Cro

One Year Later

Jonathan Worthington
Edument

At last year's Swiss Perl Workshop...

At last year's Swiss Perl Workshop...

...I revealed Cro

A set of libraries and tools for building distributed systems in Perl 6

- Designed for Perl 6
- Async from the ground up
- Initial HTTP/WebSocket focus

So, where are we one year later?

Well, we've got a logo now...

6

releases

> 800

commits

(across all Cro project repositories in the last year)

26

code contributors

(people who committed or had a merged PR)

2

sponsored features

(excluding Edument's sponsorship)

First usages in production

(we don't know how many; users survey?)

So, what's new?

`cro web`

A web UI for stubbing Cro services, automatically restarting them on changes, viewing their logs, etc.

<demo>

`Cro::HTTP::Test`

A library for writing automated tests for a HTTP service

Primarily for services built in Cro - but can be used against any HTTP endpoint

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

sub routes() is export {
    route {
        get -> 'is-prime', Int $number {
            content 'application/json',
                { :$number, :prime($number.is-prime) }
        }
    }
}
```

```
use Routes;
use Cro::HTTP::Test;

test-service routes(), {
    test get('/is-prime/42'),
            json => { number => 42, prime => False }
    test get('/is-prime/71'),
            json => { number => 71, prime => True }
}

done-testing;
```

```
use Routes;
use Cro::HTTP::Test;

test-service routes(), {
    test-given '/is-prime/', {
        test get('42'),
                json => { number => 42, prime => False }
        test get('71'),
                json => { number => 71, prime => True }
    }
}

done-testing;
```

```
test post('/get-prime', json => { min => 1000, max => 2000 }),
        status => 200,
        json => { .<number>.is-prime }
```

```
post -> 'get-prime' {
    request-body -> (:$min!, :$max!) {
        content 'application/json', {
            number => ($min..$max).grep(*.is-prime).pick
        }
    }
}
```

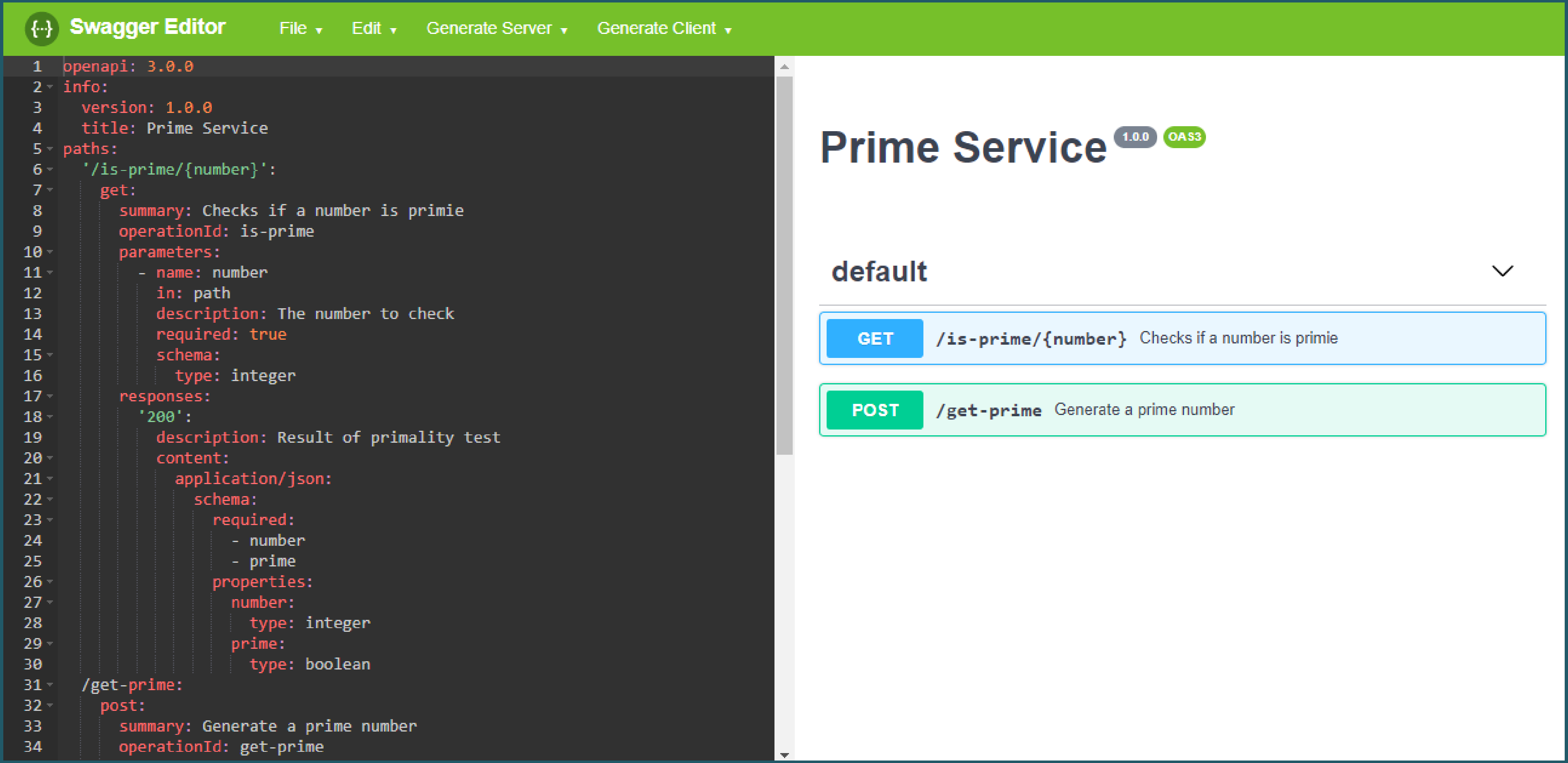
```
test post('/get-prime', json => { min => 1000, max => 2000 }),
        status => 200,
        json => { .<number>.is-prime }
test post('/get-prime', json => { min => 20, max => 22 }),
        status => 404;
```

