Ready Steady Code

Ruby Mechanize Handbook

An easy to follow introduction to crawling and scraping websites with Ruby Mechanize

by Tim Craft

http://readysteadycode.com/ruby-mechanize-handbook

Building a basic web crawler

Hello mechanized world

If you haven't used mechanize before, here's a super quick introduction:

```
require 'mechanize'

mechanize = Mechanize.new

page = mechanize.get('http://stackoverflow.com')

puts page.at('title').text
```

This little Ruby script demonstrates the core features you need for crawling and scraping web pages: making HTTP requests, and extracting data from HTML.

Mechanize wraps Ruby's net/http library and a number of other libraries to abstract away the details of crawling and navigating the web. HTML parsing is provided by the nokogiri and libxml2 libraries.

Admittedly not very interesting, so let's build a web crawler!

Building a basic web crawler

One of the common use cases for mechanize is crawling all of the web pages in a single domain. This can be used for indexing content, converting content into different formats, checking and validating content, extracting structured data, and more. A naive implementation of a crawler script might look like this:

```
require 'mechanize'

mechanize = Mechanize.new

page = mechanize.get('http://example.com')

links = page.links

while link = links.shift
  page = mechanize.click(link)

  puts page.title

  links.concat(page.links)
end
```

The script fetches the root web page; extracts the links; then loops through the links, navigating to each page and extracting more links to crawl.

This script *may* work for some websites, but not for most. Can you guess what the potential problems are?

Ignoring visited links

One issue with naively following links is that it's unlikely the script will terminate– it'll get stuck in an infinite loop following circular links between pages and crawl the website indefinitely.

Mechanize automatically tracks which pages it visits, just like a web browser. We can use that functionality to skip pages that have already been visited, for example:

```
require 'mechanize'

mechanize = Mechanize.new

page = mechanize.get('http://example.com')

links = page.links

while link = links.shift
  next if mechanize.visited?(link)

  page = mechanize.click(link)

  puts page.title

  links.concat(page.links)
end
```

By default the mechanize agent will keep 50 pages in the history. If the website you're crawling has more than 50 pages you'll want to set the max_history config parameter to a higher number, like this:

```
mechanize.max_history = 1000
```

You can also set max_history to nil for unlimited history.

Ignoring external links

Another issue with the script is that it will follow *any* link, including links to external websites on different domains. We can fix that by checking that the link is on the same domain like this:

```
require 'mechanize'

mechanize = Mechanize.new

host = 'example.com'

page = mechanize.get('http://' + host)

links = page.links

while link = links.shift
  uri = mechanize.resolve(link.uri)

  next unless uri.host == host

  next if mechanize.visited?(link)

  page = mechanize.click(link)

  puts page.title

  links.concat(page.links)
end
```

The #resolve method is used to make sure we have an absolute URI to check, otherwise we'd have to deal with relative links. Then we check if the URI host is the same as the host we started on.

Ignoring unsupported schemes

Another assumption embedded in the script so far is that all links are clickable. Spoiler: they aren't. If the website you're crawling contains any mailto: or ftp: links you'll get a Mechanize::UnsupportedSchemeError exception.

We can fix this by rescuing from the exception:

```
require 'mechanize'

mechanize = Mechanize.new

host = 'example.com'

page = mechanize.get('http://' + host)

links = page.links

while link = links.shift
  begin
    uri = mechanize.resolve(link.uri)

    next unless uri.host == host

    next if mechanize.visited?(link)

    page = mechanize.click(link)

    puts page.title

    links.concat(page.links)
  rescue Mechanize::UnsupportedSchemeError
  end
end
```

We now have a script which will crawl all the pages on a single domain, ignoring links that have already been visited, external links which we aren't interested in, and links that mechanize doesn't support.

Try it against a small website of your own to see how it works.

Let's look at some more changes which will make the script more robust and fault tolerant with real world websites.

Navigating real world websites

Observing robots.txt rules

The “robots exclusion standard” (aka robots.txt) is a mechanism for websites to indicate which web pages shouldn't be accessed by crawlers and bots. As responsible web citizens we want to support the standard.

Mechanize has built-in support for processing robots.txt files and applying the rules they contain, and can be enabled like this:

```
mechanize.robots = true
```

Attempting to access a page that is disallowed by the site’s robots.txt file will raise a Mechanize::RobotsDisallowedError exception, so to make our script fault tolerant we’ll need to rescue the exception:

```
require 'mechanize'

mechanize = Mechanize.new

mechanize.robots = true

host = 'example.com'

page = mechanize.get('http://' + host)

links = page.links

while link = links.shift
  begin
    uri = mechanize.resolve(link.uri)

    next unless uri.host == host

    next if mechanize.visited?(link)

    page = mechanize.click(link)

    puts page.title

    links.concat(page.links)
  rescue Mechanize::UnsupportedSchemeError
  rescue Mechanize::RobotsDisallowedError
  end
end
```

Alternatively we could check if links are disallowed before clicking on them using the agent #robots_allowed? method, but given we need to handle other exceptions there isn’t much benefit to doing it that way.

