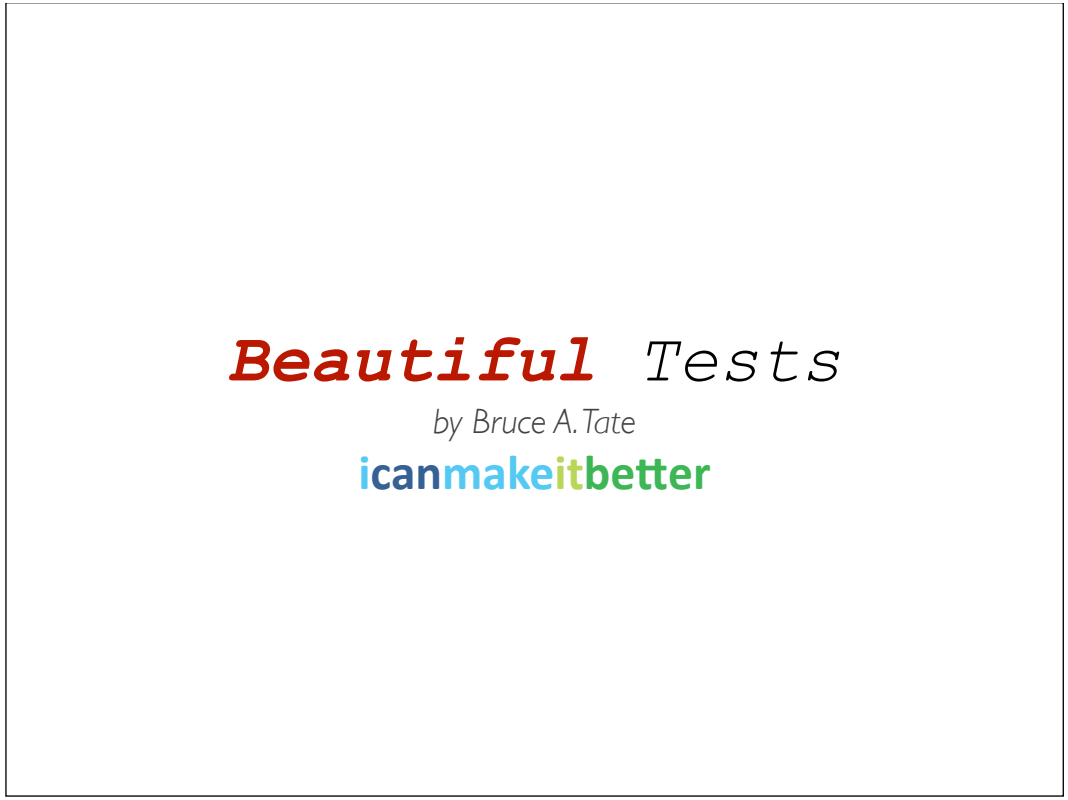
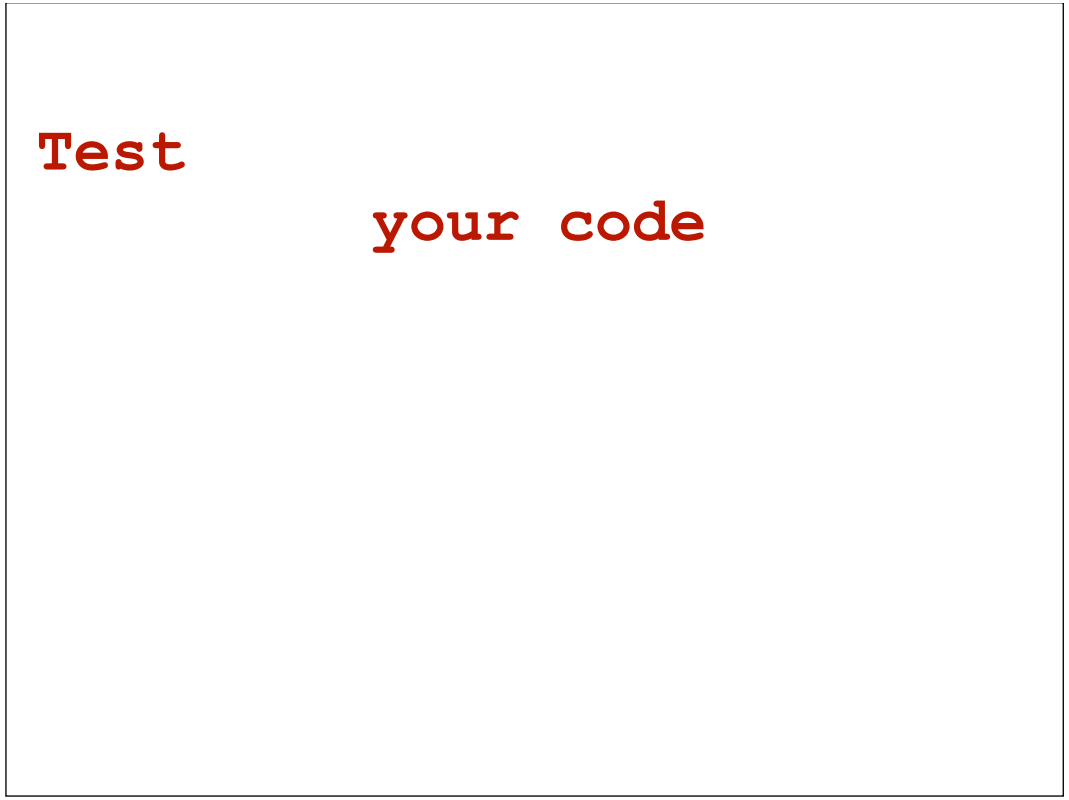


It’s good to be here... I almost wasn’t.

Test
  all of your code
    with
      beautiful,
        dry,
          fast
            tests



Many of us come from a testing culture... that's good.

Testing tools are in their infancy... that's bad.

This talk is about a set of band aids we added to existing tools until ExUnit can come around.