```
post -> 'get-prime' {
    request-body -> (:$min!, :$max!) {
        with ($min..$max).grep(*.is-prime).pick -> $number {
            content 'application/json', { :$number }
        }
        else {
            not-found;
        }
    }
}
```

There's now support for implementing an OpenAPI specification using Cro

`Cro::OpenAPI::RoutesFromDefinition`

- Avoids repeating path and method
- Enforces validation rules
- Otherwise, just like a route block



```yaml
openapi: 3.0.0
info:
  version: 1.0.0
  title: Prime Service
paths:
  '/is-prime/{number}':
    get:
      summary: Checks if a number is primie
      operationId: is-prime
      parameters:
        - name: number
          in: path
          description: The number to check
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: Result of primality test
          content:
            application/json:
              schema:
                required:
                  - number
                  - prime
                properties:
                  number:
                    type: integer
                  prime:
                    type: boolean
  /get-prime:
    post:
      summary: Generate a prime number
      operationId: get-prime
```

Prime Service 1.0.0 OAS3

default

GET /is-prime/{number} Checks if a number is primie

POST /get-prime Generate a prime number

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Prime Service
paths:
  /is-prime/{number}:
    get:
      ...
  /get-prime:
    post:
      ...
```

```yaml
/is-prime/{number}:
   get:
     summary: Checks if a number is prime
     operationId: is-prime
     parameters:
       - name: number
         in: path
         description: The number to check
         required: true
         schema:
           type: integer
     responses:
       ...
```

```yaml
/is-prime/{number}:
   get:
     ...
     responses:
       '200':
         description: Result of primality test
         content:
           application/json:
             schema:
               required:
                 - number
                 - prime
               properties:
                 number:
                   type: integer
                 prime:
                   type: boolean
```

```yaml
/get-prime:
   post:
     summary: Generate a prime number
     operationId: get-prime
     requestBody:
       required: true
       content:
         application/json:
           schema:
             required:
               - min
               - max
             properties:
               min:
                 type: integer
               max:
                 type: integer
     responses:
       ...
```

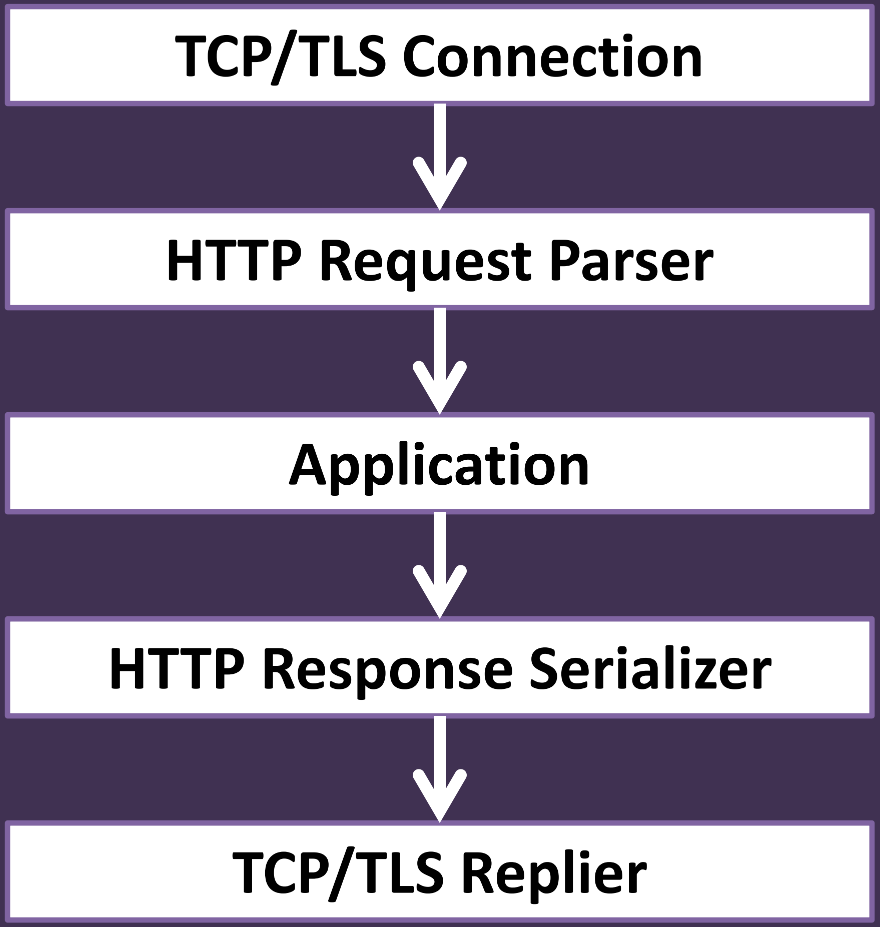
```yaml
/get-prime:
   post:
     ...
     responses:
       '200':
         description: Generated a prime in the range
         content:
           application/json:
             schema:
               required:
                 - number
               properties:
                 number:
                   type: integer
       '400':
         description: No prime in the range specified
```

```
openapi 'api.json'.IO, {
    ...
}
```

```
openapi 'api.json'.IO, {
    operation 'is-prime', -> Int $number {
        content 'application/json',
            { :$number, :prime($number.is-prime) }
    }

    ...
}
```

