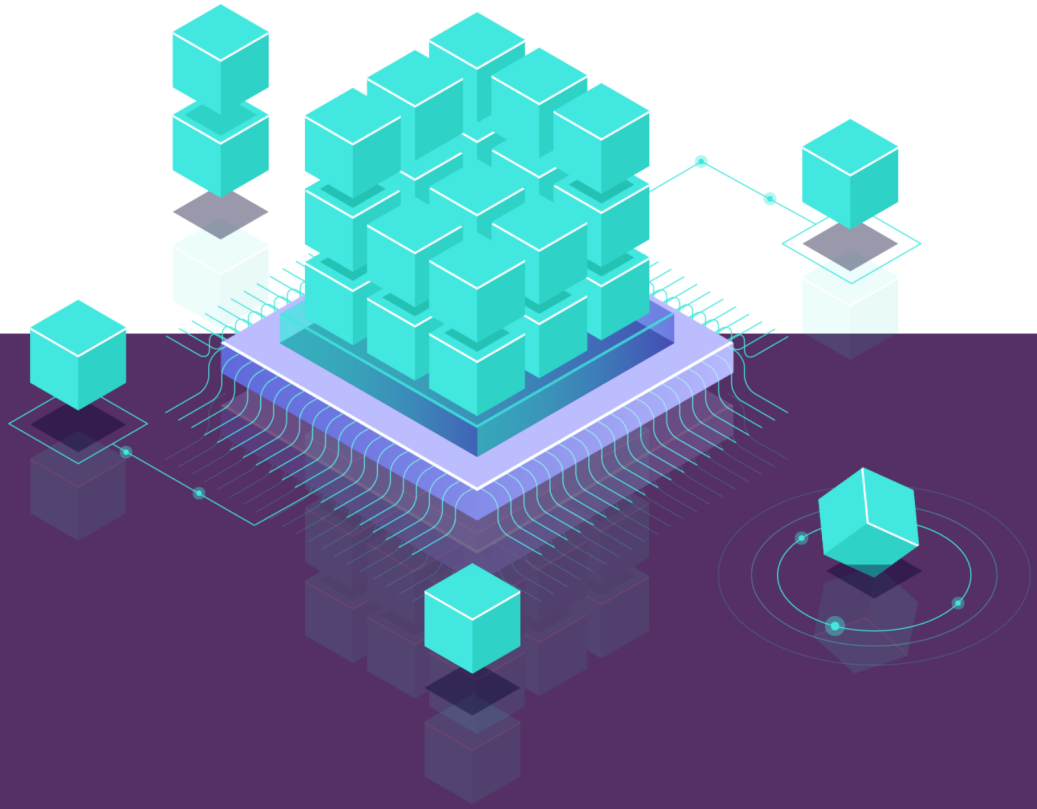
Hands-on Elixir & OTP

Cryptocurrency trading bot



Kamil Skowron

To my wife Sandra for putting up with my crazy ideas

Hands-on Elixir & OTP: Cryptocurrency trading bot

Kamil Skowron

0.3.1

Contents

Preface

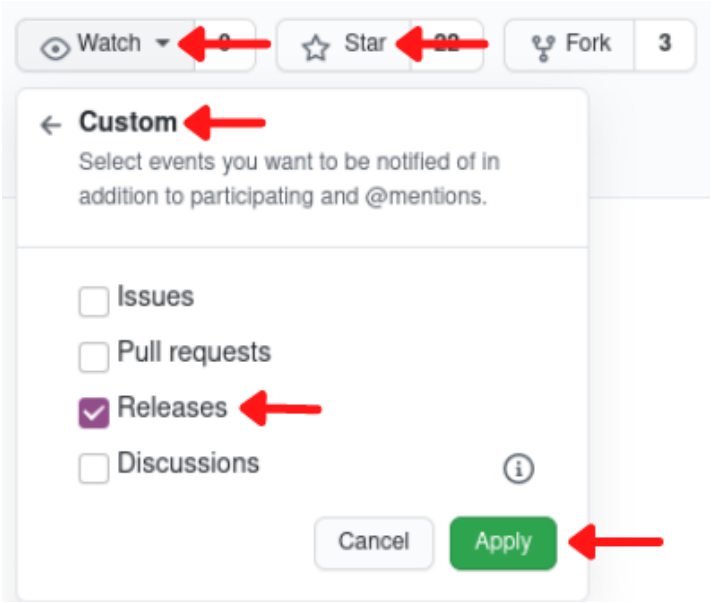
Want to learn Elixir & OTP by creating a real-world project?

With "Hands-on Elixir & OTP: Cryptocurrency trading bot", you will **gain hands-on experience by writing an interesting software project from scratch.** We will explore all the key abstractions and essential principles through iterative implementation improvements.

We will start by creating a new umbrella application, subscribing to WebSocket streams, implementing a basic trading flow, and focusing on improving it by expanding on the topics like supervision trees, resiliency, refactoring using macros, utilising the Registry, testing and others.

This book is 80% complete - chapters 1-17 are finished, and I'll add more content soon. It's also a loosely written representation of the Hands-on Elixir & OTP: Cryptocurrency trading bot video course released on YouTube.

To get notified about updates to this book just "watch" the source code's repository and don't forget to leave a star:



Limit of Liability/Disclaimer of Warranty

THIS BOOK IS NOT FINANCIAL ADVICE

THE SOFTWARE/BOOK IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE/BOOK OR THE USE OR OTHER DEALINGS IN THE SOFTWARE/BOOK.

PDF & EPUB

To keep this book up to date as well as publicly available for people that can't afford to pay for it, it's available in HTML format online free of charge.

PDF & EPUB formats are available to backers using either GitHub Sponsors(it supports both one-offs payments as well as "monthly" plans) or Gumroad.

Preface

In recent years Elixir programming language gained a lot of interest in the industry.

Its unmatched parallelization capabilities are unique and powerful, making it a great candidate for highly concurrent systems like the ones trading assets on exchanges.

In this book, we will go through the development process of a cryptocurrency trading bot in Elixir. We will start ground up and chapter by chapter progress with the implementation ending up with a fully-fledged *naive* trading strategy. We will be designing process supervision trees, describing why specific decisions were taken and how will they impact the system going forward.

By any stretch of the imagination, I don't believe that "this is *the only* way"(nor even the best way) to create a cryptocurrency trading bot in Elixir. This book focuses more on building a real-life project and iterating over it, taking decisions on the way as it would happen in a real work environment. Some parts will be "perfect" the first time around, but there are also others, where we will take compromises to "get it to working" and then when the time is right, we will refactor them as we will gain a better understanding of the domain problem.

Who this book is for

This book will be a great resource for everyone that already knows the basics of Elixir and wants to get a feel of how developing a non-trivial system looks like using it.

Readers do not need deep knowledge about cryptocurrencies to follow along as I will shy away from crypto/trading jargon as well as will explain it whenever it's unavoidable.

This is not a book focused on trading strategies, neither it's financial advice to trade at all. The main focus of this book is to showcase how the implementation of even complex problems can be achieved in Elixir by simple processes working together in an orchestrated manner.

The strategy described in this book is naive and most probably will end up losing money, but that's not the point of this book. **As we will build up the strategy we will face a spectrum of problems that developers face at work. It's a great primer if you want to get your first "project" behind your belt**.

So, if you've already gone through the motions and learned Elixir and OTP but still feel like you need to get your hands dirty with a "real problem" to "make it stick", this book is for you.

What this book covers

This book is an ongoing project, and at present, it contains the following chapters:

- Chapter 1 - Stream live cryptocurrency prices from the Binance WSS

Stream live cryptocurrency prices (trade events) from the Binance exchange. Starting grounds up, we will create a new umbrella project and a `streamer` application inside it. The streamer application will use a Websocket client called `WebSockex` to connect with the Binance API and receive a live feed. After receiving the event as a JSON string, we will decode it using the `jason` library and convert it to our data struct. At the end of the chapter, we will see decoded trade events being logged to the terminal.

- Chapter 2 - Create a naive trading strategy - single trader without supervision

In this chapter, we will create our first *naive* trading strategy. We will create another application inside our umbrella called `naive`. We will put data streamed to our `streamer` application to good use by sending it over to the `naive` application. We will start with a very basic solution consisting of a single process called `trader` that will utilize the `GenServer` behaviour. It will allow us to go through the full trading cycle and give us something that "works".

- Chapter 3 - Introduce PubSub as a communication method

To allow our trading strategy to scale to multiple parallel traders, we need to find a way to distribute the latest prices (trade events) to those multiple traders. We will introduce PubSub to broadcast messages from the streamer(s) to the trader(s). PubSub will allow us to break hardcoded references between applications in our umbrella and that will become a pattern that we will utilize moving forward.

- Chapter 4 - Mock the Binance API

Besides historical prices (trade events), to perform backtesting, we need to be able to mock placing orders and get trade events back as they are filled. In this chapter, we will focus on developing the solution that will allow our trader to "trade" without contacting the Binance exchange(for people without Binance accounts), it will also allow us to backtest our trading strategy.

- Chapter 5 - Enable parallel trading on multiple symbols

Our basic strategy implementation from the last chapter is definitely too basic to be used in a "production environment" - it can't be neither scaled nor it is fault-tolerant. In this chapter, we will upgrade our naive strategy to be more resilient. This will require a supervision tree to be created and will allow us to see different supervision strategies in action and understand the motivation behind using and stacking them.

- Chapter 6 - Introduce a `buy_down_interval` to make a single trader more profitable

At this moment our `Naive.Trader` implementation will blindly place a buy order at the price of the last trade event. Whenever the `Naive.Trader` process will finish trade, a new `Naive.Trader` process will be started and it will end up placing a buy order at the same price as the price of the previous sell order. This will cost us double the fee without gaining any advantage and would cause further complications down the line, so we will introduce a `buy_down_interval` which will allow the `Naive.Trader` processes to place a buy order below the current trade event's price.

- Chapter 7 - Introduce a trader budget and calculating the quantity

Since the second chapter, our `Naive.Trader` processes are placing orders with a hardcoded quantity of 100. In this chapter, we will introduce a budget that will be evenly split between the `Naive.Trader` processes using chunks. We will utilize that budget to calculate quantity (to be able to do that we need to fetch further `step_size` information from the Binance API).

- Chapter 8 - Add support for multiple transactions per order

Our `Naive.Trader` implementation assumes that our orders will be filled within a single transaction, but this isn't always the case. In this chapter, we will discuss how could we implement the support for multiple transactions per order and race conditions that could occur between the bot and the Binance API.

- Chapter 9 - Run multiple traders in parallel

With PubSub, supervision tree, buy down and budget in place we can progress with scaling the number of traders. This will require further improvements to our trading strategy like introducing a `rebuy_interval`. At the end of this chapter, our trading strategy will be able to start and run multiple traders in parallel.

- Chapter 10 - Fine-tune trading strategy per symbol

Currently, the naive strategy works based on settings hardcoded in the `leader` module. To allow for fine-tuning the naive trading strategy per symbol we will introduce a new database together with the table that will store trading settings.

- Chapter 11 - Supervise and autostart streaming

In the last chapter, we introduced a new database inside the `naive` application to store default settings, in this chapter we will do the same for the `streamer` application. Inside the settings, there will be a `status` flag that will allow us to implement the autostarting functionality on initialization using Task abstraction.

- Chapter 12 - Start, stop, shutdown, and autostart trading

To follow up after autostarting streaming we will apply the same trick to the trading supervision tree using Task abstraction. We will need to introduce a new supervision level to achieve the correct supervision strategy.

- Chapter 13 - Abstract duplicated supervision code

As both the `naive` and the `streamer` applications contain almost the same copy-pasted code that allows us to start, stop and autostart workers. We will look into how could we abstract the common parts of that implementation into a single module. We will venture into utilizing the `__using__` macro to get rid of the boilerplate.

- Chapter 14 - Store trade events and orders inside the database

To be able to backtest the trading strategy, we need to have historical prices (trade events) and a list of orders that were placed stored in the database, which will be the focus of this chapter. At this moment, the latest prices (trade events) are broadcasted to PubSub topic and traders are subscribing to it. We will create a new application called `data_warehouse` inside our umbrella that will be responsible for subscribing to the same PubSub topics and storing incoming prices (trade events) in the Postgres database. We will update the `Naive.Trader` module to broadcast orders as traders will place them.

Then we will move on to adding supervision similar to the one from the `naive` and the `streamer` applications but this time we will show how we could avoid using both common module and macros by utilizing the `Registry` module.

- Chapter 15 - Backtest trading strategy

In this chapter, we will be backtesting our trading strategy by developing a publisher inside the DataWarehouse application. It will stream trade events from the database to broadcast them to the `TRADE_EVENTS:#{symbol}` PubSub topic. It will use the same topic as data would be streamed directly from the Binance. From the trader's perspective, it won't any difference and will cause normal trading activity that will be stored inside the database to be analyzed later.

- Chapter 16 - End-to-end testing

We've reached the stage where we have a decent solution in place, and to ensure that it's still working correctly after any future refactoring, we will add tests. We will start with the "integration"/"end-to-end"(E2E) test, which will confirm that the whole "trading" works. To perform tests at this level, we will need to orchestrate databases together with processes and broadcast trade events from within the test to cause our trading strategy to place orders. We will be able to confirm the right behaviour by checking the database after running the test.

- Chapter 17 - Mox rocks

In the previous chapter, we've implemented the end-to-end test that required a lot of prep work, and we were able to see the downsides of this type of test clearly. This chapter will focus on implementing a more granular test that will utilize the mox package to mock out the dependencies of the `Naive.Trader`. We will look into how the mox works and how we will need to modify our code to use it.

Contributing, Errata and Source code

The book is written using R Markdown(it's a very similar syntax to the GitHub markdown but supports many more features including code execution, etc.) and converted to final form (for example PDF) using the bookdown app. This means that editing a chapter is as simple as editing the markdown source of that chapter.

There are two repositories related to this book(both hosted on Github):

- source code of the book itself
- code written across the book where the final code of each chapter has its own branch

In regards to contributions - I would love to follow the standard process of forking, making changes, opening PR(please look is there a branch for the next version and point to it instead of `main`), merging, and releasing a new version of the book.

This books has also the **GitHub Discussions** enabled for both the book's repo as well as source code's repo, please feel welcome to start any discussions related to book there.

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

Chapter 2

Create a naive trading strategy - a single trader process without supervision

2.1 Objectives

- create another supervised application inside the umbrella to store our trading strategy
- define callbacks for events dependent on the state of the trader
- push events from the streamer app to the naive app

2.2 Initializiation

To develop our *naive* strategy will need to create a new supervised application inside our umbrella project:

```
cd apps
mix new naive --sup
```

We can now focus on creating a `trader` abstraction inside that newly created application. First we need to create a new file called `trader.ex` inside `apps/naive/lib/naive/`.

Let's start with a skeleton of a GenServer:

```
# /apps/naive/lib/naive/trader.ex
defmodule Naive.Trader do
  use GenServer

  require Logger

  def start_link(args) do
    GenServer.start_link(__MODULE__, args, name: :trader)
```

```
  end

  def init(args) do
    {:ok, args}
  end
end
```

Our module uses the GenServer behavior and to fulfill its "contract", we need to implement the `init/1` function. The `start_link/1` function is a convention and it allows us to register the process with a name(it's a default function that the `Supervisor` will use when starting the Trader). We also add a `require Logger` as we will keep on logging across the module.

Next, let's model the state of our server:

```
  # /apps/naive/lib/naive/trader.ex
  defmodule State do
    @enforce_keys [:symbol, :profit_interval, :tick_size]
    defstruct [
      :symbol,
      :buy_order,
      :sell_order,
      :profit_interval,
      :tick_size
    ]
  end
```

Our trader needs to know:

- what symbol does it need to trade ("symbol" here is a pair of assets for example "XRPUSDT", which is XRP to/from USDT)
- placed buy order (if any)
- placed sell order (if any)
- profit interval (what net profit % we would like to achieve when buying and selling an asset - single trade cycle)
- tick_size (yes, I know, jargon warning. We can't ignore it here as it needs to be fetched from Binance and it's used to calculate a valid price. Tick size differs between symbols and it is the smallest acceptable price movement up or down. For example in the physical world tick size for USD is a single cent, you can't sell something for $1.234, it's either $1.23 or $1.24 (a single cent difference between those is the tick size) - more info here.

Our strategy won't be able to work without symbol, profit_interval nor tick_size so we added them to the `@enforce_keys` attribute. This will ensure that we won't create an invalid `%State{}` without those values.

As now we know that our GenServer will need to receive those details via args, we can update pattern matching in `start_link/1` and `init/1` functions to confirm that passed values are indeed maps:

```
# /apps/naive/lib/naive/trader.ex
def start_link(%{} = args) do
  ...
end

def init(%{symbol: symbol, profit_interval: profit_interval}) do
  ...
end
```

As we are already in the `init/1` function we will need to modify it to fetch the `tick_size` for the passed symbol and initialize a fresh state:

```
# /apps/naive/lib/naive/trader.ex
def init(%{symbol: symbol, profit_interval: profit_interval}) do
  symbol = String.upcase(symbol)

  Logger.info("Initializing new trader for #{symbol}")

  tick_size = fetch_tick_size(symbol)

  {:ok,
   %State{
     symbol: symbol,
     profit_interval: profit_interval,
     tick_size: tick_size
   }}
end
```

We are uppercasing the symbol above as Binance's REST API accepts only uppercased symbols.

It's time to connect to Binance's REST API. The easiest way to do that will be to use the binance module.

As previously, looking through the module's docs on Github, we can see the `Installation` section. We will follow the steps mentioned there, starting from adding `binance` to the deps in `/apps/naive/mix.exs`:

```
# /apps/naive/mix.ex
defp deps do
  [
    {:binance, "~> 1.0"},
    {:decimal, "~> 2.0"},
    {:streamer, in_umbrella: true}
  ]
end
```

Besides adding the `:binance` module, we also added `:decimal` and the `:streamer`. The decimal module will help us to calculate the buy and sell prices (without the decimal module we would have problems with precision). Lastly, we need to include the `:streamer` application(created in the first chapter) as we will use the `%Streamer.Binance.TradeEvent{}` struct inside the naive application.

We need to run `mix deps.get` to install our new deps.

We can now get back to the `trader` module and focus on fetching the tick size from Binance:

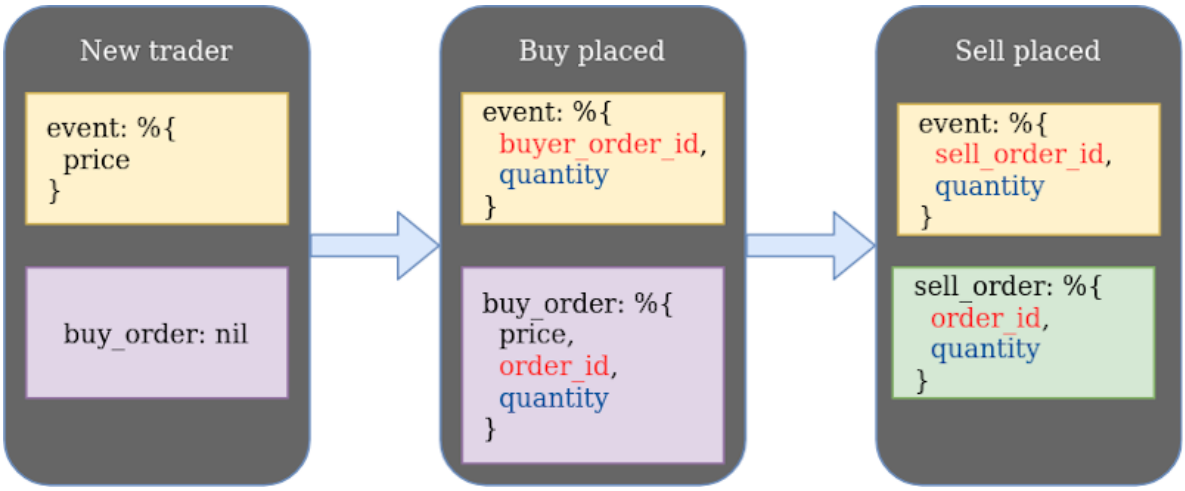
```
# /apps/naive/lib/naive/trader.ex
defp fetch_tick_size(symbol) do
  Binance.get_exchange_info()
  |> elem(1)
  |> Map.get(:symbols)
  |> Enum.find(&(&1["symbol"] == symbol))
  |> Map.get("filters")
  |> Enum.find(&(&1["filterType"] == "PRICE_FILTER"))
  |> Map.get("tickSize")
end
```

We are using `get_exchange_info/0` to fetch the list of symbols, which we will filter to find the symbol that we are requested to trade. Tick size is defined as a `PRICE_FILTER` filter. Here's the link to the documentation listing all keys in the result. In a nutshell, that's how the important parts of the result look like:

```
{:ok, %{
  ...
  symbols: [
    %{
      "symbol": "ETHUSDT",
      ...
      "filters": [
        ...
        %{"filterType: "PRICE_FILTER", "tickSize": tickSize, ...}
      ],
      ...
    }
  ]
}}
```

2.3 How trading strategy will work?

Our trader process has an internal state that will serve as an indicator of its step in the trade cycle. The following diagram shows 3 possible trader states that trader will progress through from left to right:



Our trader will be receiving trade events sequentially and take decisions based on its own state and trade event's contents.

We will focus on a trader in 3 different states:

- a new trader without any orders
- a trader with a buy order placed
- a trader with a sell order placed.

First state - A new trader

The trader doesn't have any open orders which we can confirm by pattern matching on the `buy_order` field from its state. From the incoming event, we can get the current price which we will use in the buy order that the trader will place.

Second state - Buy order placed

After placing a buy order, the trader will be pattern matching to confirm that has incoming event filled his buy order otherwise ignoring it. When a trade event matching the order id of the trader's buy order will arrive, it means that the buy order got filled(simplification - our order could be filled in two or more transactions but implementation in this chapter won't cater for this case, it will always assume that it got filled in a single transaction) and the trader can now place the sell order based on the expected profit and the buy_price.

Third state - Sell order placed

Finally, in a very similar fashion to the previous state, the trader will be pattern matching to confirm that the incoming event has filled his sell order(matching order id), otherwise ignore it. When a trade event matching the order id of trader's sell order will arrive, which means that the sell order got filled(simplification as described above) and the full trade cycle has ended and the trader can now exit.

2.3.1 Implementation of the first scenario

Enough theory :) back to the editor, we will implement the first scenario. Before doing that let's alias Streamer's TradeEvent struct as we will rely on it heavily in pattern matching.

```
# /apps/naive/lib/naive/trader.ex
alias Streamer.Binance.TradeEvent
```

We are also aliasing the `%Streamer.Binance.TradeEvent{}` struct as we will rely on it heavily in pattern matching.

To confirm that we are dealing with a "new" trader, we will pattern match on `buy_order: nil` from its state:

```
# /apps/naive/lib/naive/trader.ex
def handle_cast(
      %TradeEvent{price: price},
      %State{symbol: symbol, buy_order: nil} = state
    ) do
  quantity = "100" # <= Hardcoded until chapter 7

  Logger.info("Placing BUY order for #{symbol} @ #{price}, quantity: #{quantity}")

  {:ok, %Binance.OrderResponse{} = order} =
    Binance.order_limit_buy(symbol, quantity, price, "GTC")

  {:noreply, %{state | buy_order: order}}
end
```

For the time being, we will keep the quantity hardcoded as this chapter will get really long otherwise - don't worry, we will refactor this in one of the next chapters.

After confirming that we deal with the "new" trader(by pattern matching on the `buy_order` field from the state), we can safely progress to placing a new buy order. We just need to remember to return the updated state as otherwise, the trader will go on a shopping spree, as every next incoming event will cause further buy orders(the above pattern match will continue to be successful).

2.3.2 Implementation of the second scenario

With that out of the way, we can now move on to monitoring for an event that matches our buy order id and quantity to confirm that our buy order got filled:

```
# /apps/naive/lib/naive/trader.ex
def handle_cast(
      %TradeEvent{
        buyer_order_id: order_id,
        quantity: quantity
```

```
        },
        %State{
          symbol: symbol,
          buy_order: %Binance.OrderResponse{
            price: buy_price,
            order_id: order_id,
            orig_qty: quantity
          },
          profit_interval: profit_interval,
          tick_size: tick_size
        } = state
      ) do
    sell_price = calculate_sell_price(buy_price, profit_interval, tick_size)

    Logger.info(
      "Buy order filled, placing SELL order for " <>
        "#{symbol} @ #{sell_price}), quantity: #{quantity}"
    )

    {:ok, %Binance.OrderResponse{} = order} =
      Binance.order_limit_sell(symbol, quantity, sell_price, "GTC")

    {:noreply, %{state | sell_order: order}}
  end
```

We will implement calculating sell price in a separate function based on buy price, profit interval, and tick_size.

Our pattern match will confirm that indeed our buy order got filled(order_id and quantity matches) so we can now proceed with placing a sell order using calculated sell price and quantity retrieved from the buy order. Again, don't forget to return the updated state as otherwise, the trader will keep on placing sell orders for every incoming event.

To calculate the sell price we will need to use precise math and that will require a custom module. We will use the Decimal module, so first, let's alias it at the top of the file:

```
# /apps/naive/lib/naive/trader.ex
alias Decimal, as: D
```

Now to calculate the correct sell price, we can use the following formula which gets me pretty close to expected value:

```
  # /apps/naive/lib/naive/trader.ex
  defp calculate_sell_price(buy_price, profit_interval, tick_size) do
    fee = "1.001"

    original_price = D.mult(buy_price, fee)
```

```
    net_target_price =
      D.mult(
        original_price,
        D.add("1.0", profit_interval)
      )

    gross_target_price = D.mult(net_target_price, fee)

    D.to_string(
      D.mult(
        D.div_int(gross_target_price, tick_size),
        tick_size
      ),
      :normal
    )
  end
```

First, we will hardcode the fee to 0.1% which we will refactor in one of the future chapters.

We started by calculating the `original_price` which is the buy price together with the fee that we paid on top of it.

Next, we enlarge the originally paid price by profit interval to get `net_target_price`.

As we will be charged a fee for selling, we need to add the fee again on top of the net target sell price(we will call this amount the `gross_target_price`).

Next, we will use the tick size as Binance won't accept any prices that aren't divisible by the symbols' tick sizes so we need to "normalize" them on our side.

2.3.3 Implementation of the third scenario

Getting back to handling incoming events, we can now add a clause for a trader that wants to confirm that his sell order was filled:

```
# /apps/naive/lib/naive/trader.ex
def handle_cast(
      %TradeEvent{
        seller_order_id: order_id,
        quantity: quantity
      },
      %State{
        sell_order: %Binance.OrderResponse{
          order_id: order_id,
          orig_qty: quantity
        }
      } = state
```

```
      ) do
    Logger.info("Trade finished, trader will now exit")
    {:stop, :normal, state}
  end
```

When the sell order was successfully filled(confirmed by pattern matching above), there's nothing else to do for the trader, so it can return a tuple with `:stop` atom which will cause the trader process to terminate.

2.3.4 Implementation fallback scenario

A final callback function that we will need to implement will just ignore all incoming events as they were not matched by any of the previous pattern matches:

```
  # /apps/naive/lib/naive/trader.ex
  def handle_cast(%TradeEvent{}, state) do
    {:noreply, state}
  end
```

We need this callback for cases where our trader has an "open" order(not yet filled) and the incoming event has nothing to do with it, so it needs to be ignored.

2.3.5 Updating the Naive interface

Now we will update an interface of our `naive` application by modifying the Naive module to allow to send an event to the trader:

```
# /apps/naive/lib/naive.ex
defmodule Naive do
  @moduledoc """
  Documentation for `Naive`.
  """
  alias Streamer.Binance.TradeEvent

  def send_event(%TradeEvent{} = event) do
    GenServer.cast(:trader, event)
  end
end
```

We will use the fact that we registered our trader process with a name to be able to cast a message to it.

2.3.6 Updating streamer app

To glue our apps together for the time and keep things simple in this chapter we will modify the streamer process to simply call our new `Naive` interface directly by appending the following function call at the end of the `process_event/1` function inside the `Streamer.Binance` module:

```
  # /apps/streamer/lib/streamer/binance.ex
  defp process_event(%{"e" => "trade"} = event) do
    ...
    Naive.send_event(trade_event)
  end
```

This creates a two-way link between the streamer and the naive app. In the next chapter, we will fix that as in the perfect world those apps shouldn't even be aware of existence of each other.

2.3.7 Access details to Binance

Inside the config of our umbrella project we create a new file `config/secrets.exs`. We will use this for our Binance account access details.

```
# /config/secrets.exs
import Config

config :binance,
  api_key: "YOUR-API-KEY-HERE",
  secret_key: "YOUR-SECRET-KEY-HERE"
```

We don't want to check this file in, so we add it to our `.gitignore`:

```
# .gitignore
config/secrets.exs
```

Finally, we update our main config file to include it using import_config:

```
# /config/config.exs

# Import secrets file with Binance keys if it exists
if File.exists?('config/secrets.exs') do
  import_config('secrets.exs')
end
```

Important note: To be able to run the below test and perform real trades, a Binance account is required with a balance of at least 20 USDT. In the 4th chapter, we will focus on creating a `BinanceMock` that will allow us to run our bot *without* the requirement for a real Binance account. You don't need to test run it now if you don't need/want to have an account.

2.3.8 Test run

Now it's time to give our implementation a run for its money. Once again, to be able to do that you will need to have at least 20 USDT tokens in your Binance's wallet and you will lose just under 0.5% of your USDTs(as "expected profit" is below 0 to quickly showcase the full trade cycle) in the following test:

```
$ iex -S mix
...
iex(1)> Naive.Trader.start_link(%{symbol: "XRPUSDT", profit_interval: "-0.01"})
13:45:30.648 [info] Initializing new trader for XRPUSDT
{:ok, #PID<0.355.0>}
iex(2)> Streamer.start_streaming("xrpusdt")
{:ok, #PID<0.372.0>}
iex(3)>
13:45:32.561 [info] Placing BUY order for XRPUSDT @ 0.25979000, quantity: 100
13:45:32.831 [info] Buy order filled, placing SELL order for XRPUSDT @ 0.2577, quantity: 100
13:45:33.094 [info] Trade finished, trader will now exit
```

After starting the IEx session, start the trader process with a map containing the symbol and profit interval. To be able to quickly test the full trade cycle we will pass a sub-zero profit interval instead of waiting for the price increase.

Next, we will start streaming on the same symbol, please be aware that this will cause an immediate reaction in the trader process.

We can see that our trader placed a buy order at 25.979c per XRP, it was filled in under 300ms, so then the trader placed a sell order at ~25.77c which was also filled in under 300ms. This way the trader finished the trade cycle and the process can terminate.

That's it. **Congratulations!** You just made your first algorithmic trade and you should be proud of that! In the process of creating that algorithm, we touched on multiple topics including GenServer and depending on its state and external data (trade events) to perform different actions - this is a very common workflow that Elixir engineers are following and it's great to see it in action.

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

Source code for this chapter can be found at Github

Chapter 3

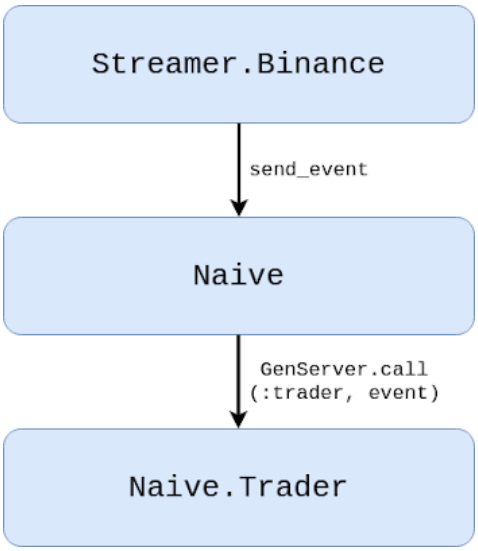
Introduce PubSub as a communication method

3.1 Objectives

- consider reasons why introducing a PubSub communication would be beneficial
- implement the PubSub communication between the `Streamer.Binance` and the `Naive.Trader`(s)

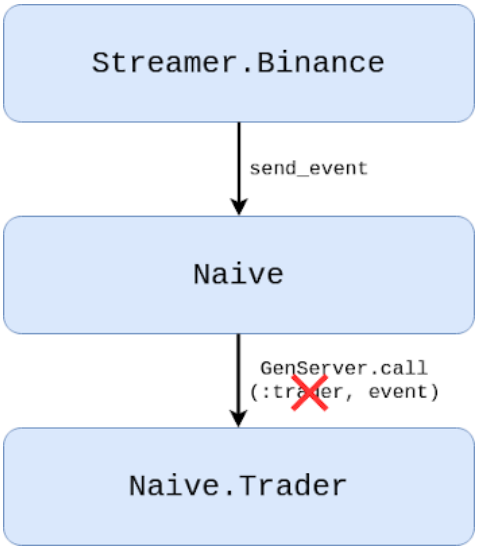
3.2 Design

First, let's look at the current situation:



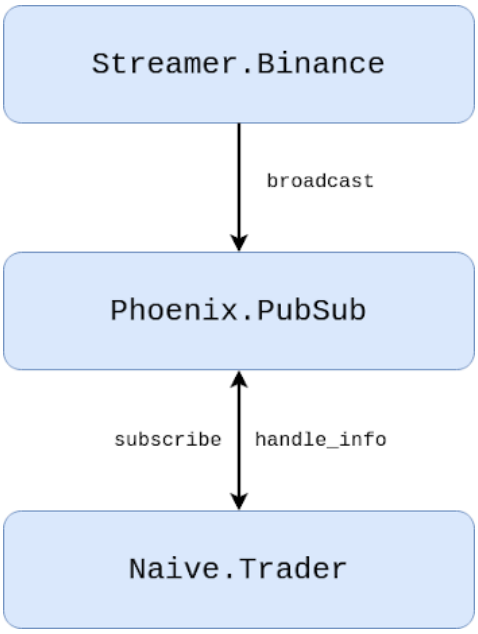
We started with the Binance streamer calling the `send_event/1` function on the `Naive` module. The `Naive` module then calls the trader process using the GenServer's `cast/2` function(via its registered name).

The next step in the process of extending our trading strategy will be to scale it to run multiple `Naive.Trader` processes in parallel. To be able to do that we will need to remove the option to register the `trader` process with a name(as only one process can be registered under a single name).



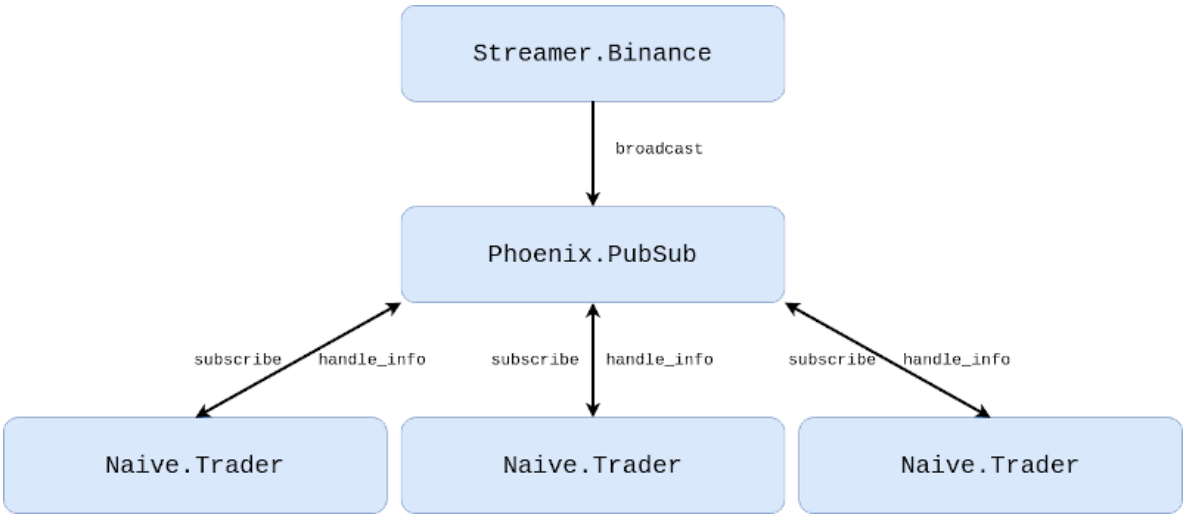
The second issue with that design was the fact that the `Streamer` needs to be aware of all processes that are interested in the streamed data and explicitly push that information to them.

To fix those issues we will invert the design and introduce a PubSub mechanism:

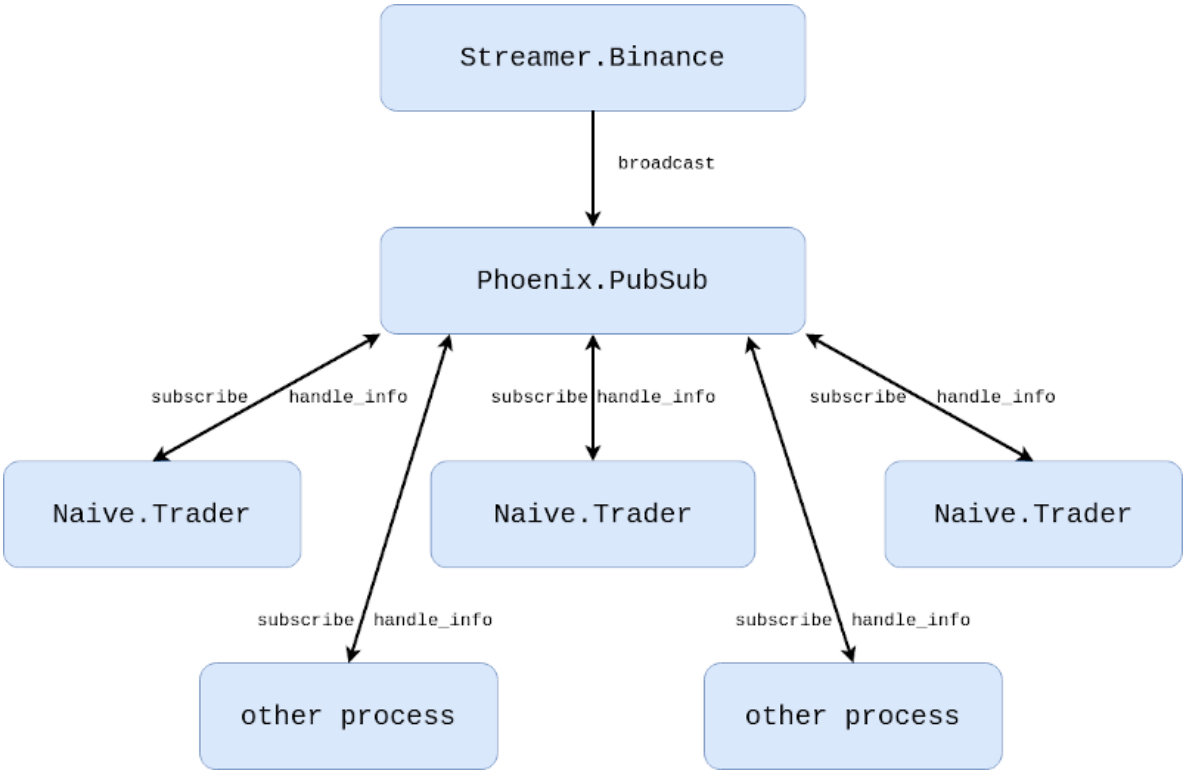


The streamer will broadcast trade events to the PubSub topic and whatever is interested in that data, can subscribe to the topic and it will receive the broadcasted messages. There's no coupling between the `Streamer` and `Naive` app anymore.

We can now introduce multiple traders that will subscribe to the topic and they will receive messages from the PubSub:



Going even further down the line we can picture that system could consist of other processes interested in the streamed data. An example of those could be a process that will save all streamed information to the database to be utilized in backtesting later on:



3.3 Implementation

We will start by adding a `Phoenix.PubSub` library to both `Streamer` and `Naive` app(as both will be using it, `Streamer` app as a broadcaster and `Naive` app as a subscriber).

Scrolling down through its readme on GitHub we can see that we need to add `:phoenix_pubsub` to list of dependencies:

```
# /apps/streamer/mix.exs & /apps/naive/mix.exs
  defp deps do
    [
      ...
      {:phoenix_pubsub, "~> 2.0"},
      ...
    ]
  end
```

Remember to place it so the list will keep alphabetical order. The second step in the readme says that we need to add PubSub as a child of our app. We need to decide where we will put it, `Streamer` sounds like a good starting point. We will modify the `/apps/streamer/lib/streamer/application.ex` module by appending the PubSub to it:

```
# /apps/streamer/lib/streamer/application.ex
  def start(_type, _args) do
    children = [
      {
        Phoenix.PubSub,
        name: Streamer.PubSub, adapter_name: Phoenix.PubSub.PG2
      }
    ]
    ...
  end
```

We will add the `:adapter_name` option to instruct PubSub to use `pg` adapter, which will give us distributed process groups.

