Developer Testing

Chantri Polprasert

Readings

- Steve McConnell, Code Complete: A Practical Handbook of Software Construction, 2nd edition, Microsoft Press, 2004
- 2020. Available at http://guides.rubyonrails.org/testing.html.- Matt Wynne and Asklak Hellesøy, The Cucumber Book: Behaviour-Driven
- Development for Testers and Developers, Pragmatic Programmers, 2012.- Paul Duvall, Steve Matyas, and Andrew Glover, Continuous Integration:
- Improving Software Quality and Reducing Risk, Addison-Wesley, 2008.- Andrew Hunt and David Thomas, The Pragmatic Programmer: From
- Journeyman to Master, Pragmatic Bookshelf, 2009.
- https://guides.rubyonrails.org/testing.html

Outline

- Introduction
- Testing methodology
- Test-first vs Test-Driven Development
- Unit and integration testing in Rails

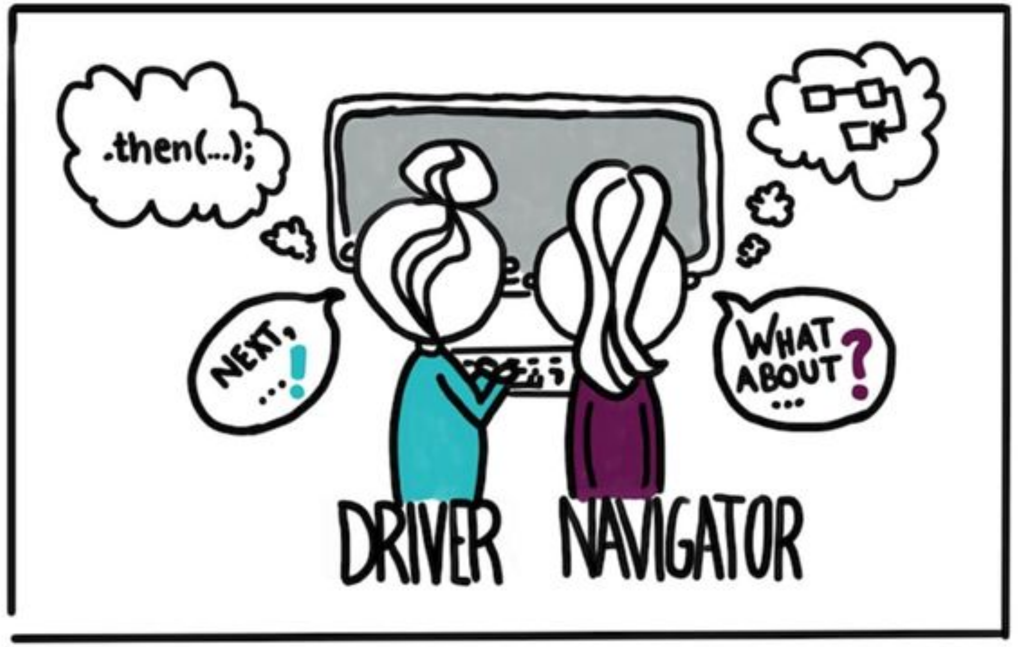
Introduction

- Full stack applications are like any other type of software: Quality assurance processes and tools are critical.
- Here we ask: how can we keep quality high during full stack application development?
- This CANNOT BE STRESSED ENOUGH: a rigorous developer testing methodology is the single most effective technique for delivering high quality applications on time and on budget.

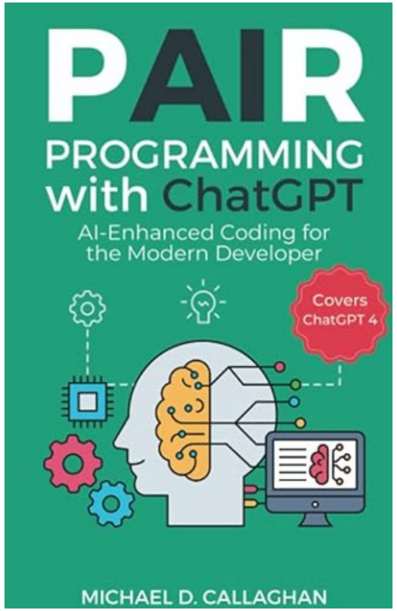
Introduction

- According to McConnell, the main techniques for uncovering defects during development are collaborative construction and developer testing.
- Collaborative techniques like pair programming, formal code inspections, and walkthroughs can have a huge positive effect on software quality, and tend to detect defects when they are less costly to fix.
- Extreme programming: an Agile project management methodology that targets speed and simplicity with short development cycles and less documentation.
- Other forms of testing like stress testing and usability testing take place separately from development and are performed by specialized test personnel.
- Those topics are for another class, though! We will focus on developer testing.

