BRINGING BEAM INTO WORLD OF MULTIMEDIA

MEMBRANE FRAMEWORK

@ElixirMembrane @mspanc @swmansion #CodeBEAMSF

CHAPTER 1

WHO IS THIS GUY ON THE STAGE?

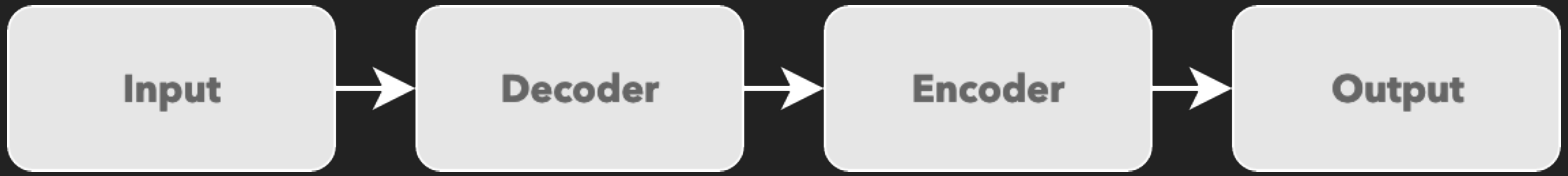
WHO IS THIS GUY ON THE STAGE?

- Marcin Lewandowski (@mspanc)
- Founded first IT company in the high school and keep founding stuff
- Used to work in the media industry for a few years
- Came back to the IT with tons of ideas on how to make state of the art software for the media
- Founded RadioKit - a startup making software for radio stations in Elixir
- Merged RadioKit with Software Mansion - a consultancy based in Poland (@swmansion)
- Extracted Membrane from RadioKit, backed it up by a full-time team in the Software Mansion and released it as an open source (@ElixirMembrane)

CHAPTER 2

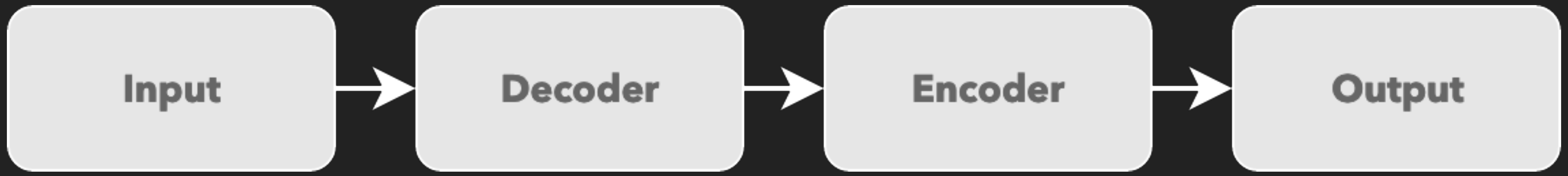
WHAT IS THE PROBLEM?

A HELLO WORLD OF MULTIMEDIA PROCESSING



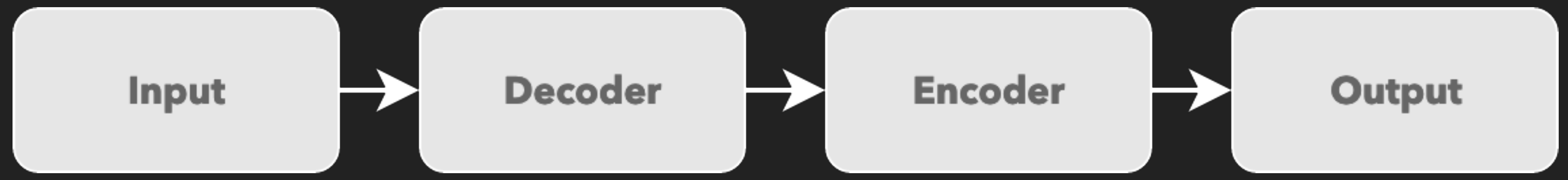
A HELLO WORLD OF MULTIMEDIA PROCESSING

- Take some audio (e.g. MP3) from some file or stream
- Decode it
- Encode it with a different bitrate or quality into the same or a different format
- Store it somewhere