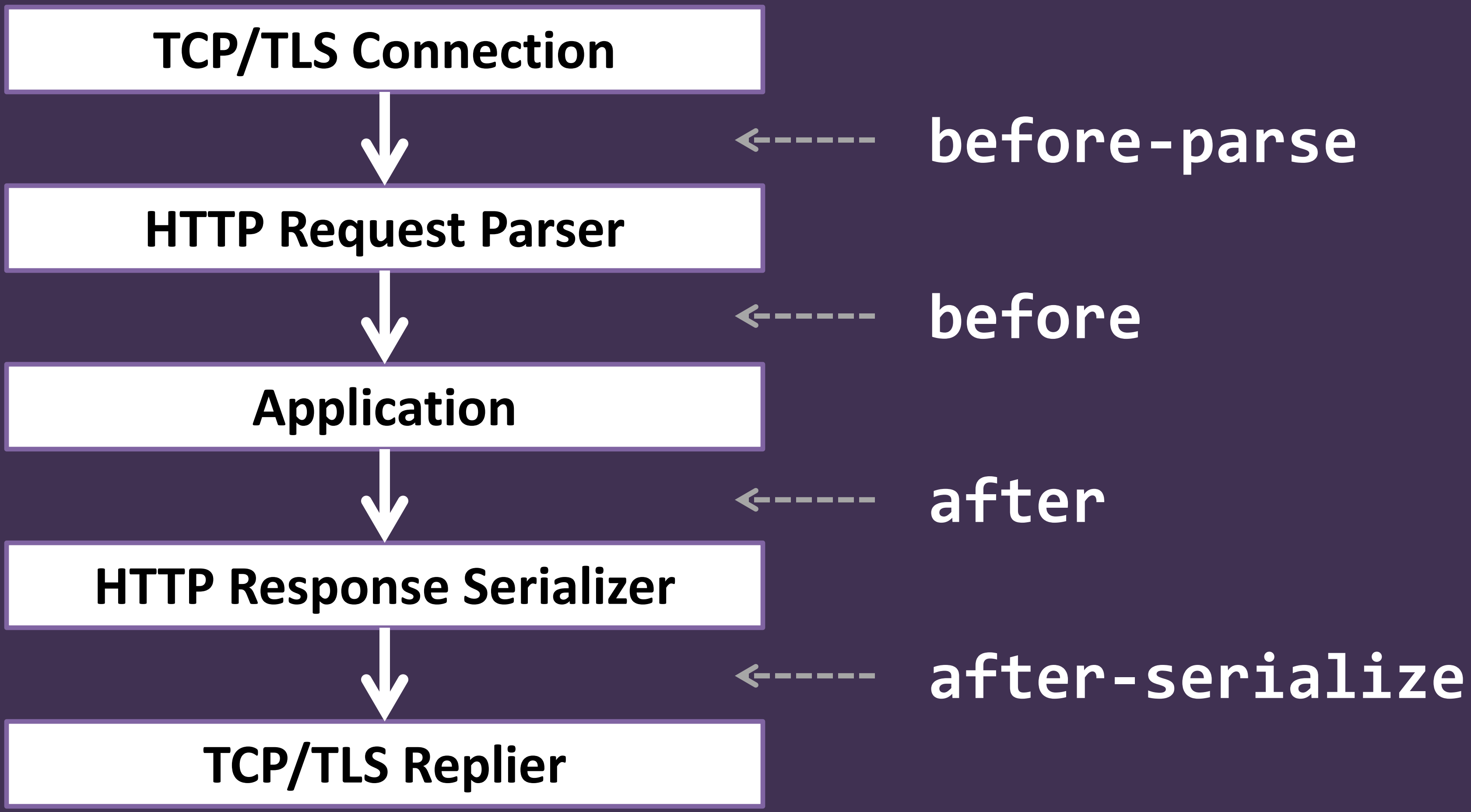
```
openapi 'api.json'.IO, {
    operation 'is-prime', -> Int $number {
        content 'application/json',
            { :$number, :prime($number.is-prime) }
    }

    operation 'get-prime', -> {
        request-body -> (:$min, :$max) {
            with ($min..$max).grep(*.is-prime).pick -> $number {
                content 'application/json', { :$number }
            }
            else {
                not-found;
            }
        }
    }
}
```

