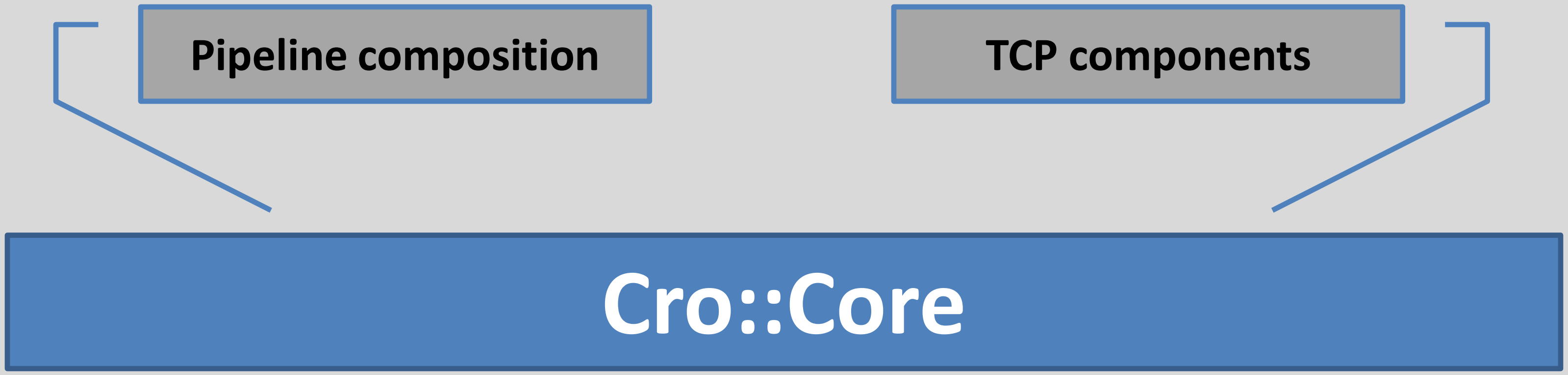
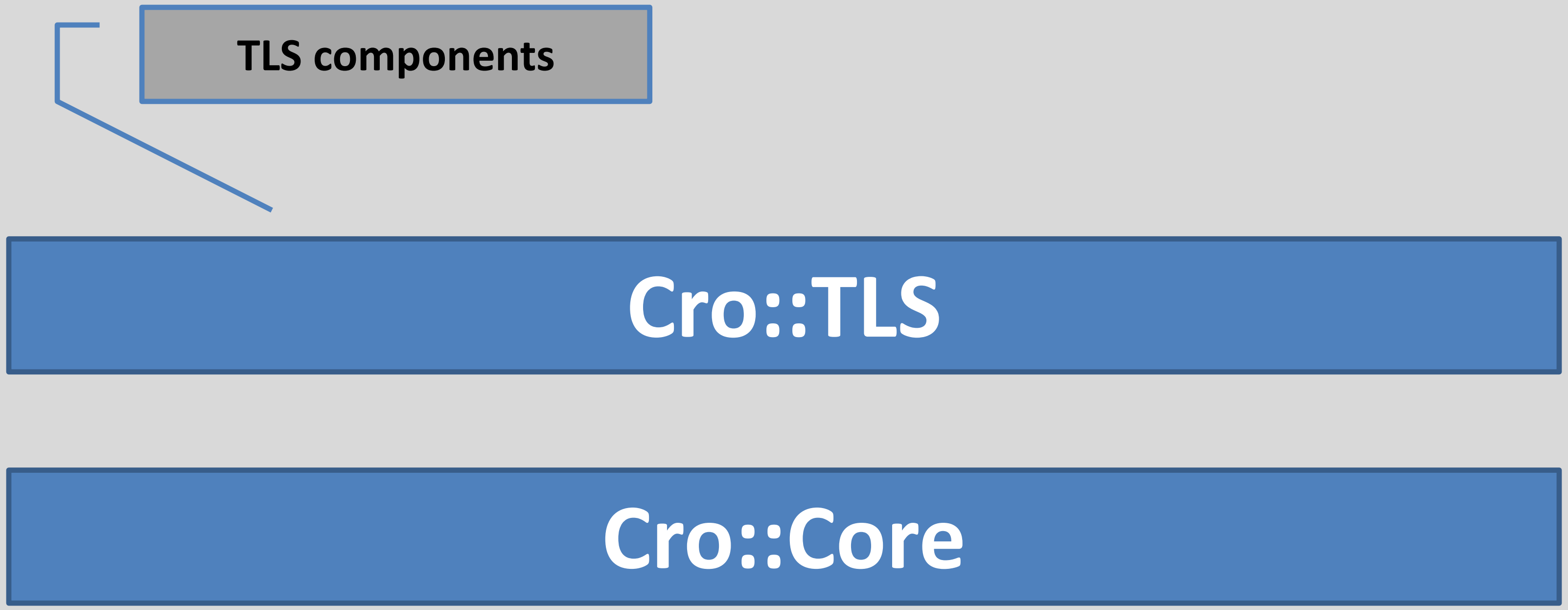
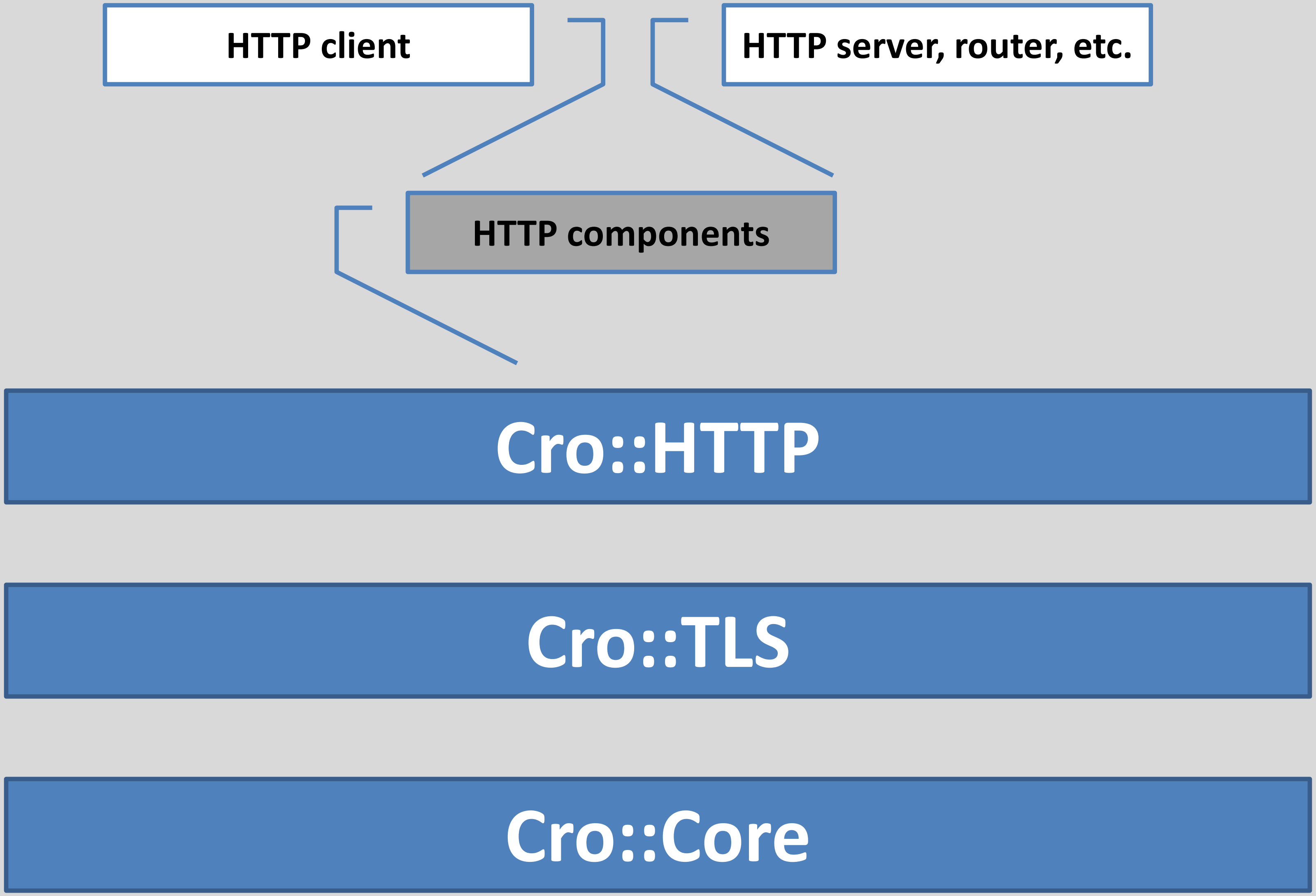
Cro Router Essentials