We will now modify the streamer to broadcast a message to PubSub topic instead of using the `Naive` module's function:

```
# /apps/streamer/lib/streamer/binance.ex
  defp process_event(...) do
    ...
    Phoenix.PubSub.broadcast(
      Streamer.PubSub,
      "TRADE_EVENTS:#{trade_event.symbol}",
      trade_event
    )
  end
```

Inside the trader on init we need to subscribe to the "TRADE_EVENTS" PubSub channel:

```
# /apps/naive/lib/naive/trader.ex
  def init(...) do
    ...
    Phoenix.PubSub.subscribe(
      Streamer.PubSub,
      "TRADE_EVENTS:#{symbol}"
    )
    ...
  end
```

Next, we need to convert all `handle_cast` callbacks to `handle_info` inside our `Trader` module as PubSub doesn't use `GenServer.cast/2` to send messages over to subscribers.

The final change will be to remove the `send_event` function from the `Naive` module as it's no longer required.

Our update is now finished so we can start an iex session to see how it works.

First, we will start a streamer process that will broadcast messages to PubSub. Next, we will start trading on the same symbol. On init, the trader will subscribe to a PubSub channel and it will make a full trade cycle.

```
$ iex -S mix
...
iex(1)> Streamer.start_streaming("xrpusdt")
{:ok, #PID<0.483.0>}
iex(2)> Naive.Trader.start_link(%{symbol: "XRPUSDT", profit_interval: "-0.01"})
23:46:37.482 [info]  Initializing new trader for XRPUSDT
{:ok, #PID<0.474.0>}
23:46:55.179 [info]  Placing BUY order for XRPUSDT @ 0.29462000, quantity: 100
23:46:55.783 [info]  Buy order filled, placing SELL order for XRPUSDT @ 0.29225),
quantity: 100.00000000
23:46:56.029 [info]  Trade finished, trader will now exit
```

This shows that the new trader process successfully subscribed to the PubSub, received the broadcasted messages, placed buy/sell orders, and terminated after the full trade cycle finished.

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

Source code for this chapter can be found at Github

Chapter 4

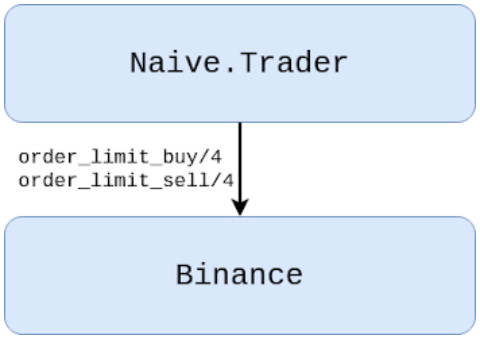
Mock the Binance API

4.1 Objectives

- design the binance mock application
- create a new app
- implement getting exchange info
- implement placing buy and sell orders
- implement callback for incoming trade events
- upgrade trader and config
- test the implementation

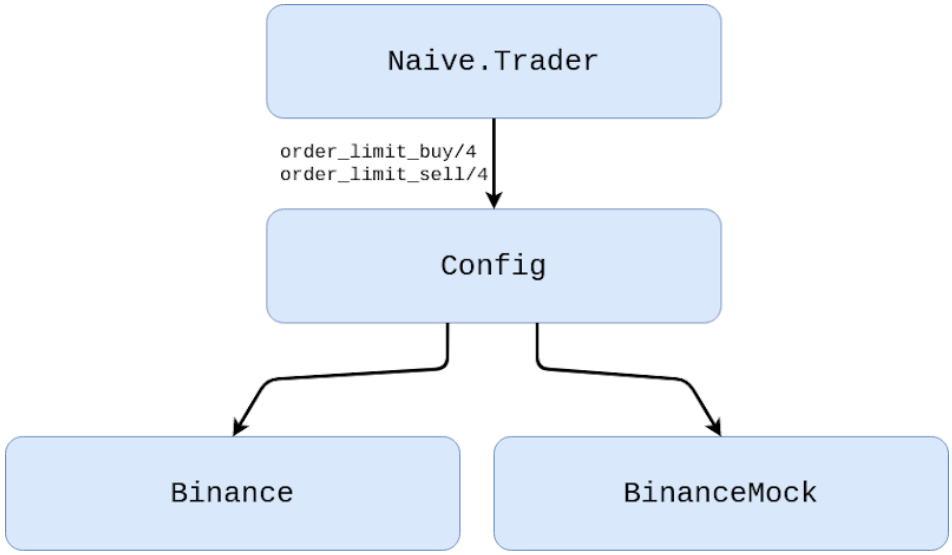
4.2 Design

First, let's start with the current state:

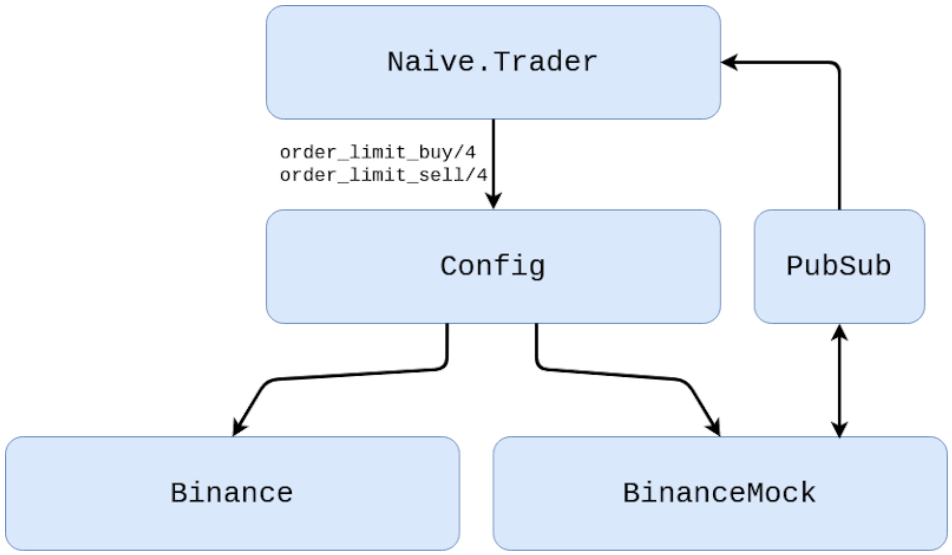


Currently, our trader is using the `Binance` module to place buy/sell orders and get exchange info. The `get_exchange_info/0` function doesn't require a Binance account as it's publicly available information so we can call the `Binance` lib directly from our module. The remaining ones(buying/selling) require a `Binance` account and some coins/tokens inside its wallet. We need to mock those inside our module.

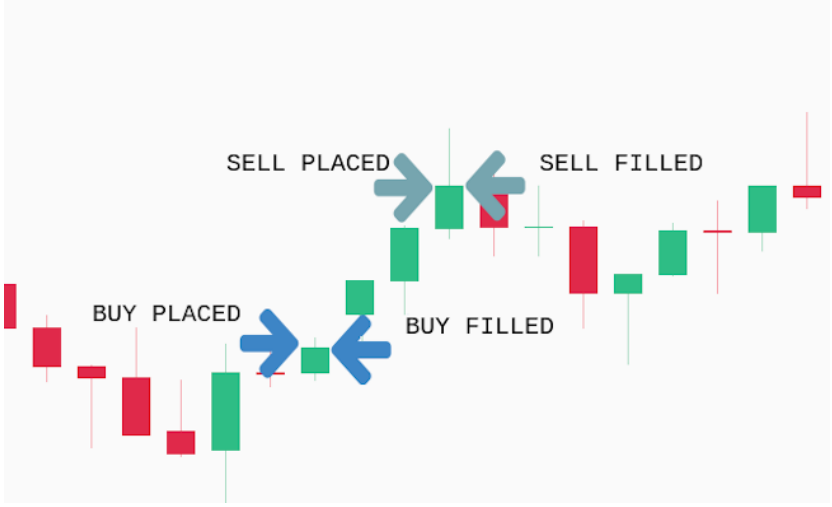
We will update the trader to fetch the Binance's module name from the config:



We will set up a config so it points to the Binance client to be used - either Binance or BinanceMock. Regards the BinanceMock itself it will have the same interface as the Binance module. It will need to store both buy and sell orders and it will allow us to retrieve them. That will cover the REST functions but Binance also streams back trade events for those orders as they get filled, that's why BinanceMock will also need to broadcast fake events to the "TRADE_EVENTS:#{symbol}" PubSub topic so the trader will pick them up:



When exactly should we broadcast those fake trade events? Well, the best thing that we can do is make the `BinanceMock` process subscribe to the trade events stream and try to broadcast fake trade events whenever the price of orders would be matched:



Starting from the arrow on the left, our naive strategy will place an order at the current price. In this hypotetical scenario, the price raised for a moment after placing the buy order, so BinanceMock will keep on waiting until a trade event will get broadcasted from the PubSub with a price *below* the buy order's price. At that moment BinanceMock will generate a fake trade event and broadcast it to the same PubSub topic. The trader will get that event and assume that it came from the Binance and that the buy order got filled so it will place a sell order. Similar to the buy order, BinanceMock will keep on waiting until a trade event will get broadcasted from the PubSub with the price *above* the sell order's price. At that moment BinanceMock will generate a fake trade event and broadcast it to the same PubSub topic.

Enough theory for now, let's get our hands dirty with some coding.

4.3 Create "BinanceMock" app

We will start by creating a new supervised app called `BinanceMock`:

```
$ cd apps
$ mix new binance_mock --sup
```

The next step will be to update the `BinanceMock` module to be a GenServer.

We will utilize:

- the `Decimal` module for comparing the prices
- the `Logger` module to log

As well as we will define internal `%State{}` struct that will hold:

- map called `order_books` for each traded symbol
- list of symbols that mock subscribed to
- last generated id - for consistent generating of unique ids for fake trade events

`order_books` map will consist of `:"#{symbol}` => `%OrderBook{}`. We will define the `%OrderBook{}` struct as 3 lists `buy_side`, `sell_side` and `historical`:

```
# /apps/binance_mock/lib/binance_mock.ex
defmodule BinanceMock do
  use GenServer

  alias Decimal, as: D

  require Logger

  defmodule State do
    defstruct order_books: %{}, subscriptions: [], fake_order_id: 1
  end

  defmodule OrderBook do
    defstruct buy_side: [], sell_side: [], historical: []
  end

  def start_link(_args) do
    GenServer.start_link(__MODULE__, nil, name: __MODULE__)
  end

  def init(_args) do
    {:ok, %State{}}
  end
end
```

4.4 Implement getting exchange info

As it was mentioned before, to retrieve exchange info we can just call Binance's function directly as its publicly available information:

```
# /apps/binance_mock/lib/binance_mock.ex
  def get_exchange_info do
    Binance.get_exchange_info()
  end
```

4.5 Implement placing buy and sell orders

For buy and sell limit orders we will write a helper function as the logic is the same for both order sides:

```
# /apps/binance_mock/lib/binance_mock.ex
  def order_limit_buy(symbol, quantity, price, "GTC") do
    order_limit(symbol, quantity, price, "BUY")
  end

  def order_limit_sell(symbol, quantity, price, "GTC") do
    order_limit(symbol, quantity, price, "SELL")
  end
```

The "order_limit" helper function will:

- generate a fake order based on symbol, quantity, price, and side
- cast a message to the BinanceMock process to add the fake order
- return with a tuple with `%OrderResponse{}` struct to be consistent with the Binance module:

```
# /apps/binance_mock/lib/binance_mock.ex
  defp order_limit(symbol, quantity, price, side) do
    %Binance.Order{} =
      fake_order =
      generate_fake_order(
        symbol,
        quantity,
        price,
        side
      )

    GenServer.cast(
      __MODULE__,
      {:add_order, fake_order}
    )

    {:ok, convert_order_to_order_response(fake_order)}
  end
```

We can now move on to implementing the `handle_cast/2` callback to `:add_order` to the order book for the symbol from the order. It needs to subscribe to the `TRADE_EVENTS:#{symbol}` topic for the symbol from the order and add the order to the correct order book:

```
# /apps/binance_mock/lib/binance_mock.ex
  def handle_cast(
        {:add_order, %Binance.Order{symbol: symbol} = order},
        %State{
          order_books: order_books,
          subscriptions: subscriptions
        } = state
      ) do
    new_subscriptions = subscribe_to_topic(symbol, subscriptions)
    updated_order_books = add_order(order, order_books)

    {
      :noreply,
      %{
        state
        | order_books: updated_order_books,
          subscriptions: new_subscriptions
      }
    }
  end
```

We will start with the implementation of the `subscribe_to_topic/2` function. We need to make sure that the symbol is uppercased as well as check have we already subscribed to that topic. Otherwise, we can safely use the PubSub module to subscribe to the `TRADE_EVENTS:#{symbol}` topic for this symbol. We need to remember to append the symbol to the list of subscription and return the updated list:

```
# /apps/binance_mock/lib/binance_mock.ex
  defp subscribe_to_topic(symbol, subscriptions) do
    symbol = String.upcase(symbol)
    stream_name = "TRADE_EVENTS:#{symbol}"

    case Enum.member?(subscriptions, symbol) do
      false ->
        Logger.debug("BinanceMock subscribing to #{stream_name}")

        Phoenix.PubSub.subscribe(
          Streamer.PubSub,
          stream_name
        )
```

```
        [symbol | subscriptions]

      _ ->
        subscriptions
    end
  end
```

Next, time for implementation of the `add_order` function. First, we need to get the order book for the symbol of the order. Depends on the side of the order we will update either the `buy_side` or `sell_side` list remembering that both sides are sorted. We are sorting them so we can easily grab all orders that should be filled whenever trade event arrived, this will become clearer as we will write a handle callback for incoming trade events:

```
# /apps/binance_mock/lib/binance_mock.ex
  defp add_order(
         %Binance.Order{symbol: symbol} = order,
         order_books
       ) do
    order_book =
      Map.get(
        order_books,
        :"#{symbol}",
        %OrderBook{}
      )

    order_book =
      if order.side == "SELL" do
        Map.replace!(
          order_book,
          :sell_side,
          [order | order_book.sell_side]
          |> Enum.sort(&D.lt?(&1.price, &2.price))
        )
      else
        Map.replace!(
          order_book,
          :buy_side,
          [order | order_book.buy_side]
          |> Enum.sort(&D.gt?(&1.price, &2.price))
        )
      end

    Map.put(order_books, :"#{symbol}", order_book)
  end
```

Now we need to follow up and implement the functions that we referred to previously - those are `generate_fake_order` and `convert_order_to_order_response`.

Starting with the `generate_fake_orders`, it's a function that takes a symbol, quantity, price, and side and based on those values returns a `Binance.Order` struct. To return the struct we will need to generate a unique id for each faked order - this is where `fake_order_id` will be used(callback implemented later). This way we will be able to run tests multiple times using the BinanceMock and always get the same ids:

```
# /apps/binance_mock/lib/binance_mock.ex
  defp generate_fake_order(symbol, quantity, price, side)
       when is_binary(symbol) and
              is_binary(quantity) and
              is_binary(price) and
              (side == "BUY" or side == "SELL") do
    current_timestamp = :os.system_time(:millisecond)
    order_id = GenServer.call(__MODULE__, :generate_id)
    client_order_id = :crypto.hash(:md5, "#{order_id}") |> Base.encode16()

    Binance.Order.new(%{
      symbol: symbol,
      order_id: order_id,
      client_order_id: client_order_id,
      price: price,
      orig_qty: quantity,
      executed_qty: "0.00000000",
      cummulative_quote_qty: "0.00000000",
      status: "NEW",
      time_in_force: "GTC",
      type: "LIMIT",
      side: side,
      stop_price: "0.00000000",
      iceberg_qty: "0.00000000",
      time: current_timestamp,
      update_time: current_timestamp,
      is_working: true
    })
  end
```

We can now focus on converting the `Binance.Order` to the `Binance.OrderResponse` struct. As `Binance.Order` struct contains almost all of the same fields that the `Binance.OrderResponse` struct, we can use `struct` function without exclamation mark to ignore all additional fields. The only field that has a different name is `transact_time` field which is called `time` in the `Binance.Order` struct - we can fix that separately:

```
# /apps/binance_mock/lib/binance_mock.ex
  defp convert_order_to_order_response(%Binance.Order{} = order) do
    %{
      struct(
        Binance.OrderResponse,
        order |> Map.to_list()
      )
      | transact_time: order.time
    }
  end
```

The last function to finish support for placing buy and sell orders is to add a callback that will iterate the fake order id and return it:

```
# /apps/binance_mock/lib/binance_mock.ex
  def handle_call(
        :generate_id,
        _from,
        %State{fake_order_id: id} = state
      ) do
    {:reply, id + 1, %{state | fake_order_id: id + 1}}
  end
```

4.6 Implement order retrival

We can now move on to retrieving the orders. First, we need to add an interface function that will call our BinanceMock GenServer:

```
# /apps/binance_mock/lib/binance_mock.ex
  def get_order(symbol, time, order_id) do
    GenServer.call(
      __MODULE__,
      {:get_order, symbol, time, order_id}
    )
  end
```

The callback itself is pretty straightforward. We will need to get an order book for the passed symbol. As we don't know the order's side, we will concat all 3 lists(buy_side, sell_side, and historical) and try to find an order that will match passed symbol, time, and order_id:

```
# /apps/binance_mock/lib/binance_mock.ex
  def handle_call(
        {:get_order, symbol, time, order_id},
        _from,
        %State{order_books: order_books} = state
      ) do
    order_book =
      Map.get(
        order_books,
        :"#{symbol}",
        %OrderBook{}
      )

    result =
      (order_book.buy_side ++
         order_book.sell_side ++
         order_book.historical)
      |> Enum.find(
        &(&1.symbol == symbol and
            &1.time == time and
            &1.order_id == order_id)
      )

    {:reply, {:ok, result}, state}
  end
```

4.7 Implement callback for incoming trade events

Finally, we need to handle incoming trade events(streamed from the PubSub topic). We need to implement a callback that will:

- get the order book for the symbol from the trade event
- use the `take_while/2` function on the buy orders with prices that are *greater* than the current price - we can update their status to filled.
- use the `take_while/2` function again, this time to sell orders with prices *less* than the current price, we will also update their statuses to filled.
- concat both lists of filled orders, convert them to trade events, and broadcast them to the PubSub's TRADE_EVENTS topic.
- remove the filled orders from buy and sell lists and put them into the historical list.

Here we can clearly see the benefit of sorting the lists, we can use functions like `take_while/2` and `drop/2` instead of `filter/2` and `reject/2`(later ones will go through whole lists which could become a bottleneck when multiple open orders would be active):

```
# /apps/binance_mock/lib/binance_mock.ex
  def handle_info(
        %Streamer.Binance.TradeEvent{} = trade_event,
        %{order_books: order_books} = state
      ) do
    order_book =
      Map.get(
        order_books,
        :"#{trade_event.symbol}",
        %OrderBook{}
      )

    filled_buy_orders =
      order_book.buy_side
      |> Enum.take_while(&D.lt?(trade_event.price, &1.price))
      |> Enum.map(&Map.replace!(&1, :status, "FILLED"))

    filled_sell_orders =
      order_book.sell_side
      |> Enum.take_while(&D.gt?(trade_event.price, &1.price))
      |> Enum.map(&Map.replace!(&1, :status, "FILLED"))

    (filled_buy_orders ++ filled_sell_orders)
    |> Enum.map(&convert_order_to_event(&1, trade_event.event_time))
    |> Enum.each(&broadcast_trade_event/1)

    remaining_buy_orders =
      order_book.buy_side
      |> Enum.drop(length(filled_buy_orders))

    remaining_sell_orders =
      order_book.sell_side
      |> Enum.drop(length(filled_sell_orders))

    order_books =
      Map.replace!(
        order_books,
        :"#{trade_event.symbol}",
        %{
          buy_side: remaining_buy_orders,
```

```
          sell_side: remaining_sell_orders,
          historical:
            filled_buy_orders ++
              filled_sell_orders ++
              order_book.historical
        }
      )

    {:noreply, %{state | order_books: order_books}}
  end
```

Inside the callback we referred to two new functions that we will implement now(`convert_order_to_event` and `broadcast_trade_event`).

Starting with the `convert_order_to_event` function, it will simply return a new `Streamer.Binance.TradeEvent` struct filled with data. An interesting thing to observe here is that again all values are predictable and function will return the same values for the same input - this will become beneficial for backtesting over and over again and comparing the behavior between runs:

```
# /apps/binance_mock/lib/binance_mock.ex
  defp convert_order_to_event(%Binance.Order{} = order, time) do
    %Streamer.Binance.TradeEvent{
      event_type: order.type,
      event_time: time - 1,
      symbol: order.symbol,
      trade_id: Integer.floor_div(time, 1000),
      price: order.price,
      quantity: order.orig_qty,
      buyer_order_id: order.order_id,
      seller_order_id: order.order_id,
      trade_time: time - 1,
      buyer_market_maker: false
    }
  end
```

Broadcasting trade events to PubSub will be the last function that will finish the implementation of `BinanceMock` for now. It's safe to assume that the incoming symbol will be uppercased as it comes from the exchange (the symbol is part of the topic name which is case-sensitive):

```
# /apps/binance_mock/lib/binance_mock.ex
  defp broadcast_trade_event(%Streamer.Binance.TradeEvent{} = trade_event) do
    Phoenix.PubSub.broadcast(
      Streamer.PubSub,
      "TRADE_EVENTS:#{trade_event.symbol}",
```

```
      trade_event
    )
  end
```

That finishes the `BinanceMock` implementation. Now, we need to add it to the children list of the application so it starts automatically:

```
# /apps/binance_mock/lib/binance_mock/application.ex
...
  def start(_type, _args) do
    children = [
      {BinanceMock, []}
    ]
    ...
  end
end
```

4.8 Upgrade trader and config

We can move on to the `Naive.Trader` module where we will add an attribute that will point to the Binance client dictated by config:

```
  # /apps/naive/lib/naive/trader.ex
  @binance_client Application.compile_env(:naive, :binance_client)
```

We need to replace all direct calls to the `Binance` module for calls to the `@binance_client` attribute inside the `Naive.Trader`:

```
# /apps/naive/lib/naive/trader.ex


...
  @binance_client.order_limit_buy(
...
  @binance_client.order_limit_sell
...
  @binance_client.get_exchange_info()
...
```

As the `Naive.Trader` is now relying on the config to specify which Binance client should they use, we need to add it to the config:

```
# /config/config.exs

config :naive,
  binance_client: BinanceMock
```

The last modification to our system will be to modify the `mix.exs` of the `binance_mock` app to list all deps required for it to work:

```
# /apps/binance_mock/mix.exs
...
  defp deps do
    [
      {:binance, "~> 1.0"},
      {:decimal, "~> 2.0"},
      {:phoenix_pubsub, "~> 2.0"},
      {:streamer, in_umbrella: true}
    ]
  end
...
```

We also add `:binance_mock` to the list of deps of the `naive` app(as the Naive app will use either `Binance` or `BinanceMock` to "trade"):

```
# /apps/naive/mix.exs
...
  defp deps do
    [
      ...
      {:binance_mock, in_umbrella: true}
      ...
    ]
  end
...
```

4.9 Test the implementation

We can now see the BinanceMock in action. First, we will start an iex session and double-check that the BinanceMock process is alive.

```
$ iex -S mix
...
iex(1)> Process.whereis(BinanceMock)
#PID<0.320.0> # <- confirms that BinanceMock process is alive
iex(2)> Streamer.start_streaming("xrpusdt")
{:ok, #PID<0.332.0>}
iex(3)> Naive.Trader.start_link(
%{symbol: "XRPUSDT", profit_interval: "-0.001"}
)
00:19:39.232 [info]  Initializing new trader for XRPUSDT
{:ok, #PID<0.318.0>}
00:19:40.826 [info]  Placing BUY order for XRPUSDT @ 0.29520000, quantity: 100
00:19:44.569 [info]  Buy order filled, placing SELL order for XRPUSDT @ 0.29549),
quantity: 100.0
00:20:09.391 [info]  Trade finished, trader will now exit
```

As config already points to it so we can continue as previously by starting the streaming and trading on the symbol. The trader is using the `BinanceMock` and it looks like everything works as it would be dealing with a real exchange.

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

Source code for this chapter can be found at Github

Chapter 5

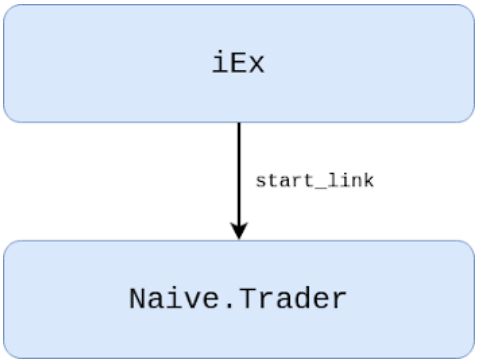
Enable parallel trading on multiple symbols

5.1 Objectives

- design supervision tree that will allow trading using multiple traders in parallel per symbol
- update application supervisor
- implement `Naive.Server`
- implement `Naive.SymbolSupervisor`

5.2 Introduction - architectural design

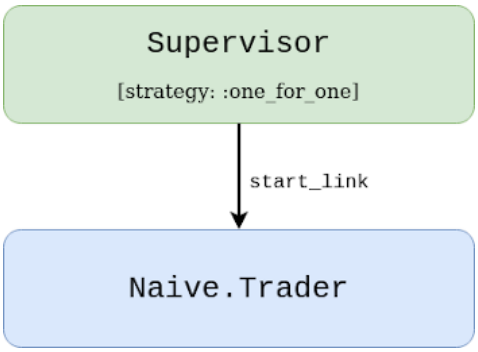
In the second chapter, we implemented a basic trader which goes through the trading cycle. Inside the iEx session, we were starting the `Naive.Trader` process using the `start_link/1` function:



The `GenServer.start_link/3` creates a link between IEx's process and new `Naive.Trader` process. Whenever a trader process terminates(either finishes the trading cycle or there was an error), a new one won't get started as there's no supervision at all.

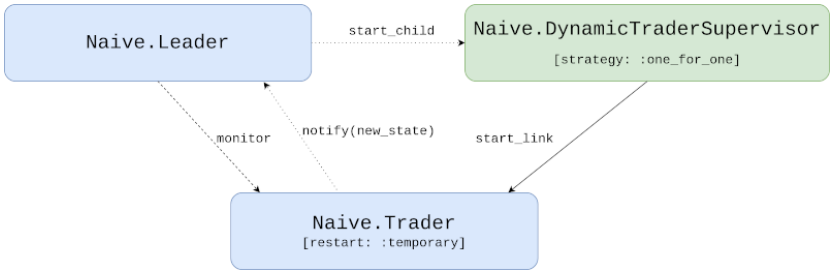
We can do much better than that with a little bit of help from Elixir and OTP.

Let's introduce a supervisor above our trader process. It will start a new trader process whenever the previous one finished/crashed:



This looks much better but there are few problems with it. So, when the trader will start to place orders it will be in *some* state(it will hold buy/sell orders) that the supervisor won't be aware of. In case of trader crashing, the supervisor will start a new trader *without* any knowledge of possibly placed orders or any other information from the state(it will be started with a "fresh" state).

To fix that we need to keep a copy of the trader's state outside of the trader process - that's why we will introduce a new server called `Naive.Leader` that will keep track of traders' data:



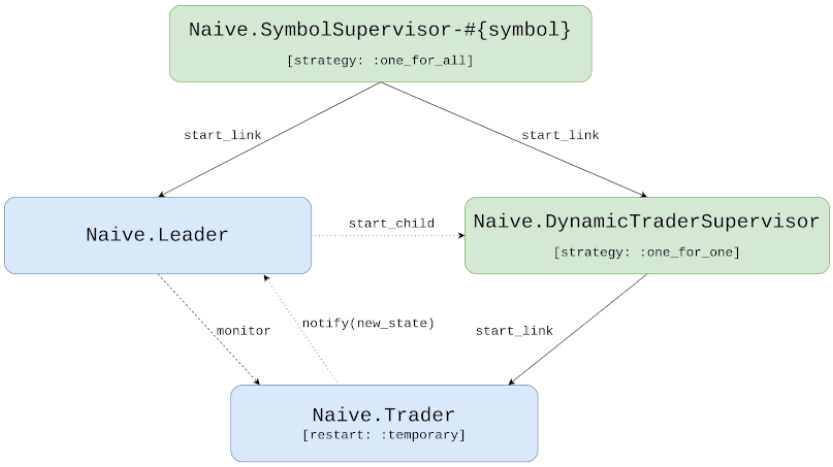
The `Naive.Leader` will become the interface to start new *traders*. It will call the `start_child/1` function of the Supervisor, then consequently `DynamicTraderSupervisor` will call the `start_link/1` function of our `Naive.Trader` module.

We can also see that our `Naive.Trader`'s are now started with the `temporary restart` option. Setting this option will disallow the Supervisor from restarting the traders on its own. The responsibility of restarting traders will now be shifted to the leader. The leader will monitor the traders and restart them to a correct state when any crashes.

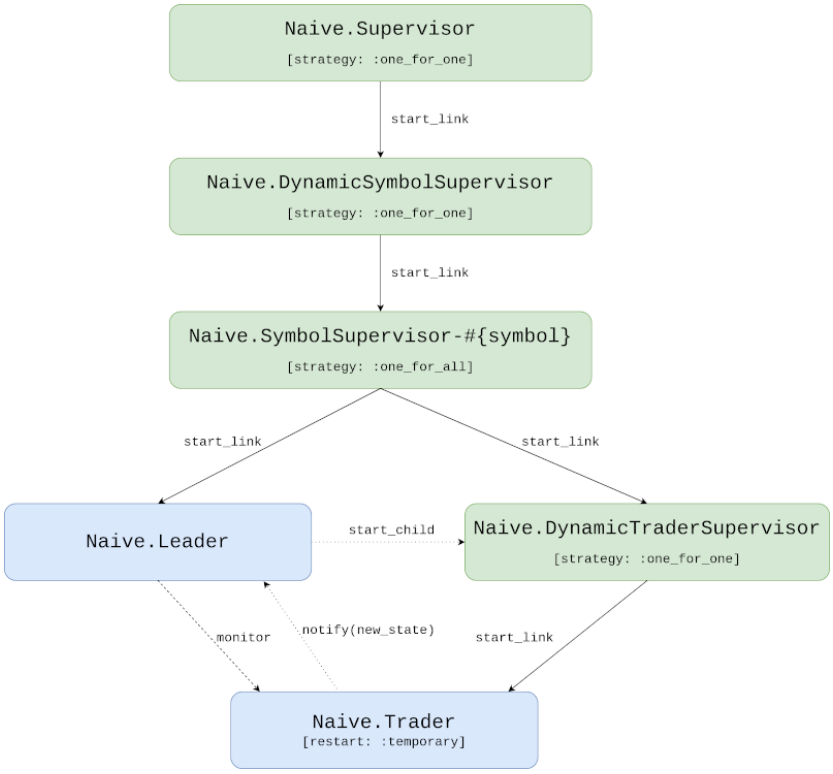
As trader state will get updated, it will notify the leader about its new state to be stored. This way whenever a trader process would crash, the leader will be able to start a new trader process with the last known state.

This setup will also allow us to start and supervise multiple traders for a single symbol which our naive strategy will require in the future(next chapter).

For each symbol that we will be trading on we need the above trio of services(Leader + DynamicTraderSupervisor + Trader), to effectively initialize(and supervise) them we will add an `Naive.SymbolSupervisor` that will start both `Naive.Leader` and `Naive.Dynamic`:



We will need multiple symbol supervisors, one for each symbol that we would like to trade on. As with traders, they will be dynamically started on demand, this should give us a hint that we need another dynamic supervisor that will supervise symbol supervisors and will be the direct child of our `Naive.Supervisor`(`Naive.Application` module):



You could ask yourself why we don't need some additional server to track which symbols are traded at the moment (in the same way as `Naive.Leader` tracks `Naive.Traders`). The answer is that we don't need to track them as we register

all `Naive.SymbolSupervisor`s with a name containing a symbol that they trade on. This way we will always be able to refer to them by registered name instead of PIDs/refs.

Here's what happens starting from the top of the graph:

- the `Naive.Application` is our top-level application's supervisor for the `naive` app, it was auto-generated as a part of the `naive` app
- it has a single child `Naive.DynamicSymbolSupervisor`, which has strategy one_for_one and all of its children are `Naive.SymbolSupervisor`s
- `Naive.SymbolSupervisor` process will start 2 further children: the `Naive.Leader` and `DynamicTraderSupervisor`, both created on init
- the `Naive.Leader` will ask `DynamicTraderSupervisor` to start the `Naive.Trader` child process(es)

This can be a little bit confusing at the moment but it will get a lot easier as we will write the code. Let's get to it!

5.2.1 Update application supervisor

Let's start by adding a `Naive.DynamicSymbolSupervisor` and a server to the children list of the `Naive.Application` supervisor:

```
# /apps/naive/lib/naive/application.ex
def start(_type, _args) do
  children = [
    {
      DynamicSupervisor,
      strategy: :one_for_one,
      name: Naive.DynamicSymbolSupervisor
    }
  ]

  ...
end
```

5.2.2 Add interface method

We will now add an interface method to the `Naive` module that will instruct `Naive.DynamicSymbolSupervisor` to start `Naive.SymbolSupervisor`(to be implemented next) as its child:

```
# /apps/naive/lib/naive.ex
def start_trading(symbol) do
  symbol = String.upcase(symbol)

  {:ok, _pid} =
    DynamicSupervisor.start_child(
```

```
        Naive.DynamicSymbolSupervisor,
        {Naive.SymbolSupervisor, symbol}
      )
  end
```

5.3 Implement `Naive.SymbolSupervisor`

Next, time for the `Naive.SymbolSupervisor`, the first step will be to create a file called `symbol_supervisor.ex` inside `apps/naive/lib/naive` directory. There's no point in using the DynamicSupervisor, as we know the children that we would like to start automatically on init. This is a full implementation of the supervisor and it's a simple as just listing child processes inside the init function:

```
# /apps/naive/lib/naive/symbol_supervisor.ex
defmodule Naive.SymbolSupervisor do
  use Supervisor

  require Logger

  def start_link(symbol) do
    Supervisor.start_link(
      __MODULE__,
      symbol,
      name: :"#{__MODULE__}-#{symbol}"
    )
  end

  def init(symbol) do
    Logger.info("Starting new supervision tree to trade on #{symbol}")

    Supervisor.init(
      [
        {
          DynamicSupervisor,
          strategy: :one_for_one,
          name: :"Naive.DynamicTraderSupervisor-#{symbol}"
        },
        {Naive.Leader, symbol}
      ],
      strategy: :one_for_all
    )
  end
end
```

It's advised to keep supervisor processes slim.

We registered the `Naive.SymbolSupervisor` processes with names, which will help us understand the supervision tree inside the observer GUI(it will also allow us to stop those supervisors in the future).

As mentioned previously whenever either the `Naive.Leader` or `Naive.DynamicSymbolSupervisor-#{symbol}` would crash we would like to kill the other child process as we won't be able to recover the state - it's just easier to init both again.

5.4 Implement `Naive.Leader`

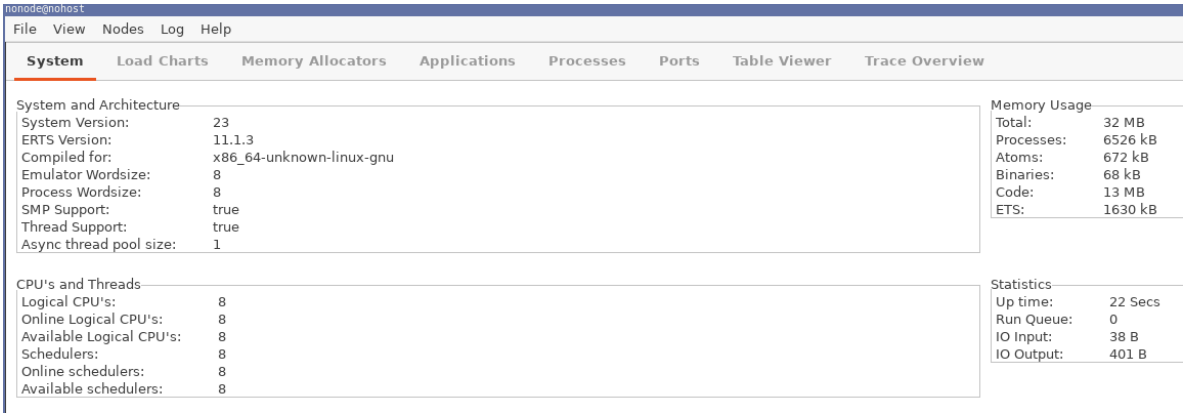
It's time for the `Naive.Leader` module, again, the first step will be to create a file called the `leader.ex` inside `apps/naive/lib/naive` directory. At this moment it will be a skeleton GenServer implementation just to get the code to compile:

```
# /apps/naive/lib/naive/leader.ex
defmodule Naive.Leader do
  use GenServer

  def start_link(symbol) do
    GenServer.start_link(
      __MODULE__,
      symbol,
      name: :"#{__MODULE__}-#{symbol}"
    )
  end

  def init(symbol) do
    {:ok, %{symbol: symbol}}
  end
end
```

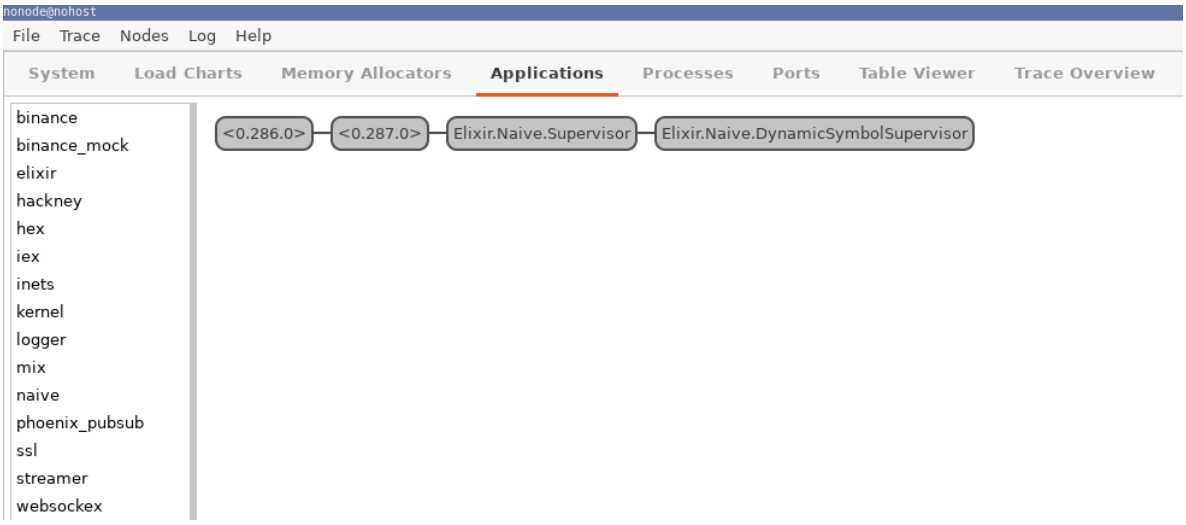
At this moment we have half of the supervision tree working so we can give it a spin in iex. Using the observer we will be able to see all processes created when the `start_trading/1` function gets called:

```
$ iex -S mix
...
iex(1)> :observer.start()
```

The above function will open a new window looking as follows:



To clearly see the supervision tree we will click on the "Applications" tab at the top - the following tree of processes will be shown on the left:



If any other process tree is visible, go to the list on the left and select the `naive` application.

The `Naive.Supervisor` is our `Naive.Application` module(you can confirm that by checking the `name` option send to the `start_link` function inside the module). It starts the `Naive.DynamicSymbolSupervisor`.

We can now call the `Naive.start_trading/1` function couple time to see how the tree will look like with additional processes(go back to the `iex` session):

```
...
iex(2)> Naive.start_trading("adausdt")
23:14:40.974 [info]  Starting new supervision tree to trade on ADAUSDT
{:ok, #PID<0.340.0>}
iex(3)> Naive.start_trading("xrpusdt")
23:15:12.117 [info]  Starting new supervision tree to trade on XRPUSDT
{:ok, #PID<0.345.0>}
```

We can see that two new branches were created:

- `SymbolSupervisor-ADAUSDT`
- `SymbolSupervisor-XRPUSDT`

Each of them contains a `Naive.Leader` and `DynamicTraderSupervisor`.

5.4.1 Updating the `leader` module

Let's jump back to extending a leader implementation to get those traders running.