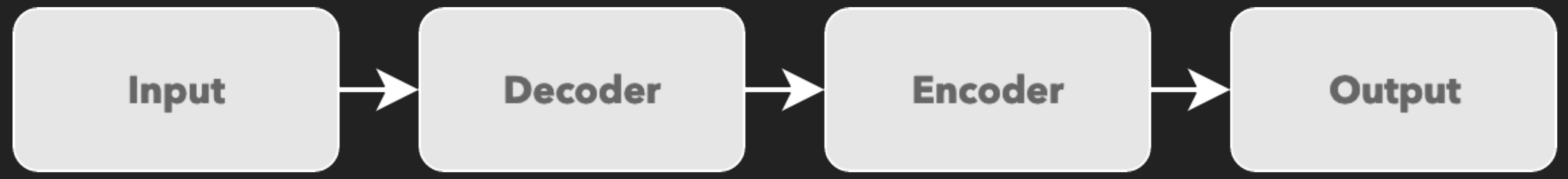
WHAT CAN GO WRONG?

- Any element can have lower throughput than the preceding part of the pipeline
- Decoder mail fail (or partially fail) due to the malformed input
- Input stream may be enriched by the metadata or other sorts of headers that decoder will not handle (e.g. ID3v1, v2.x, Xing)
- Input stream can be possibly interleaved by some additional metadata (e.g. internet radio)



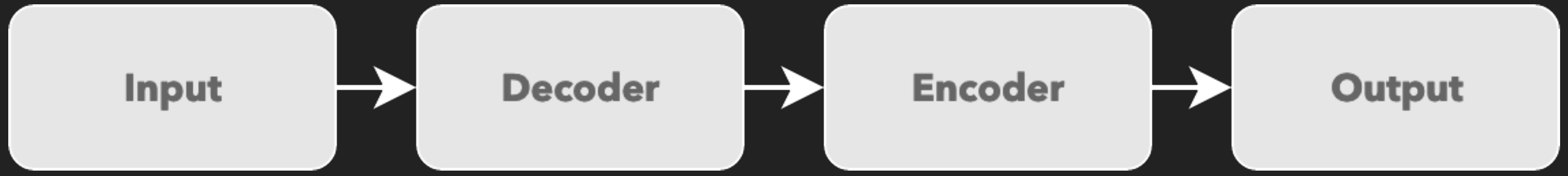
WHAT CAN GO WRONG?

- Decoder can output different raw audio formats (e.g. S16LE, 44100 Hz, stereo)
- The encoder for given format may handle just a subset of raw audio formats
- Format conversion and resampling may be lossy



WHAT CAN GO WRONG?

- Underlying native libraries have extremely diverse APIs and assumptions
- Virtually every Elixir library being used have tons of native dependencies, recursively
- There’s no uniform way to handle dependencies in the build process even within one platform (e.g. not everything ships with pkg-config on Linux)
- There are many platforms and many compilers
- Forcing `mix compile` to work out of the box in such project is a challenge by itself



A REAL LIFE EXAMPLE

- An application that allows to make a TV-like stream from an event like this
- Multiple audio/video inputs mixed in real time, while audio and video streams can be mixed independently
- Inputs can be delivered over the network or via native system APIs for capturing audio/video
- Decode and encode on the GPU if possible
- Distributed over HLS/RTSP/WebRTC

CHALLENGES (AKA WHAT CAN GO WRONG?)

- Audio/video synchronization
- Inputs can fail randomly but outputs have to keep going and vice versa
- Different buffer types (e.g. pointers to the GPU memory)
- Reliability (NIFs vs C nodes vs Ports)
- Clocks synchronization & time skew
- Dynamic reconfiguration of the pipeline
- Bi-directional communication (QoS, FEC etc.)
- Different formats supported by the different clients
- Scaling

CHAPTER 3

HOW TO DO MULTIMEDIA STREAMING AND STAY SANE?

