Painless Test Driven Development

With Elixir & Phoenix

Kat Tornwall

Kat Tornwall

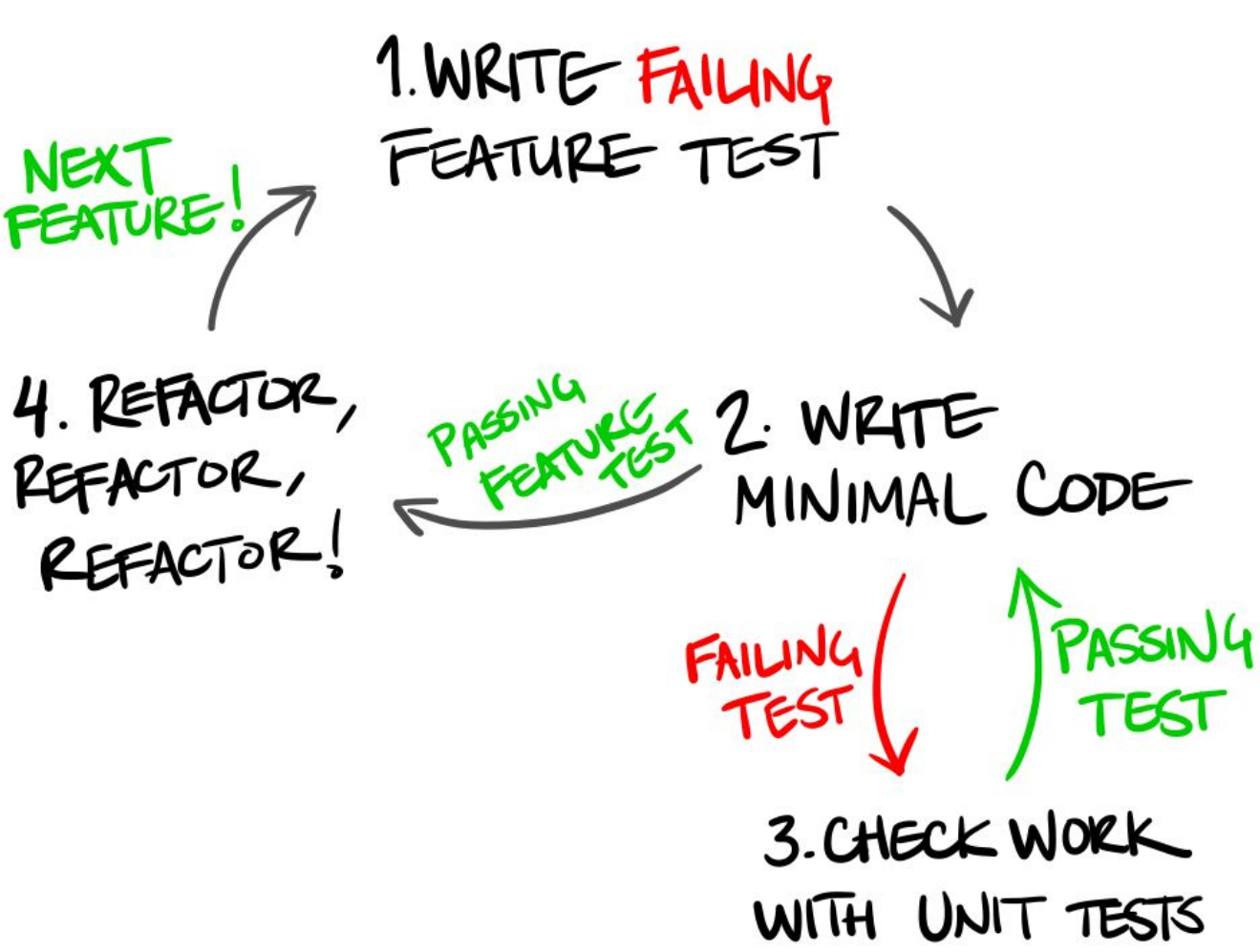
@ktornwall (Twitter, Github, Slack)

EVERYTHING BUT THE HOUSE

What is test driven development?

Test driven development

- Start development of features by writing a test, then writing the minimum amount of code to make the test pass
- Automated testing grows with the project, and gives developers a tight feedback loop
    - Did my code break anything? Run the tests!