It's now much easier to create and consume middleware



Requests are pushed through an asynchronous pipeline

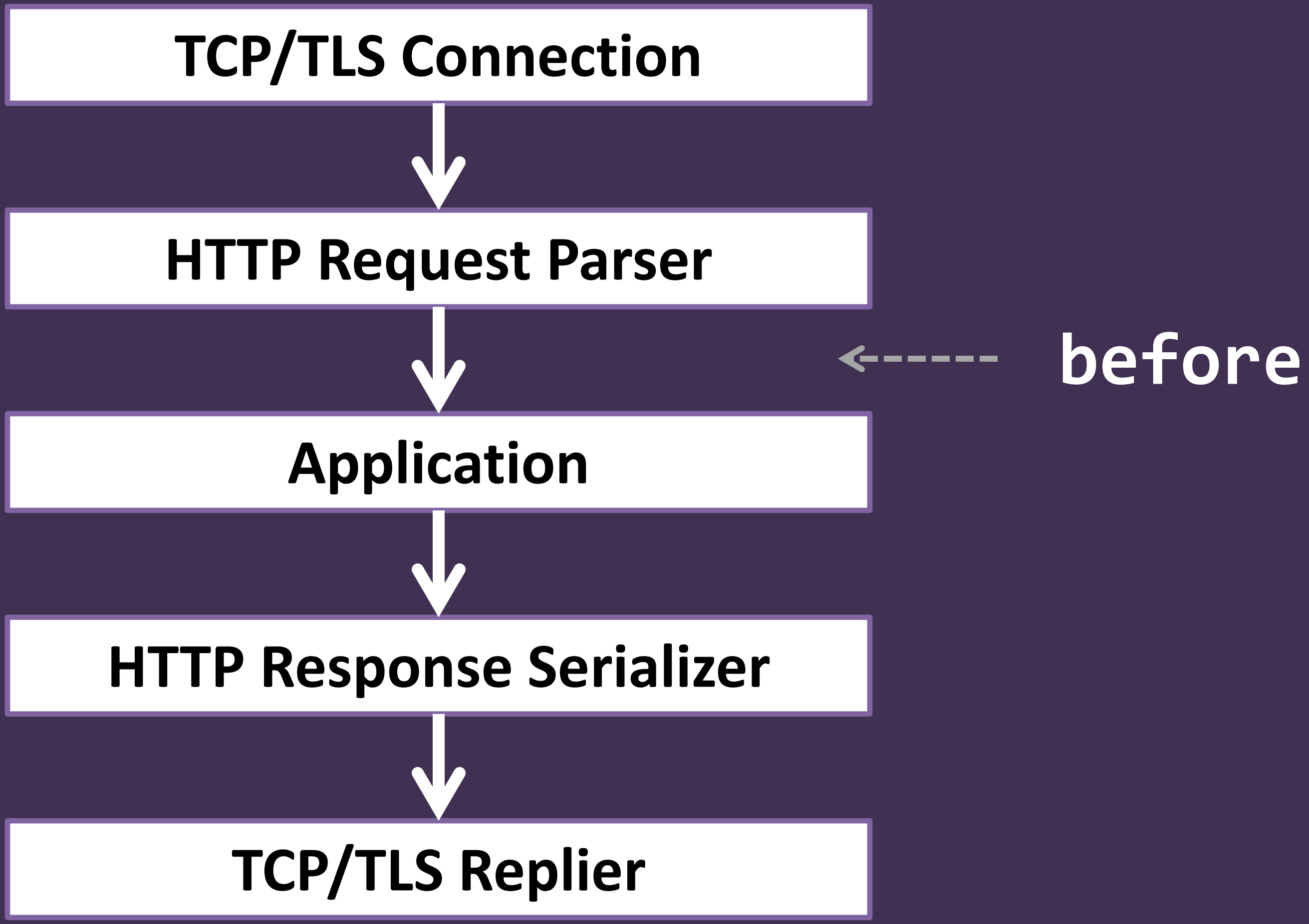
Server-level middleware



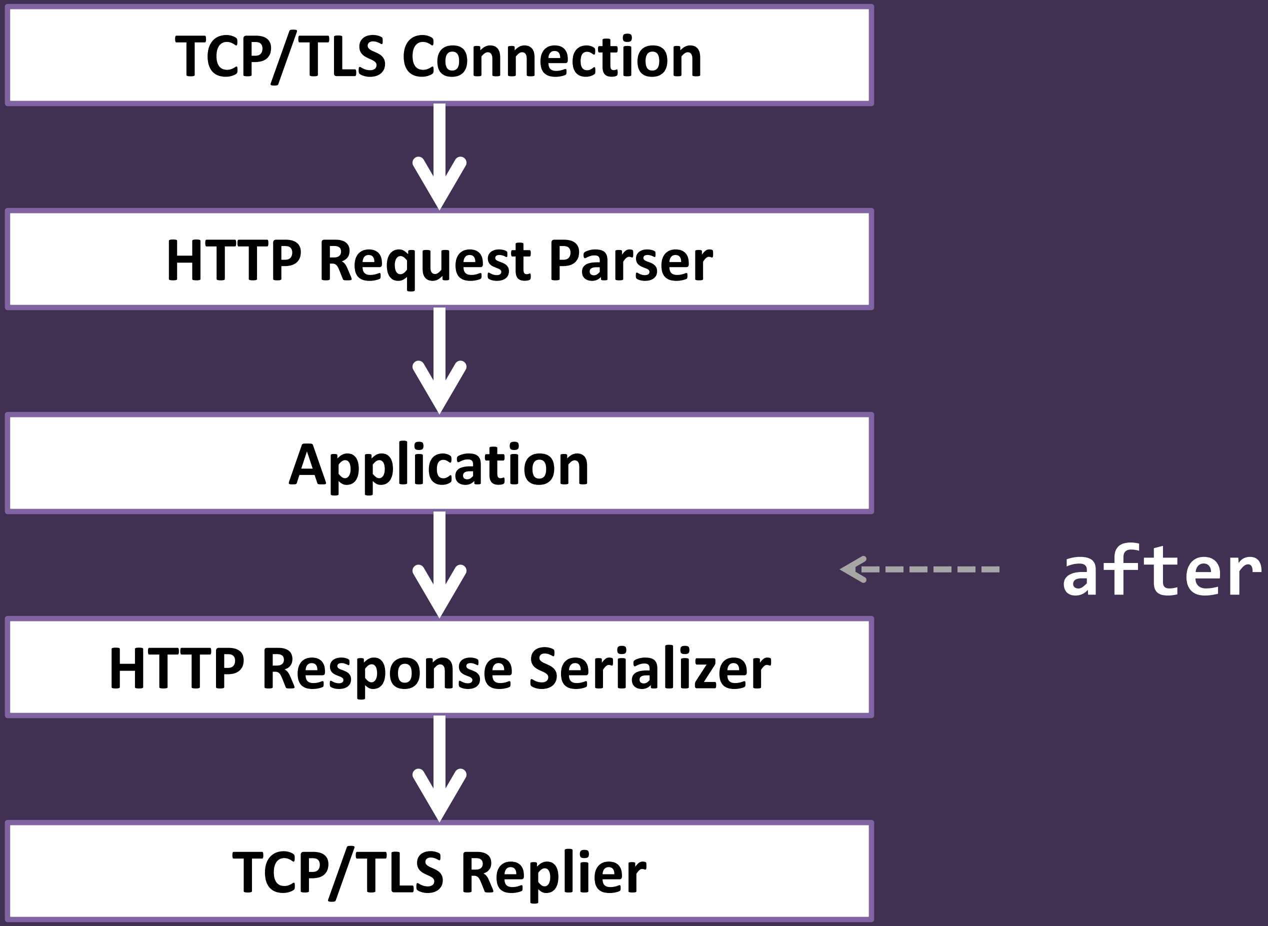
Could always just write transforms before now