We will introduce a leader's state that will consist of a symbol, setting, and a list of traders' data. Trader data will hold PID, ref, and state of the trader:

```
# /apps/naive/lib/naive/leader.ex
  ...
  alias Naive.Trader

  require Logger

  @binance_client Application.compile_env(:naive, :binance_client)

  defmodule State do
    defstruct symbol: nil,
              settings: nil,
              traders: []
  end

  defmodule TraderData do
    defstruct pid: nil,
              ref: nil,
              state: nil
  end
```

We will use a `handle_continue` callback which was introduced in Erlang 21 to initialize the leader asynchronously. To do that we will return a tuple starting with a `:continue` atom from inside the init function:

```
  # /apps/naive/lib/naive/leader.ex
  def init(symbol) do
    {:ok,
      %State{
          symbol: symbol
      }, {:continue, :start_traders}}
  end
```

The `Naive.Leader` will fetch symbol settings and based on them, it will build the state for traders so they don't need to fetch the same settings again. It will also start as many traders there were set under chunks key in setting:

```
# /apps/naive/lib/naive/leader.ex
# below init()
def handle_continue(:start_traders, %{symbol: symbol} = state) do
  settings = fetch_symbol_settings(symbol)
  trader_state = fresh_trader_state(settings)
  traders = for _i <- 1..settings.chunks,
            do: start_new_trader(trader_state)

  {:noreply, %{state | settings: settings, traders: traders}}
end
```

Fetching symbol settings will be hardcoded for time being to keep this chapter focused. We will also move the code responsible for fetching tick size from the `Naive.Trader` to the `Naive.Leader` and hardcode the rest of the values:

```
# /apps/naive/lib/naive/leader.ex
defp fetch_symbol_settings(symbol) do
  tick_size = fetch_tick_size(symbol)

  %{
    symbol: symbol,
    chunks: 1,
    # -0.12% for quick testing
    profit_interval: "-0.0012",
    tick_size: tick_size
  }
end

defp fetch_tick_size(symbol) do
  @binance_client.get_exchange_info()
  |> elem(1)
  |> Map.get(:symbols)
  |> Enum.find(&(&1["symbol"] == symbol))
  |> Map.get("filters")
  |> Enum.find(&(&1["filterType"] == "PRICE_FILTER"))
  |> Map.get("tickSize")
end
```

Additionally, we need to create a helper method that we used inside the `handle_continue/2` callback called `fresh_trader_state/1`:

```
  # /apps/naive/lib/naive/leader.ex
  # place this one above the `fetch_symbol_settings` function
  defp fresh_trader_state(settings) do
    struct(Trader.State, settings)
  end
```

Starting a new trader isn't any different from the code that we already wrote to start a new `Naive.SymbolSupervisor`. We need to call the `DynamicSupervisor.start_child/2` function and start to monitor the process:

```
  # /apps/naive/lib/naive/leader.ex
  defp start_new_trader(%Trader.State{} = state) do
    {:ok, pid} =
      DynamicSupervisor.start_child(
        :"Naive.DynamicTraderSupervisor-#{state.symbol}",
        {Naive.Trader, state}
      )

    ref = Process.monitor(pid)

    %TraderData{pid: pid, ref: ref, state: state}
  end
```

5.4.2 Updating the `Naive.Trader` module

Now we can update the `Naive.Trader`, first, we will set restart to be temporary to avoid restarting it by the `Naive.DynamicTraderSupervisor`:

```
# /apps/naive/lib/naive/trader.ex
defmodule Naive.Trader do
  use GenServer, restart: :temporary
  ...
```

Next, we will update the `start_link/1` and `init/1` functions to take the state instead of building it from args:

```
  # /apps/naive/lib/naive/trader.ex
  def start_link(%State{} = state) do
    GenServer.start_link(__MODULE__, state)
  end

  def init(%State{symbol: symbol} = state) do
    symbol = String.upcase(symbol)
```

```
  Logger.info("Initializing new trader for symbol(#{symbol})")

  Phoenix.PubSub.subscribe(
    Streamer.PubSub,
    "TRADE_EVENTS:#{symbol}"
  )

  {:ok, state}
end
```

Next, we need to update two `handle_info/2` callbacks that change the state of the `Naive.Trader` process(when placing buy order and when placing sell order). They will need to notify the `Naive.Leader` that the state is changed before returning it:

```
# /apps/naive/lib/naive/trader.ex
...

def handle_info(
      ...
    ) do
  Logger.info("Placing buy order (#{symbol}@#{price})")
  ...
  new_state = %{state | buy_order: order}
  Naive.Leader.notify(:trader_state_updated, new_state)
  {:noreply, new_state}
end

def handle_info(
      ...
    ) do
  ...
  Logger.info("Buy order filled, placing sell order ...")
  ...

  new_state = %{state | sell_order: order}
  Naive.Leader.notify(:trader_state_updated, new_state)
  {:noreply, new_state}
end
...
```

5.4.3 Finalizing `Naive.Leader` implementation

Now we need to get back to the `Naive.Leader` where we will implement the notifying logic. We will start with the notify function that will just call the `Naive.Leader` process:

```
# /apps/naive/lib/naive/leader.ex
# below init

def notify(:trader_state_updated, trader_state) do
  GenServer.call(
    :"#{__MODULE__}-#{trader_state.symbol}",
    {:update_trader_state, trader_state}
  )
end
```

Now, it's time for a callback function that will handle the trader state update. As this is a `handle_call/3` callback we have access to the trader PID which sent the notification message. We will try to find that trader in the list of traders. If that's successful we will update the cached state for that trader locally:

```
# /apps/naive/lib/naive/leader.ex
# below handle_continue
def handle_call(
  {:update_trader_state, new_trader_state},
  {trader_pid, _},
  %{traders: traders} = state
) do
  case Enum.find_index(traders, &(&1.pid == trader_pid)) do
    nil ->
      Logger.warn(
        "Tried to update the state of trader that leader is not aware of"
      )
      {:reply, :ok, state}

    index ->
      old_trader_data = Enum.at(traders, index)
      new_trader_data = %{old_trader_data | :state => new_trader_state}

      {:reply, :ok, %{state | :traders =>
        List.replace_at(traders, index, new_trader_data)}}
  end
end
```

Another callback functions that we will need to provide are two `handle_info/2` functions that will handle the trade finished scenario as well as crashed trader.

First, trade finished scenario. As previously, we will try to find the trader data in the traders list. If that's successful, we will start a new trader with a fresh state. We will also overwrite existing trader data locally(as PID, ref, and state changed):

```
# /apps/naive/lib/naive/leader.ex
# below state updated handle_call callback
def handle_info(
      {:DOWN, _ref, :process, trader_pid, :normal},
      %{traders: traders, symbol: symbol, settings: settings} = state
    ) do
  Logger.info("#{symbol} trader finished trade - restarting")

  case Enum.find_index(traders, &(&1.pid == trader_pid)) do
    nil ->
      Logger.warn(
        "Tried to restart finished #{symbol} " <>
          "trader that leader is not aware of"
      )

      {:noreply, state}

    index ->
      new_trader_data = start_new_trader(fresh_trader_state(settings))
      new_traders = List.replace_at(traders, index, new_trader_data)

      {:noreply, %{state | traders: new_traders}}
  end
end
```

Here we will assume that whenever the reason that the `Naive.Trader` process died is `:normal` that means that we stopped it after trade cycle finished.

The final callback that we need to provide will handle the scenario where the trader crashed. We would like to find the cached state of the crashed trader and start a new one with the same state and then update the local cache as PID and ref will change for that trader:

```
# /apps/naive/lib/naive/leader.ex
# below trade finished handle_info callback
def handle_info(
      {:DOWN, _ref, :process, trader_pid, reason},
      %{traders: traders, symbol: symbol} = state
    ) do
  Logger.error("#{symbol} trader died - reason #{reason} - trying to restart")

  case Enum.find_index(traders, &(&1.pid == trader_pid)) do
```

```
      nil ->
        Logger.warn(
          "Tried to restart #{symbol} trader " <>
            "but failed to find its cached state"
        )

        {:noreply, state}

      index ->
        trader_data = Enum.at(traders, index)
        new_trader_data = start_new_trader(trader_data.state)
        new_traders = List.replace_at(traders, index, new_trader_data)

        {:noreply, %{state | traders: new_traders}}
    end
  end
```

5.4.4 IEx testing

That finishes the implementation part, let's jump into the IEx session to see how it works.

We will start the observer first, then we will start trading on any valid symbol.

When our trader will start, you should be able to right-click and select "Kill process"(leave the reason as kill) and click "OK". At that moment you should see that the PID of the trader changed and we can also see a log message from the leader.

```
$ iex -S mix
...
iex(1)> :observer.start()
:ok
iex(2)> Naive.start_trading("xrpusdt")

00:04:35.041 [info]  Starting new supervision tree to trade on XRPUSDT
{:ok, #PID<0.455.0>}
00:04:37.697 [info]  Initializing new trader for XRPUSDT
iex(3)>
00:08:01.476 [error] XRPUSDT trader died - trying to restart
00:08:01.476 [info]  Initializing new trader for XRPUSDT
```

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

Source code for this chapter can be found at Github

Chapter 6

Introduce a `buy_down_interval` to make a single trader more profitable

6.1 Objectives

- present reasons why to introduce `buy_down_interval`
- add `buy_down_interval` to `Naive.Trader`'s state and calculate buy price
- add `buy_down_interval` to `Naive.Trader`'s state compiled by the `Naive.Leader`
- manually test the implementation inside iex

6.2 Why we need to buy below the current price? Feature overview

The `Naive.Trader` process(marked in above diagram with blue color) at the arrival of the first trade event, immediately places a buy order at the current price. At the moment when the buy order gets filled, it places the sell order which later also gets filled.

The Trader A exits and a new trader B is started which again immediately places a buy order *at the same price* as the previous trader just sold. When this gets filled sell order gets placed and the loop continues on and on.

We can see that there's a problem here as we just paid a fee twice(once for selling by the Trader A and once for buying by the Trader B) without really gaining anything(the Trader A could just hold the currency and could simply cash in on double profit in this specific situation).

The solution is to be more clever about our buy order's price. The idea is simple, instead of placing a new buy order at the current price(price from the last TradeEvent), we will introduce a `buy_down_interval`:

So every new `Naive.Trader` process as it receives the first trade event, the trader will take its price and will calculate a decreased price by using the `buy_down_interval` value(for example 0.005 would be 0.5%) and place a buy order at that calculated price.

When looking at the chart above we can figure out that `buy_down_interval` should never be smaller than double the fee(at the moment of writing transaction fee is 0.1%) that you are paying per transaction.

6.3 `Naive.Trader` implementation

Let's open the `Naive.Trader` module's file(`/apps/naive/lib/naive/trader.ex`) and add `buy_down_interval` to its state:

```
# /apps/naive/lib/naive/trader.ex
...
defmodule State do
  @enforce_keys [
    :symbol,
    :buy_down_interval, # <= add this line
    :profit_interval,
    :tick_size
  ]
```

```
  defstruct [
    :symbol,
    :buy_order,
    :sell_order,
    :buy_down_interval, # <= add this line
    :profit_interval,
    :tick_size
  ]
end
...
```

Next, we need to update the initial `handle_info/2` callback which places the buy order. We need to retrieve the `buy_down_interval` and the `tick_size` from the `state` of the trader to be able to calculate the buy price. We will put the logic to calculate that price in a separate function at the end of the file:

```
# /apps/naive/lib/naive/trader.ex
...
def handle_info(
      %TradeEvent{price: price},
      %State{
        symbol: symbol,
        buy_order: nil,
        buy_down_interval: buy_down_interval, # <= add this line
        tick_size: tick_size                  # <= add this line
      } = state
    ) do
  price = calculate_buy_price(price, buy_down_interval, tick_size)
  # ^ add above call
  ...
```

To calculate the buy price we will use a very similar method to the one used before to calculate the sell price. First, we will need to cast all variables into the `Decimal` structs and then, we will simply subtract the `buy_down_interval` of the price from the price. The number that we will end up with won't necessarily be a legal price as every price needs to be divisible by the `tick_size` which we will assure in the last calculation:

```
# /apps/naive/lib/naive/trader.ex
...
defp calculate_buy_price(current_price, buy_down_interval, tick_size) do
  # not necessarily legal price
  exact_buy_price =
    D.sub(
      current_price,
      D.mult(current_price, buy_down_interval)
    )
```

```
    D.to_string(
      D.mult(
        D.div_int(exact_buy_price, tick_size),
        tick_size
      ),
      :normal
    )
  end
  ...
```

6.4 `Naive.Leader` implementation

Next, we need to update the `Naive.Leader` as it needs to add `buy_down_interval` to the `Naive.Trader`'s state:

```
  # /apps/naive/lib/naive/leader.ex
  defp fetch_symbol_settings(symbol) do
    ...

    %{
      symbol: symbol,
      chunks: 1,
      # 0.01% for quick testing
      buy_down_interval: "0.0001", # <= add this line
      # -0.12% for quick testing
      profit_interval: "-0.0012",
      tick_size: tick_size
    }
  end
  ...
```

6.4.1 IEx testing

That finishes the `buy_down_interval` implementation, we will jump into the IEx session to see how it works, but before that, for a moment we will change the logging level to `debug` to see current prices:

```
# config/config.exs
...
config :logger,
  level: :debug # <= updated for our manual test
...
```

After starting the streaming we should start seeing log messages with current prices. As we updated our implementation we should place our buy order below the current price as it's visible below:

```
$ iex -S mix
...
iex(1)> Streamer.start_streaming("FLMUSDT")
{:ok, #PID<0.313.0>}
iex(2)> Naive.start_trading("FLMUSDT")
21:16:14.829 [info]  Starting new supervision tree to trade on FLMUSDT
...
21:16:16.755 [info]  Initializing new trader for FLMUSDT
...
21:16:20.000 [debug] Trade event received FLMUSDT@0.15180000
21:16:20.009 [info]  Placing BUY order for FLMUSDT @ 0.1517, quantity: 100
```

As we can see our `Naive.Trader` process placed a buy order below the current price (based on the most recent trade event received)

[Note] Please remember to revert the change to logger level as otherwise there's too much noise in the logs.

[Note 2] Please remember to run the `mix format` to keep things nice and tidy.

Source code for this chapter can be found at Github

Chapter 7

Introduce a trader budget and calculating the quantity

7.1 Objectives

- fetch step_size
- append budget and step_size to the `Trader`'s state compiled by the `Leader`
- append budget and step_size to the `Trader`'s state
- calculate quantity

7.2 Fetch `step_size`

In the 2nd chapter we hardcoded `quantity` to 100, it's time to refactor that. We will need `step_size` information from the Binance which we are already retrieving together with `tick_size` in the `exchangeInfo` call(but not getting it out from the response). So we will rename the `fetch_tick_size/1` function to `fetch_symbol_filters/1` which will allow us to return multiple filters(`tick_size` and `step_size`) from that function.

```
# /apps/naive/lib/naive/leader.ex
...
defp fetch_symbol_settings(symbol) do
  symbol_filters = fetch_symbol_filters(symbol) # <= updated fetch_tick_size

  Map.merge(
    %{
      symbol: symbol,
      chunks: 1,
      budget: 20,
      # -0.01% for quick testing
      buy_down_interval: "0.0001",
```

```
      # -0.12% for quick testing
      profit_interval: "-0.0012"
    },
    symbol_filters
  )
end

defp fetch_symbol_filters(symbol) do  # <= updated fetch_tick_size
  symbol_filters =
    @binance_client.get_exchange_info()
    |> elem(1)
    |> Map.get(:symbols)
    |> Enum.find(&(&1["symbol"] == symbol))
    |> Map.get("filters")

  tick_size =
    symbol_filters
    |> Enum.find(&(&1["filterType"] == "PRICE_FILTER"))
    |> Map.get("tickSize")

  step_size =
    symbol_filters
    |> Enum.find(&(&1["filterType"] == "LOT_SIZE"))
    |> Map.get("stepSize")

  %{
    tick_size: tick_size,
    step_size: step_size
  }
end
```

Instead of reassigning the filters one by one into the settings, we will merge them together(#1). Additionally, we will introduce a `budget`(#2) which will be shared across all traders of the symbol. Also, we don't need to assign `tick_size` here as it's part of the settings that are merged.

7.3 Append `budget` and `step_size` to the `Trader`'s state inside the `Leader`

The `budget` needs to be added to the `%State{}`(`step_size` will be automatically passed on by `struct/2`) of the trader inside `fresh_trader_state/1`(where we initialize the state of traders). Before we will assign it we need to divide it by the number of chunks as each trader gets only a chunk of the budget:

```
  # /apps/naive/lib/naive/leader.ex
  defp fresh_trader_state(settings) do
    %{
      struct(Trader.State, settings) |
      budget: D.div(settings.budget, settings.chunks)
    }
  end
```

In the code above we are using the `Decimal` module(aliased as `D`) to calculate the budget - we need to alias it at the top of `Naive.Leader`'s file:

```
# /apps/naive/lib/naive/leader.ex
defmodule Naive.Leader do
  use GenServer

  alias Decimal, as: D # <= add this line
  alias Naive.Trader
  ...
```

7.4 Append `budget` and `step_size` to the `Trader`'s state

We need to add both `budget` and `step_size` to the `Naive.Trader`'s state struct:

```
  # /apps/naive/lib/naive/trader.ex
  ...
  defmodule State do
    @enforce_keys [
      :symbol,
      :budget, # <= add this line
      :buy_down_interval,
      :profit_interval,
      :tick_size,
      :step_size # <= add this line and comma above
    ]
    defstruct [
      :symbol,
      :budget, # <= add this line
```

```
      :buy_order,
      :sell_order,
      :buy_down_interval,
      :profit_interval,
      :tick_size,
      :step_size # <= add this line and comma above
    ]
  end
  ...
```

7.5 Calculate quantity

Jumping back to the `handle_info/2` where the `Naive.Trader` places a buy order, we need to pattern match on the `step_size` and `budget` then we will be able to swap hardcoded quantity with the result of calling the `calculate_quantity/3` function:

```
# /apps/naive/lib/naive/trader.ex
...
def handle_info(
      %TradeEvent{price: price},
      %State{
        symbol: symbol,
        budget: budget, # <= add this line
        buy_order: nil,
        buy_down_interval: buy_down_interval,
        tick_size: tick_size,
        step_size: step_size # <= add this line
      } = state
    ) do
  ...
  quantity = calculate_quantity(budget, price, step_size)
  ...
```

To calculate quantity we will just divide the `budget` by the `price` with a caveat that it's possible (as with calculating the price) that it's not a legal quantity value as it needs to be divisible by `step_size`:

```
# /apps/naive/lib/naive/trader.ex
# add below at the bottom of the file
...
defp calculate_quantity(budget, price, step_size) do
  # not necessarily legal quantity
  exact_target_quantity = D.div(budget, price)
```

```
    D.to_string(
      D.mult(
        D.div_int(exact_target_quantity, step_size),
        step_size
      ),
      :normal
    )
  end
```

7.5.1 IEx testing

That finishes the `quantity`(and `budget`) implementation, we will jump into the IEx session to see how it works.

First, start the streaming and trading on the same symbol and a moment later you should see a variable amount of quantity that more or less uses the full allowed budget:

```
$ iex -S mix
...
iex(1)> Streamer.start_streaming("XRPUSDT")
{:ok, #PID<0.313.0>}
iex(2)> Naive.start_trading("XRPUSDT")
21:16:14.829 [info]  Starting new supervision tree to trade on XRPUSDT
21:16:16.755 [info]  Initializing new trader for XRPUSDT
21:16:20.009 [info]  Placing BUY order for XRPUSDT @ 0.29506, quantity: 67.7
21:16:23.456 [info]  Buy order filled, placing SELL order for XRPUSDT @ 0.29529,
quantity: 67.7
```

As we can see our `Naive.Trader` process is now buying and selling based on passed budget.

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

Source code for this chapter can be found at Github

Chapter 8

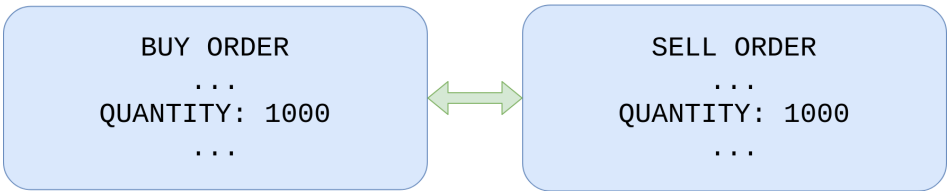
Add support for multiple transactions per order

8.1 Objectives

- describe the issue with the current implementation
- improve buy order filled callback
- implement buy order "filled" callback
- improve sell order callback

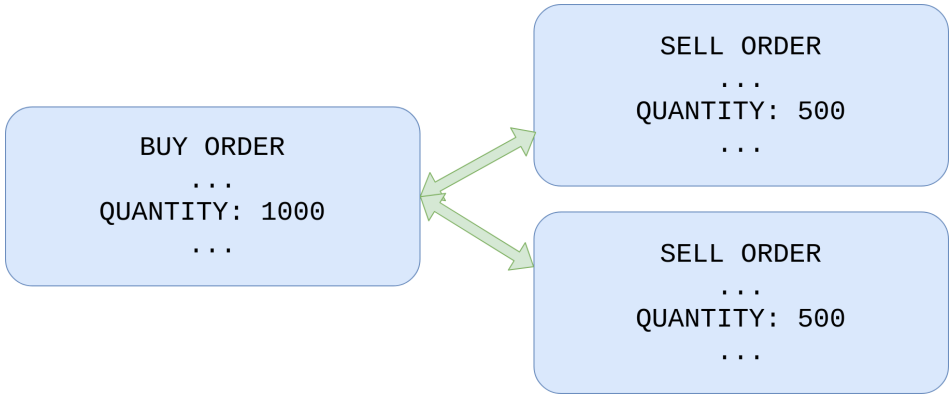
8.2 The issue with the current implementation

Currently, `Naive.Trader` process is placing a buy order and it's assuming that it will be filled by a *single* opposite sell order(we are pattern matching on quantity to confirm that):



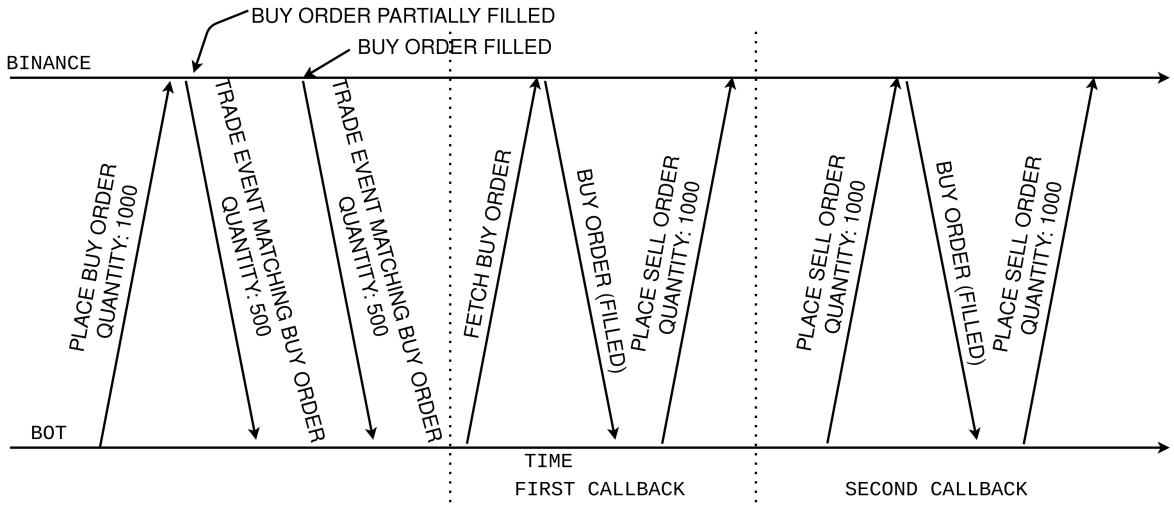
Here we can see our buy order for 1000 units(on the left) and other trader's sell order(on the right) for 1000 units. This(order fully filled in a single transaction) is a case most of the time but it's not ALWAYS the case.

Sometimes our order will be filled by two or more transactions:



The easiest and the safest way to check has this event filled our order fully is to fetch our order again from Binance at the moment when trade event filling our order arrives.

The problem with this approach is that sometimes we will run into a race condition:



From the left, first, we are sending a buy order for quantity 1000 to the Binance. It hangs for a while until it gets filled by 2 transactions that happened very quickly. Quickly enough for us to receive both messages almost in the same moment.

When our bot will handle the first one it will fetch the buy order which is already filled. It will cause the trader to place a sell order but then there's another trade event waiting in the message box. It will be handled by another callback that will again fetch the order and place another sell order to be placed and that's obviously not correct.

What we need to do is to update the status of the buy order after the first fetch(if it's filled) so when the second trade event arrives we will ignore it(this will require an additional callback).

The same issue will appear when placing a sell order and dealing with multiple simultaneous transactions.

8.3 Improve buy order filled callback

First, we need to modify the callback which monitors incoming trade events for ones filling its buy order and then places sell order. We need to remove pattern matching assuming that a single trade event will fill our buy order - we need to drop quantity check as well as add:

```
# /apps/naive/lib/naive/trader.ex
def handle_info(
      %TradeEvent{
        buyer_order_id: order_id # <= quantity got removed from here
      },
      %State{
        symbol: symbol,
        buy_order:
          %Binance.OrderResponse{
            price: buy_price,
            order_id: order_id,
            orig_qty: quantity,
            transact_time: timestamp # <= timestamp added to query order
          } = buy_order, # <= buy order to update it
        profit_interval: profit_interval,
        tick_size: tick_size
      } = state
    ) do
```

Now we can fetch our buy order to check is it already filled. We will get the `Binance.Order` struct instead of the `Binance.OrderResponse` that we normally deal with. At this moment we will simply update our `Binance.OrderResponse` struct from the state:

```
# /apps/naive/lib/naive/trader.ex
# inside the same callback
def handle_info(
    ...
    ) do
  {:ok, %Binance.Order{} = current_buy_order} =
    @binance_client.get_order(
      symbol,
      timestamp,
      order_id
    )

  buy_order = %{buy_order | status: current_buy_order.status}
  ...
```

The rest of the logic inside this callback will depend on the `status` of the buy order. If our buy order is "filled" we would like to follow the existing logic but also update the `buy_order` field inside the state of the trader process. On the other hand, if our order is not yet filled the only thing to do is to update the `buy_order` field inside the state of the Trader process.

Here's an updated body below the above changes(few variables got renamed for clarity as we are now fetching the order):

```
# /apps/naive/lib/naive/trader.ex
# inside the same callback
buy_order = ....

  {:ok, new_state} =
    if buy_order.status == "FILLED" do
      sell_price = calculate_sell_price(buy_price, profit_interval, tick_size)

      Logger.info(
        "Buy order filled, placing SELL order for " <>
          "#{symbol} @ #{sell_price}, quantity: #{quantity}"
      )

      {:ok, %Binance.OrderResponse{} = order} =
        @binance_client.order_limit_sell(symbol, quantity, sell_price, "GTC")

      {:ok, %{state | buy_order: buy_order, sell_order: order}}
    else
      Logger.info("Buy order partially filled")
      {:ok, %{state | buy_order: buy_order}}
    end

  Naive.Leader.notify(:trader_state_updated, new_state)
  {:noreply, new_state}
end
```

As we are branching our logic and both paths are updating the state, we will return it together with an `:ok` atom to be able to pattern match it and assign it as a new state.

8.4 Implement buy order "filled" callback

The above callback covers the case where we will get multiple transactions filling our buy order. We aren't yet covering for the race condition described at the beginning of this chapter. When another trade event matching `buyer_order_id` would arrive, the above callback would be used and another sell order would be placed. To avoid that we need to add a new callback *ABOVE* the one that we just edited that will match `buyer_order_id` together with "filled" `status` and it will simply ignore that trade event as we know that sell event needed to be placed by previous trade event:

```
# /apps/naive/lib/naive/trader.ex
# place this callback ABOVE callback from previous section
def handle_info(
      %Streamer.Binance.TradeEvent{
        buyer_order_id: order_id
      },
      %State{
        buy_order: %Binance.OrderResponse{
          order_id: order_id, # <= confirms that it's event for buy order
          status: "FILLED" # <= confirms buy order filled
        },
        sell_order: %Binance.OrderResponse{} # <= confirms sell order placed
      } = state
    ) do
  {:noreply, state}
end
```

8.5 Improve sell order callback

Let's move on to the callback where the trader receives a trade event matching the sell order's id (about line 135 inside the `Naive.Trader` module).

We need to modify the header of our callback in the following ways:

- drop both pattern matches on `quantity` as we already know that trade event could partially fill our order (#1)
- get `symbol` out of state (#2)
- get `transact_time` out of the `sell_order` (used to fetch `get_order`) (#3)
- assign `sell_order` to a variable (#4)

```
# /apps/naive/lib/naive/trader.ex
def handle_info(
      %TradeEvent{
        seller_order_id: order_id # `quantity` check removed below (#1)
      },
      %State{
```

```
        symbol: symbol, (#2)
        sell_order:
          %Binance.OrderResponse{
            order_id: order_id,
            transact_time: timestamp # `transact_time` to `get_order` (#3)
          } = sell_order # to update order (#4)
      } = state
    ) do
```

Moving to the body of the function, we need to:

- fetch current state of our sell order
- update `status` of our `sell_order` from Trader's state
- branch out the logic based on `status` of the `sell_order`:
  - log and return the `:stop` atom to stop the GenServer

  or

  - update the state with new `sell_order` and continue

Here's the full body of our callback:

```
  # /apps/naive/lib/naive/trader.ex
  # inside the callabck
  {:ok, %Binance.Order{} = current_sell_order} =
    @binance_client.get_order(
      symbol,
      timestamp,
      order_id
    )

  sell_order = %{sell_order | status: current_sell_order.status}

  if sell_order.status == "FILLED" do
    Logger.info("Trade finished, trader will now exit")
    {:stop, :normal, state}
  else
    Logger.info("Sell order partially filled")
    new_state = %{state | sell_order: sell_order}
    {:noreply, new_state}
  end
```

8.6 Test the implementation

Testing this feature is a bit tricky as it requires trading on real Binance exchange(as our BinanceMock always fills orders with a single transaction) as well as race condition to happen :) Not that easy but even without race condition we should still test that code works as expected with BinanceMock:

```
$ iex -S mix
...
iex(1)> Naive.start_trading("XRPUSDT")
23:27:35.977 [info]  Starting new supervision tree to trade on XRPUSDT
{:ok, #PID<0.331.0>}
23:27:39.073 [info]  Initializing new trader for XRPUSDT
iex(2)> Streamer.start_streaming("XRPUSDT")
{:ok, #PID<0.345.0>}
23:31:57.044 [info]  Initializing new trader for XRPUSDT
23:31:57.888 [info]  Placing BUY order for XRPUSDT @ 0.28031, quantity: 71.3
23:32:01.023 [info]  Buy order filled, placing SELL order for XRPUSDT @ 0.28053,
quantity: 71.30000000
23:33:08.865 [info]  Trade finished, trader will now exit
23:33:08.865 [info]  XRPUSDT Trader finished - restarting
```

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

Source code for this chapter can be found at Github

Chapter 9

Run multiple traders in parallel

9.1 Objectives

- describe and design the required functionality
- implement rebuy in the `Naive.Trader`
- implement rebuy in the `Naive.Leader`
- improve logs by assigning ids to traders

9.2 Describe and design the required functionality

At this moment, inside the `Naive.Leader` we have a silly code that starts all of the traders at the same moment:

```
# /apps/naive/lib/naive/leader.ex
...
traders =
  for _i <- 1..settings.chunks,
      do: start_new_trader(trader_state)
...
```

All the changes we made in this episode will enable us to fix this.

Let's say that we placed a buy order that got filled and the price has fallen before reaching the sell level. We can see here that we missed a nice opportunity to buy more as price drops and make money as it climbs back:

We will implement an additional trade event callback inside the `Naive.Trader` that will keep checking the price after the buy order has been filled. Whenever a price drops below the `buy_order`'s `price` by `rebuy_interval` we will notify the `Naive.Leader` to start the new `Naive.Trader` process:

The `Naive.Leader` keeps track of how many `Naive.Trader`s are running and needs to honor the number of `chunks` set up in the settings (one chunk == one trader).

To stop the `Naive.Trader`s from continuously notifying about a drop in the price we will also introduce a boolean flag that will track has the `Naive.Leader` been already notified.

9.3 Implement rebuy inside `Naive.Trader`

We will start by adding the `rebuy_interval` and the `rebuy_notified` to the trader's state:

```
# /apps/naive/lib/naive/trader.ex
...
defmodule State do
  @enforce_keys [
    :symbol,
    :budget,
```

```
      :buy_down_interval,
      :profit_interval,
      :rebuy_interval, # <= add this field
      :rebuy_notified, # <= add this field
      :tick_size,
      :step_size
    ]
    defstruct [
      :symbol,
      :budget,
      :buy_order,
      :sell_order,
      :buy_down_interval,
      :profit_interval,
      :rebuy_interval, # <= add this field
      :rebuy_notified, # <= add this field
      :tick_size,
      :step_size
    ]
  end
```

Rebuy logic should be placed almost as the last callback just before the one that ignores all events. We will need to retrieve the current `price` and `buy_price` and confirm that we didn't notify the leader yet(`rebuy_notified` flag):

```
  # /apps/naive/lib/naive/trader.ex
  ...
  # sell filled callback here
  ...
  def handle_info(
        %TradeEvent{
          price: current_price
        },
        %State{
          symbol: symbol,
          buy_order: %Binance.OrderResponse{
            price: buy_price
          },
          rebuy_interval: rebuy_interval,
          rebuy_notified: false
        } = state
      ) do

  end
  # catch all callback here
```

We need to calculate is the current price below the rebuy interval. If yes we will notify the leader and update the boolean flag. We will abstract calculation to separate function(for readability) that we will write below:

```
  # /apps/naive/lib/naive/trader.ex
  # body of the above callback
  if trigger_rebuy?(buy_price, current_price, rebuy_interval) do
    Logger.info("Rebuy triggered for #{symbol} trader")
    new_state = %{state | rebuy_notified: true}
    Naive.Leader.notify(:rebuy_triggered, new_state)
    {:noreply, new_state}
  else
    {:noreply, state}
  end
```

As mentioned before, we will set the `rebuy_notified` boolean flag to true and notify the leader using the `notify` function with the dedicated atom.

At the bottom of the module we need to add the `trigger_rebuy?` helper function:

```
# /apps/naive/lib/naive/trader.ex
# bottom of the module
defp trigger_rebuy?(buy_price, current_price, rebuy_interval) do
  rebuy_price =
    D.sub(
      buy_price,
      D.mult(buy_price, rebuy_interval)
    )

  D.lt?(current_price, rebuy_price)
end
```

9.4 Implement rebuy in the `Naive.Leader`

Moving on to the `Naive.Leader` module, we can get update starting of the traders automatically by the leader to starting just one inside `handle_continue`:

```
# /apps/naive/lib/naive/leader.ex
def handle_continue(:start_traders, %{symbol: symbol} = state) do
  ...
  traders = [start_new_trader(trader_state)] # <= updated part

  ...
end
```

We will need to add a new clause of the `notify` function that will handle the rebuy scenario:

```
# /apps/naive/lib/naive/leader.ex
# add below current `notify` function
def notify(:rebuy_triggered, trader_state) do
  GenServer.call(
    :"#{__MODULE__}-#{trader_state.symbol}",
    {:rebuy_triggered, trader_state}
  )
end
```

We need to add a new `handle_call` function that will start new traders only when there are still chunks available(enforce the maximum number of parallel traders) - let's start with a header:

```
# /apps/naive/lib/naive/leader.ex
# place this one after :update_trader_state handle_call
def handle_call(
      {:rebuy_triggered, new_trader_state},
      {trader_pid, _},
      %{traders: traders, symbol: symbol, settings: settings} = state
    ) do

end
```

There are few important details to make note of:

- we need the trader's PID to be able to find it details inside the list of traders
- we need settings to confirm the number of chunks to be able to limit the maximum number of parallel traders

Moving on to the body of our callback. As with other ones, we will check can we find a trader inside the list of traders, and based on that we will either start another one(if we didn't reach the limit of chunks) or ignore it:

```
  # /apps/naive/lib/naive/leader.ex
  # body of our callback
  case Enum.find_index(traders, &(&1.pid == trader_pid)) do
    nil ->
      Logger.warn("Rebuy triggered by trader that leader is not aware of")
      {:reply, :ok, state}

    index ->
      old_trader_data = Enum.at(traders, index)
      new_trader_data = %{old_trader_data | :state => new_trader_state}
      updated_traders = List.replace_at(traders, index, new_trader_data)
```

```
      updated_traders =
        if settings.chunks == length(traders) do
          Logger.info("All traders already started for #{symbol}")
          updated_traders
        else
          Logger.info("Starting new trader for #{symbol}")
          [start_new_trader(fresh_trader_state(settings)) | updated_traders]
        end

      {:reply, :ok, %{state | :traders => updated_traders}}
  end
```

In the above code, we need to remember to update the state of the trader that triggered the rebuy inside the `traders` list as well as add the state of a new trader to that list.

As with other setting we will hardcode the `rebuy_interval`(inside the `fetch_symbol_settings` function) and assign them to trader's state(inside the `fresh_trader_state` function):

```
# /apps/naive/lib/naive/leader.ex
defp fresh_trader_state(settings) do
  %{
    struct(Trader.State, settings)
    | budget: D.div(settings.budget, settings.chunks),
      rebuy_notified: false # <= add this line
  }
end

defp fetch_symbol_settings(symbol) do
  ...

  Map.merge(
    %{
      ...
      chunks: 5, # <= update to 5 parallel traders max
      budget: 100, # <= update this line
      ...
      profit_interval: "-0.0012",
      rebuy_interval: "0.001" # <= add this line
    },
    symbol_filters
  )
end
```

We also update the `chunks` and the `budget` to allow our strategy to start up to 5 parallel traders with a budget of 20

USDT each(100/5) as Binance has minimal order requirement at about $15(when using the `BinanceMock` this doesn't really matter).

9.5 Improve logs by assigning ids to traders

The final change will be to add an `id` to the trader's state so we can use it inside log messages to give us meaningful data about what's going on(otherwise we won't be able to tell which message was logged by which trader).

First, let's add the `id` into the `Naive.Leader`'s fresh_trader_state as it will be defined per trader:

```
# /apps/naive/lib/naive/leader.ex
defp fresh_trader_state(settings) do
  %{
    struct(Trader.State, settings)
    | id: :os.system_time(:millisecond), # <= add this line
      budget: D.div(settings.budget, settings.chunks),
      rebuy_notified: false
  }
end
```

Now we can move on to the `Naive.Trader` and add it to the `%State{}` struct as well as we will modify every callback to include that id inside log messages:

```
# /apps/naive/lib/naive/trader.ex
defmodule State do
  @enforce_keys [
    :id,
    ...
  ]
  defstruct [
    :id,
    ...
  ]
end

...

def init(%State{id: id, symbol: symbol} = state) do
  ...

  Logger.info("Initializing new trader(#{id}) for #{symbol}")

  ...
end
```

```
def handle_info(
      %TradeEvent{price: price},
      %State{
        id: id,
        ...
      } = state
    ) do
  ...

  Logger.info(
    "The trader(#{id}) is placing a BUY order " <>
      "for #{symbol} @ #{price}, quantity: #{quantity}"
  )

  ...
end

def handle_info(
      %TradeEvent{
        buyer_order_id: order_id
      },
      %State{
        id: id,
        ...
      } = state
    ) do
  ...
      Logger.info(
        "The trader(#{id}) is placing a SELL order for " <>
          "#{symbol} @ #{sell_price}, quantity: #{quantity}."
      )
      ...
      Logger.info("Trader's(#{id} #{symbol} BUY order got partially filled")
      ...
end

def handle_info(
      %TradeEvent{
        seller_order_id: order_id
      },
      %State{
        id: id,
```

```
          ...
        } = state
      ) do
    ...
      Logger.info("Trader(#{id}) finished trade cycle for #{symbol}")
      ...
      Logger.info("Trader's(#{id} #{symbol} SELL order got partially filled")
      ...
  end

  def handle_info(
        %TradeEvent{
          price: current_price
        },
        %State{
          id: id,
          ...
        } = state
      ) do
      ...
      Logger.info("Rebuy triggered for #{symbol} by the trader(#{id})")
      ...
  end
```

That finishes the implementation part - we should now be able to test the implementation and see a dynamically growing number of traders for our strategy based on price movement.