Test
  all of your code

if it is **worth writing**

if it is **worth writing**
    it is **worth testing**

don't let your **customers**
test your **code**

we use **excoveralls**

https://github.com/parroty/excoveralls

setup

```elixir
defmodule Chat.Mixfile do
  defp deps do
    [{:excoveralls, only: :test}]
  end

  defp test(args) do
    Mix.Task.run("test", [])

    Mix.shell.info("")
    Mix.shell.info("$ mix coveralls
                    - coverage overview")
    Mix.shell.info("$ mix coveralls.detail FILENAME
                    - line-by-line coverage of file")
  end

  defp cli_env do
    [coveralls: :test,
     "coveralls.detail": :test]
  end
end
```

https://gist.github.com/batate/fd9e7569b3861a80a0b3

report

```
[imbe] (develop=) ➔ mix coveralls
Compiled lib/chat/room_supervisor.ex
Compiled lib/chat.ex
Compiled lib/chat/attachment_cache.ex
Compiled lib/chat/socket/handler.ex
Compiled lib/chat/room.ex
Generated chat.app
.....................................................................................
.........................

Finished in 8.1 seconds (2.4s on load, 5.6s on tests)
101 tests, 0 failures

Randomized with seed 167783
----------------
COV    FILE                                        LINES RELEVANT   MISSED
100.0% lib/chat.ex                                    98       18        0
100.0% lib/chat/attachment_cache.ex                  221       81        0
100.0% lib/chat/crypto.ex                             47       17        0
100.0% lib/chat/http/process.ex                       34       13        0
100.0% lib/chat/models.ex                             16        4        0
100.0% lib/chat/models/company.ex                     89        1        0
100.0% lib/chat/models/multi_attachment.ex            45       13        0
  0.0% lib/chat/models/page.ex                        11        0        0
100.0% lib/chat/models/survey.ex                      83        4        0
100.0% lib/chat/models/survey_question.ex             80       12        0
100.0% lib/chat/models/survey_response.ex             66        8        0
100.0% lib/chat/models/token.ex                       51        8        0
```

This is hard to read unless you have zeros on the right hand side!

```
100.0% lib/chat/models/company.ex                    89        1        0
100.0% lib/chat/models/multi_attachment.ex           45       13        0
  0.0% lib/chat/models/page.ex                       11        0        0
100.0% lib/chat/models/survey.ex                     83        4        0
100.0% lib/chat/models/survey_question.ex            80       12        0
100.0% lib/chat/models/survey_response.ex            66        8        0
100.0% lib/chat/models/token.ex                      51        8        0
100.0% lib/chat/models/user.ex                       80        7        0
100.0% lib/chat/mongo.ex                            118       37        0
100.0% lib/chat/mongo/cursor.ex                      92       20        0
100.0% lib/chat/mongo/model.ex                      105       39        0
100.0% lib/chat/mongo/process.ex                    176       65        0
100.0% lib/chat/mongo/query.ex                       54       16        0
100.0% lib/chat/mongo/worker.ex                      75       14        0
  0.0% lib/chat/repo.ex                              14        0        0
100.0% lib/chat/room.ex                             786      329        0
100.0% lib/chat/room_supervisor.ex                   29        7        0
100.0% lib/chat/router.ex                            15        2        0
100.0% lib/chat/socket.ex                            20        2        0
100.0% lib/chat/socket/handler.ex                   363      169        0
100.0% lib/chat/socket/process.ex                    87       25        0
100.0% lib/chat/socket/sockjs.ex                     62       20        0
100.0% lib/chat/topic.ex                            167       41        0
100.0% lib/chat/topic/gc.ex                          46       13        0
[TOTAL] 100.0%
----------------

$ mix coveralls                 - coverage overview
$ mix coveralls.detail FILENAME - line-by-line coverage of file
```

The bottom line makes it easier for our devs to do the right thing.

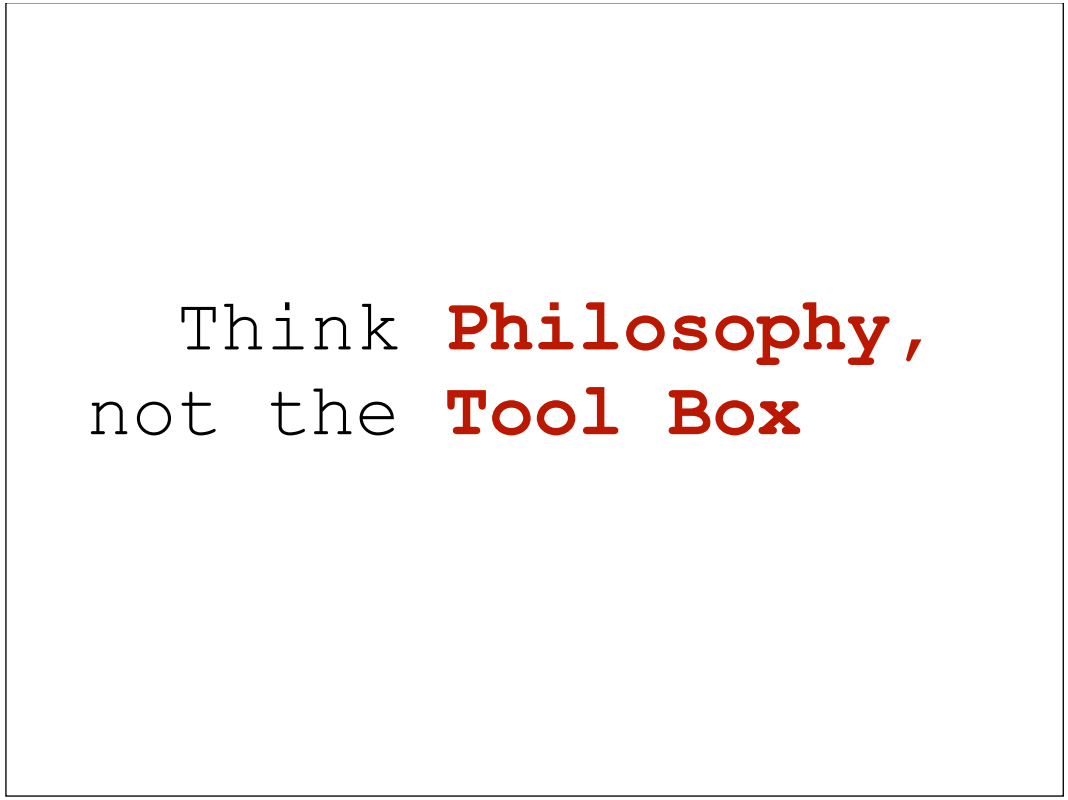
detailed report

```
...
defp receive_nonmember(false, conn, s, chat) do
  room = Room.join(s.room, chat, s.user, s.token)

  if s.role != "admin" do
    # Filter unpublished questions
    questions = Enum.filter(room.questions, & &1.published)

    # Filter hidden messages
    questions =
      Enum.map(questions, fn question ->
        answers = Enum.reject(question.answers, & &1.hidden)
        %{question | answers: answers}
      end)

    # Filter blocked members
...
```

Test
all of your code
with ???

tests

When we decided to code Elixir for our production servers, we had a decision to make.
want a good elixir... compressed schedules



We want to show you real code so that means context. But the tool set doesn't matter.
That said...