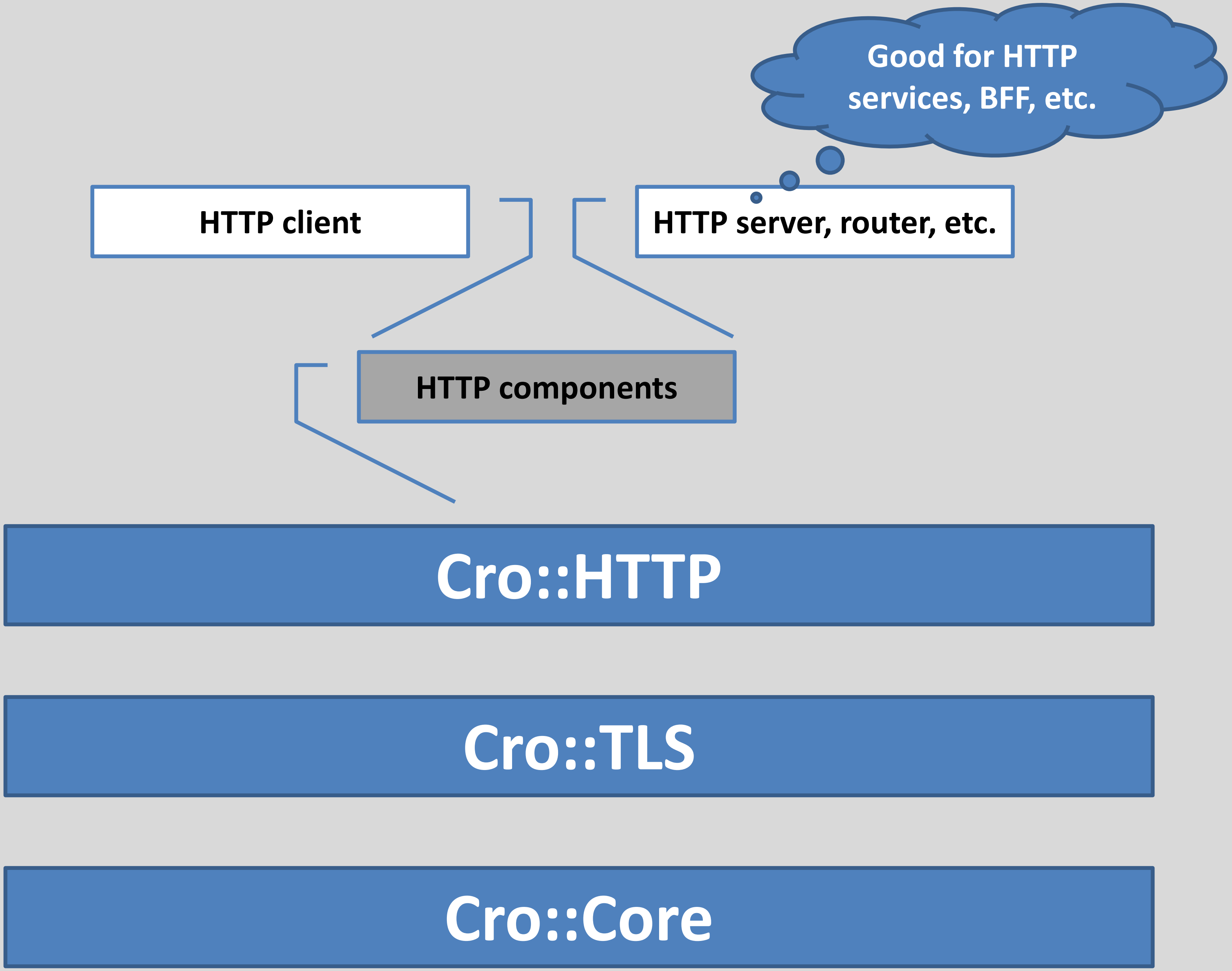
Jonathan Worthington | Edument

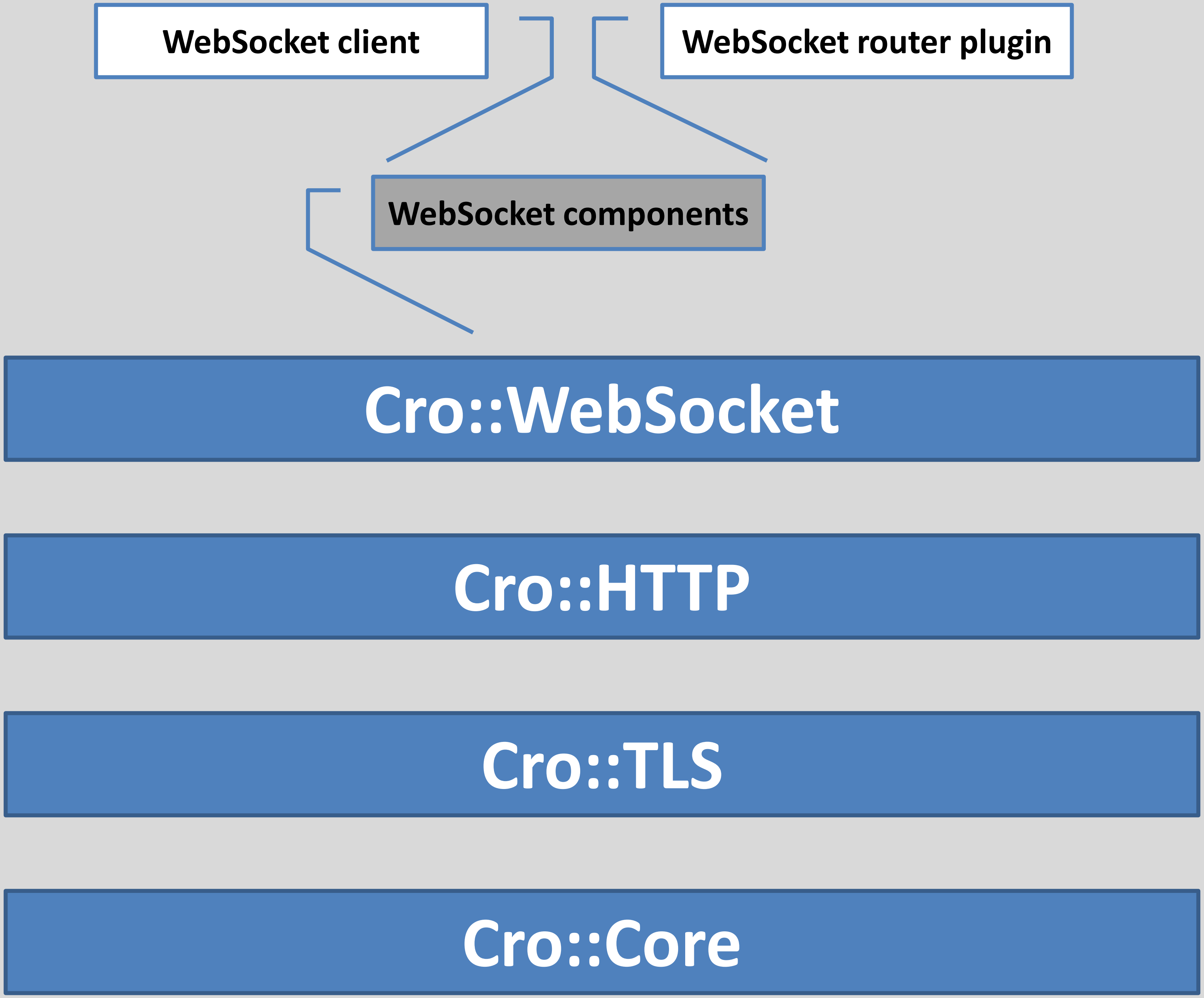
So what is Cro anyway?

