Things you may not know about Cro

Jonathan Worthington | Edument
Cro
Libraries for building distributed systems in Raku
Asynchronous pipeline concept at its core
Popular for web services and web applications

Me
Raku runtime and compiler architect and developer
Leader of Edument in Prague
Comma IDE product manager
Cro founder and architect
Consulting focus on developer tooling and Raku
Cro isn't just for HTTP server-side stuff. It includes a HTTP client too!
And what's more...

It uses the very same Request and Response classes on the client side as on the server side.

It's offers an asynchronous API.
A simple request

# Use the module
use Cro::HTTP::Client;

# Get the response object (available as soon as the headers
# are received)
my $response = await Cro::HTTP::Client.get('https://raku.org');
say "{{.name}: {{.value}}}" for $response.headers;

# Get the response body (once we have received it all)
my $body = await $response.body;
say "Body is $body.chars() chars long";
# Use the module
use Cro::HTTP::Client;

# Make a request to an endpoint that produces JSON
my $response = await Cro::HTTP::Client.get:
    'https://api.github.com/users/MoarVM/repos';

# Thanks to the content-type header, automatically deserialized
my @repos := await $response.body;

# So we can do this:
say bag @repos.map(*<language>); # Bag(C(7) HTML(2))

(We can write and plug in body parsers for other kinds of response if desired)
# Make a request for a (sort of) large file
use Cro::HTTP::Client;
my $response = await Cro::HTTP::Client.get:
    'http://jnthn.net/papers/2020-cic-rakuast.pdf';

# Receive the body asynchronously
my $expected = $response.header('content-length');
react whenever $response.body-byte-stream -> Blob $chunk {
    # Report how much we have received
    state $so-far += $chunk.bytes;
    say "$so-far bytes ({{Int(100 * $so-far / $expected)}%})";
}
Set defaults for all requests at construction time

my constant ACCESS_TOKEN = 'REDACTED';
my $client = Cro::HTTP::Client.new:
  base-uri => 'https://api.github.com',
  auth => { username => 'jnthn', password => ACCESS_TOKEN };

# Make a request that uses the defaults.
my $response = await $client.post: '/gists',
  content-type => 'application/vnd.github.v3+json',
  body => {
    description => 'Hello world',
    files => {
      'hello.raku' => { content => 'say "Hello world";' } }
  }
);
say await($response.body)<html_url>;
And more...

Configurable redirect following
Pluggable body parsers/serializers
(JSON, form, and multipart included as standard)
Persistent connections
HTTP/2.0
Proxy support
Cookie jar
Cro does WebSockets, both server-side and client-side
Deal with WebSockets using a Raku Supply-based API

Neatly integrated with the Cro HTTP router on the server side
Example: PollShare

A WebSocket API where many clients can connect and send URLs to be polled

If the content at the URL changes, we notify the client

Only poll each URL once, even if many clients are interested
<to the code>
Writing an OpenAPI 3 specification?

There's a Cro module to ease implementing it!
Don't repeat the routes, just mention operation IDs

Validation of incoming requests (and optionally of outgoing responses)
A route specification

/clone_dataset:
  post:
    summary: Clone a dataset
    operationId: cloneDataset
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/CloneDataset"
    responses:
      '204':
        description: Dataset cloned
      '409':
        description: Dataset clone failed
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Error"
A route specification

/clone_dataset:
  post:
    summary: Clone a dataset
    operationId: cloneDataset
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/CloneDataset"
    responses:
      '204':
        description: Dataset cloned
      '409':
        description: Dataset clone failed
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Error"
Type specification

CloneDataset:
  type: object
  required:
    - newUsername
    - oldDataset
    - newDataset
  properties:
    newUsername:
      description: Owner of the cloned dataset
      type: string
    oldDataset:
      description: Name of the source dataset
      type: string
    newDataset:
      description: Name of the cloned dataset
      type: string
Load the Cro OpenAPI module

```perl
use Cro::HTTP::Router;
use Cro::OpenAPI::RoutesFromDefinition;

(Which is built using OpenAPI::Model, OpenAPI::Schema::Validate, which are not tied to Cro and provide a generic OpenAPI core implementation)
```
Write a sub...

sub api-routes(Str $schema-path, Agrammon::Web::Service $ws) {
    ...
}

(Which receives the path to the schema, along with an object that carries the business logic; as with Cro route blocks, we should keep them about HTTP, and injecting the business logic object aids testability)
...specify the schema...

sub api-routes(Str $schema-path, Agrammon::Web::Service $ws) {
  openapi $schema-path.IO, {
    ...
  }
}

(In this case by providing an IO::Path to the OpenAPI schema file, which will be loaded; alternatively, a string containing the schema itself may be provided)
sub api-routes(Str $schema-path, Agrammon::Web::Service $ws) {
  openapi $schema-path.IO, {
    operation 'cloneDataset', -> {
      # ...
    }
    # ...
  }
  # ...
}

