Rails Testing

TDD BASICS

Red-Green Loop

1. Write the smallest possible test case that matches what we need to program.
2. Run the test and watch it fail. This gets you into thinking how to write only the code that makes it pass.
3. Write some code with the goal of making the test pass.
4. Run your test suite. Repeat steps 3 and 4 until all tests pass.
5. Go back and refactor your new code, making it as simple and clear as possible while keeping the test suite green.

4-Phase Test

1. **Setup**: create the necessary preconditions for a test (usually automated by test framework)
2. **Exercise**: run the code you're trying to test
3. **Verify**: make sure that you got the expected result
4. **Teardown**: clean up after your test so that it doesn't interact (usually automated by test framework)

Specs vs Tests: specs are tests but meant to be written in a more natural language (baseline for RSpec and supported by Mintiest)

Database Cleaning Strategies

- Understanding database cleaning strategies in tests
- Difference Between Truncation, Transaction, and Deletion Database Strategies

———————————————————— ——————————————————— —— ——————————————————— ——————————————

RAILS DEFAULTS

Rails Default Testing Suite

- **Minitest** unit testing framework (the *test* directory and contained files are auto generated upon creation of a new Rails app)
  - It's all RUBY: test files are Ruby classes and test cases are Ruby methods
- **Fixtures** are a testing construct used as a way to organize data that you want to test against
- **Test::Unit** module was a predecessor to Minitest

Fixtures

- Represent ActiveRecord's built-in way of handling test data. They are well-integrated with Rails out-of-the-box, and they are a bit faster than the alternative, Factory Girl. To create a fixture, you populate a YAML file, which creates records in a test database directly, skipping your initialization methods.
- Instances of ActiveRecord objects
- Formatted in YAML (1 file per model)
- Allow for ERB
- Rails loads all fixtures at once from ‘test/fixtures’ for models and controllers test (1. Remove any existing data from Test DB table for corresponding fixture 2. Load fixture data into table 3. Dump fixture data into variable in case of direct access)

Articles in Favor of Minitest and Fixtures:
- Testing with Minitest
- 7 Reasons I’m Sticking with Minitest and Fixtures in Rails
- Use Minitest for Your Next Rails Project
- Bow Before Minitest
- RailsConf 2015 - Ruby on Rails on Minitest by the author of Minitest (~ 33 mins.)
- How I Test Rails Apps with Minitest, Capybara, and Guard

Guides to Testing with Minitest and Fixtures
- *A Guide to Testing Rails Applications*
- *Minitest Quick Reference*
- *Testing Basics (w/ Minitest)*
- *Getting Started with Minitest*
- *Minitest with Rails*
- *How to Test Rails Models with Minitest*

Fixing Fixtures
—————————————————— ————————————————
—— ———————————————————— ———————————————

DMZ

Articles in Favor of Neither
- Balanced Opinion
- Fuck testing?

“I have to disagree. I've used both, and the projects that use fixtures have been more maintainable over time. You've used some red herrings to bash fixtures. Fixtures are loaded in specs with a key as per Dan McClain: users(:marko). And it's not a magic value, it's in the fixture. There are other fixture features that most don't bother investigating because they hear that factories are oh so wonderful, like it is easy to make associations, and that they are parsed as erb so you can put generated data in there (consider loops to generate a bunch of fixture data for tests that require specific numbering).
I've used factory girl. If you have to model a complex layout then it gets pretty hard to

see what is happening very quickly. They are slow if you need to test what happens when you hit the database. It isn't ideal to hit the database, but sometimes you have to, and having factory girl commit an entire structure is slow compared to it being loaded up front with fixtures. Fixtures allow you to take advantage of each spec being wrapped in a database transaction, factories do not.
FactoryGirl and other factories are add on gems. It has changed syntax over the years. I've an old project that uses it, and not much time. It is pretty buggered now, every other gem is up to date, but I cannot update that one without a ton of work. There were bugs in this earlier version of factory girl that required some patching, urgh even worse. Another project, same situation, fixtures, just works and has always worked, and I will bet it'll always work going forward.
And the big projects I've worked on use fixtures. So telling anyone new to rails who might be reading not to use them is terrible advice, you might come across them and have to use them anyway. My advice is, use fixtures with anger, get to know them. Try factories too, see what you think. Consider how both will stand up to the tests of time and then make a choice for your needs." — Random commenter on a FactoryGirl article (rebutting the article)

——————————————————————— ———————————————————— —— ——————————————————————— ————————————————

RUBY COMMUNITY FAVORITE

Community-preferred Alternative

- **RSpec** testing framework for Rails created for Behavior-driven Development (BDD) with a domain-specific language (DSL) for a more natural syntax
- **FactoryGirl** is a fixtures replacement that allows for "factories" that create test objects by running an ActiveRecord's object's initialization code
  - Is RSpec a dependency of FactoryGirl?
- Other possible gems to complete the testing custom testing suite below:
  - gem 'capybara'
  - gem 'database_cleaner'
  - gem 'launchy'
  - gem 'selenium-webdriver'