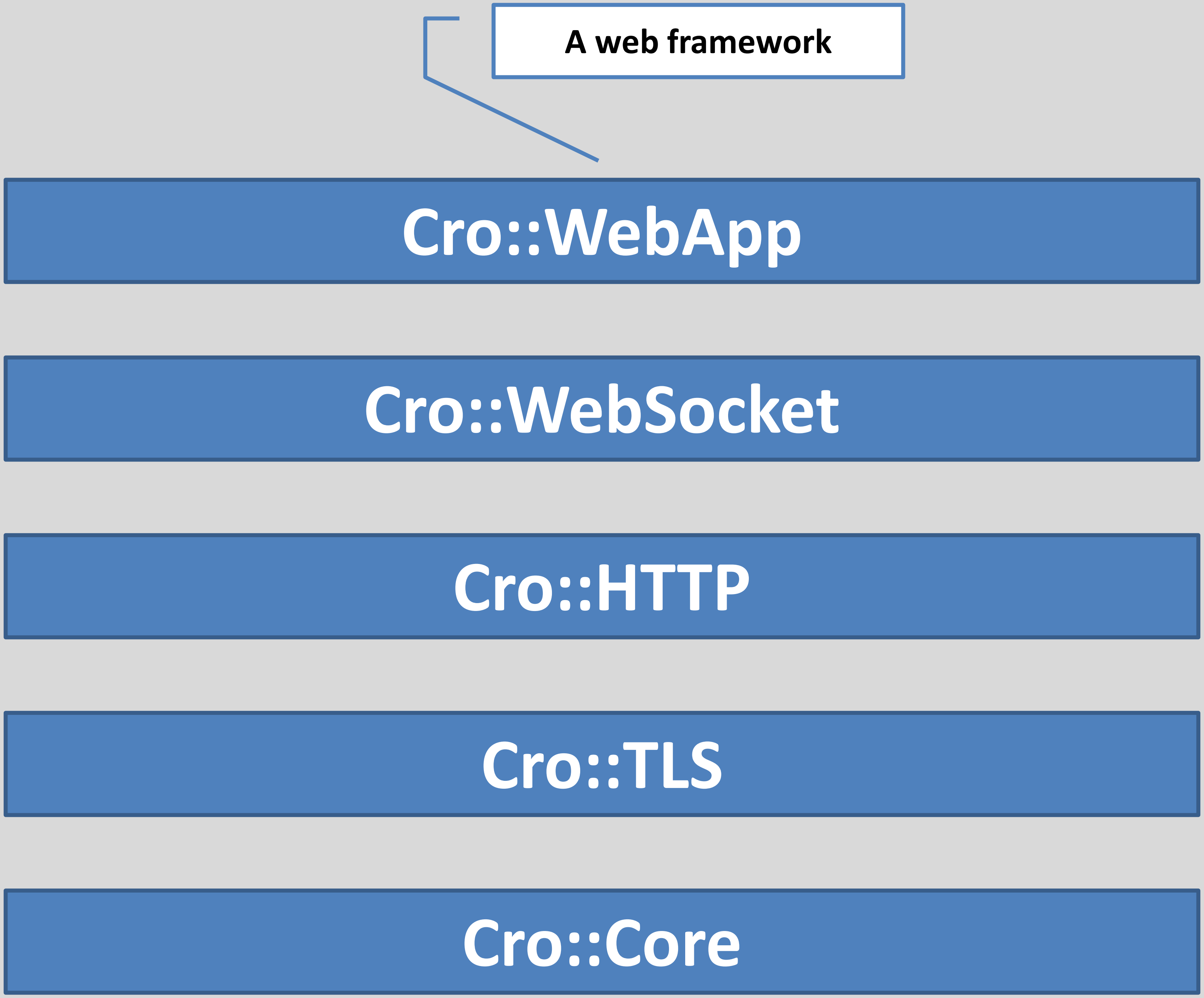

Pipeline composition
TCP components
Cro::Core


TLS components
Cro::TLS
Cro::Core


HTTP client
HTTP server, router, etc.
HTTP components
Cro::HTTP
Cro::TLS
Cro::Core


Good for HTTP services, BFF, etc.
HTTP client
HTTP server, router, etc.
HTTP components
Cro::HTTP
Cro::TLS
Cro::Core


WebSocket client
WebSocket router plugin
WebSocket components
Cro::WebSocket
Cro::HTTP
Cro::TLS
Cro::Core


A web framework
Cro::WebApp
Cro::WebSocket
Cro::HTTP
Cro::TLS
Cro::Core

A web framework

Finally!