We use...

- ExUnit
- ShouldI (batate/shouldi)
- Blacksmith (batate/blacksmith)

We hope to push as much of this into shouldi as possible

We use...

- ExUnit **(our goal: 100%)**
- ShouldI (batate/shouldi)
- Blacksmith (batate/blacksmith)

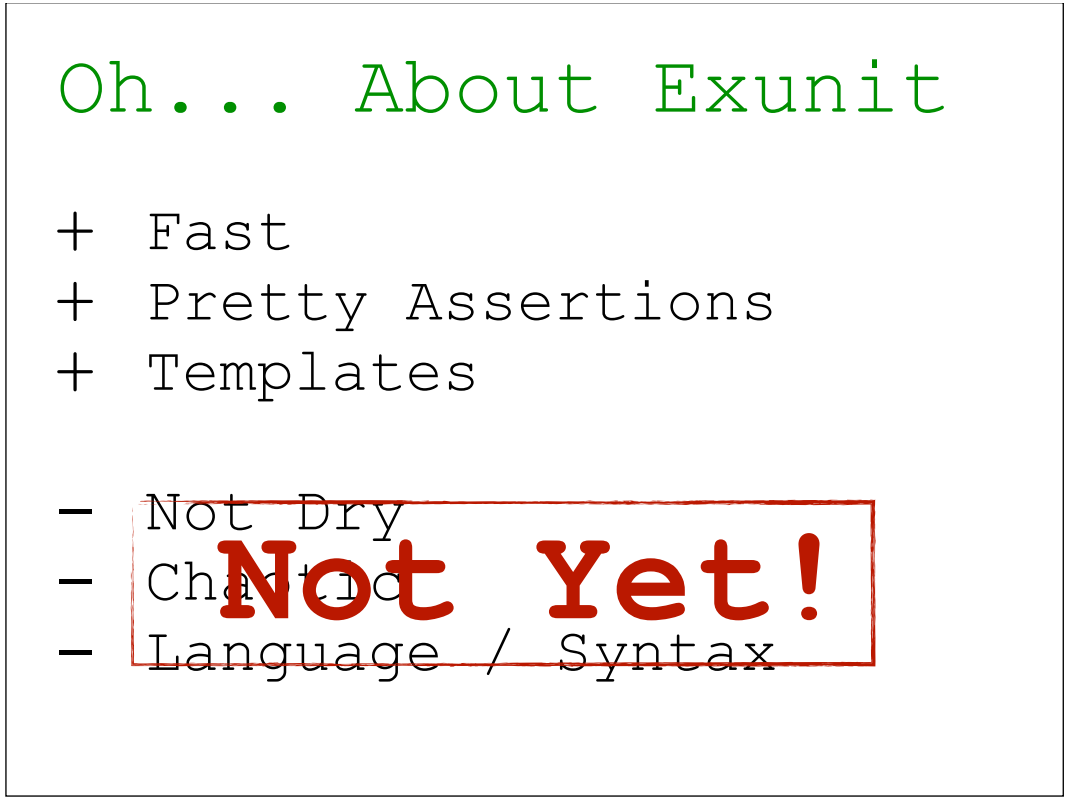
Oh... About Exunit

+ Fast
+ Pretty Assertions
+ Templates

Oh... About Exunit

+ Fast
+ Pretty Assertions
+ Templates

- Not Dry
- Chaotic
- Language / Syntax



Our goal is to work directly with the core team to improve tests where we can help.

```ruby
setup do
  # universal setup
end

test "a get" do
  ...
end

test "logged in get" do
  login_user
  ...
end

test "logged in post" do
  login_user
  ...
end
```

```ruby
setup do
  # universal setup
end

test "a get" do
  ...
end

test "logged in get" do
  login_user
  ...
end

test "logged in post" do
  login_user
  ...
end
```

```ruby
setup do
  # universal setup
end

test "a get" do
  ...
end

test "logged in get" do
  login_user
  ...
end

test "logged in post" do
  login_user
  ...
end
```

This becomes a big problem as overarching tests get nested.
in models: persistent vs not; error vs happy path; etc.
So setup code can be a big pain

a **horror** story

```
test "gets and updates many levels deep dependencies" do
  Mix.Project.push DepsOnGitApp


  in_fixture "no_mixfile", fn ->
    Mix.Tasks.Deps.Get.run []


    message = "* Getting git_repo (#{fixture_path("git_repo")})"
    assert_received {:mix_shell, :info, [^message]}


    message = "* Getting deps_on_git_repo (#{fixture_path("deps_on_git_repo")})"
    assert_received {:mix_shell, :info, [^message]}


    assert File.exists?("deps/deps_on_git_repo/mix.exs")
    assert File.rm("deps/deps_on_git_repo/.fetch") == :ok
    assert File.exists?("deps/git_repo/mix.exs")
```

Testing is good, but it's not enough to test.
This test is from the Elixir framework. This test is more insufficient tooling.

```
    # Compile git repo but unload it so...
    Mix.Tasks.Deps.Compile.run ["git_repo"]
    assert File.exists?("_build/dev/lib/git_repo/ebin")
    Code.delete_path("_build/dev/lib/git_repo/ebin")

    # Deps on git repo loads it automatically on compile
    Mix.Task.reenable "deps.loadpaths"
    Mix.Tasks.Deps.Compile.run ["deps_on_git_repo"]
    assert File.exists?("_build/dev/lib/deps_on_git_repo/ebin")
  end
after
  purge [GitRepo, GitRepo.Mix]
end
```

You can see that the test creator wants to do the right thing, but can’t.

```
    # Compile git repo but unload it so...
    Mix.Tasks.Deps.Compile.run ["git_repo"]
    assert File.exists?("_build/dev/lib/git_repo/ebin")
    Code.delete_path("_build/dev/lib/git_repo/ebin")

    # Deps on git repo loads it automatically on compile
    Mix.Task.reenable "deps.loadpaths"
    Mix.Tasks.Deps.Compile.run ["deps_on_git_repo"]
    assert File.exists?("_build/dev/lib/deps_on_git_repo/ebin")
  end
after
  purge [GitRepo, GitRepo.Mix]
end
```

