Full-stack webapp testing with Selenium and Rails

- Alex Chaffee
- Brian Takita

PIVOTAL LABS

Abstract

- Want to test your entire Web 2.0 app, from AJAX and DHTML through browser-rendered HTML into a live instance of your application and database? Most web testing frameworks have gaps in their coverage: JUnit and Test::Unit miss the client frontend; JSUnit misses the server backend; web testing frameworks miss some or all of the JavaScript. With Selenium we have a framework that can test the whole application, from browser-executed JavaScript, through a live application backend, then back to assertions on browser-rendered DOM code. Selenium RC takes this further: since you write your tests in your application language, your tests can do data setup and assertions based directly on server-side domain objects that may be inaccessible or only partially accessible from the client side. On our teams we have used and developed a series of helper methods and assertions that allow testing of AJAX and DHTML functions as well.

  In this tutorial we outline the architecture of Selenium RC and walk through code and examples illustrating how to do full-stack testing against a Ruby on Rails application.

PIVOTAL LABS

Selenium Features

- Selenium runs in the browser
- Executes your app in a frame
- Simulates user actions via JavaScript
- Goes all the way to the server and back
- Complementary to JSUnit
    - JSUnit tests JS pages only, not interaction with server
- Fun to watch

First Example

- Basic example (login_test.rb)
  - ```
    def test_login
      open "/"
      assert_text_present("Please sign in")

      type "username", "jdoe"
      type "password", "password"
      click "submit"
      assert_text_present("Welcome, John Doe")

      open "/"
      assert_text_present("Welcome, John Doe")
    end
    ```
- Sort of a DSL ("Selenese")
- Written in Ruby
  - Can also use Java, .NET, Python, Perl, etc.

PIVOTAL LABS
    ```

Accessing Your World

- Can access database and/or domain objects
  - ```
    def test_signup
      open "/signup"
      assert_text_present("Please sign up")
      type "username", "noob"
      type "fullname", "New Bie"
      type "password", "secret"
      type "password_verify", "secret"
      click "submit"
      assert_text_present("Welcome, New Bie")

      user = User.find_by_username("noob")
      assert user.authenticate("secret")
    end
    ```


    ```

Or even email!

```
def test_signup_via_email
  open "/signup"
  assert_text_present("Please sign up")
  type "email", "noob@example.com"
  click "submit"
  assert_text_present("An email has been sent to noob@example.com")
  wait_for { !ActionMailer::Base.deliveries.empty? }
  user = User.find_by_email("noob@example.com”)
  assert !user.verified?
  link = extract_link(ActionMailer::Base.deliveries[0].body)
  open link
  wait_for do
    user.reload
    user.verified?
  end
end

def extract_link(s)
 /http:\/\/\S*/.match(s)[0]
end
```

Or even email!

```
def test_signup_via_email
  open "/signup"
  assert_text_present("Please sign up")
  type "email", "noob@example.com"
  click "submit"
  assert_text_present("An email has been sent to noob@example.com")
  wait_for { !ActionMailer::Base.deliveries.empty? }
  user = User.find_by_email("noob@example.com”)
  assert !user.verified?
  link = extract_link(ActionMailer::Base.deliveries[0].body)
  open link
  wait_for do
    user.reload
    user.verified?
  end
end

def extract_link(s)
 /http:\/\/\S*/.match(s)[0]
end
```

PIVOTAL LABS

Or even email!

```
def test_signup_via_email
  open "/signup"
  assert_text_present("Please sign up")
  type "email", "noob@example.com"
  click "submit"
  assert_text_present("An email has been sent to noob@example.com")
  wait_for { !ActionMailer::Base.deliveries.empty? }
  user = User.find_by_email("noob@example.com”)
  assert !user.verified?
  link = extract_link(ActionMailer::Base.deliveries[0].body)
  open link
  wait_for do
    user.reload
    user.verified?
  end
end

def extract_link(s)
 /http:\/\/\S*/.match(s)[0]
end
```

Or even email!

```
def test_signup_via_email
  open "/signup"
  assert_text_present("Please sign up")
  type "email", "noob@example.com"
  click "submit"
  assert_text_present("An email has been sent to noob@example.com")
  wait_for { !ActionMailer::Base.deliveries.empty? }
  user = User.find_by_email("noob@example.com")
  assert !user.verified?
  link = extract_link(ActionMailer::Base.deliveries[0].body)
  open link
  wait_for do
    user.reload
    user.verified?
  end
end

def extract_link(s)
 /http:\/\/\S*/.match(s)[0]
end
```