Cro::WebApp

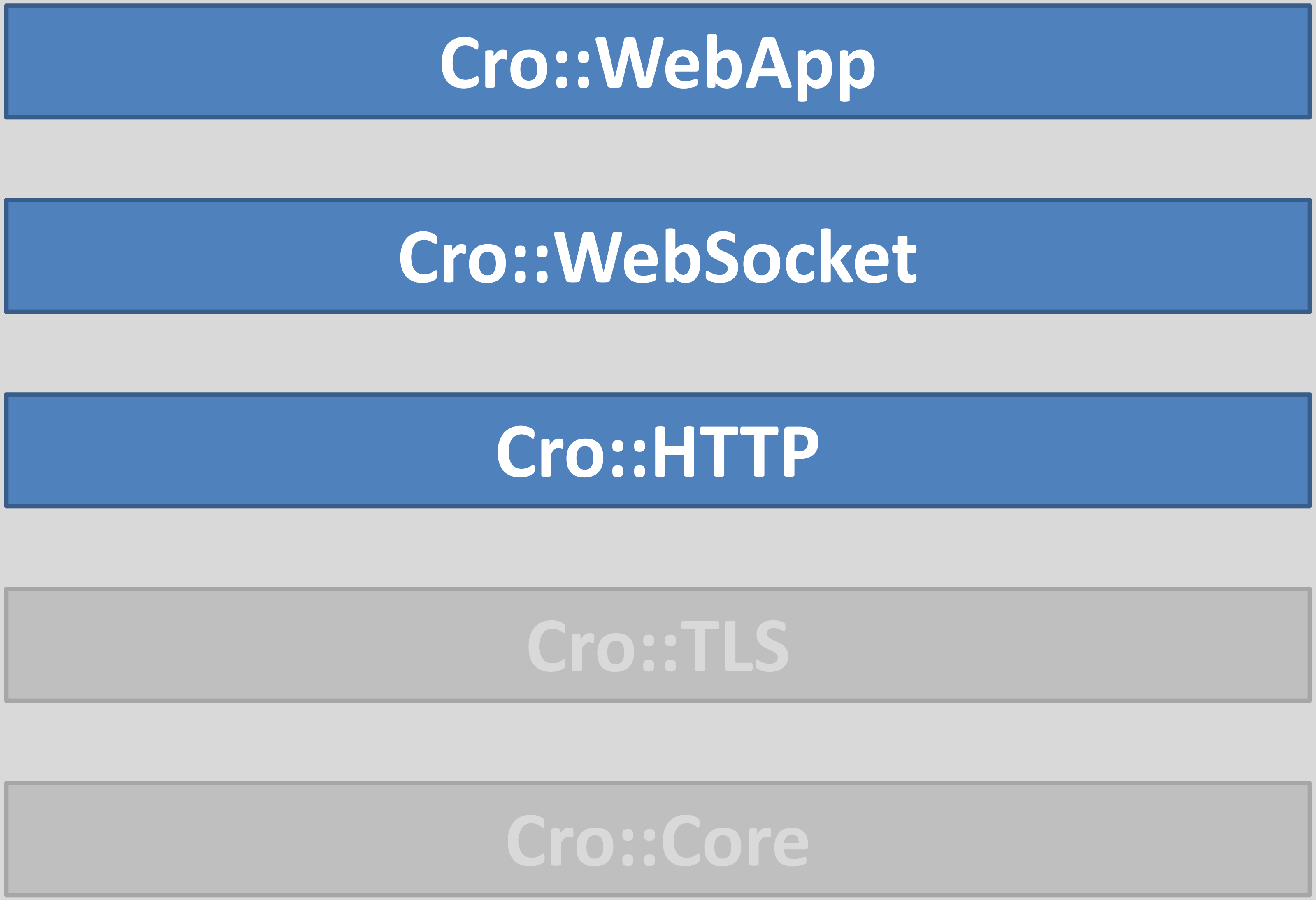
Cro::WebSocket

Cro::HTTP

Cro::TLS

Cro::Core

Today we'll "just" be considering...



What is the router?

Maps incoming HTTP requests to logic that handles them

Not where business / domain logic lives!
Only the mapping of that into HTTP.

Lots of things "plug in" to the Cro HTTP router

WebSocket support
Templates and form handling
OpenAPI

Setup

Either write code like this...

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

my $application = route {
    get -> {
        content 'text/plain', "Hello world!\n";
    }
}

my Cro::Service $http = Cro::HTTP::Server.new:
        :port(20000), :$application;
$http.start;
react whenever signal(SIGINT) {
    $http.stop;
    exit;
}
```

...or use the `cro` CLI...

```
$ cro stub http example example/
```

A name for the service

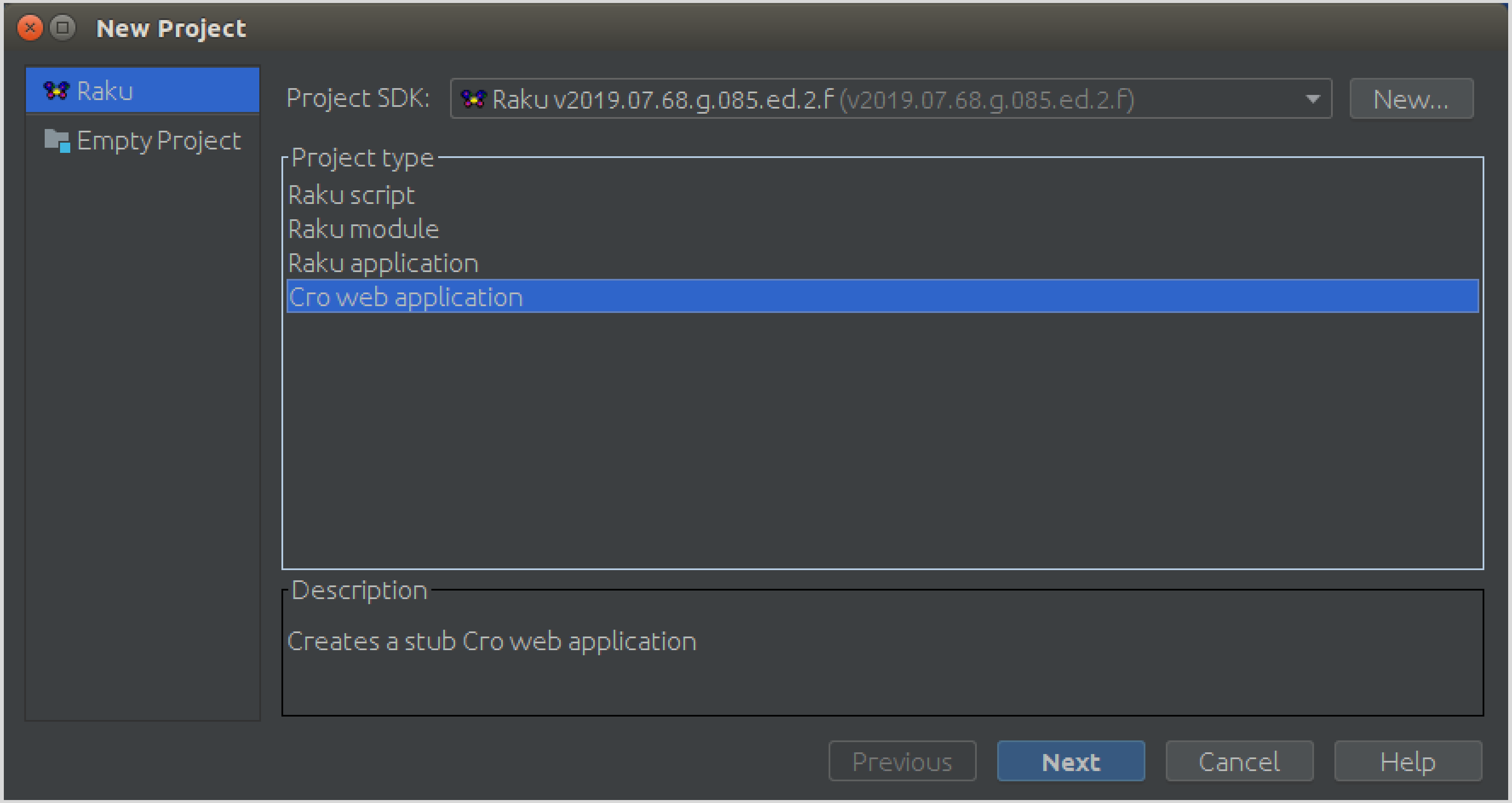
Where to put it

Answer its questions

Get a stub service created

Routes are in `lib/Routes.pm6`

...or use Comma IDE


New Project

Raku

Empty Project

Project SDK: Raku v2019.07.68.g.085.ed.2.f (v2019.07.68.g.085.ed.2.f)

Project type
- Raku script
- Raku module
- Raku application
- Cro web application

Description

Creates a stub Cro web application

Route segment matching

Routing uses Raku signatures

```
route {
    # No parameters: GET /
    get -> {
        content 'text/plain', 'Welcome!';
    }

    # One literal parameter: GET /about
    get -> 'about' {
        content 'text/plain', 'About Us';
    }

    # Two literal parameters: GET /services/development
    get -> 'services', 'development' {
        content 'text/plain', 'We write code!';
    }
}
```

Parameters receive segment values

```
route {
    # Matches anything in second segment:
    # GET /product/42
    get -> 'product', $id {
        content 'text/plain', "About product ID: $id";
    }

    # However, literal segments win:
    # GET /product/search
    get -> 'product', 'search' {
        content 'text/plain', 'Search products';
    }
}
```

Types

```
route {
    # Force it to be, and parse it as, an Int
    get -> 'product', Int $id {
        content 'text/plain', "About product ID: $id";
    }

