Rewrite Paths and Track 404s

online.pragmaticstudio.com/courses/elixir/modules/8

Notes

Exercise: Rewriting

URLs are commonly rewritten to make them prettier or to juice up a page's SEO. Just for practice, add a function to the pipeline that rewrites requests for `/bears?id=1` to `/bears/1`, `/bears?id=2` to `/bears/2`, and so on. Here's an example request:

```
request = """
GET /bears?id=1 HTTP/1.1
Host: example.com
User-Agent: ExampleBrowser/1.0
Accept: */*

"""
```

Show Answer

Exercise: Emojify Worthy Responses

Just for fun, suppose you want your web server to decorate all responses that have a 200 status with emojies before and after the actual content.

Write an `emojify` function that emojifies those responses and plug that function into the pipeline.

- Show Hint
- Show Answer

Exercise: Handle DELETE Differently

Change the `DELETE` route you created in a previous exercise to pattern match on the `conv` map, rather than taking three arguments.

Show Answer

Thought Experiment

Can you think of other useful functions to add to the pipeline, either to modify the request before its routed or modify the response before it's sent back? You may need to consider additional fields in the conversation map that are affected by the functions.

Alternative Approach Using Regular Expressions

In the first exercise above, you rewrote requests for `/bears?id=1`. Now suppose you want

to do the same for lions, tigers, and so on. But rather than writing separate `rewrite_path` function clauses for each wildthing, you want to handle them all in one generic `rewrite_path` function.

You might expect you could change the pattern to include a `thing` variable, like so:

```
def rewrite_path(%{path: "/" <> thing <> "?id=" <> id} = conv) do
  %{ conv | path: "/#{thing}/#{id}" }
end
```

But that gives the following compile error:

```
** (CompileError) a binary field without size is only allowed at the end of a binary
pattern
```

When pattern matching strings, you can't use a variable on the left side of the `<>` operator.

A more flexible approach is to use a regular expression. (Every time we make a course, someone asks about regular expressions. If you're that someone, here you go!)

For example, suppose we have the following path:

```
iex> path = "/bears?id=1"
```

Here's how to define a regular expression literal in Elixir:

```
~r{regexp}
```

The `~r` is called a *sigil* and the braces `{ }` are delimiters for the regular expression itself. For example, here's a regular expression that matches `/bears?id=1`, `/lions?id=7`, `/tigers?id=100`, and so on:

```
iex> regex = ~r{\/(\w+)\?id=(\d+)}
```

It matches a literal `/` character followed by one or more word characters, followed by the literal `?id=` followed by one or more digits.

The `Regex` module defines functions for working with regular expressions. For example, we can check that it matches the path by calling the `match?` function:

```
iex> Regex.match?(regex, path)
true
```

Here's another way to write the same regular expression and capture the matching values:

```
iex> regex = ~r{\/(?<thing>\w+)\?id=(?<id>\d+)}
```

Notice we added `?` before the word characters and `?` before the digit characters. This says we want to capture the word characters as `thing` and the digit characters as `id`.

Now we can call the `named_captures` function which returns the given captures as a map:

```
iex> Regex.named_captures(regex, path)
%{"id" => "1", "thing" => "bears"}
```

Hey, now that's handy!

And just to show that it works with other matching paths such as `/lions?id=7`:

```
iex> path = "/lions?id=7"

iex> Regex.named_captures(regex, path)
%{"id" => "7", "thing" => "lions"}
```

If no captures are found, it returns `nil`. For example, this path is missing the id:

```
iex> path = "/bears"

iex> Regex.named_captures(regex, path)
nil
```

Armed with this newfound knowledge, we can write a generic `rewrite_path` function:

```
def rewrite_path(%{path: path} = conv) do
  regex = ~r{\/(?<thing>\w+)\?id=(?<id>\d+)}
  captures = Regex.named_captures(regex, path)
  rewrite_path_captures(conv, captures)
end
```

Since `Regex.named_captures` returns a map or `nil`, we can delegate to a new set of `rewrite_path_captures` function clauses and pattern match on the argument:

```
def rewrite_path_captures(conv, %{"thing" => thing, "id" => id}) do
  %{ conv | path: "/#{thing}/#{id}" }
end

def rewrite_path_captures(conv, nil), do: conv
```

Notice that we pass the `conv` as the first argument and the result of `Regex.named_captures` as the second argument. If the regular expression found captures, then the first function is called which uses the values of the `thing` and `id` keys in the map to update the `path` in the `conv` map. Otherwise, if no captures were found, then the second function is called which returns the `conv` map unchanged.

Pretty cool, eh?

Open-Ended Exercise: Use the Logger

In the video we tracked a 404 by simply printing a warning to the console. If you want to get a tad fancier, you can use the built-in `Logger` module which supports various levels of logging and a wide range of configuration options.

First, spend a minute perusing the documentation using the `h` helper function in an `iex` session. Did we mention that `Logger` has a *lot* of options?

Then to use `Logger` in your code, you'll first need to require it by adding the following line inside of your `Servy.Handler` module:

```
require Logger
```

The `Logger` module uses Elixir macros, so you have to `require` it for the macros can do their magic.

Then you can call logging functions, such as:

```
Logger.info "It's lunchtime somewhere."
Logger.warn "Do we have a problem, Houston?"
Logger.error "Danger Will Robinson!"
```

Go ahead and have some fun with it, but don't forget to come back and actually continue on with the course!

Code So Far

The code for this video is in the `rewriting` directory found within the `video-code` directory of the code bundle.

Go To Next Video and mark this step complete!