Chapter 1

Stream live cryptocurrency prices from the Binance WSS

1.1 Objectives

- create a new umbrella app
- create a supervised application inside an umbrella
- connect to Binance's WebSocket Stream using the WebSockex module
- define a TradeEvent struct that will hold incoming data
- decode incoming events using the Jason module

1.2 Create a new umbrella app

As we are starting from scratch, we need to create a new umbrella project:

```
mix new hedgehog --umbrella
```

1.3 Create a supervised application inside an umbrella

We can now proceed with creating a new supervised application called `streamer` inside our umbrella:

```
cd hedgehog/apps
mix new streamer --sup
```

1.4 Connect to Binance's WebSocket Stream using the WebSockex module

To establish a connection to Binance API's stream, we will need to use a WebSocket client. The module that we will use is called WebSockex. Scrolling down to the `Installation` section inside the module's readme on Github, we are instructed what dependency we need to add to our project.

We will append `:websockex` to the `deps` function inside the `mix.exs` file of the `streamer` application:

```
  # /apps/streamer/mix.exs
  defp deps do
    [
      {:websockex, "~> 0.4"}
    ]
  end
```

As we added a dependency to our project, we need to fetch it using `mix deps.get`.

We can now progress with creating a module that will be responsible for streaming. We will create a new file called `binance.ex` inside the `apps/streamer/lib/streamer` directory.

From the readme of WebSockex module, we can see that to use it we need to create a module that will implement the `WebSockex` behavior:

```
# WebSockex's readme
defmodule WebSocketExample do
  use WebSockex

  def start_link(url, state) do
    WebSockex.start_link(url, __MODULE__, state)
  end

  def handle_frame({type, msg}, state) do
    IO.puts "Received Message - Type: #{inspect type} -- Message: #{inspect msg}"
    {:ok, state}
  end

  def handle_cast({:send, {type, msg} = frame}, state) do
    IO.puts "Sending #{type} frame with payload: #{msg}"
    {:reply, frame, state}
  end
end
```

We will copy the whole code above across to our new `binance.ex` file.

The first step will be to update the module name to match our file name:

```
# /apps/streamer/lib/streamer/binance.ex
defmodule Streamer.Binance do
```

In the spirit of keeping things tidy - we will now remove the `handle_cast/2` function (the last function in our module) as we won't be sending any messages back to Binance via WebSocket (to place orders etc - Binance provides a REST API which we will use in the next chapter).

Next, let's look up what URL should we use to connect to Binance's API. Binance has a separate WSS (Web Socket Streams) documentation at Github.

Scrolling down we can see the `General WSS information` section where 3 important pieces of information are listed:

- The base endpoint is: `wss://stream.binance.com:9443`
- Raw streams are accessed at `/ws/<streamName>`
- All symbols for streams are *lowercase*

We can see that the full endpoint for raw streams(we will be using a "raw" stream) will be `wss://stream.binance.com:9443/ws/` with stream name at the end (together with lowercased symbol).

Note: In the context of Binance API, "raw" means that no aggregation was performed before broadcasting the data on WebSocket.

Let's introduce a module attribute that will hold the full raw stream endpoint which will be used across the module:

```
# /apps/streamer/lib/streamer/binance.ex
@stream_endpoint "wss://stream.binance.com:9443/ws/"
```

Now back in Binance's WSS documentation we need to search for "Trade Streams". "trade" in the context of this documentation means an exchange of assets(coins/tokens) by two sides (buyer and seller). Our future trading strategy will be interested in the "latest price" which is simply the last trade event's price.

We can see that docs are pointing to the following stream name:

```
Stream Name: <symbol>@trade
```

Together, our full URL looks like: "wss://stream.binance.com:9443/ws/@trade". To give a concrete example: the raw trade events stream URL for symbol XRPUSDT is: "wss://stream.binance.com:9443/ws/xrpusdt@trade" (remember that symbols need to be lowercased, otherwise no trade events will get streamed - there's *no* error).

Back to the IDE, we will now modify the `start_link/2` function to use Binance API's URL:

```
  # /apps/streamer/lib/streamer/binance.ex
  def start_link(symbol) do
    symbol = String.downcase(symbol)

    WebSockex.start_link(
      "#{@stream_endpoint}#{symbol}@trade",
      __MODULE__,
      nil
    )
  end
```

Instead of passing an URL, we modified the function to accept a `symbol`, downcase it and use it together with the module's `@stream_endpoint` attribute to build a full URL.