Introduction: pair programming



https://dev.to/documatic/pair-programming-best-practices-and-tools-154j)



https://www.amazon.com/Pair-Programming-ChatGPT-AI-Enhanced-Developer/dp/B0C5G9ZMZX

Introduction: Quality improvement during development

- So, we want to understand how MPAs, SPAs, and APIs can and should be tested during construction, by developers, using white box/glass box techniques.
- I say it again, as it CANNOT BE STRESSED ENOUGH: a rigorous developer testing methodology is the single most effective technique for delivering high quality applications on time and on budget.

Outline

- Introduction
- Testing methodology
- Test-first vs Test-Driven Development
- Unit and integration testing in Rails

Testing methodology: Types of developer testing

According to McConnell, the developer testing methods are:

- Unit: testing small units of software written by a single programmer or team in isolation from the complete system
- Component: execution of a module involving the work of multiple programmers
- Integration: combined execution of two or more classes, packages, components, or subsystems
- System: execution of the production system integrated with other software and hardware systems. It tests for security, performance, resource loss, timing problems, and other issues that can’t be tested at lower levels of integration.
- Regression: repeating previously executed test cases to find defects (normally run after every change).
- Acceptance testing: the final stage of software testing, conducted by the end-user or client, to verify that a system meets business and functional requirements before its final release. (UAT, BDD)

Not everyone agrees on these names, but it is fairly complete.

Level	Scope / Goal	Typical Tools	When to Run
Unit Test	Verify correctness of smallest code units (functions, classes)	JUnit, pytest, NUnit, Minitest	Every code commit
Component Test	Verify that a self-contained functional unit (component) — such as a controller action, or view component — works correctly on its own, but possibly with its immediate dependencies mocked.	RSpec, React Testing Library, Enzyme, Angular TestBed	After each feature implementation
Integration Test	Validate interactions between modules/services	Testcontainers, Postman, SuperTest	Before merging to main branch
System Test	Validate complete system behavior end-to-end	Selenium, Cypress, Playwright	Pre-release testing
Acceptance Test	Confirm system meets user requirements (UAT, BDD)	Cucumber, Behave, Robot Framework	Before deployment

Testing methodology: Another view

Another take: from The Pragmatic Programmer, types of testing are:

- Unit: testing individual modules
- Integration: unit testing of larger subsystems integrating more than one module
- Validation and verification: checking that the system is what users need
- Resource exhaustion, errors, and recovery: checking behavior when the system is out of memory, etc.
- Performance: testing behavior under load
- Usability: testing with real users under real conditions

According to these guys, almost all types of testing can and should be done as regression tests.

Testing categories

- **White box (glass box)**: examine the internal structures or workings of an application. Require a deep understanding of the source code and design.
    - Internal Perspective
    - Test cases designed based on the internal workings of the application
    - Automation Potential
    - Types of testing: Unit, Integration, System
- **Black box**: evaluate the functionality of an application without any knowledge of its internal code structure or implementation
    - User-Centric
    - No Internal Knowledge Required:
    - Focus on Functionality
    - Types of testing: Functional, Non-Functional, Regression, User-Acceptance
- **Grey box:** combine elements of both black box and white box testing

Testing methodology: Testing tips

Some important tips for making testing central to the development of your application:

- Get a production build up and running on a test server early on in the project, give your stakeholders access to get some feedbacks.
- Create an automated regression test suite and keep adding to it as defects are found.
- Use a continuous integration tool such as Jenkins or GitLab CI to automatically run the test suite everytime the code is pushe, and generate quality metric reports.
- Add appropriate tests as you code, and ensure the regression test suite passes before every commit to the central repository.
- Use an issue tracker such as Trac, Redmine, or Github.
- When fixing a bug, first create a failing test case for that bug.
- Last but not least: test ruthlessly (see Pragmatic Programmer): more on next page.

Testing methodology: Testing tips

- When testing your code, remember: you must hope to find errors in yourcode!

  From Hunt and Thomas's The Pragmatic Programmer: "Most developers hate testing. They tend to test gently, subconsciously knowing where the code will break and avoiding the weak spots. Pragmatic Programmers are different. We are driven to find our bugs now, so we don't have to endure the shame of others finding our bugs later."

- Test Early. Test Often. Test Automatically.
- Elaborate test plans are less useful than suites of automated tests built gradually over time.
- Coding Ain't Done 'Til All the Tests Run

Testing methodology: Testing mentality

Along the same lines, according to McConnell:

- Developers like to write clean test cases that verify expected behavior under normal circumstances.
- Mature testers write 5 dirty tests for every clean test.

To be good developers, we must also be mature (or “ruthless”) testers.

