Automated Testing of Web Mapping Applications

Brooks Robertson @brooksjbr
Jeremy Schneider @ymerejredienhcs
Tom Wayson @tomwayson
Anatomy of a web application

- UI/client app
- API/Service Layer
- Backend/Business Logic
The Test Pyramid

- Server response time
- End-to-end integration
- Page rendering
- Longer running
- Connectivity
- Protocol
- CRUD
- Data Integrity
- Fast/Rapid feedback
- Isolated
- Deterministic
- Run locally
- Live with code

Cost
Who writes what and how much?

- Unit
- Integration
- UI
- Performance

- Feature Engineers
- QA Engineers
Where to start?
Front-end Unit and Component Tests

Tom Wayson @tomwayson
Front-end vs Back-end Tests

- Unit
- Integration
- UI
- Performance

Front end vs Back end
Unit tests

Test a "unit" of code in isolation
Primary value of unit tests:
1. drive code structure
2. code is correct
3. catch regression
average web mapping application
4 rules for super unit tests
write high value tests
test only your code
test one thing at a time
refactor ruthlessly
test one thing!
“logic”
“logic”

“action”
“logic”

“action”
“logic with calls”

“action functions”
“action functions”

often (too) easy to test

tests are high-value

when formulating an algorithm

write high value tests!
Don’t Test This Function

```javascript
function add(a, b) {
    return a + b;
}
```
Demo: Test Driving an Algorithm

"Simple" Charts in Open Data

@tomwayson/arcgis-chart-utils
“high-value” tests focus on the “logic”

given a, b, c…

was fn 1 called with a, c?
did fn 3 call fn 4 with c?
testing logic

spies & fakes!
testing logic

spies & fakes!
Don’t Test the Map
map access
Separation of Concerns
(hint: use a framework)
UI components

controllers
map service
(or "map controller")
a facade
map service

mapService.new("Map", ...)
mapService.showLayer(...)
mapService.destroyMap(...)

map
spies & fakes!

spy:
how many times was the fn() called?
what parameters were passed?

fake (a.k.a stub):
replace implementation of a fn() 
can also spy on the fake
spies & fakes!

```javascript
var stub = sinon.stub(mapService, "newMap");
controllerUnderTest.showMap();
assert.ok(stub.calledOnce, 'newMap was called once');
```
Demo: Stubbing a Map Service

```
const stub = this.stub(mapService, 'newMap');
```

@tomwayson/ambitious-arcgis-app
test frameworks
Which is right for me?

Use whatever works with your app framework
Which is right for me?

Use whatever works for you
Which is right for me?

Use whatever
Writing tests makes your code better.
Writing tests makes you code better.
Testable code is more robust.
Testable code is more understandable.
Testable code is more modular.
Testable code is more reusable.
Component Tests
“mini” e2e tests
mimic user interaction
UIComponent
Config
Util
Service
App Infrastructure
API

partial isolation
fake expensive operations (async API calls, etc)
makes this easy
Thank you!
https://github.com/tomwayson/dev-summit-2017-front-end-testing
Top of Testing Pyramid

- Manual Exploratory Testing & Sanity Checking
- Automated UI Tests
- Automated Acceptance Tests
- Automated Integration Tests
- Automated Unit Tests

Cost & Run Time
Browser & Visual Testing

- Selenium/WebDriver
- Screenshot Comparison
Selenium Testing

Human-ish tool to drive browser and verify application behavior

• Hands/Fingers
• Eyes
• Brain
• Voice
Selenium

Hands:

• Navigate to sites – type into URL Bar
• Type stuff in fields
• Move mouse
• Click buttons
Selenium

Eyes:

Look at page (DOM) through small keyhole!
Selenium

Eyes:

Look at page (DOM) through small keyhole!
Selenium

Eyes:

Look at page (DOM) through small keyhole!
Brains/voice – Test Framework:

- RSpec, Cucumber, Junit, Test::Unit, TestNG…
- API bindings in many languages
Basic Selenium Flow

• Get driver
• Go to page
• Find element and interact with it
• Verify expected new page contents
Get driver

```ruby
# run like this: PATH=..:$PATH ruby basic_selenium_example.rb
require 'selenium-webdriver'

# download geckodriver from https://github.com/mozilla/geckodriver/releases
# and put it in this directory.
driver = Selenium::WebDriver.for :firefox, marionette: true

driver.navigate.to "http://google.com"

if /toolsqa.com/.match(driver.page_source)
  puts "YAY!"
else
  puts "BOO!"
end

driver.quit
```
# run like this: PATH=.::$PATH ruby basic_selenium_example.rb
require 'selenium-webdriver'