1. WRITE FAILING FEATURE TEST
2. WRITE MINIMAL CODE
3. CHECK WORK WITH UNIT TESTS
4. REFACTOR, REFACTOR, REFACTOR!
NEXT FEATURE!
PASSING FEATURE TEST
FAILING TEST
PASSING TEST

“I get paid for code that works, not for tests, so my philosophy is to **test as little as possible to reach a given level of confidence**”

Kent Beck

Why do I love test driven development?

What makes testing painful?

So many tools...

What should I even use?

List from h4cc/awesome-elixir

Testing

Libraries for testing codebases and generating test data.

- amrita - A polite, well mannered and thoroughly upstanding testing framework for Elixir.
- blacksmith - Data generation framework for Elixir.
- blitzy - A simple HTTP load tester in Elixir.
- cobertura_cover - Writes a coverage.xml from `mix test --cover` file compatible with Jenkins' Cobertura plugin.
- ecto_it - Ecto plugin with default configuration for repos for testing different ecto plugins with databases.
- efrisby - A REST API testing framework for erlang.
- espec - BDD test framework for Elixir inspired by RSpec.
- espec_phoenix - ESpec for Phoenix web framework.
- ex_machina - Flexible test factories for Elixir. Works out of the box with Ecto and Ecto associations.
- ex_parameterized - Simple macro for parametarized testing.
- ex_spec - BDD-like syntax for ExUnit.
- ex_unit_fixtures - A library for defining modular dependencies for ExUnit tests.
- ex_unit_notifier - Desktop notifications for ExUnit.
- excheck - Property-based testing library for Elixir (QuickCheck style).
- factory_girl_elixir - Minimal implementation of Ruby's factory_girl in Elixir.
- faker - Faker is a pure Elixir library for generating fake data.
- fqc - FiFo Quickcheck helper, a set of helpers for running EQC.
- gimei - Gimei is a pure Elixir library for generating Japanese fake data.
- hound - Elixir library for writing integration tests and browser automation.
- hypermock - HTTP request stubbing and expectation Elixir library.
- katt - KATT (Klarna API Testing Tool) is an HTTP-based API testing tool for Erlang.
- kovacs - A simple ExUnit test runner.
- meck - A mocking library for Erlang.
- mix_erlang_tasks - Common tasks for Erlang projects that use Mix.
- mix_eunit - A Mix task to execute eunit tests.
- mix_test_watch - Automatically run your Elixir project's tests each time you save a file.
- mixunit - An EUnit task for Mix based projects.
- mock - Mocking library for the Elixir language.
- pavlov - BDD framework for your Elixir projects.
- plug_test_helpers - A simple testing DSL for Plugs.

Where’s my test?

Long test files can be hard to understand and navigate

Acceptance tests

often require knowledge of the DOM to understand

```
test "Searching for a summoner" do
  summoner = insert(
    :summoner,
    name: "katzenbar"
  )

  navigate_to "/"
  fill_field(
    {:class, "qa-summoner-search-input"},
    Summoner.name
  )
  click(
    {:class, "qa-summoner-search-submit"}
  )

  assert current_path == "/summoners"
  assert find_element(
    :class,
    "qa-summoner-#{summoner.id}"
  )
end
```

I don’t want to debug my tests

They should tell me what is wrong!

How do we fix these problems?

Always be aiming for these goals

Keep it simple
Make it easy to understand
➜ Eliminate pain points

League of Elixir Conf
Example on Github

https://github.com/ktornwall/elixir_conf_2016_demo


x09

So many tools... keep it simple!

Only include what you need

Minimal Viable Toolset

- Test framework (ExUnit)
- Browser integration, if needed (Hound)

(This space intentionally left blank)

Then add new tools as needed

- When you start to feel pain when writing tests, it's time to add a new tool
- Examples
  - Generating models by hand is becoming time consuming... (ex_machina)
  - I want random, reasonable fake data in my generated models... (faker)

Can’t find tests… make it easier to understand!

Organize your test files

Filename conventions

- My favorite way to organize my tests is to match the directory structure and filenames of my code
  - Test file is easy to find because the location is predictable
- Acceptance tests go into their own folders, and the folder/file structure matches my Phoenix routes
  - A lot of people like actions as their filenames, do what makes the most sense for you and your team

Long test files hard to work with?
Make it easy to understand!

Break tests into multiple files

Breaking up test files

- Modules that have a wide focus may end up with an unmanageable number of tests for the single file
- Create a folder with that module's name, and make files focusing on single aspects (like one file per function)

Spending too much time building your test data?
Make it simple!