But the framework is fighting against him.


```
    # Compile git repo but unload it so...
    Mix.Tasks.Deps.Compile.run ["git_repo"]
    assert File.exists?("_build/dev/lib/git_repo/ebin")
    Code.delete_path("_build/dev/lib/git_repo/ebin")

    # Deps on git repo loads it automatically on compile
    Mix.Task.reenable "app.loadpaths"
    Mix.Tasks.Deps.Compile.run ["deps_on_git_repo"]
    assert File.exists?("_build/dev/lib/deps_on_git_repo/ebin")
  end
after
  purge [GitRepo, GitRepo.Mix]
end
```

We are going to stamp this chaotic.
It violates principles of coupling and single purpose.
Also, the reporting can’t help us out as much as it should

Shouldi

+ Fast
\+ Pretty Assertions
\+ Templates-Code

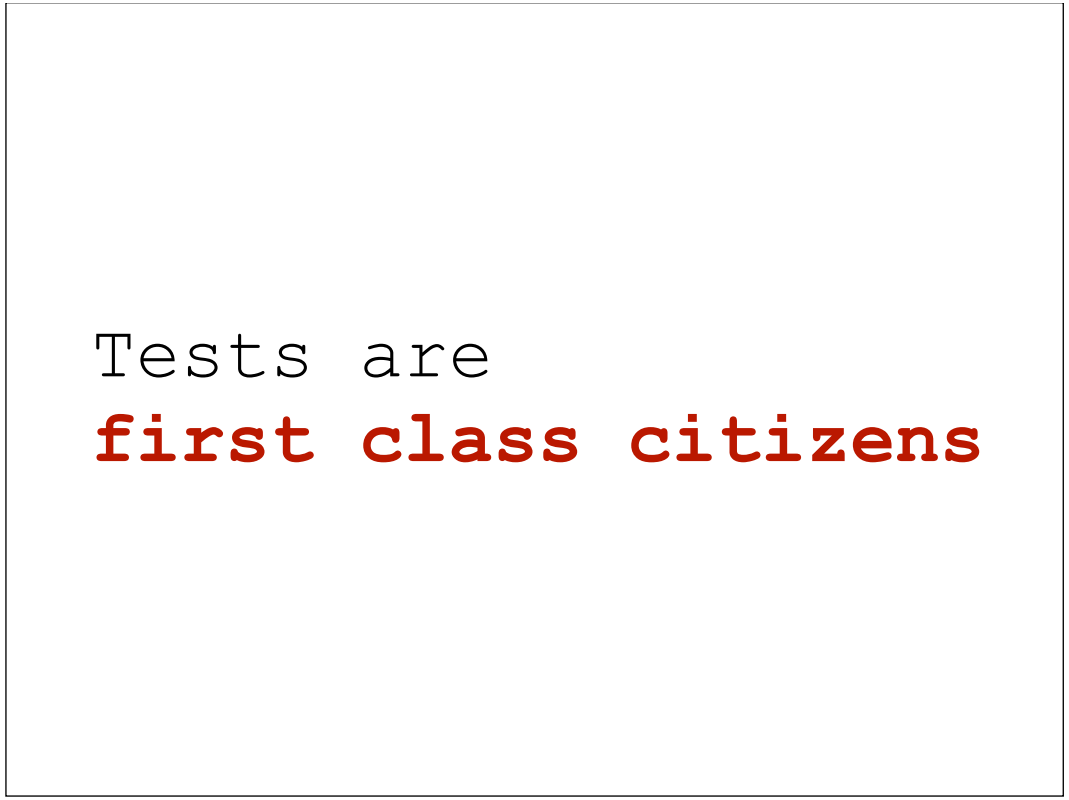
– ~~Not~~ **Dry**
– ~~Chaotic~~ **Beautiful**
± **Language** **/** Syntax

Blacksmith

\+ **Fast**
\+ Pretty Assertions
\+ Templates-Code and **Data**

\- ~~Not~~ **Dry Data**
\- ~~Chaotic~~ **Beautiful Data**
± **Language** / Syntax

Beautiful is important.



You see,

Language Matters

Said another way, language shapes thought. Syntax shapes language.

```
test "chat" do
  chat = Chat.create(...)

  assert something_about_chat
end
```

Why do we get names like this over and over?
Because the language of “test” isn’t strong enough.

The language is for the designers of the framework, not the test.

```
test "chat" do
  chat = Chat.create(...)

  assert something_about_chat
end
```

The word we use here

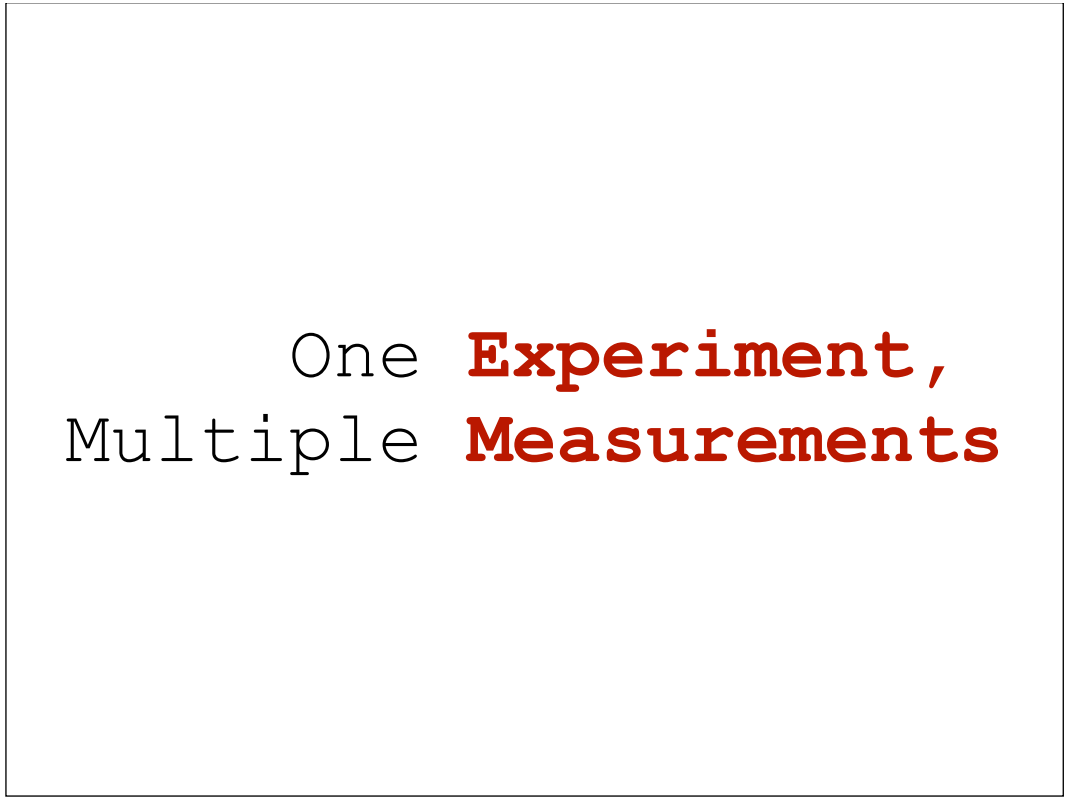
```
test "chat" do
  chat = Chat.create(...)

  assert something_about_chat
end
```

The word we use here

should language improves the thought process: single purpose experiment.

```
should "create chat" do
  chat = Chat.create(...)

  assert something_about_chat
end
```

Push this language into the framework
and we'll be reminded to give all tests better names
and a single purpose.



