CHAPTER 9

Client-Side Web and APIs

The web is the infinite source of information, and it’s right there for you to tap. Lots of services can be used straight away, in other cases you have to do some massaging, and in yet other cases you need to be properly authorized. In all these cases, Raku is there to help you.

Recipe 9-1. Query a GeoIP Database

Problem

You have a bunch of Internet addresses in your log file, and you need to know where they come from to check out which countries are interested in your content.

Solution

Use GeoIP2, a module in the Raku ecosystem, to query a MaxMind database, a database that includes information about geolocation of groups of IPs.

How It Works

You’ve got in your hands a whole set of files with logs from your recipe web server, and you’d like to know where all those IPs are from. That way, you can create more content about regional cuisine. Or maybe you just want to know out of sheer curiosity. Proprietary solutions will give you that information, but it goes along with some loss of privacy for your users, so it’s better to develop your own.

Fortunately, there’s a company, called MaxMind, that produces a series of databases. These databases are in an open format and you can use them to query about IPs. You can access them through the GeoIP2 downloadable module.
These databases, however, are proprietary. There are several ways to get one of them:

- Your company might be extremely interested in one, and it will buy one for you.
- There are open source tools written in several languages, like Perl, which can produce data in that format, and they are open source. See [https://github.com/maxmind/MaxMind-DB-Writer-perl](https://github.com/maxmind/MaxMind-DB-Writer-perl). You might want to create your own, limited, database, using for instance IPs within your company’s VPN (if it’s a multinational one).
- Finally, you can download GeoLite databases for free. These have limited precision, but we’re only interested in the country and continent, so this will do.

We will create a simple script that checks the IP of the machine you’re using to connect to the Internet and tells you which country and continent you are from.

```perl
use GeoIP2;

my $ip = qx{curl -s ifconfig.me};
my $geo = GeoIP2->new( path => 'Chapter-9/GeoLite2-Country.mmdb' );
my $location = $geo->locate( ip => $ip );

say "The IP is in $location<country><names><en>, $location<continent><names><en>";
```

Since locally we will only know the local IP (or IPs) your system is connected to, which will typically be a local IP issued by your router, we use the ifconfig.me Internet service via curl to obtain our own IP. We need to have curl installed and accessible, and the -s will make it return the IP without printing anything else to the screen (s = silent). The qx quoting construct is yet another way of running external programs, similar to what we did in Chapters 1 and 2 with shell and run.

We then initialize the database object with the path to the database file (that previously I had registered to download; you can do the same by registering at [https://dev.maxmind.com/geoip/geoipt2/geolite2/](https://dev.maxmind.com/geoip/geoipt2/geolite2/)). We call the locate method to obtain the location, which will be returned in a data structure that includes the names of countries and continents in several formats, as well as some geocoding information. We’ll just access
the name (stored in the `<names>` key) of the country and continent in English (stored in a key that represents the language code, `<en>`). This will print the following for me:

The IP is in Spain, Europe

    Right on!
    If what you want is to parse the log, you need to process this log using any of the text-processing recipes you’ve seen so far.

**Recipe 9-2. Download and Extract Information from a Website**

**Problem**

You need to get some information that is included on a website.

**Solution**

Download the page using an external CLI tool or using one of the available libraries, such as `HTTP::UserAgent`, `Cro::HTTP`, or `LWP::Simple`. As their names imply, `LWP::Simple` is simpler, `HTTP::UserAgent` gives you a bit more flexibility to create specific headers, and `Cro::HTTP` is the best designed and maintained. You will probably will not need all the functionality of `Cro::HTTP`. Once the content is downloaded, use regular expressions to capture the information you’re interested in or, if that’s not easy or possible, capture the information in an HTML document object model parser such as `DOM::Tiny`.

**How It Works**

There’s a lot of information on the web. If only there was a way to get it and process it...

But there is! From the beginning of time, *scraping* has been a way to gather information and data that is semi-structured and published on the web. However, scraping is still an arcane art, with many different degrees of freedom: from how the information is structured, to more esoteric challenges like throttling (making request responses slower when there are repeated requests from an IP) or terms of service (you can be banned from a site if you download information from it, even if you’re not actually publishing that information anywhere else).
There are also technical challenges. Essentially, when you are scraping, you will need to download the page written in HTML and then parse that HTML for the information you need. In many cases (for instance, if that information has a clear prefix, or is structured in a certain way), parsing is straightforward. In some others, you will need to parse the whole page to the DOM and then look for a particular leaf of the tree that hangs from a certain branch.