However, new roles get rid of the boilerplate

Cro::HTTP::Middleware::Request

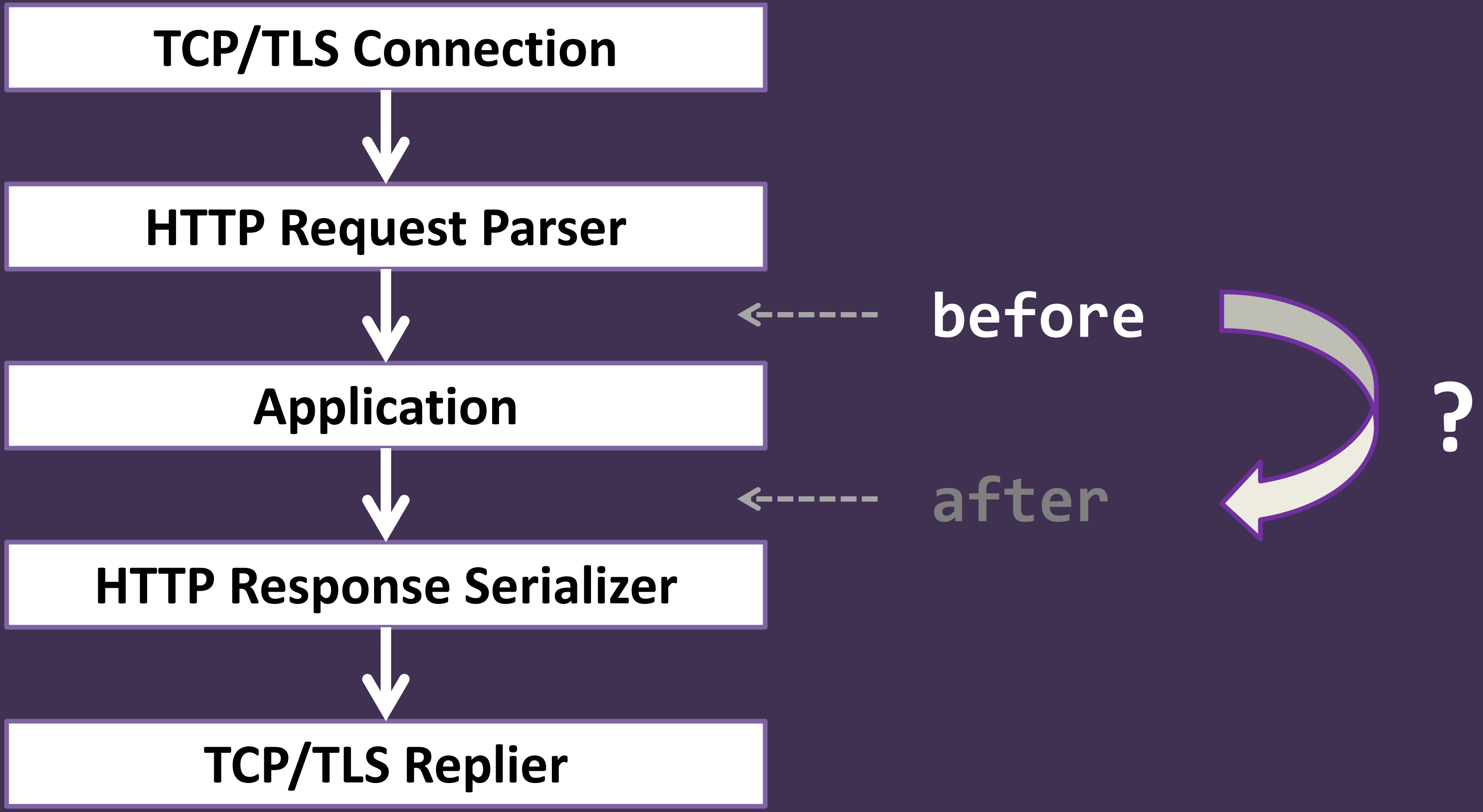


Cro::HTTP::Middleware::Response



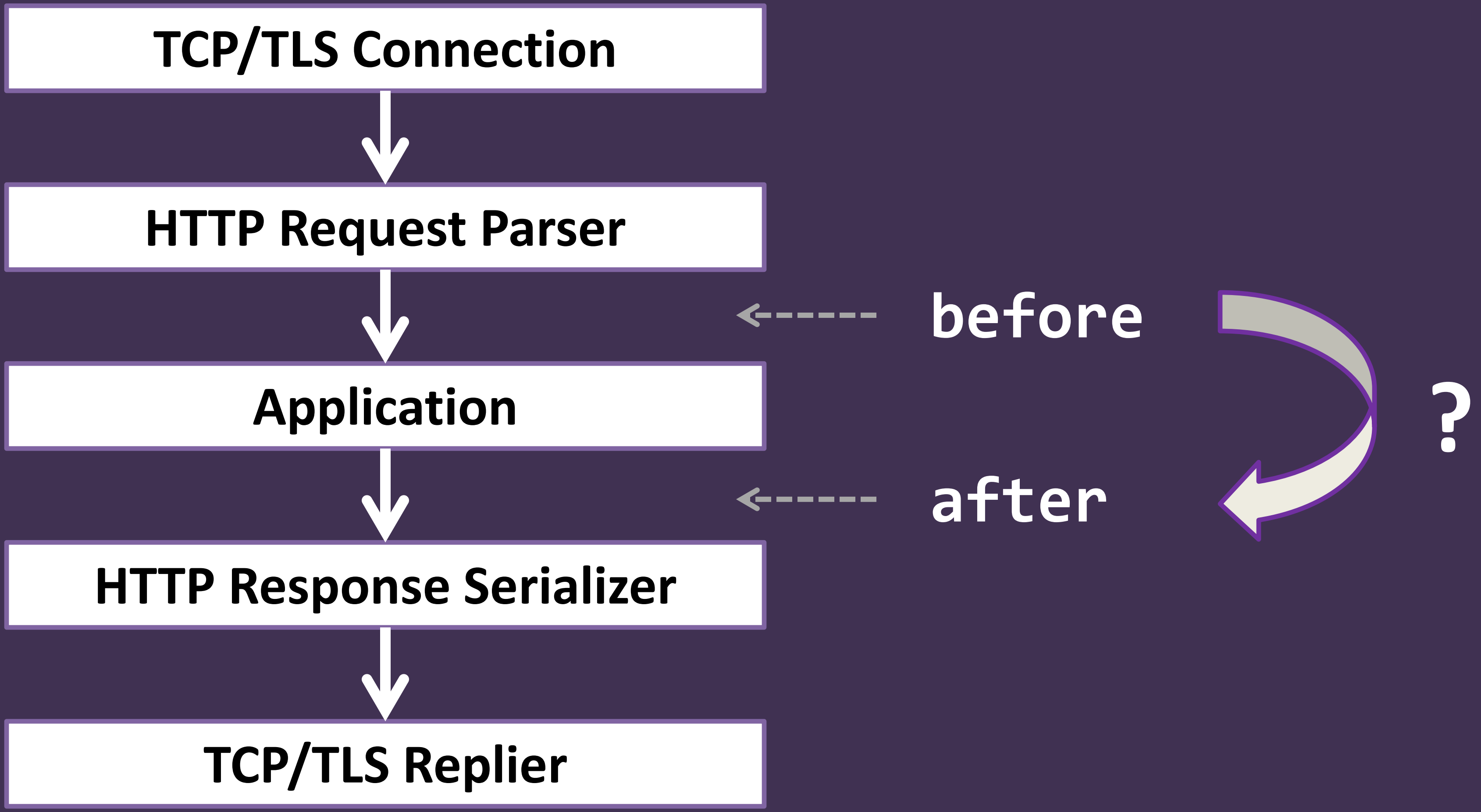
```
class HSTS does Cro::HTTP::Middleware::Response {
    has Int $.max-age = 31536000;

    method process(Supply $responses) {
        supply whenever $responses -> $rep {
            $rep.append-header: 'Strict-transport-security',
                "max-age=$!max-age"
            emit $rep;
        }
    }
}
```

Cro::HTTP::Middleware::Conditional



```
class LocalOnly does Cro::HTTP::Middleware::Conditional {
    method process(Supply $requests) {
        supply whenever $requests -> $request {
            if $request.connection.peer-host eq '127.0.0.1' | '::1' {
                # It's local, so continue processing.
                emit $req;
            }
            else {
                # It's not, so emit a 403 forbidden response.
                emit Cro::HTTP::Response.new(:$request, :403status);
            }
        }
    }
```

Cro::HTTP::Middleware::RequestResponse



Middleware can also be applied in `route` blocks

Apply middleware written as a class

```
my $app = route {
    before LocalOnly.new;
    after HSTS.new;

    ...
}
```

Write simple middleware inline

```
my $app = route {
    before {
        forbidden unless .connection.peer-host eq '127.0.0.1' | '::1';
    }
    after {
        header 'Strict-transport-security',
            'max-age=31536000; includeSubDomains'
    }

    ...
}
```

before / after

Composed around the `route` block

Run unconditionally

before-matched / after-matched

Wrap around a handler

Run if a route was matched

(These are the Cro 0.8.0 semantics. Prior to that, before/after were used to mean what before-matched and after-matched now mean, and there was no direct equivalent to the new before/after semantics.)