For example, for a function to calculate factorial (n!), write 5 dirty test cases of this function:

Clean vs Dirty Test Cases

Aspect	Clean Test	Dirty Test
Definition	Uses valid, expected, well-formed inputs to verify correct functionality.	Uses invalid, unexpected, or malicious inputs to test robustness and error handling.
Goal	Ensure the function works correctly for normal use cases.	Ensure the system behaves safely and predictably under invalid or extreme conditions.
Focus	Correctness and accuracy of expected output.	Stability, security, and error management.
Example (Factorial)	`factorial(5)` → returns `120`.	`factorial(-3)` → raises `ValueError`.
Example (Login)	username=`"admin"`, password=`"1234"` → success.	username=`"admin' OR '1'='1"`, password=`"anything"` → should fail (SQL injection).
Testing Type	Functional or positive testing.	Negative or robustness testing.
Outcome When Fails	Function logic error.	Input validation or security flaw.

Testing methodology: Test first

- Whenever possible, use a test first methodology.
- Write test cases before writing the code that makes the test pass.
- Benefits:
    - Writing test cases before coding helps you to think about the design (what the app should do) before starting coding.
    - Writing test cases before coding helps detect/refine poorly-specified requirements.
    - Writing test cases before coding helps you detect defects sooner than you would otherwise.
- Finding defects early = Saving time + saving money

Testing methodology: Basis testing

- At the very least, we try to test every line of code. Code coverage tools will tell us our test coverage.
- Basis testing goes further, trying to more fully exercise every line of code by analyzing the branches and loops.
- We construct a case for the straight-through run, then a case for each branch/loop.
- For if statements, we try to test all possible combinations of the predicates (inputs).

SimpleCov gem version 0.22.0 stable no status maintainability A docs

Testing methodology: Coverage analysis

- In basis testing we aim to ensure complete coverage of our source code base.
- To ensure good coverage as your source code base grows, use the coverage tools for your framework during regression testing.

For Ruby/Rails, try the simplecov gem using the following in your Gemfile:

```
group :test do
  gem 'simplecov', :require => false
end
```

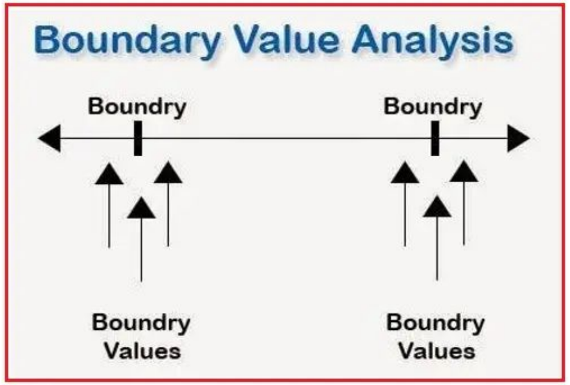
- Note that coverage is a minimal requirement however!
- You also need data analysis.

In test/test helper.rb

```
require 'simplecov'
SimpleCov.start 'rails'

ENV['RAILS_ENV'] ||= 'test'
require_relative '../config/environment'
require 'rails/test_help'

class ActiveSupport::TestCase
  fixtures :all
end
```

Testing methodology: Boundary testing

- Software testing technique focused on identifying errors at the edges of input ranges.
- This method is based on the observation that defects are more likely to occur at the boundaries than within the ranges themselves.
- Try to create test cases for things that are off-by-one.
- More generally, try to come up with all of the kinds of bad data your system might encounter.
- **Exercise**: Provide examples of the boundary test cases for an input that accepts values from 0 to 100



Testing methodology: More tips

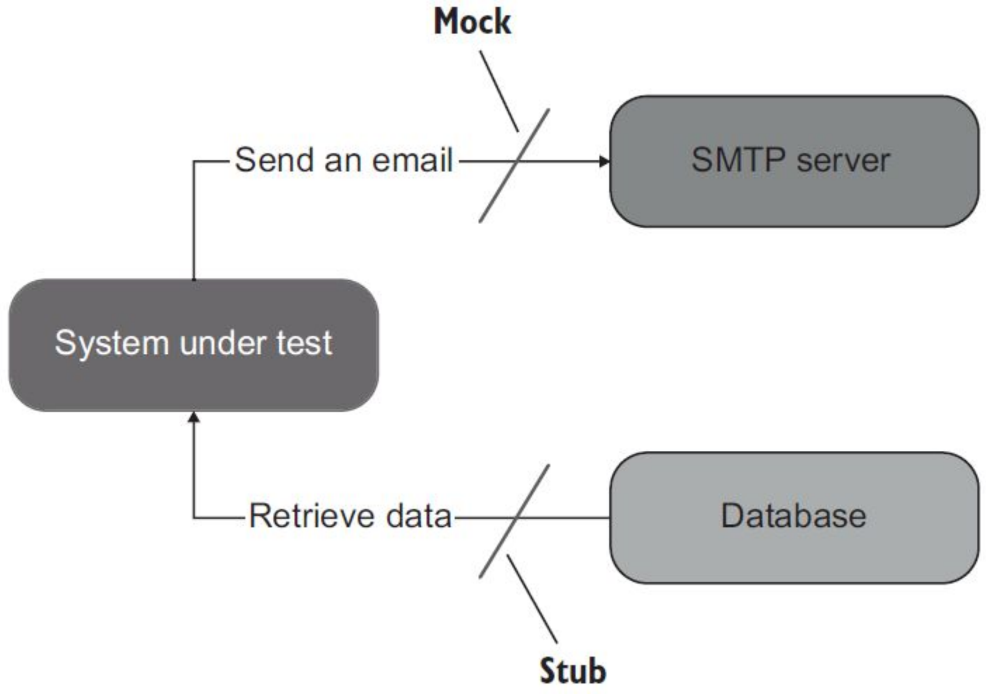
Some more testing tips:

- Learn to use your test framework effectively.
  - Ruby: minitest, RSpec, etc.
  - BDD: Cucumber, etc.
  - JavaScript: PhantomJS, Puppeteer, etc.
- The framework(s) to use depend on your technology stack, past experience, etc.
- For 100% coverage, you will normally need test doubles (mocks and stubs) to simulate responses of external systems or environmental conditions.
- Consider using random data generators to ensure robustness over different values.

Mocks and Stubs test

- Types of test doubles used in unit testing to simulate the behavior or state of real objects or components
- Stubs
  - Provide fixed responses to method calls. No logic. (Return fake data from DB query)
  - Used for state verification and provide predetermined responses to method calls, allowing you to test specific scenarios.
  - Help to emulate incoming interactions
- Mocks
  - Used for behavior verification, ensuring that the correct interactions occur between the tested unit and its dependencies.
  - Used when you want to verify specific interactions with dependencies and ensure that the unit being tested acts correctly in response.
  - Help to emulate and examine outcoming interactions

Mocks and Stubs test



Unit Testing Principles, Practices, and Patterns - Manning

- Sending an email is an outcoming interaction: an interaction that results in a side effect in the SMTP server
- A test double emulating such an interaction is a mock
- Retrieving data from the database is an incoming interaction; it doesn’t result in a side effect. The corresponding test double is a stub.

Exercise

Write test cases that cover every line of codes

```
Start
Input A, B, C
If (A > 5 and B < 10) or (C == 3) then
  If A % 2 == 0 then
    Print "Even A"
  Else If B % 3 == 0 then
    Print "Divisible by 3"
  Else
    Print "Other case"
  End If
Else If A < 0 and B > 0 then
  Print "Negative A, Positive B"
Else
  Print "None apply"
End If
End
```

Outline

- Introduction
- Testing methodology
- Test-first vs Test-Driven Development
- Unit and integration testing in Rails

Test-First vs. Test-Driven Development: Test-First

- In the industry you will find some confusion in terms.
- Test-first simply means we write tests before we write code.
- Sometimes test-first is called “red-green development”.
- It is equally applicable to any kind of test (a small unit, a big integration test, a UAT, and so on).
- TDD (and BDD, to be discussed later) are both test-first methodologies but more specific

Test-First vs. Test-Driven Development: Test-Driven Development

- Test-Driven Development (TDD) emerged from Kent Beck’s XP discipline.
- TDD is sometimes called “red-green-refactor” development.
- TDD is a very specific methodology for small units:
    a. Add a test
    b. Run all tests and see if the new one fails
    c. Write some code
    d. Run tests [repeat from c. until all tests pass]
    e. Refactor code
    f. Repeat from a.

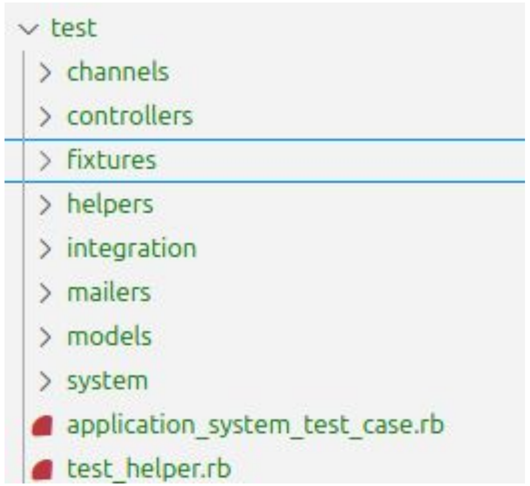
We won’t get into TDD much but we’ll see how BDD inherits from TDD and has changed the way we do automated user acceptance testing.