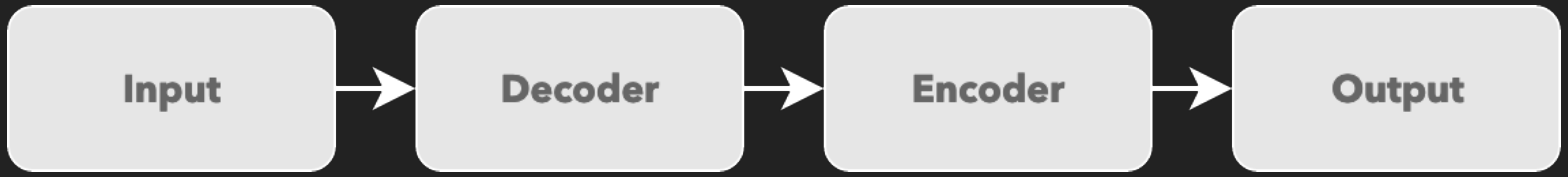
ABSTRACTION LAYER

inspired by GStreamer

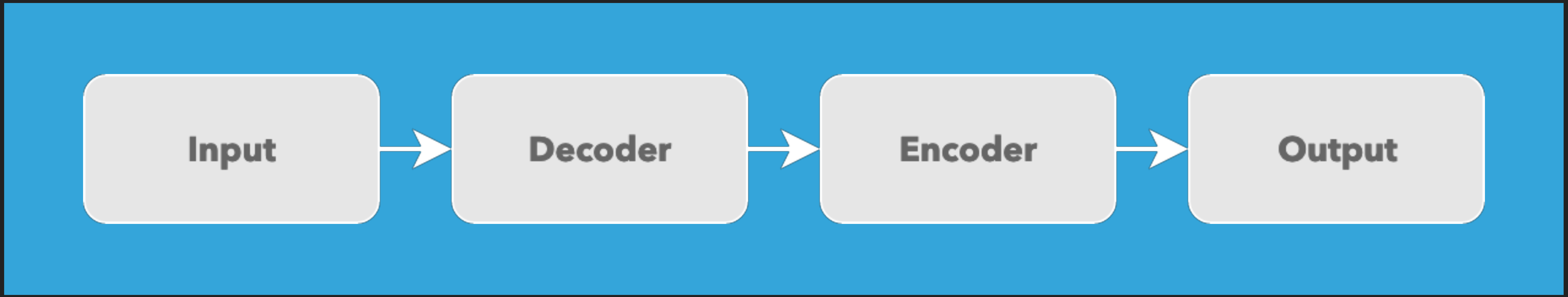
(but trying to avoid its limitations and design flaws)

(and having a bit different objectives)

ABSTRACTION LAYER



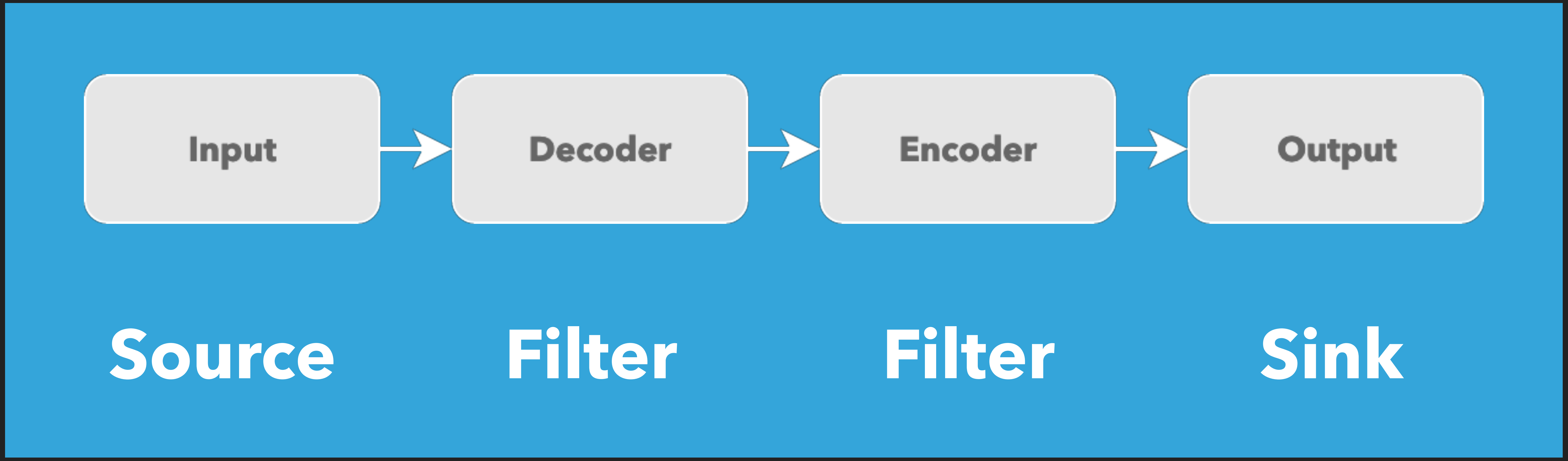
ABSTRACTION LAYER



Pipeline

(contains Elements)