PIVOTAL LABS

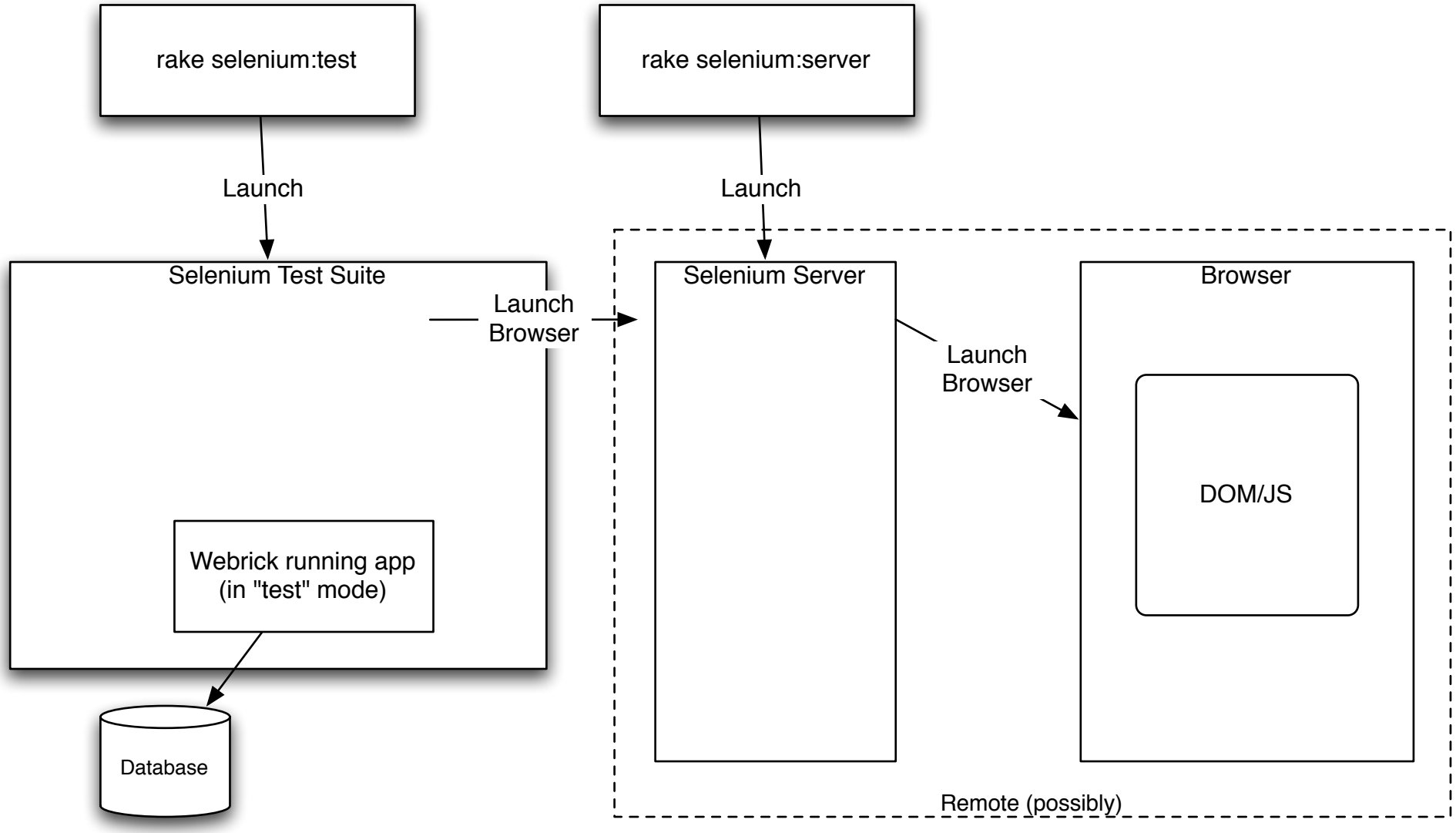
Or even email!

```
def test_signup_via_email
  open "/signup"
  assert_text_present("Please sign up")
  type "email", "noob@example.com"
  click "submit"
  assert_text_present("An email has been sent to noob@example.com")
  wait_for { !ActionMailer::Base.deliveries.empty? }
  user = User.find_by_email("noob@example.com”)
  assert !user.verified?
  link = extract_link(ActionMailer::Base.deliveries[0].body)
  open link
  wait_for do
    user.reload
    user.verified?
  end
end

def extract_link(s)
 /http:\/\/\S*/.match(s)[0]
end
```

