Testing the Impossible

Joe Ferris
factory_girl, shoulda, hoptoad, copycopter, capybara-webkit
Almost called the talk “Testing Safari”
 Didn’t want people to think of the web browser
HTTP
Threads
Forking
Logging
Caching
Rails Integration
Write Placeholders

Using the lovely Ruby/Rails I18n API
Viewing locally uploads to Copycopter
Uploaded content is editable, so don’t worry about exact copy
Edited copy is visible in production within 5 minutes
require "copycopter_client"
CopycopterClient.configure do |
  config|
  config.api_key = 'magic numbers'
end
github.com/thoughtbot/copycopter_client
- Had to be automatic
- Had to be easy to setup
- Can’t affect production performance
100% C0 Coverage
100% Test-driven

We’ve never written a line of code in the client without writing a failing test first
I haven’t killed myself yet

Testing shouldn’t be painful
I’m going to go through some of the pieces of the client that I found difficult or tedious and go over the solutions that made them more enjoyable to test.
Don’t Mix Concerns
Application mostly talks to I18nBackend. Originally, the backend got all of its information from a Sync class. Responsible for polling in a background thread, managing access between threads, fetching content, and handling process changes (forks, exits, etc).
class Sync
  def initialize; end
  def start; end
  def stop; end
  def keys; end
  def wait_for_download; end
  def flush; end
  def download; end

  private

  def poll; end
  def sync; end
  def with_queued_changes; end
  def lock; end
  def spawner; end
  def passenger_spawner; end
  def unicorn_spawner; end
  def register_job_hooks; end
  def perform; end
  def register_spawn_hooks; end
  def worker_loop; end
end

Large number of private methods, crappy verb name
Long test runs ruin the tight feedback loop of TDD
Finished in 43.81 seconds
26 examples, 0 failures

Long test runs ruin the tight feedback loop of TDD
How do you decide which pieces are separate “concerns?”
Single Responsibility Principle

“Every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class.”

Still kind of vague
**Single Responsibility Principle**

Every object should have a single reason to change.

Robert Martin: only one “reason to change”
Sync has at least three: changing the cache API, changing polling, or changing process guards
class Sync
  def initialize; end
  def start; end
  def stop; end
  def keys; end
  def wait_for_download; end
  def flush; end
  def download; end

private

  def poll; end
  def sync; end
  def with_queued_changes; end
  def lock; end
  def spawner; end
  def passenger_spawner; end
  def unicorn_spawner; end
  def register_job_hooks; end
  def perform; end
  def register_spawn_hooks; end
  def worker_loop; end
end
class Sync
  def initialize; end
  def start; end
  def stop; end
  def keys; end
  def wait_for_download; end
  def flush; end
  def download; end

private
  def poll; end
  def sync; end
  def with_queued_changes; end
  def lock; end
  def spawner; end
  def passenger_spawner; end
  def unicorn_spawner; end
  def register_job_hooks; end
  def perform; end
  def register_spawn_hooks; end
  def worker_loop; end
end
class Sync
  def initialize; end
  def start; end
  def stop; end
  def keys; end
  def wait_for_download; end
  def flush; end
  def download; end

  private

  def poll; end
  def sync; end
  def with_queued_changes; end
  def lock; end
  def spawner; end
  def passenger_spawner; end
  def unicorn_spawner; end
  def register_job_hooks; end
  def perform; end
  def register_spawn_hooks; end
  def worker_loop; end
end
class Sync
  def initialize; end
  def start; end
  def stop; end
  def keys; end
  def wait_for_download; end
  def flush; end
  def download; end

private

  def poll; end
  def sync; end
  def with_queued_changes; end
  def lock; end
  def spawner; end
  def passenger_spawner; end
  def unicorn_spawner; end
  def register_job_hooks; end
  def perform; end
  def register_spawn_hooks; end
  def worker_loop; end
end
CopycopterClient::Poller
   it polls after being started
   it doesn't poll before being started
   stops polling when stopped
   stops polling with an invalid api key
   logs an error if the background thread can't start
   flushes the log when polling
   starts from the top-level constant