Outline

- Introduction
- Testing methodology
- Test-first vs Test-Driven Development
- Unit and integration testing in Rails

Unit and integration testing in Rails: Regression testing

- Recall The Pragmatic Programmer's admonition that all tests should be regression tests.
- Creating an automated regression test suite is easy with Rails' testing features.
  - Model tests are unit tests for ActiveRecord model classes.
  - Controller tests perform functional testing of each controller action.
  - Integration tests verify that longer sequences of request-response pairs behave correctly.
  - There are specialized tests for Web Socket functionality (channel tests), asynchronous task handling (job tests), and email gateways (mailer tests).
  - System test: full browser testing



```
test
  channels
  controllers
  fixtures
  helpers
  integration
  mailers
  models
  system
  application_system_test_case.rb
  test_helper.rb
```

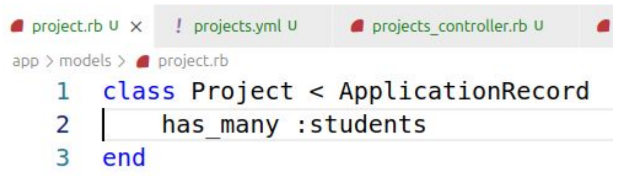
- Fixtures are a way of organizing test data
- The test_helper.rb: holds the default configuration for your tests.
- application_system_test_case.rb holds the default configuration for your system tests.

Controller testing (test/controllers)

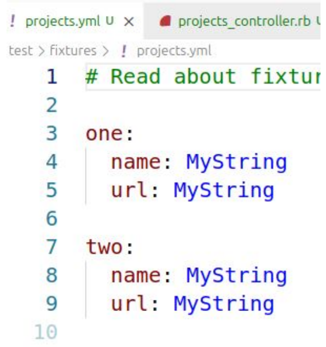
```ruby
class ProjectsControllerTest < ActionDispatch::IntegrationTest
  setup do
    @project = projects(:one)
  end

  test "should get index" do
    get projects_url
    assert_response :success
  end
```

Model testing

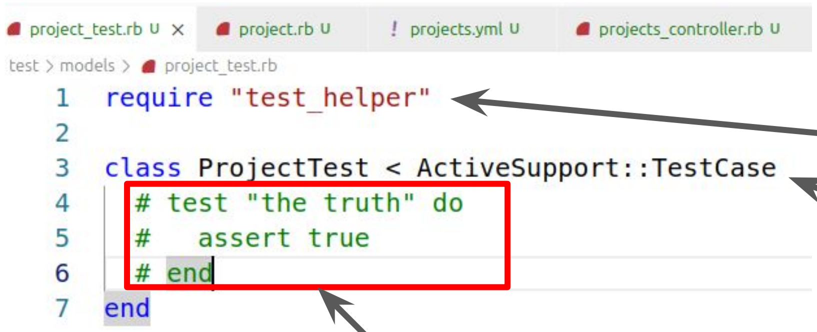
- When the model is created (rails g model project name:string, url:string), test stubs are created in the test directory:
    - app/models/project.rb
    - test/models/project_test.rb
    - test/fixtures/projects.yml



```ruby
class Project < ApplicationRecord
    has_many :students
end
```



```yaml
# Read about fixtur

one:
  name: MyString
  url: MyString

two:
  name: MyString
  url: MyString

```



```ruby
require "test_helper"

class ProjectTest < ActiveSupport::TestCase
  # test "the truth" do
  #   assert true
  # end
end
```

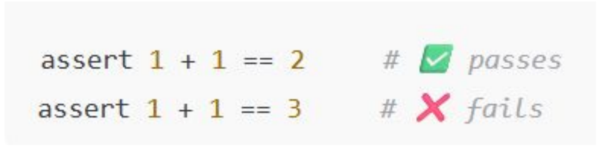
Load default configuration to run the test (included in all the tests)

The ProjectTest class defines a test case because it inherits from ActiveSupport::TestCase

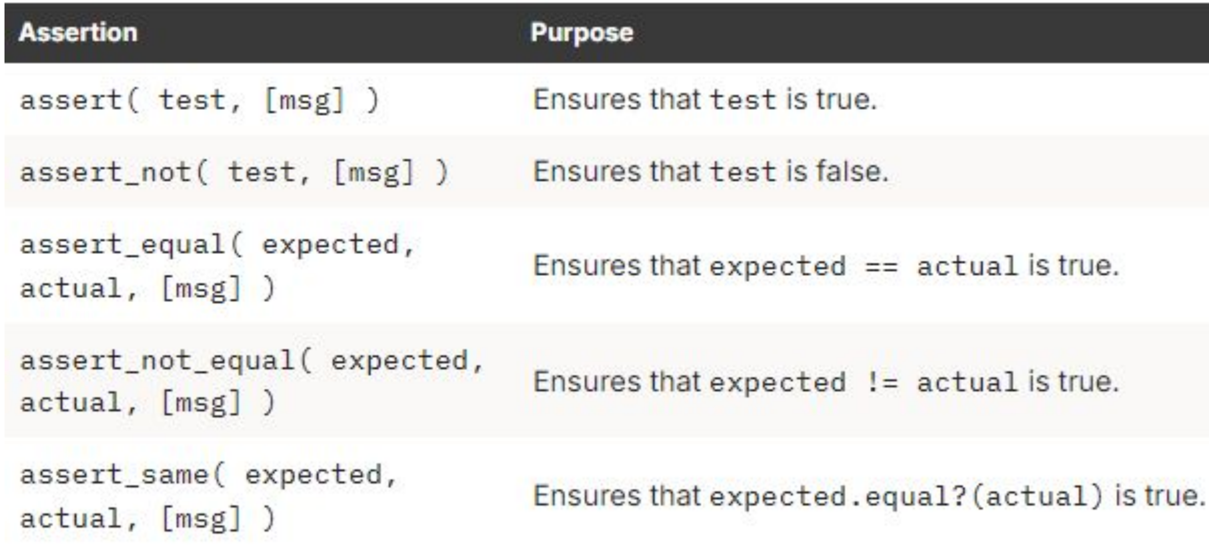
A test method (test name and a block)