---

**Note** In yet others, you might get lucky and obtain the information from a data div or from a user-accessible JSON file.

That’s the case also for recipes. There’s a myriad of sites that publish recipes daily, or where recipes are laid out, often in a format that has a certain structure.

---

**Note** Unfortunately, a microformat called hRecipes for including recipes in HTML is no longer fashionable, and I doubt it’s ever been.

For a myriad of reasons, however, it’s probably better if we stick to open license sites like Wikipedia. Wikibooks includes a whole lot of recipes at [https://en.wikibooks.org/wiki/Category:Recipes](https://en.wikibooks.org/wiki/Category:Recipes), which pretty much follow the same format. This script will download one of them and extract the ingredients.

```perl
use HTTP::UserAgent;
my $URL = @*ARGS[0] // "https://en.wikibooks.org/wiki/Cookbook:Apple_Pie_I";
my $recipr = HTTP::UserAgent.new;
my $response = $recipr.get($URL);
die $response.status-line unless $response.is-success;
my $ingredients = ( $response.content.split(/"h2"/))[1];
my @ingredients = ($ingredients ~~ m:g/"\Cookbook:(\w+)/);
say @ingredients.map( ~*[0] ).unique;
```

You will need to download the HTTP::UserAgent module before using it; it’s a frequently updated module in the ecosystem.

---

168
Note  As with other modules in this fast-moving ecosystem, there might be some quirk in installation with the version of Raku you’re using. If you don’t want to live on the edge, use the Rakudo Star distribution, which includes many other useful modules.

The script will take the URL for a recipe or use the one for the apple pie, because what’s better than apple pie, right? It will instantiate a version of the HTTP user agent, and will fetch the URL in the next sentence. If there’s an error, it will bail out with a message.

Note  For the sake of getting to the point, we’ll be skipping checks and graceful handling of errors. For instance, we’re not checking the URL so that it effectively has the right pattern. That’s left to the reader. Tip: A simple regex will do.

If everything is okay, we can proceed to extract the information. We know that there’s going to be an H2 section called Ingredients; that’s the first H2 segment indeed. So rightfully we split using the simple <h2> and take the second one, index 1 in the array.

Note  That’s one good thing about Wikipedia: clean markup. It would probably be a bit more difficult to split sections if they were marked with some other bit of markup and CSS classes.

Within that section, we realize that for every ingredient there’s a link to a section that describes different recipes that use that ingredient. They all follow the same pattern: https://en.wikibooks.org/wiki/Cookbook: + ingredient (and it will always be a set of word-like characters). Thus, the pattern we use: m:g/"\Cookbook:\(\w+\)\)/:g will match all of them, and the parentheses will capture just the word, so we get a set of Match objects in an array. We need to extract the precise string that’s been matched: .map( ~*[0] ) will get the first match in a Match and will stringify it. We extract the unique elements out of this array, just in case any of them are repeated. This will print something like this:

(Flour Margarine Lard Salt Water Apple Lemon_Juice Butter Sugar Cornstarch Cinnamon Nutmeg Milk)
Note  Yes, there seems to be a section devoted to recipes with water. Hmm, yummy! “This soup is delicious! What was your secret ingredient?” “Water”

Please note that the ingredients that consist of two separate words, like Lemon Juice, use an underscore to separate them; underscore is also matched with \w, so no problem here.

This was moderately successful, but we still need to be a bit crafty to get to the information we want, by extracting it from an URL that has a certain shape.

Tip  Scraping is an arcane art and you need to be crafty all the time. You can also learn the art and craft of scraping from Website Scraping with Python and Practical Web Scraping for Data Science, two excellent books published by Apress. Although the language is different, the techniques and methodology is not that different.

