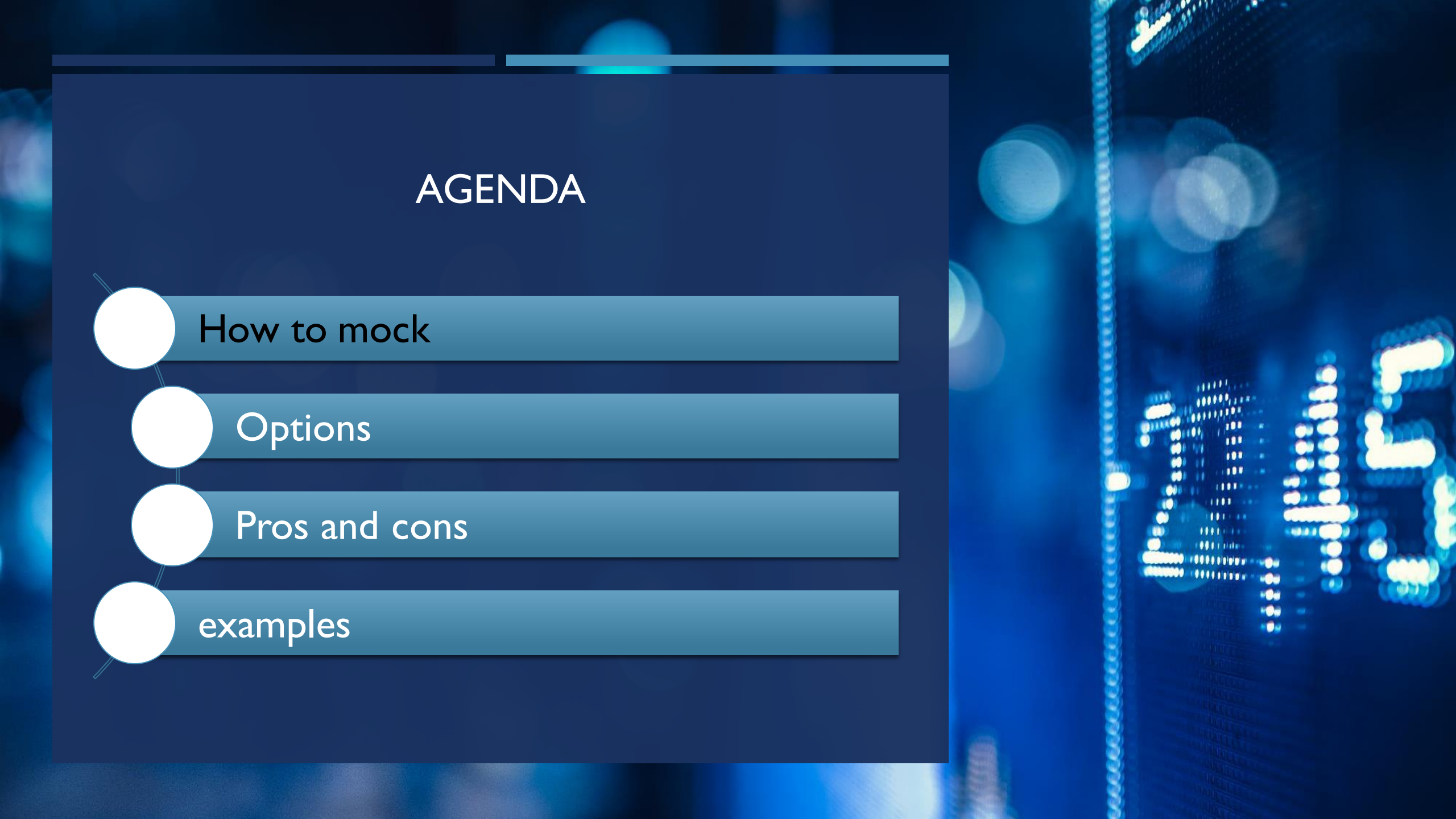

ELIXIR MOCKING

AGENDA

- How to mock
- Options
- Pros and cons
- examples



OPTIONS

- NO TOOL
- MOX
- MOCK
- MECKS UNIT

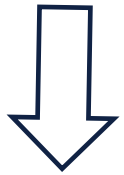
NO TOOL

USING BEHAVIOUR

- What is a **behaviour (~interface)**
- Example:
  - **@Callback** (specs/function signature)

```elixir
defmodule Calculator do
  @callback add(integer(), integer()) :: integer()
  @callback mult(integer(), integer()) :: integer()
end
```