At this moment streaming of trade events already works which we can test using `iex`:

```
$ iex -S mix
...
iex(1)> Streamer.Binance.start_link("xrpusdt")
{:ok, #PID<0.335.0>}
Received Message - Type: :text -- Message: "{\"e\":\"trade\", \"E\":1603226394741,
  \"s\":\"XRPUSDT\",\"t\":74608889,\"p\":\"0.24373000\",\"q\":\"200.00000000\",
  \"b\":948244411,\"a\":948244502,\"T\":1603226394739,\"m\":true,\"M\":true}"
```

We can see the messages logged above because we copied the sample implementation from WebSockex's readme where `handle_frame/2` function uses `IO.puts/1` to print out all incoming data. The lesson here is that every incoming message from Binance will cause the `handle_frame/2` callback to be called with the message and the process' state.

Just for reference, our module should look currently as follows:

```
# /apps/streamer/lib/streamer/binance.ex
defmodule Streamer.Binance do
  use WebSockex

  @stream_endpoint "wss://stream.binance.com:9443/ws/"

  def start_link(symbol) do
    symbol = String.downcase(symbol)

    WebSockex.start_link(
      "#{@stream_endpoint}#{symbol}@trade",
      __MODULE__,
      nil
```

```
    )
  end

  def handle_frame({type, msg}, state) do
    IO.puts "Received Message - Type: #{inspect type} -- Message: #{inspect msg}"
    {:ok, state}
  end
end
```

1.5 Decode incoming events using the Jason module

Currently, all incoming data from WebSocket is encoded as a JSON. To decode JSON we will use the jason module.

Scrolling down to the `Installation` section inside the module's readme, we can see that we need to add it to the dependencies and we can start to use it right away.

Let's open the `mix.exs` file of the `streamer` application and append the `:jason` dependency to the list inside `deps` function:

```
  # /apps/streamer/mix.exs
  defp deps do
    [
      {:jason, "~> 1.2"},
      {:websockex, "~> 0.4"}
    ]
  end
```

As previously, don't forget to run `mix deps.get` to fetch the new dependency.

Looking through the documentation of the Jason module we can see `encode!/2` and `decode!/2` functions, both of them have exclamation marks which indicate that they will throw an error whenever they will be unable to successfully encode or decode the passed value.

This is less than perfect for our use case as we would like to handle those errors in our own way(technically we could just use try/rescue but as we will find out both `encode/2` and `decode/2` are available).

We will go a little bit off-topic but I would highly recommend those sorts of journeys around somebody's code. Let's look inside the Jason module. Scrolling down in search of `decode/2` (without the exclamation mark) we can see it about line 54:

```
  # /lib/jason.ex
  def decode(input, opts \\ []) do
    input = IO.iodata_to_binary(input)
    Decoder.parse(input, format_decode_opts(opts))
  end
```

It looks like it uses the `parse/2` function of a `Decoder` module, let's scroll back up and check where it's coming from. At line 6:

```
# /lib/jason.ex
alias Jason.{Encode, Decoder, DecodeError, EncodeError, Formatter}
```

we can see that `Decoder` is an alias of the `Jason.Decoder`. Scrolling down to the `Jason.Decoder` module we will find a `parse/2` function about line 43:

```
  # /lib/decoder.ex
  def parse(data, opts) when is_binary(data) do
    key_decode = key_decode_function(opts)
    string_decode = string_decode_function(opts)
    try do
      value(data, data, 0, [@terminate], key_decode, string_decode)
    catch
      {:position, position} ->
        {:error, %DecodeError{position: position, data: data}}
      {:token, token, position} ->
        {:error, %DecodeError{token: token, position: position, data: data}}
    else
      value ->
        {:ok, value}
    end
  end
```

Based on the result of decoding it will either return `{:ok, value}` or `{:error, %Decode.Error{...}}` we can confirm that by digging through documentation of the module on the hexdocs.

Once again, the point of this lengthy investigation was to show that Elixir code is readable and easy to understand so don't be thrown off when documentation is a little bit light, quite opposite, contribute to docs and code as you gain a better understanding of the codebase.

We can now get back to our `Streamer.Binance` module and modify the `handle_frame/2` function to decode the incoming JSON message. Based on the result of `Jason.decode/2` we will either call the `process_event/2` function or log an error. Here's the new version of the `handle_frame/2` function:

```
  # /apps/streamer/lib/streamer/binance.ex
  def handle_frame({_type, msg}, state) do
    case Jason.decode(msg) do
      {:ok, event} -> process_event(event)
      {:error, _} -> Logger.error("Unable to parse msg: #{msg}")
    end

    {:ok, state}
  end
```

Please make note that `type` is now prefixed with an underscore as we aren't using it at the moment.