Handling redirect errors

Mechanize will raise a Mechanize::RedirectLimitReachedError exception for websites that contain circular redirects, so let's rescue from that one as well:

```
require 'mechanize'

mechanize = Mechanize.new

mechanize.robots = true

host = 'example.com'

page = mechanize.get('http://' + host)

links = page.links

while link = links.shift

  begin
    uri = mechanize.resolve(link.uri)

    next unless uri.host == host

    next if mechanize.visited?(link)

    page = mechanize.click(link)

    puts page.title

    links.concat(page.links)
  rescue Mechanize::UnsupportedSchemeError
  rescue Mechanize::RobotsDisallowedError
  rescue Mechanize::RedirectLimitReachedError
  end
end
```

By default mechanize limits the number of redirects to 20. If you want your crawler to follow fewer redirects you can set the redirection_limit config parameter to a different number, like this:

```
mechanize.redirection_limit = 5
```

You might also want to log the error in your application.

Handling response errors

Real world websites typically contain broken links. When mechanize follows these links and gets a 404 response it'll raise a Mechanize::ResponseCodeError exception. Ditto for any other 4xx client error, or 5xx server error.

If you're proactively looking for these errors (e.g. to list broken links on a website) you'll want to log them. Even if you aren't looking for them you'll need to handle them in order to make your crawler fault tolerant. Maybe you decide to wait a few seconds and retry the request, maybe you queue it up again to retry later, maybe you decide to drop and ignore it. Whatever your strategy you'll need to handle the errors. For example:

```
require 'mechanize'

mechanize = Mechanize.new

mechanize.robots = true

host = 'example.com'

page = mechanize.get('http://' + host)

links = page.links

while link = links.shift
  begin
    uri = mechanize.resolve(link.uri)

    next unless uri.host == host

    next if mechanize.visited?(link)

    page = mechanize.click(link)

    puts page.title

    links.concat(page.links)
  rescue Mechanize::UnsupportedSchemeError
  rescue Mechanize::RobotsDisallowedError
  rescue Mechanize::RedirectLimitReachedError
  rescue Mechanize::ResponseCodeError => exception
    if exception.response_code == '404'
      puts "broken link from #{link.page.uri} to #{link.uri}"
    else
      # ignore other errors
    end
  end
end
```

The exception response_code attribute returns the response code as a string, which you can use to perform different actions for different errors.

If you prefer to handle certain client or server errors within the “happy path” of your code you can tell mechanize to allow specific error codes like this:

```
mechanize.agent.allowed_error_codes << '404'
```

With this configuration mechanize will return Mechanize::Page objects instead of raising exceptions. Make sure to update your code accordingly!

Following meta refresh tags

Meta refresh is a mechanism for instructing web browsers to refresh a page after a given time interval using an HTML tag. Although discouraged by the W3C many older websites still use it. Mechanize has support for following these redirects, which can be enabled like this:

```
mechanize.follow_meta_refresh = true
```

Consider that this can result in pages being visited twice, if a page has a meta refresh tag which redirects to another page which the crawler has already visited. This is because the crawler checks if it has visited a link, not a page, and in the case of meta refresh it won't know what page it'll end up on before clicking the link.

Handling timeout errors

Timeout errors occur when there is a significant delay in connecting to or reading from an unresponsive website.

Mechanize provides #open_timeout and #read_timeout config attributes for setting the timeout values of the underlying Net::HTTP connection. The default timeout values will depend on which kind of Ruby you are using—in MRI 2.3.0 they should both be 60 seconds, in earlier versions of MRI the read timeout is 60 seconds but the open timeout is nil.

In practice you'll probably want to set both of these timeouts to lower values in order to minimise time spent waiting for slow responses. For example, to set both of the timeouts to 10 seconds:

```
mechanize = Mechanize.new

mechanize.open_timeout = 10
mechanize.read_timeout = 10
```

You'll also need to adjust your script to handle Net::OpenTimeout and Net::HTTP::Persistent::Error exceptions so that your script doesn't crash.

Why not Net::ReadTimeout exceptions? An implementation detail of mechanize: the underlying Net::HTTP::Persistent library retries read timeouts automatically, and then raises its own exception.

