Level 5–2

The Mix Tool

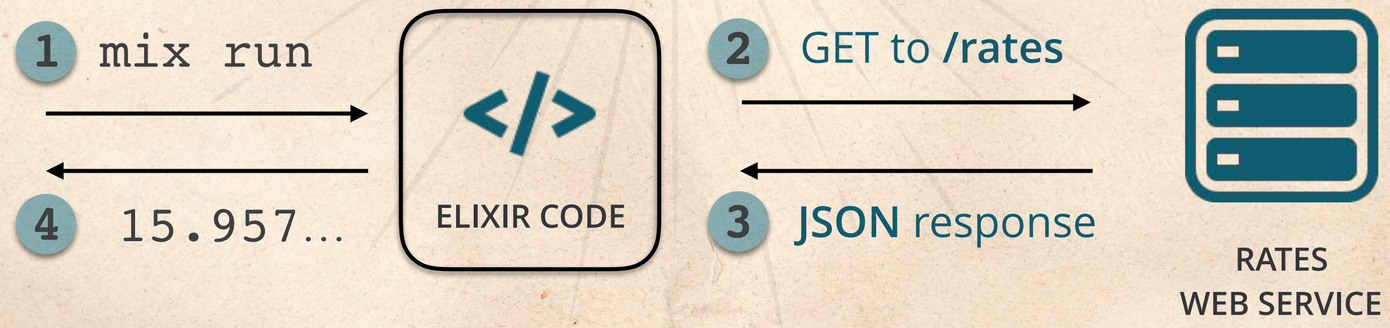
Working With Third-party Dependencies

Converting From Euro to Dollar

Let's write a new function from_euro_to_dollar() **that takes an amount in € euros as its single argument and converts it to** US$ **dollars. We'll fetch the rate of the day from an external web service API.**

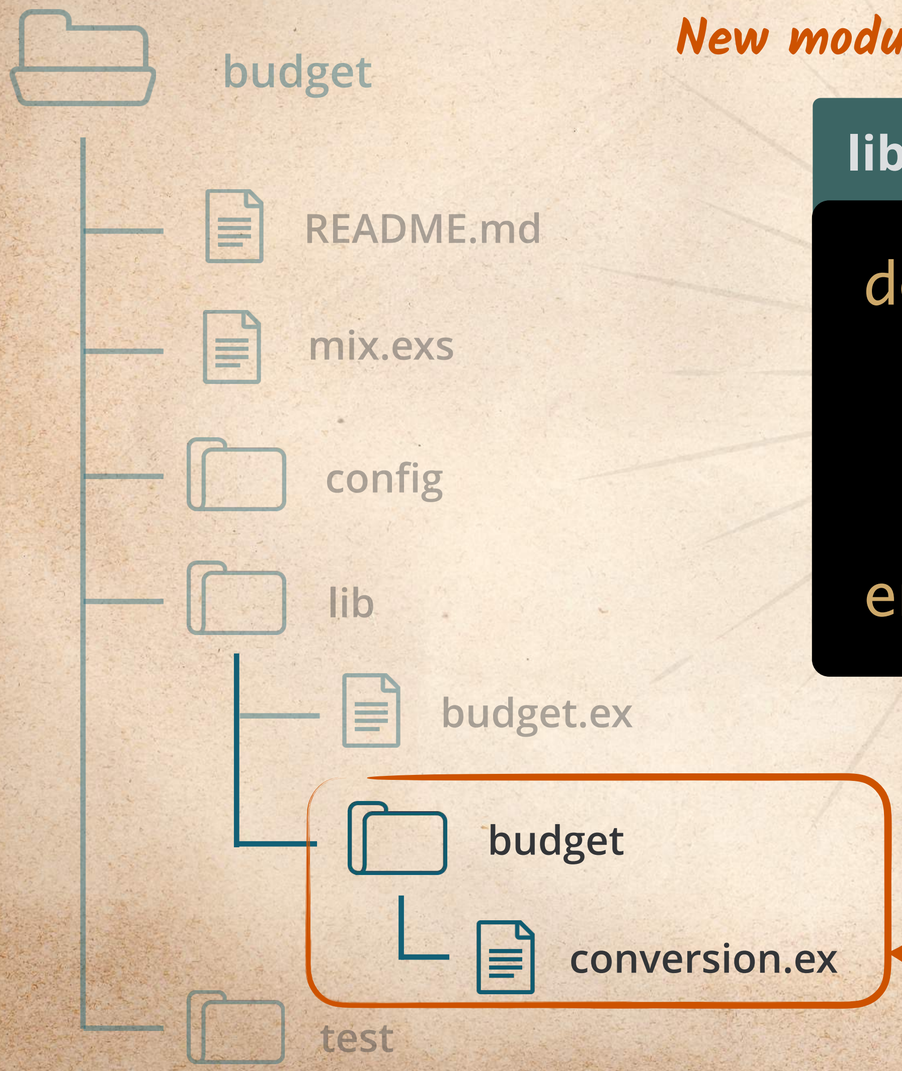
```
$ mix run -e "Budget.Conversion.from_euro_to_dollar(15) |> IO.puts"
```

```
15.957446808510639
```



Creating a New Module

The new function will be part of the Conversion **module, which itself is a submodule of** Budget.