Assertions

- Key component of the testing framework in Rails
- Used to validate that the code behaves as expected.
- assert is a statement that checks if a condition is true.
  - If the condition is true → ✅ test passes
  - If the condition is false → ❌ test fails



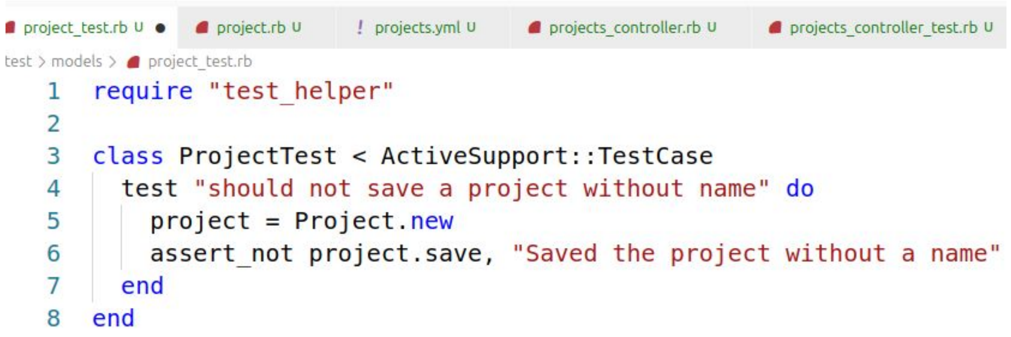
```
assert 1 + 1 == 2      # ✅ passes
assert 1 + 1 == 3      # ❌ fails
```



Assertion	Purpose
`assert( test, [msg] )`	Ensures that `test` is true.
`assert_not( test, [msg] )`	Ensures that `test` is false.
`assert_equal( expected, actual, [msg] )`	Ensures that `expected == actual` is true.
`assert_not_equal( expected, actual, [msg] )`	Ensures that `expected != actual` is true.
`assert_same( expected, actual, [msg] )`	Ensures that `expected.equal?(actual)` is true.

First Failing Test

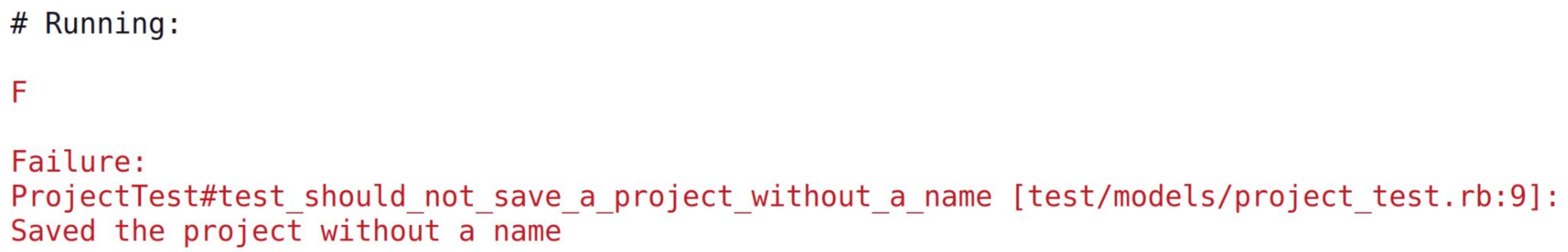
- Add a failing test case: (In the studentdb project, a project without name cannot be saved)
- Try this in project_test.rb and run ‘rails test test/models/project_test.rb



```ruby
require "test_helper"

class ProjectTest < ActiveSupport::TestCase
  test "should not save a project without name" do
    project = Project.new
    assert_not project.save, "Saved the project without a name"
  end
end
```

Pass when the result is nil or false

First Failing Test



```
# Running:

F

Failure:
ProjectTest#test_should_not_save_a_project_without_a_name [test/models/project_test.rb:9]:
Saved the project without a name
```