Our overarching philosophy: one experiment, multiple measurements.

```elixir
test "chat" do
  bucket = create_bucket
  assert %{__struct__: "Bucket"} = bucket
  assert Bucket.empty?(bucket)

  Bucket.add(bucket, 1)
  assert bucket.contents == [1]
end
```

Multiple experiments, Multiple measurements

Tightly coupled, encourages abuse.

```elixir
setup context do
  assign bucket: create_bucket
end

should "create struct bucket", context do
  assert %{__struct__: "Bucket"} = context.bucket
end

should "be empty", context do
  assert Bucket.empty?(bucket)
end

should "add to bucket", context do
  Bucket.add(bucket, 1)
  assert bucket.contents == 1
end

should "remove from bucket", context ...
```

Our Experiment

:) / :(

```
setup context do
  assign bucket: create_bucket
end
```

```
should "create struct bucket", context do
  assert %{__struct__: "Bucket"} = context.bucket
end

should "be empty", context do
  assert Bucket.empty?(bucket)
end

should "add to bucket", context do
  Bucket.add(bucket, 1)
  assert bucket.contents == 1
end

should "remove from bucket", context ...
```

This is a compromise. It will allow us to tailor some concepts in advance of changes in exunit

```
setup context do
  assign bucket: create_bucket
end

should "create struct bucket", context do
  assert %{__struct__: "Bucket"} = context.bucket
end

should "be empty", context do
  assert Bucket.empty?(bucket)
end

should "add to bucket", context do
  Bucket.add(bucket, 1)
  assert bucket.contents == 1
end

should "remove from bucket", context ...
```

We can improve...
these should blocks are sometimes patterns that can be expanded through macros

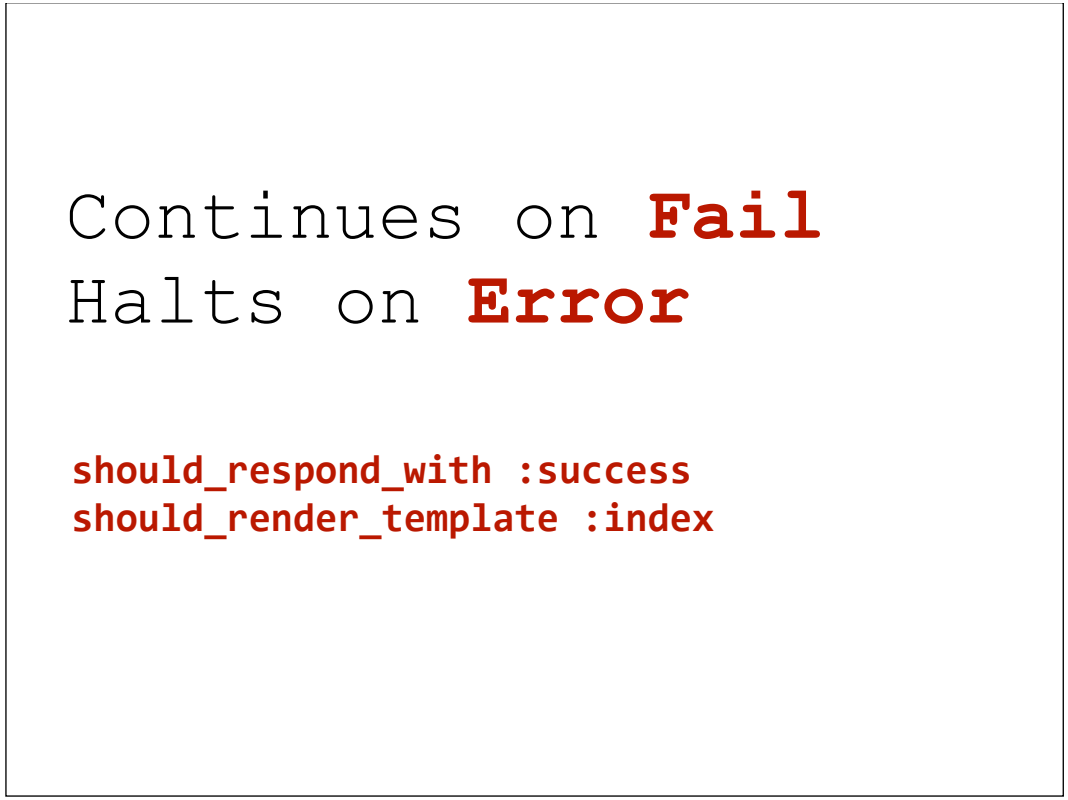
```
setup context do
  assign bucket: create_bucket
end

should_match_key :bucket, %{ __struct__: "Bucket" }
should_match_key :bucket, %{ contents:  [] }

should "add to bucket", context do
  Bucket.add(bucket, 1)
  assert bucket.contents == 1
end

should "remove from bucket", context ...
```

In both forms, we have one experiment and multiple measurements.

```
Tests the context.

should_have_key
should_match_key

Or tests for Plug connection.

should_respond_with :success
should_render_template :index

Or any framework specific matchers...
```

In both forms, we have one experiment and multiple measurements.

We are selling our soul here... macros instead of functions.