    # Or an unsigned Int:
    get -> 'product', UInt $id {
        content 'text/plain', "About product ID: $id";
    }

    # Or limit it to 32 bits:
    get -> 'product', uint32 $id {
        content 'text/plain', "About product ID: $id";
    }
}
```

Constraints

```
route {
    # Declare a subset type matching UUIDs
    my subset UUIDv4 of Str where /^
            <[0..9a..f]> ** 12
            4 <[0..9a..f]> ** 3
            <[89ab]> <[0..9a..f]> ** 15
            $/;

    # Segment must contain a UUIDv4:
    # GET /product/afb47801aa9c454db7037cd3502ada4c
    get -> 'product', UUIDv4 $id {
        content 'text/plain', "About product ID: $id";
    }
}
```

Optional and slurpy

```
route {
    # Get all the menu or just one section
    # GET /menu
    # GET /menu/appetizers
    get -> 'menu', $section? {
        content 'text/plain', $section
                ?? "Just the $section"
                !! "All the food!";
    }

    # Get a page in the documentation
    # GET /docs
    # GET /docs/cro/http/router
    get -> 'docs', *@path {
        content 'text/plain',
                "You want the doc at @path.join('/')";
    }
}
```

Serving static content

Single files

```
route {
    # Respond to requests for the favicon with a
    # particular file (media type `image/x-icon`
    # chosen automatically from extension)
    # GET /favicon.ico
    get -> 'favicon.ico' {
        static 'static-content/favicon.ico'
    }
}
```

Everything below a subdirectory

```
route {
    # Any file within a directory
    # GET /js/app.js
    get -> 'js', $file {
        static 'compiled-frontend/', $file
    }

    # Or any path below a directory (includes
    # protection against traversal attacks)
    # GET /images/food/jalfrezi.jpg
    get -> 'images', *@path {
        static 'static-content/images', @path
    }
}
```

Further static serving options

```
route {
    # Specify index files for directories
    # GET /pages         (serves pages/index.html)
    # GET /pages/foo/    (serves pages/foo/index.html)
    get -> 'pages', *@path {
        static 'pages', @path, indexes => <index.html>;
    }

    # Customize the mime type for `.foo` files
    get -> 'downloads', *@path {
        static 'files', @path, mime-types => {
            'foo' => 'application/vnd.acme.foo'
        }
    }
}
```

Producing responses

What is content really doing?

```
multi content(Str $content-type, $body,
        :$enc = $body ~~ Str ?? 'utf-8' !! Nil --> Nil) {
    # Defaulting the status code to 200
    response.status //= 200;

    # Setting the content type and maybe charset
    with $enc {
        response.append-header('Content-type',
                "$content-type; charset=$_");
    }
    else {
        response.append-header('Content-type', $content-type);
    }

    # Setting the response body
    response.set-body($body);
}
```

Forming the response body

- Router action sets the `response.body` (normally via. a function, such as `content`)
- Response serializer asks the response's `BodySerializerSelector` for a serializer
- The response body is serialized using the chosen `BodySerializer`

Default set of body serializers

JSON

(Body is Array/Hash, JSON content-type)

Binary

(Body is a Blob or Buf)

Text

(Body is a Str)

Supply

(Body is a Supply)

Producing a JSON response

```
route {
    # Just specify the content type, and the JSON body
    # serializer will be selected, and turn the data
    # structure into JSON.
    get -> 'product', Int $id {
        content 'application/json', {
            :$id, :name('Kashmiri chili powder'),
            :description('Beautifully red and spicy!'),
            :tags<cooking indian chili>
        }
    }
}
```

(Also chosen for any media type with `+json` suffix)

Producing a streaming response

```
route {
    # Provide a Supply body. It must emit binary Blobs.
    # It will be tapped by the response serializer, and
    # the data sent using the chunked transfer encoding.
    get -> 'ticks' {
        content 'text/plain', supply {
            whenever Supply.interval(1) {
                emit "$_\n".encode('utf-8');
            }
        }
    }
}
```

Writing a custom body serializer

```
use Cro::HTTP::BodySerializers;
use YAMLish;

class YAMLBodySerializer does Cro::HTTP::BodySerializer {
    # Should it serialize this response?
    method is-applicable($response, $body --> Bool) {
        $response.content-type.type-and-subtype eq 'text/yaml'
    }

    # If so, how? Should return a Supply.
    method serialize($response, $body --> Supply) {
        my $yaml = save-yaml $body;
        my $binary = $yaml.encode('utf-8');
        self!set-content-length($response, $binary.bytes);
        supply emit $binary
    }
}
```

Using a custom body serializer

```
route {
    # Tell our route block to add it to the set of
    # possible serializers for all responses that it
    # produces
    body-serializer YAMLBodySerializer;

    # Produce the content type that it's looking for
    get -> 'product', Int $id {
        content 'text/yaml', {
            :$id, :name('Kashmiri chili powder'),
            :description('Beautifully red and spicy!'),
            :tags<cooking indian chili>
        }
    }
}
```

Redirects

```
route {
    # Temporary redirect while shop unavailable
    get -> 'shop', *@ {
        redirect '/news';
    }

    # Permanent redirect to new blog location
    get -> 'blog', Int $post-id {
        redirect "https://blog.our.domain/$post-id",
                :permanent;
    }
}
```

Error responses

```
route {
    # Various helper functions exist for producing
    # common error responses
    get -> 'advent', Int $day {
        if $day <= Date.today.day {
            content 'text/plain',
                    'A post of the day for you';
        }
        else {
            # Content type and content are optional
            not-found 'text/plain',
                    'Naughty naughty! Not yet!';
        }
    }
}
```

Error responses

There are also functions for...

`bad-request`
`forbidden`
`conflict`

For others, use `set-status`, optionally followed by `content`

Templates

`Cro::WebApp::Template`

Part of the `Cro::WebApp` distribution
(so if you're just building services, you don't need to ship it)

Conditionals, interpolation, subroutines, modules, automatic escaping of data, etc.

Supported in Comma IDE
(syntax highlighting, auto-complete, navigation)

An example template

```
<h2>Latest News</h2>
<@stories>
  <&story($_)>
</@>

<:sub story($s)>
  <h3><$s.headline></h3>
  <p class="date">Posted <$s.posted> by <$s.author></p>
  <?$s.exclusive>
    <div class="exclusive">EXCLUSIVE!</div>
  </?>
  <p><$s.summary></p>
</:>
```

An example template

```
<h2>Latest News</h2>
<@stories>
  <&story($_)>
</@>

<:sub story($s)>
  <h3><$s.headline></h3>
  <p class="date">Posted <$s.posted> by <$s.author></p>
  <?$s.exclusive>
    <div class="exclusive">EXCLUSIVE!</div>
  </?>
  <p><$s.summary></p>
</:>
```

@ sigil tag is an iteration

An example template

```
<h2>Latest News</h2>
<@stories>
  <&story($_)>
</@>

<:sub story($s)>
  <h3><$s.headline></h3>
  <p class="date">Posted <$s.posted> by <$s.author></p>
  <?$s.exclusive>
    <div class="exclusive">EXCLUSIVE!</div>
  </?>
  <p><$s.summary></p>
</:>
```

& sigil tag calls a subroutine

An example template

```
<h2>Latest News</h2>
<@stories>
  <&story($_)>
</@>

<:sub story($s)>
  <h3><$s.headline></h3>
  <p class="date">Posted <$s.posted> by <$s.author></p>
  <?$s.exclusive>
    <div class="exclusive">EXCLUSIVE!</div>
  </?>
  <p><$s.summary></p>
</:>
```

: sigil tag is used for making declarations

An example template