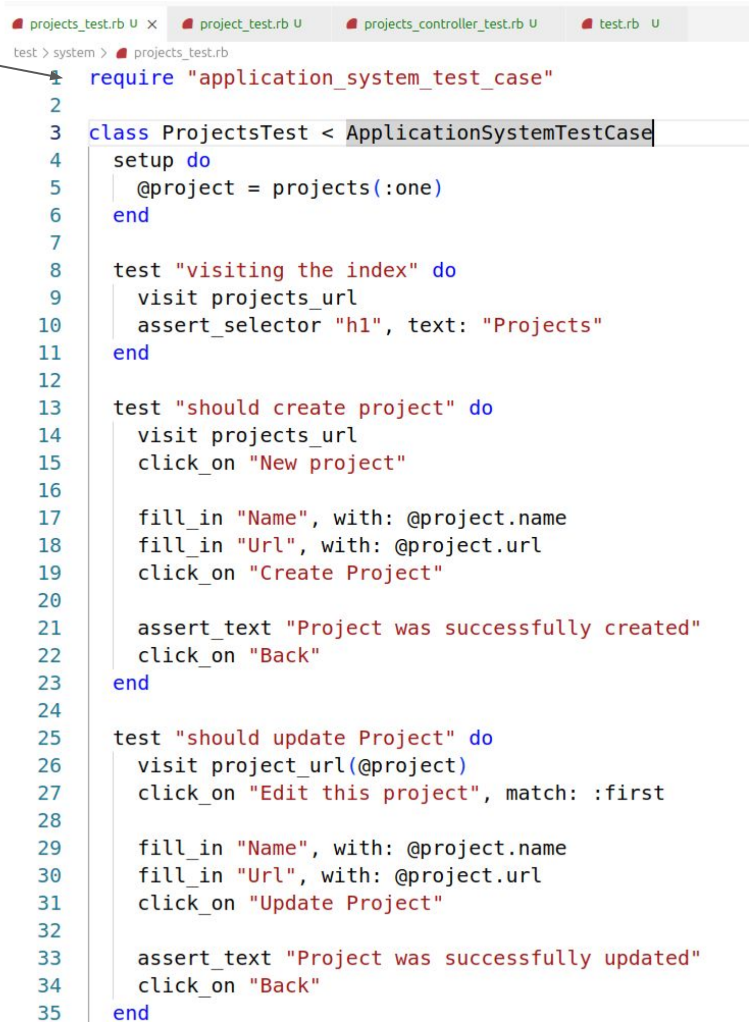
- How to solve this? (How to make the test false?)
- Approach: we first wrote a test which fails for a desired functionality, then we wrote some code which adds the functionality and finally we ensured that our test passes. This approach to software development is referred to as Test-Driven Development (TDD).

Loads Rails' system test setup (which uses Capybara).

System testing

- Test user interactions with your application, running tests in either a real or a headless browser.
- Use 'test/system' directory
- Each file typically tests a complete flow for one feature or resource.
- System tests use Capybara under the hood.

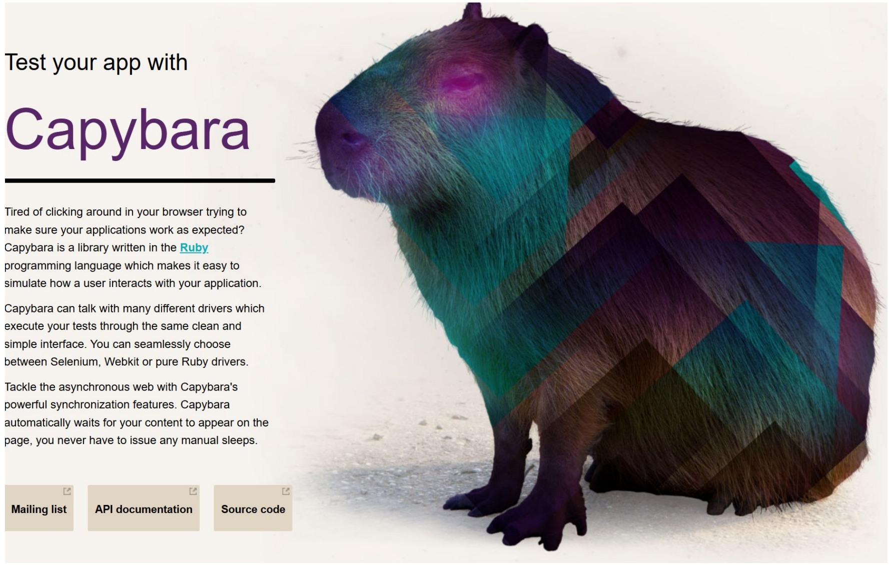
Capybara is a Ruby library that lets you simulate how a real user interacts with your web app — by visiting pages, clicking links, filling out forms, and checking what's visible on the screen.


projects_test.rb U
project_test.rb U
projects_controller_test.rb U
test.rb U
test > system > projects_test.rb

```
require "application_system_test_case"

class ProjectsTest < ApplicationSystemTestCase
  setup do
    @project = projects(:one)
  end

  test "visiting the index" do
    visit projects_url
    assert_selector "h1", text: "Projects"
  end

  test "should create project" do
    visit projects_url
    click_on "New project"

    fill_in "Name", with: @project.name
    fill_in "Url", with: @project.url
    click_on "Create Project"

    assert_text "Project was successfully created"
    click_on "Back"
  end

  test "should update Project" do
    visit project_url(@project)
    click_on "Edit this project", match: :first

    fill_in "Name", with: @project.name
    fill_in "Url", with: @project.url
    click_on "Update Project"

    assert_text "Project was successfully updated"
    click_on "Back"
  end
```

Capybara

- A popular testing framework in Ruby designed for automating web application testing.
- It simulates user interactions, allowing developers to write acceptance tests that verify the functionality of their applications.



