Understanding the Hydra testing stack

- I want to understand Hydra better
- I want to know more about Hydra's Architecture (i.e. the Hydra stack)
- How is Hydra tested?
- How do I create Hydra-framework-friendly fixtures

Resources

- Hierarchy of Promises
- ProjectHydra on Github
- ProjectHydra Labs on Github

Dive into a Git Repository

Look at ActiveFedora's Badges

- Build Status - The gem is tested via Travis-CI.org
- Gem Version - The gem is published to and distributed via Rubygems.org
- Dependencies - The gem's dependencies are monitored by Gemnasium
- Code Coverage - The gem's test coverage is gathered and reported by Coveralls.io

These are all free services (for open source projects) that Hydra leverages.

There are other resources available for open source projects:

- Code Coverage and potential Code Smells can be gathered by Codeclimate.com.
- Documentation suggestions can be gathered by Inch-CI.org
- Documentation is gathered at Rubydoc.info
- Monitor for style guide violations via HoundCI.com

For most Hydra projects, look in the .travis.yml file at the root of the project. If there is a script: key, that is how TravisCI will build and test the gem and how you might go about doing it. In many cases the default rake command will build and test the gem.

Reviewing Tests

- Does this need to be tested?
- Are they sufficient?
- Are they testing the right thing?
- Are they in the right spot / properly organized?
- Are they maintainable?

There are two fundamental questions to when thinking about tests.

- How will I verify I have done the right thing?
- How will I verify I have made it the right way?

Before you begin writing production code, ask those two questions and formulate your answers. Those answers are your test plan. In many cases you can automate your test plan.

Done the Right Thing

These are your acceptance tests. You are testing the system from the outside in. They will be slow as they must load the whole system.

Aim for only a few of these; Perhaps even outside of your applications code. Fitnesse is a great option for acceptance tests.

Made the Right Way

These are your unit tests. You are testing from within the system. They should be fast so you have a quick feedback loop. You will have many of these.

Treat them as documentation that expresses intent.

Rules of Engagement

Establish your allowable threshold for Code Coverage. On my projects, I require 100% code coverage under unit tests.

For each file in app/ and lib/, there will exist a corresponding file in spec/. In Atom (via rails-rspec) and Sublime (via Rails Go to Spec) I map Cmd+. to jump between live and spec files. This command creates the file if one does not exist.

Unit tests should test a single class and how it collaborates with other objects. Collaborators should be injected into either the method you are testing or as part of object instantiation. You will begin to tease apart the interface of various objects.

Use a code style enforcement tool such as Rubocop and HoundCI. Your team should agree on the style guide and stick to it. Any violations should be reviewed and accepted.

By using automated enforcement of style guides, your code review process can focus on what is being added or changed; And not indentation, file size concerns, unused variables, etc.

At Notre Dame, we use the Commitment gem to say the code is broken if:

- A test fails (via RSpec)
- Code coverage is not 100% (via SimpleCov)
- A vulnerability is detected (via Brakeman)
- SCSS does not pass a linter (via scss-lint)
- JS does not pass a linter (via jshintrb)
- A style violation is detected (via Rubocop and our .hound.yml config)

I go a step further run a pre-push git hook that runs rake for our applications. On a failure, git will not allow me to push the changes. (I can get around that with git push --no-verify)

Reviewing Pull Requests

You are reviewing both the commit message and the code that has changed. Take a moment and reorient yourself.

- Does the commit message convey why something was done?
- Can you read the code and understand how its working?
- Are there comments saying why a possibly confusing choice was made?
- Does each spec's description jive with the code of the spec?
- Does it look like the tests are testing one thing?
- Are the tests "creating the whole world" just to verify something rather small?
- What questions come to mind when you read the changes?

Ask these questions of yourself and engage in the pull request process.

Tips and Tricks for Testing

- Maintainability
- RSpec syntax
- Speed kills
- Mocking & Stubbing
- Capybara v. RSpec v. Cucumber
- Composition over Inheritance
- Narrow Interfaces

Maintainable code must be easy to refactor and extend. Confident refactoring of code requires confidence in your test suite and quick feedback. Therefore your test suite must run fast and have good coverage.

Hint: If an object is hard to test, break it apart.

With RSpec 3, I make use of the double method. I use doubles to be a test proxy for a more complicated collaborator. It is possible (and I do) go crazy with doubles. In doing so I begin thinking about the interfaces of the collaborators.