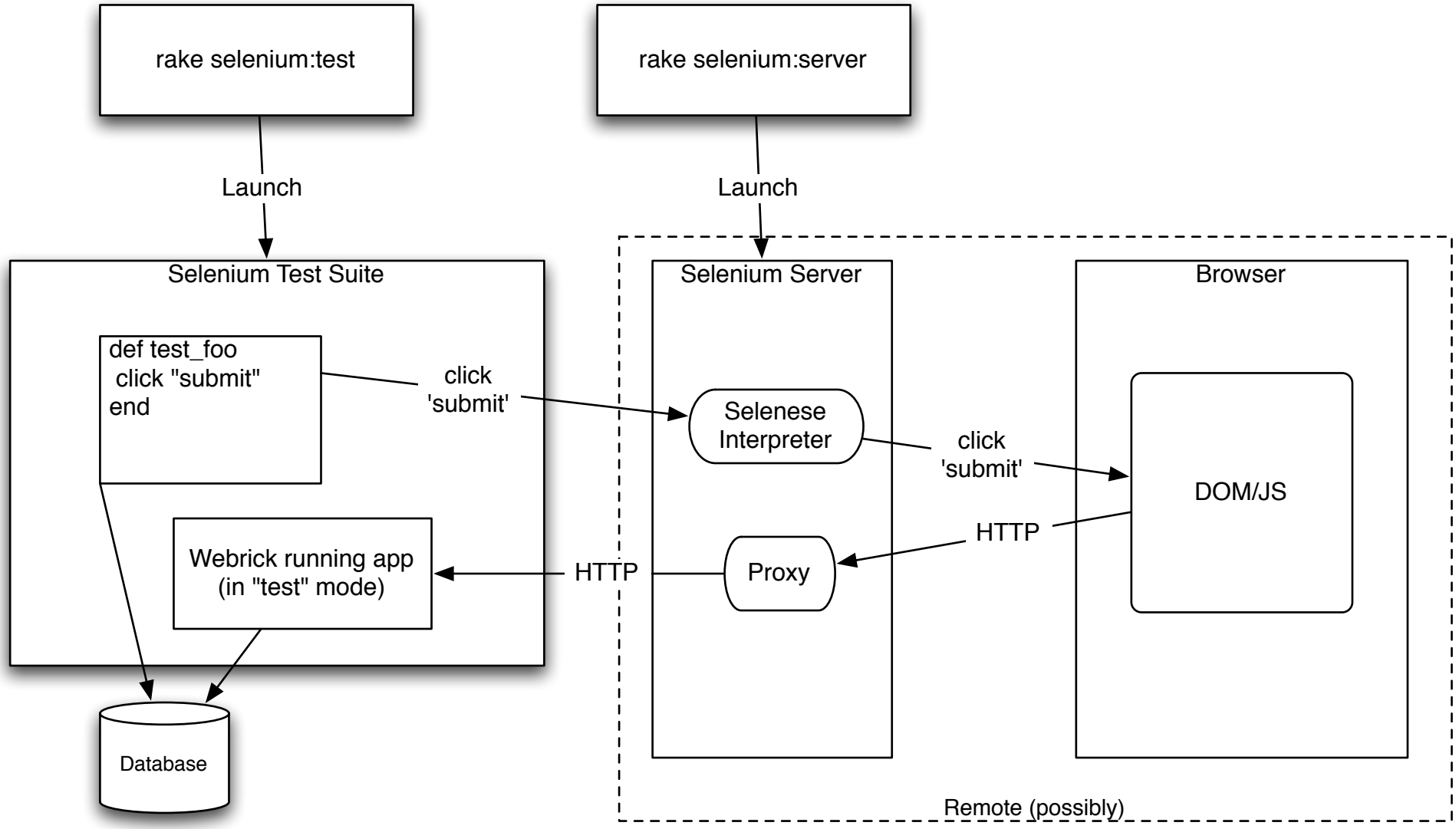
And don’t forget to extract a DSL

```
def login(user_name,password)
  open('/login')
  type "id=login", user_name
  type "id=password", password
  click_and_wait "commit"
end

def logout
  open('/logout')
end
```

Selenium RC Architecture: launch



Selenium RC Architecture: run test



Multiple Platform Support

- OS (Unix, Windows, Mac)
- Browser (IE, Firefox, Safari)
- The Selenium Farm (Continuous Integration)