Let’s try to use the DOM. While this will, in general, get you pieces of data with much more precision, it should be taken into account that modern DOMs are, in general, dynamic, so you can’t get them from the source (you would need a headless browser of which, for the time being, there are none for Raku). Additionally, you need the DOM structure to be static, and many sites change that routinely. Let’s stick to this same Wikibooks source, which is cleaner and scrapable in that aspect.

```perl
use WWW;
use DOM::Tiny;

my $URL = @*ARGS[0] // "https://en.wikibooks.org/wiki/Cookbook:Apple_Pie_I";
my $content = get($URL);

die "That $URL didn't work" unless $content;

my @all-lis = DOM::Tiny.parse( $content.split(/"h2"/)[1]).find('li').map: ~*;

my @my-lis = @all-lis.grep( /title..Cookbook ":"/)
    .map( { DOM::Tiny.parse( $_ ) } );

say @my-lis.map( "→ " ~ *_.all-text).unique.join("\n");
```
We’ve switched to a different library with HTTP commands, the simply and aptly named WWW, which gives us a simple get command to download from the web (it can also parse JSON on the fly). DOM::Tiny will provide the DOM-parsing abilities.

Although HTML is a document structure description language, there’s not really a great way to provide a top-down structure for it. For instance, division in sections does not show in the structure (it could, by using the section tag; however, it’s not used here). That is why, to extract the ingredient section, we still have to do the same tag-based splitting we did in the previous version. That way, we make sure that what we parse really contains what we’re interested in. The HTML for this page is quite clean, meaning that there’re no semantic class attributes anywhere to be seen. If the ul tag that contains our list items had a class='ingredients' tag, it would have been way easier. Be that as it may, from looking at the source we see that every ingredient is in an <li> tag, so find("li") will get all of them in a Raku Seq (that is, a sequence of items over which you can iterate). Learn all about sequences at https://docs.raku.org/type/Seq or from Chapter 5 of Perl 6 Quick Syntax Reference. We need to do additional checking, so we render them back to HTML by stringifying them via ~*.

What we need to do is the next grep: if there’s no link to an ingredient page, it’s not an ingredient. On the apple pie page, they are all this way, but we’ll stay on the safe side and actually check this. There could be some additional instructions about, I don’t know, using some special pan or pre-heating the oven (you always need to pre-heat the oven). We parse them again, simply because it’s the simplest way to get rid of the markup and get all the text, which we do next using .all-text, another DOM::Tiny method.

The result will be as follows:

→ 8 oz (225 g) plain flour
→ 4 oz (110 g) margarine
→ 2 oz (55 g) lard
→ pinch of salt
→ 2 tablespoons cold water
→ 1 lb (500 g) apples, sliced
→ 2 tbsp lemon juice
→ 1 oz (28 g) salted butter
→ 2½ c sugar, and additional for sprinkling
→ ¼ c flour
→ 2½ tbsp cornstarch
→ cinnamon
→ nutmeg
→ 1 oz (28 g) milk
→ sugar

Again, we use unique because cinnamon is repeated (and maybe a bit overrated), since it’s used in two parts of the pie. That way, we get all ingredients together, and we can subject them to additional processing (using a grammar, for instance, which we’ll get to later).

You will scrape if there’s absolutely no way out of it. If you get to use an API, things will be much easier. We’ll get to that next.

Recipe 9-3. Use a Web API to Get Information from a Site

Problem

You need to download information from a site that makes data available via a REST API.

Solution

Use a web client in Raku, such as WWW or Cro::HTTP, or a specific module for the API if it exists in the ecosystem.

How It Works

Recipes and food, in general, are thriving areas on the web.

---

**Note** Even more so during the Coronavirus pandemic, when a big part of the world was confined at home and had more time to spare.

There’s a constellation of websites that can be consulted daily, but that has also created a whole industry of services that provide content for those sites, as well as added value that goes from composing recipes to checking all kinds of information about ingredients. There’s Yummly, which provides a pay-per-use API, and many others. We’ll
settle for Edamam. It does provide a free tier, limited to five petitions/minute, if you sign up as a developer. So please do so for this recipe at https://developer.edamam.com/edamam-recipe-api and get an app ID and an API key.

use Cro::HTTP::Client;
use URI::Encode;

my $appID = %*ENV{'EDAMAM_APP_ID'};
my $api-key = %*ENV{'EDAMAM_API_KEY'};
my $api-req = "\&app_id=$appID\&app_key=$api-key";
my $ingredient = @*ARGS[0] // "water";
my $cro = Cro::HTTP::Client.new(base-uri => "https://api.edamam.com/");
my $response = await $cro.get("search?q=" ~ uri_encode($ingredient) ~ $api-req);
my %data = await $response.body;

say %data<hits>.map( *<recipe><label> ).join: "\n";

Cro is an amazing piece of work, a framework for distributed applications that includes all kinds of goodies for different protocols, including of course this HTTP client we are using here. Unlike the clients we have been using before, it’s asynchronous. We’ve worked with asynchrony before, but in this case it’s entirely appropriate. You don’t really know when the response to a request is going to arrive, and keeping the rest of the program hanged up waiting for it will result in low performance. In cases where you’ll be doing a single request and processing it serially, it’s probably okay (as we have done in the previous recipes) to just fire the request and wait for the result; as a matter of fact, this is what we do here. But later.