New module part of the Budget module

lib/budget/conversion.ex

```
defmodule Budget.Conversion do
  def from_euro_to_dollar(amount) do
      ...
  end
end
```

Create new folder and new file

Declaring Third-party Dependencies

We use the mix.exs file to declare **library dependencies** our program depends on.

mix.exs

```
defmodule Budget.Mixfile do
  ...

  defp deps do
    [{:httpoison, "~> 0.10.0"}, {:poison, "~> 3.0"}]
  end
end
```

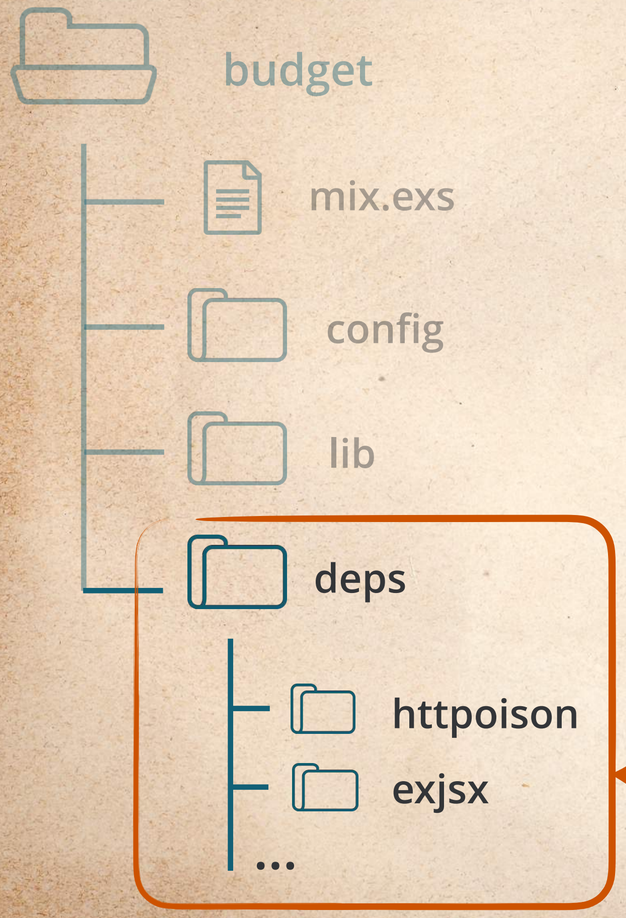
Version numbers following Semantic Versioning

Third-party library dependencies

List of tuples

Installing Third-party Dependencies

The command mix deps.get **fetches dependencies from a remote repository and installs them locally.**




```
$ mix deps.get
```

```
Running dependency resolution
* Getting httpoison (Hex package)
  Checking package (https://repo.hex.pm/tarballs/httpoison-0.10.0.tar)
  Using locally cached package
* Getting poison (Hex package)
  Checking package (https://repo.hex.pm/tarballs/poison-3.0.0.tar)
  Using locally cached package
...
```

Each third-party dependency is stored inside the deps directory.

Making HTTP Calls With the HTTPoison Library

The HTTPoison library is what we'll use to make HTTP calls to the remote web service.

lib/budget/conversion.ex

```
defmodule Budget.Conversion do
  def from_euro_to_dollar(amount) do
    url = "cs-currency-rates.codeschool.com/currency-rates"
    case HTTPoison.get(url) do
      {:ok, response} -> parse(response) |> convert(amount)
      {:error, _} -> "Error fetching rates"
    end
  end
  ...
end
```

Takes result of parse(response) as first argument

Using pattern matching to determine whether the HTTP call was successful

Parsing JSON With the JSX library

We use pattern matching to store the response body on the json_response **variable and the** Poison **library to parse JSON to an Elixir *tuple.***

lib/budget/conversion.ex

```
defmodule Budget.Conversion do
  ...
  defp parse(%{status_code: 200, body: json_response}) do
    Poison.Parser.parse(json_response)
  end
  ...
end
```

Returns a tuple

defp ***means it's a private function, not to be called from outside its enclosing module.***

From JSON to List of Tuples

The parse **function converts the JSON response from the remote server to a tuple, and passes it as the first argument to the** convert **function.**

```
[
  { "currency": "euro", "rate": 0.94 },
  { "currency": "pound", "rate": 0.79 }
]
```

JSON response

RATES
WEB SERVICE

JSON response

```
parse(response) |> convert( ,amount)
```

Elixir tuple

```
{:ok, [
        %{"currency" => "euro", "rate" => 0.94},
        %{"currency" => "pound", "rate" => 0.79}
      ]}
```

Finding Rates and Converting

The convert **function grabs the list of tuples via pattern matching and calls** find_euro **to find the rate for € euro. Lastly, it performs the conversion operation.**

lib/budget/conversion.ex

```
defmodule Budget.Conversion do
  ...
  defp convert({:ok, rates}, amount) do
    rate = find_euro(rates)
    amount / rate
  end
  ...
end
```

Pattern matching

Using Recursion to Find the Rate

We'll use pattern matching and recursion to find the rate for € euro from the list of all rates available.

lib/budget/conversion.ex

```
defmodule Budget.Conversion do
  ...
  defp find_euro([%{"currency" => "euro", "rate" => rate} | _]) do
    rate
  end
  defp find_euro([_ | tail]) do
    find_euro(tail)
  end
  defp find_euro([]) do
    raise "No rate found for Euro"
  end
end
```

When this match is successful...

...we return the rate!

No match on first element, so the function calls itself with the rest of the list.

No match and no more elements on the list, so we interrupt the program by raising an error.

Running the Complete Program

We can run the program using mix run **and see the expected results printed to the screen.**


```
$ mix run -e "Budget.Conversion.from_euro_to_dollar(15) |> IO.puts"
```

```
15.957446808510639
```

Running With the Rates Web Service Down

If the rates web service is unavailable, running the program prints the friendly error message.


```
$ mix run -e "Budget.Conversion.from_euro_to_dollar(15) |> IO.puts"
```

```
Error fetching rates
```