12. Creating a Webservice, and Declarative APIs

Nowadays, it seems like every software must be reachable through the network, in the cloud.

In this spirit, we'll take a look at a simple way to create a web service in Raku by using Cro[1], a set of libraries that makes it easy to write asynchronous web clients and services. The name Cro comes from a terrible pun: it allows me to write microservices, my *cro* services.

Later in this chapter we'll take a look at how Cro achieves its declarative API.

12.1 Getting Started with Cro

We'll reuse the code from chapter 4, which converts UNIX timestamps into ISO-formatted datetime strings and vice versa, and now expose them through HTTP.

The first part, converting from UNIX tiemstamp to ISO date, goes like this:

```
use Cro::HTTP::Router;
use Cro::HTTP::Server;

my $application = route {
    get -> 'datetime', Int $timestamp {
        my $dt = DateTime.new($timestamp);
        content 'text/plain', "$dt\n";
    }
}
my $port = 8080;
my Cro::Service $service = Cro::HTTP::Server.new(
    :host<0.0.0.0>,
    :$port,
    :$application
);

$service.start;
say "Application started on port $port";
react whenever signal(SIGINT) { $service.stop; done; }
```

In this example, we see the subroutines `route`, `get` and `content` that are exported by the modules `Cro::HTTP::Router`.

[1]https://cro.services/

`route` takes a block as an argument, and returns an application. Inside the block, we can call `get` (or other HTTP verb functions such as `post` or `put`) to declare *routes*, pieces of code that Cro calls for us when somebody requests the matching URL through HTTP.

Here the route declaration starts as `get -> 'datetime', Int $timestamp`. The `->` arrow introduces a signature, and Cro interprets each argument as a part of a slash-delimited URL. In our example, the URL that matches the signature is `/datetime/` followed by an integer, like `/datetime/1578135634`. When Cro receives such a request, it uses the constant string `datetime` to identify the route, and puts the `1578135634` into the variable `$timestamp`.

The logic for converting the timestamp to a `DateTime` object is familiar from chapter 4, the only difference is that instead of using `say` to print the result to standard output, we use the `content` function to serve the back to the HTTP requester. This is necessary because each HTTP response needs to declare its content type, so that for example a browser knows whether to render the response as HTML, as an image etc. The `text/plain` content type denotes, as the name says, plain text that is not to be interpreted in any special way.

The code after is classical plumbing: it instantiates a `Cro::HTTP::Server` object at a given TCP port (here 8080, feel free to change it to your liking) and our collection of one meager route, and then tells it to start serving HTTP requests. We chose the host `0.0.0.0` (which means: bind to all IP addresses) so that if you run the application in a Docker container, it can be reached from the host. If you do not use docker, using `127.0.0.1` or `localhost` is safer, as it doesn't expose the application to other machines in the network.

Then finally, the shocker line:

```
react whenever signal(SIGINT) { $service.stop; done; }
```

The `signal()` function returns a Supply[2], which is an asynchronous data stream, here of inter-process communication signals. `signal(SIGINT)` specifically only emits events when the process receives the `INT` or *interrupt* signal, which you can typically create by pressing the keys Ctrl+C in your terminal.

`react` is usually used in its block form, `react { .... }`, and shorted here because it applies to only one statement. It runs and dispatches supplies in `whenever` statements until the code calls the `done` function (or all the streams finish, which doesn't happen for the signal streams).

So, inside `react`, `whenever signal(SIGINT) { ... }` calls the code marked by `...` each time the `SIGINT` signal is received – in which case we stop the HTTP server and exit the `react` construct.

If you want to handle other signals, such as SIGTERM (which is used by the `kill` system command), you can replace `signal(SIGIINT)` by

```
signal(SIGINT).merge(signal(SIGTERM)
```

All of this is a complicated way to exit the program when somebody presses Ctrl+C.

[2]https://docs.raku.org/language/concurrency#Supplies

Due to the asynchronous nature of Cro, you could also do other things here, like processing other supplies in the `react` block (like period timers, streams of file change events), while the HTTP server is merrily running.

To run this, first install the `cro` and `Cro::HTTP::Test` modules with the `zef` package manger:

```
$ zef install --/test cro Cro::HTTP::Test
```

where the `--/test` option tells `zef` not to run the module tests, which both take a long time, and require some local infrastructure that you are unlikely to have available.