  - **@Behaviour** (This module **must** implement the callbacks/functions from above!)
  - You can have many implementations:



```elixir
defmodule Calculators.NormalCalculator do
  @behaviour Calculator

  def add(x, y), do: x + y
  def mult(x, y), do: x * y
end
```

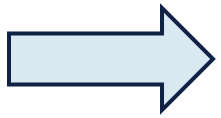
```elixir
defmodule Calculators.KostasCalculator do
  @behaviour Calculator

  def add(1, 1), do: 3
  def add(1, 6), do: 98

  def mult(3, 3), do: 93
  def mult(4, 3), do: 77
end
```

USING BEHAVIOURS

- WHAT WE HAVE SO FAR?

USING BEHAVIOURS

- SOME **CALLBACKS** (*JUST FUNCTION SPECS*)
- TWO IMPLEMENTATIONS (MODULES THAT IMPLEMENT THE CALLBACKS)



```elixir
defmodule Calculator do
  @callback add(integer(), integer()) :: integer()
  @callback mult(integer(), integer()) :: integer()
end
```

```elixir
defmodule Calculators.NormalCalculator do
  @behaviour Calculator

  def add(x, y), do: x + y
  def mult(x, y), do: x * y
end
```

```elixir
defmodule Calculators.KostasCalculator do
  @behaviour Calculator

  def add(1, 1), do: 3
  def add(1, 6), do: 98

  def mult(3, 3), do: 93
  def mult(4, 3), do: 77
end
```

USING BEHAVIOURS

- THE GOAL: **TO CHANGE IMPLEMENTATION PER ENVIROMENT**

USING BEHAVIOURS

- WHY TO DO THAT?

USING BEHAVIOURS

- A STORED PROCEDURE NOT READY YET
- AN ENDPOINT FROM A SPANISH TEAM IS NOT DONE

- **FREE TO CONTINUE THE DEVELOPMENT**
- **WE ARE NOT DEPENDENT ON A SPECIFIC IMPLEMENTATION**
- **WE CAN CONTINUE WRITING UNIT TESTS**

USING BEHAVIOURS

- HOW WE CAN CHANGE IMPLEMENTATION PER ENVIROMENT?

USING BEHAVIOURS

- **CONFIG FILES**! (CONFIG.EXS, DEV.EXS, TEST.EXS ...)
- **Application.get_env/3**

USING BEHAVIOURS

- **In dev.exs**

```
use Mix.Config

app = Mix.Project.config()[:app]

config app, :calculator, Calculators.NormalCalculator
```

VALUE IN DEV

- **In test.exs**

```
use Mix.Config

app = Mix.Project.config()[:app]

config app, :calculator, Calculators.KostasCalculator
```

APP NAME KEY

```
defmodule Math do
  @calculator Application.get_env(:some_elixir_play, :calculator)

  def perform_addition_and_mulitplication(a, b) do
    @calculator.add(a, b) + @calculator.mult(a, b)
  end
end
```

USING BEHAVIOURS – CALCULATOR IN DEV

- We alternate implementations per environment!
  - **Implementation:**

```elixir
defmodule Calculators.NormalCalculator do
  @behaviour Calculator

  def add(x, y), do: x + y
  def mult(x, y), do: x * y
end
```

  - **Code:**

```elixir
defmodule Math do
  @calculator Application.get_env(:some_elixir_play, :calculator)

  def perform_addition_and_mulitplication(a, b) do
    @calculator.add(a, b) + @calculator.mult(a, b)
  end
end
```

@calculator === Calculators.NormalCalculator

  - **Call code from DEV:**

```
λ iex -S mix
Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
iex> Math.perform_addition_and_mulitplication(1, 1)
3
```

USING BEHAVIOURS – CALCULATOR IN TEST

- We alternate implementations per environment!
  - **Implementation:**

```elixir
defmodule Calculators.KostasCalculator do
  @behaviour Calculator

  def add(1, 1), do: 3
  def add(1, 6), do: 98

  def mult(1, 1), do: 93
  def mult(4, 3), do: 77
end
```

  - **Code:**

```elixir
defmodule Math do
  @calculator Application.get_env(:some_elixir_play, :calculator)

  def perform_addition_and_mulitplication(a, b) do
    @calculator.add(a, b) + @calculator.mult(a, b)
  end
end
```

