WHAT IS TESTING?

Checking that code does what it's supposed to

TYPES OF TESTING

- **Manual testing**: a human checks that code does what it's supposed to do
- **Automated testing**: a human gets a computer to check that code does what it's supposed to do

EXAMPLE OF MANUAL VS AUTOMATED

If you have a name presence validation in your User model, how would you check that it's working?

Manual

- Go into the Rails console and create a user without a name; OR
- Fill out and submit the Sign Up form without a name

Automated

- Run code that creates a user without a name; OR
- Run code that fills out and submits the Sign Up form without a name

TESTING PHILOSOPHIES

- Verification testing ("test after")
  1. Write code
  2. Write tests

- Test first/TDD
  1. Write tests
  2. Write code

LEVELS OF TESTS

- Unit tests
  - Tests individual methods in isolation
- Integration/Feature/Acceptance/System tests
  - Check that all the "units" of your app work together to produce the functionality you expect

LET'S LOOK AT A TEST

We're going to use a framework called MiniTest

WHY TEST?

Save time - Simulate user interaction so you don't have to

WHY TEST?

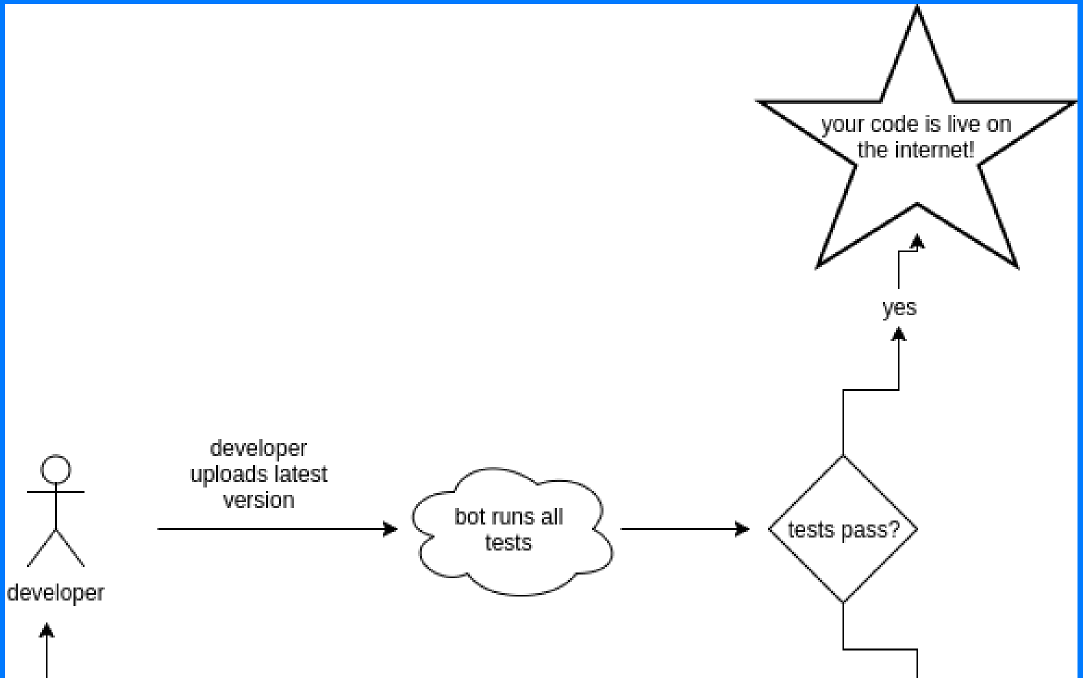
Safety harness - Regularly run tests as you work on an application to catch bugs as they develop

WHY TEST?

Safety harness - It's less risky to refactor/improve/update existing code if you have tests that will tell you if anything breaks

WHY TEST?

Safety harness - Running tests in the production environment can catch bugs before deploying new code


your code is live on the internet!
yes
developer uploads latest version
bot runs all tests
tests pass?
developer

WHY TEST?

Change the way you think about and write your code (for the better)

WHY TEST?