Good Things Come To Those Who Wait

- Web 2.0 is Asynchronous and Dynamic
  - The “A” in "AJAX"
  - The "D" in "DHTML"
- The problem:
  - Since they happen unpredicatably, and inside the context of a single page, they’re harder to test

Good Things Come To Those Who Wait

- wait_for function
  - executes in Ruby
- wait_for_condition action
  - takes a JavaScript expression, evals it in browser
- wait_for_condition is complementary to wait_for
  - JavaScript can't poll DB or domain objects
  - Write your conditions in the language you prefer

More examples

- Tracker
  - project_page_editing_test
- Peer2Patent
  - create_prior_art_test
  - registration_mode_test

What kind of tests?

- Integration Tests
  - Test multiple layers of your app working together
- Acceptance Tests
  - Test specific customer features and/or scenarios
- UI tests
  - Test complicated or detailed UI widgets
- Smoke Tests
  - Special case of Integration Tests
  - Make sure your app doesn't blow up

Wait, do you mean Integration Tests or Integration Tests?

- Selenium tests are integration tests in the abstract
- Not derived from Rails Integration Tests
- Rails ITs are faster
- Selenium tests are better for testing AJAX, DHTML, widgets, full-stack integration

- Where you can do a test in Ruby instead of Selenium... you should.

When to use Selenium?

- Test full stack
- Test JS->server interaction
- Test emails
- Smoke testing
- Testing in multiple browsers

Particular Strengths of Selenium Tests

- AJAX/RJS tests
  - Test what state the browser is in after the RJS/AJAX call
- JS tests
  - Test JavaScript interacting with your DOM
  - Not just functional/behavioral JS (which JSUnit is great for)

When not to use Selenium?

- For every edge case
  - Want to test happy path, and one or two error paths
- When speed is of the essence
  - e.g. in a suite you run a lot, or when your tests start taking more than a few minutes
  - see next slide...
- When you're testing fine-grained features
  - Selenium tests tend to sprawl out
    - setup/teardown are more expensive, so you want to test more things per test
  - Unit tests should be small and isolated

Slow-lenium

- Selenium is much faster than a human, but still much slower than unit tests. How do we deal with that?
- Fast Suite / Slow Suite
  - Run Fast suite before checkin
  - Run Slow suite in Continuous Integration
- When appropriate, convert Selenium tests into unit tests
  - Consider bootstrapping with Selenium tests, then replacing them with isolated tests once your app is up and running

Locators

- Specify how to locate an HTML element
  - id
    - "id" attribute
    - e.g. id=userName
  - name
    - "name" attribute
    - followed by optional "value" or "index" filter
    - e.g. name=flavor value=chocolate

Locators (cont.)

- identifier
  - first id, then name
- dom
  - JavaScript expression
  - e.g. dom=document.forms['icecreamForm'].flavorDropdown
- xpath
  - e.g. xpath=//a[contains(@href,'#id1')]/@class

Locators (cont.)

- link
  - selects the <A>` element whose Text content matches the given text
  - e.g. link=Click Here To Continue
- css
  - css selector
  - e.g. css=a[href="#id3"]
- Custom Locators
- By default, it's "identifier",
  - or "xpath" if it starts with "//"
  - or "dom" if it starts with "document."

Tips

- use ids when possible
    - XPath makes your tests brittle and dependent on page structure
- use slow mode for debugging
- user_extensions.js

Gotchas

- Disable Transactional Fixtures for Selenium tests
- Reload all fixtures before every test
  - consistent state
  - order-independent
- set allow_concurrency=true
- cookies stay between tests
  - can lead to tests passing when run alone, failing when run together
  - you can eval JS to clear cookies between tests
- Firefox profiles

Test-driving from Selenium

- Some people like it, some don't
- Pro:
  - Top-down
  - When test passes, feature is done
  - Stay close to well-defined customer requirements
- Con:
  - Still vulnerable to edge cases
  - Brittle in the face of UI changes
- You usually iterate through all layers of tests, several times over

Pivotal rake integration

- launches server locally
- runs tests locally and/or remotely

Pivotal Ruby Classes

- selenium_suite.rb
- selenium_helper.rb

Selenium On Rails

- Generates HTML (FIT-style)
- Can't go into your DB or domain
- Any custom test code has to be JavaScript
- Does it work remotely?

Q&A

Experimentation

References

- www.openqa.org—selenium-rc <http://www.openqa.org/selenium-rc/>

Timeline

- 90 minutes with Q&A interspersed