Finished in 9.02 seconds
7 examples, 0 failures
CopycopterClient::Cache
  provides access to downloaded data
  doesn't upload without changes
  uploads changes when flushed
  downloads changes
  downloads and uploads when synced
  handles connection errors when flushing
  handles connection errors when downloading
  blocks until the first download is complete
  doesn't block if the first download fails
  doesn't block before downloading
  doesn't return blank copy
  flushes from the top level
  given locked mutex
    synchronizes read access to keys between threads
    synchronizes read access to the key list between threads
    synchronizes write access to keys between threads

Finished in 4.16 seconds
15 examples, 0 failures

Extracted most functionality out of sync, renamed to cache
CopycopterClient::ProcessGuard
starts polling from a worker process
registers passenger hooks from the passenger master
starts polling from a passenger worker
registers unicorn hooks from the unicorn master
starts polling from a unicorn worker
flushes when the process terminates
flushes after running a resque job

Finished in 0.07981 seconds
7 examples, 0 failures
<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>43.81 seconds, 26 examples</td>
<td>4.16 seconds, 15 examples</td>
</tr>
<tr>
<td></td>
<td>9.02 seconds, 7 examples</td>
</tr>
<tr>
<td></td>
<td>0.08 seconds, 7 examples</td>
</tr>
<tr>
<td>Total:</td>
<td></td>
</tr>
<tr>
<td>13.26 seconds, 29 examples</td>
<td></td>
</tr>
</tbody>
</table>

A few more examples, but 3 times as fast
Specs are also clearer and less brittle
class Poller
  def initialize; end
  def start; end
  def stop; end

private

  def poll; end
  def delay; end
end
class Cache
  def initialize; end
  def keys; end
  def wait_for_download; end
  def flush; end
  def download; end
  def sync; end

  private

  def with_queued_changes; end
  end
  def lock; end
  def pending; end
end

Three classes this size are more manageable
class ProcessGuard
  def initialize; end
  def start; end

  private

  def start_polling; end
  def spawner; end
  def passenger_spawner; end
  def unicorn_spawner; end
  def register_spawn_hooks; end
  def register_passenger_hook; end
  end
  def register_unicorn_hook; end
  def register_exit_hooks; end
  def register_job_hooks; end
end

Still lots of private methods; maybe room to go (note prefixes)
Inherit Less
module RequestSync
  include ActiveSupport::Concern

  included do
    before_filter :download_from_copycopter
    after_filter :flush_to_copycopter
  end

module InstanceMethods
  def download_from_copycopter
    CopycopterClient.download
  end

  def flush_to_copycopter
    CopycopterClient.flush
  end
end
end
end

In development, Copycopter syncs before/after every request
How do you test this crap
Rails specific, not really easy to follow
Prefer composition to inheritance

Mixins are inheritance; avoid them where possible
Can’t be used or tested without the class you’re mixing into
irb> module Crap; end
=> nil
irb> class Facebook; include Crap; end
=> Facebook
irb> Facebook.ancestors
=> [Facebook, Crap, Object, Kernel]
module CopycopterClient

class RequestSync

  def initialize(app, options)
    @app = app
    @cache = options[:cache]
  end

  def call(env)
    @cache.download
    response = @app.call(env)
    @cache.flush
    response
  end
end
end
describe `RequestSync` do
  let(:cache) { {} }  
  let(:response) { 'response' }  
  let(:env) { 'env' }  
  let(:app) { stub('app', :call => response) }  
  before { cache.stubs(flush => nil, download => nil) }  
  subject { RequestSync.new(app, :cache => cache) }  

  it "invokes the upstream app" do 
    result = subject.call(env) 
    app.should have_received(:call).with(env) 
    result.should == response 
  end

  it "flushes defaults" do 
    subject.call(env) 
    cache.should have_received(flush) 
  end

  it "downloads new copy" do 
    subject.call(env) 
    cache.should have_received(download) 
  end
end