Handling international domain names

An international domain name is one which contains non-ASCII characters. DNS is restricted to ASCII characters, but international domain names are supported by the IDNA standard which defines conversion to an “ASCII Compatible Encoding”.

For example: the international domain name Bücher.ch converts to xn--bcher-kva.ch, which is ASCII compatible.

How does this affect your code? Ruby’s built-in URI library doesn’t support international domain names. Attempting to parse a URI with an international domain name directly will raise a URI::InvalidURIError exception, and you might also run into SocketError exceptions which aren’t as easy to diagnose.

Why SocketError exceptions? Mechanize rescues from URI::InvalidURIError exceptions before escaping the href and re-parsing it, returning a URI with a non-existent domain. Attempting to connect to a non-existent domain will then raise a SocketError exception.

So what do you do if you need to handle international domain names? Use the addressable gem to normalize URI values.

The best place to normalize URI values is when they are extracted from HTML responses. Before using addressable you might have code that looks like this:

```
uri = mechanize.resolve(link.uri)
```

Using addressable to parse the raw href value would look like this:

```
normalized_uri = Addressable::URI.parse(link.href).normalize

uri = mechanize.resolve(normalized_uri)
```

The Mechanize::Link#uri method uses the built-in URI library, so instead call the #href attribute to get the raw href value and normalize that with addressable.

If your script is seeded with URI values from an outside data source you’ll also need to make sure they have been normalized.

Different ways to select links

So far the examples have been using page.links, which returns all the links in the page. Useful for when you want to crawl every page on a site but often you'll want to use mechanize to search for more specific links, and there are some helper methods available to make that easier. For example, here's how you'd use the link_with method to search for a single "Sign up" link:

```
link = page.link_with(text: 'Sign up')
```

Use the dom_id option to select a link with a specific id:

```
link = page.link_with(dom_id: 'signup')
```

In addition to link_with there is also a links_with method for selecting multiple links matching the given criteria. For example, here's how you'd use the links_with method to search for all the nav links on a page:

```
nav_links = page.links_with(dom_class: /\bnav-link\b/)
```

Links only have one id value but can have multiple class names, hence the use of a regular expression to match the class name. The options are named dom_id and dom_class instead of id and class because they match against the Ruby methods of the Mechanize::Page::Link class, not the attributes directly.

A similar approach can be used to select all links with a specific extension. For example, selecting all links to PDF documents:

```
document_links = page.links_with(href: /\.pdf\z/)
```

If you need to select links based on the "rel" attribute or more complex matching logic you'll need to filter page.links directly instead of using the helper methods. For example, selecting a rel=next link for pagination:

```
next_link = page.links.detect { |link| link.rel?('next') }
```

Instead of using the link helper methods you can also search for elements directly using the nokogiri search methods. For example:

```
nav_links = page.search('a.nav-link')
```

This could be useful if you want to extract some logic for testing or re-use which only has nokogiri as a dependency. The key difference to note is that this will return different types of objects to the link helper methods.

Different ways to click links

The easiest way to click a link is using the click method:

```
link.click
```

Can't get more readable than that!

Only Mechanize::Page::Link objects have the click method, so if you're getting NoMethodError exceptions from that line it's probably because you're selecting links using the nokogiri search methods.

Another way to click links is to use the Mechanize#click method:

```
mechanize.click(link)
```

This works with both link objects and nokogiri objects, so is a much better choice if you are selecting both. It also makes it easier to identify where in your code the crawling and navigation is happening, and easier to then group those concerns together around the mechanize object.

The Mechanize#get and Mechanize#download methods don't accept link objects or nokogiri objects, so if you need to pass links back to those methods you'll first need to map the objects to their href attributes.

Getting data out

Selecting HTML elements

The easiest way to identify specific elements in a document is to search for them using CSS selectors. These are more familiar for web developers than complicated XPath expressions, and isolate your scraping code from simple structural changes to the data, such as nesting an element within another element.

Mechanize uses the nokogiri gem to parse HTML, and exposes a number of nokogiri methods on page objects which can be used as an entry point for selecting elements.

Use the #at method to search for a single element, and the #search method to search for multiple elements. For example:

```
element = page.at(selector)

elements = page.search(selector)
```

Remember to handle the nil/empty case if the HTML you are scraping doesn't always include the elements you want to select.

Extracting text & attributes

Use the #text method to extract the textual content within an element (or the equivalent but more tedious to type #inner_text method).