```
<h2>Latest News</h2>
<@stories>
  <&story($_)>
</@>

<:sub story($s)>
  <h3><$s.headline></h3>
  <p class="date">Posted <$s.posted> by <$s.author></p>
  <?$s.exclusive>
    <div class="exclusive">EXCLUSIVE!</div>
  </?>
  <p><$s.summary></p>
</:>
```

$ sigil tag is interpolation...

An example template

```
<h2>Latest News</h2>
<@stories>
  <&story($_)>
</@>

<:sub story($s)>
  <h3><$s.headline></h3>
  <p class="date">Posted <$s.posted> by <$s.author></p>
  <?$s.exclusive>
    <div class="exclusive">EXCLUSIVE!</div>
  </?>
  <p><$s.summary></p>
</:>
```

…and we can index into the data too

An example template

```
<h2>Latest News</h2>
<@stories>
  <&story($_)>
</@>

<:sub story($s)>
  <h3><$s.headline></h3>
  <p class="date">Posted <$s.posted> by <$s.author></p>
  <?$s.exclusive>
    <div class="exclusive">EXCLUSIVE!</div>
  </?>
  <p><$s.summary></p>
</:>
```

? sigil tag is "if"
! sigil tag is "unless"

Using the template

```
route {
    # Specify directory holding templates
    template-location 'templates';

    # Render a template as the response body
    get -> 'news' {
        template 'summary.crotmp', { stories => [
            {
                :headline('Something happened'),
                :author('Jonathan'),
                :posted(Date.today.yyyy-mm-dd),
                :exclusive,
                :summary('It was amazing!'),
            },
            # ...
        ] };
    }
}
```

A page layout macro

```
<:macro layout($title)>
  <html lang="en">
  <head>
    <meta charset="UTF-8">
    <title><$title></title>
  </head>
  <body>
    <:body>
  </body>
 </html>
</:>
```

Use this to render the inner content

Applying the layout

```
<:use 'layout.crotmp'>

<|layout('Latest news')>
  <h2>Latest News</h2>
  <@stories>
    <&story($_)>
  </@>
</|>

<:sub story($s)>
  <h3><$s.headline></h3>
  <p class="date">Posted <$s.posted> by <$s.author></p>
  <?$s.exclusive>
    <div class="exclusive">EXCLUSIVE!</div>
  </?>
  <p><$s.summary></p>
</:>
```

Use the template declaring the layout

Applying the layout

```
<:use 'layout.crotmp'>

<|layout('Latest news')>
  <h2>Latest News</h2>
  <@stories>
    <&story($_)>
  </@>
</|>

<:sub story($s)>
  <h3><$s.headline></h3>
  <p class="date">Posted <$s.posted> by <$s.author></p>
  <?$s.exclusive>
    <div class="exclusive">EXCLUSIVE!</div>
  </?>
  <p><$s.summary></p>
</:>
```

Apply the layout macro to the content

Query strings, cookies, and headers

Query string values

```
route {
    # Take query string parameters using named parameters.
    # Remember that these are optional by default!
    # GET /search?color=blue
    # GET /search?color=blue&max-price=100
    get -> 'search', :$color, :$min-price, :$max-price {
        content 'text/plain', qq:to/CONTENT/
            Color: {$color // 'any'}
            Price: {$min-price // 0 } to {$max-price // '*'}
            CONTENT
    }
}
```

(These can also be typed and constrained.)

Headers and cookies

```
route {
    # Use the `is header` trait for getting headers
    get -> 'browser', :$user-agent is header {
        content 'text/plain', $user-agent
                ?? "You appear to be using $user-agent"
                !! "Your client is rather shy";
    }

    # And the `is cookie` trait for getting cookies
    get -> 'last-visit', :$last-visit is cookie {
        set-cookie 'last-visit', Date.today.yyyy-mm-dd;
        content 'text/plain', $last-visit
                ?? "You last visited on $last-visit"
                !! "You did not visit before";
    }
}
```

Request bodies

Different ways to get the body

Do The Right Thing

```
request-body -> $body { ... }
```

Text

```
request-body-text -> $text { ... }
```

Binary

```
request-body-blob -> $binary { ... }
```

Supply of bytes as they arrive over network

```
request.body-byte-stream
```

Different ways to get the body

Do The Right Thing

```
request-body -> $body { ... }
```

Te

```
request-body-
```

`BodyParserSelector` and `BodyParser` objects

Binary

```
request-body-blob -> $binary { ... }
```

Supply of bytes as they arrive over network

```
request.body-byte-stream
```

Default set of body parsers

WWWFormUrlEncoded

application/x-www-form-urlencoded content type

MultiPartFormData

multipart/form-data content type

JSON

application/json or *+json content type

TextFallback

text/* content type

BinaryFallback

If all else fails

Getting a JSON body

```
route {
    # The block is invoked with the JSON object as soon as it
    # has arrived over the network and been decoded (using
    # await so we don't block an OS thread)
    # PUT /reviews → 204 No Content response
    put -> 'reviews' {
        request-body -> %json {
            say "Should save %json.raku()";
        }
    }
}
```

Signature-based unpacking/validation

```
route {
    # Ratings should be between 1 and 5
    subset Rating of Int where 1..5;

    # Use signature to unpack the JSON, checking it along
    # the way. If there's no match, automatic 400 Bad Request
    # response.
    # PUT /reviews → 204 No Content | 400 Bad Request
    put -> 'reviews' {
        request-body -> (Rating :$rating!, Str :$comment) {
            say "$rating / 5 ($comment)";
        }
    }
}
```

(But consider using OpenAPI for more complex or public APIs.)

A custom YAML body parser

```
use Cro::HTTP::BodyParsers;
use YAMLish;

class YAMLBodyParser does Cro::BodyParser {
    method is-applicable(Cro::HTTP::Message $message --> Bool) {
        with $message.content-type {
            .type-and-subtype eq 'text/yaml'
        }
        else { False }
    }

    method parse(Cro::HTTP::Message $message --> Promise) {
        start load-yaml await $message.body-text
    }
}
```

Using the YAML body parser

```
route {
    # Specify that all requests processed by this router
    # should consider our YAML body parser
    body-parser YAMLBodyParser;

    # Then this lot automatically works with YAML too!
    subset Rating of Int where 1..5;
    put -> 'reviews' {
        request-body -> (Rating :$rating!, Str :$comment) {
            say "$rating / 5 ($comment)";
        }
    }
}
```

Forms

`Cro::WebApp::Form`

Aim to take the tedium out of dealing with creating and handling web forms

Define the form as a class, using traits to specify controls and some validations

Template built-in to render the form

Very new, not so mature as the rest

(True of March 2020, if reading months later, likely already not true)

Define the form

```
class Review does Cro::WebApp::Form {
    has Str $.name;
    has Int $.rating is required
            is min(1) is max(5);
    has Str $.comment is required
            is multiline(rows => 3, cols => 80)
            is maxlength(1000);
}
```

Write a couple of templates