Using mocks, but with trusted, stable interfaces
Don’t Depend on Concretions
class Poller
  def initialize(options)
    @client = Client.new
    @cache = Cache.new(client)
    # ...
    end
  end
end

Instantiates classes in low level components
let(:client) { FakeClient.new }
let(:cache) { Cache.new }

def build_poller(config = {})
  Client.stubs(:new => client)
  Cache.stubs(:new => cache)
  poller = Poller.new(default_config.update(config))
end
Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend upon details. Details should depend upon abstractions.
Dependency Injection

“separates behavior from dependency resolution, thus decoupling highly dependent components”

(Wikipedia)
class Poller
  def initialize(options)
    @client = Client.new
    @cache = Cache.new(client)
    # ...
  end
  # ...
end  
end
class Poller
  def initialize(cache, options)
    @cache = cache
    # ...
  end
  # ...
  # ...
end
let(:client) { FakeClient.new }
let(:cache) { Cache.new(client, 
    :logger => FakeLogger.new) }

def build_poller(config = {})
    poller = Poller.new(cache, 
        default_config.update(config))
end
Dependency graph has direction
Most components depend on only one or two others
No component builds another except for the main/container
Avoid class methods
before do
  reset_config
end

def set_public_env
  CopycopterClient.configure do |config|
    config.environment_name = 'production'
  end
end
Prefer Instance Methods

This may mean you create more classes or pass more arguments
project = add_project
project.update(
    'draft' => {
        'key.one' => "expected one",
        'key.two' => "expected two"
    }
)
client = build_client(:api_key => project.api_key, :public => false)
blurbs = nil

client.download { |yielded| blurbs = yielded }

blurbs.should == {
    'key.one' => 'expected one',
    'key.two' => 'expected two'
}
Escape from globals – class methods are little more than global methods with a namespace
Avoid Soup
module CopycopterClient

class << self
  def remote_lookup_disabled; end
  def disable_remote_lookup; end
  def enable_remote_lookup; end
  def report_ready; end
  def report_environment_info; end
  def report_response_body; end
  def environment_info; end
  def writeVerboseLog; end
  def logger; end
  def configure; end
  def copy_for; end

  private

  def fetch; end
  def perform_caching; end
  def edit_link; end
  def url; end
end
end
module CopycopterClient
  class << self
    # lots of methods
  end
end
end
it "returns the default content for a key that doesn't exist" do
  set_development_env
  reset_webmock
  stub_request(:post, /.*copycopter.*/).
    to_return(:status => 200,
      :body   => "Posted to test.key")
  stub_request(:get, /copycopter.*/).
    to_return(:status => 404,
      :body   => "Blurb not found: test.key")
  CopycopterClient.copy_for("test.key", "default content").
    should == "default content"
end
Once you have composed components, you have seams
it "polls after being started" do
  poller = build_poller
  poller.start

  client['test.key'] = 'value'
  wait_for_next_sync

  cache['test.key'].should == 'value'
end
Testing Design

Techniques so far are application design techniques
Test Scope

- Unit, Component, Isolation
- Functional, Acceptance, Integration, Full Stack
Scenario: copycopter detects updates to copy

Given the "abc123" project has the following blurbs:

<table>
<thead>
<tr>
<th>key</th>
<th>draft content</th>
</tr>
</thead>
<tbody>
<tr>
<td>en.users.index.controller-test</td>
<td>Old content</td>
</tr>
</tbody>
</table>

When I write to "app/controllers/users_controller.rb" with:

```ruby
class UsersController < ActionController::Base
  def index
    @text = t("users.index.controller-test", :default => "default")
  end
end
```

When I route the "users" resource
And I write to "app/views/users/index.html.erb" with:

```erb
<%= @text %>
```

When I start the application
And I visit /users/
Then the response should contain "Old content"

When the the following blurbs are updated in the "abc123" project:

<table>
<thead>
<tr>
<th>key</th>
<th>draft content</th>
</tr>
</thead>
<tbody>
<tr>
<td>en.users.index.controller-test</td>
<td>New content</td>
</tr>
</tbody>
</table>