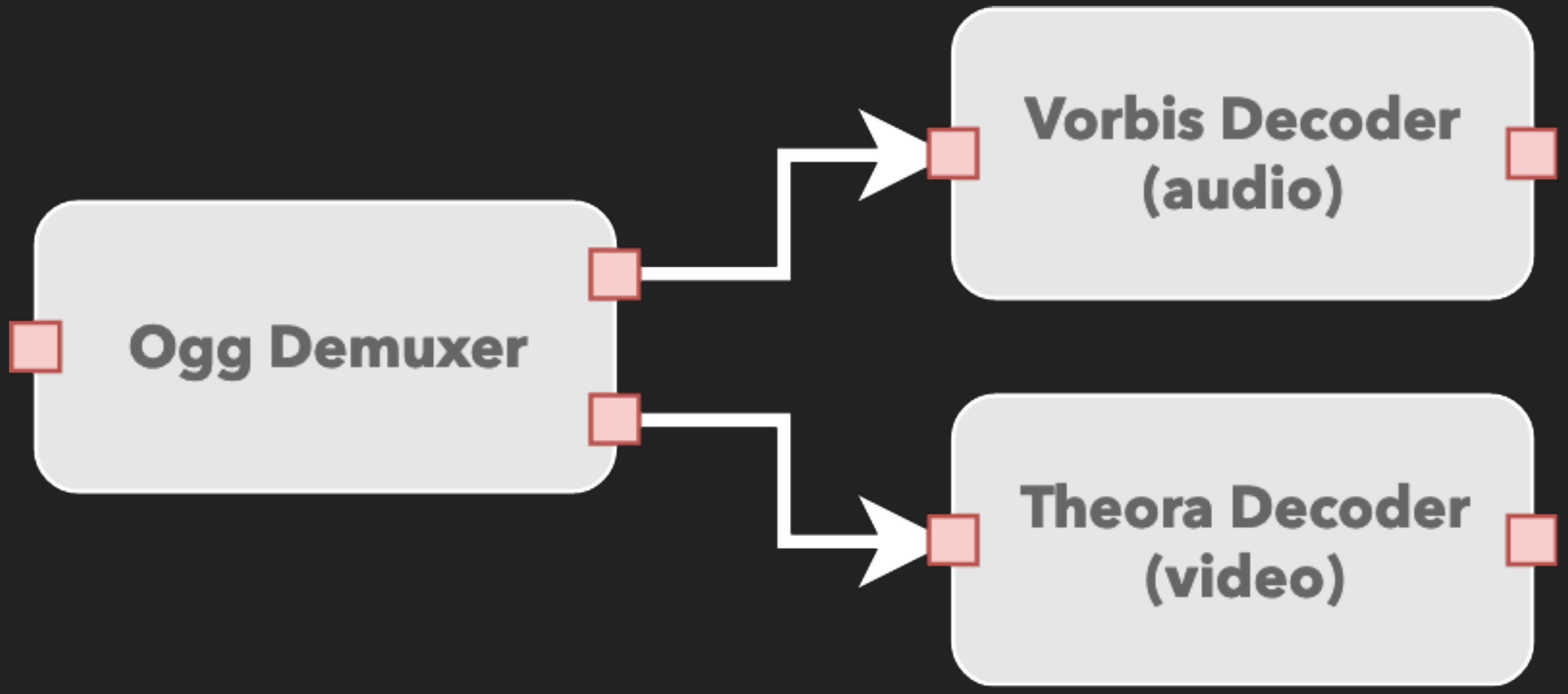
ABSTRACTION LAYER



Elements

(Sources, Filters or Sinks)