Session handling and auth are implemented as middleware

Declare a session/user object

```
class My::App::Session does Cro::HTTP::Auth {
    has $.is-logged-in;
    has $.is-admin;
    has @.recently-viewed-items;
}
```

If needed, declare Perl 6 subset types to distinguish types of user

```
subset Admin of My::App::Session where .is-admin;
subset LoggedIn of My::App::Session where .is-logged-in;
```

Match on them in routes

```
my $app = route {
    get -> LoggedIn $user, 'my', 'profile' {
        # Use $user in some way
    }

    get -> Admin, 'system', 'log' {
        # Just use the type and don't name a variable, if
        # the session/user object is not needed
    }
}
```

Apply session middleware

```
my $app = route {
    before Cro::HTTP::Session::InMemory[My::App::Session].new(
        expiration => Duration.new(60 * 15),
        cookie-name => 'MY_SESSION_COOKIE_NAME'
    );

    ...
}
```

Middleware included for:

Persistent sessions
Basic authentication
JSON Web Tokens

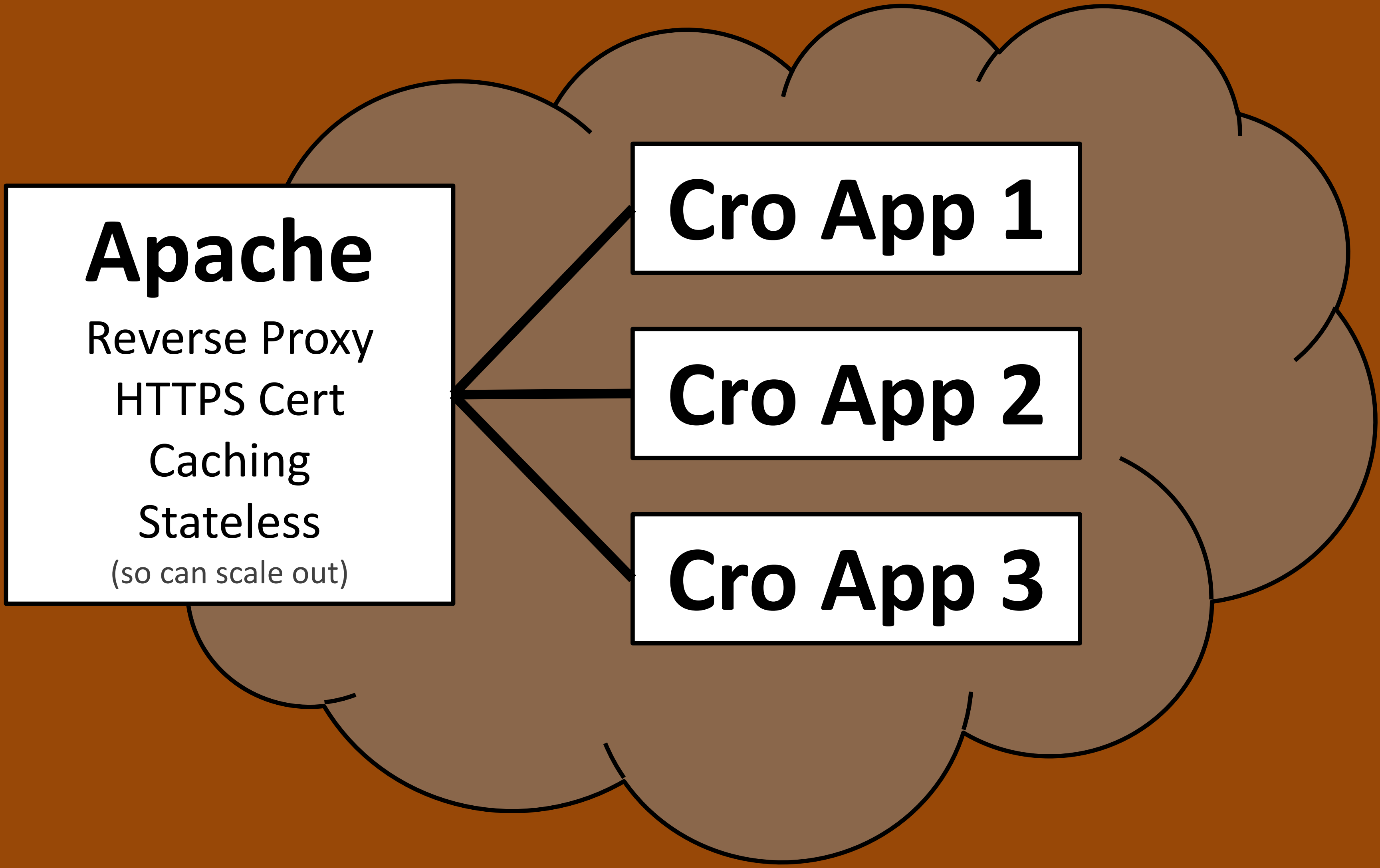
Web-based login/logout is left for the application to handle