And I visit /users/
Then the response should contain "New content"

Use full stack tests for each major functional path
Scenario: copycopter detects updates to copy with scope

Given the "abc123" project has the following blurbs:

<table>
<thead>
<tr>
<th>key</th>
<th>draft content</th>
</tr>
</thead>
<tbody>
<tr>
<td>en.users.index.controller-test</td>
<td>Old content</td>
</tr>
</tbody>
</table>

When I write to "app/controllers/users_controller.rb" with:

```ruby
class UsersController < ActionController::Base
  def index
    @text = I18n.t("controller-test", :default => "default", :scope => [:users, :index])
  end
end
```

When I route the "users" resource

And I write to "app/views/users/index.html.erb" with:

```erb
<%= @text %>
```

When I start the application

And I visit /users/

Then the response should contain "Old content"

When the the following blurbs are updated in the "abc123" project:

<table>
<thead>
<tr>
<th>key</th>
<th>draft content</th>
</tr>
</thead>
<tbody>
<tr>
<td>en.users.index.controller-test</td>
<td>New content</td>
</tr>
</tbody>
</table>

And I visit /users/

Then the response should contain "New content"
it "queues missing keys with scope" do
  default = 'default value'
  result =
    subject.translate('en', 'key', :default => default,
                      :scope => ['test'])
  result.should == default
  cache['en.test.key'].should == default
end

Use specs to cover details, edge cases, and minor alternate paths
Restrict Fixture Scope
before do
  cache.update({
    'en.key.one' => 'Expected',
    'fr.key.one' => 'French',
    # More examples
  })
end

# Fifty lines later...

it "looks up an array of defaults" do
  backend.translate('en', 'key.three', :
    :default => ['key.two', 'key.one']).
    should == 'Expected'
end

You need to keep jumping back and forth, changing the fixture data can unexpectedly create failures (or cause tests to pass when they shouldn’t); fixture data becomes unmaintainable
Which of the following is the largest?

A: A Peanut
B: An Elephant
C: The Moon
D: A Kettle
Use fresh fixtures
it "looks up an array of defaults" do
  cache['en.key.one'] = "Expected"
  backend = build_backend
  backend.translate('en', 'key.three',
                   :default => [:"key.two", :"key.one"]).
             should == 'Expected'
end

Create what you need when you need it; don’t let unrelated data exist for a test
Avoid Test Helpers
it "logs that it performed a download" do
  logger = stub('logger')
  logger.expects(:info).with("Downloaded translations")
  client = build_client_with_project(:logger => logger)
  client.download
end

Start with a simple stub
it "logs that it performed a download" do
  logger = stub('logger')
  logger.expects(:info).with("Downloaded translations")
  logger.stubs(:error)
  logger.stubs(:warning)
  logger.stubs(:debug)
  logger.stubs(:flush)
  client = build_client_with_project(:logger => logger)
  client.download
end

Need to stub out full API
include LoggerStubs

it "logs that it performed a download" do
  logger = stub_logger
  logger.expects(:info).with("Downloaded translations")
  client = build_client_with_project(:logger => logger)
  client.download
end

Reusable stubs, but using mixins and test reads a little awkwardly
1) CopycopterClient::Cache handles connection errors when downloading
   Failure/Error: logger.expects(:error).with(failure)
   not all expectations were satisfied
   unsatisfied expectations:
   - expected exactly once, not yet invoked: #<Mock:logger>.error('server is napping')
   satisfied expectations:
   - allowed any number of times, not yet invoked: #<Mock:logger>.error(any_parameters)
   - allowed any number of times, already invoked once:
     #<Mock:logger>.info(any_parameters)