ABSTRACTION LAYER

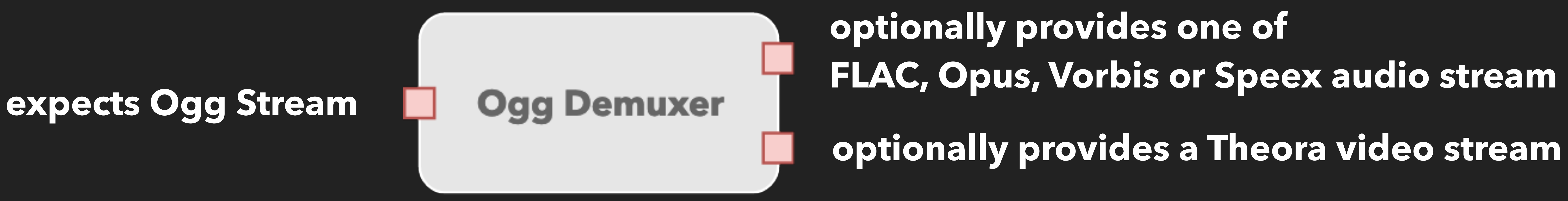


Pads

(static or dynamic)
(pull or push)

defined per Element type

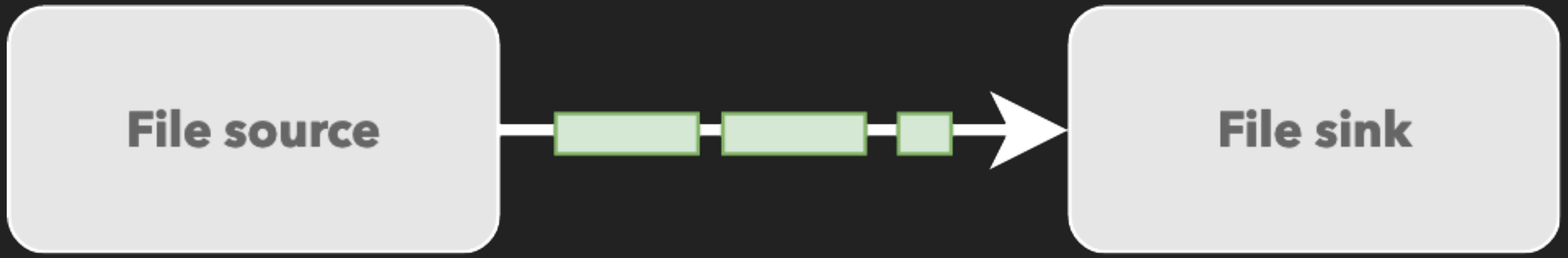
ABSTRACTION LAYER


expects Ogg Stream
Ogg Demuxer
optionally provides one of FLAC, Opus, Vorbis or Speex audio stream
optionally provides a Theora video stream