# download geckodriver from https://github.com/mozilla/geckodriver/releases
# and put it in this directory.
driver = Selenium::WebDriver.for :firefox, marionette: true

driver.navigate.to "http://google.com"

element = driver.find_element({name: 'q'})

element.send_keys "Selenium Tutorials"

element.send_keys :enter

sleep 2 # not recommended to use sleep; we should use an implicit wait

if /toolsqa.com/.match(driver.page_source)
  puts "YAY!"
else
  puts "BOO!"
end

driver.quit
Locate elements

```
# run like this: PATH=..:$PATH ruby basic_selenium_example.rb
require 'selenium-webdriver'

# download geckodriver from https://github.com/mozilla/geckodriver/releases
# and put it in this directory.
driver = Selenium::WebDriver.for :firefox, marionette: true

driver.navigate to "http://google.com"

element = driver.find_element({name: 'q'})

element.send_keys Selenium Tutorials
element.send_keys :enter
sleep 2 # not recommended to use sleep; we should use an implicit wait

if /toolsqa.com/.match(driver.page_source)
  puts "YAY!"
else
  puts "BOO!"
end

driver.quit
```
# run like this: PATH=..:$PATH ruby basic_selenium_example.rb
require 'selenium-webdriver'

driver = Selenium::WebDriver.for :firefox, marionette: true

driver.navigate.to "http://google.com"

element = driver.find_element({name: 'a'})

```
  element.send_keys "Selenium Tutorials"
  element.send_keys :enter
```

sleep 2 # not recommended to use sleep; we should use an implicit wait

if /toolsqa.com/.match(driver.page_source)
  puts "YAY!"
else
  puts "BOO!"
end

driver.quit
# run like this: PATH=.|:$PATH ruby basic_selenium_example.rb
require 'selenium-webdriver'

# download geckodriver from https://github.com/mozilla/geckodriver/releases
# and put it in this directory.
driver = Selenium::WebDriver.for :firefox, marionette: true

driver.navigate.to "http://google.com"

element = driver.find_element({name: 'q'})

# not recommended to use sleep; we should use an implicit wait
sleep 2

element.send_keys "Selenium Tutorials"
element.send_keys :enter

if /toolsqa.com/.match?(driver.page_source)
  puts "YAY!"
else
  puts "BOO!"
end

driver.quit
# run like this: PATH=..:$PATH ruby basic_selenium_example.rb
require 'selenium-webdriver'

# download geckodriver from https://github.com/mozilla/geckodriver/releases
# and put it in this directory.
driver = Selenium::WebDriver.for :firefox, marionette: true

driver.navigate.to "http://google.com"
element = driver.find_element({name: 'q'})
element.send_keys "Selenium Tutorials"
element.send_keys :enter
sleep 2 # not recommended to use sleep; we should use an implicit wait

if /toolsqa.com/.match(driver.page_source)
  puts "YAY!"
else
  puts "BOO!"
end

driver.quit
Visual Regression Testing

- Create baseline screenshots
- Perform scenarios
- Capture and compare screenshots to known-good baseline
Visual Testing

• Easy! Doesn’t know or care how page is built
• Powerful! Lots of bang for your buck – equivalent to many laboriously-created DOM-based assertions
Visual Testing

(example baseline)

<table>
<thead>
<tr>
<th>Test name</th>
<th>Browser</th>
<th>Size</th>
<th>Screenshot</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataset</td>
<td>Chrome</td>
<td>1024x768</td>
<td></td>
</tr>
<tr>
<td>dataset_zoom</td>
<td>Chrome</td>
<td>1024x768</td>
<td></td>
</tr>
</tbody>
</table>
Visual Testing

(example pass)
Visual Testing

(example failure in text)
Visual Testing

(example failure in graphic)
Visual Testing

- Screenshots of the same scenario can differ slightly at the pixel level
- Captures *everything* in the browser window
- Image only – what about target of a link?
Visual Testing

• Screenshots of the same scenario can differ slightly at the pixel level
• Captures everything in the browser window
  - 3/16/2017 9:14:06 AM
• Image only – what about target of a link?
Drawback Mitigations:

- Tune “fuzz-factor” to allow for small pixel diffs
- Hide uninteresting elements before taking screenshot
- Pay for full-featured system (ex: Applitools Eyes)
- Use Selenium to check targets
Cons:

- Expensive to write
- Slow to run
- Require maintenance – “transparent” DOM changes will break locators
Browser and Visual Testing

Considerations:

- Better for regression tests
- Be selective in what to test
- Coordinate with dev for testability
- Use "page object model"
Browser and Visual Testing

Pros:

- Enables testing of full stack, end-to-end
- Can test in multiple browsers
- Validates the app and deploy/env
- Exercises the web app the same way a user does
Demos

Let’s test some Web Mapping!

• Selenium in action
• With visual testing added
Resources

**Selenium**

- Dave Haeffner: [http://elementalselenium.com/](http://elementalselenium.com/)
- Sauce Labs (ex: [https://saucelabs.com/resources/articles/selenium-tips-css-selectors](https://saucelabs.com/resources/articles/selenium-tips-css-selectors))
- [https://gist.github.com/YmerejRedienhcs/28ec449f758056223076fa55d0954a72#file-selenium-cheat-sheet-md](https://gist.github.com/YmerejRedienhcs/28ec449f758056223076fa55d0954a72#file-selenium-cheat-sheet-md)
- Page Object model: [http://elementalselenium.com/tips/7-use-a-page-object](http://elementalselenium.com/tips/7-use-a-page-object)

**Visual Testing**

- Spectre Open Source visual testing: [https://github.com/wearefriday/spectre](https://github.com/wearefriday/spectre)
- Applitools Eyes visual testing SAAS: [https://applitools.com/](https://applitools.com/)