(Meaning we leave knowledge about the URL structure exclusively in the OpenAPI specification, rather than repeating it here)
...take the session/user...

sub api-routes(Str $schema-path, Agrammon::Web::Service $ws) {
    openapi $schema-path.IO, {
        operation 'cloneDataset', -> LoggedIn $user {
            # ...
        }
        # ...
    }
    # ...
}

(This isn't anything to do with OpenAPI, just the usual Cro way of obtaining the current session using an initial parameter)
...destructure the request...

```perl
sub api-routes(Str $schema-path, Agrammon::Web::Service $ws) {
    openapi $schema-path.IO, {
        operation 'cloneDataset', -> LoggedIn $user {
            request-body -> ( :newUsername($new-username),
                :oldDataset($old-dataset),
                :newDataset($new-dataset) ) {
                # ...
            }
        }
    }
    # ...
}
```

(Safe in the knowledge that it has been validated according to the schema)
...call the business logic...

```perl
sub api-routes(Str $schema-path, Agrammon::Web::Service $ws) {
    openapi $schema-path.IO, {
        operation 'cloneDataset', -> LoggedIn $user {
            request-body -> ( :newUsername($new-username),
                :oldDataset($old-dataset),
                :newDataset($new-dataset) ) {
                $ws.clone-dataset($user, $new-username, $old-dataset,
                 $new-dataset);
            }
        }
    }
    # ...
}
```
...and map errors to HTTP

```perl
sub api-routes(Str $schema-path, Agrammon::Web::Service $ws) {
    openapi $schema-path.IO, {
        operation 'cloneDataset', -> LoggedIn $user {
            request-body -> ( :newUsername($new-username),
                              :oldDataset($old-dataset),
                              :newDataset($new-dataset) ) {
                $ws.clone-dataset($user, $new-username, $old-dataset,
                                $new-dataset);
                CATCH {
                    when X::Agrammon::DB::Dataset::AlreadyExists {
                        conflict 'application/json', %( error => .message );
                    }
                }
            }
        }
    }
}
```
Use it in our top-level routes

sub routes(Agrammon::Web::Service $ws) is export {
  my $schema = 'share/agrammon.openapi';
  route {
    # The OpenAPI-based routes
    include api-routes($schema, $ws);
    # Static content routes (HTML, CSS, JS)
    include static-content($root);
    # Various non-API routes
    include application-routes($ws);
  }
}

(In simpler cases, we can pass the OpenAPI routes directly to Cro::HTTP::Server)
Cro::HTTP::Test eases testing our Cro route implementations

(Or we can use it against any HTTP URL that we want to write tests for)
Gather our trusty testing tools...

# The usual test stuff (for plan, subtest, etc.)
use Test;

# The Cro HTTP testing module
use Cro::HTTP::Test;

# For mocks/stubs of our business logic
use Test::Mock;
Create a fake user session, to test routes needing auth

my $fake-auth = mocked(
    # The session type
    Agrammon::Web::SessionUser,
    # Fake some of its methods
    returning => { :id(42), :logged-in, }
);

(Not needed if you have no such routes to test)
Create a mock of the business logic object

my $fake-service = mocked(Agrammon::Web::Service);

(We can fake return values, even computing them based on the input values, or exception throws if we want; by default, we get an object that accepts, but ignores, the method calls, just logging them)
Create the Cro routes we'll test against and fake the auth

subtest 'Clone dataset' => {
  test-service routes($fake-service), :$fake-auth, {
    ...
  }
}

(This is where having the routes sub take an object implementing the business logic shows its use in letting us test our routes!)
...specify the path we'd like to test against...

```perl
subtest 'Clone dataset' => {  
  test-service routes($fake-service), :$fake-auth, {  
    test-given '/clone_dataset', {  
      ...  
    }  
  }  
}
```

(We don't have to do it this way if there's just one request; `test-given` is useful for many tests of one endpoint, common headers, etc.)
...perform a test request and assert against the result...

```php
subtest 'Clone dataset' => {
    test-service routes($fake-service), :$fake-auth, {
        test-given '/clone_dataset', {
            test post(json => {
                :newUsername('foo'),
                :oldDataset('DatasetC'),
                :newDataset('DatasetD')
            }),
            status => 204;
        }
    }
    ...
}
```
...and check we called the correct business logic

```
subtest 'Clone dataset' => {
    test-service routes($fake-service), :$fake-auth, {
        test-given '/clone_dataset', {
            test post(json => {
                :newUsername('foo'),
                :oldDataset('DatasetC'),
                :newDataset('DatasetD')
            }),
            status => 204;
        }
    }
    check-mock $fake-service,
    *.called('clone_dataset',
        with => \($fake-auth, 'foo', 'DatasetC', 'DatasetD'),
        times => 1);
}
```
Writing Raku using the Comma IDE?

It has some features especially for working with Cro
Thank you!

@ jonathan@edument.cz
W jnthn.net
Twitter jnthnwrthngtn
Gplus jnthn