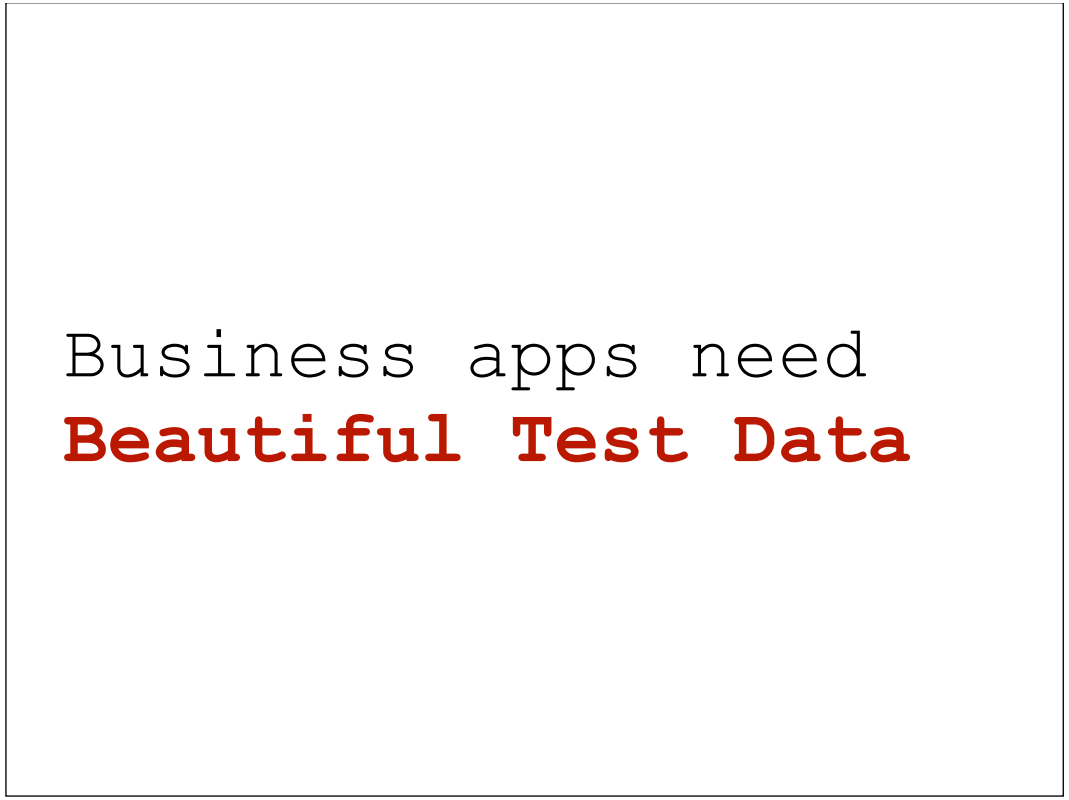


In both forms, we have one experiment and multiple measurements.

We are selling our soul here... macros instead of functions.

Business apps need

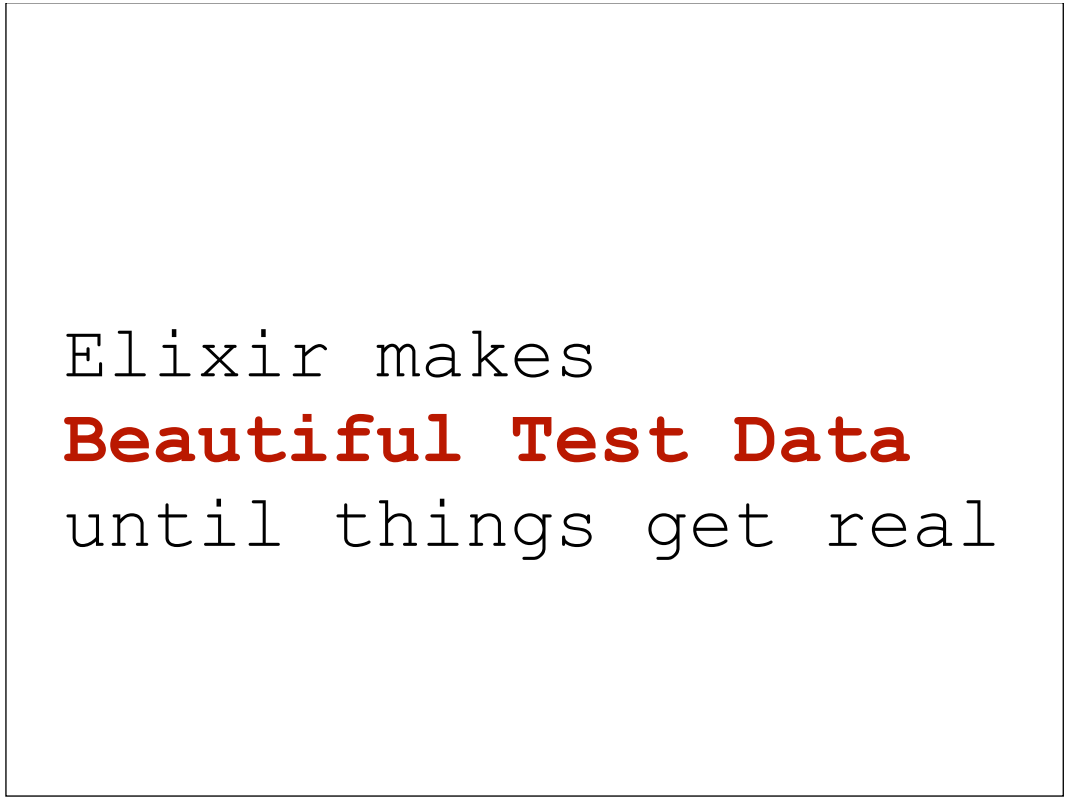
Test Data



Remember, tests are first class citizens

Elixir makes

Beautiful Test Data



Persistence
Different formats JSON, structs, maps
# of attributes

Persistence
Data issues can swallow tests

Mostly just a functional library with a few key macros

Create structured data for tests

Blacksmith **Templates**

```elixir
defmodule Forge do
  use Blacksmith
  register :user,
    name: Faker.Name.first_name,


    description: Faker.Lorem.sentence




end
```

Faker is a library that creates fake data

```elixir
defmodule Forge do
  use Blacksmith
  register :user,
    name: Faker.Name.first_name,


    description: Faker.Lorem.sentence,

    always_the_same: "string"



end
```

```elixir
defmodule Forge do
  use Blacksmith
  register :user,
    name: Faker.Name.first_name,
    email: Sequence.
       next(:email, &"test#{&1}@example.com"),
    description: Faker.Lorem.sentence,

    always_the_same: "string"



end
```

Maybe you have a database backed test that should be isolated

In that test, email must be unique

```elixir
defmodule Forge do
  use Blacksmith
  register :user,
    name: Faker.Name.first_name,
    email: Sequence.
       next(:email, &"test#{&1}@example.com"),
    description: Faker.Lorem.sentence,
    roles: [],
    always_the_same: "string"

  register :admin,
    [prototype: :user],
    roles: ["admin"]
end
```

Maybe some data should be based on other data
Second form of the register function has options (in the second position)