The second important thing to note is that we are using `Logger` so it needs to be `required` at the beginning of the module:

```
  # /apps/streamer/lib/streamer/binance.ex
  require Logger
```

Before implementing the `process_event/2` function we need to create a structure that will hold the incoming trade event's data.

Let's create a new directory called `binance` inside the `apps/streamer/lib/streamer/` and a new file called `trade_event.ex` inside it.

Our new module will hold all the trade event's information but we will also use readable field names(you will see the incoming data below). We can start by writing a skeleton module code:

```
# /apps/streamer/lib/streamer/binance/trade_event.ex
defmodule Streamer.Binance.TradeEvent do
  defstruct []
end
```

We can refer to Binance's docs to get a list of fields:

```
{
  "e": "trade",     // Event type
  "E": 123456789,   // Event time
  "s": "BNBUSDT",   // Symbol
  "t": 12345,       // Trade ID
  "p": "0.001",     // Price
  "q": "100",       // Quantity
  "b": 88,          // Buyer order ID
  "a": 50,          // Seller order ID
  "T": 123456785,   // Trade time
  "m": true,        // Is the buyer the market maker?
  "M": true         // Ignore
}
```

Let's copy them across and convert the comments to update the `defstruct` inside the `Streamer.Binance.TradeEvent` module's struct to following:

```
  # /apps/streamer/lib/streamer/binance/trade_event.ex
  defstruct [
    :event_type,
    :event_time,
    :symbol,
```

```
    :trade_id,
    :price,
    :quantity,
    :buyer_order_id,
    :seller_order_id,
    :trade_time,
    :buyer_market_maker
  ]
```

That's all for this struct, we can now get back to implementing the `process_event/2` function inside the `Streamer.Binance` module. We will map every field of the response map to the `%Streamer.Binance.TradeEvent` struct. A useful trick here would be to copy the list of fields once again from the struct and assign the incoming fields one by one. Inside the header of the function, we will pattern match on event type(a field called "e" in the message) to confirm that indeed we received a trade event). In the end, the `process_event/2` function should look as follows:

```
# /apps/streamer/lib/streamer/binance.ex
defp process_event(%{"e" => "trade"} = event) do
  trade_event = %Streamer.Binance.TradeEvent{
    :event_type => event["e"],
    :event_time => event["E"],
    :symbol => event["s"],
    :trade_id => event["t"],
    :price => event["p"],
    :quantity => event["q"],
    :buyer_order_id => event["b"],
    :seller_order_id => event["a"],
    :trade_time => event["T"],
    :buyer_market_maker => event["m"]
  }

  Logger.debug(
    "Trade event received " <>
      "#{trade_event.symbol}@#{trade_event.price}"
  )
end
```

We added the `Logger.debug/2` function to be able to see logs of incoming trade events.

Lastly, before testing our implementation, let's add a nice interface to our `streamer` application that allows starting streaming:

```
# /apps/streamer/lib/streamer.ex
defmodule Streamer do
  @moduledoc """
  Documentation for `Streamer`.
  """

  def start_streaming(symbol) do
    Streamer.Binance.start_link(symbol)
  end
end
```

The final version of the `Streamer.Binance` module should look like this.

The last step will be to add the `Logger` configuration into the main `config/config.exs` file. We will set the `Logger` level to `:debug` for a moment to be able to see incoming trade events:

```
# /config/config.exs
config :logger,
  level: :debug
```

This finishes the implementation part of this chapter, we can now give our implementation a whirl using `iex`:

```
$ iex -S mix
...
iex(1)> Streamer.start_streaming("xrpusdt")
{:ok, #PID<0.251.0>}
23:14:32.217 [debug] Trade event received XRPUSDT@0.25604000
23:14:33.381 [debug] Trade event received XRPUSDT@0.25604000
23:14:35.380 [debug] Trade event received XRPUSDT@0.25605000
23:14:36.386 [debug] Trade event received XRPUSDT@0.25606000
```

As we can see, the streamer is establishing a WebSocket connection with Binance's API and its receiving trade events. It decodes them from JSON to `%Streamer.Binance.TradeEvent` struct and logs a compiled message. Also, our interface hides implementation details from the "user" of our application.

We will now flip the `Logger` level back to `info` so the output won't every incoming trade event:

```
# /config/config.exs
config :logger,
  level: :info
```

[Note] Please remember to run the `mix format` to keep things nice and tidy.

Source code for this chapter can be found at Github