For example, to extract the title text from a page:

```
title = page.at('title').text
```

Nokogiri automatically decodes any HTML entities within the text. Whitespace is preserved, so you may want to strip or otherwise process the resulting string. If you need to map
 line break tags to newlines you'll need to implement that logic yourself by recursively looping through child nodes.

Attributes can be accessed using square brackets notation. For example, here's how you might extract the description from a meta tag:

```
description = page.at('meta[name="description"]')['content']
```

You can also call the #to_h method to get a hash of names and values for all of the attributes on the element.

Downloading files & saving images

To download a linked file, use the Mechanize#download method.

If you have selected a mechanize link object you can download the file like this:

```
link = page.link_with(id: id)

mechanize.download(link.href, filename)
```

If you have selected a nokogiri element object you can download the file like this:

```
element = page.at(selector)

mechanize.download(element['href'], filename)
```

A couple of extra helper methods make it straightforward to download all the images on a page:

```
page.images.each do |image|
  image.save
end
```

The page #images method returns all the images on the page (as Mechanize::Image objects). The image #save method downloads the image to the current directory.

If you only want to select a subset of the images on the page, use the #images_with method to specify selection criteria. For example, to only download .jpg images:

```
page.images_with(src: /\.jpg\Z/).each do |image|
  image.save
end
```

You can also pass a filename to the image #save method.

Extracting microdata with Mida

Microdata is an HTML specification for embedding metadata into web pages, with standard vocabularies for products, events, people, creative works etc.

You can use the mida gem to extract microdata from mechanize pages.

Construct a Mida::Document with the root Nokogiri::HTML::Document object and the URI of the page, like this:

```
document = Mida::Document.new(page.root, page.uri)
```

You can then list out the microdata items and their properties like this:

```
document.items.each do |item|
  puts "#{item.type} item with #{item.properties.size} properties"
end
```

Make sure you have version 0.4.0 of the mida gem installed: either add it to your Gemfile and install it with bundler, or install it directly with gem install.

Exporting CSV files

CSV can be used for exporting tabular data in a format that can easily be read by other languages and software. You can export CSV files from your mechanize scripts by traversing the structure of the HTML table. For example:

```
require 'mechanize'
require 'csv'

mechanize = Mechanize.new

url = 'https://en.wikipedia.org/wiki/Table_(information)'

page = mechanize.get(url)

table = page.at('.wikitable')

table_data = table.search('tr').map do |row|
  row.search('th, td').map { |cell| cell.text.strip }
end

CSV.open('example.csv', 'w+') do |csv|
  table_data.each do |row|
    csv << row
  end
end
```

Mechanize is used to fetch a wikipedia article, and a simple class name CSS selector is used to select the first formatted table in the article.

The data is extracted from the table by selecting descendant rows and mapping each row to its descendant cell values (remember that tables can have both th and td elements). The table data is then exported to a CSV file by looping over each row, using Ruby's built-in CSV library.

Exporting XLSX spreadsheets

CSV is a great go-to format for exporting tabular data, but sometimes you need to output something more "user friendly" for non-developers. Here's how you can use the axlsx gem to adapt the previous example script to export a spreadsheet:

```
require 'mechanize'
require 'axlsx'

mechanize = Mechanize.new

url = 'https://en.wikipedia.org/wiki/Table_(information)'

page = mechanize.get(url)

table = page.at('.wikitable')

table_data = table.search('tr').map do |row|
  row.search('th, td').map { |cell| cell.text.strip }
end

table_caption = table.at('caption').text

Axlsx::Package.new do |package|
  package.workbook.add_worksheet(name: table_caption) do |sheet|
    table_data.each do |row|
      sheet << row
    end
  end

  package.serialize('example.xlsx')
end
```

The first half of the script remains unchanged, which demonstrates the benefit of keeping crawling, scraping, and exporting concerns separate in your code.

The table caption is extracted so it can be used to name the worksheet. The new code at the bottom creates the spreadsheet structure. Not very exciting as-is but you could adapt this example to add multiple worksheets, cell formatting, cell formulas, even line charts and bar charts.

Exporting PDF documents

What if the data you're exporting is primarily textual, not tabular? PDF documents might be a good choice. Here's how you can use the pdfkit gem to export a mechanize page to a PDF document:

```
require 'mechanize'
require 'pdfkit'

mechanize = Mechanize.new

page = mechanize.get('http://example.com/')

pdf = PDFKit.new(page.root.to_s, page_size: 'A4')

pdf.to_file('example.pdf')
```

Make sure you have the pdfkit gem installed, as well as wkhtmltopdf (binaries available via the wkhtmltopdf-binary gem).

The page #root method returns a Nokogiri::HTML::Document object which you can modify to exclude content you don't want in the PDF.