9.6 Test the implementation

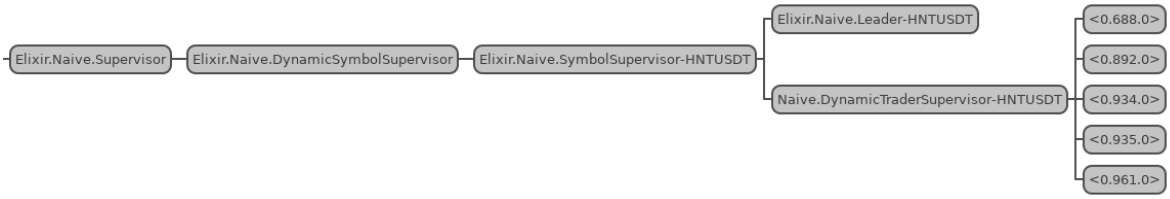
Let's start an iEx session and open the `:observer`(inside go to "Applications" tab and click on `naive` from the list of the left) so we will be able to see how the number of traders is growing as well as PIDs are changing which means that they are finishing the full trades:

```
$ iex -S mix
...
iex(1)> :observer.start()
...
iex(2)> Naive.start_trading("HNTUSDT")
10:22:05.018 [info]  The trader(1616754009963) is placing a BUY order for HNTUSDT @ 8.175,
quantity: 2.446
10:22:11.665 [info]  Rebuy triggered for HNTUSDT by the trader(1616754009963)
10:22:11.665 [info]  Starting new trader for HNTUSDT
10:22:11.665 [info]  Initializing new trader(1616754131665) for HNTUSDT
```

```
10:22:11.665 [info]  The trader(1616754009963) is placing a SELL order for HNTUSDT @ 8.181,
quantity: 2.446.
10:22:11.665 [info]  The trader(1616754131665) is placing a BUY order for HNTUSDT @ 8.157,
quantity: 2.451
10:22:58.339 [info]  Trader(1616754009963) finished trade cycle for HNTUSDT
10:22:58.339 [info]  HNTUSDT trader finished trade - restarting
10:22:58.339 [info]  Initializing new trader(1616754178339) for HNTUSDT
10:22:58.339 [info]  The trader(1616754178339) is placing a BUY order for HNTUSDT @ 8.212,
quantity: 2.435
10:23:13.232 [info]  Rebuy triggered for HNTUSDT by the trader(1616754178339)
10:23:13.232 [info]  Starting new trader for HNTUSDT
10:23:13.232 [info]  Initializing new trader(1616754193232) for HNTUSDT
10:23:13.232 [info]  The trader(1616754178339) is placing a SELL order for HNTUSDT @ 8.218,
quantity: 2.435.
10:23:31.120 [info]  The trader(1616754193232) is placing a BUY order for HNTUSDT @ 8.194,
quantity: 2.44
10:23:31.121 [info]  Trader(1616754178339) finished trade cycle for HNTUSDT
10:23:31.122 [info]  HNTUSDT trader finished trade - restarting
10:23:31.122 [info]  Initializing new trader(1616754211122) for HNTUSDT
10:24:31.891 [info]  The trader(1616754211122) is placing a BUY order for HNTUSDT @ 8.198,
quantity: 2.439
10:25:24.155 [info]  The trader(1616754211122) is placing a SELL order for HNTUSDT @ 8.204,
quantity: 2.439.
10:25:24.155 [info]  The trader(1616754193232) is placing a SELL order for HNTUSDT @ 8.2,
quantity: 2.44.
10:25:24.155 [info]  Rebuy triggered for HNTUSDT by the trader(1616754211122)
10:25:24.155 [info]  Starting new trader for HNTUSDT
10:25:24.156 [info]  Initializing new trader(1616754324155) for HNTUSDT
10:25:24.156 [info]  Rebuy triggered for HNTUSDT by the trader(1616754193232)
10:25:24.156 [info]  The trader(1616754324155) is placing a BUY order for HNTUSDT @ 8.183,
quantity: 2.444
10:25:24.156 [info]  Starting new trader for HNTUSDT
10:25:24.156 [info]  Initializing new trader(1616754324156) for HNTUSDT
10:25:24.156 [info]  The trader(1616754324156) is placing a BUY order for HNTUSDT @ 8.176,
quantity: 2.446
10:25:24.156 [info]  The trader(1616754324155) is placing a SELL order for HNTUSDT @ 8.189,
quantity: 2.444.
10:25:37.527 [info]  Trader(1616754324155) finished trade cycle for HNTUSDT
10:25:37.528 [info]  HNTUSDT trader finished trade - restarting
10:25:37.528 [info]  Initializing new trader(1616754337528) for HNTUSDT
10:25:37.528 [info]  The trader(1616754337528) is placing a BUY order for HNTUSDT @ 8.192,
quantity: 2.441
10:25:37.530 [info]  Trader(1616754211122) finished trade cycle for HNTUSDT
```

```
10:25:37.530 [info]  Trader(1616754193232) finished trade cycle for HNTUSDT
10:25:37.530 [info]  HNTUSDT trader finished trade - restarting
10:25:37.530 [info]  Initializing new trader(1616754337530) for HNTUSDT
10:25:37.530 [info]  HNTUSDT trader finished trade - restarting
10:25:37.530 [info]  Initializing new trader(1616754337530) for HNTUSDT
10:25:40.015 [info]  Rebuy triggered for HNTUSDT by the trader(1616754337528)
10:25:40.015 [info]  The trader(1616754337530) is placing a BUY order for HNTUSDT @ 8.179,
quantity: 2.445
10:25:40.015 [info]  All traders already started for HNTUSDT
```

And our observer shows the following:



We can clearly see that our strategy dynamically scaled from 1 to 5 parallel traders and they were going through different trading steps without any problems - I think that's really cool to see and it wasn't difficult to achieve in Elixir.

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

Source code for this chapter can be found at Github

Chapter 10

Fine-tune trading strategy per symbol

10.1 Objectives

- describe and design the required functionality
- add docker to project
- set up `ecto` inside the `naive` app
- create and migrate the DB
- seed symbols' settings
- update the `Naive.Leader` to fetch settings

10.2 Describe and design the required functionality

At this moment the settings of our naive strategy are hardcoded inside the `Naive.Leader`:

```
# /apps/naive/lib/naive/leader.ex
...
defp fetch_symbol_settings(symbol) do
  symbol_filters = fetch_symbol_filters(symbol)

  Map.merge(
    %{
      symbol: symbol,              # <=
      chunks: 5,                   # <=
      budget: 100,                 # <=
      # -0.01% for quick testing   # <=
      buy_down_interval: "0.0001", # <= all of those settings
      # -0.12% for quick testing   # <=
      profit_interval: "-0.0012",  # <=
      rebuy_interval: "0.001"      # <=
```

```
      },
      symbol_filters
    )
  end
  ...
```

The problem about those is that they are hardcoded and there's no flexibility to define them per symbol at the moment.

In this chapter, we will move them out from this file into the Postgres database.

10.3 Add docker to project

The requirements for this section are `docker` and `docker-compose` installed in your system.

Inside the main directory of our project create a new file called `docker-compose.yml` and fill it with the below details:

```
# /docker-compose.yml
version: "3.2"
services:
  db:
    image: postgres:latest
    restart: always
    environment:
      POSTGRES_PASSWORD: "hedgehogSecretPassword"
    ports:
      - 5432:5432
    volumes:
      - ../postgres-data:/var/lib/postgresql/data
```

If you are new to docker here's the gist of what the above will do:

- it will start a single service called "db"
- "db" service will use the `latest` version of the `postgres` (image) inside the docker container (`latest` version as tagged per https://hub.docker.com/_/postgres/)
- we map TCP port 5432 in the container to port 5432 on the Docker host(format container_port:hosts_port)
- we set up environmental variable inside the docker container that will be used by the Postgres app as a password for the default (`postgres`) user
- `volumes` option maps the directory from inside of the container to the host. This way we will keep the state of the database between restarts.

We can now start the service using `docker-compose`:

```
$ docker-compose up -d
Creating hedgehog_db_1 ... done
```

To validate that it works we can run:

```
$ docker ps -a
CONTAINER ID   IMAGE             COMMAND                  CREATED         STATUS
PORTS                     NAMES
98558827b80b   postgres:latest   "docker-entrypoint.sh"   4 seconds ago   Up 4 seconds
0.0.0.0:5432->5432/tcp   hedgehog_db_1
```

10.4 Set up `ecto` inside the `naive` app

Let's start by adding database access to the `naive` application. The first step is to add the Ecto module together with the Postgrex ecto's driver package to the `deps` function inside the `mix.exs` file. As we are going to use Enums inside Postgres, we need to add the EctoEnum module as well:

```
  # /apps/naive/mix.exs
  defp deps do
    [
      {:binance, "~> 1.0"},
      {:binance_mock, in_umbrella: true},
      {:decimal, "~> 2.0"},
      {:ecto_sql, "~> 3.0"},     # <= New line
      {:ecto_enum, "~> 1.4"},    # <= New line
      {:phoenix_pubsub, "~> 2.0"},
      {:postgrex, ">= 0.0"},     # <= New line
      {:streamer, in_umbrella: true}
    ]
  end
```

Remember about installing those deps using:

```
$ mix deps.get
```

We can now use the ecto generator to add an the ecto repository to the Naive application:

```
$ cd apps/naive
$ mix ecto.gen.repo -r Naive.Repo
* creating lib/naive
* creating lib/naive/repo.ex
```

```
* updating ../../config/config.exs
Don't forget to add your new repo to your supervision tree
(typically in lib/naive/application.ex):

    {Naive.Repo, []}

And to add it to the list of Ecto repositories in your
configuration files (so Ecto tasks work as expected):

    config :naive,
      ecto_repos: [Naive.Repo]
```

Back to the IDE, the generator updated our `config/config.exs` file with the default access details to the database, we need to modify them to point to our Postgres docker instance as well as add a list of ecto repositories for our naive app (as per instruction above):

```
# /config/config.exs
config :naive,                # <= added line
  ecto_repos: [Naive.Repo],   # <= added line
  binance_client: BinanceMock # <= merged from existing config

config :naive, Naive.Repo,
  database: "naive",                  # <= updated
  username: "postgres",               # <= updated
  password: "hedgehogSecretPassword", # <= updated
  hostname: "localhost"
```

Here we can use `localhost` as inside the `docker-compose.yml` file we defined port forwarding from the container to the host(Postgres is available at localhost:5432). We also merged the existing `binance_client` setting together with the new `ecto_repos` setting.

The last step to be able to communicate with the database using `Ecto` will be to add the `Naive.Repo` module(created by generator) to the children list of the `Naive.Application`:

```
# /apps/naive/lib/naive/application.ex
...
  def start(_type, _args) do
    children = [
      {Naive.Repo, []}, # <= added line
      {
        DynamicSupervisor,
        strategy: :one_for_one, name: Naive.DynamicSymbolSupervisor
      }
    ]
    ...
```

10.5 Create and migrate the DB

We can now create a new naive database using the `mix` tool, after that we will be able to generate a migration file that will create the `settings` table:

```
$ mix ecto.create -r Naive.Repo
The database for Naive.Repo has been created
$ cd apps/naive
$ mix ecto.gen.migration create_settings
* creating priv/repo/migrations
* creating priv/repo/migrations/20210202223209_create_settings.exs
```

We can now copy the current hardcoded settings from the `Naive.Leader` module and use them as a column list of our new `settings` table. All of the below alterations need to be done inside the `change` function of our migration file:

```
# /apps/naive/priv/repo/migrations/20210202223209_create_settings.exs
...
  def change do
    create table(:settings) do
      add(:symbol, :text, null: false)
      add(:chunks, :integer, null: false)
      add(:budget, :decimal, null: false)
      add(:buy_down_interval, :decimal, null: false)
      add(:profit_interval, :decimal, null: false)
      add(:rebuy_interval, :decimal, null: false)
    end
  end
```

At this moment we just copied the settings and converted them to columns using the `add` function. We need now to take care of the `id` column. We need to pass `primary_key: false` to the `create table` macro to stop it from creating the default integer-based `id` column. Instead of that we will define the `id` column ourselves with `:uuid` type and pass a flag that will indicate that it's the primary key of the `settings` table:

```
# /apps/naive/priv/repo/migrations/20210202223209_create_settings.exs
...
    create table(:settings, primary_key: false) do
      add(:id, :uuid, primary_key: true)
      ...
```

We will also add create and update timestamps that come as a bundle when using the `timestamps()` function inside the `create table` macro:

```
# /apps/naive/priv/repo/migrations/20210202223209_create_settings.exs
...
    create table(...) do
      ...

      timestamps() # <= both create and update timestamps
    end
    ...
```

We will add a unique index on the symbol column to avoid any possible duplicates:

```
# /apps/naive/priv/repo/migrations/20210202223209_create_settings.exs
...
    create table(...) do
      ...
    end

    create(unique_index(:settings, [:symbol]))
  end
...
```

We will now add the `status` field which will be an Enum. It will be defined inside a separate file and `alias`'ed from our migration, this way we will be able to use it from within the migration and the inside the `lib` code. First, we will apply changes to our migration and then we will move on to creating the Enum module. Here's the full implementation of migration for reference:

```
# /apps/naive/priv/repo/migrations/20210202223209_create_settings.exs
defmodule Naive.Repo.Migrations.CreateSettings do
  use Ecto.Migration

  alias Naive.Schema.TradingStatusEnum

  def change do
    TradingStatusEnum.create_type()

    create table(:settings, primary_key: false) do
      add(:id, :uuid, primary_key: true)
      add(:symbol, :text, null: false)
      add(:chunks, :integer, null: false)
      add(:budget, :decimal, null: false)
      add(:buy_down_interval, :decimal, null: false)
```

```
      add(:profit_interval, :decimal, null: false)
      add(:rebuy_interval, :decimal, null: false)
      add(:status, TradingStatusEnum.type(), default: "off", null: false)

      timestamps()
    end

    create(unique_index(:settings, [:symbol]))
  end
end
```

That finishes our work on the migration file. We will now focus on `TradingStatusEnum` implementation. First, we need to create a `schema` directory inside the `apps/naive/lib/naive` directory and file called `trading_status_enum.ex` and place below logic (defining the enum) in it:

```
# /apps/naive/lib/naive/schema/trading_status_enum.ex
import EctoEnum

defenum(Naive.Schema.TradingStatusEnum, :trading_status, [:on, :off])
```

We used the `defenum` macro from the `ecto_enum` module to define our enum. It's interesting to point out that we didn't need to define a new module as `defenum` macro takes care of that for us.

Let's run the migration to create the table, unique index, and the enum:

```
$ mix ecto.migrate
00:51:16.757 [info]  == Running 20210202223209 Naive.Repo.Migrations.CreateSettings.change/0
forward
00:51:16.759 [info]  execute "CREATE TYPE public.trading_status AS ENUM ('on', 'off')"
00:51:16.760 [info]  create table settings
00:51:16.820 [info]  create index settings_symbol_index
00:51:16.829 [info]  == Migrated 20210202223209 in 0.0s
```

We can now create a schema file for the `settings` table so inside the `/apps/naive/lib/naive/schema` create a file called `settings.ex`. We will start with a skeleton implementation of schema file together with the copied list of columns from the migration and convert to `ecto`'s types using it's docs:

```
# /apps/naive/lib/naive/schema/settings.ex
defmodule Naive.Schema.Settings do
  use Ecto.Schema

  alias Naive.Schema.TradingStatusEnum

  @primary_key {:id, :binary_id, autogenerate: true}
```

```
  schema "settings" do
    field(:symbol, :string)
    field(:chunks, :integer)
    field(:budget, :decimal)
    field(:buy_down_interval, :decimal)
    field(:profit_interval, :decimal)
    field(:rebuy_interval, :decimal)
    field(:status, TradingStatusEnum)

    timestamps()
  end
end
```

10.6 Seed symbols' settings

So we have all the pieces of implementation to be able to create DB, migrate the `settings` table, and query it using Ecto. To be able to drop the hardcoded settings from the `Naive.Leader` we will need to fill our database with the "default" setting for each symbol. To achieve that we will define default settings inside the `config/config.exs` file and we will create a seed script that will fetch all pairs from Binance and insert a new config row inside DB for each one.

Let's start by adding those default values to the config file(we will merge them into the structure defining `binance_client` and `ecto_repos`):

```
# config/config.exs
config :naive,
  ecto_repos: [Naive.Repo],
  binance_client: BinanceMock,
  trading: %{
    defaults: %{
      chunks: 5,
      budget: 1000,
      buy_down_interval: "0.0001",
      profit_interval: "-0.0012",
      rebuy_interval: "0.001"
    }
  }
```

Moving on to the seeding script, we need to create a new file called `seed_settings.exs` inside the `/apps/naive/lib/naive/priv/` directory. Let's start by aliasing the required modules and requiring the `Logger`:

```
# /apps/naive/priv/seed_settings.exs
require Logger

alias Decimal
alias Naive.Repo
alias Naive.Schema.Settings
```

Next, we will get the Binance client from the config:

```
# /apps/naive/priv/seed_settings.exs
...
binance_client = Application.get_env(:naive, :binance_client)
```

Now, it's time to fetch all the symbols(pairs) that Binance supports:

```
# /apps/naive/priv/seed_settings.exs
...
Logger.info("Fetching exchange info from Binance to create trading settings")

{:ok, %{symbols: symbols}} = binance_client.get_exchange_info()
```

Now we need to fetch default trading settings from the config file as well as the current timestamp:

```
# /apps/naive/priv/seed_settings.exs
...
%{
  chunks: chunks,
  budget: budget,
  buy_down_interval: buy_down_interval,
  profit_interval: profit_interval,
  rebuy_interval: rebuy_interval
} = Application.get_env(:naive, :trading).defaults

timestamp = NaiveDateTime.utc_now()
  |> NaiveDateTime.truncate(:second)
```

We will use the default settings for all rows to be able to insert data into the database. Normally we wouldn't need to set `inserted_at` and `updated_at` fields as Ecto would generate those values for us when using `Repo.insert/2` but we won't be able to use this functionality as it takes a *single* record at the time. We will be using `Repo.insert_all/3` which is a bit more low-level function without those nice features like filling timestamps(sadly). Just to be crystal clear - `Repo.insert/2` takes *at least a couple of seconds*(on my machine) for 1000+ symbols currently supported by Binance, on the other hand `Repo.insert_all/3`, will insert all of them in a couple of hundred milliseconds.

As our structs will differ by only the `symbol` column we can first create a full struct that will serve as a template:

```
# /apps/naive/priv/seed_settings.exs
...
base_settings = %{
  symbol: "",
  chunks: chunks,
  budget: Decimal.new(budget),
  buy_down_interval: Decimal.new(buy_down_interval),
  profit_interval: Decimal.new(profit_interval),
  rebuy_interval: Decimal.new(rebuy_interval),
  status: "off",
  inserted_at: timestamp,
  updated_at: timestamp
}
```

We will now map each of the retrieved symbols and inject them to the `base_settings` structs and pushing all of those to the `Repo.insert_all/3` function:

```
# /apps/naive/priv/seed_settings.exs
...
Logger.info("Inserting default settings for symbols")

maps = symbols
  |> Enum.map(&(%{base_settings | symbol: &1["symbol"]}))

{count, nil} = Repo.insert_all(Settings, maps)

Logger.info("Inserted settings for #{count} symbols")
```

10.7 Update the `Naive.Leader` to fetch settings

The final step will be to update the `Naive.Leader` to fetch the settings from the database. At the top of the module add the following:

```
# /apps/naive/lib/naive/leader.ex
  ...
  alias Naive.Repo
  alias Naive.Schema.Settings
  ...
```

Now we need to modify the `fetch_symbol_settings/1` to fetch settings from DB instead of the hardcoded map. We will use `Repo.get_by!/3` as we are unable to trade without settings. The second trick used here is `Map.from_struct/1`

that is required here as otherwise result would become the `Naive.Schema.Settings` struct(this would cause problems further down the line as we are iterating on the returned map and would get the `protocol Enumerable not implemented for %Naive.Schema.Settings` error):

```
# /apps/naive/lib/naive/leader.ex
  ...
  defp fetch_symbol_settings(symbol) do
    symbol_filters = fetch_symbol_filters(symbol)
    settings = Repo.get_by!(Settings, symbol: symbol)

    Map.merge(
      symbol_filters,
      settings |> Map.from_struct()
    )
  end
  ...
```

We can now run the seeding script to fill our database with the default settings:

```
$ cd apps/naive
$ mix run priv/seed_settings.exs
18:52:29.341 [info]  Fetching exchange info from Binance to create trading settings
18:52:31.571 [info]  Inserting default settings for symbols
18:52:31.645 [info]  Inserted settings for 1276 symbols
```

We can verify that records were indeed inserted into the database by connecting to it using the `psql` application:

```
$ psql -Upostgres -hlocalhost
Password for user postgres: # <= use 'postgres' password here
...
postgres=# \c naive
You are now connected to database "naive" as user "postgres".
naive=# \x
Expanded display is on.
naive=# SELECT * FROM settings;
-[ RECORD 1 ]-----+-------------------------------------
id                | 159c8f32-d571-47b2-b9d7-38bb42868043
symbol            | ETHUSDT
chunks            | 5
budget            | 1000
buy_down_interval | 0.0001
profit_interval   | -0.0012
rebuy_interval    | 0.001
status            | off
```

```
inserted_at       | 2021-02-02 18:52:31
updated_at        | 2021-02-02 18:52:31

# press arrows to scroll, otherwise press `q`

naive=# SELECT COUNT(*) FROM settings;
-[ RECORD 1 ]
count | 1276

naive=# \q # <= to close the `psql`
```

That confirms that there are 1276 settings inside the database that will allow us to continue trading which we can check by running our app inside the IEx(from the main project's directory):

```
$ iex -S mix
...
iex(1)> Naive.start_trading("NEOUSDT")
19:20:02.936 [info]  Starting new supervision tree to trade on NEOUSDT
{:ok, #PID<0.378.0>}
19:20:04.584 [info]  Initializing new trader(1612293637000) for NEOUSDT
```

The above log messages confirm that the `Naive.Leader` was able to fetch settings from the database that were later put into the `Naive.Trader`'s state and passed to it.

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

Source code for this chapter can be found at Github

Chapter 11

Supervise and autostart streaming

11.1 Objectives

- describe and design the required functionality
- register the `Streamer.Binance` processes with names
- set up `ecto` inside the `streamer` app
- create and migrate the db
- seed default settings
- implement the supervision tree and start streaming functionality
- implement the stop functionality
- implement the autostart streaming functionality
- test the implementation

11.2 Describe and design the required functionality

At this moment there's no supervision around the streamer processes, whenever an error would occur inside the `Streamer.Binance` process, it will die and never come back up.

That's less than perfect, but we can use supervisors to the rescue.

We will create a new `Streamer.DynamicStreamerSupervisor` module inside our `streamer` application that will supervise the `Streamer.Binance` processes.

Next, we will consider a list of functionalities that we would like it to provide:

- start streaming. This will require a new `Streamer.Binance` process started under the `Streamer.DynamicStreamerSupervisor`. We will put logic responsible for starting that process inside the `Streamer.DynamicStreamerSupervisor` module.
- stop streaming. To be able to stop the `Streamer.Binance` process streaming on a specific symbol, we will need to know that process' PID based only on symbol string(ie. "ETHUSDT"). To make that possible, we will need to register every `Streamer.Binance` process with a name that we will be able to "reverse-engineer" based only on symbol string for example: `:"#{__MODULE__}-#{symbol}"`

- autostart streaming. At the start of streaming on a symbol, we should persist that action as a symbol's streaming setting inside the database. We will need to generate a new Ecto.Repo, configure, create and migrate DB (just like in the last chapter for the `naive` app) to be able to retrieve that list. We will write a logic that will fetch settings of the symbols, autostart the ones that are enabled and place all that logic inside the `Streamer.DynamicStreamerSupervisor` module. We will introduce a Task child process that will utilize the logic from the `Streamer.DynamicStreamerSupervisor` to fetch those enabled symbols and start `Streamer.Binance` processes on startup - we will describe all of this separately in its section in this chapter.

11.3 Register the `Streamer.Binance` processes with names

To be able to perform start/stop streaming on a symbol we will first need to be able to figure out the PID of the `Streamer.Binance` process that is streaming that symbol.

The first change that we need to apply will be to register `Streamer.Binance` processes with names by passing the 4th argument to the `WebSockex.start_link/4` function:

```
# /apps/streamer/lib/streamer/binance.ex
def start_link(symbol) do
  Logger.info(
    "Binance streamer is connecting to websocket " <>
      "stream for #{symbol} trade events"
  )

  WebSockex.start_link(
    "#{@stream_endpoint}#{String.downcase(symbol)}@trade", # <= lowercase symbol
    __MODULE__,
    nil,
    name: :"#{__MODULE__}-#{symbol}" # <= uppercase symbol
  )
end
```

Few things worth mention here:

- we are getting the uppercase symbol but inside the URL we need to use a lowercase symbol so we will introduce a new separate variable to be used in the URL
- we are registering the process using the uppercase symbol so the name will remain consistent with the `naive` application's processes
- to register process we are sending keyword list as the 4th argument to custom `start_link/4` function of `WebSockex` module (link to source - again, no need to be afraid of reading the source code in Elixir, that's the beauty of it)

11.4 Set up `ecto` inside the `streamer` app

In the same fashion as in the last chapter, we will need to set up the database inside the `streamer` app. We will use the same Postgres server(docker container) that we've set up inside docker in the last chapter, just a separate database, so there's no need to update the `docker-compose.yml` file.

As previously the first step will be to add the `ecto` modules and Postgres related packages into `deps` inside the `mix.exs` file of the `streamer` app. Additionally, we will add the `binance` module that we will use to fetch all symbols supported by the exchange(to generate default settings as we've done for the `naive` application. We are unable to use the `BinanceMock` as this will cause the circular dependency [Binance Mock depends on the streamer app]):

```
  # /apps/streamer/mix.exs
  ...
  defp deps do
    [
      {:binance, "~> 1.0"}, # <= used to retrieve symbols list(exchangeInfo)
      {:ecto_sql, "~> 3.0"},  # <= added dependency
      {:ecto_enum, "~> 1.4"}, # <= added dependency
      {:jason, "~> 1.2"},
      {:phoenix_pubsub, "~> 2.0"},
      {:postgrex, ">= 0.0"},  # <= added dependency
      {:websockex, "~> 0.4"}
    ]
  end
```

Run `mix deps.get` to install new dependencies.

We can now use `ecto` generator to add an `ecto` repository to the Streamer application:

```
$ cd apps/streamer
$ mix ecto.gen.repo -r Streamer.Repo
* creating lib/streamer
* creating lib/streamer/repo.ex
* updating ../../config/config.exs
...
```

Update the config to match access details to Postgres' docker instance:

```
# /config/config.exs
config :streamer,             # <= added line
  ecto_repos: [Streamer.Repo] # <= added line

config :streamer, Streamer.Repo,
  database: "streamer",               # <= database updated
  username: "postgres",               # <= username updated
```

```
  password: "hedgehogSecretPassword", # <= password updated
  hostname: "localhost"
```

The last step will be to update the `children` list of the `Streamer.Application` module:

```
# /apps/streamer/lib/streamer/application.ex
...
  def start(_type, _args) do
    children = [
      {Streamer.Repo, []}, # <= repo added
      {
        Phoenix.PubSub,
        name: Streamer.PubSub, adapter_name: Phoenix.PubSub.PG2
      }
    ]
    ...
```

11.5 Create and migrate the db

We can now create a new streamer database using the `mix` tool, after that we will be able to generate a migration file that will create the `settings` table:

```
$ mix ecto.create -r Streamer.Repo
The database for Streamer.Repo has been created
$ cd apps/streamer
$ mix ecto.gen.migration create_settings
* creating priv/repo/migrations
* creating priv/repo/migrations/20210203184805_create_settings.exs
```

We can safely start just with `id`, `symbol` and `status` columns, where the last one will follow the same enum idea from the previous chapter:

```
# /apps/streamer/priv/repo/migrations/20210203184805_create_settings.exs
defmodule Streamer.Repo.Migrations.CreateSettings do
  use Ecto.Migration

  alias Streamer.Schema.StreamingStatusEnum

  def change do
    StreamingStatusEnum.create_type()

    create table(:settings, primary_key: false) do
      add(:id, :uuid, primary_key: true)
```

```
      add(:symbol, :text, null: false)
      add(:status, StreamingStatusEnum.type(), default: "off", null: false)

      timestamps()
    end

    create(unique_index(:settings, [:symbol]))
  end
end
```

That finishes our work on the migration file, we need to add the `StreamingStatusEnum` in the same way as in the last chapter (create a `schema` directory inside the `apps/streamer/lib/streamer` directory and a new file called `streaming_status_enum.ex` and place below logic (defining the enum) in it:

```
# /apps/streamer/lib/streamer/schema/streaming_status_enum.ex
import EctoEnum

defenum(Streamer.Schema.StreamingStatusEnum, :streaming_status, [:on, :off])
```

Let's run the migration to create the table, unique index, and the enum:

```
$ mix ecto.migrate
21:31:56.850 [info]  == Running 20210203184805
Streamer.Repo.Migrations.CreateSettings.change/0 forward
21:31:56.850 [info]  execute "CREATE TYPE public.streaming_status AS ENUM ('on', 'off')"
21:31:56.851 [info]  create table settings
21:31:56.912 [info]  create index settings_symbol_index
21:31:56.932 [info]  == Migrated 20210203184805 in 0.0s
```

We can now create a schema file for the `settings` table. Inside the `/apps/streamer/lib/streamer/schema` directory create a file called `settings.ex`:

```
# /apps/streamer/lib/streamer/schema/settings.ex
defmodule Streamer.Schema.Settings do
  use Ecto.Schema

  alias Streamer.Schema.StreamingStatusEnum

  @primary_key {:id, :binary_id, autogenerate: true}

  schema "settings" do
    field(:symbol, :string)
    field(:status, StreamingStatusEnum)
```

```
    timestamps()
  end
end
```

We are now ready to query the table but first, we need to insert the default settings into the database.

11.6 Seed default settings

As with the settings inside the `naive` application, we will fetch all symbols from binance and bulk insert them into the database.

First let's create a new file called `seed_settings.exs` inside the `apps/streamer/priv` directory. As this file is nearly the same as the one from the last chapter I will skip the lengthy explanation - this is the script:

```
# /apps/streamer/priv/seed_settings.exs
require Logger

alias Decimal
alias Streamer.Repo
alias Streamer.Schema.Settings

Logger.info("Fetching exchange info from Binance to create streaming settings")

{:ok, %{symbols: symbols}} = Binance.get_exchange_info()

timestamp = NaiveDateTime.utc_now()
  |> NaiveDateTime.truncate(:second)

base_settings = %{
  symbol: "",
  status: "off",
  inserted_at: timestamp,
  updated_at: timestamp
}

Logger.info("Inserting default settings for symbols")

maps = symbols
  |> Enum.map(&(%{base_settings | symbol: &1["symbol"]}))

{count, nil} = Repo.insert_all(Settings, maps)

Logger.info("Inserted settings for #{count} symbols")
```

Don't forget to run the seeding script before progressing forward:

```
$ cd apps/streamer
$ mix run priv/seed_settings.exs
22:03:46.675 [info]  Fetching exchange info from Binance to create streaming settings
22:03:51.386 [info]  Inserting default settings for symbols
22:03:51.448 [info]  Inserted settings for 1277 symbols
```

11.7 Implement the supervision tree and start streaming functionality

Let's start by creating a new file called `dynamic_streamer_supervisor.ex` inside the `/apps/streamer/lib/streamer` directory. Let's start with a default implementation from the docs (updated with correct module and process names):

```
# /apps/streamer/lib/streamer/dynamic_streamer_supervisor.ex
defmodule Streamer.DynamicStreamerSupervisor do
  use DynamicSupervisor

  def start_link(init_arg) do
    DynamicSupervisor.start_link(__MODULE__, init_arg, name: __MODULE__)
  end

  def init(_init_arg) do
    DynamicSupervisor.init(strategy: :one_for_one)
  end
end
```

Next, we will add the `start_streaming/1` function at the bottom of the `Streamer.DynamicStreamerSupervisor` module:

```
# /apps/streamer/lib/streamer/dynamic_streamer_supervisor.ex
  ...
  def start_streaming(symbol) when is_binary(symbol) do
    case get_pid(symbol) do
      nil ->
        Logger.info("Starting streaming on #{symbol}")
        {:ok, _settings} = update_streaming_status(symbol, "on")
        {:ok, _pid} = start_streamer(symbol)

      pid ->
        Logger.warn("Streaming on #{symbol} already started")
        {:ok, _settings} = update_streaming_status(symbol, "on")
        {:ok, pid}
```

```
    end
  end
```

To unpack above - we are checking is there a streamer process already running for the passed symbol and based on the result of that check, we either start the new streaming process(and update the symbol's settings) or just update the symbol's settings.

Inside the `start_streaming/1` function, we are using 3 helper functions that we need to add at the bottom of the file:

```
# /apps/streamer/lib/streamer/dynamic_streamer_supervisor.ex
  defp get_pid(symbol) do
    Process.whereis(:"Elixir.Streamer.Binance-#{symbol}")
  end

  defp update_streaming_status(symbol, status)
       when is_binary(symbol) and is_binary(status) do
    Repo.get_by(Settings, symbol: symbol)
    |> Ecto.Changeset.change(%{status: status})
    |> Repo.update()
  end

  defp start_streamer(symbol) do
    DynamicSupervisor.start_child(
      Streamer.DynamicStreamerSupervisor,
      {Streamer.Binance, symbol}
    )
  end
```

The above functions are quite self-explanatory, `get_pid/1` is a convenience wrapper, `update_streaming_status/2` will update the status field for the passed symbol, `start_streamer/1` will instruct the `Streamer.DynamicStreamerSupervisor` to start a new `Streamer.Binance` process with symbol passed as the first argument.

The last step to get the above function to work(and future ones in this module) would be to add a `require`, an `import`, and a few `alias`es at the top of the module:

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

  import Ecto.Query, only: [from: 2]

  alias Streamer.Repo
  alias Streamer.Schema.Settings
```

As we added a new `start_streaming/1` logic inside the `Streamer.DynamicStreamerSupervisor`, we need to replace the `start_streaming/1` implementation inside the `Streamer` module:

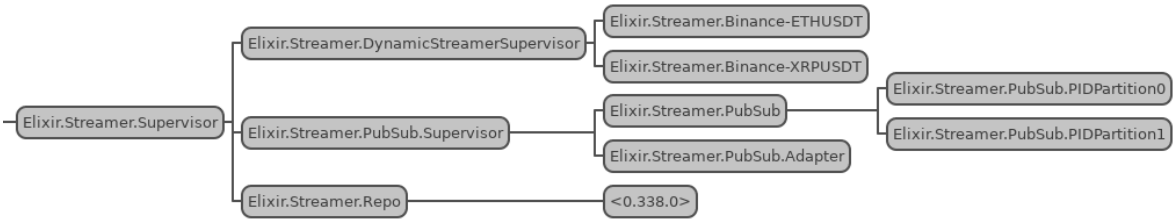
```
# /apps/streamer/lib/streamer.ex
  ...
  alias Streamer.DynamicStreamerSupervisor

  def start_streaming(symbol) do
    symbol
    |> String.upcase()
    |> DynamicStreamerSupervisor.start_streaming()
  end
```

As we don't need to put any logic inside the `Streamer.start_streaming/1` function, we can just delegate the call straight to the `Streamer.DynamicStreamerSupervisor` module.

The last step will be to append the `Streamer.DynamicStreamSupervisor` to the `children` list of the `Streamer.Application`:

```
  # /apps/streamer/lib/streamer/application.ex
  def start(_type, _args) do
    children = [
      ...
      {Streamer.DynamicStreamerSupervisor, []}
    ]
```

At this moment our supervision tree already works and all streamer processes are being monitored by the `Streamer.DynamicStreamerSupervisor`:



11.8 Implement the stop functionality

As we can see, we are now registering the `Streamer.Binance` processes with names that contain their symbols. We will be able to retrieve PIDs of those registered processes just by simply passing the symbol string(ie. "ETHUSDT") into `get_pid/1`, which will allow us to then request termination of those processes by the `Streamer.DynamicStreamerSupervisor`.

Let's write a `stop_streaming/1` logic inside the `Streamer.DynamicStreamerSupervisor` module(put it above first private function):

```
  # /apps/streamer/lib/streamer/dynamic_streamer_supervisor.ex
  def stop_streaming(symbol) when is_binary(symbol) do
    case get_pid(symbol) do
      nil ->
        Logger.warn("Streaming on #{symbol} already stopped")
        {:ok, _settings} = update_streaming_status(symbol, "off")

      pid ->
        Logger.info("Stopping streaming on #{symbol}")

        :ok =
          DynamicSupervisor.terminate_child(
            Streamer.DynamicStreamerSupervisor,
            pid
          )

        {:ok, _settings} = update_streaming_status(symbol, "off")
    end
  end
```

`stop_streaming/1` looks very similar to `start_streaming/1`, we are checking is there already a `Streamer.Binance` process registered for that symbol, and we either ask the `Streamer.DynamicStreamerSupervisor` to terminate it for us (using the `DynamicSupervisor.terminate_child/2` function + update the status) or just update the status to be `off`.

We need to update the `Streamer` module to provide the interface to stop streaming on a symbol:

```
# /apps/streamer/lib/streamer.ex
  ...
  def stop_streaming(symbol) do
    symbol
    |> String.upcase()
    |> DynamicStreamerSupervisor.stop_streaming()
  end
  ...
```

11.9 Implement the autostart streaming functionality

Currently, whenever we will shutdown the elixir app, settings persist in the database but streamers are not started on the next init.

To fix this, we will add `autostart_streaming/0` inside the `Streamer.DynamicStreamerSupervisor`.

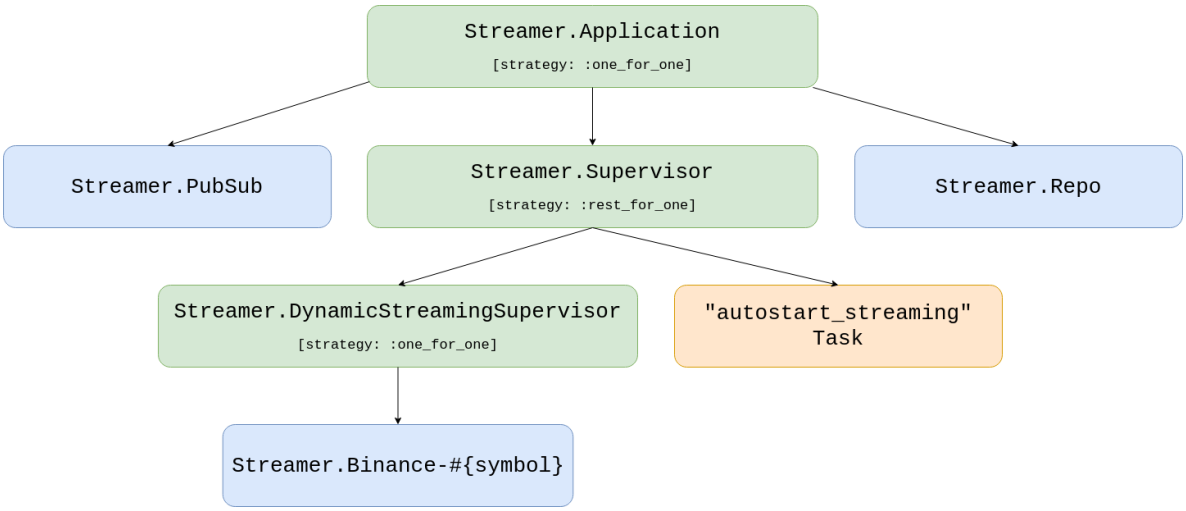
Note: It very important to differentiate between module and process. We will put our autostarting logic inside the *module* but the `Streamer.DynamicStreamerSupervisor` *process* won't run it.

We will introduce a new Task process that will execute all the autostarting logic.

That will cover the problem of the Supervisor executing too much business logic (as the Task will execute it), but how will we supervise them together? At init both will start, the `Streamer.DynamicStreamerSupervisor` first, and then Task will ask it to start multiple children and that will work fine. The problem occurs when the `Streamer.DynamicStreamerSupervisor` would die because of any reason. Currently, it's supervised using the `one_for_one` strategy(and the `Task` would be as well) which means that it will get started again by the `Streamer.Application` process but at that moment the "autostarting" `Task` won't be there anymore to start streaming on all enabled symbols.

We can clearly see that whenever the `Streamer.DynamicStreamerSupervisor` will die it needs to be started again *together* with the "autostart" `Task` and this won't fit our current `Streamer.Application`'s strategy.

In cases like those, a new level of supervision needs to be introduced that will have a different supervision strategy for those "coupled" processes. We will rename the process name of the `Streamer.Application` (which is currently registered as `Streamer.Supervisor`) to `Streamer.Application`. Then we will introduce the new `Streamer.Supervisor` module and register it under the same name. We will attach both `Streamer.DynamicStreamerSupervisor` and Task to the `Streamer.Supervisor` and assign it with the `rest_for_one` strategy which will restart the Task whenever `Streamer.DynamicStreamerSupervisor` would die:



Let's start by creating the `autostart_streaming/0` functionality inside the `Streamer.DynamicStreamerSupervisor`:

```
# /apps/streamer/lib/streamer/dynamic_streamer_supervisor.ex

# add below after the `init/1` function
def autostart_streaming do
  fetch_symbols_to_stream()
```

```
    |> Enum.map(&start_streaming/1)
  end

  # and this at the end of the module
  defp fetch_symbols_to_stream do
    Repo.all(
      from(s in Settings,
        where: s.status == "on",
        select: s.symbol
      )
    )
  end
```

`autostart_streaming/0` function fetches all symbols from the `settings` table with `status == "on"` then it passes them one by one into the `start_streaming/1` function using `Enum.map/2`.

We can now focus on referring to the above autostarting logic inside the new supervisor that we will create now. Let's start by creating a new file called `supervisor.ex` inside the `/apps/streamer/lib/streamer/` directory and fill it with default Supervisor implementation:

```
# /apps/streamer/lib/streamer/supervisor.ex
defmodule Streamer.Supervisor do # <= updated module name
  use Supervisor

  def start_link(init_arg) do
    Supervisor.start_link(__MODULE__, init_arg, name: __MODULE__)
  end

  def init(_init_arg) do
    children = [

    ]

    Supervisor.init(children, strategy: :one_for_one)
  end
end
```

We can now update the strategy to `rest_for_one`:

```
# /apps/streamer/lib/streamer/supervisor.ex
  def init(_init_arg) do
    ...
    Supervisor.init(children, strategy: :rest_for_one) # <= strategy updated
  end
```

The last step inside our new supervisor will be to add 2 children: `Streamer.DynamicStreamerSupervisor` and `Task` (that will autostart the symbol streamers):

```
# /apps/streamer/lib/streamer/supervisor.ex
  def init(_init_arg) do
    children = [
      {Streamer.DynamicStreamerSupervisor, []},
      {Task,
       fn ->
         Streamer.DynamicStreamerSupervisor.autostart_streaming()
       end}
    ]
    ...
  end
```

The final update in this chapter will be to replace the `Streamer.DynamicStreamerSupervisor` as one of the children inside the `Streamer.Application` module and update the name that the application process registers under:

```
# /apps/streamer/lib/streamer/application.ex
    ...
    children = [
      {Streamer.Repo, []},
      {
        Phoenix.PubSub,
        name: Streamer.PubSub, adapter_name: Phoenix.PubSub.PG2
      },
      {Streamer.Supervisor, []} # <= updated supervisor
    ]

    opts = [strategy: :one_for_one, name: Streamer.Application] # <= updated name
    ...
```

11.10 Test the implementation

Let's start an IEx session and call the `start_streaming/1` function twice for two different symbols and then exit using double Ctrl+c:

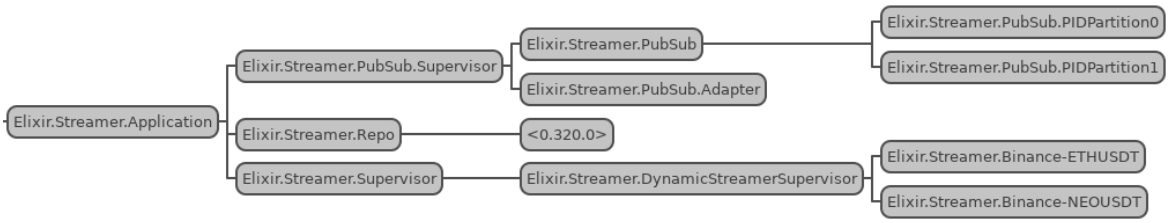
```
$ iex -S mix
...
iex(1)> Streamer.start_streaming("ethusdt")
18:51:39.809 [info]  Starting streaming on ETHUSDT
{:ok, #PID<0.370.0>}
iex(2)> Streamer.start_streaming("neousdt")
```

```
18:51:47.288 [info]  Starting streaming on NEOUSDT
{:ok, #PID<0.377.0>}
```

Now, open a *new* IEx session and look at the output:

```
$ iex -S mix
...
iex(1)>
18:53:48.920 [info]  Starting streaming on ETHUSDT
18:53:50.306 [info]  Starting streaming on NEOUSDT
```

We can also confirm that streamer processes are there by using `:observer.start()`:



Inside the same `iex` session run the following:

```
iex(5)> Streamer.stop_streaming("neousdt")
18:57:37.205 [info]  Stopping streaming on NEOUSDT
{:ok,
 %Streamer.Schema.Settings{
   ...
 }}
iex(6)> Streamer.stop_streaming("ethusdt")
18:57:51.553 [info]  Stopping streaming on ETHUSDT
{:ok,
 %Streamer.Schema.Settings{
   ...
 }}
```

Stop the IEx session and start a new one - streamers shouldn't be autostarted anymore.

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

Source code for this chapter can be found at Github

Chapter 12

Start, stop, shutdown and autostart trading

12.1 Objectives

- describe and design the required functionality
- (re-)implement the start trading functionality
- implement the stop trading functionality
- implement the autostart trading functionality
- implement the shutdown trading functionality
- test the implementation

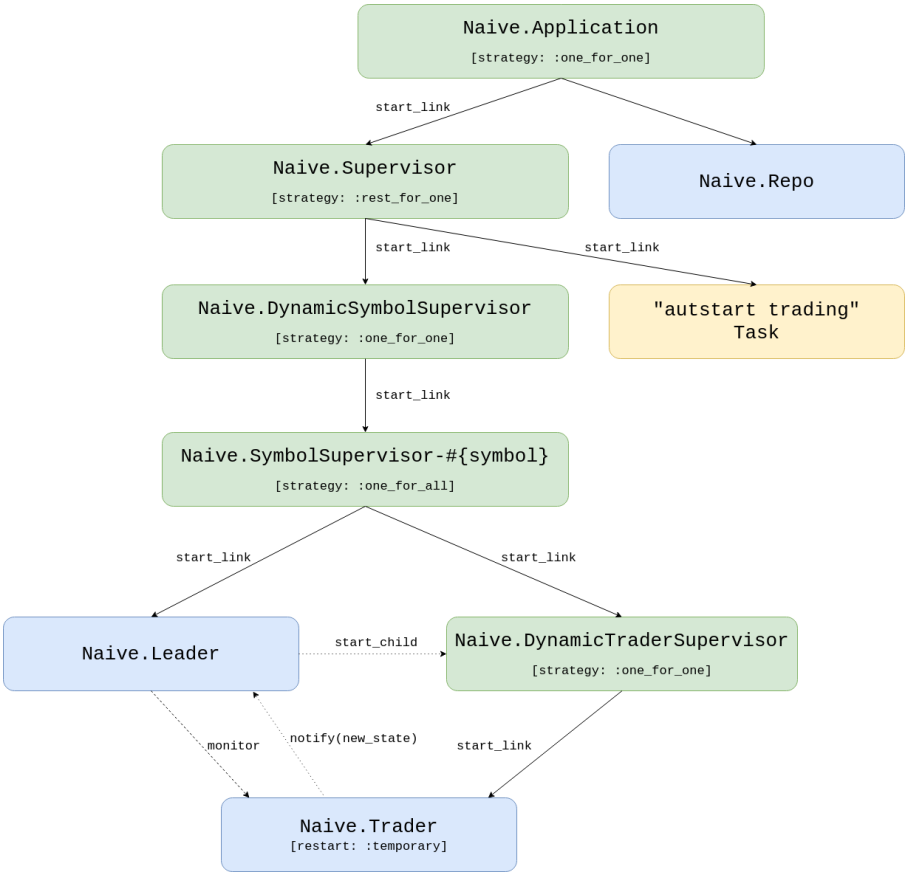
12.2 Describe and design the required functionality

In the 10th chapter, we've introduced the Postgres database inside the `naive` application together with the settings per symbol.

In this chapter, we will progress forward to provide additional trading functionality that will be similar to the functionality implemented in the last chapter for the `streaming` application:

- **stop trading** - as the `Naive.SymbolSupervisor` processes are registered with names that can be easily reverse engineered, we should be able to utilize the `Process.where_is/1` function to retrieve the PIDs and instruct the `Naive.DynamicSymbolSupervisor` to terminate those child processes. Again, we need to put that logic somewhere so we will implement the `Naive.DynamicSymbolSupervisor` as a full module using the `DynamicSupervisor` behavior.
- **start_trading** - as our `Naive.DynamicSymbolSupervisor` will now be a module we will be able to remove the `start_trading/1` implementation from the `Naive` module and reimplement it inside the `Naive.DynamicSymbolSupervisor` module. It will follow the same pattern of checking for PID, starting the `Naive.SymbolSupervisor` process and flipping the `status` flag inside the `settings` table's row for that symbol.

- **shutdown trading** - sometimes abruptly stopping trading won't be the best solution, it would be better to allow the `Naive.Trader` processes to finish their ongoing trades. To be able to do that we will need to inform the `Naive.Leader` process assigned to the symbol that the settings for that symbol have been updated and that should cause the `Naive.Leader` process to withhold starting new `Naive.Trader` processes and terminate the whole tree when the last trader will finish.
- **autostart trading** - this will be a very similar implementation to the one from the last chapter. It will require introducing a new supervisor(we will follow the same naming convention: rename `Naive.Application`'s registered process name to `Naive.Application`, create a new supervisor called `Naive.Supervisor`) and utilize the `Task` process to execute the autostarting logic.



12.3 (Re-)Implement the start trading functionality

To (re-)implement the `start_trading/1` functionality we will need to create a new file called `dynamic_symbol_supervisor.ex` inside the `/apps/naive/lib/naive` directory that will use the `DynamicSupervisor` behavior. Previously we have been using default `DynamicSupervisor` implementation(one of the children of the `Naive.Application` - to be substituted with the below module):

```
# /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
defmodule Naive.DynamicSymbolSupervisor do # <= module updated
  use DynamicSupervisor
```

```
  require Logger # <= Logger added

  import Ecto.Query, only: [from: 2] # <= added for querying

  alias Naive.Repo             # <= added for querying/updating
  alias Naive.Schema.Settings  # <= added for querying/updating

  def start_link(init_arg) do
    DynamicSupervisor.start_link(__MODULE__, init_arg, name: __MODULE__)
  end

  def init(_init_arg) do
    DynamicSupervisor.init(strategy: :one_for_one)
  end
end
```

The above code is a default implementation from the DynamicSupervisor docs with some additional imports, require and aliases as we will use them in this chapter.

Our `start_trading/1` implementation is almost the same as one for the `streamer` application from the last chapter:

```
# /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
  ...
  def start_trading(symbol) when is_binary(symbol) do
    symbol = String.upcase(symbol)

    case get_pid(symbol) do
      nil ->
        Logger.info("Starting trading of #{symbol}")
        {:ok, _settings} = update_trading_status(symbol, "on")
        {:ok, _pid} = start_symbol_supervisor(symbol)

      pid ->
        Logger.warn("Trading on #{symbol} already started")
        {:ok, _settings} = update_trading_status(symbol, "on")
        {:ok, pid}
    end
  end
  ...
```

together with additional helper functions:

```
# /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
  defp get_pid(symbol) do
    Process.whereis(:"Elixir.Naive.SymbolSupervisor-#{symbol}")
  end

  defp update_trading_status(symbol, status)
       when is_binary(symbol) and is_binary(status) do
    Repo.get_by(Settings, symbol: symbol)
    |> Ecto.Changeset.change(%{status: status})
    |> Repo.update()
  end

  defp start_symbol_supervisor(symbol) do
    DynamicSupervisor.start_child(
      Naive.DynamicSymbolSupervisor,
      {Naive.SymbolSupervisor, symbol}
    )
  end
```

Both implementation and helper functions are almost the same as the ones inside the `naive` application. It could be tempting to abstract some of the logic away but remember that we should treat all applications in our umbrella project as standalone services that should not share any code if possible(we broke that rule for the `TradeEvent` struct from the `streamer` app but we could easily just make a lib with that struct that would be shared between two applications). I would shy away from sharing any business logic between applications in the umbrella project.

There are two additional places where we need to make updates to get our `start_trading/1` to work again:

- we need to update the `children` list inside the `Naive.Application`:

```
# /apps/naive/lib/naive/application.ex
    ...
    children = [
      {Naive.Repo, []},
      {Naive.DynamicSymbolSupervisor, []} # <= replacement of DynamicSupervisor
    ]
```

- we need to replace the `start_trading/1` implementation inside the `Naive` module to `defdelegate` macro(as we don't have any logic to run there):

```
# /apps/naive/lib/naive.ex
...
  alias Naive.DynamicSymbolSupervisor
```

```
  defdelegate start_trading(symbol), to: DynamicSymbolSupervisor
...
```

At this moment we are again able to use the `Naive.start_trading/1` function to start trading on a symbol (behind the scenes it will use logic from the new `Naive.DynamicSymbolSupervisor` module).

12.4 Implement the stop trading functionality

Stop trading will require a change in two places, first inside the `Naive.DynamicSymbolSupervisor` where we will place the termination logic:

```
# /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
  ...
  def stop_trading(symbol) when is_binary(symbol) do
    symbol = String.upcase(symbol)

    case get_pid(symbol) do
      nil ->
        Logger.warn("Trading on #{symbol} already stopped")
        {:ok, _settings} = update_trading_status(symbol, "off")

      pid ->
        Logger.info("Stopping trading of #{symbol}")

        :ok =
          DynamicSupervisor.terminate_child(
            Naive.DynamicSymbolSupervisor,
            pid
          )

        {:ok, _settings} = update_trading_status(symbol, "off")
    end
  end
  ...
```

The second change we need to make is to create a forwarding interface using `defdelegate` inside the `Naive` module:

```
# /apps/naive/lib/naive.ex
  ...
  defdelegate stop_trading(symbol), to: DynamicSymbolSupervisor
  ...
```

That pretty much finishes the `stop_trading/1` functionality. We are now able to start and stop(what was previously not available) trading on a symbol.

12.5 Implement the autostart trading functionality

To implement the autostarting we will need to (in a similar fashion as in the last chapter) add a new supervision level that will be dedicated to supervising the `Naive.DynamicSymbolSupervisor` and the "autostarting" `Task`.

Let's create a new file called `supervisor.ex` inside the `/apps/naive/lib/naive` directory and (as in the last chapter) we will add the `Naive.DynamicSymbolSupervisor` and the `Task` to its children list. We will also update the supervision strategy to `:rest_for_one`:

```
# /apps/naive/lib/naive/supervisor.ex
defmodule Naive.Supervisor do
  use Supervisor

  def start_link(init_arg) do
    Supervisor.start_link(__MODULE__, init_arg, name: __MODULE__)
  end

  def init(_init_arg) do
    children = [
      {Naive.DynamicSymbolSupervisor, []},                # <= added
      {Task,                                              # <= added
       fn ->                                              # <= added
         Naive.DynamicSymbolSupervisor.autostart_trading() # <= added
       end}                                               # <= added
    ]

    Supervisor.init(children, strategy: :rest_for_one) # <= strategy updated
  end
end
```

Now we need to swap the `Naive.DynamicSymbolSupervisor` to `Naive.Supervisor` in the `children` list of the `Naive.Application`, as well as update the registered process' name of the `Naive.Application`:

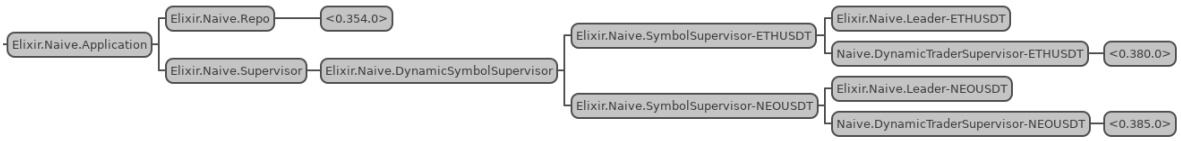
```
# /apps/naive/lib/naive/application.ex
  ...
  def start(_type, _args) do
    children = [
      {Naive.Repo, []},
      {Naive.Supervisor, []} # <= replacement for DynamicSymbolSupervisor
    ]

    opts = [strategy: :one_for_one, name: Naive.Application] # <= name updated
```

Finally, we need to implement `autostart_trading/0` inside the `Naive.DynamicSymbolSupervisor` module as our new `Task` refers to it:

```
# /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
  ...
  # add the below function after the `init/1` function
  def autostart_trading do
    fetch_symbols_to_trade()
    |> Enum.map(&start_trading/1)
  end

  ...

  # and this helper at the end of the module
  defp fetch_symbols_to_trade do
    Repo.all(
      from(s in Settings,
        where: s.status == "on",
        select: s.symbol
      )
    )
  end
  ...
```

Those are the same (excluding updated function names) as inside the `streamer` application. We are fetching enabled symbols and starting new `Naive.SymbolSupervisor` processes for each one.

At this moment we can already see our implementation in action:



At this moment we are able to test the current implementation inside the IEx:

```
$ iex -S mix
...
iex(1)> Naive.start_trading("ethusdt")
21:35:30.207 [info]  Starting trading of ETHUSDT
21:35:30.261 [info]  Starting new supervision tree to trade on ETHUSDT
{:ok, #PID<0.372.0>}
21:35:33.344 [info]  Initializing new trader(1612647333342) for ETHUSDT
iex(3)> Naive.start_trading("neousdt")
```

```
21:35:54.128 [info]  Starting trading of NEOUSDT
21:35:54.169 [info]  Starting new supervision tree to trade on NEOUSDT
{:ok, #PID<0.383.0>}
21:35:56.007 [info]  Initializing new trader(1612647356007) for NEOUSDT
21:38:07.699 [info]  Stopping trading of NEOUSDT
{:ok,
 %Naive.Schema.Settings{
     ...
 }}
```

We can now exit the IEx and start a new one:

```
$ iex -S mix
...
21:39:16.894 [info]  Starting trading of ETHUSDT
21:39:16.938 [info]  Starting new supervision tree to trade on ETHUSDT
21:39:18.786 [info]  Initializing new trader(1612647558784) for ETHUSDT
iex(1)>
```

The above logs confirm that the `naive` application autostarts the previously enabled symbols(using the `start_trading/1` function) as well as `stop_trading/1` updates the status inside the database (so the symbol isn't autostarted at the next initialization).

12.6 Implement the shutdown trading functionality

Last but not least, we will move on to the `shutdown_trading/1` functionality. Let's start with the simplest part which is delegating the function call to the `Naive.DynamicSymbolSupervisor` module from the `Naive` module(interface):

```
  # /apps/naive/lib/naive.ex
  ...
  defdelegate shutdown_trading(symbol), to: DynamicSymbolSupervisor
  ...
```

Next, we will create a `shutdown_trading/1` function inside the `Naive.DynamicSymbolSupervisor` module where we will check is there any trading going on for that symbol(the same as for start/stop), and in case of trading happening we will inform the `Naive.Leader` process handling that symbol that settings have been updated:

```
# /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
  ...
  def shutdown_trading(symbol) when is_binary(symbol) do
    symbol = String.upcase(symbol)

    case get_pid(symbol) do
```

```
      nil ->
        Logger.warn("Trading on #{symbol} already stopped")
        {:ok, _settings} = update_trading_status(symbol, "off")

      _pid ->
        Logger.info("Shutdown of trading on #{symbol} initialized")
        {:ok, settings} = update_trading_status(symbol, "shutdown")
        Naive.Leader.notify(:settings_updated, settings)
        {:ok, settings}
    end
  end
  ...
```

The crucial part of the implementation above is the `notify(:settings_updated, settings)` where we inform the `Naive.Leader` process that it needs to update trading settings.

Currently, the `Naive.Leader` module does *not* support updating the settings after startup - let's add a new interface function together with a callback function that will handle settings updating:

```
# /apps/naive/lib/naive/leader.ex
  ...
  # add the below function as the last clause of the `notify` function
  def notify(:settings_updated, settings) do
    GenServer.call(
      :"#{__MODULE__}-#{settings.symbol}",
      {:update_settings, settings}
    )
  end

  # add the below handler function as the last clause of `handle_call` function
  def handle_call(
        {:update_settings, new_settings},
        _,
        state
      ) do
    {:reply, :ok, %{state | settings: new_settings}}
  end
```

Ok, we have a way to update the settings of the `Naive.Leader` process "on the go" but what effects should the `shutdown` state have on the `Naive.Leader`'s actions?

There are two places that require modification:

- whenever the `Naive.Trader` process will finish the trade cycle, `Naive.Leader` process should *not* start a new one, as well as check, was that the last trader process and if that was the case it needs to call the `Naive.stop_trading/1` function with its symbol to terminate whole supervision tree for that symbol

- whenever the `Naive.Leader` process will receive a `rebuy` notification, it should just ignore it when the symbol is in the `shutdown` state.

Let's look at the updated implementation of the "end of trade" handler:

```
# /apps/naive/lib/naive/leader.ex
  ...
  def handle_info(
        {:DOWN, _ref, :process, trader_pid, :normal},
        %{traders: traders, symbol: symbol, settings: settings} = state
      ) do
    Logger.info("#{symbol} trader finished trade - restarting")

    case Enum.find_index(traders, &(&1.pid == trader_pid)) do
      nil ->
        Logger.warn(
          "Tried to restart finished #{symbol} " <>
            "trader that leader is not aware of"
        )

        if settings.status == "shutdown" and traders == [] do # <= additional check
          Naive.stop_trading(state.symbol)
        end

        {:noreply, state}

      index ->
        new_traders =
          if settings.status == "shutdown" do # <= refactored code
            Logger.warn(
              "The leader won't start a new trader on #{symbol} " <>
                "as symbol is in the 'shutdown' state"
            )

            if length(traders) == 1 do
              Naive.stop_trading(state.symbol)
            end

            List.delete_at(traders, index)
          else
            new_trader_data = start_new_trader(fresh_trader_state(settings))
            List.replace_at(traders, index, new_trader_data)
          end
```

```
        {:noreply, %{state | traders: new_traders}}
    end
  end
  ...
```

As visible in the above code, whenever the `Naive.Trader` process will finish the trade cycle, the `Naive.Leader` process will check can it find a record of that trader in its state (no changes here). We will modify the callback so the leader process will check the `settings.status`. In the `shutdown` status it checks wheater it was the last trader in the `traders` list, to terminate the whole tree at that time(using the `Naive.stop_trading/1` function).

The second callback that we need to modify is the `rebuy` triggered:

```
# /apps/naive/lib/naive/leader.ex
  ...
  def handle_call(
        {:rebuy_triggered, new_trader_state},
        {trader_pid, _},
        %{traders: traders, symbol: symbol, settings: settings} = state
      ) do
    case Enum.find_index(traders, &(&1.pid == trader_pid)) do
      nil ->
        Logger.warn("Rebuy triggered by trader that leader is not aware of")
        {:reply, :ok, state}

      index ->
        old_trader_data = Enum.at(traders, index)
        new_trader_data = %{old_trader_data | :state => new_trader_state}
        updated_traders = List.replace_at(traders, index, new_trader_data)

        updated_traders =
          if settings.chunks == length(traders) do
            Logger.info("All traders already started for #{symbol}")
            updated_traders
          else
            if settings.status == "shutdown" do
              Logger.warn(
                "The leader won't start a new trader on #{symbol} " <>
                  "as symbol is in the 'shutdown' state"
              )

              updated_traders
            else
              Logger.info("Starting new trader for #{symbol}")
              [start_new_trader(fresh_trader_state(settings)) | updated_traders]
```

```
            end
          end

        {:reply, :ok, %{state | :traders => updated_traders}}
    end
  end
  ...
```

In the above `rebuy_triggered` handler function we added branching on the `settings.status` and we simply ignore the rebuy notification when the symbol is in the `shutdown` status.

The final change will be to create a new migration that will update the `TradingStatusEnum` to have a `shutdown` option:

```
$ cd apps/naive
$ mix ecto.gen.migration update_trading_status
* creating priv/repo/migrations/20210205232303_update_trading_status.exs
```

Inside the generated migration file we need to excute a raw SQL command:

```
# /apps/naive/priv/repo/migrations/20210205232303_update_trading_status.exs
defmodule Naive.Repo.Migrations.UpdateTradingStatus do
  use Ecto.Migration

  @disable_ddl_transaction true

  def change do
    Ecto.Migration.execute "ALTER TYPE trading_status ADD VALUE IF NOT EXISTS 'shutdown'"
  end
end
```

We need to apply the same change to the `Naive.Schema.TradingStatusEnum`:

```
# /apps/naive/lib/naive/schema/trading_status_enum.ex
import EctoEnum

defenum(Naive.Schema.TradingStatusEnum, :trading_status, [:on, :off, :shutdown])
```

Don't forget to run `mix ecto.migrate` to run the new migration.

We can now test the `shutdown_trading/1` functionality inside the IEx:

```
$ iex -S mix
...
iex(1)> Streamer.start_streaming("ethusdt")
21:46:26.651 [info]  Starting streaming on ETHUSDT
```

```
{:ok, #PID<0.372.0>}
iex(2)> Naive.start_trading("ethusdt")
21:46:42.830 [info]  Starting trading of ETHUSDT
21:46:42.867 [info]  Starting new supervision tree to trade on ETHUSDT
{:ok, #PID<0.379.0>}
21:46:44.816 [info]  Initializing new trader(1612648004814) for ETHUSDT
...
21:47:52.448 [info]  Rebuy triggered for ETHUSDT by the trader(1612648004814)
...
21:49:58.900 [info]  Rebuy triggered for ETHUSDT by the trader(1612648089409)
...
21:50:58.927 [info]  Rebuy triggered for ETHUSDT by the trader(1612648198900)
...
21:53:27.202 [info]  Rebuy triggered for ETHUSDT by the trader(1612648326215)
21:53:27.250 [info]  Rebuy triggered for ETHUSDT by the trader(1612648325512)
21:53:27.250 [info]  All traders already started for ETHUSDT

# at this moment we have 5 `Naive.Trader` processes trading in parallel

iex(4)> Naive.shutdown_trading("ethusdt")
21:55:01.556 [info]  Shutdown of trading on ETHUSDT initialized
{:ok,
 %Naive.Schema.Settings{
     ...
 }}
...
22:06:58.855 [info]  Trader(1612648407202) finished trade cycle for ETHUSDT
22:06:58.855 [info]  ETHUSDT trader finished trade - restarting
22:06:58.855 [warn]  The leader won't start a new trader on ETHUSDTas symbol is in
shutdown state
22:07:50.768 [info]  Trader(1612648325512) finished trade cycle for ETHUSDT
22:07:50.768 [info]  ETHUSDT trader finished trade - restarting
22:07:50.768 [warn]  The leader won't start a new trader on ETHUSDTas symbol is in
shutdown state
22:07:50.857 [info]  Trader(1612648326215) finished trade cycle for ETHUSDT
22:07:50.857 [info]  ETHUSDT trader finished trade - restarting
22:07:50.857 [warn]  The leader won't start a new trader on ETHUSDTas symbol is in
shutdown state
22:07:51.079 [info]  Trader(1612648089409) finished trade cycle for ETHUSDT
22:07:51.079 [info]  ETHUSDT trader finished trade - restarting
22:07:51.079 [warn]  The leader won't start a new trader on ETHUSDTas symbol is in
shutdown state
22:08:05.401 [info]  Trader(1612648004814) finished trade cycle for ETHUSDT
```

```
22:08:05.401 [info]  ETHUSDT trader finished trade - restarting
22:08:05.401 [warn]  The leader won't start a new trader on ETHUSDTas symbol is in
shutdown state
22:08:05.401 [info]  Stopping trading of ETHUSDT
```

As we can see from the logs above, our `naive` strategy grown from 1 to 5 `Naive.Trader` processes running in parallel, then we called the `shutdown_trading/1` function. In the `shutdown` status, the `Naive.Leader` process ignored `rebuy` notifications and wasn't starting any new `Naive.Trader` processes as the old ones were finishing. At the moment when the last `Naive.Trader` process finished the trade cycle, the `Naive.Leader` called `stop_trading/1` on "it's" symbol, terminating the whole supervision tree for that symbol.

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

Source code for this chapter can be found at Github

Chapter 13

Abstract duplicated supervision code

13.1 Objectives

- overview of requirements
- pseudo generalize `Core.ServiceSupervisor` module
- utilize pseudo generalized code inside the `Naive.DynamicSymbolSupervisor`
- implement a truly generic `Core.ServiceSupervisor`
- use the `Core.ServiceSupervisor` module inside the `streamer` application

13.2 Overview of requirements

In the last few chapters, we went through adding and modifying the dynamic supervision tree around the `naive` and `streamer` applications' workers. Initially, we just copied the implementation from the `streamer` application to the `naive` application (with a few minor tweaks like log messages). That wasn't the most sophisticated solution and we will address this copy-paste pattern in this chapter.

We will write an "extension" of the `DynamicSupervisor` that allows to start, stop and autostart workers. Just to keep things simple we will create a new application inside our umbrella project where we will place the logic. This will save us from creating a new repo for time being.

Our new `Core.ServiceSupervisor` module will hold all the logic responsible for starting, stopping, and autostarting worker processes. To limit the boilerplate inside the implementation modules (like `Naive.DynamicSymbolSupervisor` or `Streamer.DynamicStreamerSupervisor`) we will utilize the `use` macro that will dynamically generate low-level wiring for us.

13.3 Pseudo generalize Core.ServiceSupervisor module

Let's start by creating a new *non-supervised* application called `core` inside our umbrella project. At this moment our "abstraction code" will sit inside it just to keep things simple as otherwise, we would need to create a new repo and jump between codebases which we will avoid for time being:

```
$ cd apps
$ mix new core
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/core.ex
* creating test
* creating test/test_helper.exs
* creating test/core_test.exs
...
```

We can now create a new directory called `core` inside the `apps/core/lib` directory and a new file called `service_supervisor.ex` inside it where we will put all abstracted starting/stopping/autostarting logic.

Let's start with an empty module:

```
# /apps/core/lib/core/service_supervisor.ex
defmodule Core.ServiceSupervisor do

end
```

The first step in our refactoring process will be to move(cut) all of the functions from the `Naive.DynamicSymbolSupervisor` (excluding the `start_link/1`, `init/1` and `shutdown_trading/1`) and put them inside the `Core.ServiceSupervisor` module which should look as follows:

```
# /apps/core/lib/core/service_supervisor.ex
defmodule Core.ServiceSupervisor do

  def autostart_trading do
    ...
  end

  def start_trading(symbol) when is_binary(symbol) do
    ...
  end

  def stop_trading(symbol) when is_binary(symbol) do
```

```
    ...
  end

  defp get_pid(symbol) do
    ...
  end

  defp update_trading_status(symbol, status)
       when is_binary(symbol) and is_binary(status) do
    ...
  end

  defp start_symbol_supervisor(symbol) do
    ...
  end

  defp fetch_symbols_to_trade do
    ...
  end
end
```

All of the above code is `trading` related - we need to rename functions/logs to be more generic.

Starting with `autostart_trading/0` we can rename it to `autostart_workers/0`:

```
  # /apps/core/lib/core/service_supervisor.ex
  ...
  def autostart_workers do       # <= updated function name
    fetch_symbols_to_start()     # <= updated function name
    |> Enum.map(&start_worker/1) # <= updated function name
  end
  ...
```

As we updated two functions inside the `autostart_workers/0` we need to update their implementations.

The `start_trading/1` will become `start_worker/1`, internally we will inline the `start_symbol_supervisor/1` function(move it's contents inside the `start_worker/1` function and remove the `start_symbol_supervisor/1` function) as it's used just once inside this module as well as `update_trading_status/2` need to be renamed to `update_status/2`.

The `fetch_symbols_to_trade/0` will get updated to `fetch_symbols_to_start/0`:

```
  # /apps/core/lib/core/service_supervisor.ex
  def start_worker(symbol) when is_binary(symbol) do # <= updated name
    case get_pid(symbol) do
      nil ->
```

```
      Logger.info("Starting trading of #{symbol}")
      {:ok, _settings} = update_status(symbol, "on") # <= updated name

      {:ok, _pid} =
        DynamicSupervisor.start_child(
          Naive.DynamicSymbolSupervisor,
          {Naive.SymbolSupervisor, symbol}
        )  # ^^^^^^ inlined `start_symbol_supervisor/1`

    pid ->
      Logger.warn("Trading on #{symbol} already started")
      {:ok, _settings} = update_status(symbol, "on") # <= updated name
      {:ok, pid}
  end
end

...

defp fetch_symbols_to_start do # <= updated name
  ...
end
```

Inside the above code we updated the `update_trading_status/2` call to `update_status/2` so we need to update the function header to match:

```
# /apps/core/lib/core/service_supervisor.ex
defp update_status(symbol, status) # <= updated name
    when is_binary(symbol) and is_binary(status) do
   ...
end
```

Last function to rename in this module will be the `stop_trading/1` to `stop_worker/1`, we also need to update calls to `update_trading_status/2` to `update_status/2` as it was renamed:

```
# /apps/core/lib/core/service_supervisor.ex
def stop_worker(symbol) when is_binary(symbol) do # <= updated name
  case get_pid(symbol) do
    nil ->
      Logger.warn("Trading on #{symbol} already stopped")
      {:ok, _settings} = update_status(symbol, "off") # <= updated name

    pid ->
      Logger.info("Stopping trading of #{symbol}")
```

```
        :ok =
          DynamicSupervisor.terminate_child(
            Naive.DynamicSymbolSupervisor,
            pid
          )

        {:ok, _settings} = update_status(symbol, "off") # <= updated name
    end
  end
```

At this moment we have a pseudo-generic implementation of `start_worker/1` and `stop_worker/1` inside the `Core.ServiceSupervisor` module. Function names are generic but they still refer to `Repo`, `Settings`, and other modules specific to the `naive` app's implementation. We are probably in a worse situation than we have been before starting this refactoring ;) but don't fear this was just the first step on the way to abstract away that starting, stopping, and autostarting code.

13.4 Utilize pseudo generalized code inside the Naive DynamicSymbolSupervisor

Before we will jump back to the `naive`'s application modules we need to add the `core` application the dependencies of the `naive` application:

```
# /apps/naive/mix.exs
defp deps do
  [
    {:binance, "~> 1.0"},
    {:binance_mock, in_umbrella: true},
    {:core, in_umbrella: true}, # <= core dep added
    ....
```

Let's get back to the `Naive.DynamicSymbolSupervisor` where we expect functions that we just cut out to exist like `start_trading/1` or `stop_trading/1`.

Let's reimplement more generic versions of those functions as just simple calls to the `Core.ServiceSupervisor` module:

```
# /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
...
def autostart_workers do
  Core.ServiceSupervisor.autostart_workers()
end

def start_worker(symbol) do
```

```
    Core.ServiceSupervisor.start_worker(symbol)
  end

  def stop_worker(symbol) do
    Core.ServiceSupervisor.stop_worker(symbol)
  end
```

We also need to update the `shutdown_trading/1` function as we removed all the private functions that it relies on:

```
  # /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
  def shutdown_worker(symbol) when is_binary(symbol) do # <= updated name
    case Core.ServiceSupervisor.get_pid(symbol) do      # <= module added
      nil ->
        Logger.warn("Trading on #{symbol} already stopped")
        {:ok, _settings} = Core.ServiceSupervisor.update_status(symbol, "off")
                                                # ^^^ updated name + module

      _pid ->
        Logger.info("Shutdown of trading on #{symbol} initialized")
        {:ok, settings} = Core.ServiceSupervisor.update_status(symbol, "shutdown")
                                                # ^^^ updated name + module
        Naive.Leader.notify(:settings_updated, settings)
        {:ok, settings}
    end
  end
```

As we were moving all the private helper functions we didn't make them public so the `Naive.DynamicSymbolSupervisor` module can use them - we will fix that now together with temporary aliases/require/imports at the top of the `Core.ServiceSupervisor`:

```
# /apps/core/lib/core/service_supervisor.ex
defmodule Core.ServiceSupervisor do

  require Logger                    # <= added require

  import Ecto.Query, only: [from: 2] # <= added import

  alias Naive.Repo                  # <= added alias
  alias Naive.Schema.Settings       # <= added alias

  ...

  def get_pid(symbol) do # <= updated from private to public
    ...
```

```
  end

  def update_status(symbol, status) # <= updated from private to public
      when is_binary(symbol) and is_binary(status) do
    ...
  end
```

As `fetch_symbols_to_start/0` is only used internally by the `Core.ServiceSupervisor` module itself, we don't need to make it public.

We can also remove the `aliases` and `import` from the `Naive.DynamicSymbolSupervisor` as it won't need them anymore.

The next step will be to add `ecto` to the deps of the `core` application as it will make db queries now:

```
  # /apps/core/mix.exs
  defp deps do
    [
      {:ecto_sql, "~> 3.0"}
    ]
  end
```

As we modified the interface of the `Naive.DynamicSymbolSupervisor` (for example renamed `start_trading/1` to `start_worker/1` and others) we need to modify the `Naive.Supervisor`'s children list - more specifically the Task process:

```
# /apps/naive/lib/naive/supervisor.ex
      ...
      {Task,
       fn ->
         Naive.DynamicSymbolSupervisor.autostart_workers() # <= func name updated
       end}
       ...
```

The last step will be to update the interface of the `naive` application:

```
# /apps/naive/lib/naive.ex
  alias Naive.DynamicSymbolSupervisor

  def start_trading(symbol) do
    symbol
    |> String.upcase()
    |> DynamicSymbolSupervisor.start_worker()
  end
```

```
def stop_trading(symbol) do
  symbol
  |> String.upcase()
  |> DynamicSymbolSupervisor.stop_worker()
end

def shutdown_trading(symbol) do
  symbol
  |> String.upcase()
  |> DynamicSymbolSupervisor.shutdown_worker()
end
```

Believe it or not, but at this moment(ignoring all of the warnings because we created a circular dependency between the `core` and the `naive` applications - which we will fix in the next steps) our application *runs* just fine! We are able to start and stop trading, autostarting works as well.

13.5 Implement a truly generic `Core.ServiceSupervisor`

Ok. Why did we even do this? What we are aiming for is a separation between the interface of our `Naive.DynamicSymbolSupervisor` module (like `start_worker/1`, `autostart_workers/0` and `stop_worker/1`) and the implementation which is now placed inside the `Core.ServiceSupervisor` module.

That's all nice and to-some-extent understandable but `Core.ServiceSupervisor` module is *not* a generic module. We can't use it inside the `streaming` application to supervise the `Streamer.Binance` processes.

So, what's the point? Well, we can make it even more generic!

13.5.1 First path starting with the `fetch_symbols_to_start/0` function

Moving on to full generalization of the `Core.ServiceSupervisor` module. We will start with the helper functions first as they are the ones doing the work and they need to be truly generalized first:

```
# /apps/core/lib/core/service_supervisor.ex
def fetch_symbols_to_start do
  Repo.all(
    from(s in Settings,
      where: s.status == "on",
      select: s.symbol
    )
  )
end
```

The `fetch_symbols_to_start/0` function uses `Repo` and `Settings` that are aliased at the top of the `Core.ServiceSupervisor` module. This just won't work with any other applications as Streamer will require its own `Repo` and `Settings` modules etc.

To fix that we will *pass* both `repo` and `schema` as arguments to the `fetch_symbols_to_start/0` function which will become `fetch_symbols_to_start/2`:

```
# /apps/core/lib/core/service_supervisor.ex
def fetch_symbols_to_start(repo, schema) do # <= args added
  repo.all( # <= lowercase `repo` is an argument not aliased module
    from(s in schema, # <= settings schema module passed as arg
      where: s.status == "on",
      select: s.symbol
    )
  )
end
```

This will have a knock-on effect on any functions that are using `fetch_symbols_to_start/0` - now they need to use `fetch_symbols_to_start/2` and pass appropriate `Repo` and `Schema` modules.

So, the `fetch_symbols_to_start/0` is referenced by the `autostart_workers/0` - we will need to modify it to pass the repo and schema to the `fetch_symbols_to_start/2` and as it's inside the `Core.ServiceSupervisor` module it needs to get them passed as arguments:

```
# /apps/core/lib/core/service_supervisor.ex
def autostart_workers(repo, schema) do # <= args added
  fetch_symbols_to_start(repo, schema) # <= args passed
  |> Enum.map(&start_worker/1)
end
```

Going even further down the line, `autostart_workers/0` is referenced by the `autostart_workers/0` inside the `Naive.DynamicSymbolSupervisor` module. As this module is (`naive`) application-specific, it is a place where `repo` and `schema` are known from the context - for the `naive` application `repo` is the `Naive.Repo` module and `schema` is the `Naive.Schema.Settings` module:

```
# /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
...
def autostart_workers do
  Core.ServiceSupervisor.autostart_workers(
    Naive.Repo,           # <= new value passed
    Naive.Schema.Settings # <= new value passed
  )
end
```

This finishes the first of multiple paths that we need to follow to fully refactor the `Core.ServiceSupervisor` module.

13.5.2 Second path starting with the `update_status/2`

Let's don't waste time and start from the other helper function inside the `Core.ServiceSupervisor` module. This time we will make the `update_status/2` function fully generic:

```
# /apps/core/lib/core/service_supervisor.ex
def update_status(symbol, status, repo, schema) # <= args added
    when is_binary(symbol) and is_binary(status) do
  repo.get_by(schema, symbol: symbol) # <= using dynamic repo and shcema modules
  |> Ecto.Changeset.change(%{status: status})
  |> repo.update() # <= using dynamic repo module
end
```

As previously we added `repo` and `schema` as arguments and modified the body of the function to utilize them instead of hardcoded modules (aliased at the top of the `Core.ServiceSupervisor` module).

In the same fashion as previously, we need to check "who" is using the `update_status/2` and update those calls to `update_status/4`.

The function is used inside the `start_worker/1` and the `stop_worker/1` inside the `Core.ServiceSupervisor` module so as previously we need to bubble them up(pass via arguments to both `start_worker/1` and `stop_worker/1` functions):

```
# /apps/core/lib/core/service_supervisor.ex
def start_worker(symbol, repo, schema) when is_binary(symbol) do # <= new args
  ...
      {:ok, _settings} = update_status(symbol, "on", repo, schema) # <= args passed
  ...
      {:ok, _settings} = update_status(symbol, "on", repo, schema) # <= args passed
  ...
end

def stop_worker(symbol, repo, schema) when is_binary(symbol) do # <= new args
  ...
      {:ok, _settings} = update_status(symbol, "off", repo, schema) # <= args passed
  ...
      {:ok, _settings} = update_status(symbol, "off", repo, schema) # <= args passed
  ...
end
```

As we modified both `start_worker/1` and `stop_worker/1` by adding two additional arguments we need to update all references to them and here is where things branch out a bit.

We will start with `start_worker/1` function (which is now `start_worker/3`) - it's used by the `autostart_workers/2` inside `Core.ServiceSupervisor` module. The `autostart_workers/2` function already has `repo` and `schema` so we can just pass them to the `start_worker/3`:

```
# /apps/core/lib/core/service_supervisor.ex
def autostart_workers(repo, schema) do
  fetch_symbols_to_start(repo, schema)
  |> Enum.map(&start_worker(&1, repo, schema)) # <= args passed
end
```

Both the `start_worker/3` and the `stop_worker/3` function are used by the functions inside the `Naive.DynamicSymbolSupervisor` module. We need to pass the `Repo` and `Schema` in the same fashion as previously with the `autostart_workers/2` function:

```
# /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
def start_worker(symbol) do
  Core.ServiceSupervisor.start_worker(
    symbol,
    Naive.Repo,            # <= new arg passed
    Naive.Schema.Settings  # <= new arg passed
  )
end

def stop_worker(symbol) do
  Core.ServiceSupervisor.stop_worker(
    symbol,
    Naive.Repo,           # <= new arg passed
    Naive.Schema.Settings # <= new arg passed
  )
end
```

At this moment there's no code inside the `Core.ServiceSupervisor` module referencing the `aliased` `Repo` nor `Schema` modules so we can safely remove both aliases - definitely, we are moving in the right direction!

Btw. Our project still works at this stage, we can start/stop trading and it autostarts trading.

13.5.3 Third path starting with the `get_pid/1` function

Starting again from the most nested helper function - this time the `get_pid/1`:

```
# /apps/core/lib/core/service_supervisor.ex
def get_pid(symbol) do
  Process.whereis(:"Elixir.Naive.SymbolSupervisor-#{symbol}")
end
```

We can see that it has a hardcoded `Naive.SymbolSupervisor` worker module - we need to make this part dynamic by using the `worker_module` argument:

```
# /apps/core/lib/core/service_supervisor.ex
def get_pid(worker_module, symbol) do  # <= arg added
  Process.whereis(:"#{worker_module}-#{symbol}") # <= arg used
end
```

Moving up to functions that are referencing the `get_pid/1` function, those will be the `start_worker/3` and the `stop_worker/3` function.

As those are the two last functions to be updated, we will look into them more closely to finish our refactoring in this 3rd run. At this moment both need to add `worker_module` as both are calling the `get_pid/2` function. Looking at both function we can see two other hardcoded details:

- inside log message there are words "trading" - we can replace them so we will utilize the `worker_module` and `symbol` arguments
- there are two references to the `Naive.DynamicSymbolSupervisor` which we will replace with the `module` argument
- there is one more reference to the `Naive.SymbolSupervisor` module which we will replace with the `worker_module` argument

Let's look at updated functions:

```
# /apps/core/lib/core/service_supervisor.ex
# module and worker_module args added vvvv
def start_worker(symbol, repo, schema, module, worker_module)
    when is_binary(symbol) do
  case get_pid(worker_module, symbol) do # <= worker_module passed
    nil ->
      Logger.info("Starting #{worker_module} worker for #{symbol}")
                  # ^^^ dynamic text
      {:ok, _settings} = update_status(symbol, "on", repo, schema)
      {:ok, _pid} = DynamicSupervisor.start_child(module, {worker_module, symbol})
                                                          # ^^^ args used

    pid ->
      Logger.warn("#{worker_module} worker for #{symbol} already started")
                    # ^^^ dynamic text
      {:ok, _settings} = update_status(symbol, "on", repo, schema)
      {:ok, pid}
  end
end

# module and worker_module added as args vvvv
```

```
  def stop_worker(symbol, repo, schema, module, worker_module)
      when is_binary(symbol) do
    case get_pid(worker_module, symbol) do # <= worker_module passed
      nil ->
        Logger.warn("#{worker_module} worker for #{symbol} already stopped")
                            # ^^^ dynamic text
        {:ok, _settings} = update_status(symbol, "off", repo, schema)

      pid ->
        Logger.info("Stopping #{worker_module} worker for #{symbol}")
                            # ^^^ dynamic text
        :ok = DynamicSupervisor.terminate_child(module, pid) # <= arg used
        {:ok, _settings} = update_status(symbol, "off", repo, schema)
    end
  end
```

Inside both the `start_worker/5` and the `stop_worker/5` functions we modified:

- `get_pid/1` to pass the `worker_module`
- `Logger`'s messages to use the `worker_module` and `symbol`
- `DynamicSupervisor`'s functions to use the `module` and the `worker_module`

Again, as we modified `start_worker/5` we need to make the last change inside the `Core.ServiceSupervisor` module - `autostart_workers/2` uses the `start_worker/5` function:

```
  # /apps/core/lib/core/service_supervisor.ex
  def autostart_workers(repo, schema, module, worker_module) do # <= args added
    fetch_symbols_to_start(repo, schema)
    |> Enum.map(&start_worker(&1, repo, schema, module, worker_module)) # <= args added
  end
```

Just for reference - the final function headers look as following:

```
  # /apps/core/lib/core/service_supervisor.ex
defmodule Core.ServiceSupervisor do
  def autostart_workers(repo, schema, module, worker_module) do
    ...
  end

  def start_worker(symbol, repo, schema, module, worker_module)
      when is_binary(symbol) do
    ...
  end
```

```
  def stop_worker(symbol, repo, schema, module, worker_module)
      when is_binary(symbol) do
    ...
  end

  def get_pid(worker_module, symbol) do
    ...
  end

  def update_status(symbol, status, repo, schema)
      when is_binary(symbol) and is_binary(status) do
    ...
  end

  def fetch_symbols_to_start(repo, schema) do
    ...
  end
end
```

That finishes the 3rd round of updates inside the `Core.ServiceSupervisor` module, now we need to update the `Naive.DynamicSymbolSupervisor` module to use updated functions(and pass required arguments):

```
  # /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
  ...
  def autostart_workers do
    Core.ServiceSupervisor.autostart_workers(
      Naive.Repo,
      Naive.Schema.Settings,
      __MODULE__,             # <= added arg
      Naive.SymbolSupervisor  # <= added arg
    )
  end

  def start_worker(symbol) do
    Core.ServiceSupervisor.start_worker(
      symbol,
      Naive.Repo,
      Naive.Schema.Settings,
      __MODULE__,             # <= added arg
      Naive.SymbolSupervisor  # <= added arg
    )
  end

  def stop_worker(symbol) do
```

```
    Core.ServiceSupervisor.stop_worker(
      symbol,
      Naive.Repo,
      Naive.Schema.Settings,
      __MODULE__,             # <= added arg
      Naive.SymbolSupervisor  # <= added arg
    )
  end

  def shutdown_worker(symbol) when is_binary(symbol) do
    case Core.ServiceSupervisor.get_pid(Naive.SymbolSupervisor, symbol) do # <= arg added
      nil ->
        Logger.warn("#{Naive.SymbolSupervisor} worker for #{symbol} already stopped")
                                                              # ^^^ updated

        {:ok, _settings} =
          Core.ServiceSupervisor.update_status(
            symbol,
            "off",
            Naive.Repo,
            Naive.Schema.Settings
          ) # ^^^ args added

      _pid ->
        Logger.info(
          "Initializing shutdown of #{Naive.SymbolSupervisor} worker for #{symbol}"
        )                                       # ^^^ updated

        {:ok, settings} =
          Core.ServiceSupervisor.update_status(
            symbol,
            "shutdown",
            Naive.Repo,
            Naive.Schema.Settings
          ) # ^^^ additional args passed

        Naive.Leader.notify(:settings_updated, settings)
        {:ok, settings}
    end
  end
```

We needed to update referrences inside the `shutdown_trading/1` function as well, as it calls `get_pid/2` and `update_status/4` functions.

We are now done with refactoring the `Core.ServiceSupervisor` module, it's completely generic and can be used inside both `streamer` and the `naive` applications.

At this moment to use the `Core.ServiceSupervisor` module we need to write interface functions in our supervisors and pass multiple arguments in each one - again, would need to use of copies those functions inside `streamer` and `naive` application. In the next section we will look into how could we leverage Elixir macros to remove that boilerplate.

13.6 Remove boilerplate using `use` macro

Elixir provides a way to `use` other modules. Idea is that inside the `Naive.DynamicSymbolSupervisor` module we are `use`ing the `DynamicSupervisor` module currently but we could use `Core.ServiceSupervisor`:

```
# /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
defmodule Naive.DynamicSymbolSupervisor do
  use Core.ServiceSupervisor
```

To be able to `use` the `Core.ServiceSupervisor` module it needs to provide the `__using__/1` macro. As the simplest content of that macro, we can use here would be just to use the `DynamicSupervisor` inside:

```
  # /apps/core/lib/core/service_supervisor.ex
  defmacro __using__(opts) do
    IO.inspect(opts)
    quote location: :keep do
      use DynamicSupervisor
    end
  end
```

How does this work? As an oversimplification, you can think about it as Elixir will look through the contents of `quote`'s body(everything between `quote ... do` and `end`) in search for the `unquote` function which can inject dynamic content. All of this will become much clearer as we will go through the first example but the important part is that after executing any potential `unquote`s inside the `quote`'s body, Elixir will grab that code as it would be just part of code and place it inside the `Naive.DynamicSymbolSupervisor` module at compile time(we will also see the result of `IO.inspect/1` at compilation).

At this moment(after swapping to `use Core.ServiceSupervisor`) our code still works and it's exactly as we would simply have `use DynamicSupervisor` inside the `Naive.DynamicSymbolSupervisor` module - as at compilation, it will be swapped to it either way(as per contents of the `__using__/1` macro).

As the `autostart_workers/0` function is a part of the boilerplate, we will move it from the `Naive.DynamicSymbolSupervisor` module to the `Core.ServiceSupervisor` module inside the `__using__/1` macro.

Ok, but it has all of those other `naive` application-specific arguments - where will we get those?

That's what that `opts` argument is for inside the `__using__/1` macro. When we call `use Core.ServiceSupervisor` we can pass an additional keyword list which will contain all `naive` application-specific details:

```
# /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
defmodule Naive.DynamicSymbolSupervisor do
  use Core.ServiceSupervisor,
    repo: Naive.Repo,
    schema: Naive.Schema.Settings,
    module: __MODULE__,
    worker_module: Naive.SymbolSupervisor
```

We can now update the `__using__/1` macro to assign all of those details to variables(instead of using `IO.inspect/1`):

```
  # /apps/core/lib/core/service_supervisor.ex
  defmacro __using__(opts) do
    {:ok, repo} = Keyword.fetch(opts, :repo)
    {:ok, schema} = Keyword.fetch(opts, :schema)
    {:ok, module} = Keyword.fetch(opts, :module)
    {:ok, worker_module} = Keyword.fetch(opts, :worker_module)
    ...
```

At this moment we can use those *dynamic* values to generate code that will be specific to the implementation module for example `autostart_workers/0` that we moved from the `Naive.DynamicSymbolSupervisor` module and will need to have different values passed to it(like `Streamer.Binance` as `worker_module`) for the `streamer` application. We can see that it requires inserting those dynamic values inside the `autstart_workers/0` but how to dynamically inject arguments - `unquote` to the rescue. When we will update the `autostart_workers/0` function from:

```
      # sample moved code from the `Naive.DynamicSymbolSupervisor` module
      def autostart_workers do
          Core.ServiceSupervisor.autostart_workers(
          Naive.Repo,
          Naive.Schema.Settings,
          __MODULE__,
          Naive.SymbolSupervisor
        )
      end
```

to:

```
      # updated code that will become part of the `__using__/1` macro
      def autostart_workers do
        Core.ServiceSupervisor.autostart_workers(
          unquote(repo),
```

```
        unquote(schema),
        unquote(module),
        unquote(worker_module)
      )
    end
```

At the end, generated code that will be "pasted" to the `Naive.DynamicSymbolSupervisor` module at compile time will be:

```
  # compiled code attached to the `Naive.DynamicSymbolSupervisor` module
  def autostart_workers do
    Core.ServiceSupervisor.autostart_workers(
      Naive.Repo,
      Naive.Schema.Settings,
      Naive.DynamicSymbolSupervisor,
      Naive.SymbolSupervisor
    )
  end
```

This way we can dynamically create functions for any application(for the `streamer` application, it will generate function call with the `Streamer.Repo`, `Streamer.Schema.Settings` args, etc).

We can apply that to all of the passed variables inside the `autostart_workers/0` function - just for reference full macro will look as follows:

```
# /apps/core/lib/core/service_supervisor.ex
defmacro __using__(opts) do
  {:ok, repo} = Keyword.fetch(opts, :repo)
  {:ok, schema} = Keyword.fetch(opts, :schema)
  {:ok, module} = Keyword.fetch(opts, :module)
  {:ok, worker_module} = Keyword.fetch(opts, :worker_module)

  quote location: :keep do
    use DynamicSupervisor

    def autostart_workers do
      Core.ServiceSupervisor.autostart_workers(
        unquote(repo),
        unquote(schema),
        unquote(module),
        unquote(worker_module)
      )
    end
  end
end
```

You can think about the above macro that it will substitute the `unquote(..)` parts with passed values and then it will grab the whole contents between `quote ... do` and `end` and it will paste it to the `Naive.DynamicSymbolSupervisor` module at compile-time - we can visualize generated/"pasted" code as:

```
# generated by the `__using__/1` macro injected into the `Naive.DynamicSymbolSupervisor`
use DynamicSupervisor

def autostart_workers do
  Core.ServiceSupervisor.autostart_workers(
    Naive.Repo,
    Naive.Schema.Settings,
    Naive.DynamicSymbolSupervisor,
    Naive.SymbolSupervisor
  )
end
```

This is exactly the code that we had before inside the `Naive.DynamicSymbolSupervisor` module but now it's stored away inside the `Core.ServiceSupervisor`'s `__using__/1` macro and it doesn't need to be implemented/copied across twice into two apps anymore.

We can now follow the same principle and move `start_worker/1` and `stop_worker/1` from the `Naive.DynamicSymbolSupervisor` module into `__using__/1` macro inside the `Core.ServiceSupervisor` module:

```
    # /apps/core/lib/core/service_supervisor.ex
    # append the below before the end of the __using__/1 macro
    def start_worker(symbol) do
      Core.ServiceSupervisor.start_worker(
        symbol, # <= this needs to stay as variable
        unquote(repo),
        unquote(schema),
        unquote(module),
        unquote(worker_module)
      )
    end

    def stop_worker(symbol) do
      Core.ServiceSupervisor.stop_worker(
        symbol, # <= this needs to stay as variable
        unquote(repo),
        unquote(schema),
        unquote(module),
        unquote(worker_module)
      )
    end
```

Here we have an example of an execution time variable called `symbol` that we should *not* `unquote` as it will be different per function call (source code should have `symbol` variable there not for example `"NEOUSDT"`).

At this moment the `Naive.DynamicSymbolSupervisor` consists of only `start_link/1`, `init/1` and `shutdown_worker/1`, it's under 50 lines of code and works exactly as before refactoring. All of the boilerplate was moved to the `Core.ServiceSupervisor` module.

We left the `shutdown_worker/1` function as it's specific to the `naive` application, but inside it, we utilize both the `get_pid/2` and the `update_status/4` functions where we are passing the `naive` application-specific variables(like `Naive.Repo`).

To make things even nicer we can create convenience wrappers for those two functions inside the `__using__/1` macro:

```
    # /apps/core/lib/core/service_supervisor.ex
    # add below at the end of `quote` block inside `__using__/1`
    defp get_pid(symbol) do
      Core.ServiceSupervisor.get_pid(
        unquote(worker_module),
        symbol
      )
    end

    defp update_status(symbol, status) do
      Core.ServiceSupervisor.update_status(
        symbol,
        status,
        unquote(repo),
        unquote(schema)
      )
    end
```

As those will get compiled and "pasted" into the `Naive.DynamicSymbolSupervisor` module we can utilize them inside the `shutdown_worker/1` function as they would be much simpler naive application-specific local functions:

```
# /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
def shutdown_worker(symbol) when is_binary(symbol) do
  case get_pid(symbol) do # <= macro provided function
    nil ->
      Logger.warn("#{Naive.SymbolSupervisor} worker for #{symbol} already stopped")
      {:ok, _settings} = update_status(symbol, "off") # <= macro provided function

    _pid ->
      Logger.info(
        "Initializing shutdown of #{Naive.SymbolSupervisor} worker for #{symbol}"
      )
      {:ok, settings} = update_status(symbol, "shutdown") # <= macro provided function
```

```
        Naive.Leader.notify(:settings_updated, settings)
        {:ok, settings}
    end
  end
```

And now, a very last change - I promise ;) Both the `start_link/1` and the `init/1` functions are still referencing the `DynamicSupervisor` module which could be a little bit confusing - let's swap those calls to use the `Core.ServiceSupervisor` module (both to not confuse people and be consistent with the `use` macro):

```
  # /apps/naive/lib/naive/dynamic_symbol_supervisor.ex
  def start_link(init_arg) do
    Core.ServiceSupervisor.start_link(__MODULE__, init_arg, name: __MODULE__)
  end

  def init(_init_arg) do
    Core.ServiceSupervisor.init(strategy: :one_for_one)
  end
```

As we don't want/need to do anything different inside the `Core.ServiceSupervisor` module than the `DynamicSupervisor` is doing we can just delegate both of those inside the `Core.ServiceSupervisor` module:

```
  # /apps/core/lib/core/service_supervisor.ex
  defdelegate start_link(module, args, opts), to: DynamicSupervisor
  defdelegate init(opts), to: DynamicSupervisor
```

That finishes our refactoring of both the `Naive.DynamicSymbolSupervisor` and the `Core.ServiceSupervisor` modules.

We can test to confirm that everything works as expected:

```
$ iex -S mix
iex(1)> Naive.start_trading("NEOUSDT")
21:42:37.741 [info]  Starting Elixir.Naive.SymbolSupervisor worker for NEOUSDT
21:42:37.768 [info]  Starting new supervision tree to trade on NEOUSDT
{:ok, #PID<0.464.0>}
21:42:39.455 [info]  Initializing new trader(1614462159452) for NEOUSDT
iex(2)> Naive.stop_trading("NEOUSDT")
21:43:08.362 [info]  Stopping Elixir.Naive.SymbolSupervisor worker for NEOUSDT
{:ok,
 %Naive.Schema.Settings{
   ...
 }}
iex(3)> Naive.start_trading("HNTUSDT")
```

```
21:44:08.689 [info]  Starting Elixir.Naive.SymbolSupervisor worker for HNTUSDT
21:44:08.723 [info]  Starting new supervision tree to trade on HNTUSDT
{:ok, #PID<0.475.0>}
21:44:11.182 [info]  Initializing new trader(1614462251182) for HNTUSDT
BREAK: (a)bort (A)bort with dump (c)ontinue (p)roc info (i)nfo
       (l)oaded (v)ersion (k)ill (D)b-tables (d)istribution
$ iex -S mix
21:47:22.119 [info]  Starting Elixir.Naive.SymbolSupervisor worker for HNTUSDT
21:47:22.161 [info]  Starting new supervision tree to trade on HNTUSDT
21:47:24.213 [info]  Initializing new trader(1614462444212) for HNTUSDT
iex(1)> Naive.shutdown_trading("HNTUSDT")
21:48:42.003 [info]  Initializing shutdown of Elixir.Naive.SymbolSupervisor worker for
HNTUSDT
{:ok,
 %Naive.Schema.Settings{
   ...
 }}
```

The above test confirms that we can start, stop, and shut down trading on a symbol as well as autostarting of trading works.

13.7 Use the `Core.ServiceSupervisor` module inside the `streamer` application

As we are happy with the implementation of the `Core.ServiceSupervisor` module we can upgrade the `streamer` application to use it.

We need to start with adding the `core` application to the list of dependencies of the `streamer` application:

```
# /apps/streamer/mix.exs
  defp deps do
    [
      {:binance, "~> 1.0"},
      {:core, in_umbrella: true}, # <= core added to deps
      ...
```

We can now move on to the `Streamer.DynamicStreamerSupervisor` where we will remove everything (really everything including `imports`, `aliases` and even `require`) beside the `start_link/1` and the `init/1`. As with the `Naive.DynamicSymbolSupervisor` we will `use` the `Core.ServiceSupervisor` and pass all required options - *full* implementation of the `Streamer.DynamicStreamerSupervisor` module should look as follows:

```
# /apps/streamer/lib/streamer/dynamic_streamer_supervisor.ex
defmodule Streamer.DynamicStreamerSupervisor do
  use Core.ServiceSupervisor,
    repo: Streamer.Repo,
    schema: Streamer.Schema.Settings,
    module: __MODULE__,
    worker_module: Streamer.Binance

  def start_link(init_arg) do
    Core.ServiceSupervisor.start_link(__MODULE__, init_arg, name: __MODULE__)
  end

  def init(_init_arg) do
    Core.ServiceSupervisor.init(strategy: :one_for_one)
  end
end
```

Not much to add here - we are `use`ing the `Core.ServiceSupervisor` module and passing options to it so it can macro generates `streamer` application-specific wrappers(like `start_worker/1` or `stop_worker/1` with required repo, schema, etc) around generic logic from the `Core.ServiceSupervisor` module.

Using the `Core.Servicesupervisor` module will have an impact on the interface of the `Streamer.DynamicStreamerSupervisor` as it will now provide functions like `start_worker/1` instead of `start_streaming/1` etc.

As with the `naive` application, we need to update the Task function inside the `Streamer.Supervisor` module:

```
# /apps/streamer/lib/streamer/supervisor.ex
      ...
      {Task,
       fn ->
         Streamer.DynamicStreamerSupervisor.autostart_workers()
       end}
      ...
```

As well as main `Streamer` module needs to forward calls instead of delegating:

```
  # /apps/streamer/lib/streamer.ex
  alias Streamer.DynamicStreamerSupervisor

  def start_streaming(symbol) do
    symbol
    |> String.upcase()
    |> DynamicStreamerSupervisor.start_worker()
  end

  def stop_streaming(symbol) do
    symbol
    |> String.upcase()
    |> DynamicStreamerSupervisor.stop_worker()
  end
```

We can run a quick test to confirm that indeed everything works as expected:

```
$ iex -S mix
iex(1)> Streamer.start_streaming("NEOUSDT")
22:10:38.813 [info]  Starting Elixir.Streamer.Binance worker for NEOUSDT
{:ok, #PID<0.465.0>}
iex(2)> Streamer.stop_streaming("NEOUSDT")
22:10:48.212 [info]  Stopping Elixir.Streamer.Binance worker for NEOUSDT
{:ok,
 %Streamer.Schema.Settings{
   __meta__: #Ecto.Schema.Metadata<:loaded, "settings">,
   id: "db8c9429-2356-4243-a08f-0d0e89b74986",
   inserted_at: ~N[2021-02-25 22:15:16],
   status: "off",
   symbol: "NEOUSDT",
   updated_at: ~N[2021-02-27 22:10:48]
 }}
iex(3)> Streamer.start_streaming("LTCUSDT")
22:26:03.361 [info]  Starting Elixir.Streamer.Binance worker for LTCUSDT
{:ok, #PID<0.490.0>}
BREAK: (a)bort (A)bort with dump (c)ontinue (p)roc info (i)nfo
       (l)oaded (v)ersion (k)ill (D)b-tables (d)istribution
^C
$ iex -S mix
...
22:26:30.775 [info]  Starting Elixir.Streamer.Binance worker for LTCUSDT
```

This finishes the implementation for both the `streamer` and the `naive` application. We are generating dynamic functions(metaprogramming) using Elixir macros which is a cool exercise to go through and feels like superpowers ;)

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

Source code for this chapter can be found at Github

Chapter 14

Store trade events and orders inside the database

14.1 Objectives

- overview of requirements
- create a new `data_warehouse` application in the umbrella
- connect to the database using Ecto
- store trade events' data
- store orders' data
- implement supervision

14.2 Overview of requirements

In the next chapter, we will move on to testing our strategy against historical data(aka backtesting - I will explain that process in the next chapter). What we need to have in place before we will be able to do that is both trade events and orders stored in the database.

Starting with the trade events. The `streamer` application could store trade events from Binance inside its database but how would that work if we would like to introduce another source of non-streamed trade events(ie. flat files, HTTP polling). It would be better if the `Streamer.Binance` process would keep on streaming those trade events as it is and we would create a new application that would subscribe to the existing `TRADE_EVENTS:#{symbol}` topic and store them in the database.

A similar idea applies to the orders' data. At this moment the `naive` application uses the Binance module to place orders. We could store them inside the `naive` application's database but how would that work if we would like to introduce another trading strategy. Holding data in separate databases for each strategy would cause further complications in future reporting, auditing, etc.

To store trade events' and orders' data we will create a new application called `data_warehouse` inside our umbrella

project. It will subscribe to a `TRADE_EVENTS:#{symbol}` stream as well as `ORDERS:#{symbol}` stream, convert broadcasted data to its own representations(structs) and store it inside the database.

Trade events are already broadcasted to the PubSub topic, orders on the other hand aren't. We will need to modify the `Naive.Trader` module to broadcast the new and updated orders to the `ORDERS:#{symbol}` topic.

After implementing the basic worker that will store the incoming data(trade events and orders) inside the database, we will look into adding a supervision tree utilizing Elixir Registry. It will allow us to skip registering every worker with a unique atom and will offer an easy lookup to fetch PIDs instead.

14.3 Create a new `data_warehouse` application in the umbrella

Let's start by creating a new application called `data_warehouse` inside our umbrella:

```
$ cd apps
$ mix new data_warehouse --sup
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/data_warehouse.ex
* creating lib/data_warehouse/application.ex
* creating test
* creating test/test_helper.exs
* creating test/data_warehouse_test.exs
...
```

14.4 Connect to the database using Ecto

We can now follow similar steps as previously and add required dependencies (like the `ecto`) to its `deps` by modifying its `mix.exs` file:

```
  # /apps/data_warehouse/mix.exs
  defp deps do
    [
      {:ecto_sql, "~> 3.0"},
      {:ecto_enum, "~> 1.4"},
      {:phoenix_pubsub, "~> 2.0"},
      {:postgrex, ">= 0.0.0"},
      {:streamer, in_umbrella: true}
    ]
  end
```

Additionally, we added the `phoenix_pubsub` module to be able to subscribe to the PubSub topic and the `streamer` application to be able to use its `Streamer.Binance.TradeEvent` struct.

We can now jump back to the terminal to install added dependencies and generate a new `Ecto.Repo` module:

```
$ mix deps.get
  ...
$ cd apps/data_warehouse
$ mix ecto.gen.repo -r DataWarehouse.Repo
* creating lib/data_warehouse
* creating lib/data_warehouse/repo.ex
* updating ../../config/config.exs
```

Before we will be able to create migrations that will create our tables we need to update the generated configuration inside the `config/config.exs` file:

```
# /config/config.exs
...
config :data_warehouse,            # <= added line
  ecto_repos: [DataWarehouse.Repo] # <= added line

config :data_warehouse, DataWarehouse.Repo,
  database: "data_warehouse",         # <= updated line
  username: "postgres",               # <= updated line
  password: "hedgehogSecretPassword", # <= updated line
  hostname: "localhost"
...
```

and add the `DataWarehouse.Repo` module to the children list of the `DataWarehouse.Application`'s process:

```
    # /apps/data_warehouse/lib/data_warehouse/application.ex
    ...
    children = [
      {DataWarehouse.Repo, []}
    ]
    ...
```

The last step will be to create a database by running `mix ecto.create -r DataWarehouse.Repo` command.

This ends up the setup of the `Ecto` - we can now move on to the implementation of storing the orders and the trade events.

14.5 Store trade events' data

The first step to store trade events inside the database will be to create a table that will hold our data. We will start by creating the migration:

```
$ cd apps/data_warehouse
$ mix ecto.gen.migration create_trade_events
* creating priv/repo/migrations
* creating priv/repo/migrations/20210222224514_create_trade_events.exs
```

The `Streamer.Binance.TradeEvent` struct will serve as a list of columns for our new `trade_events` table. Here's the full implementation of our migration:

```
# /apps/data_warehouse/priv/repo/migrations/20210222224514_create_trade_events.exs
defmodule DataWarehouse.Repo.Migrations.CreateTradeEvents do
  use Ecto.Migration

  def change do
    create table(:trade_events, primary_key: false) do
      add(:id, :uuid, primary_key: true)
      add(:event_type, :text)
      add(:event_time, :bigint)
      add(:symbol, :text)
      add(:trade_id, :integer)
      add(:price, :text)
      add(:quantity, :text)
      add(:buyer_order_id, :bigint)
      add(:seller_order_id, :bigint)
      add(:trade_time, :bigint)
      add(:buyer_market_maker, :bool)

      timestamps()
    end
  end
end
```

We added the additional `id` field to easily identify each trade event and our timestamps for monitoring.

Let's run the migration so it will create a new `trade_events` table for us:

```
$ mix ecto.migrate
```

The next step will be to create a new directory called `schema` inside the `apps/data_warehouse/lib/data_warehouse` directory. Inside it, we need to create a new schema file called `trade_event.ex`. We can copy across the same columns from the migration straight to schema:

```
# /apps/data_warehouse/lib/data_warehouse/schema/trade_event.ex
defmodule DataWarehouse.Schema.TradeEvent do
  use Ecto.Schema

  @primary_key {:id, :binary_id, autogenerate: true}

  schema "trade_events" do
    field(:event_type, :string)
    field(:event_time, :integer)
    field(:symbol, :string)
    field(:trade_id, :integer)
    field(:price, :string)
    field(:quantity, :string)
    field(:buyer_order_id, :integer)
    field(:seller_order_id, :integer)
    field(:trade_time, :integer)
    field(:buyer_market_maker, :boolean)

    timestamps()
  end
end
```

At this moment we should be able to execute crud(create, read[select], update, delete) operations over the table using the above struct.

Currently, we can already store the trade events' data inside the database so we can move on to collecting it. Trade events are getting broadcasted by the `Streamer.Binance` process here:

```
    # /apps/streamer/lib/streamer/binance.ex
    ...
    Phoenix.PubSub.broadcast(
      Streamer.PubSub,
      "TRADE_EVENTS:#{trade_event.symbol}",
      trade_event
    )
    ...
```

We will implement a `subscriber` process that will be given a PubSub topic and will store incoming data inside the database.

Let's start by creating a new folder called `subscriber` inside the `apps/data_warehouse/lib/data_warehouse` directory together with a new file called `worker.ex` inside it:

```
# /apps/data_warehouse/lib/data_warehouse/subscriber/worker.ex
defmodule DataWarehouse.Subscriber.Worker do
  use GenServer

  require Logger

  defmodule State do
    @enforce_keys [:topic]
    defstruct [:topic]
  end

  def start_link(topic) do
    GenServer.start_link(
      __MODULE__,
      topic,
      name: :"#{__MODULE__}-#{topic}"
    )
  end

  def init(topic) do
    {:ok,
     %State{
       topic: topic
     }}
  end
end
```

At this moment it's just a box standard implementation of the `GenServer` with a state struct containing a single key(`:topic`). We need to update the `init/1` function to subscribe to the PubSub topic:

```
# /apps/data_warehouse/lib/data_warehouse/subscriber/worker.ex
  def init(topic) do
    Logger.info("DataWarehouse worker is subscribing to #{topic}")

    Phoenix.PubSub.subscribe(
      Streamer.PubSub,
      topic
    )
    ...
```

Next, we need to add a handler for received messages:

```
# /apps/data_warehouse/lib/data_warehouse/subscriber/worker.ex
  def handle_info(%Streamer.Binance.TradeEvent{} = trade_event, state) do
    opts =
      trade_event
      |> Map.from_struct()

    struct!(DataWarehouse.Schema.TradeEvent, opts)
    |> DataWarehouse.Repo.insert()

    {:noreply, state}
  end
```

As we did in the case of the `Naive.Trader`, all incoming messages trigger a `handle_info/2` callback with the contents of the message and the current state of the subscriber worker. We just convert that incoming trade event to a map and then that map to the `TradeEvent` struct that gets inserted into the database.

This finishes storing of trade events implementation which we can test by in the interactive shell by running:

```
$ iex -S mix
...
iex(1)> Streamer.start_streaming("XRPUSDT")
00:48:30.147 [info]  Starting Elixir.Streamer.Binance worker for XRPUSDT
{:ok, #PID<0.395.0>}
iex(2)> DataWarehouse.Subscriber.Worker.start_link("TRADE_EVENTS:XRPUSDT")
00:49:48.204 [info]  DataWarehouse worker is subscribing to TRADE_EVENTS:XRPUSDT
{:ok, #PID<0.405.0>}
```

After a couple of minutes we can check the database using `psql`:

```
$ psql -Upostgres -h127.0.0.1
Password for user postgres:
...
postgres=# \c data_warehouse;
You are now connected to database "data_warehouse" as user "postgres".
data_warehouse=# \x
Expanded display is on.
data_warehouse=# SELECT * FROM trade_events;
-[ RECORD 1 ]------+------------------------------------
id                 | f6eae686-946a-4e34-9c33-c7034c2cad5d
event_type         | trade
event_time         | 1614041388236
symbol             | XRPUSDT
trade_id           | 152765072
```

```
price              | 0.56554000
quantity           | 1199.10000000
buyer_order_id     | 1762454848
seller_order_id    | 1762454775
trade_time         | 1614041388235
buyer_market_maker | f
inserted_at        | 2021-02-23 00:49:48
...
```

As we can see in the above output, trade events are now getting stored inside the database.

14.6 Store orders' data

In the same fashion as with trade events' data above, to store orders data we will create an `orders` table inside a new migration:

```
$ cd apps/data_warehouse
$ mix ecto.gen.migration create_orders
* creating priv/repo/migrations/20210222224522_create_orders.exs
```

The list of columns for this table will be a copy of `Binance.Order` struct returned from the Binance exchange:

```
# /apps/data_warehouse/priv/repo/migrations/20210222224522_create_orders.exs
defmodule DataWarehouse.Repo.Migrations.CreateOrders do
  use Ecto.Migration

  def change do
    create table(:orders, primary_key: false) do
      add(:order_id, :bigint, primary_key: true)
      add(:client_order_id, :text)
      add(:symbol, :text)
      add(:price, :text)
      add(:original_quantity, :text)
      add(:executed_quantity, :text)
      add(:cummulative_quote_quantity, :text)
      add(:status, :text)
      add(:time_in_force, :text)
      add(:type, :text)
      add(:side, :text)
      add(:stop_price, :text)
      add(:iceberg_quantity, :text)
      add(:time, :bigint)
      add(:update_time, :bigint)
```

```
      timestamps()
    end
  end
end
```

We updated all of the shortened names like `orig_qty` to full names like `original_quantity`.

Let's run the migration so it will create a new `orders` table for us:

```
$ mix ecto.migrate
```

We can copy the above fields list to create a schema module. First, let's create a new file called `order.ex` inside the `apps/data_warehouse/lib/data_warehouse/schema` directory:

```
# /apps/data_warehouse/lib/data_warehouse/schema/order.ex
defmodule DataWarehouse.Schema.Order do
  use Ecto.Schema

  @primary_key {:order_id, :integer, autogenerate: false}

  schema "orders" do
    field(:client_order_id, :string)
    field(:symbol, :string)
    field(:price, :string)
    field(:original_quantity, :string)
    field(:executed_quantity, :string)
    field(:cummulative_quote_quantity, :string)
    field(:status, :string)
    field(:time_in_force, :string)
    field(:type, :string)
    field(:side, :string)
    field(:stop_price, :string)
    field(:iceberg_quantity, :string)
    field(:time, :integer)
    field(:update_time, :integer)

    timestamps()
  end
end
```

We can now add a handler to our `DataWarehouse.Subscriber.Worker` that will convert the `Binance.Order` struct to `DataWarehouse.Schema.Order` and store data inside the database:

```
# /apps/data_warehouse/lib/data_warehouse/subscriber/worker.ex
  def handle_info(%Binance.Order{} = order, state) do
    data =
      order
      |> Map.from_struct()

    struct(DataWarehouse.Schema.Order, data)
    |> Map.merge(%{
      original_quantity: order.orig_qty,
      executed_quantity: order.executed_qty,
      cummulative_quote_quantity: order.cummulative_quote_qty,
      iceberg_quantity: order.iceberg_qty
    })
    |> DataWarehouse.Repo.insert(
      on_conflict: :replace_all,
      conflict_target: :order_id
    )

    {:noreply, state}
  end
  ...
```

In the above code, we are copying the matching fields using the `struct/2` function but all other fields that aren't 1 to 1 between two structs won't be copied, so we need to merge them in the second step(using the `Map.merge/2` function). We are also using the `on_conflict: :replace_all` option to make the `insert/2` function act as it would be `upsert/2`(to avoid writing separate logic for inserting and updating the orders).

Having all of this in place we will now be able to store broadcasted orders' data in the database but there's nothing actually broadcasting them.

We need to modify the `Naive.Trader` module to broadcast the `Binance.Order` whenever it places buy/sell orders or fetches them again:

```
    # /apps/naive/lib/naive/trader.ex
    ...
    # inside placing initial buy order callback
    {:ok, %Binance.OrderResponse{} = order} =
      @binance_client.order_limit_buy(symbol, quantity, price, "GTC")

    :ok = broadcast_order(order)
    ...
```

```
  # inside buy order (partially) filled callback
  {:ok, %Binance.Order{} = current_buy_order} =
    @binance_client.get_order(
      symbol,
      timestamp,
      order_id
    )

  :ok = broadcast_order(current_buy_order)
  ...

      # inside the same callback in case of buy order filled
      {:ok, %Binance.OrderResponse{} = order} =
        @binance_client.order_limit_sell(symbol, quantity, sell_price, "GTC")

      :ok = broadcast_order(order)
  ...