Documentation - Good tests describe how your code behaves in a readable way

```
When /I sign in/ do
  within("#sign_in_form") do
    fill_in 'Email', with: 'user@example.com'
    fill_in 'Password', with: 'password'
  end
  click_button 'Sign in'
  expect(page).to have_content 'Welcome user@example.com!'
end
```

A GOOD TEST IS ISOLATED

- Focuses on testing one thing
- i.e. a single assertion per unit test

```
### BAD
def test_colour_blue_has_right_colour_level
  blue = Colour.new('#0000ff')

  red_level = blue.rgb['r']
  green_level = blue.rgb['g']
  blue_level = blue.rgb['b']

  assert_equal(red_level, 0)
  assert_equal(green_level, 0)
  assert_equal(blue_level, 255)
end

### GOOD
def test_colour_blue_has_right_red_level
  blue = Colour.new('#0000ff')
  red_level = blue.rgb['r']
  assert_equal(red_level, 0)
end

def test_colour_blue_has_right_green_level
  blue = Colour.new('#0000ff')
  green_level = blue.rgb['g']
  assert_equal(green_level, 0)
end

def test_colour_blue_has_right_blue_level
  blue = Colour.new('#0000ff')
  blue_level = blue.rgb['b']
  assert_equal(blue_level, 255)
end
```

MENTALITY OF WRITING A TEST

- Arrange, act, assert

  or

- Given, when, then

  or

- Setup, exercise, verify(, teardown)

```ruby
class Calculator
  def new(array_of_numbers)
    @array_of_numbers = array_of_numbers
  end

  def average
    # code that calculates the average of the numbers in @array_of_numbers
  end
end

class TestAverage < MiniTest::Test

  def test_average
    # arrange
    my_calculator = Calculator.new([1,2,3]) # initialize an instance of the class

    # act
    average = my_calculator.average # call the method on the instance

    # assert
    assert_equal(2, average) # check the result
  end
end
```

```ruby
# arrange!
def setup
  @contact = Contact.create('Grace', 'Hopper', 'grace@hopper.com', 'computer scientist')
end
...
def test_find
  # arrange happened in setup

  # act

  actual_value = Contact.find(@contact.id)

  # assert
  expected_value = @contact

  assert_equal(expected_value, actual_value)
end
```

ARRANGE

- check for unrelated reasons a test could fail

```
# BAD
def test_user_is_invalid_without_name
  # arrange
  user = User.new

  # act
  is_valid = user.valid?

  # assert
  assert_equal(is_valid, false)
end
```

```
# GOOD
def test_user_is_invalid_without_name
  # arrange
  user = User.new(email: "me@gmail.com")

  # act
  is_valid = user.valid?

  # assert
  assert_equal(is_valid, false)
end
```

ASSERT

- hard-code your expected results, don't generate them

```
# BAD
def test_f_to_c
  # act
  actual_value = f_to_c(50)

  # assert
  assert_equal(f_to_c(50), actual_value)
end
```

```
# GOOD
def test_average
  # act
  actual_value = f_to_c(50)

  # assert
  assert_equal(10, actual_value)
end
```

ASSERT

- make your assertions as specific as possible

```
# BAD
def test_user_is_invalid_without_name
  # arrange
  user = User.new(email: "me@gmail.com")

  # act
  is_valid = user.valid?

  # assert
  assert_equal(is_valid, false)
end
```

```
# GOOD
def test_user_is_invalid_without_name
  # arrange
  user = User.new(email: "me@gmail.com")

  # act
  user.save

  # assert
  expected = ["Name can't be blank"]
  actual = user.errors.full_messages
  assert_equal(expected, actual)
end
```

TESTING SIDE-EFFECTS VS. RETURN VALUES

```
# arrange!
def setup
  @contact = Contact.create('Grace', 'Hopper', 'grace@hopper.com', 'computer scientist')
end
...
def test_delete
  # arrange happened in setup

  # act
  @contact.delete

  # assert
  actual_value = Contact.all
  expected_value = []

  assert_equal(expected_value, actual_value)
end
```

```ruby
# arrange!
def setup
  @contact = Contact.create('Grace', 'Hopper', 'grace@hopper.com', 'computer scientist')
end
...
def test_update
  # act
  @contact.update('note', 'wrote the first compiler in 1952')


  # assert
  actual_value = @contact.note
  expected_value = 'wrote the first compiler in 1952'

  assert_equal(expected_value, actual_value)
end
```