Caps

defined per Pad

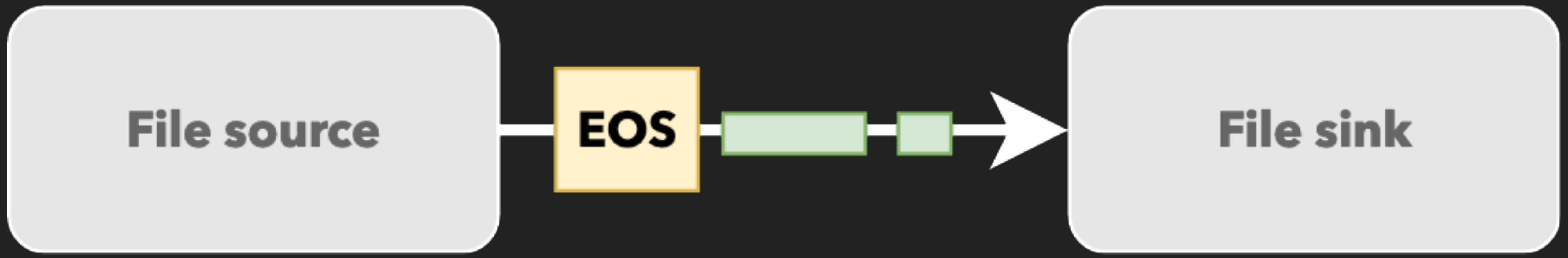
ABSTRACTION LAYER



Buffers

data chunks that flow between linked Pads

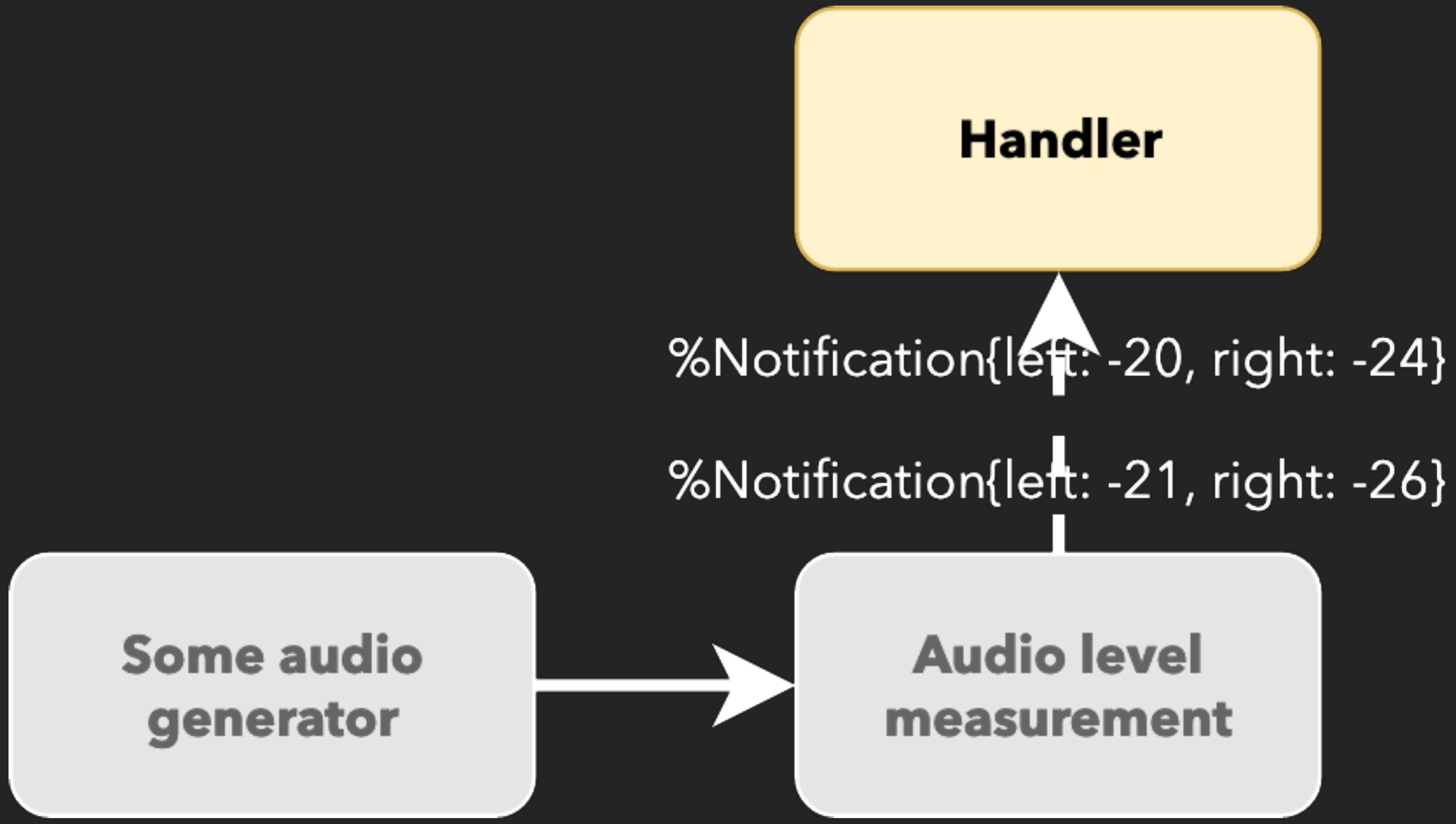
ABSTRACTION LAYER



Events

signals that flow aligned to Buffers

ABSTRACTION LAYER


Handler
%Notification{left: -20, right: -24}
%Notification{left: -21, right: -26}
Some audio generator
Audio level measurement