Calling wrong method, test output isn't useful
logger.error("Downloaded translations")
1) CopycopterClient::Cache handles connection errors when downloading
   Failure/Error: logger.expects(:error).with(failure)
   not all expectations were satisfied
   unsatisfied expectations:
   - expected exactly once, not yet invoked: #<Mock:logger>.error('server is napping')
   satisfied expectations:
   - allowed any number of times, not yet invoked: #<Mock:logger>.error(any_parameters)
   - allowed any number of times, already invoked once:
     #<Mock:logger>.info(any_parameters)

Same failure
Test Doubles

- Stubs
-Mocks
- Fakes
- Proxies

Gerard Meszaros, xUnit Patterns
Write Fakes
class FakeLogger
  def initialize
    # ...
  end

  def info(message = nil, &block)
    log(:info, message, &block)
  end

  # ...

  def log(severity, message = nil, &block)
    message ||= block.call
    @entries[severity] << message
  end

  def has_entry?(level, expected_entry)
    @entries[level].any? { |actual_entry|
      actual_entry.include?(expected_entry)
    }
  end
end

Stricter, easier to be DRY than with stubs/mocks
Allow more natural verification than mocks
Easier to distribute
it "logs that it performed a download" do
  logger = FakeLogger.new
  client = build_client_with_project(:logger => logger)
  client.download { |ignore| }
  logger.should have_entry(:info, "Downloaded translations")
end
RSpec::Matchers.define :have_entry do [severity, entry]
  match do |logger|
    @logger = logger
    logger.has_entry?(severity, entry)
  end

  failure_message_for_should do
    "Expected #{severity}#{entry.inspect}: got entries:
    #{entries}" end

  failure_message_for_should_not do
    "Unexpected #{severity}#{entry.inspect}: got entries:
    #{entries}" end

  def entries
    lines = @logger.entries.inject([]) do |result, (severity, entries)|
      if entries.empty?
        result
      else
        result << "#{severity}:#{entries.join(\"\n\")}" end
    end
    lines.join(\n        \n"
end
end

Good matchers are a bit of work
CopycopterClient::Cache handles connection errors when downloading
Failure/Error: logger.should have_entry(error, failure)
Expected error("server is napping"); got entries:

  info:
  server is napping

Output is worth it; if you have to retest use of a component several times, matchers/fakes pay for themselves
Rails
Stay Generic

- Use middleware when possible
- Avoid monkey patching
- Keep business logic outside engine parts
- Use Railties only for configuration-initialization

Don’t reference Rails components directly, ie logger
module CopycopterClient
  class Railtie < ::Rails::Railtie
    initializer :initialize_copycopter_rails do
      CopycopterClient.configure(false) do |config|
        config.environment_name = ::Rails.env
        config.logger           = ::Rails.logger
        config.framework        = "Rails: #{::Rails::VERSION::STRING}"
        config.middleware       = ::Rails.configuration.middleware
      end
    end
    rake_tasks do
      load "tasks/copycopter_client_tasks.rake"
    end
  end
end
Integration Test

- Generate a Rails application
- Install your gem
- Exercise major paths
- Boot a server process if you need to
- Test across several versions

Cucumber, Aruba, Capybara, Appraisal
Given I have a copycopter project with an api key of "abc123"
When I generate a rails application
And I configure the copycopter client with api key "abc123"
When /^I configure the copycopter client with api key "((\^)\^)"$/ do |api_key|
write_file("config/initializers/copycopter.rb", <<-RUBY)
  CopycopterClient.configure do |config|
    config.api_key = "#{api_key}"
    config.polling_delay = 1
    config.host = 'localhost'
    config.secure = false
    config.port = #{FakeCopycopterApp.port}
  end
RUBY
append_to_file("Gemfile", <<-RUBY)
  gem "copycopter_client", :path => PROJECT_ROOT
RUBY
end
Isolation Test

- Unit test components without Rails
- Don’t even load ActiveSupport
- Test Rails-specific components separately
- Use fakes

RSpec, Mocha, Bourne, Fakes
Don’t Be Afraid

Don’t back down, don’t test something “later,” don’t understand something “later”
- Separation of concerns
- Prefer composition over inheritance
- Create seams
- Balance integration and isolation tests
- Write fakes