```
submit-review.crotmp
```

```
<h1>Submit a review</h1>
<&form(.form)>
```

```
thankyou.crotmp
```

```
<h1>Thank you!</h1>
<p>Your opinion is valuable to you.</p>
```

Write some routes, and we're done!

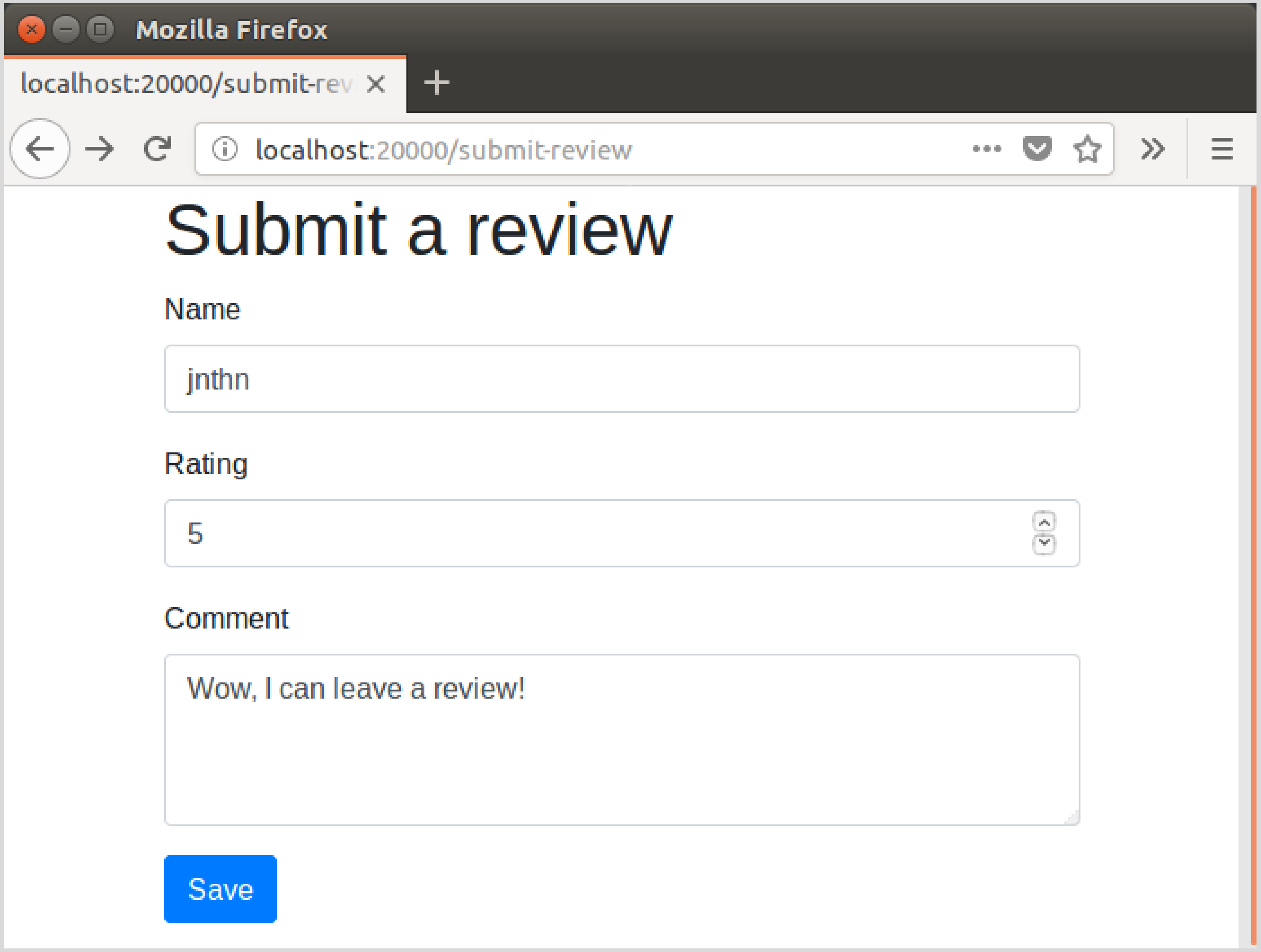
```
# Handle the initial request for the form
get -> 'submit-review' {
    template 'submit-review.crotmp',
            { form => Review.empty };
}

# Handle submitted data; if invalid, render the form again
post -> 'submit-review' {
    form-data -> Review $form {
        if $form.is-valid {
            say "Save $form.raku()";
            template 'thankyou.crotmp';
        }
        else {
            template 'submit-review.crotmp', { :$form };
        }
    }
}
```

Use Bootstrap so it looks nicer

```
<:use Cro::WebApp::Template::Bootstrap>
<html>
  <head>
    <&bs-head>
  </head>
  <body>
    <|bs-container>
      <h1>Submit a review</h1>
      <&bs-form(.form)>
    </|>
  </body>
</html>
```

Use Bootstrap so it looks nicer



Submit a review

Name

jnthn

Rating

5

Comment

Wow, I can leave a review!

Middleware

Ways to hook into the pipeline

`before`

Applied before any route matching takes place

`before-matched`

Applies before running a matched route's handler

`after-matched`

Applies after running a matched route's handler

`after`

Applies after routing, whether a route matched or not

Server level before/after

Nothing to do with the router, applies to everything

Add cacheability headers to assets

```
route {
    # Add a cache-control header to everything route that is
    # handled by this route block.
    after-matched {
        cache-control :public, :max-age(180);
    }

    get -> 'css', *@path {
        static 'static-content/css', @path
    }
    get -> 'js', *@path {
        static 'static-content/js', @path
    }
    get -> 'images', *@path {
        static 'static-content/images', @path
    }
}
```

Render a template for 404 errors

```
route {
    # This must be after, since we want it to run when we
    # failed to match a route
    after {
        if .status == 404 {
            template 'not-found.crotmp';
        }
    }

    ...
}
```

Block vs. class middleware

Block

Specific to the router; can't be used at the server level

Can use the router functions

Get called with request or response as the topic

Re-use achieved by factoring out to a sub

Implemented in terms of class middleware

Class

Can be used at server and router level

Can't use router functions

You get the request or response `Supply`

Re-use? Sure, it's just a class!

A bit less overhead

Randomly delay 10% of requests

```
use Cro::HTTP::Middleware;

class DelayMonkey does Cro::HTTP::Middleware::Request {
    method process(Supply $requests --> Supply) {
        supply whenever $requests -> $req {
            if rand < 0.1 {
                whenever Promise.in((1..5).pick) {
                    emit $req;
                }
            }
            else {
                emit $req;
            }
        }
    }
}
```

Using DelayMonkey

```
route {
    # Insert the middleware by passing it to before (it's
    # possible with before-matched too)
    before DelayMonkey;

    ...
}
```

Composing routers

Why not just one `route` block?

Difficult to maintain

We'd probably prefer 10 focused `route` blocks of 100 lines each than one 1000 line `route` block!

Different middleware requirements

We'd probably like to put cache control headers onto our assets - but certainly not onto everything!

Repeating a prefix gets boring

Do we really want to write `'shop'` 50 times for all 50 routes under `/shop`?

Flattening inclusion with `include`

```
# Factor them out (sub could be in a module and exported)
sub assets() {
    route {
        after-matched {
            cache-control :public, :max-age(180);
        }

        get -> 'css', *@path {
            static 'static-content/css', @path
        }
    }
}

# Include them at the top level route block.
my $top-level = route {
    include assets();
    ...
}
```

Flattening inclusion with `include`

```
# Factor them            ub could be in a module and exported)
sub assets
    route
        af
                                                    80);
        }

        get -> 'css', *@path {
            static 'static-content/css', @path
        }
    }
}

# Include them at the top level route block.
my $top-level = route {
    include assets();
    ...
}
```

Compiled into a single route matcher → can factor out without breaking routing

Prefixing with `include`

```
# Routes for the blog
sub blog() {
    route {
        get -> { ... }
        get -> Int $post-id { ... }
        get -> Int $post-id, 'comments' { ... }
    }
}

# Host them under /blog
my $top-level = route {
    include blog => blog();
    ...
}
```

More examples of `include`

```
my $top-level = route {
    # We can actually call include once and specify a whole
    # load of different mappings.
    include
        assets(),
        blog => blog(),
        # When wanting multiple prefix segments, pass them
        # as a list; if you use a "/" in a string it means a
        # url-encoded "/".
        <shop products> => products(),
        <shop basket> => basket();
}
```

What you can't do with `include`