Notifications

signals that are not aligned to Buffers

MEMBRANE CORE

MEMBRANE CORE (SOME ARE STILL WORK IN PROGRESS)

- Handles lifecycle of Elements and Pipelines
- Builds the actual process and supervision trees
- Provides error handling mechanisms
- Provides ability to link elements via their pads and
- Implements backpressure
- Implements A/V sync
- Implements clock sync

MEMBRANE CORE (SOME ARE STILL WORK IN PROGRESS)

- Provides logging
- Provides advanced inspection features (such as visualizing the pipeline in the real time)
- Handles different buffer types (or memory types)
- Provides set of internal APIs for extending framework by creating elements, caps etc.
- Many many more…

SUPPLEMENTARY LIBRARIES

BUNDLEX - DEPENDENCY MANAGER FOR THE NATIVE CODE

- Resolves C dependencies
- Resolves linker/compiler flags
- Automatically finds Erlang’s C headers
- Handles some of the multi-platform issues

https://bit.ly/membrane-bundlex-1-cb

UNIFEX - ABSTRACTION LAYER OVER C CODE

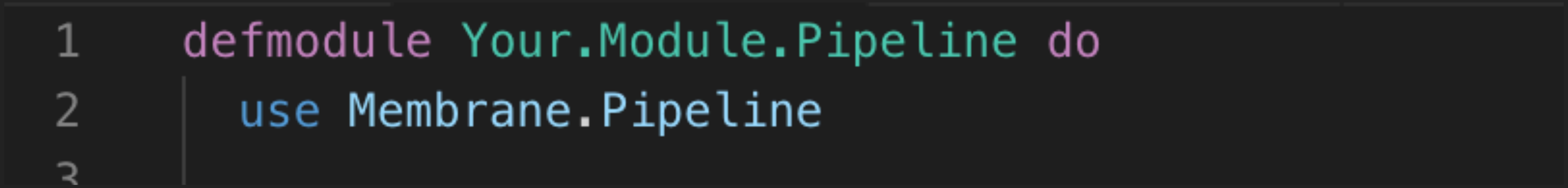
- Makes boilerplate unnecessary
- Forces to make clear definitions of the C<->Elixir interface
- Generates a lot of useful helper functions and eases state handling in the C code
- In the future it will allow to write C code once and ideally run it as NIFs, C nodes or Ports without making any changes