  # inside sell order (partially) filled callback
  {:ok, %Binance.Order{} = current_sell_order} =
    @binance_client.get_order(
      symbol,
      timestamp,
      order_id
    )

  :ok = broadcast_order(current_sell_order)
  ...
```

Above 4 places send both the `Binance.OrderResponse` and the `Binance.Order` structs - our `broadcast_order/1` function needs to be able to handle them both. Add the following at the bottom of the `Naive.Trader` module:

```
# /apps/naive/lib/naive/trader.ex
defp broadcast_order(%Binance.OrderResponse{} = response) do
  response
  |> convert_to_order()
  |> broadcast_order()
end

defp broadcast_order(%Binance.Order{} = order) do
  Phoenix.PubSub.broadcast(
    Streamer.PubSub,
    "ORDERS:#{order.symbol}",
    order
```

```
    )
  end

  defp convert_to_order(%Binance.OrderResponse{} = response) do
    data =
      response
      |> Map.from_struct()

    struct(Binance.Order, data)
    |> Map.merge(%{
      cummulative_quote_qty: "0.00000000",
      stop_price: "0.00000000",
      iceberg_qty: "0.00000000",
      is_working: true
    })
  end
```

As `DataWarehouse.Subscriber.Worker` process expects only the `Binance.Order` structs to be broadcasted, we first check is it the `Binance.OrderResponse` struct and convert the passed value to the `Binance.Order` struct (if that's the case) and only then broadcast it to the PubSub topic.

The converting logic as previously uses the `struct/2` function but it also merges in default values that are missing from the much smaller `Binance.OrderResponse` struct(with comparison to the `Binance.Order`).

At this moment we will be able to store orders inside the database and we can check that by running:

```
$ iex -S mix
...
iex(1)> DataWarehouse.Subscriber.Worker.start_link("ORDERS:NEOUSDT")
22:37:43.043 [info]  DataWarehouse worker is subscribing to ORDERS:XRPUSDT
{:ok, #PID<0.400.0>}
iex(2)> Naive.start_trading("NEOUSDT")
22:38:39.741 [info]  Starting Elixir.Naive.SymbolSupervisor worker for NEOUSDT
22:38:39.832 [info]  Starting new supervision tree to trade on NEOUSDT
{:ok, #PID<0.402.0>}
22:38:41.654 [info]  Initializing new trader(1614119921653) for NEOUSDT
iex(3)> Streamer.start_streaming("NEOUSDT")
22:39:23.786 [info]  Starting Elixir.Streamer.Binance worker for NEOUSDT
{:ok, #PID<0.412.0>}
22:39:27.187 [info]  The trader(1614119921653) is placing a BUY order for NEOUSDT @ 37.549,
quantity: 5.326
22:39:27.449 [info]  The trader(1614119921653) is placing a SELL order for NEOUSDT @ 37.578,
quantity: 5.326.
```

At this moment inside the DataWarehouse's database we should see orders:

```
$ psql -Upostgres -h127.0.0.1
Password for user postgres:
...
postgres=# \c data_warehouse;
You are now connected to database "data_warehouse" as user "postgres".
data_warehouse=# \x
Expanded display is on.
data_warehouse=# SELECT * FROM orders;
-[ RECORD 1 ]--------------+---------------------------------
order_id                   | 1
client_order_id            | C81E728D9D4C2F636F067F89CC14862C
symbol                     | NEOUSDT
price                      | 38.16
original_quantity          | 5.241
executed_quantity          | 0.00000000
cummulative_quote_quantity | 0.00000000
status                     | FILLED
time_in_force              | GTC
type                       | LIMIT
side                       | BUY
stop_price                 | 0.00000000
iceberg_quantity           | 0.00000000
time                       | 1614120906320
update_time                | 1614120906320
inserted_at                | 2021-02-23 22:55:10
updated_at                 | 2021-02-23 22:55:10
-[ RECORD 2 ]--------------+---------------------------------
order_id                   | 2
client_order_id            | ECCBC87E4B5CE2FE28308FD9F2A7BAF3
symbol                     | NEOUSDT
price                      | 38.19
original_quantity          | 5.241
executed_quantity          | 0.00000000
cummulative_quote_quantity | 0.00000000
status                     | NEW
time_in_force              | GTC
type                       | LIMIT
side                       | SELL
stop_price                 | 0.00000000
iceberg_quantity           | 0.00000000
time                       |
update_time                |
```

```
inserted_at                | 2021-02-23 22:55:10
updated_at                 | 2021-02-23 22:55:10
```

The first record above got inserted and updated as its state is "FILLED", the second one wasn't updated yet as it's still in "NEW" state - that confirms that the upsert trick works.

That finishes the implementation of storing orders inside the database.

14.7 Implement supervision

Currently, we have a `DataWarehouse.Subscriber.Worker` process that will take care of storing data into the database, but sadly if anything will go wrong inside our worker and it will crash there's no supervision in place to restart it.

The supervision tree for the `data_warehouse` application will be similar to ones from the `naive` and `streamer` apps but different enough to *not* use the `Core.ServiceSupervisor` abstraction.

For example, it doesn't use the `symbol` column, it works based on the `topic` column. This would require changes to the `Core.ServiceSupervisor`'s functions like `update_status/4` or `fetch_symbols_to_start/2`, we could update them to accept column name but that would need to be passed through other functions. We can see that this is probably not the best approach and the further we will get the more complex it will become. The second issue would be that we are registering all processes with names and that can be problematic as the list of processes will start to grow(as we can imagine in the case of the `data_warehouse` application).

The better approach would be to mix the DynamicSupervisor together with Registry.

The `DynamicSupervisor` will supervise the `Subscriber.Worker`s and instead of keeping track of them by registering them using atoms we will start them `:via` Elixir `Registry`.

We will add all functionality that we implemented for `naive` and `streamer` applications. We will provide the functions to start and stop storing data on passed PubSub topics as well as store those topics inside the database so storing will be autostarted.

14.7.1 Create `subscriber_settings` table

To provide autostarting function we need to create a new migration that will create the `subscriber_settings` table:

```
$ cd apps/data_warehouse
$ mix ecto.gen.migration create_subscriber_settings
* creating priv/repo/migrations/20210227230123_create_subscriber_settings.exs
```

At this moment we can copy the code to create the `settings` table(enum and index as well) from the `streamer` application and tweak it to fit the `data_warehouse` application. So the first important change (besides updating namespaces from `Streamer` to `DataWarehouse`) will be to make a note that we have a setting per topic - not per symbol as for the `naive` and `streamer` applications:

```
# /apps/data_warehouse/priv/repo/migrations/20210227230123_create_subscriber_settings.exs
defmodule DataWarehouse.Repo.Migrations.CreateSubscriberSettings do
  use Ecto.Migration

  alias DataWarehouse.Schema.SubscriberStatusEnum

  def change do
    SubscriberStatusEnum.create_type()

    create table(:subscriber_settings, primary_key: false) do
      add(:id, :uuid, primary_key: true)
      add(:topic, :text, null: false)
      add(:status, SubscriberStatusEnum.type(), default: "off", null: false)

      timestamps()
    end

    create(unique_index(:subscriber_settings, [:topic]))
  end
end
```

Both schema and enum will be almost identical to the ones from the `streamer` application - we can simply copy those files and apply basic tweaks like updating the namespace:

```
$ cp apps/streamer/lib/streamer/schema/settings.ex \
apps/data_warehouse/lib/data_warehouse/schema/subscriber_settings.ex
$ cp apps/streamer/lib/streamer/schema/streaming_status_enum.ex \
apps/data_warehouse/lib/data_warehouse/schema/subscriber_status_enum.ex
```

Remember about updating the `symbol` column to `topic` as well as table name inside the `DataWarehouse.Schema.SubscriberSettings`:

```
# /apps/data_warehouse/lib/data_warehouse/schema/subscriber_settings.ex
defmodule DataWarehouse.Schema.SubscriberSettings do
  use Ecto.Schema

  alias DataWarehouse.Schema.SubscriberStatusEnum

  @primary_key {:id, :binary_id, autogenerate: true}
```

```
  schema "subscriber_settings" do
    field(:topic, :string)
    field(:status, SubscriberStatusEnum)

    timestamps()
  end
end
```

Inside `apps/data_warehouse/lib/data_warehouse/schema/subscriber_status_enum.ex` we need to swap references of `Streamer` to `DataWarehouse` and references of `StreamingStatusEnum` to `SubscriberStatusEnum`:

```
# /apps/data_warehouse/lib/data_warehouse/schema/subscriber_status_enum.ex
import EctoEnum

defenum(DataWarehouse.Schema.SubscriberStatusEnum, :subscriber_status, [:on, :off])
```

Don't forget to run the migration:

```
$ mix ecto.migrate
```

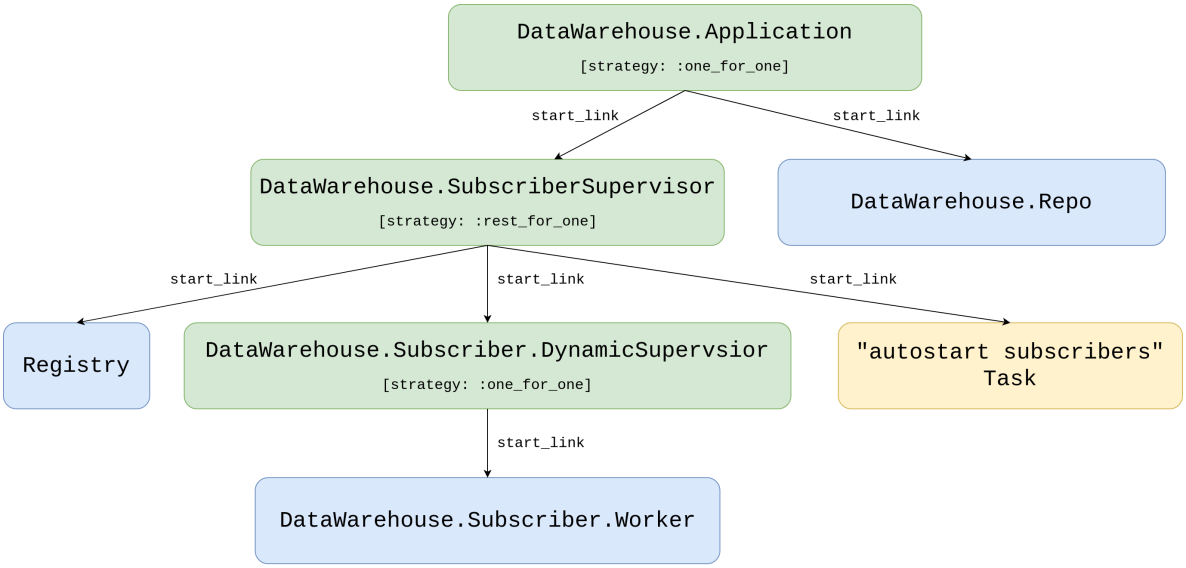
At this moment we have all pieces in place to execute queries on our new table. In this place, we can think about the seeding script. For the `data_warehouse` specifically, we won't need to provide that script as we don't know in advance what topic names we will use. Instead of seeding settings in advance, our code will "upsert"(using `insert` function) settings when `start_storing/1` or `stop_storing/1` are called.

14.7.2 Redesign supervision using Registry

We can now focus on drafting a supervision tree for the `data_warehouse` application. At this moment we have only the `DataWarehouse.Subscriber.Worker` and the `DataWarehouse.Application` modules.

As it was with the case of `naive` and `streamer` applications, we will need an additional level of supervision to cater for "autostarting" `Task` as well as, in the case of the `data_warehouse` application the `Registry`.

The full supervision tree will look as follows:



Everything looks very similar to the supervision tree that we created in the `streamer` and the `naive` applications but there's an additional `Registry` that is supervised by the `SubscriberSupervisior` process.

The idea is that inside the `Worker` module's `start_link/1` we will register worker processes using :via tuple. Internally, GenServer will utilize `Registry`'s functions like `register_name/2` to add process to the registry under the `topic` string. This way we will be able to retrieve PIDs assigned to topics using those topic strings instead of registering each worker process with an atom name.

Just as previously the `DynamicSupervisor` will be in charge of supervising the `Worker` processes and it won't be even aware that we are using the `Registry` to keep track of `topic => PID` association.

14.7.3 Create the `DataWarehouse.Subscriber.DynamicSupervisor` module

Let's start by creating a new file called `dynamic_supervisor.ex` inside the `apps/data_warehouse/lib/data_warehouse/subscriber` directory and put default dynamic supervisor implementation inside:

```
# /apps/data_warehouse/lib/data_warehouse/subscriber/dynamic_supervisor.ex
defmodule DataWarehouse.Subscriber.DynamicSupervisor do
  use DynamicSupervisor

  def start_link(_arg) do
    DynamicSupervisor.start_link(__MODULE__, [], name: __MODULE__)
  end

  def init(_arg) do
    DynamicSupervisor.init(strategy: :one_for_one)
```

```
  end
end
```

As we will put all our logic related to autostarting, starting and stopping inside this module we can already add aliases, import and require:

```
# /apps/data_warehouse/lib/data_warehouse/subscriber/dynamic_supervisor.ex
  require Logger

  alias DataWarehouse.Repo
  alias DataWarehouse.Schema.SubscriberSettings
  alias DataWarehouse.Subscriber.Worker

  import Ecto.Query, only: [from: 2]

  @registry :subscriber_workers
```

Additionally, we added the `@registry` module attribute that we will use to retrieve PID for the specific topic.

We can move on to implementing `autostart_workers/0` which will look very similar to the ones that we implemented in the `streamer` and the `naive` applications:

```
  # /apps/data_warehouse/lib/data_warehouse/subscriber/dynamic_supervisor.ex
  ...
  def autostart_workers do
    Repo.all(
      from(s in SubscriberSettings,
        where: s.status == "on",
        select: s.topic
      )
    )
    |> Enum.map(&start_child/1)
  end

  defp start_child(args) do
    DynamicSupervisor.start_child(
      __MODULE__,
      {Worker, args}
    )
  end
```

We can see that we are querying the database for a list of `topics`(not symbols) and we are calling `start_child/2` for each result.

The `start_worker/1` is where the `Registry` will shine as we won't need to check is there already a process running for that topic - we can leave that check to the `Registry`. If there's a process already running for that topic it will just return a tuple starting with `:error` atom:

```
# /apps/data_warehouse/lib/data_warehouse/subscriber/dynamic_supervisor.ex
...
def start_worker(topic) do
  Logger.info("Starting storing data from #{topic} topic")
  update_status(topic, "on")
  start_child(topic)
end
...
defp update_status(topic, status)
     when is_binary(topic) and is_binary(status) do
  %SubscriberSettings{
    topic: topic,
    status: status
  }
  |> Repo.insert(
    on_conflict: :replace_all,
    conflict_target: :topic
  )
end
```

As we are not seeding the database with the default settings we will use the `insert/2` function with options(as previously) to make it work as it would be an "upsert" function.

Last function in this module will be `stop_worker/1` which uses private `stop_child/1` function. The `stop_child/1` function shows how to retrieve `PID` of the process assigned to the passed `topic`:

```
# /apps/data_warehouse/lib/data_warehouse/subscriber/dynamic_supervisor.ex
...
def stop_worker(topic) do
  Logger.info("Stopping storing data from #{topic} topic")
  update_status(topic, "off")
  stop_child(topic)
end
...
defp stop_child(args) do
  case Registry.lookup(@registry, args) do
    [{pid, _}] -> DynamicSupervisor.terminate_child(__MODULE__, pid)
    _ -> Logger.warn("Unable to locate process assigned to #{inspect(args)}")
  end
end
```

That is a full implementation of the `DataWarehouse.Subscriber.DynamicSupervisor` module and it's almost as slim as one from the last chapter where we leveraged macros to achieve that lightness. Using the `Registry` is the preferred way to manage a list of identifiable processes. We won't run into an issue of overusing the atoms(as they are not garbage collected, we could hit that limit sooner or later).

14.7.4 Register Worker processes using :via

The above `DynamicSupervisor` module assumes that Workers are registered inside the `Registry` - to make this happen we will need to update the `start_link/1` function of the `DataWarehouse.Subscriber.Worker` module:

```
# /apps/data_warehouse/lib/data_warehouse/subscriber/worker.ex
...
def start_link(topic) do
  GenServer.start_link(
    __MODULE__,
    topic,
    name: via_tuple(topic)
  )
end
...
defp via_tuple(topic) do
  {:via, Registry, {:subscriber_workers, topic}}
end
...
```

Passing the `:name` option to the `GenServer`'s `start_link/3` function we instruct it to utilize the `Registry` module to register processes under topic names.

14.7.5 Create a new supervision level for Registry, Task and the DynamicSupervisor

We have the lowest level modules - the `Worker` and the `DynamicSupervisor` implemented - time to add a new `Supervisor` that will start the `Registry`, the `DynamicSupervisor`, and the autostart storing `Task`. First create a new file called `subscriber_supervisor.ex` inside the `apps/data_warehouse/lib/data_warehouse` directory:

```
# /apps/data_warehouse/lib/data_warehouse/subscriber_supervisor.ex
defmodule DataWarehouse.SubscriberSupervisor do
  use Supervisor

  alias DataWarehouse.Subscriber.DynamicSupervisor

  @registry :subscriber_workers
```

```
  def start_link(_args) do
    Supervisor.start_link(__MODULE__, [], name: __MODULE__)
  end

  def init(_args) do
    children = [
      {Registry, [keys: :unique, name: @registry]},
      {DynamicSupervisor, []},
      {Task,
       fn ->
         DynamicSupervisor.autostart_workers()
       end}
    ]

    Supervisor.init(children, strategy: :rest_for_one)
  end
end
```

The important part here will be to match the `Registry` name to the one defined inside the `DynamicSupervisor` and the `Worker` modules.

14.7.6 Link the `SubscriberSupervisor` to the `Application`

We need to update the `DataWarehouse.Application` module to start our new `DataWarehouse.SubscriberSupervisor` process as well as register itself under name matching to its module(just for consistency with other applications):

```
  # /apps/data_warehouse/lib/data_warehouse/application.ex
  ...
  def start(_type, _args) do
    children = [
      {DataWarehouse.Repo, []},
      {DataWarehouse.SubscriberSupervisor, []} # <= new module added
    ]

    # See https://hexdocs.pm/elixir/Supervisor.html
    # for other strategies and supported options
    opts = [strategy: :one_for_one, name: __MODULE__] # <= name updated
    Supervisor.start_link(children, opts)
  end
  ...
```

14.7.7 Add interface

The final step will be to add an interface to the `DataWarehouse` application to start and stop storing:

```
# /apps/data_warehouse/lib/data_warehouse.ex
alias DataWarehouse.Subscriber.DynamicSupervisor

def start_storing(stream, symbol) do
  to_topic(stream, symbol)
  |> DynamicSupervisor.start_worker()
end

def stop_storing(stream, symbol) do
  to_topic(stream, symbol)
  |> DynamicSupervisor.stop_worker()
end

defp to_topic(stream, symbol) do
  [stream, symbol]
  |> Enum.map(&String.upcase/1)
  |> Enum.join(":")
end
```

Inside the above functions, we are just doing a couple of sanity checks on the case of the passed arguments assuming that both topics and stream are uppercase.

14.7.8 Test

The interface above was the last step in our implementation, we can now test that all works as expected:

```
$ iex -S mix
...
iex(1)> DataWarehouse.start_storing("TRADE_EVENTS", "NEOUSDT")
19:34:00.740 [info]  Starting storing data from TRADE_EVENTS:NEOUSDT topic
19:34:00.847 [info]  DataWarehouse worker is subscribing to TRADE_EVENTS:NEOUSDT
{:ok, #PID<0.429.0>}
iex(2)> DataWarehouse.start_storing("TRADE_EVENTS", "NEOUSDT")
19:34:04.753 [info]  Starting storing data from TRADE_EVENTS:NEOUSDT topic
{:error, {:already_started, #PID<0.459.0>}}
iex(3)> DataWarehouse.start_storing("ORDERS", "NEOUSDT")
19:34:09.386 [info]  Starting storing data from ORDERS:NEOUSDT topic
19:34:09.403 [info]  DataWarehouse worker is subscribing to ORDERS:NEOUSDT
{:ok, #PID<0.431.0>}
BREAK: (a)bort (A)bort with dump (c)ontinue (p)roc info (i)nfo
```

```
       (l)oaded (v)ersion (k)ill (D)b-tables (d)istribution
^C%
$ iex -S mix
...
19:35:30.058 [info]  DataWarehouse worker is subscribing to TRADE_EVENTS:NEOUSDT
19:35:30.062 [info]  DataWarehouse worker is subscribing to ORDERS:NEOUSDT
# autostart works ^^^
iex(1)> Naive.start_trading("NEOUSDT")
19:36:45.316 [info]  Starting Elixir.Naive.SymbolSupervisor worker for NEOUSDT
19:36:45.417 [info]  Starting new supervision tree to trade on NEOUSDT
{:ok, #PID<0.419.0>}
iex(3)>
19:36:47.484 [info]  Initializing new trader(1615221407466) for NEOUSDT
iex(2)> Streamer.start_streaming("NEOUSDT")
16:37:39.660 [info]  Starting Elixir.Streamer.Binance worker for NEOUSDT
{:ok, #PID<0.428.0>}
...
iex(3)> DataWarehouse.stop_storing("trade_events", "NEOUSDT")
19:39:26.398 [info]  Stopping storing data from trade_events:NEOUSDT topic
:ok
iex(4)> DataWarehouse.stop_storing("trade_events", "NEOUSDT")
19:39:28.151 [info]  Stopping storing data from trade_events:NEOUSDT topic
19:39:28.160 [warn]  Unable to locate process assigned to "trade_events:NEOUSDT"
:ok
iex(5)> [{pid, nil}] = Registry.lookup(:subscriber_workers, "ORDERS:NEOUSDT")
[{#PID<0.417.0>, nil}]
iex(6)> Process.exit(pid, :crash)
true
16:43:40.812 [info]  DataWarehouse worker is subscribing to ORDERS:NEOUSDT
```

As we can see even this simple implementation handles starting, autostarting, and stopping. It also gracefully handles starting workers when one is already running as well as stopping when there none running.

As a challenge, you could update the `naive` and the `streamer` application to use the `Registry` and remove `Core.ServiceSupervisor` module as it was superseded by the above solution - here's the link to PR(pull request) that sums up the required changes.

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

Source code for this chapter can be found at Github

Chapter 15

Backtest trading strategy

15.1 Objectives

- overview of requirements
- implement the storing task
- test the backtesting

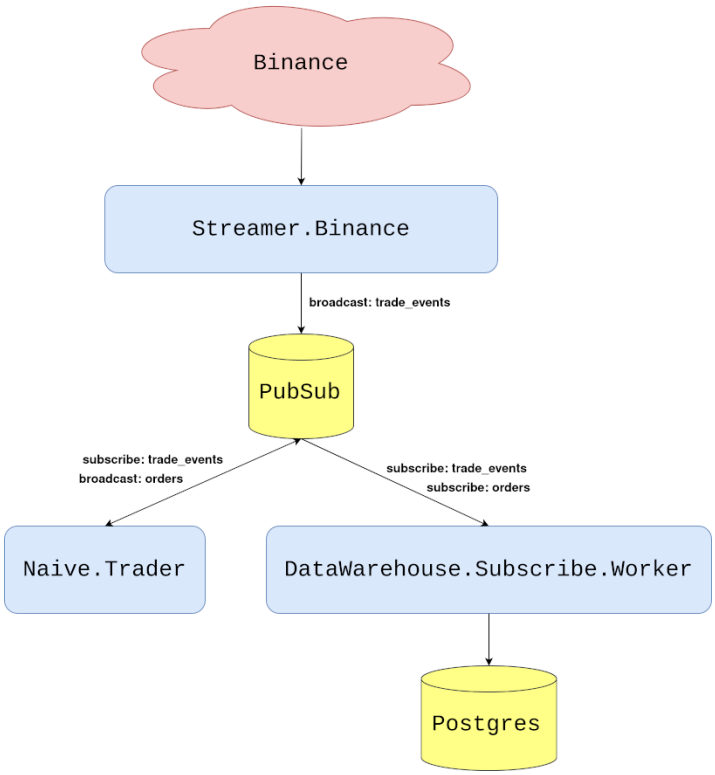
15.2 Overview of requirements

In the last chapter, we started storing trade events and orders in the database which will be crucial for backtesting, which we will focus on in this chapter.

Backtesting is a procedure of running historical data through the system and observing how our strategy would perform as if we would run it "in the past". Backtesting works on assumption that the market will behave in a similar fashion in the future as it was in the past.

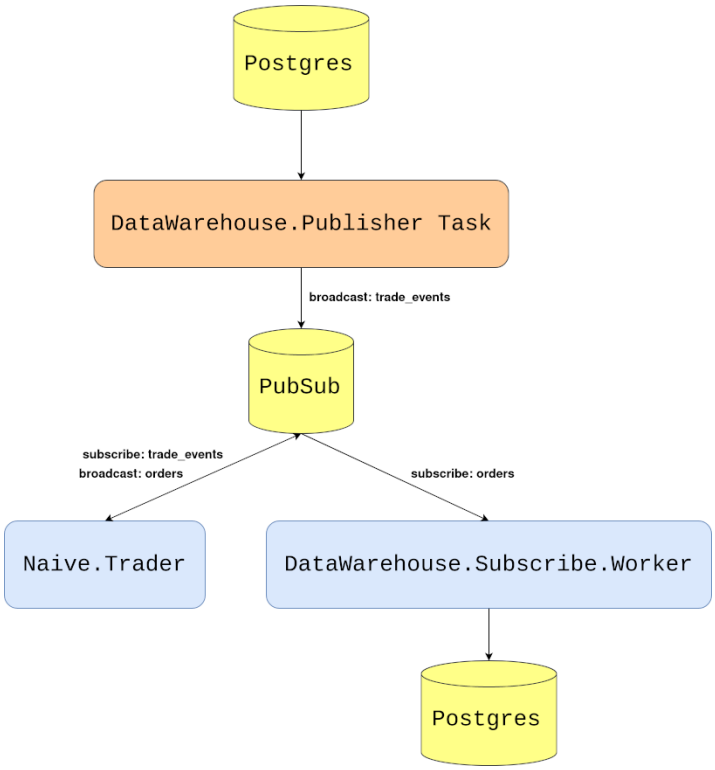
At this moment we are receiving the trade events from the Binance through WebSocket. The `Streamer.Binance` process is handling those messages by parsing them from JSON string to map, then converting them to structs and broadcasting them to the `TRADE_EVENTS:#{symbol}` PubSub topic. The `Naive.Trader` subscribes to the `TRADE_EVENTS:#{symbol}` topic and takes decisions based on incoming data. As it places buy and sell orders it broadcasts them to the `ORDERS:#{symbol}` PubSub topic. The `DataWarehouse.Subscriber.Worker` processes subscribe to both trade events and orders topics and store incoming data inside the database.

We can visualize this flow like that:



To backtest we can substitute the `Streamer.Binance` process with a `Task` that will `stream` trade events' data from the database and broadcasts it to the `TRADE_EVENTS:#{symbol}` PubSub topic(the same topic as the `Streamer.Binance` process).

From the perspective of the `Naive.Trader` it *does not* make any difference who is broadcasting those trade events. This should be a clear indication of the value of publish/subscribe model that we implemented from the beginning. It allows us to swap producer and consumers freely to backtest our trading strategies:



15.3 Implement the storing task

We will start by creating a new file called `publisher.ex` inside the `apps/data_warehouse/lib/data_warehouse` directory. We will start by implementing the basic `Task` behavior:

```
# /apps/data_warehouse/lib/data_warehouse/publisher.ex
defmodule DataWarehouse.Publisher do
  use Task

  def start_link(arg) do
    Task.start_link(__MODULE__, :run, [arg])
  end

  def run(arg) do
    # ...
  end
end
```

To be able to query the database we will import `Ecto` and require `Logger` for logging:

```
# /apps/data_warehouse/lib/data_warehouse/publisher.ex
...
import Ecto.Query, only: [from: 2]

require Logger
...
```

We can now modify the `run/1` function to expect specific `type`, `symbol`, `from`, `to` and `interval`:

```
# /apps/data_warehouse/lib/data_warehouse/publisher.ex
...
def run(%{
      type: :trade_events,
      symbol: symbol,
      from: from,
      to: to,
      interval: interval
    }) do
  ...
```

Inside the body of the `run/1` function, first, we will convert `from` and `to` Unix timestamps by using private helper functions as well as make sure that the passed symbol is uppercase:

```
# /apps/data_warehouse/lib/data_warehouse/publisher.ex
...
def run(%{
      ...
    }) do
  symbol = String.upcase(symbol)

  from_ts =
    "#{from}T00:00:00.000Z"
    |> convert_to_ms()

  to_ts =
    "#{to}T23:59:59.000Z"
    |> convert_to_ms()
end
...
defp convert_to_ms(iso8601DateString) do
  iso8601DateString
  |> NaiveDateTime.from_iso8601!()
```

```
    |> DateTime.from_naive!("Etc/UTC")
    |> DateTime.to_unix()
    |> Kernel.*(1000)
  end
```

Next, we will select data from the database but because of possibly hundreds of thousands of rows being selected and because we are broadcasting them to the PubSub every x ms it could take a substantial amount of time to `broadcast` all of them. Instead of `select`ing data and storing all of it in the memory, we will use `Repo.stream/1` function to keep `broadcast`ing it on the go. Additionally, we will add `index` to the data to be able to log info messages every 10k messages. The last thing that we need to define will be the timeout value - the default value is 5 seconds and we will change it to `:infinity`:

```
# /apps/data_warehouse/lib/data_warehouse/publisher.ex
def run(%{
      ...
    }) do
  ...
  DataWarehouse.Repo.transaction(
    fn ->
      from(te in DataWarehouse.Schema.TradeEvent,
        where:
          te.symbol == ^symbol and
            te.trade_time >= ^from_ts and
            te.trade_time < ^to_ts,
        order_by: te.trade_time
      )
      |> DataWarehouse.Repo.stream()
      |> Enum.with_index()
      |> Enum.map(fn {row, index} ->
        :timer.sleep(interval)

        if rem(index, 10_000) == 0 do
          Logger.info("Publisher broadcasted #{index} events")
        end

        publish_trade_event(row)
      end)
    end,
    timeout: :infinity
  )

  Logger.info("Publisher finished streaming trade events")
end
```

Finally, the above code uses the `publish_trade_event/1` helper function which converts DataWarehouse's TradeEvent to the Streamer's TradeEvent to broadcast the same structs as the `streamer` application:

```
  # /apps/data_warehouse/lib/data_warehouse/publisher.ex
  ...
  defp publish_trade_event(%DataWarehouse.Schema.TradeEvent{} = trade_event) do
    new_trade_event =
      struct(
        Streamer.Binance.TradeEvent,
        trade_event |> Map.to_list()
      )

    Phoenix.PubSub.broadcast(
      Streamer.PubSub,
      "TRADE_EVENTS:#{trade_event.symbol}",
      new_trade_event
    )
  end
```

We also need to remember about keeping the interface tidy so we will add `publish_data` to the `DataWarehouse` module:

```
  # /apps/data_warehouse/lib/data_warehouse.ex
...
  def publish_data(args) do
    DataWarehouse.Publisher.start_link(args)
  end
...
```

This finishes our implementation - we should be able to stream trade events from the database to the PubSub using the above Task which we will do below.

15.4 Test the backtesting

For consistency and ease of testing/use, I prepared an compressed single data of trade events for XRPUSDT(2019-06-03). We can download that file from GitHub using `wget`:

```
$ cd /tmp
$ wget https://github.com/Cinderella-Man/binance-trade-events \
/raw/master/XRPUSDT/XRPUSDT-2019-06-03.csv.gz
```

We can now uncompress the archive and load those trade events into our database:

```
$ gunzip XRPUSDT-2019-06-03.csv.gz
$ PGPASSWORD=hedgehogSecretPassword psql -Upostgres -h localhost -ddata_warehouse \
-c "\COPY trade_events FROM '/tmp/XRPUSDT-2019-06-03.csv' WITH (FORMAT csv, delimiter ';');"
COPY 206115
```

The number after the word `COPY` in the response indicates the number of rows that got copied into the database.

We can now give it a try and run full backtesting but first let's clean the orders table:

```
$ psql -Upostgres -h127.0.0.1
Password for user postgres:
...
postgres=# \c data_warehouse
You are now connected to database "data_warehouse" as user "postgres".
data_warehouse=# DELETE FROM orders;
DELETE ...
```

We can now start a new iex session where we will start trading(the `naive` application) as well as storing orders(the `data_warehouse` application) and instead of starting the `Streamer.Binance` worker we will start the `DataWarehouse.Publisher` task with arguments matching the imported day and symbol:

```
$ iex -S mix
...
iex(1)> DataWarehouse.start_storing("ORDERS", "XRPUSDT")
19:17:59.596 [info]  Starting storing data from ORDERS:XRPUSDT topic
19:17:59.632 [info]  DataWarehouse worker is subscribing to ORDERS:XRPUSDT
{:ok, #PID<0.417.0>}
iex(2)> Naive.start_trading("XRPUSDT")
19:18:16.293 [info]  Starting Elixir.Naive.SymbolSupervisor worker for XRPUSDT
19:18:16.332 [info]  Starting new supervision tree to trade on XRPUSDT
{:ok, #PID<0.419.0>}
19:18:18.327 [info]  Initializing new trader(1615288698325) for XRPUSDT
iex(3)> DataWarehouse.publish_data(%{
  type: :trade_events,
  symbol: "XRPUSDT",
  from: "2019-06-02",
  to: "2019-06-04",
  interval: 5
})
{:ok, #PID<0.428.0>}
19:19:07.532 [info]  Publisher broadcasted 0 events
19:19:07.534 [info]  The trader(1615288698325) is placing a BUY order for
XRPUSDT @ 0.44391, quantity: 450.5
```

```
19:19:07.749 [info]  The trader(1615288698325) is placing a SELL order for
XRPUSDT @ 0.44426, quantity: 450.5.
...
19:20:07.568 [info]  Publisher broadcasted 10000 events
...
19:21:07.571 [info]  Publisher broadcasted 20000 events
19:22:07.576 [info]  Publisher broadcasted 30000 events
...
19:39:07.875 [info]  Publisher broadcasted 200000 events
19:39:44.576 [info]  Publisher finished streaming trade events
```

From the above log, we can see that it took about 20 minutes to run 206k records through the system(a lot of that time[17+ minutes] was indeed the 5ms `sleep`).

After the streaming finished we can check out the orders table inside the database to figure out how many trades we made and what income have they generated.

```
$ psql -Upostgres -h127.0.0.1
Password for user postgres:
...
postgres=# \c data_warehouse
You are now connected to database "data_warehouse" as user "postgres".
data_warehouse=# SELECT COUNT(*) FROM orders;
 count
-------
   224
(1 row)
```

By looking at the orders we can figure out some performance metrics but that's less than perfect to get answers to simple questions like "what's the performance of my strategy?". We will address that and other concerns in future chapters.

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

Source code for this chapter can be found at Github

Chapter 16

End-to-end testing

16.1 Objectives

- decide on the tested functionality
- implement basic test
- introduce environment based config files
- add convenience aliases
- cache initial seed data inside a file
- update seeding scripts to use the BinanceMock
- introduce the core application

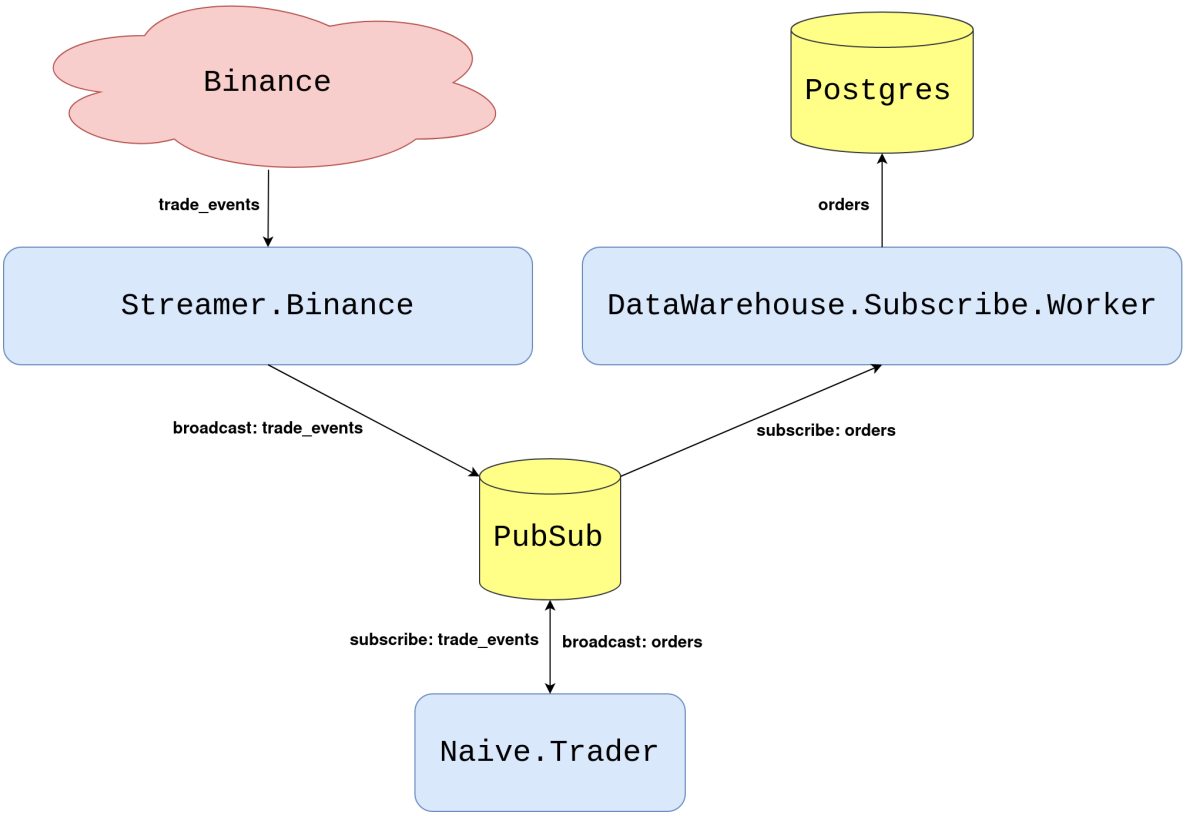
16.2 Decide on the tested functionality

We've reached the stage where we have a decent solution in place, and to ensure that it's still working correctly after any future refactoring, we will add tests. We will start with the "integration"/"end-to-end"(E2E) test, which will confirm that the whole "trading" works.

To perform tests at this level, we will need to orchestrate databases together with processes and broadcast trade events from within the test to cause our trading strategy to place orders. We will be able to confirm the right behavior by checking the database after running the test.

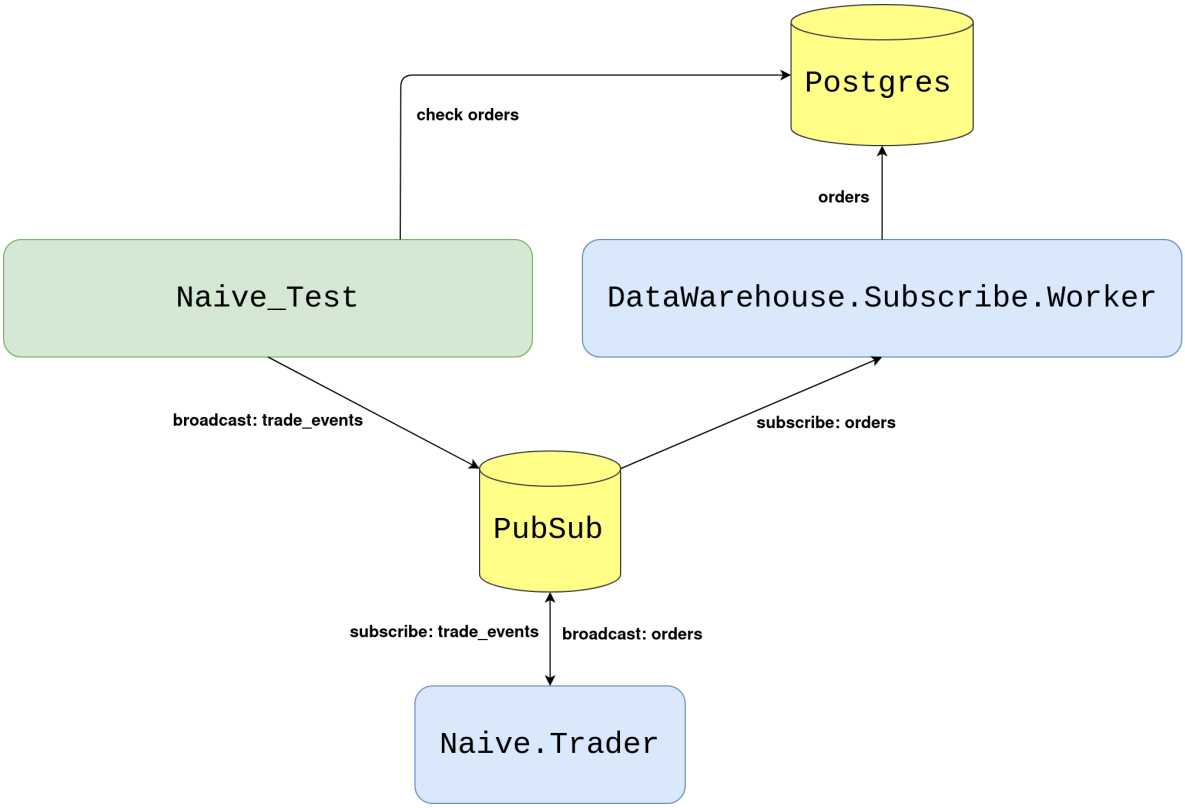
You should have the **least** amount of this type of test as they are very brittle and require substantial effort to set up and maintain. My personal rule of thumb is that only major "happy paths" should be tested this way.

Let's look at the current flow of data through our system:



Currently, the `Streamer.Binance` establishes a WebSocket connection with Binance. It decodes the incoming trade events and broadcasts them to PubSub(`TRADE_EVENTS:#{symbol}` topics). PubSub then sends them to the `Naive.Trader` processes. As those processes place orders(or orders get filled), they broadcast orders to the PubSub(`ORDERS:#{symbol}` topics). The `DataWarehouse.Subscriber.Worker` processes subscribe to the broadcasted orders and store them in the database.

What we could do is to stop all the `Streamer.Binance` processes and broadcast trade events directly from the test. We would then be able to fine-tune the prices inside those events to run through the full trade cycle:



This would allow us to fetch orders from the database to confirm that trading indeed happened.

16.3 Implement basic test

We will place our test inside the `NaiveTest` module inside the `apps/naive/test/naive_test.exs` file. First, we will need to alias multiple modules that we will use to either initialize or confirm the results:

```
# /apps/naive/test/naive_test.exs
...
alias DataWarehouse.Schema.Order
alias Naive.Schema.Settings, as: TradingSettings
alias Streamer.Binance.TradeEvent

import Ecto.Query, only: [from: 2]
```

Now we can update the generated test to have a tag in front of it:

```
# /apps/naive/test/naive_test.exs
...
@tag integration: true
```

We will use this tag to select only this test when we are running the integration tests.

The first step will be to update the trading settings to values that will cause trading activity:

```
# /apps/naive/test/naive_test.exs
...
test "Naive trader full trade(buy + sell) test" do
  symbol = "XRPUSDT"

  # Step 1 - Update trading settings

  settings = [
    profit_interval: 0.001,
    buy_down_interval: 0.0025,
    chunks: 5,
    budget: 100.0
  ]

  {:ok, _} =
    TradingSettings
    |> Naive.Repo.get_by!(symbol: symbol)
    |> Ecto.Changeset.change(settings)
    |> Naive.Repo.update()
```

As we updated the trading settings, we can now start trading:

```
  # /apps/naive/test/naive_test.exs
  # `test` function continued
  ...
  # Step 2 - Start trading on symbol

  Naive.start_trading(symbol)
```

Before we start broadcasting events, we need to ensure that the `DataWarehouse` application will store resulting orders into the database:

```
    # /apps/naive/test/naive_test.exs
    # `test` function continued
    ...
    # Step 3 - Start storing orders

    DataWarehouse.start_storing("ORDERS", "XRPUSDT")
    :timer.sleep(5000)
```

Additionally, as seen in the above code, we need to allow some time(5 seconds above) to initialize trading and data storing processes.

We can now move on to broadcasting trade events:

```
    # /apps/naive/test/naive_test.exs
    # `test` function continued
    ...
    # Step 4 - Broadcast 9 events

    [
      # buy order palced @ 0.4307
      generate_event(1, "0.43183010", "213.10000000"),
      generate_event(2, "0.43183020", "56.10000000"),
      generate_event(3, "0.43183030", "12.10000000"),
      # event at the expected buy price
      generate_event(4, "0.4307", "38.92000000"),
      # event below the expected buy price
      # it should trigger fake fill event for placed buy order
      # and palce sell order @ 0.4319
      generate_event(5, "0.43065", "126.53000000"),
      # event below the expected sell price
      generate_event(6, "0.43189", "26.18500000"),
      # event at exact the expected sell price
      generate_event(7, "0.4319", "62.92640000"),
      # event above the expected sell price
      # it should trigger fake fill event for placed sell order
      generate_event(8, "0.43205", "345.14235000"),
      # this one should trigger buy order for a new trader process
      generate_event(9, "0.43210", "3201.86480000")
    ]
    |> Enum.each(fn event ->
      Phoenix.PubSub.broadcast(
        Core.PubSub,
```