```
# A route block using before or after...
sub cms() {
    route {
        after {
            if .status == 404 {
                template 'not-found.crotmp';
            }
        }
        ...
    }
}

# ...cannot be used with a flattening include, since there'd
# be no way to know whether to run the middleware without
# matching, but before/after are independent of that!
my $top-level = route {
    include cms => cms();  # Error!
    ...
}
```

Instead, use delegate

```
# A route block using before or after...
sub cms() {
    route {
        after {
            if .status == 404 {
                template 'not-found.crotmp';
            }
        }
        ...
    }
}

# ...can be used with delegate. Note that we need to say
# that all routes under cms should be delegated, using *.
my $top-level = route {
    delegate <cms *> => cms();
    ...
}
```

Instead, use delegate

```
# A route block using before or aft
sub cms() {
    route
        af
                template 'not-found.crotmp';
            }
        }
        ...
    }
}
```

Makes a single entry into our route table, and the inner route block does its own dispatch

```
# ...can be used with delegate. Note that we need to say
# that all routes under cms should be delegated, using *.
my $top-level = route {
    delegate <cms *> => cms();
    ...
}
```

Instead, use delegate

```
# A route block using before or afte
sub cms() {
    route
        af
                template 'not-found.crotmp';
            }
        }
        ...
    }
}

# ...can be used with delegate. Note that we need to say
# that all routes under cms should be delegated, using *.
my $top-level = route {
    delegate <cms *> => cms();
    ...
}
```

In fact, the target need not be a route block; it can be any `Cro::Transform` that maps HTTP requests into HTTP responses.

Sessions and auth

A session is just a class

```
# It's most convenient if we arrange for our session object
# to do the Cro::HTTP::Auth marker role. It's not mandatory,
# but you'll have to work a little harder in your routes
# if you don't.
my class ExampleSession does Cro::HTTP::Auth {
    has Int $.views = 0;

    method add-view() {
        $!views++;
    }
}
```

Add a session store, and we're done

```
route {
    # Session store is just a piece of middleware. It must
    # be applied with before, not before-matched, if we're
    # to use it to do auth-based routing based. There's also
    # various persistent alternatives.
    before Cro::HTTP::Session::InMemory[ExampleSession].new:
            cookie-name => 'MY_TEST_SITE';

    # Take the session as the first parameter (this is where
    # the Cro::HTTP::Auth marker comes in!)
    get -> ExampleSession $session, 'count-views' {
        $session.add-view();
        content 'text/plain', "$session.views() view(s)";
    }
}
```

Add a session store, and we're done

```
route {
    # Session store is just a piece of middleware. It must
    # be applied with before, not before-matched, if we're
    # to use it to do auth-based routing based. There's also
    # various persistent alternatives.
    before Cro::HTTP::Session::InMemory[ExampleSession].new:
            cookie-name => 'MY_TEST_SITE';

    # Take the session as the first parameter (this is where
    # the Cro::HTTP::Auth marker comes in!)
    get -> ExampleSession $session, 'count-views' {
        $session.add-view();
        content 'text/plain', "$session.views() view(s)";
    }
}
```

This is bound to whatever is in `request.auth` (and that's what the session middleware populates)

Setup for basic authentication

```
# Again, a session class, which we'll have created with a
# username on successful login.
my class ExampleSession does Cro::HTTP::Auth {
    has Str $.username;

    method logged-in() {
        ?$!username
    }
}

# A class implementing the Cro::HTTP::Auth::Basic role,
# providing the method that does the password check.
my class OurAuth does Cro::HTTP::Auth::Basic[ExampleSession,
                                              'username'] {
    method authenticate($user, $pass) {
        $pass eq 'hunter2'
    }
}
```

Use basic authentication

```
route {
    # Add our basic auth implementation as middleware.
    before OurAuth.new(realm => 'Test site');

    # Subset type for a session where we're logged in (this
    # is not so important for basic auth, but when doing
    # form-based login would matter).
    subset LoggedIn of ExampleSession where .logged-in;

    # Make sure we have a logged in user; automatic 401 if
    # we don't have one.
    get -> LoggedIn $user, 'test' {
        content 'text/plain', 'You are logged in';
    }
}
```

WebSockets

Of course the demo is a chat app...

```
# We'll send a HTML page that contains the WebSocket
# JavaScript; it's in a Cro template to keep it out of
# the route code.
get -> 'chat' {
    template 'chat.crotmp';
}
```

The HTML / JavaScript

```
<script>
  var name = window.prompt("Who are you?");
  var ws = new WebSocket("ws://localhost:20000/chat/ws");
  ws.onmessage = function (message) {
      var div = document.createElement("div");
      div.innerText = message.data;
      document.getElementById('messages').appendChild(div);
  }
  function send() {
      var textbox = document.getElementById('to-send');
      ws.send("<" + name + "> " + textbox.value);
      textbox.value = '';
  }
</script>
<div id="messages"></div>
<form>
  <input type="text" id="to-send" />
  <button type="button" onclick="send()">Send</button>
</form>
```

The WebSocket handler

```
# We use this to broadcast messages to all clients
my $chatter = Supplier.new;

# WebSocket handler for the chat application
get -> 'chat', 'ws' {
    web-socket -> $incoming {
        supply {
            whenever $incoming -> $ws-message {
                whenever $ws-message.body-text {
                    $chatter.emit($_);
                }
            }
            whenever $chatter {
                emit $_;
            }
        }
    }
}
```

The WebSocket handler

```
# We use this to broadcast messages to all clients
my $chatter = Supplier.new;

# WebSocket handler for the chat application
get -> 'chat', 'ws' {
    web-socket -> $incoming {
        supply {
            whenever $incomi     -> $ws-message {
                whenever $ws-me      e.body-text {
                    $chatter.emi
                }
            }
            whenever $chatte  {
                emit $_;
            }
        }
    }
}
```

A Supply of messages received from the client

The WebSocket handler

```
# We use this to broadcast messages to all clients
my $chatter = Supplier.new;

# WebSocket handler for the chat application
get -> 'chat', 'ws' {
    web-socket -> $incoming {
        supply {
            whenever $incoming -> $ws-message {
                whenever $w -message.body-text {
                    $chatter.  it($_);
                }
            }
            whenever $chatte
                emit $_;
            }
        }
    }
}
```

Whenever the client sends us a message...

The WebSocket handler

```
# We use this to broadcast messages to all clients
my $chatter = Supplier.new;

# WebSocket handler for the chat application
get -> 'chat', 'ws' {
    web-socket -> $incoming {
        supply {
            whenever $incoming -> $ws-message {
                whenever $ws-message.body-text {
                    $chatter.emit($_);
                }
            }
            whenever $chatter {
                emit $_;
            }
        }
    }
}
```

...wait for its body text to arrive, and then...

The WebSocket handler

```
# We use this to broadcast messages to all clients
my $chatter = Supplier.new;

# WebSocket handler for the chat application
get -> 'chat', 'ws' {
    web-socket -> $incoming {
        supply {
            whenever $incoming -> $ws-message {
                whenever $ws-message.body-text {
                    $chatter.emit($_);
                }
            }
            whenever $chatter {
                emit $_;
            }
        }
    }
}
```

...broadcast them to everyone

The WebSocket handler

```
# We use this to broadcast messages to all clients
my $chatter = Supplier.new;

# WebSocket handler for the chat application
get -> 'chat', 'ws' {
    web-socket -> $incoming {
        supply {
            whenever $incoming -> $ws-message {
                whenever $ws-message.body-text {
                    $chatter.emit($_);
                }
            }
            whenever $chatter {
                emit $_;
            }
        }
    }
}
```

Whenever any message is broadcast, send it to the client

Questions?

Learn more: https://cro.services/