Dealing with forms

Filling & submitting forms

A common use case for mechanize is automation. Whether you need to log in to a website, query data using search forms, or submit data to the server, you'll need to programmatically fill out some HTML forms.

Let's start with a simple example which demonstrates the basics of finding a form, setting input values, and submitting the form:

```
form = page.form_with(id: 'search-form')

form['q'] = 'keywords'

form.submit
```

If you know the id or the name of the form you need then the #form_with method is the easiest way to select it. If the form doesn't have a predictable id or name then either you can select the form positionally, or with arbitrary matching logic:

```
form = page.forms.first

form = page.forms.find { |form| ... }
```

Mechanize forms have a Hash-like interface for setting form input values, which is a simple and convenient way to fill out a form when you know the names of the inputs you need to include.

The form can then be submitted with a call to the form's #submit method.

Selecting the correct submit button

HTML forms have different types of submit buttons: input type=submit, input type=button, input type=image, and button type=submit.

By default mechanize won't include any values from these submit buttons, which isn't a problem until you have a form which has submit buttons with values that need to be submitted to the server.

You can find buttons in the form either using the #button_with method, or by iterating through the #buttons array:

```
button = form.button_with(value: 'yes')

button = form.buttons.find { |button| ... }
```

Passing the button as an argument to the form's #submit method will then include the button name and value in the request.

Alternatively if you know the name and the value of the button you want to select you can just set the value on the form directly:

```
form['button_name'] = 'button_value'
```

Less common, but for forms with image submit buttons you'll also need to set the x and y coordinates of the "click" that you're trying to simulate:

```
button.x = 100

button.y = 100
```

Remember that forms can have multiple submit buttons, so selecting the first button won't always be the correct behaviour to implement.

Selecting drop-down options

Form objects have some helper methods for selecting options from drop-down select lists. If you know the name of the select list and the value of the option you want to select, you can set the value on the form:

```
form[select_list_name] = option_value
```

This also works with multi select lists, just specify an array of values:

```
form[select_list_name] = [option_value_1, option_value_2]
```

Alternatively if you don't know the name of the select list you can use the #field_with method to locate the element by id or class name:

```
select_list = form.field_with(id: id)
```

Similarly you can then use the #option_with method to locate a select list option by id, class name, or its text:

```
option = select_list.option_with(text: text)
```

Once you have the correct option, call the #select method to select it:

```
option.select
```

This also works with multi select lists. For more complex selection criteria you can loop through all the options within the select list:

```
select_list.options.each { |option| ... }
```

You can also use #select_all and #select_none on the select_list to select all of the options and none of the options respectively.

Toggling checkboxes & radio buttons

Form objects also have a number of different helper methods for selecting checkbox and radio button elements. Use the #checkbox_with or #radiobutton_with methods if you can locate the element by name, id, or class name:

```
checkbox = form.checkbox_with(name: name)

radio_button = form.radiobutton_with(id: id)
```

Alternatively you can find the elements positionally or with arbitrary matching logic by iterating through #checkboxes or #radiobuttons, similar to locating the form on the page. For example:

```
checkbox = form.checkboxes.first

radio_button = form.radiobuttons.find { |button| ... }
```

Once you've found the correct checkbox or radio button, call the #check method to check or select it:

```
checkbox.check

radio_button.check
```

You can also call #uncheck if the field is checked by default and you need it to be unchecked, or #click to toggle the field according to its current state.

Uploading files

To upload a file, use the #file_upload_with method to locate the file input, and then set its file_name attribute to the file path:

```
file_input = form.file_upload_with(name: 'photo')

file_input.file_name = 'path/to/photo.jpg'
```

If you need to upload data from memory or a datastore instead of the filesystem you can instead specify the file_data and mime_type attributes:

```
file_input.file_data = '{"foo":"bar"}'

file_input.mime_type = 'application/json'
```

Mechanize automatically detects the encoding type of the form and encodes the request data as multipart form data.

Troubleshooting

Using a logger

It can be frustrating trying to figure out why your scraper code isn't working when you don't have any clear error messages pointing you to the root of the problem. Thankfully mechanize comes with some built-in logging functionality which can help you get some visibility into the requests it makes "under the hood".

You can configure logging by setting the #log config attribute on the mechanize object to a regular Ruby logger object. For example:

```
require 'mechanize'
require 'logger'

mechanize = Mechanize.new
mechanize.log = Logger.new(STDOUT)
mechanize.log.level = Logger::INFO
```

With logging configured you'll get logger output that includes some details of the mechanize requests and responses which looks like this:

```
INFO -- : Net::HTTP::Get: /
INFO -- : status: Net::HTTPOK 1.0 200
```

For a production application you can send that output to a log management service as you would with your regular application logs.