Make factories and helpers do the work for you

Use factories to build data

- Libraries like `ex_machina` allow you to generate example models quickly
  - Works with or without Ecto to fit your needs
- Factories can act as documentation for what your models might look like
  - Is name a full name, a first name, or a screen name? Your factory can tell you!

Use helpers for common setup

- Helpers are useful for setting up complex scenarios
- May be global
  - `sign_in` has been a helper in most of my Phoenix acceptance tests
- Can also be local
  - `build_annie` might be used in my demo app to test data of a real League champion

Acceptance tests can be hard to understand…
But we can fix that!

Write the test that tells your user's story, the way they would

I love writing acceptance tests

- Acceptance tests should tell your user's story
  - How they are actually going to use the application
- A good suite of acceptance tests can document how you expect users to interact with your application

Acceptance test are often tightly coupled to the DOM and hard to read...

```elixir
test "Searching for a summoner" do
  summoner = insert(:summoner, name: "katzenbar")

  navigate_to "/"
  fill_field({:class, "qa-summoner-search-input"}, summoner.name)
  click({:class, "qa-summoner-search-submit"})

  assert current_path == "/summoners"
  assert find_element(:class, "qa-summoner-#{summoner.id}")
end
```

Make it easy to understand!

Use page modules to abstract away DOM interactions

Page modules

- Encapsulates interactions with the browser
  - Pages
  - Common forms
  - Header
- Make changes to the DOM? Code only needs changed in one place
- Functions in a page module are
  - Actions (interacting with the DOM or navigation)
  - Checks (inspecting the DOM to see we have the right data)

Page module actions

- Function names are actions that users would take
  - `visit`
  - `view_mastery`
  - `fill_form`

Page module assertions

- Function names end with a question mark and return a boolean
  - `current_page?`
  - `has_mastery?`

Tests sometimes don’t tell us why they are failing…
Eliminate this pain point!

Write better assertions

Assertions are more than checks

- Think through the assertions that you are writing so that they give you understandable errors
  - Use intermediate assertions when accessing a long list of properties
  - Check that you are on the correct URL before looking for elements on the wrong page

Sometimes making meaningful assertions can seem hard…

```
~/dev/elixir_conf_2016_demo master*
❯ mix test test/acceptance/summoners/index_test.exs


  1) test Searching for a summoner (ExConf.Acceptance.SummonersIndexTest)
     test/acceptance/summoners/index_test.exs:6
     Expected truthy, got false
     code: SummonerIndexPage.current_page?(summoner)
     stacktrace:
       test/acceptance/summoners/index_test.exs:12: (test)



Finished in 1.4 seconds
1 test, 1 failure

Randomized with seed 670099
```

Custom assertions

- ExUnit provides a way to print custom messages on failing assertions
- We can take this one step further and provide custom assertion macros for common scenarios

```
~/dev/elixir_conf_2016_demo master* 8s
❯ mix test test/acceptance/summoners/index_test.exs
Compiling 1 file (.ex)


  1) test Searching for a summoner (ExConf.Acceptance.SummonersIndexTest)
     test/acceptance/summoners/index_test.exs:6
     Incorrect path, expected "/summoners" to be "/summoners?summoner[name]=katzenbar"
     stacktrace:
       test/acceptance/summoners/index_test.exs:12: (test)



Finished in 3.8 seconds
1 test, 1 failure

Randomized with seed 344238
```

Other tips and tricks

`mix test` has sweet options

- Read `mix help test` and see all the cool goodies
  - `--trace` gives nicer test output, though it does not support asynchronous tests
  - `--stale` (introduced in Elixir 1.3) runs only test files that have changed since you last ran stale tests
  - Filters are extremely useful, I use `@tag :current` and `mix test --only current` to focus on specific acceptance tests when practicing TDD

Limit use of setup blocks

- I have found setting up my models at the beginning of my tests *enjoyable*
  - Clearer what the current context is
  - Easier to only create what is needed for that test
- Setup blocks are still really useful for running helper tasks
  - Signing in before performing actions that require authorization

Asynchronous browser tests

- Ecto 2 made it really easy to run asynchronous browser tests with Hound
- Cut test time in half on a larger project
- See blog post on how to set this up: http://rockwood.me/2016/concurrent-feature-tests-with-phoenix/

Questions?

Kat Tornwall @ktornwall

https://github.com/ktornwall/elixir_conf_2016_demo