn't have tests**, then introduce test coverage before changing the code.
4. If you're **fixing a bug**, write a test to prove the failure before going in and fixing it.

The Rails Testing Roadmap

Your Development Environment

- Install & Setup `simplecov`
- Set up a key command to quickly run the test you're working on

Acceptance Tests

Tests involving the browser are typically slow. At the same time, without acceptance tests, you can't be 100% sure that all the pieces of your system work as expected. So a happy medium is this:

- Write an acceptance test for the "happy path"
- Save testing the edge cases for the controller (or other objects) tests
- Rely on CSS classes & ids for your test assertions. This ensures your tests don't have to change too often. For example, favor something like `expect(page).to have_css('#greeting')` over `expect(page).to have_content('hello')`
- Use a headless browser

Controllers

At a minimum, here's what to test for in your controllers:

- Check that the right view template is rendered
- Check redirects
- Ensure flash messages are set

- Ensure instance variables are set as expected
- If applicable (and simple enough to do so), check that emails are sent and/or the database is updated

Ideally, I try to stay clear of putting business logic in my controllers. **If my controller spec has tests for more than the above, I start to consider extracting the relevant functionality into service objects.**

I also get two extra benefits with my controller tests:

1. The tests won't pass if the route is not defined.
2. By using `render_views`, I ensure that any view errors are caught.

Because of this, I don't write view and route tests.

Models

My model specs typically only contain tests for the following:

- Columns (something like `it { is_expected.to have_db_column(:name).of_type(:string) }`)
- Scopes
- Validations
- Associations
- Delegations

If your spec contains tests for more than the above, you might want to consider if extracting this functionality would make your code easier to deal with.

POROs and Other Custom Rails Objects

Most business logic in a decent sized Rails app will (or should) reside outside the MVC structure. Subsequently, the lion's share of your tests will run on these objects:

- Validators
- Serializers
- Workers
- Helpers
- Mailers
- POROs (things like Form Objects, Value Objects etc)

As far as you can, aim to isolate your tests for these objects from other parts of the system. Though it won't always be possible because of the way Rails is designed, sustained effort in this area is bound to pay off in more ways than one:

1. Your tests will run faster. This will make your development process more enjoyable.
2. Difficulty isolating an object under test (for example, with deeply nested stubs) can be a sign that your object is doing too much, or knows too much about other objects in the system. When this happens to me, I often reconsider my design and think of ways I can reduce my object's responsibilities.

Javascript

- Use Jasmine to write your javascript specs
- The higher your test coverage, the more confidence you will have in changing your code when the time comes to do so.

External Services

- Stub external requests with VCR and WebMock
- **Caution:** Keep an eye out for changes to the API

Test Speed: Rules of Thumb

Martin Fowler says:

...your test suites should run fast enough that you're not discouraged from running them frequently enough. And frequently enough is so that when they detect a bug ***there's a sufficiently small amount of work to look through that you can find it quickly.***

According to Kent Beck: *Your test suite as a whole should take* ***no longer than 10 minutes to run*****.**