Debugging request headers

Sometimes a server won't like the requests you're sending and you'll get responses you didn't expect. When trying to diagnose issues like that it helps to see the HTTP headers that are being sent back and forth.

In order to debug request headers simply set the log level to Logger::DEBUG instead of Logger::INFO. For example:

```
mechanize.log.level = Logger::DEBUG
```

With the log level set to Logger::DEBUG you'll get additional logger output that includes the request headers and the response headers. It looks like this:

```
request-header: accept-encoding => gzip,deflate,identity
request-header: accept => */*
request-header: user-agent => Mechanize/2.7.5 Ruby/2.4.1p111 (...)
request-header: host => example.com
...
response-header: content-type => text/html
```

Whether it's appropriate or not to include that level of log output in production depends on your application. If you aren't sure you can always add in a MECHANIZE_LOG_LEVEL environment variable which you can use to temporarily lower or raise the log level without a code change.

Debugging with irb

When debugging complex issues in development it can often help to "step through" your code line by line to determine the root of the problem.

Ruby has a built-in debugger library (debug.rb) and there's also the more featureful byebug gem, but you can also step through your code manually with irb.

Simply fire up an irb session, require mechanize, and copy/paste the lines of your code one by one. For example:

```
$ irb -r mechanize
irb(main):001:0> mechanize = Mechanize.new
...
irb(main):002:0> mechanize.get('http://httpbin.org/get')
```

Check each line does what you expect and tweak until it's working, then update your scraper code to fix the problem.

Quick and dirty? Yep. But no need to install any gems, or dig into documentation to find out what the debugger commands are. And this technique isn't specific to web scraping, so it's something you can use to debug any of your Ruby code.

Debugging with Charles

Another approach to debugging is to use a network proxy like Charles to capture your mechanize requests and responses for more detailed inspection.

You can proxy your mechanize requests through Charles like this:

```
mechanize = Mechanize.new
mechanize.set_proxy('localhost', 8888)
```

Specifying the proxy is straightforward with the #set_proxy config method, but there are a couple more steps you need to take if you want to capture and analyse SSL traffic (which is likely given an increasing number of websites are using https).

First you need to enable SSL proxying within Charles: go to `Proxy -> SSL Proxying Settings...` and add the hostnames that you want to enable SSL proxying for.

If you run your script at this stage it'll fail with an OpenSSL::SSL::SSLError exception when it tries to connect, so next you need to configure mechanize to use the Charles root certificate. Here's the code:

```
cert_store = OpenSSL::X509::Store.new
cert_store.set_default_paths
cert_store.add_file('charles-ssl-proxying-certificate.pem')

mechanize.agent.http.cert_store = cert_store
```

Go to `Help -> SSL Proxying -> Save Charles Root Certificate...` within Charles to save the root certificate somewhere within your project.

If everything is setup correctly you should see your scraper requests showing up in Charles, where you can view the requests and responses in full.

Detecting DOM changes

If you already have a working scraper then the most likely cause of it failing is a change to the DOM structure of the website you're scraping.

Consider how this manifests in your mechanize code. If you search for a single element and it doesn't exist you'll get nil, and your script will fail with a NoMethodError exception. If you search for multiple elements and they don't exist then your script will fail silently (difficult to diagnose!).

You could sprinkle your code with defensive programming logic to assert that elements exist, but that can be unwieldy and quickly bloat your code. You could write your own select method with built-in existence checking, but then you'd have to rewrite all your code which uses chains of nokogiri method calls into nested calls to your own method (lots of work, and the resulting code is harder to modify).

What if you could detect DOM changes automatically, with no changes to your existing scraper code? Spoiler: you can, with refinements.

Refinements are a Ruby 2.0 feature which can be used to modify the behaviour of existing classes without monkey-patching things globally. Here's how you can use refinements to modify the behaviour of the nokogiri #search method:

```
class NokogiriSearchError < StandardError
  module Refinement
    refine Nokogiri::XML::Searchable do
      def search(*args)
        result = super(*args)

        return result unless result.empty?

        selector = if respond_to?(:css_path)
          css_path
        else
          first.css_path.sub(/:nth-of-type\(1\)$/, '')
        end

        message = "could not find #{args.join(', ')}"
        message << " within #{selector}" unless selector.empty?

        raise NokogiriSearchError, message
      end
    end
  end
end
```

This refinement code contains a number of relatively advanced Ruby features, so don't worry if you don't understand it all.