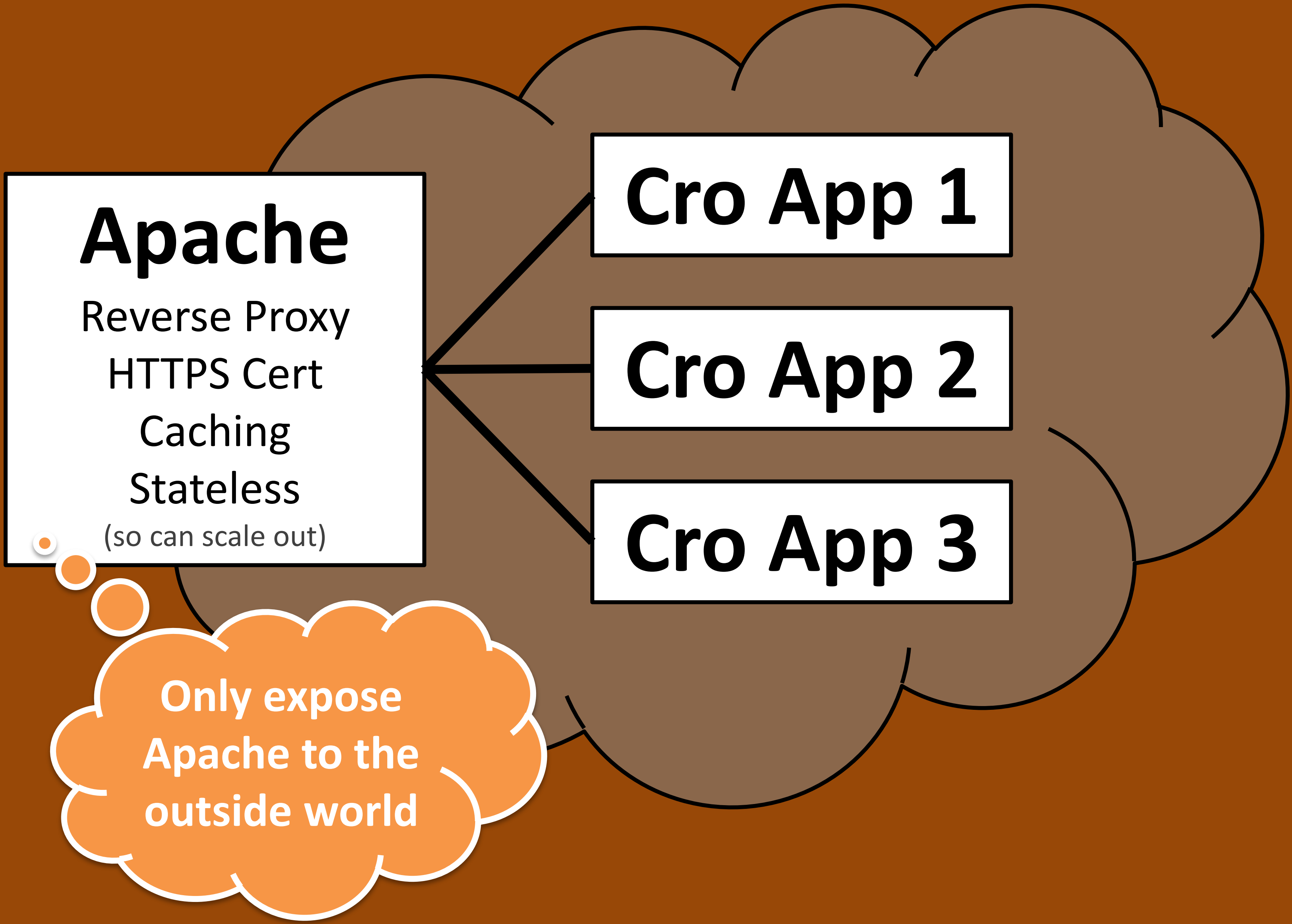
Further assistance planned in the future Cro::HTTP::WebApp

Stubbed projects now include a `Dockerfile`

And we provide several Cro base images, to give you quicker container builds

All of our Cro applications at Edument are deployed in containers running on a Kubernetes cluster


Apache
Reverse Proxy
HTTPS Cert
Caching
Stateless
(so can scale out)
Cro App 1
Cro App 2
Cro App 3


Apache
Reverse Proxy
HTTPS Cert
Caching
Stateless
(so can scale out)
Cro App 1
Cro App 2
Cro App 3
Only expose Apache to the outside world

Place all static assets in a route block and apply middleware to add a cache control header

```
sub assets() {
    route {
        after-matched {
            cache-control :public, :max-age(180);
        }
        get -> 'css', *@path {
            static 'static-content/css', @path
        }
        get -> 'js', *@path {
            static 'static-content/js', @path
        }
    }
}
```

And where are we going

in the next year?

`Cro::HTTP::WebApp`

If folks are going to use Cro as a web framework, we should serve them better

So far, we've a 6-y template engine in development (and, uh, production...)

Planning some login/logout plumbing, CSRF protection, and so forth

Reverse Proxy Support

Initial implementation coming in Cro 0.8.0

Makes the easy things easy:

```
# /user/foo proxied to http://user-service/foo
delegate <user *> => Cro::HTTP::ReverseProxy.new:
    to => 'http://user-service/';
```

Plus many features to make the harder things (request/response processing) possible

Reliability patterns

Timeouts

Retries

Throttling

Circuit breaker

Portability

MacOS

Windows

Wherever our userbase leads us

When folks miss something, or stub their toe on something repeatedly, we look for ways to make things better

Questions?

Web: http://cro.services/
Twitter: @croservices
IRC: #cro on freenode.org