If you use Docker to run your raku programs, you can use the image `moritzlenz/raku-fundamentals`, which builds on Rakudo Star and includes the necessary `cro` modules. If you go that route, you must also use docker's `--expose` command line option to make the service available on the host, otherwise it's only reachable from within the container. Then the command line looks like this:

```
$ docker run --rm --publish 8080:8080 -v $PW/raku -w /raku \
    -it moritzlenz/raku-fundamentals raku datetime.p6
```

We can test the service on the command line with a HTTP client like `curl`[3]:

```
$ curl http://127.0.0.1:8080/datetime/1578135634
2020-01-04T11:00:34Z
```

12.2 Expanding the Service

Now that we have a minimalistic but working service, we can incorporate the conversion from an ISO datetime string to a UNIX timestamp. We'll just be looking at the `route` block, everything stays the same. Here's one approach to implement it:

```
my token date-re {
    ^
    \d**4 '-' \d**2 '-' \d** 2 # date part YYYY-MM-DD
    [
        ' '
        \d**2 ':' \d**2 ':' \d**2 # time
    ]?
    $
}

my $application = route {
    get -> 'datetime', Int $timestamp {
        my $dt = DateTime.new($timestamp);
        content 'text/plain', "$dt\n";
```

[3]https://curl.haxx.se/

```
    }
    get -> 'datetime', Str $date_spec where &date-re {
        my ($date_str, $time_str) = $date_spec.split(' ');
        my $date = Date.new($date_str);
        my $datetime;
        if $time_str {
            my ( $hour, $minute, $second ) = $time_str.split(':');
            $datetime = DateTime.new( :$date, :$hour, :$minute, :$second);
        }
        else {
            $datetime = $date.DateTime;
        }
        content "text/plain", $datetime.posix ~ "\n";
    }
}
```

We start with a regex that defines how the datetime format that we want to accept looks like, and store it in the variable `&date-re`. Then in the `route { ... }` block, we add a second `get` call with this signature:

```
get -> 'datetime', Str $date_spec where &date-re { ... }
```

This defines a second route, under a similar url as before, `/datetime/YYYY-MM-DD HH:MM:SS` (where the time part is optional). The logic is again copied from chapter 4, so no surprises here. The only difference is that with the command line application, the command line parser split the date and time part for us, which we now explicitly do with a call to `.split(' ')`.

When we test this code with `curl` or a browser, we need to remember that we cannot directly include a space in an URL directly, but need to escape that as `%20`. After starting our extended service, we can call `curl` again to test it:

```
$ curl http://127.0.0.1:8080/datetime/2020-01-04%2011:00:34
1578135634
```

Most modern webservices tend to respond with JSON data, which we can achieve by passing a JSON-serializable data structure like a hash to the `content` function:

```
    # in the first route
    content 'application/json', {
        input  => $timestamp,
        result => $dt.Str,
    }

    # in the second route
    content "application/json", {
        input  => $date_spec,
        result => $datetime.posix,
    }
```

12.3 Testing

Testing a web application can be a bit of a pain sometimes. You have to start the application server, but first you need to find a free port where it can listen, and then you make your requests to the server, and tear it down afterwards.

With a tiny bit of restructuring and the `Cro::HTTP::Test` module, all of this can be avoided.

For the restructuring, let's put our call to `route` into a subroutine of our own, and put the server setup into a `MAIN` function:

```
sub routes() {
    return route {
        # same route definitions as before
    }
}

multi sub MAIN(Int :$port = 8080, :$host = '0.0.0.0') {
    my Cro::Service $service = Cro::HTTP::Server.new(
        :$host
        :$port,
        application => routes(),
    );

    $service.start;
    say "Application started on port $port";
    react whenever signal(SIGINT) { $service.stop; done; }
}
```

You can start the HTTP server as before, now with the added benefit of being able to override the port and the host (the IP that the server listens on) through the command line.

Our goal was testing, so let's add another `MAIN` multi for that.

```
multi sub MAIN('test') {
    use Cro::HTTP::Test;
    use Test;
    test-service routes(), {
        test get('/datetime/1578135634'),
            status => 200,
            json => {
                result => "2020-01-04T11:00:34Z",
                input => 1578135634 ,
            };
        test get('/datetime/2020-01-04%2011:00:34'),
            status => 200,
            json => {
                input => '2020-01-04 11:00:34',
                result => 1578135634,
            };
    }
    done-testing;
}
```

We meet our newest friend, sub `test-service`. We call it with two arguments, the routes to be tested and a block with our tests. Inside this block, the `get()` routine calls the appropriate routes without any server being started, and returns an object of type `Cro::HTTP::Test::TestRequest`. With the `test` routine we can check that this test response fulfills our expectations, here regarding the response code (`status`) and the JSON response body.

We can run the tests by adding the `test` command line parameter, and get test output like this:

```
$ raku datetime.p6 test
    ok 1 - Status is acceptable
    ok 2 - Content type is recognized as a JSON one
    ok 3 - Body is acceptable
    1..3
ok 1 - GET /datetime/1578135634
    ok 1 - Status is acceptable
    ok 2 - Content type is recognized as a JSON one
    ok 3 - Body is acceptable
    1..3
ok 2 - GET /datetime/2020-01-04%2011:00:34
1..2
```

Each call to `test` produces one test in the output, and a subtest (designated by indentation) for each individual comparison.