Here's how you'd use the refinement:

```
using NokogiriSearchError::Refinement

mechanize = Mechanize.new

page = mechanize.get(url)

page.at('body').search('.some .selector').each do |li|
  puts li.text
end
```

Try it with a url of your choice and different selectors. Search for a selector that doesn't exist and you'll get a helpful error message which pinpoints the problem:

```
could not find .some .selector within html > body
```

All without having to change your existing scraper code. Pretty cool!

You can achieve a similar result with monkey-patching but it's more involved because you have to re-open and modify at least 3 different classes.

Other useful things to know

Saving & loading cookie state

If you're automating the process of logging into a website then it makes sense to save cookies like web browsers do. This will speed up any subsequent script runs, and cut down on network requests which benefits both your script and the server.

You can save the mechanize cookie state with a single call to the cookie jar #save method, for example:

```
mechanize.cookie_jar.save('cookies.yml')
```

By default mechanize will only save persistent cookies, ignoring any session cookies. Specify the session option if you want to include session cookies:

```
mechanize.cookie_jar.save('cookies.yml', session: true)
```

By default mechanize will save the cookies in YAML format, which is portable between different programming languages. Mechanize also supports Mozilla's cookies.txt format, which you can use by specifying the format option:

```
mechanize.cookie_jar.save('cookies.txt', format: :cookiestxt)
```

Note that the YAML format isn't compatible with some older versions of mechanize.

If you need to persist the cookie state somewhere other than a file you can potentially implement your own "saver" subclass, but it's probably easier just to iterate over the cookies and serialize them directly.

Manually adding cookies

Often you'll want to have mechanize use cookies that have been set elsewhere. You might want to use a cookie from a manual web browser session when testing, or pass a cookie from a selenium driver when the server sets the cookie using javascript. The simplest way to add a cookie to mechanize is to parse the original Set-Cookie header value together with the origin URI of the page that set the cookie:

```
uri = 'https://example.com/login'

set_cookie = '__uid=xxx; ... path=/; Secure; HttpOnly'

mechanize.cookie_jar.parse(set_cookie, uri)
```

If you can't easily get the Set-Cookie header value, or you need to adjust the individual components of the cookie you can instead construct a cookie object and add that to the cookie jar. For example:

```
cookie = Mechanize::Cookie.new({
  name: 'uid',
  value: 'xxx',
  domain: 'example.com',
  path: '/',
  for_domain: true,
  expires: Time.now + 3600,
  httponly: true,
  secure: true
})

mechanize.cookie_jar.add(cookie)
```

For more detail on the cookie jar interface check the http-cookie gem that mechanize uses for cookie handling.

Executing Javascript with Poltergeist

One disadvantage of mechanize is that it can't parse JavaScript, so if you're trying to crawl a website with a JavaScript based login flow or a JavaScript search interface you'll need to get creative.

You could reverse engineer the JavaScript and mimic its HTTP interactions with mechanize, but that's time consuming and prone to breaking. You could switch to PhantomJS and write your scraper code in JavaScript, but that's not much use to Ruby developers building Ruby projects. A better option might be to use the poltergeist gem, which wraps PhantomJS and exposes it to Ruby. For example, here's how you could pass cookies from mechanize to poltergeist:

```
require 'mechanize'
require 'capybara/poltergeist'

mechanize = Mechanize.new

page = mechanize.get('http://example.com')

session = Capybara::Session.new(:poltergeist)

mechanize.cookie_jar.each do |cookie|
  session.driver.set_cookie(cookie.name, cookie.value, {
    domain: cookie.domain,
    expires: cookie.expires
  })
end

session.visit('http://example.com')

version = session.evaluate_script('jQuery.fn.jquery')

puts "Using jQuery #{version}"
```

Just a few lines of code to pass the cookies over, and then you can use poltergeist methods like #evaluate_script to interact with JavaScript. You could also reverse this around and use poltergeist for a JavaScript based login, and then pass those cookies back to mechanize.

See HOWTO scrape websites with Ruby & Poltergeist for more examples on how to get started with poltergeist and a capybara/poltergeist cheatsheet.

Rate limiting / throttling requests

Some websites use rate limiting techniques to protect themselves against badly behaving clients. You can identify rate limiting from HTTP 429 Too Many Requests responses or the presence of X-RateLimit headers, but the server might just return other error responses which cannot easily be distinguished from regular errors.

From the perspective of the server, a "benign" web crawler script which consumes much of the server's resources isn't all that different from a malicious denial of service attack. It's therefore good practice to proactively throttle your web crawlers.