Blacksmith **Config**

```elixir
defmodule Blacksmith.Config do
  def save(repo, map) do
    repo.insert(map)
  end

  def save_all(repo, list) do
    Enum.map(list, &repo.insert/1)
  end
end
```

Blacksmith **Usage**

```
user = Forge.user

user = Forge.saved_user Models.User,
         name: "Will Override"

admin = Forge.admin

admin = Forge.admin_list 5
```

Test
all of your code
with
beautiful,
dry,

tests

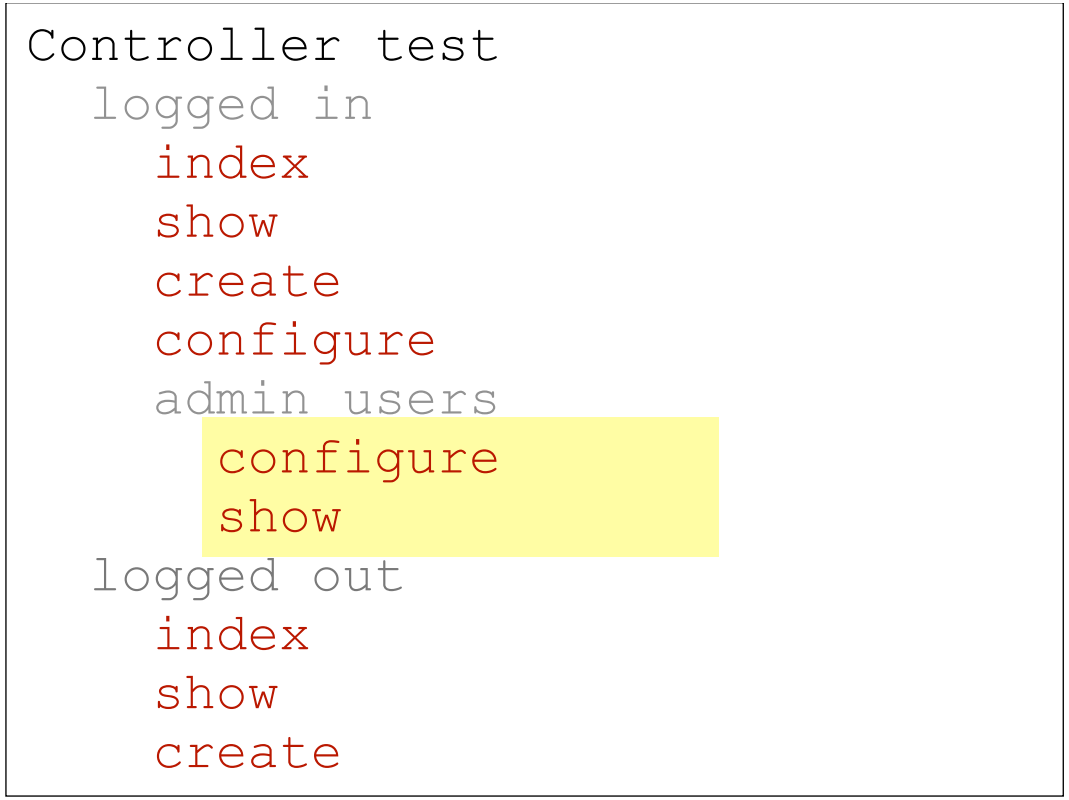
To have **dry tests,**
you need **specialized setups**
with **nested context**

Controller test

- index
- show
- create
- configure

```
Controller test
  logged in
    index
    show
    create
    configure


  logged out
    index
    show
    create
```

```
Controller test
  logged in
    index
    show
    create
    configure
    admin users
      configure
      show
  logged out
    index
    show
    create
```


Controller test
logged in
index
show
create
configure
admin users
configure
show
logged out
index
show
create

```
test "Logged in admin user gets configure" do
  part = Forge.saved_part(...)
  conn = setup_connection
  admin = Forge.saved_admin( ... )
  sign_in admin
  conn = get conn, :configure
  assert conn.status == 200
end

test "Logged in user user gets configure" do
  part = Forge.saved_part(...)
  conn = setup_connection
  user = Forge.saved_user( ... )
  sign_in user
  conn = get conn, :configure,
  assert conn.status == 301
end
```

```
test "Logged in admin user gets configure" do
  part = Forge.saved_part(...)
  conn = setup_connection
  admin = Forge.saved_admin( ... )
  sign_in admin
  conn = get conn, :configure
  assert conn.status == 200
end

test "Logged in user user gets configure" do
  part = Forge.saved_part(...)
  conn = setup_connection
  user = Forge.saved_user( ... )
  sign_in user
  conn = get conn, :configure,
  assert conn.status == 301
end
```

Too much duplication

```
with "a part and a connection" do
  ...setup for a connection and a part

  with "a logged in user" do
    ...setup for logged in user

    tests for logged in user

  with "a logged in admin" do
    ...make the user an admin

    tests for logged in admin
```

Too much duplication

```
with "a get to configure" do
  setup do
    assign conn: get( :configure, context.part.id)
  end

  should_respond_with :success
  should_render_template :configure
end
```

now the test is easy

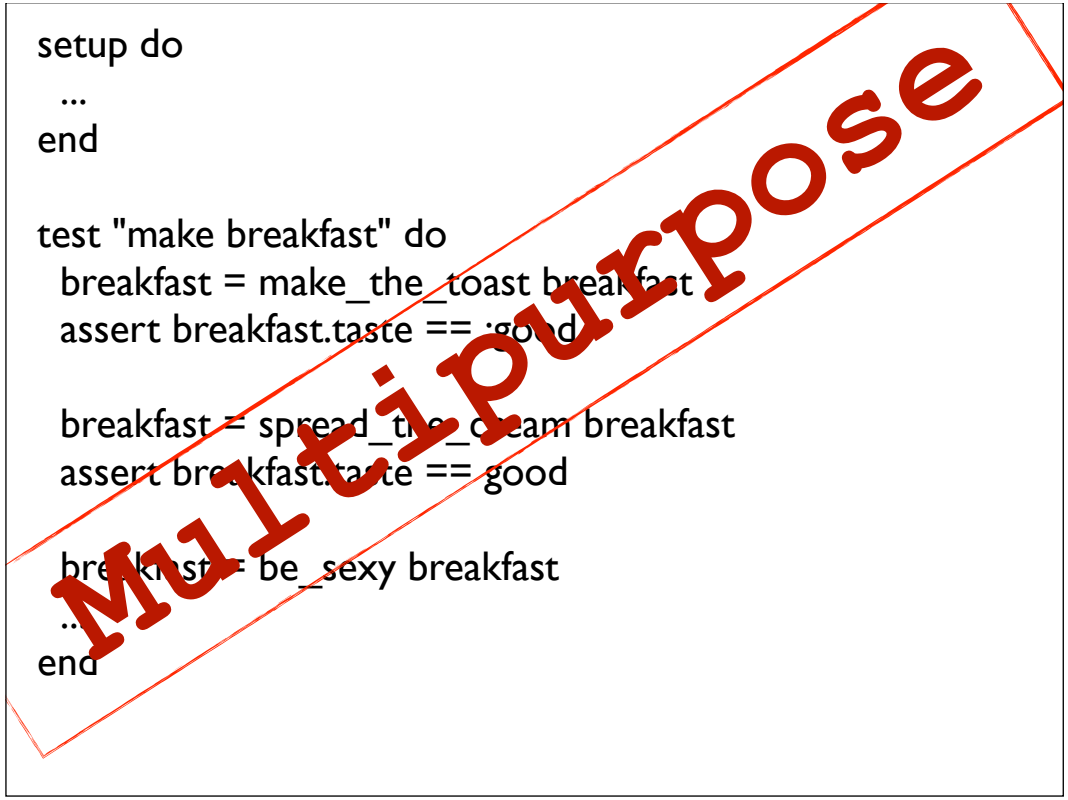
Test
all of your code
with
beautiful,
dry,
fast
tests

