High-Level Transformations

Notes

HTTP Request and Response

To save you the trouble of typing it, here's the request we used in the video:

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

"""
```

And here's the expected response:

```
expected_response = """
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 20

Bears, Lions, Tigers
"""
```

Pipelines As Outlines

At this point we have an outline for our program expressed as a pipeline of functions:

```
request
|> parse
|> route
|> format_response
```

The program doesn't do anything interesting yet since the functions simply return hard-coded data. But we like to start this way as it helps us think through the program as a series of data transformations.

And it just so happens it's also a to-do list! We'll implement each of the functions in subsequent modules.

Nested Function Calls

In the video, we started by sequencing the three transformation functions in a traditional style using intermediate variables:

```
def handle(request) do
  conv = parse(request)
  conv = route(conv)
  format_response(conv)
end
```

You could condense this a bit and remove the temporary variables by nesting the function calls like so:

```
format_response(route(parse(request)))
```

But this is difficult to read as the functions resolve from the inside out.

Using the pipe operator lets us unwind these nested calls by chaining the functions together in a more readable style:

```
request
|> parse
|> route
|> format_response
```

It transforms the request by *piping* the result of each function into the next function. Here's a fun fact: At compile time this code is transformed to the nested call version.

A Piping Analogy

If you've ever piped multiple commands together in the Unix operating system, then Elixir's pipe operator will be familiar. If not, think of it kinda like that game where one person whispers a story to another person, who whispers it to the next person, and so on until the last person announces the final message.

In retelling the story, it inevitably gets changed as it passes from person to person. In the same way, each function can transform what it's given and pass something entirely different to the next function in line.

Code So Far

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

Go To Next Video