Gemfile for RSpec / FactoryGirl Testing Suite
(from 2014 out of the 7 Reasons article from the Minitest section)

```
group :test do
    gem "rspec-rails"
    gem "factory_girl_rails"
    gem "capybara"
    gem "selenium-webdriver"
    gem "database_cleaner"
    gem "shoulda-matchers"
end
```

RSpec

- RSpec Intro
- RSpec database cleaning strategies

Factory Girl

- “Fixtures replacement". One important way in which Factory Girl's "factories" differ from fixtures is that in order to create a test object with Factory Girl, you actually run the object's initialization code rather than directly populating a test database. As you can probably imagine, there can be benefits to this.
- Solves the main problem with fixtures: maintenance
- A factory is an object whose sole job is to create other objects
- You can request a model instance that is always current to the DB schema (never bound to a particular phase of the app development)
- Factories are dynamically loaded from the current state of the app
- Factory Method Pattern

Factories over Fixtures

- Slower but make up for costs with benefits in flexibility and readability
- Helps avoid Mystery Guests (by defining test data in the same place as the test)
- Generates valid, one-off objects as needed
- Dynamically send data to override default attributes to create variants for specific test cases

Articles in Favor of RSpec and FactoryGirl:

- Thoughtbot (creators of FactoryGirl, may be biased…)
- Replacing Fixtures with FactoryGirl
- Railscasts — Factories not Fixtures
- Hiring Thing — Rails Testing — Factory Girl

Guides to Testing with FactoryGirl

- Testing with Factory_Girl
- Thoughtbot — Testing Antipatterns

3-Part Video for Testing with RSpec, Capybara, and FactoryGirl

- Intro to RSpec
- RSpec with Capybara
- RSpec with Factory Girl

Best Practices for Using Data Factories

Factory Linting

- Linting is the process of analyzing code to detect potential errors
- Factory Linting is the process of detecting potential errors by validating attributes set in the factory
- It’s good for avoiding least expected bugs due to false positive test results

Just Enough Data

- Leave only required data inside your factory to reduce chances of bugs

Explicit Data Testing

- Things you want to test should be set in the test files and should reflect the state of the factory being tested
- Test expectations need to use explicit factory attributes, set to provide useful info on the test

Build Over Create

- The 'create' method saves to the DB, adding overhead to the test, which could eventually slow the test down
- Use 'build' or 'build_stubbed' for tests that don't need to be written to the DB, tests that don't require queries, or test that use stubs for abstracting away the complexity of the queries

Fixed Time-based Testing

- Relative time helper (e.g.— 2.seconds.ago) can cause split-second test inconsistencies when used to assert time-related data
- Avoid this by manually specifying time, using the 'time cop' gem to freeze time and running the helper w/o risking split-second inconsistencies

Look Into:

- Ffaker gem (used in conjunction with factory_girl to create randomized data to simulate production environment)
- guard-rspec gem (auto runs tests in the background to save time writing 'run test' commands)
- Relationship between RSpec and Factory Girl (is RSpec a dependency of FactoryGirl?)

—————————————————— ——————————————————— —— ——————————————————— ———————————————

PROS AND CONS

PROS

Minitest Pros:

- Default test framework for Rails
- Written solely in Ruby (because it's just a Ruby class!)
    - "minitest doesn't reinvent anything that ruby already provides, like: classes, modules, inheritance, methods. This means you only have to learn ruby to use minitest and all of your regular OO practices like extract-method refactorings still apply"

- Reads like Ruby code rather than natural language (but can mimic RSpec syntax via Minitest::Spec) (and can be humanized with extension)
- Supports both assertion- and spec expectation-style test definitions
- Easier to refactor based on common Ruby refactoring paradigms
- Tends to have fewer LOC
- Well documented (and natively supported)
- Configured out of the box

Fixture Pros:
- Default Rails test construct for generating and organizing sample data
- Loaded all at once: providing a robust metadata test base
- Configured out of the box

RSpec Pros:
- Created for Behavior Driven Development (BDD)
- Reads like English
- Outputs in color
- Everyone's doing it: "It is the most frequently used testing library for Ruby in production applications." — Semaphore.com
- Well documented and supported

Factory Pros:
- Dynamically defined and loaded from the current state of the app
  - "Firstly, instead of defining each Fixture, you define a Factory that can be used to create models of that type. The Factory is basically just a blueprint for creating instances of that model object"
  - "This means that only the data required for the test will be in the database and it's immediately clear where that data came from."
- Maintain simple test data definitions in a single place (but manage all data related to the current test in the test itself when needed)
  - "data that the test relies on is actually explicitly defined in the test"
- Create objects with valid and predictable data without needing to specify every attribute
- Well documented and supported (FactoryGirl)

CONS

Minitest Cons:
- Reads like Ruby code rather than natural language (but can mimic RSpec syntax via Minitest::Spec) (and can be humanized with extension)
- Outputs in B&W (can be colorized w/ extensions)