Jose, you crack me up.

We were batting some test code back and forth over skype and emails. here's Jose's example

```
setup do
  ...
end

test "make breakfast" do
  breakfast = make_the_toast breakfast
  assert breakfast.taste == :good

  breakfast = spread_the_cream breakfast
  assert breakfast.taste == good

  breakfast = be_sexy breakfast
  ...
end
```

No. This will deteriorate with time.
But maybe it takes a long time to make toast. So...



We have all written this test. But we can also see this test deteriorate with time.

But maybe it takes a long time to make toast. So... we live with the consequences.

```
setup do
  ...
  %{ breakfast: create_breakfast ...}
end

...
end
```

No. This will deteriorate with time.
But maybe it takes a long time to make toast. So...

```
setup...

def make_toast ...

def spread_cream breakfast do
  breakfast = spread_the_cream breakfast
  assert breakfast.taste == good
  breakfast
end

def be_so_sexy ...
...
should...
```

No. This will deteriorate with time.
But maybe it takes a long time to make toast. So...

```
def setup

def make_toast ...
def spread_cream ...
def be_so_sexy ...
...

test "make breakfast", context do
  context.breakfast
  |> make_toast
  |> spread_cream
  |> be_so_sexy
end
```

Much better. Single purpose...
The tooling can’t help as much as we would like
Want failures reported from make_toast

Integration tests
will be

```
  ...
  step "make the toast", breakfast do
    breakfast = make_the_toast breakfast
    assert breakfast.taste == :good
    breakfast
  end

  step "spread the cream", breakfast do
    ...
  end

  step "be sexy", breakfast do
    ...
  end
end
```

2. Concurrency matters

Functional programming helps.
Database backed tests: Allow data templates which guarantee unique attributes.

3. Isolate the **Database**

Blacksmith list as a Repository

Directions

In ShouldI

- Push experiments out of setup blocks, back into tests
- Matchers are macros

Into ExUnit

- Nested Context
- Continue on Fail
- Assertion Customization
- Integration Tests into ExUnit

Test

all of your code

with

beautiful,

dry,

fast

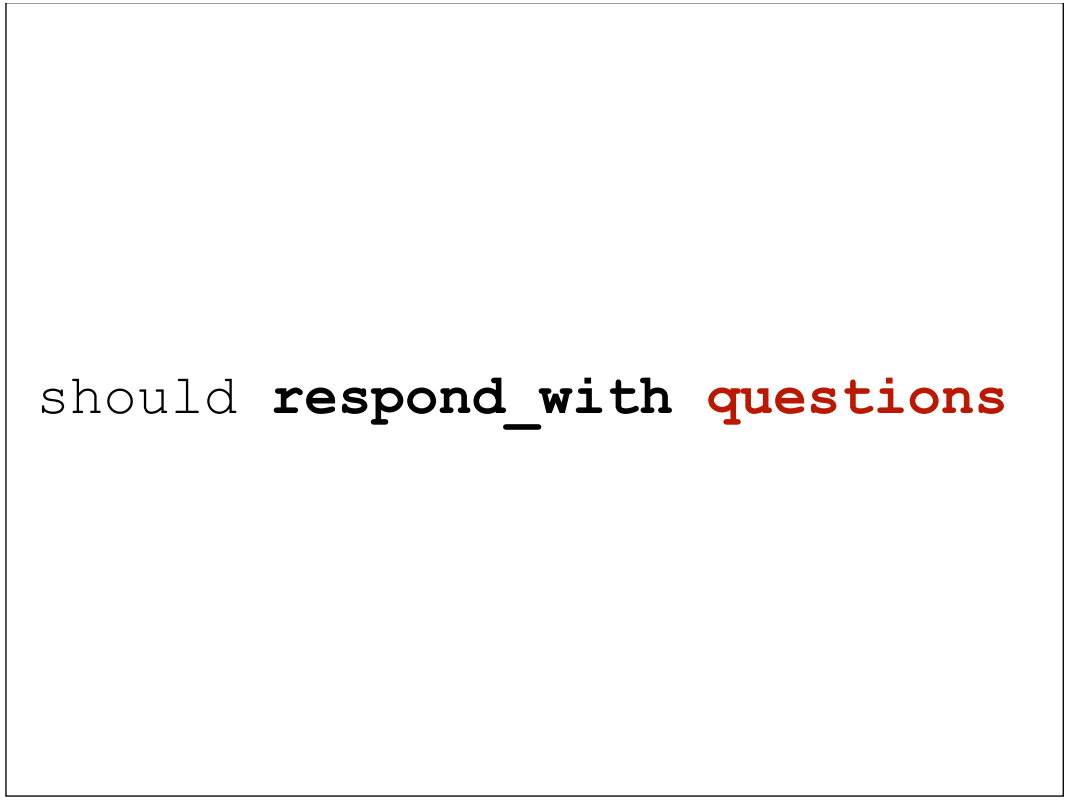
tests

Test
  all of your code
    with
      beautiful,
        dry,
          fast
            tests

Test
all of your code
with
beautiful,
dry,
fast
tests

Test
all of your code
with
beautiful,
dry,
fast
tests

Test
all of your code
with
beautiful,
dry,
fast
tests



Functional programming helps.
Database backed tests: Allow data templates which guarantee unique attributes.