The first block of statements will set up the necessary variables, taking as usual the query string from the command line if it exists. You’ll need to define environment variables by copying and pasting them from the Edamam account. It won’t work if these don’t have valid values. Since all requests will use these two values in precisely that order, we set up the $api-req variable to reuse it later.

Cro::HTTP::Client sets up a client with a base URL. It will reuse connections if the version of the protocol allows it, which is also a difference with respect to the other two modules we’ve used before. Production and features-wise, Cro is way ahead of other modules in the same niche.

CHAPTER 9  CLIENT-SIDE WEB AND APIs
Since it’s asynchronous, a request will return a promise. We `await` on that promise to get to the response. But the response itself is a promise too: it’s a `connection` and we’ll need to await on it again to get the body of the response. That explains the two awaits in sequence. The API uses `GET` to access the search function, and it uses `q` as a parameter for the query string. We build an URL, which is convenient in this case (and an alternative to giving the method a hash), since the API asks explicitly for the authentication parameters in that order (and you can’t be sure how the keys of a hash will be ordered otherwise).

`Cro::HTTP::Client` will even decode the body of the response for you and give you a Rakü data structure. By default, Edamam API will return ten `hits`, and up to a 100 for the free tier. We’ll settle for the first page; as a matter of fact, this particular query string returns several thousand hits. “hits” is one of the keys the resulting data structure returns. But we’re just interested in the recipes that are returned. The data structure will be a hash, and it will store the hits as an array under the `hits` key. Among other things, every hit will have a `recipe` key, and that recipe will have a `label` key, which will be its description. So the last statement runs a map over these hits and extracts the names of these recipes. It will go like this:

- Summer water
- Pineapple Coconut Water
- Water Toast
- Water Kefir from 'Mastering Fermentation'
- Grapefruit Sparkling Water
- Coconut-Water Gelatin
- Cucumber-Orange Water Recipe
- Tomato Water Pasta
- Rose Water Marshmallows recipes
- Rose Water Syrup

Hmm, summer water. Can’t wait to cook that. Will I be able to buy dehydrated water at my local supermarket?

In general, most APIs will use REST, and you will be able to deal with it using `Cro::HTTP::Client`. Authentication will be done in different ways, in most cases adding them as metadata to the request. The essence of dealing with APIs is there. Get your arguments together, build the request (including header metadata if you need it), and fire it, decoding the response.
In some limited cases, you will have a Raku module custom-made for a particular API: `API::Discord` will handle that conversational system, there’s Twitter for that social network, `GlotIO` for `Glot.io`, and even a thin wrapper for Wikidata, called `Wikidata::API`, that was published by yours truly.

Wikidata is the less beaten track of Wikipedia devoted to, well, data. It stores data and relationships between items of data. It includes what ingredient was used to make which recipe. As the rest of the WikiRealm, it’s crowd-sourced, so your mileage might vary. The good thing is that it has an API that needs no authentication, based on a query language called SPARQL. This query, for instance, would return all recipes that include garlic:

```sparql
SELECT ?recipe ?recipeLabel
WHERE {
  ?recipe wdt:P31*/wdt:P279* wd:Q219239;
    wdt:P527 wd:Q21546392.
  SERVICE wikibase:label { bd:serviceParam wikibase:language "en", "fr". }
}
ORDER BY UCASE(STR(?recipeLabel))
```