12.4 Adding a Web Page

We have our mini web service at a place now where another program can talk to it comfortably through JSON over HTTP, but that's not really friendly towards end users.

As a demonstration for a possible user interface, let's add an HTML page that can be viewed in a browser. Since we'll handle the data through HTTP requests triggered from javascript, we can get away with serving static files. Cro offers a helper for that called `static`, which replaces our calls to `content`. Let's add these two routes to he the `route { ... }` block:

```
        get -> { static 'index.html'; }
        get -> 'index.js' { static 'index.js'; }
```

The first one has an empty signature, and so corresponds to the `/` or root URL, and serves the file `index.html`. The second one serves a file called `index.js` with the same URL.

The `static` helper can do more, like serving whole directories while preventing malicious path traversal[4], but for our cases, the simple form is enough.

File `index.html` should be placed directly next to the raku script, and can look like this:

```
<html>
  <head>
    <title>Datetime to UNIX timestamp conversion</title>
     <script
       src="https://code.jquery.com/jquery-3.5.1.min.js"
       crossorigin="anonymous"></script>
     <script type="text/javascript" src="/index.js"></script>
  </head>
  <body>
    <h1>Convert UNIX timestamp to ISO datetime or vice versa</h1>
      <form>
        <label for="in">Input</label>
        <input name="in" id="in" placeholder="2014-01-14 10:12:00"></input>
        <button id="submit">Submit</button>
    </form>
    <h2>Result</h2>
    <ul id="result"></ul>
  </body>
</html>
```

The visible elements are just some headings, an input form and a button for submitting it, as well as an empty list for the results.

We'll add some javascript in `index.js` to bring it to life:

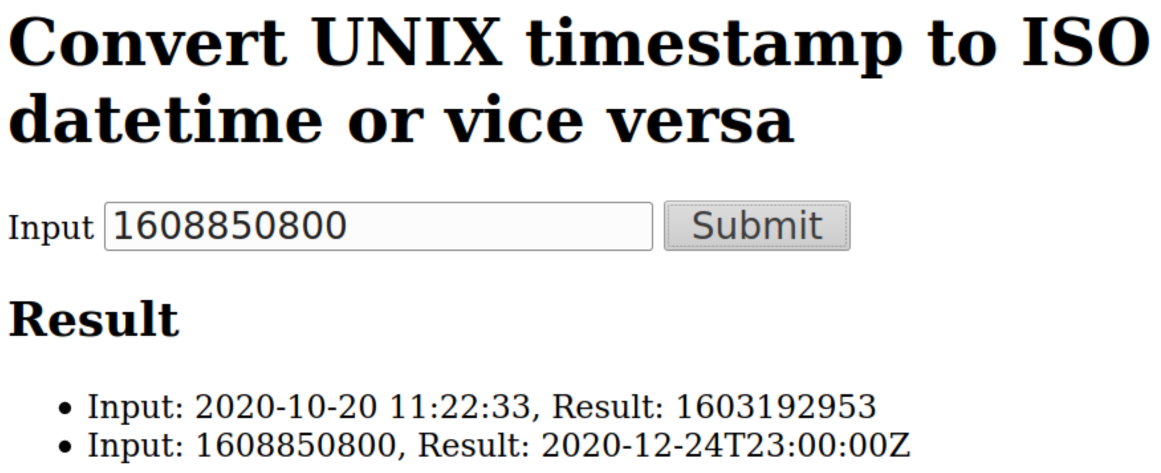
```
$(document).ready(function() {
    $('#submit').click(function(event) {
        var val = $('#in').val();
        $.get('/datetime/' + val, function(response) {
            $('#result').append(
                '<li>Input: ' + val  +
                ', Result: ' + response['result'] +
                '</li>');
        });
        event.preventDefault();
    });
});
```

This piece of code uses the jQuery[5] library and subscribes to click events on the button. When the button is pressed, it reads the text from the input element, submits it asynchronously towards

[4] https://cro.services/docs/reference/cro-http-router#Serving_static_content
[5] https://jquery.com/

the URL `/datetime/` followed by the input, and appends the result as list items to the unordered list (`<ul>`).

This is far from perfect, as the web application is missing error handling and visual appeal, but it does illustrate how you can have a pretty machine-focused API endpoints in your application, and then put a user interface on top that uses HTML and javascript.


Convert UNIX timestamp to ISO datetime or vice versa

Input 1608850800 Submit

Result

- Input: 2020-10-20 11:22:33, Result: 1603192953
- Input: 1608850800, Result: 2020-12-24T23:00:00Z

The minimalistic web frontend to the datetime conversion app.