In using these methods I'm creating many small underlying classes that can be reused and repurposed. At the same time, I'm creating outward facing classes that expose an API for public interface.

Throughout the unit level refactoring, I often won't run the Acceptance tests; They are too slow. However they are useful as they work through the system from the outside.

Acceptance tests that I run will leverage Capybara to handle the web requests. I will also take the time to make Page objects viaSitePrism. Page objects allow.

I don't write Cucumber tests because in my experience the audience and those writing the Cucumber tests are the same people. And there are better tools that require far less overhead.

Why Do My Tests Take Forever to Get Started?

First, many Hydra components require that Jetty is started (to make Solr and Fedora available). Second, your tests are likely booting up the entire Rails ecosystem and all of its dependencies.

There are some things you can do to help out.

At the top of your spec files you see the ubiquitous `require "spec_helper"`. If you don't see it, check `.rspec` in the root of your project. That directive says to load the spec_helper file.

But you can create custom require files and decide which one to use in your tests. In Cogitate, I have the following:

- spec_fast_helper.rb
- spec_active_record_helper.rb
- rails_helper.rb

Helper	Relative Load Time
spec_fast_helper.rb	x1
spec_active_record_helper.rb	x2
rails_helper.rb	x5 or more

This was tested in Cogate via: time rspec -r spec/<helper_filename>

spec/lib/cogitate/client_spec.rb

Examples

Below are a few RSpec examples. First a simple scenario that highlights dependency injection and mocks.

```
class HelloWorld
  def print(buffer: default_buffer)
    buffer.puts "Hello World"
  end
  private
  def default_buffer
    $stdout
  end
end

require 'rspec'
# I find it very helpful for testing dependency injection
require 'rspec/its'

RSpec.describe HelloWorld do
  let(:buffer) { double(puts: true) }
  subject { described_class.new }
  it 'will output to the given buffer' do
    subject.print(buffer: buffer)
    expect(buffer).to have_received(:puts).with("Hello World")
  end

  it 'will output to $stdout if none is given' do
    expect($stdout).to receive(:puts).with("Hello World")
    subject.print
  end
end
```

A more complicated scenario that highlights dependency injection with ActiveRecord objects.

```
class CreateWork
  def initialize(attributes:, validator: default_validator, **collaborators)
    self.attributes = attributes
    self.validator = validator
    self.persister = collaborators.fetch(:persister) { default_persister }
    self.notifier = collaborators.fetch(:notifier) { default_notifier }
  end
  def call
    return false unless validator.call(attributes)
    work = persister.call(attributes)
    notifier.call(work)
    work
  end
  private
  attr_accessor :attributes, :persister, :notifier, :validator
  def default_validator
    Work::Validator.method(:valid?)
  end
  def default_persister
    Work.method(:create!)
  end
  def default_notifier
    Notifier.method(:deliver_work_created_message!)
  end
end
```

```
require 'rspec'
require 'rspec/its' # https://github.com/rspec/rspec-its
RSpec.describe CreateWork do
 let(:attributes) { { name: 'A Book' } }
 let(:persister) { double('Persister', call: work) }
 let(:validator) { double('Validator', call: false) }
 let(:notifier) { double('Notifier', call: true) }
 let(:work) { double('The Work') }
 subject do
  described_class.new(
   attributes: attributes, persister: persister,
   validator: validator, notifier: notifier
  )
 end

 its(:default_validator) { should respond_to(:call) }
 its(:default_persister) { should respond_to(:call) }
 its(:default_notifier) { should respond_to(:call) }

 context 'with invalid data' do
  before { allow(validator).to receive(:call).with(attributes).and_return(false) }
  it 'will return false' do
   subject.call
   expect(validator).to have_received(:call).with(attributes)
  end
  it 'will not attempt to persist the data' do
   subject.call
   expect(persister).to_not have_received(:call)
  end
  it 'will not attempt to notify' do
   subject.call
   expect(notifier).to_not have_received(:call)
  end
```

```
  end

  context 'with valid data' do
    before { allow(validator).to receive(:call).with(attributes).and_return(true) }
    it 'will call the validator' do
      subject.call
      expect(validator).to have_received(:call).with(attributes)
    end
    it 'will call the persister' do
      subject.call
      expect(persister).to have_received(:call).with(attributes)
    end
    it 'will call the notifier' do
      subject.call
      expect(notifier).to have_received(:call).with(work)
    end
    it 'will return the work' do
      expect(subject.call).to eq(work)
    end
  end
end
```