The prefix `wdt` is used for relationships, and `wd` is for data. The two fixed pieces of data we have are `Q219239` which is, well, recipe (check it out in its URI `https://www.wikidata.org/wiki/Q219239` or by simply searching for “recipe” on `www.wikidata.org`). The two relationships are an instance of or a subclass of. A recipe for carbonara sauce is an instance of a recipe. P527 means “is composed of,” and Q21546392 is garlic. Check out the Q prefix for entities and P for relations between them. So essentially we are saying “Give me all things that seem to be a recipe and include garlic.” Spoiler: There are only two at the time of writing this. We’ll need this script to get to them:

```perl
use Wikidata::API;
my $query = "Chapter-9/ingredients.sparql".IO.slurp;
my $recipes-with-garlic = query($query);
say "Recipes with garlic:
", $recipes-with-garlic<results><bindings>
  .map: { utf8y( $_<recipeLabel><value>) };
```
sub utf8y ( $str ) {
    Buf.new( $str.ords ).decode("utf8")
}

There’s not much to it, really. Read the SPARQL query, launch a query, show the results. The results are in a relatively complex data structure, but the only part that’s interesting is the recipeLabel key that, as you have seen before, was created by the query. The rest is boilerplate (results will be the key storing the results, and bindings will show the different variables bound to results, including recipeLabel).

We had to create a little subroutine, utf8y, to deal with the results, since the JSON modules used cannot. That routine breaks down a string into its characters, reconstructs it, and returns it encoded in utf8. That will print the following:

Recipes with garlic:
(Anchoïade ratatouille)

That first word is the one that caused the existence of that routine. Those are the two only recipes that seem to use garlic. They had to be French, of course.

If you know SQL or another query language, SPARQL is not too difficult to learn, and it can really help you with lots of mundane things. In this article for the Raku Advent Calendar, Santa used it to check if what the boys and girls were asking for in their letters was actually an object https://perl6advent.wordpress.com/2017/12/03/letterops-with-perl6/.

In general, APIs will help you enrich your applications, and working with them is easy with Raku.

Recipe 9-4. Check IPs and Addresses by Querying Internet Services

Problem

You have an IP and you need to check if it’s on or if a service is available.
Solution

You can use Net::IP to manipulate addresses or Net::IP::Parse to check them, as well as IP::Random to generate random IP addresses. Sys::IP will give you local IP addresses. You can also use whois to map domains to names. In many cases, there will be a Raku module available to check a certain service, in others you’ll have to create your own querying service for APIs using one of the recipes discussed in this chapter.

How It Works

Working with IP addresses involves querying system services, as well as using protocols such as TCP for making calls and examining what’s returned. For instance, very often we need to check if a service is running, and if it’s not, do something like log a failure event or take other measures like send an email. In general, doing system calls, and doing them in a system-independent way, is not easy.

Services generally are mapped to “ports,” which are addresses within the system that usually have a agreed-upon number. These numbers are published and you usually ensure that if you need to run something, you avoid them and create your own.

Anyway, putting this together means that if you want to check if certain services are running on your system, you’ll need to check if there’s anything in that port responding to a certain protocol, which is usually TCP. You can do that with this simple script

```perl
use Sys::IP;
use Services::PortMapping;
use CheckSocket;

my $this-ip = Sys::IP.new.get_default_ip();

for <www-http ssh> -> $service {
  if check-socket(%TCPPorts{$service},$this-ip) {
    say "Your service $service is running in port %TCPPorts{$service}";
  } else {
    say "Apparently, your service $service is not running";
  }
}
```
The script is simple because there are three modules that hide all the complexity from you. Sys::IP finds the IP or IPs that the local system has, the ones that can be used to check (of course, there’s always 127.0.0.1). Services::PortMapping exports four hashes that map standard service names to ports and the reverse. And finally, CheckSocket checks if there’s a TCP service running in a port in a certain address.

So we do that in sequence: first we find local IPs via get-default_ip, and then we check two common services, ssh and http (whose standard name, according to the “Service name and Transport Protocol Port Number Registry,” is http, www, or www-http), by running check-socket over them. Different messages will be printed if they are running or not. In my case, it will print the following:

Your service www-http is running in port 80
Your service ssh is running in port 22

Which is how I discovered I had Apache httpd installed and running all along.

Raku is a general purpose language and it is as adequate for system-level tasks as the next one. It’s true that there are not as many modules in the ecosystem as in other languages that have taken that niche like Perl or Python, but there’s extensive support in the Net:: namespace of the ecosystem for protocols like DNS or BGP. Other protocols, like ICMP, are unfortunately missing. However, the Raku ecosystem is growing by several modules every week, so that might not be true by the time you read this.