```
      "TRADE_EVENTS:#{symbol}",
      event
    )

    :timer.sleep(10)
  end)

  :timer.sleep(2000)
```

The above code will broadcast trade events to the PubSub topic that the trader processes are subscribed to. It should cause 3 orders to be placed at specific prices. In the last step, we will confirm this by querying the database:

```
  # /apps/naive/test/naive_test.exs
  # `test` function continued
  ...
  # Step 5 - Check orders table

  query =
    from(o in Order,
      select: [o.price, o.side, o.status],
      order_by: o.inserted_at,
      where: o.symbol == ^symbol
    )

  [buy_1, sell_1, buy_2] = DataWarehouse.Repo.all(query)

  assert buy_1 == ["0.43070000", "BUY", "FILLED"]
  assert sell_1 == ["0.43190000", "SELL", "FILLED"]
  assert buy_2 == ["0.43100000", "BUY", "NEW"]
```

That finishes the test function. The final addition inside the `NaiveTest` module will be to add a private helper function that generates trade event for the passed values:

```
# /apps/naive/test/naive_test.exs
...

defp generate_event(id, price, quantity) do
  %TradeEvent{
    event_type: "trade",
    event_time: 1_000 + id * 10,
    symbol: "XRPUSDT",
    trade_id: 2_000 + id * 10,
    price: price,
```

```
      quantity: quantity,
      buyer_order_id: 3_000 + id * 10,
      seller_order_id: 4_000 + id * 10,
      trade_time: 5_000 + id * 10,
      buyer_market_maker: false
    }
  end
```

This finishes the implementation of the test, but as we are now using DataWarehouse's modules inside the Naive application, we need to add `data_warehouse` to the dependencies:

```
  # /apps/naive/mix.exs
  defp deps do
    [
      ...
      {:data_warehouse, in_umbrella: true, only: :test},
      ...
```

We could now run our new integration test, but it would be run against our current(development) databases. In addition, as we will need to reset all the data inside them before every test run, it could mean losing data. To avoid all of those problems, we will use separate databases for testing.

16.4 Introduce environment based config files

Currently, our new test is running in the test environment (the `MIX_ENV` environmental variable is set to `"test"` whenever we run `mix test`), but we do not leverage that fact to configure our application for example: to use test databases as mentioned above.

Configuration for our applications lives in `config/config.exs` configuration file. Inside it, we have access to the name of the environment, which we will utilize to place an environment based `import_config/1` function:

```
# /config/config.exs
# add the below at the end of the file
...
import_config "#{config_env()}.exs"
```

Now we will create multiple config files, one for each environment:

- `/config/dev.exs` for development:

```
# /config/dev.exs
import Config
```

- `/config/test.exs` for future "unit" testing:

```
# /config/test.exs
import Config
```

- `/config/integration.exs` for end-to-end testing:

```
# /config/integration.exs
import Config

config :streamer, Streamer.Repo, database: "streamer_test"

config :naive, Naive.Repo, database: "naive_test"

config :data_warehouse, DataWarehouse.Repo, database: "data_warehouse_test"
```

- `/config/prod.exs` for production:

```
# /config/prod.exs
import Config

config :naive,
  binance_client: Binance
```

After adding the above environment-based configuration files, our test will use the test databases.

There's one more remaining problem - we need to set those test databases before each test run, and as this process requires multiple steps, it's a little bit cumbersome.

16.5 Add convenience aliases

To be able to run our new integration test as easily as possible without bothering ourselves with all the database setup, we will introduce aliases in both the `streamer` and `naive` applications that will wrap seeding the databases:

```
# /apps/naive/mix.exs & /apps/streamer/mix.exs
  def project do
    [
      ...
      aliases: aliases()
    ]
  end

  defp aliases do
```

```
    [
      seed: ["run priv/seed_settings.exs"]
    ]
  end
```

Inside the main `mix.exs` file of our umbrella, we will use those with usual `ecto`'s commands like `ecto.create` and `ecto.migrate`:

```
  # /mix.exs
  def project do
    [
      ...
      aliases: aliases()
    ]
  end

  defp aliases do
    [
      setup: [
        "ecto.drop",
        "ecto.create",
        "ecto.migrate",
        "cmd --app naive --app streamer mix seed"
      ],
      "test.integration": [
        "setup",
        "test --only integration"
      ]
    ]
  end
```

We can now safely run our test:

```
$ MIX_ENV=integration mix test.integration
```

Wait... Why do we need to set the `MIX_ENV` before calling our alias?

So, as I mentioned earlier, the `mix test` command automatically assigns the `"test"` environment when called. However, our alias contains other commands like `mix ecto.create`, which without specifying the environment explicitly, would be run using the dev database. So we would set up the dev databases(drop, create, migrate & seed) and then run tests on the test databases.

So our test is now passing, but it relies on the database being setup upfront, which requires seeding using a couple of requests to the Binance API.

16.6 Cache initial seed data inside a file

Relying on the 3rd party API to be able to seed our database to run tests is a horrible idea. However, we can fix that by cache the response JSON in a file.

How will this data be tunneled into the test database?

In the spirit of limiting the change footprint, we could update the `BinanceMock` module to serve the cached data dependent on the flag - let's add that flag first:

```
# /config/config.exs
# add below lines under the `import Config` line
config :binance_mock,
  use_cached_exchange_info: false
```

```
# /config/integration.exs
# add below lines under the `import Config` line
config :binance_mock,
  use_cached_exchange_info: true
```

We can see how convenient it is to have a configuration file per environment - we enabled cached exchange info data only for the test environment.

Inside the `BinanceMock` module, we can now update the `get_exchange_info/0` function to use this configuration value to serve either cached or live exchange info response:

```
  # /apps/binance_mock/lib/binance_mock.ex
  def get_exchange_info() do
    case Application.get_env(:binance_mock, :use_cached_exchange_info) do
      true -> get_cached_exchange_info()
      _ -> Binance.get_exchange_info()
    end
  end

  # add this at the bottom of the module
  defp get_cached_exchange_info do
    {:ok, data} =
      File.cwd!()
      |> Path.split()
      |> Enum.drop(-1)
      |> Kernel.++([
        "binance_mock",
        "test",
        "assets",
        "exchange_info.json"
      ])
```

```
      |> Path.join()
      |> File.read()

    {:ok, Jason.decode!(data) |> Binance.ExchangeInfo.new()}
  end
```

As the `binance_mock` app wasn't using the `jason` package before, we need to add it to dependencies:

```
  # /apps/binance_mock/mix.exs
  defp deps do
    [
      ...
      {:jason, "~> 1.2"},
      ...
```

The above change will take care of flipping between serving the live/cached exchange info data, but we still need to manually save the current response to the file (to be used as a cached version later).

Let's open the IEx terminal to fetch the exchange info data and serialize it to JSON:

```
$ iex -S mix
iex(1)> Binance.get_exchange_info() |> elem(1) |> Jason.encode!
```

You should get the following error:

```
... of type Binance.ExchangeInfo (a struct), Jason.Encoder protocol must always be
explicitly implemented.

If you own the struct, you can derive the implementation specifying which fields
should be encoded to JSON:

    @derive {Jason.Encoder, only: [....]}
    defstruct ...

It is also possible to encode all fields, although this should be used carefully
to avoid accidentally leaking private information when new fields are added:

    @derive Jason.Encoder
    defstruct ...

Finally, if you dont own the struct you want to encode to JSON, you may use
Protocol.derive/3 placed outside of any module:

    Protocol.derive(Jason.Encoder, NameOfTheStruct, only: [...])
    Protocol.derive(Jason.Encoder, NameOfTheStruct)
```

In a nutshell, this means that the `jason` package doesn't know how to encode the `Binance.ExchangeInfo` struct. Ok, as we don't own this struct(it's a part of the `binance` package), we will follow the last suggestion and try to derive the `Jason.Encoder` module for the `Binance.ExchangeInfo` struct:

```
$ iex -S mix
iex(1)> require Protocol
Protocol
iex(2)> Protocol.derive(Jason.Encoder, Binance.ExchangeInfo)
warning: the Jason.Encoder protocol has already been consolidated, an implementation for
Binance.ExchangeInfo has no effect. If you want to implement protocols after compilation
or during tests, check the "Consolidation" section in the Protocol module documentation
  iex:2: (file)
```

Hmm... This didn't work again. The reason for it is a mechanism called the "Protocol consolidation". Long story short, Elixir at compilation time knows upfront what structs derive which protocols, and to speed things up, it consolidates them at that moment. To avoid this process being run in development, we can modify the main `mix.exs` file to disable it:

```
  # /mix.exs
  def project do
    [
      ...
      consolidate_protocols: Mix.env() == :prod
    ]
  end
```

We should now be able to encode JSON using the `jason` module:

```
$ mkdir apps/binance_mock/test/assets
$ iex -S mix
iex(1)> require Protocol
Protocol
iex(2)> Protocol.derive(Jason.Encoder, Binance.ExchangeInfo)
:ok
iex(3)> data = Binance.get_exchange_info() |> elem(1) |> Jason.encode!(pretty: true)
...
iex(4)> File.write("apps/binance_mock/test/assets/exchange_info.json", data)
:ok
```

So we have the BinanceMock updated to serve cached/live responses based on the configuration and cached exchange info response.

The last step is to ensure that seeding uses the BinanceMock module instead of using Binance directly to leverage the above implementation.

16.7 Update seeding scripts to use the BinanceMock

The seed settings script for the `Naive` application(apps/naive/priv/seed_settings.exs) already uses the `BinanceMock`.

Inside the `Streamer` application (apps/streamer/priv/seed_settings.exs) we can see that the `Binance` module is getting used. So we can update the fetching part of the script to the following to fix it:

```
# /apps/streamer/priv/seed_settings.exs
binance_client = Application.get_env(:streamer, :binance_client) # <= new

Logger.info("Fetching exchange info from Binance to create streaming settings")

{:ok, %{symbols: symbols}} = binance_client.get_exchange_info() # <= updated
```

We need to update the config to point to the `BinanceMock` for the `streamer` application in the same way as we do for the `naive` application:

```
# /config/config.exs
...
config :streamer,
  binance_client: BinanceMock, # <= added
  ecto_repos: [Streamer.Repo]
...
```

```
# /config/prod.exs
...
config :streamer,
  binance_client: Binance
```

as well as swap the `binance` to the `BinanceMock` inside the list of dependencies of the `Streamer` app:

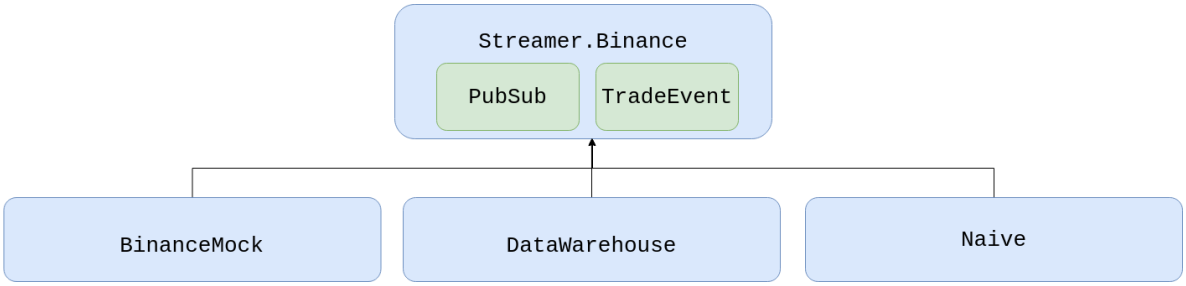
```
# /apps/streamer/mix.exs
...
  defp deps do
    [
      {:binance_mock, in_umbrella: true},
      ...
```

At this moment, we should be ready to run our test using the test database together with cached Binance response:

```
$ MIX_ENV=test mix test.integration
** (Mix) Could not sort dependencies. There are cycles in the dependency graph
```

And this is the moment when we will pay for cutting corners in the past. Let me explain. When we started this project, as we implemented communication using the PubSub topics, we put both the `PubSub` process(in the supervision tree)

and the `TradeEvent` struct inside the streamer application. The knock-on effect of this decision is that any other app in the umbrella that would like to use either `PubSub` or `TradeEvent` struct needs to depend on the `streamer` application:

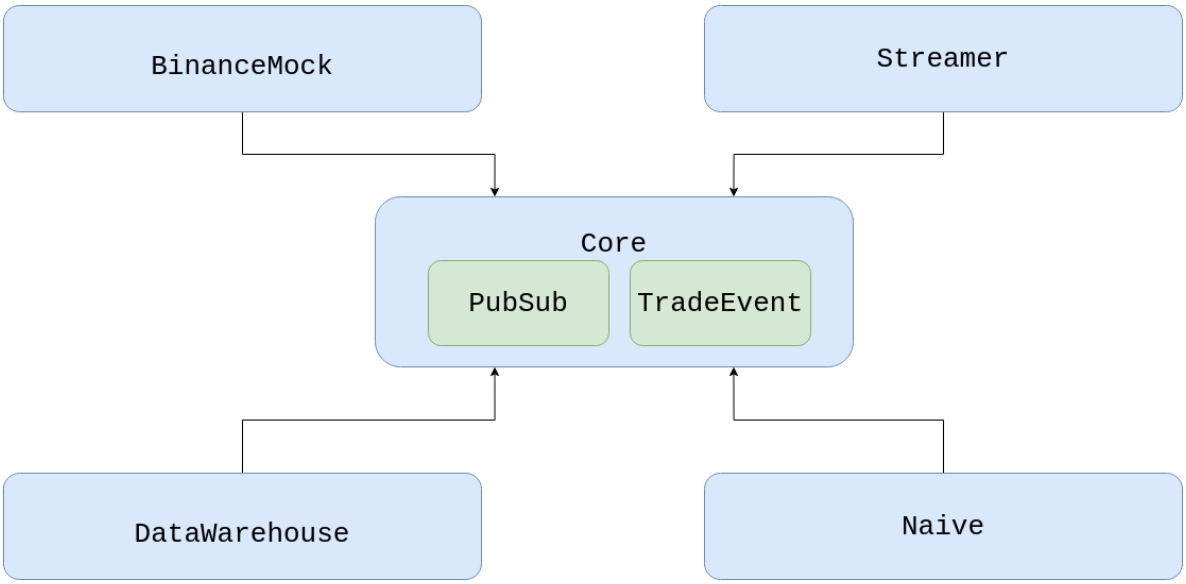


As we added the `binance_mock` application as a dependency of the streamer application, we created a dependency cycle.

This is quite typical in daily work as a software engineer. One of the common problems(besides naming things) is deciding where things belong. For example, do PubSub and TradeEvent belong in the Streamer app? Or maybe we should put it in the BinanceMock?

I believe that it should be neither of them as those applications use the PubSub and TradeEvent struct. I believe that they should be placed in neither of those applications as those applications are only using the struct and PubSub process.

What we should do instead is to create a new supervised application where we can attach the PubSub to the supervision tree and hold the system-wide structs(like TradeEvent) so every app can depend on it instead of each other:



16.8 Introduce the Core application

Let's start with create a new application:

```
$ cd apps
$ mix new --sup core
```

We can now create a new directory called `struct` inside the `apps/core/lib/core` directory and move the `TradeEvent` struct from the `streamer` app to it(in the same terminal or from the `apps` directory):

```
$ mkdir core/lib/core/struct
$ mv streamer/lib/streamer/binance/trade_event.ex core/lib/core/struct
```

Now we need to update the module to `Core.Struct.TradeEvent`:

```
# /apps/core/lib/core/struct/trade_event.ex
...
defmodule Core.Struct.TradeEvent do
```

As we moved the `TradeEvent` struct over to the `Core` application, we need to:

- update all places that reference the `Streamer.Binance.TradeEvent` to `Core.Struct.TradeEvent`
- add the `core` to the dependencies list of the `streamer` application
- remove the `streamer` from the dependencies list of all apps in the umbrella

The final step will be to move the PubSub process from the supervision tree of the Streamer application to the supervision of the Core application.

```
# /apps/streamer/lib/streamer/application.ex
  def start(_type, _args) do
    children = [
      ...
      {
        Phoenix.PubSub,
        name: Streamer.PubSub, adapter_name: Phoenix.PubSub.PG2
      }, # ^ remove it from here
      ...
    ]
```

```
# /apps/core/lib/core/application.ex
  def start(_type, _args) do
    children = [
      {
        Phoenix.PubSub,
```

```
        name: Core.PubSub, adapter_name: Phoenix.PubSub.PG2
      } # ^ add it here
    ]
```

As we changed the module name of the PubSub process(from `Streamer.PubSub` to `Core.PubSub`), we need to update all the places where it's referenced as well as add the `phoenix_pubsub` package to dependencies of the `core` application:

```
# /apps/core/mix.exs
defp deps do
  [
    {:phoenix_pubsub, "~> 2.0"}
  ]
end
```

We can now run our test that will use the test database as well as cached exchange info:

```
$ MIX_ENV=test mix test.integration
```

We should see a lot of setup log followed by a confirmation:

```
Finished in 0.03 seconds (0.00s async, 0.03s sync)
1 test, 0 failures, 1 excluded
```

This wraps up our implementation of the end-to-end test. In the next chapter, we will see how we would implement a unit test for our trading strategy.

The additional benefit of all the time that we put in the implementation of this test is that we won't need to remember how to set up local environment anymore as it's as simple as:

```
mix setup
```

Yay! :)

It's worth reiterating the downsides of the end-to-end tests:

- lack of visibility over what's broken - most of the time, we will see the result of error but not the error itself
- requirement of synchronous execution - they rely on the database(s), so they can't be run in parallel
- randomness/flakiness - as there's no feedback loop, we wait for a hardcoded amount of time, that we assume it's enough to finish initialization/execution - this can randomly fail

It's possible to implement a feedback loop and run our tests inside "sandbox" (transaction), but it's not worth it as we can invest that time into developing reliable unit tests.

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

Source code for this chapter can be found at Github

Chapter 17

Mox rocks

17.1 Objectives

- introduction to mock based tests
- add the Mox package
- investigate the `Naive.Trader` module
  - mock the `Binance` module
  - mock the `NaiveLeader` module
  - mock the `Phoenix.PubSub` module
  - mock the `Logger` module
- implement a test of the `Naive.Trader` module
- define an alias to run unit tests

17.2 Introduction to mock based tests

In the previous chapter, we've implemented the end-to-end test. It required a lot of prep work as well as we were able to see the downsides of this type of test clearly:

- we will be unable to run more than one end-to-end test in parallel as they rely on the database's state
- we need to set up the database before every test run
- we need to start processes in the correct order with the suitable parameters
- we need to wait a (guessed) hardcoded amount of time that it will take to finish the trading(this is extremely bad as it will cause randomly failing tests as people will make the time shorter to speed up tests)
- we wouldn't be able to quickly pinpoint which part error originated from as the test spans over a vast amount of the system
- logging was polluting our test output

How could we fix the above issues?

The most common way is to limit the scope of the test. Instead of testing the whole trading flow, we could focus on testing a single `Naive.Trader` process.

Focusing on a single trader process would remove the requirement for starting multiple processes before testing, but it would also bring its own challenges.

Let's look at a concrete example:

When the `Naive.Trader` process starts, it subscribes to the `TRADE_EVENTS:#{symbol}` PubSub topic. It also broadcasts updates of the orders it placed to the `ORDERS:#{symbol}` PubSub topic.

How could we break the link between the `Naive.Trader` and the PubSub(or any other module it depends on)?

We could utilize the trick that we used for the `Binance` module. We could create a module that provides the same functions as the `PubSub` module.

We know that the trader process calls `Phoenix.PubSub.subscribe/2` and `Phoenix.PubSub.broadcast/3` functions. We could implement a module that contains the same functions:

```
defmodule Test.PubSub do
    def subscribe(_, _), do: :ok
    def broadcast(_, _, _), do: :ok
end
```

The above module would satisfy the PubSub's functionality required by the `Naive.Trader` module, but this solution comes with a couple of drawbacks:

- it doesn't the passed values. It just ignores them, which is a missed opportunity to confirm that the PubSub module was called with the expected values.
- we can't define a custom implemention specific to test. This is possible by extending the implemention with test related returns(hack!)

Using the `mox` module would fix both of the problems mentioned above. With the `mox` module we can define add-hoc function implemention per test:

```
    # inside test file
    test ...
        Test.PubSubMock
        |> expect(:subscribe, fn (_module, "TRADE_EVENTS:XRPUSDT") -> :ok end)
        |> expect(:broadcast, fn (_module, "ORDERS:XRPUSDT", _order) -> :ok end)
```

There are multiple benefits to using the `mox` module instead of handcrafting the implementation:

- it allows defining functions that will pattern match values specific to each test(as in the case of the "usual" pattern matching, they will break when called with unexpected values)
- it allows defining implementations of the mocked functions based on incoming(test specific) values
- it can validate that all defined mocked functions have been called by the test
- it comes with its own tests, so we don't need to test it as it would need with our custom handcrafted mimicking module implementation

But there's a catch* ;)

For the `mox` to know what sort of functions the module provides, it needs to know its `behaviour`.

In Elixir, to define a behaviour of the module, we need to add the `@callback` attributes to it:

```
defmodule Core.Test.PubSub do
  @type t :: atom
  @type topic :: binary
  @type message :: term

  @callback subscribe(t, topic) :: :ok | {:error, term}
  @callback broadcast(t, topic, message) :: :ok | {:error, term}
end
```

A `behaviour` can be defined in a separate module if we are working with 3rd party module that doesn't provide it(like in the case of the `Phoenix.PubSub` module).

Note: The additional benefit of using the `behaviours` is that we could tell Elixir that our module *implements* the behaviour by adding the `@behaviour` attribute:

```
def MyPubSub do
    @behaviour Core.Test.PubSub
    ...
```

Using the above will cause Elixir to validate at compile time that the `MyPubSub` module implements all functions defined inside the `Core.Test.PubSub` module(otherwise it will raise compilation error).

Let's get back to the main topic. We figured out that we could mock all of the modules that the `Naive.Trader` depends on using the `mox` module.

But, how would we tell the `Naive.Trader` to use the mocked modules instead of the "real" ones when we run tests?

We could make all modules that the `Naive.Trader` depends on be dynamically injected from the configuration(based on the environment).

Another requirement to make `mox` work is to define the mocks upfront using the `Mox.defmock/2` function. It will dynamically define a new module that will be limited by the passed behaviour(we will only be able to mock[inside tests] functions defined as a part of that behaviour).

To sum up, there are a few steps to get the `mox` running:

- implement behaviours that we would like to mock(as most of the packages[like `Phoenix.PubSub`] are not coming with those)
- define mock modules using the `Mox.defmock` function
- modify the application's configuration to use the mocked module(s)
- specify mocked module's expectation inside the test

Let's move to the implementation.

17.3 Add the `mox` package

First let's add the `mox` package to the `naive` application's dependencies:

```
# /apps/naive/mix.exs
  ...
  defp deps do
    [
      ...
      {:mox, "~> 1.0", only: :test},
      ...
```

We can now run `mix deps.get` to fetch the `mox` package.

17.4 Investigate the `Naive.Trader` module

Let's investigate the `Naive.Trader` module(`/apps/naive/lib/naive/trader.ex`). We are looking for all calls to other modules - we can see:

- `Logger.info/2`
- `Phoenix.PubSub.subscribe/2`
- `@binance_client.order_limit_buy/4`
- `Naive.Leader.notify/2`
- `@binance_client.get_order/3`
- `@binance_client.order_limit_sell/4`
- `Phoenix.PubSub.broadcast/3`

So the `Naive.Trader` relies on four modules:

- `Logger`
- `Phoenix.PubSub`
- `Naive.Leader`
- `@binance_client`(either `Binance` or `BinanceMock`)

We will need to work through them one by one.

17.4.1 Mock the `Binance` module

Let's start with the binance client, as it's already a dynamic value based on the configuration.

Neither the `Binance` nor the `BinanceMock`(our dummy implementation) module doesn't provide a behaviour - let's fix that by defining the `@callback` attributes at the top of the `BinanceMock` module before the structs:

```
# /apps/binance_mock/lib/binance_mock.ex
  ...
  alias Binance.Order
  alias Binance.OrderResponse
  alias Core.Struct.TradeEvent

  @type symbol :: binary
  @type quantity :: binary
  @type price :: binary
  @type time_in_force :: binary
  @type timestamp :: non_neg_integer
  @type order_id :: non_neg_integer
  @type orig_client_order_id :: binary
  @type recv_window :: binary

  @callback order_limit_buy(
    symbol,
    quantity,
    price,
    time_in_force
  ) :: {:ok, %OrderResponse{}} | {:error, term}

  @callback order_limit_sell(
    symbol,
    quantity,
    price,
    time_in_force
  ) :: {:ok, %OrderResponse{}} | {:error, term}

  @callback get_order(
    symbol,
    timestamp,
    order_id,
    orig_client_order_id | nil,
    recv_window | nil
  ) :: {:ok, %Order{}} | {:error, term}
```

In the above code, we added three `@callback` attributes that define the binance client behaviour. For clarity, we defined

a distinct type for each of the arguments.

As we now have a binance client behaviour defined, we can use it to define a mock using the `Mox.defmock/2` function inside the `test_helper.exs` file of the `naive` application:

```
# /apps/naive/test/test_helper.exs
ExUnit.start()

Application.ensure_all_started(:mox) #1

Mox.defmock(Test.BinanceMock, for: BinanceMock) #2
```

First(#1), we need to ensure that the `mox` application has been started. Then(#2), we can tell the `mox` package to define the `Test.BinanceMock` module based on the `BinanceMock` behaviour.

As we defined the binance client behaviour and mock, we can update our configuration to use them. We want to keep using the `BinanceMock` module in the development environment, but for the `test` environment, we would like to set the mocked module generated by the `mox` package:

```
# /config/test.exs
config :naive,
  binance_client: Test.BinanceMock
```

17.4.2 Mock the `NaiveLeader` module

We can now move back to the `Naive.Trader` module to update all the hardcoded references to the `Naive.Leader` module with a dynamic attribute called `@leader` and add this attribute at the top of the module:

```
# /apps/naive/lib/trader.ex
  ...
  @leader Application.get_env(:naive, :leader)
  ...
  @leader.notify(:trader_state_updated, new_state)
  ...
  @leader.notify(:trader_state_updated, new_state)
  ...
  @leader.notify(:rebuy_triggered, new_state)
  ...
```

As it was in case of the `BinanceMock`(our dummy implementation) module, the `Naive.Leader` module doesn't provide a behaviour - let's fix that by defining the `@callback` attributes at the top of the module:

```
# /apps/naive/lib/leader.ex
  ...
  @type event_type :: atom
  @callback notify(event_type, %Trader.State{}) :: :ok
```

In the above code, we added a single `@callback` attribute that defines the naive leader behaviour. For clarity, we defined a distinct type for the `event_type` arguments.

As we now have a naive leader behaviour defined, we can use it to define a mock using the `Mox.defmock/2` function inside the `test_helper.exs` file of the `naive` application:

```
# /apps/naive/test/test_helper.exs
Mox.defmock(Test.Naive.LeaderMock, for: Naive.Leader)
```

In the above code, we've told the `mox` package to define the `Test.Naive.LeaderMock` module based on the `Naive.Leader` behaviour.

We are moving on to the configuration. As the `Naive.Leader` wasn't part of the configuration, we need to add it to the default config and test config file.

First, let's add the `:leader` key inside the `config :naive` in the default `/config/config.exs` configuration file:

```
# /config/config.exs
...
config :naive,
 binance_client: BinanceMock,
 leader: Naive.Leader,        # <= added
 ...
```

and then we need to apply the same update to the `/config/test.exs` configuration file(it will point to the module generated by the `mox` package - `Test.Naive.LeaderMock`):

```
# /config/test.exs
...
config :naive,
binance_client: Test.BinanceMock,
leader: Test.Naive.LeaderMock     # <= added
...
```

17.4.3 Mock the `Phoenix.PubSub` module

Mocking the `Phoenix.PubSub` dependency inside the `Naive.Trader` module will look very similar to the last two mocked modules.

Inside the `Naive.Trader` module we need to update all the references to the `Phoenix.PubSub` to an `@pubsub_client` attribute with value dependent on the configuration:

```
# /apps/naive/lib/trader.ex
  ...
  @pubsub_client Application.get_env(:core, :pubsub_client)
  ...
```

```
    @pubsub_client.subscribe(
      Core.PubSub,
      "TRADE_EVENTS:#{symbol}"
    )
  ...
    @pubsub_client.broadcast(
      Core.PubSub,
      "ORDERS:#{order.symbol}",
      order
    )
  ...
```

The `Phoenix.PubSub` module doesn't provide a behaviour. As we can't modify its source, we need to create a new module to define the PubSub behaviour. Let's create a new file called `test.ex` inside the `/apps/core/lib/core` directory with the following behaviour definition:

```
# /apps/core/lib/core/test.ex
defmodule Core.Test do

  defmodule PubSub do
    @type t :: atom
    @type topic :: binary
    @type message :: term

    @callback subscribe(t, topic) :: :ok | {:error, term}
    @callback broadcast(t, topic, message) :: :ok | {:error, term}
  end
end
```

As previously, we defined a couple of callbacks and additional types for each of their arguments.

Next, we will use the above behaviour to define a mock using the `Mox.defmock/2` function inside the `test_helper.exs` file of the `naive` application:

```
# /apps/naive/test/test_helper.exs
Mox.defmock(Test.PubSubMock, for: Core.Test.PubSub)
```

In the above code, we've told the `mox` package to define the `Test.PubSubMock` module based on the `Core.Test.PubSub` behaviour.

The final step will be to append the `:core, :pubsub_client` configuration to the `/config/config.exs` file:

```
# /config/config.exs
config :core,                  # <= added
 pubsub_client: Phoenix.PubSub # <= added
```

and the test `/config/test.exs` configuration file:

```
# /config/test.exs
config :core,                  # <= added
pubsub_client: Test.PubSubMock # <= added
```

17.4.4 Mock the `Logger` module

Before we dive in, we should ask ourselves why we would mock the `Logger` module?

We could raise the logging level to `error` and be done with it. Yes, that would fix all debug/info/warning logs, but we would also miss an opportunity to confirm a few details (depends on what's necessary for our use case):

- you can ensure that the log was called when the tested function was run
- you can pattern match the logging level
- you can check the message. This could be useful if you don't want to put sensitive information like banking details etc. inside log messages

Mocking the `Logger` dependency inside the `Naive.Trader` module will follow the same steps as the previous updates.

Inside the `Naive.Trader` module we need to update all the references to the `Logger` to an `@logger` attribute with value dependent on the configuration:

```
# /apps/naive/lib/trader.ex
  ...
  @logger Application.get_env(:core, :logger)
  ...
    @logger.info("Initializing new trader(#{id}) for #{symbol}")
  ...
    @logger.info(
      "The trader(#{id}) is placing a BUY order " <>
        "for #{symbol} @ #{price}, quantity: #{quantity}"
    )
  ...
        @logger.info(
          "The trader(#{id}) is placing a SELL order for " <>
            "#{symbol} @ #{sell_price}, quantity: #{quantity}."
        )
  ...
        @logger.info("Trader's(#{id} #{symbol} buy order got partially filled")
  ...
      @logger.info("Trader(#{id}) finished trade cycle for #{symbol}")
  ...
      @logger.info("Trader's(#{id} #{symbol} SELL order got partially filled")
  ...
```

```
      @logger.info("Rebuy triggered for #{symbol} by the trader(#{id})")
  ...
```

The `Logger` module doesn't provide a behaviour. As we can't modify its source, we need to create a new module to define the Logger behaviour. Let's place it inside the `Core.Test` namespace in the `/apps/core/lib/core/test.ex` file side by side with the PubSub behaviour with the following definition:

```
# /apps/core/lib/core/test.ex
defmodule Core.Test do
  ...
  defmodule Logger do
    @type message :: binary

    @callback info(message) :: :ok
  end
end
```

As previously, we defined a callback and additional type for the `message` argument.

Next, we will use the above behaviour to define a mock using the `Mox.defmock/2` function inside the `test_helper.exs` file of the `naive` application:

```
# /apps/naive/test/test_helper.exs
Mox.defmock(Test.LoggerMock, for: Core.Test.Logger)
```

In the above code, we've told the `mox` package to define the `Test.LoggerMock` module based on the `Core.Test.Logger` behaviour.

The final step will be to append the `:core, :logger` configuration to the `/config/config.exs` file:

```
# /config/config.exs
config :core,
  logger: Logger,                # <= added
  pubsub_client: Phoenix.PubSub
```

and the test `/config/test.exs` configuration file:

```
# /config/test.exs
config :core,
  logger: Test.LoggerMock,        # <= added
  pubsub_client: Test.PubSubMock
```

This finishes the updates to the `Naive.Trader` module. We made all dependencies based on the configuration values. We can now progress to writing the test.

17.5 Implement a test of the `Naive.Trader` module

Finally, we can implement the unit test for the `Naive.Trader` module.

We will create a folder called `naive` inside the `/apps/naive/test` directory and a new file called `trader_test.exs` inside it.

Let's start with an empty skeleton of the test tagged as `unit`:

```
# /apps/naive/test/naive/trader_test.exs
defmodule Naive.TraderTest do
  use ExUnit.Case
  doctest Naive.Trader

  @tag :unit
  test "Placing buy order test" do
  end
end
```

Let's add the `mox` related code above the `@tag :unit` line:

```
# /apps/naive/test/naive/trader_test.exs
  import Mox                    # <= 1

  setup :set_mox_from_context   # <= 2
  setup :verify_on_exit!        # <= 3
```

In the above code, we are:

- importing the `mox` module so we will be able to use functions like `expect/3`
- allowing any process to consume mocks defined by the test. This is crucial as tests are run as separate processes that would normally be the only ones allowed to use mocks that they define. Inside our test, we will start a new `Naive.Trader` process that needs to be able to access mocks defined in the test - hence this update
- telling `mox` to verify that all the mocks defined in the tests have been called from within those tests. Otherwise, it will flag such cases as test errors

Inside our test, we need to define implementation for all the functions that the `Naive.Trader` relies on:

```
# /apps/naive/test/naive/trader_test.exs
  ...
  test "Placing buy order test" do
    Test.PubSubMock
    |> expect(:subscribe, fn (_module, "TRADE_EVENTS:XRPUSDT") -> :ok end) # <= 1
    |> expect(:broadcast, fn (_module, "ORDERS:XRPUSDT", _order) -> :ok end)
```

```
    Test.BinanceMock
    |> expect(:order_limit_buy, fn ("XRPUSDT", "464.360", "0.4307", "GTC")  -> # <= 2
      {:ok, BinanceMock.generate_fake_order(
        "XRPUSDT",
        "464.360",
        "0.4307",
        "BUY"
      )
      |> BinanceMock.convert_order_to_order_response()}
    end)

    test_pid = self() # <= 3
    Test.Naive.LeaderMock
    |> expect(:notify, fn (:trader_state_updated, %Naive.Trader.State{}) ->
      send(test_pid, :ok) # <= 3
      :ok
    end)

    Test.LoggerMock
    |> expect(:info, 2, fn (_message) -> :ok end) # <= 4
    ...
```

It's important to note that we defined the mocked function with expected values in the above code. We expect our test to subscribe to a specific topic and broadcast to the other(#1). We are also expecting that process will place an order at the exact values that we calculated upfront. This way, our mock becomes an integral part of the test, asserting that the correct values will be passed to other parts of the system(dependencies of the `Naive.Trader` module).

Another "trick"(#3) that we can use in our mocks is to leverage the fact that we can send a message to the test process from within the mocked function. We will leverage this idea to know precisely when the trader process finished its work as the `notify/1` is the last function call inside the process' callback(`handle_info/2` inside the `Naive.Trader` module). We will assert that we should receive the message, and the test will be waiting for it before exiting(the default timeout is 100ms) instead of using hardcoded `sleep` to "hack" it to work.

The final part(#4) tells the `mox` package that `Logger.info/1` will be called twice inside the test. The `mox` will verify the number of calls to the mocked function and error if it doesn't much the expected amount.

The second part of the test is preparing the initial state for the `Naive.Trader` process, generating trade event and sending it to the process:

```
# /apps/naive/test/naive/trader_test.exs
  ...
  test "Placing buy order test" do
    ...
    trader_state = dummy_trader_state()
    trade_event = generate_event(1, "0.43183010", "213.10000000")
```

```
    {:ok, trader_pid} = Naive.Trader.start_link(trader_state)
    send(trader_pid, trade_event)
    assert_receive :ok
  end
```

As described above, the `assert_receive/1` function will cause the test to wait for the message for 100ms before quitting.

Here are the helper functions that we used to generate the initial trader state and trade event:

```
# /apps/naive/test/naive/trader_test.exs
  ...
  test "Placing buy order test" do
    ...
  end

  defp dummy_trader_state() do
    %Naive.Trader.State{
      id: 100_000_000,
      symbol: "XRPUSDT",
      budget: "200",
      buy_order: nil,
      sell_order: nil,
      buy_down_interval: Decimal.new("0.0025"),
      profit_interval: Decimal.new("0.001"),
      rebuy_interval: Decimal.new("0.006"),
      rebuy_notified: false,
      tick_size: "0.0001",
      step_size: "0.001"
    }
  end

  defp generate_event(id, price, quantity) do
    %Core.Struct.TradeEvent{
      event_type: "trade",
      event_time: 1_000 + id * 10,
      symbol: "XRPUSDT",
      trade_id: 2_000 + id * 10,
      price: price,
      quantity: quantity,
      buyer_order_id: 3_000 + id * 10,
      seller_order_id: 4_000 + id * 10,
      trade_time: 5_000 + id * 10,
      buyer_market_maker: false
```

```
    }
  end
```

The above code finishes the implementation of the test, but inside it, we used functions from the `BinanceMock` module that are private. We need to update the module by making the `generate_fake_order/4` and `convert_order_to_order_response/1` function public(and moving them up in the module, so they are next to other public functions):

```
# /apps/binance_mock/lib/binance_mock.ex
  ...
  def get_order(symbol, time, order_id) do
    ...
  end

  def generate_fake_order(symbol, quantity, price, side) # <= updated to public
    ...
  end

  def convert_order_to_order_response(%Binance.Order{} = order) do # <= updated to public
    ...
  end
...
```

We updated both of the methods to public and moved them up after the `get_order/3` function.

17.6 Define an alias to run unit tests

Our unit test should be run without running the whole application, so we need to run them with the `--no-start` argument. We should also select unit tests by tag(`--only unit`). Let's create an alias that will hide those details:

```
# /mix.exs
  defp aliases do
    [
      ...
      "test.unit": [
        "test --only unit --no-start"
      ]
    ]
  end
```

We can now run our test using a terminal:

```
MIX_ENV=test mix test.unit
```

We should see the following error:

```
21:22:03.811 [error] GenServer #PID<0.641.0> terminating
** (stop) exited in: GenServer.call(BinanceMock, :generate_id, 5000)
    ** (EXIT) no process: the process is not alive or there's no process currently
       associated with the given name, possibly because its application isn't started
```

One of the `BinanceMock` module's functions is sending a message to generate a unique id to the process that doesn't exist(as we are running our tests without starting the supervision tree[the `--no-start` argument]).

There are two ways to handle this issue:

- inside the `/apps/naive/test/test_helper.exs` file we could ensure that the `BinanceMock` is up and running by adding `Application.ensure_all_started(:binance_mock)` function call - this is a hack
- we could refactor the `BinanceMock.generate_fake_order/4` to accept `order_id` as an argument instead of sending the message internally - this should be a much better solution. Let's give it a shot.

First, let's update the `BinanceMock` module:

```
# /apps/binance_mock/lib/binance_mock.ex
  def generate_fake_order(order_id, symbol, quantity, price, side) # <= order_id added
    ...
    # remove the call to GenServer from the body
    ...
  end
  ...
  defp order_limit(symbol, quantity, price, side) do
    ...
      generate_fake_order(
        GenServer.call(__MODULE__, :generate_id), # <= order_id generation added
        symbol,
        quantity,
        price,
        side
      )
```

Now we need to update our test to pass some dummy order id from the mocked function:

```
# /apps/naive/test/naive/trader_test.exs
  ...
  test "Placing buy order test" do
    ...
      {:ok, BinanceMock.generate_fake_order(
        "12345",                        # <= order_id added
        "XRPUSDT",
        "464.360",
        "0.4307",
        "BUY"
      )
   ...
  end
```

We can now rerun our test:

```
MIX_ENV=test mix test.unit
...
Finished in 0.1 seconds (0.00s async, 0.1s sync)
2 tests, 0 failures, 1 excluded
```

Congrats! We just successfully tested placing an order without any dependencies. To avoid explicitly passing the `MIX_ENV=test` environment variable, we will add the prefered environment for our alias inside the `mix.exs` file:

```
# /mix.exs
  def project do
    [
      ...
      preferred_cli_env: [
        "test.unit": :test
      ]
  end
```

Now we can run our tests by:

```
mix test.unit
...
Finished in 0.06 seconds (0.00s async, 0.06s sync)
2 tests, 0 failures, 1 excluded
```

That's all for this chapter - to sum up, the main advantages from the `mox` based tests:

- we were able to test a standalone process/module ignoring all of its dependencies
- we were able to confirm that dependent functions were called and expected values were passed to them
- we were able to create a feedback loop where mock was sending a message back to the test, because of which, we didn't need to use `sleep`, and that resulted in a massive speed gains

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

Source code for this chapter can be found at Github