There are several projects that aim to keep the javascript code composable and maintainable, like Vue.js[6], angular[7] and React[8]. In fact, the Cro documentation comes with a tutorial for building a single page application with React and redux[9], which you should follow if you want to dive deeper into this subject.

12.5 Declarative APIs

We could now grow our application with more routes, authentication[10] and more, but instead I want to draw attention to how Cro creates its APIs.

The syntax for routes looks like this:

```
get -> 'datetime', Int $timestamp { ... }
```

The `get` here is a function call, so we could also write this as

```
get(-> 'datetime', Int $timestamp { ... });
```

The arrow `->` introduces a block with a signature. It's not really important that it's a block, an ordinary subroutine works as well:

[6]https://vuejs.org/
[7]https://angular.io/
[8]https://reactjs.org/
[9]https://cro.services/docs/intro/spa-with-cro
[10]https://cro.services/docs/http-auth-and-sessions

```
get( sub ('datetime', Int $timestamp) { ... } )
```

In fact, there is no need to put the declaration of the subroutine inside the `get` call. We could have written instead:

```
sub convert-from-timestamp('datetime', Int $timestamp) {
    my $dt = DateTime.new(+$timestamp);
    content 'application/json', {
        input  => $timestamp,
        result => $dt.Str,
    }
}
sub convert-from-isodate('datetime', Str $date_spec where $date-re) {
    # omitted for brevity
}

my $application = route {
    get &convert-from-timestamp;
    get &convert-from-isodate;
}
```

`get` is a function that takes another function as an argument, something we've seen in chapter 12. In contrast to those examples, `get` doesn't just call the function it receives, it *introspects* its signature to figure out when to call it.

We could do that too. In the context of the previous example, you could write

```
my @params = &convert-from-timestamp.signature.params;
say @params.elems;                  # => 2
say @params[0].type;                # => (Str)
say @params[0].constraints;         # => all(datetime)
```

Through the `.signature.params` method chain we obtain a list of Parameter[11] objects representing each function parameter; we can ask them for the type, additional constraints (like the string `datetime` used on the first parameter), the variable name and many more properties.

Just like `get`, `route { ... }` is a function that calls its argument function after a bit of setup. It sets up a dynamic variable that `get` latches on to, which enables `route` to return an object with information about all the calls to `get`, and so all the routes.

To illustrate this principle, let's try to write some functions that allow you to write small dispatchers based on the types, that is, call the first function with a matching type:

[11]https://docs.raku.org/type/Parameter

~~~
my &d = dispatcher {
    on -> Str $x { say "String $x" }
    on -> Date $x { say "Date $x" }
}
d(Date.new('2020-12-24'));
d("a test");

To make this aspirational example valid Raku code, we need two functions, `dispat\
cher` and `on`, that both take one callable block as an argument. `dispatcher` ne\
eds to declare a dynamic variable, and `on` performs some sanity checks and adds \
its argument to the dynamic variable:

```perl6
sub dispatcher(&body) {
    my @*CASES;
    body();
    my @cases = @*CASES;
    return sub (Mu $x) {
        for @cases -> &case {
            if $x ~~ &case.signature.params[0].type {
                return case($x)
            }
        }
        die "No case matched $x";
    }
}

sub on(&case) {
    die "Called on() outside a dispatcher block"
        unless defined @*CASES;
    unless &case.signature.params == 1 {
        die "on() expects an block with exactly one parameter"
    }
    @*CASES.push: &case;
}
~~~

Sub `dispatcher` has to copy the contents of its dynamic variable into a lexical variable `my @cases`, because the dynamic variable is scoped to the execution time of the function it is declared in, so it ceases to exist after function `dispatcher` has returned. But `dispatcher` needs the contents to do its work, iterating through the cases and calling the first one that matches. It does this in an anonymous function that it returns, so that the programmer can reuse the matcher in several code locations.

On your first read of code using Cro, you might have thought that the `route` and `get` construct looked like language extensions; instead they turn out to be cleverly named functions that receive other functions as arguments. You can use the same techniques in your own libraries and frameworks to create interfaces that feel natural to the programmer that uses them. t ##

Summary

With the Cro libraries, you can expose functionality pretty easily through HTTP. First you create some routes (code that is called by Cro when somebody requests the matching URL) in a `route { ... }` block, and then pass the return value from that block to the HTTP server. You start the server, and you're done.

Each route communicates its response by calling the `content` function, specifying both the content type and the response body; JSON serialization happens automatically for the appropriate content type.

A user interface can be created through static HTML and javascript, possibly with the help of javascript application frameworks.

We have also seen how Cro achieves a natural feel for its API by providing higher-order functions, functions that receive other functions as arguments, and perform introspection on the signatures of these functions.