Fixture Cons:
- Statically defined
  - must be changed by hand when there are changes to the schema
  - with a more complex app, with many fixtures for all scenarios (all loaded at

once), the state of the data is more ambiguous

- Defined separate from the tests where their test data is tested
- Loaded all at once: making it harder to pin down failing tests and know the state of the test data
- “Fixtures add a certain level of ambiguity to your tests and can make it difficult to write clean tests once your application reaches a certain size” —Philip Brown
- Often lead to ‘Mystery Guests’

RSpec Cons:

- Being a DSL, it reads like English but in a very opinionated manner that obscures and abstracts the code running beneath it *(“RSpec is a DSL. Minutest is Ruby.” — Adam Hawkings, “Bow Before MiniTest”)*
- Everyone’s doing it: “It is the most frequently used testing library for Ruby in production applications.” — Semaphore.com
- Requires a number of gems for fully-optimized testing suite
- Requires more configuration out of the box

Factory Cons:

- Requires a number of gems for fully-optimized testing suite
- Requires more configuration out of the box

——————————————————— ———————————————————
—— ——————————————————— ———————————————

COMBOS

Best of Both

Factories with Minitest

- Have dynamically-defined test data, loaded in the desired state when needed, and tests written in native Ruby syntax

Worst of the Worst?

Fixtures with RSpec

- Statically-defined test data, loaded all at once in an ambiguous state, and tests written in a DSL language meant to be more readable but adds a layer of abstraction

——————————————————— ———————————————————
—— ——————————————————— ———————————————

VOCAB LESSON

Test Double: an object that can stand in for a real object in a test (think stunt double)

- *Mocks:* simulated objects that mimic the behavior of real objects (you assert against it)
- *Stubs:* programs that simulate the behaviors of software components (or modules)

that a module undergoing tests depends on (you do NOT assert against it)
  - *Spies:* designed to act as an observation point that allows for the recording of method invocations for later verification
  - ***Fakes****:* objects actually have working implementations, but usually take some shortcut which makes them not suitable for production

Specs*:* basically, tests written in a more natural language syntax (may require plugin depending on the test framework)
Listing: the process of analyzing code to detect potential errors
Transaction*:* in terms of testing, transactions are defined by the lifecycle of test data being passed into and eventually removed from the Test DB. (i.e.— the test framework will make a transaction of test data with the test DB in order to run test methods against that data) (searchio.techtarget) (transactional example for RSpec)
  - "start each example with a clean database, create whatever data is necessary for that example, and then remove that data by simply rolling back the transaction at the end" — Relish for RSpec

Truncation: a database cleaning strategy that removes data (until empty) from tables in a test DB for a clean testing environment (MySQL example)
Mystery Guests*:* an antipattern in which an object defined outside your test case that is used within the test case
Mutation Methods: methods that change the state of an object
Test Hooks: testing smells that are methods in production code that only run when in testing mode

Mocks, Fakes, Stubs, and Dummies

From Martin Fowler about Mock and Stub

Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production
Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.
Mocks are what we are talking about here: objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

From xunitpattern:

Mock Object that implements the same interface as an object on which the SUT depends. We can use a Mock Object as an observation point when we need to do Behavior Verification to avoid having an Untested Requirement (see Production Bugs on page X) caused by an inability to observe side-effects of invoking methods on the SUT.
Stub : This implementation is configured to respond to calls from the SUT with the values (or exceptions) that will exercise the Untested Code (see Production Bugs on page X) within the SUT. A key indication for using a Test Stub is having Untested Code caused by the inability to control the indirect inputs of the SUT

Fake: We acquire or build a very lightweight implementation of the same functionality as provided by a component that the SUT depends on and instruct the SUT to use it instead of the real.

Database Cleaning Strategies

The database cleaning strategies refer to database terminology. I.e. those terms come from the (SQL) database world, so people generally familiar with database terminology will know what they mean.
The examples below refer to SQL definitions. DatabaseCleaner however supports other non-SQL types of databases too, but generally the definitions will be the same or similar.

Deletion

This means the database tables are cleaned using using the SQL DELETE FROM statement. This is usually slower than truncation, but may have other advantages instead.

Truncation

This means the database tables are cleaned using the TRUNCATE TABLE statement. This will simply empty the table immediately, without deleting the table structure itself or deleting individual records.

Transaction

This means using BEGIN TRANSACTION statements coupled with ROLLBACK to roll back a sequence of previous database operations. Think of it as an "undo button" for databases. I would think this is the most frequently used cleaning method, and probably the fastest since changes need not be directly committed to the DB.
Example discussion: Rspec, Cucumber: best speed database clean strategy

Reason for truncation strategy with Capybara

The best explanation was found in the Capybara docs themselves:
# Transactional fixtures do not work with Selenium tests, because Capybara
# uses a separate server thread, which the transactions would be hidden
# from. We hence use DatabaseCleaner to truncate our test database.

Cleaning requirements

You do not necessarily have to clean your database after each test case. However you need to be aware of side effects this could have. I.e. if you create, modify, or delete some records in one step will the other steps be affected by this?
Normally RSpec runs with transactional fixtures turned on, so you will never notice this when running RSpec - it will simply keep the database automatically clean for you:
https://www.relishapp.com/rspec/rspec-rails/v/2-10/docs/transactions