Test your app with

Capybara

Tired of clicking around in your browser trying to make sure your applications work as expected? Capybara is a library written in the Ruby programming language which makes it easy to simulate how a user interacts with your application.

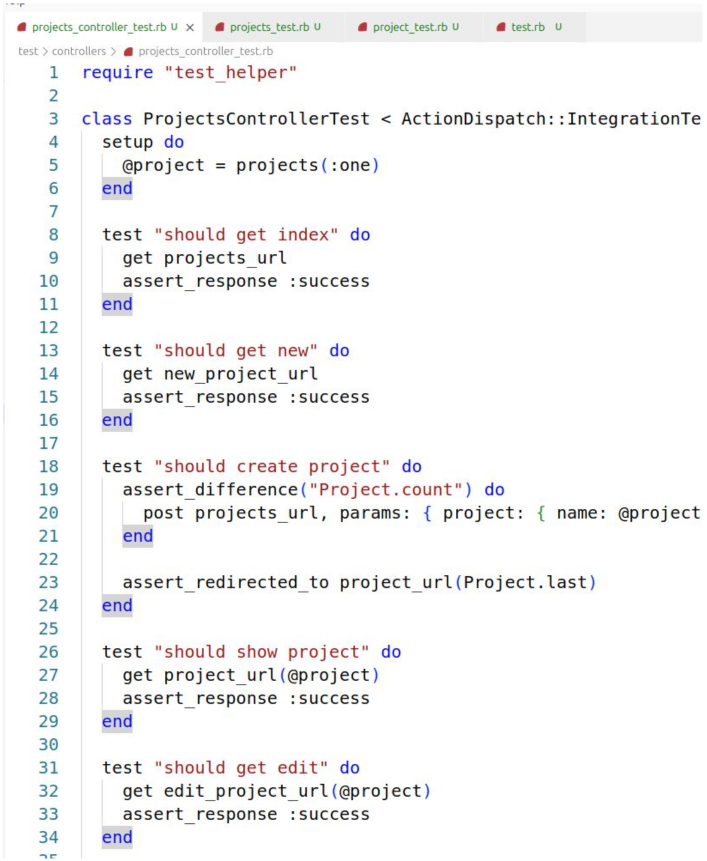
Capybara can talk with many different drivers which execute your tests through the same clean and simple interface. You can seamlessly choose between Selenium, Webkit or pure Ruby drivers.

Tackle the asynchronous web with Capybara's powerful synchronization features. Capybara automatically waits for your content to appear on the page, you never have to issue any manual sleeps.

Capybara (teamcapybara.github.io)

Functional Tests for Your Controllers

- Testing the various actions of a controller is a form of writing functional tests
- Your controllers handle the incoming web requests to your application and eventually respond with a rendered view.
- When writing functional tests, you are testing how your actions handle the requests and the expected result or response, in some cases an HTML view.
- You should test for things such as: (was the web request successful?, was the user redirected to the right page? was the user successfully authenticated?)
- The easiest way to see functional tests in action is to generate a controller using the scaffold generator:


projects_controller_test.rb U
projects_test.rb U
project_test.rb U
test.rb U
test > controllers > projects_controller_test.rb

```ruby
require "test_helper"

class ProjectsControllerTest < ActionDispatch::IntegrationTe
  setup do
    @project = projects(:one)
  end

  test "should get index" do
    get projects_url
    assert_response :success
  end

  test "should get new" do
    get new_project_url
    assert_response :success
  end

  test "should create project" do
    assert_difference("Project.count") do
      post projects_url, params: { project: { name: @project
    end

    assert_redirected_to project_url(Project.last)
  end

  test "should show project" do
    get project_url(@project)
    assert_response :success
  end

  test "should get edit" do
    get edit_project_url(@project)
    assert_response :success
  end
```

Conclusions

- Testing by the developer is a key part of a full testing strategy.
- Writing test cases before the code takes the same amount of time and effort as writing the test cases after the code, but it shortens defect-detection-debug-correction cycles.
- You can generate many test cases deterministically by using basis testing, data flow analysis, boundary analysis, classes of bad data, and classes of good data. You can generate additional test cases with error guessing.
- Automated testing is useful in general and is essential for regression testing
- In the long run, the best way to improve your testing process is to make it regular, measure it, and use what you learn to improve it.