The easiest way to do this is to use the slowweb gem to add throttling functionality to Net::HTTP (which mechanize uses internally). For example:

```
require 'mechanize'
require 'slowweb'

SlowWeb.limit('example.com', 10, 60)

mechanize = Mechanize.new

page = mechanize.get('http://example.com')
```

The call to SlowWeb.limit defines the rate limit for the given domain, in this example we limit ourselves to 10 requests per minute. Any subsequent requests from mechanize to that domain will then be throttled.

Structuring crawlers & scrapers as classes

Ruby scripts are great for getting started on a web crawler, or "quick and dirty" code spikes. As your scraping code grows in complexity in makes sense to re-structure it, just as you would with any other code. Well structured code is easier to maintain and extend; easier to test and refactor; and easier to re-use in different contexts.

A straightforward approach for structuring your scraping code is to subclass the Mechanize class and move the top-level logic into a #crawl or #scrape method. Using the example from "Building a basic web crawler" we'd get this:

```
require 'mechanize'

class BasicWebCrawler < Mechanize
  def crawl(url)
    page = get(url)

    @links = page.links

    while link = @links.shift
      page = click(link)

      visit(page)
    end
  end

  private

  def visit(page)
    puts page.title

    @links.concat(page.links)
  end
end

crawler = BasicWebCrawler.new
crawler.crawl('http://example.com')
```

With the crawler structured in this way you can then factor out different concerns into private methods or collaborator classes/modules. If you're not used to doing this kind of refactoring a good starting point would be to group together all the nokogiri code related to extracting data from page elements and moving it to a module which can be isolated from mechanize and tested independently.

Re-use in different contexts is a bit theoretical and abstract, so let's look at one of the more common use cases for that: background jobs.

Running scrapers in background jobs

Sometimes you might want to kick off a web scraping workflow within a user-facing application and have it run in the background. Running your scrapers in the background also allows you to take advantage of the error handling and retry logic that exists in background job systems instead of coding and managing that yourself.

We can adapt the BasicWebCrawler class to run as a background job using sidekiq with just a few simple tweaks. Here's the code:

```
require 'mechanize'
require 'sidekiq'

class BackgroundWebCrawler < Mechanize
  include Sidekiq::Worker

  def perform(url)
    page = get(url)

    @links = page.links

    while link = links.shift
      page = click(link)

      visit(page)
    end
  end

  private

  def visit(page)
    puts page.title

    @links.concat(page.links)
  end
end

BackgroundWebCrawler.perform_async('http://example.com')
```

By including the Sidekiq::Worker module and re-naming the top level method to #perform it's now possible the run the crawler in the background of your application without having to write complex queuing or retry logic.

The example uses sidekiq, but the same approach should also work with similarly architected systems like resque and delayed_job.

Testing with VCR & Webmock

Web scraping is inherently brittle, meaning your scripts are prone to breaking over time due to changes in website content and structure.

Testing can help, but tests for code that makes network calls will be slow and non-deterministic. The tests will be dependent on your network connection and the servers that you're testing against, any of which can fail temporarily.

Mocking out the network connection and returning canned "fake" responses with a gem like webmock or fakeweb solves both of these problems but then your tests will be inaccurate, missing most of the data returned from a "real" request.

The vcr gem allows you to record and replay HTTP interactions in your tests without these problems. Let's look at how we can use vcr and webmock together to test a simple piece of mechanize code:

```
require 'mechanize'

def get_title(url)
  mechanize = Mechanize.new

  page = mechanize.get(url)

  page.at('title').text.strip
end
```

The get_title method uses mechanize to fetch the given url and extract the page title. Not very interesting, but sufficient to test.

Make sure you have the vcr and webmock gems installed: either add them to your Gemfile and install them with bundler, or install them directly with gem install.

The vcr gem needs a little configuration:

```
require 'vcr'

VCR.configure do |config|
  config.cassette_library_dir = 'vcr'
  config.hook_into :webmock
end
```

We can then write a test for the get_title method like this:

```
require 'test/unit'

class GetTitleTest < Test::Unit::TestCase
  def test_get_title
    VCR.use_cassette('get_title') do
      assert_include 'httpbin', get_title('http://httpbin.org')
    end
  end
end
```

The first time you run the test vcr will save the “real” HTTP response into the directory specified by config.cassette_library_dir. The next time you run the test vcr will load and return the saved response without hitting the network. This gives you a fast, deterministic, and accurate approach to testing your mechanize code.

The example uses the test-unit gem, but you can use other testing libraries like minitest or rspec in a similar way.

Ruby Mechanize Handbook

First Edition, September 2017