@calculator === Calculators.KostasCalculator

  - **Call code from TEST**

```
λ mix test test\unit\some_elixir_play_test.exs
Compiling 6 files (.ex)
Generated some_elixir_play app
.

Finished in 0.03 seconds
1 test, 0 failures

Randomized with seed 806000
```

```elixir
defmodule Unit.SomeElixirPlayTest do
  use ExUnit.Case

  test "a test" do
    assert Math.perform_addition_and_mulitplication(1, 1) == 96
  end

end
```

USING BEHAVIOURS

- CONS
  - MAINTAIN CONFIG FILES
  - EXTRA MODULES IMPLEMENTATIONS

- PROS
  - CONTINUE DEVELOPMENT
  - NOT DEPENDENT IN THE ACTUAL IMPLEMENTATION
  - DON'T HAVE TO MODIFY CODE ACROSS ALL PROJECT (I.E. CHANGE OF HTTP CLIENT)
  - RUN TESTS IN AN ASYNC FASION

MOX

MOX – AN ALMOST "NO TOOL"

- MOX IS QUITE SIMILAR TO "NO TOOL" APPROACH
- LET'S SEE THE SIMILARITIES AND DIFFERENCIES

MOX – SIMILARITIES TO "NO TOOL"

- STILL NEED TO DEFINE THE CALLBACKS

```elixir
defmodule Calculator do
  @callback add(integer(), integer()) :: integer()
  @callback mult(integer(), integer()) :: integer()
end
```

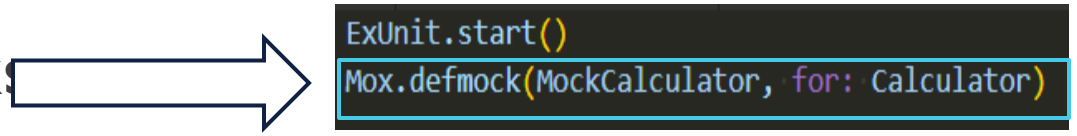
- STILL NEED TO MODIFY THE CONFIG FILES (DEV.EXS, TEST.EXS)

```elixir
use Mix.Config

app = Mix.Project.config()[:app]

config app, :calculator, Calculators.NormalCalculator
```

```elixir
use Mix.Config

app = Mix.Project.config()[:app]

config app, :calculator, MockCalculator
```

MOX – DIFFERENCES TO "NO TOOL" APPROACH

- SOME WRITING IN **TEST_HELPER.EXS**
- NO NEED TO WRITE A WHOLE MODULE



MOX – IN ACTION

- TEST USING MOX:

```elixir
defmodule Unit.SomeElixirPlayTest do
  use ExUnit.Case
  import Mox

  test "mox" do

    expect(MockCalculator, :add, fn x, y -> 30 end)
    expect(MockCalculator, :mult, fn x, y -> 20 end)

    assert Math.perform_addition_and_mulitplication(2, 3) == 50
  end
end
```

USING BEHAVIOURS

- CONS
  - SAME CONS AS "NO TOOL" APPROACH
- PROS
  - SAME PROS AS "NO TOOL" APPROACH
  - NO NEED TO DEFINE THE WHOLE MODULE

MOCK AND MECKS UNIT

MOCK

- NO BEHAVIOUR USE
- TESTS ARE EXECUTED SYNCHRONOULSY
- EXAMPLE:

```
defmodule MyTest do
  use ExUnit.Case, async: false

  import Mock

  test "multiple mocks" do
    with_mocks([
      {Map,
       [],
       [get: fn(%{}, "http://example.com") -> "<html></html>" end]},
      {String,
       [],
       [reverse: fn(x) -> 2*x end,
        length: fn(_x) -> :ok end]}
    ]) do
      assert Map.get(%{}, "http://example.com") == "<html></html>"
      assert String.reverse(3) == 6
      assert String.length(3) == :ok
    end
  end
end
```

MECKS UNIT

- BOTH ARE USING MECKS ERLANG LIB
- YOU CAN RUN THEM ASYNCRHONOUSLY

MOCK – MECK UNIT

- CONS
  - NO USE OF ABSTRACTION (E.G. BEHAVIOUR)
  - ONE CHANGE AFFECTS ALL THE CODE BASE (FOR EXAMLE, HTTP CLIENT CHANGE)

- PROS
  - NO NEED TO MAINTAIN CONFIGS
  - EASIER TO IMPLEMENT A QUICK AND DIRTY TEST? (PROBABLY THIS IS A CON)

RESOURCES

- mocks and explicit contracts
- mecks unit
- elixir school
- mock
- mecks, elixir forum
- unit and integration testing
- london school TDD VS Detroit school TDD
- why mox

QUESTIONS?

THANK YOU!