http://bit.ly/membrane-unifex-1-cb

CHAPTER 4

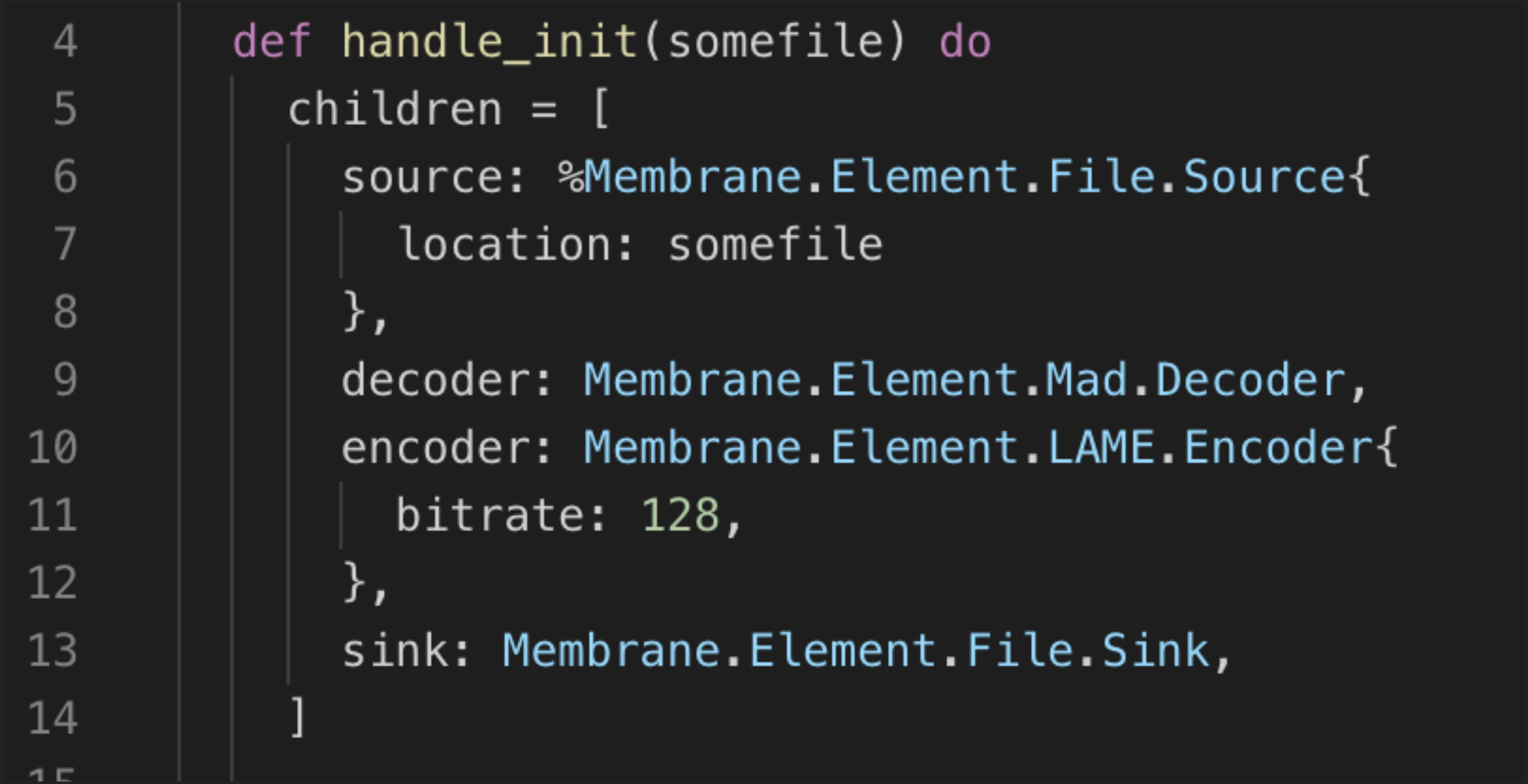
THE CODE

PIPELINE



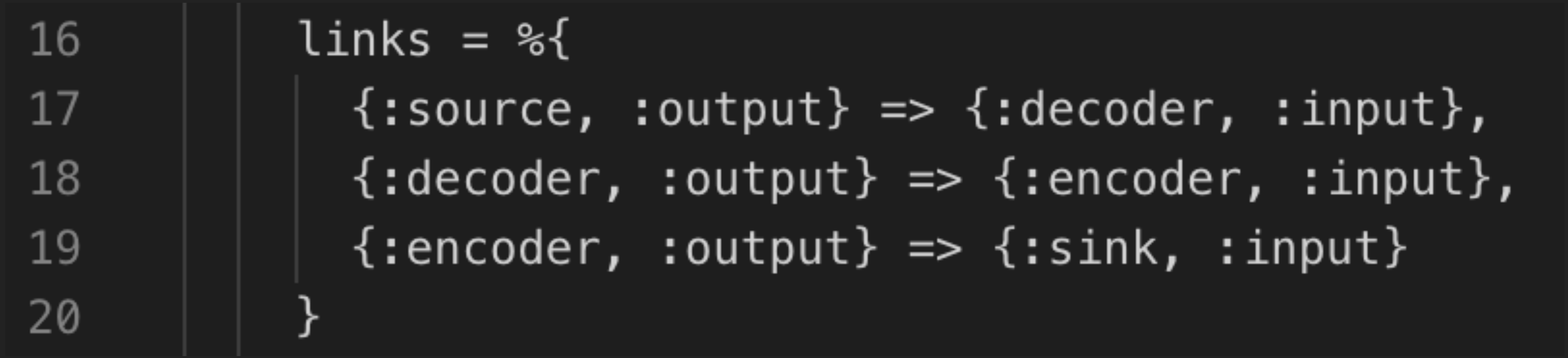
```elixir
defmodule Your.Module.Pipeline do
  use Membrane.Pipeline

```

PIPELINE



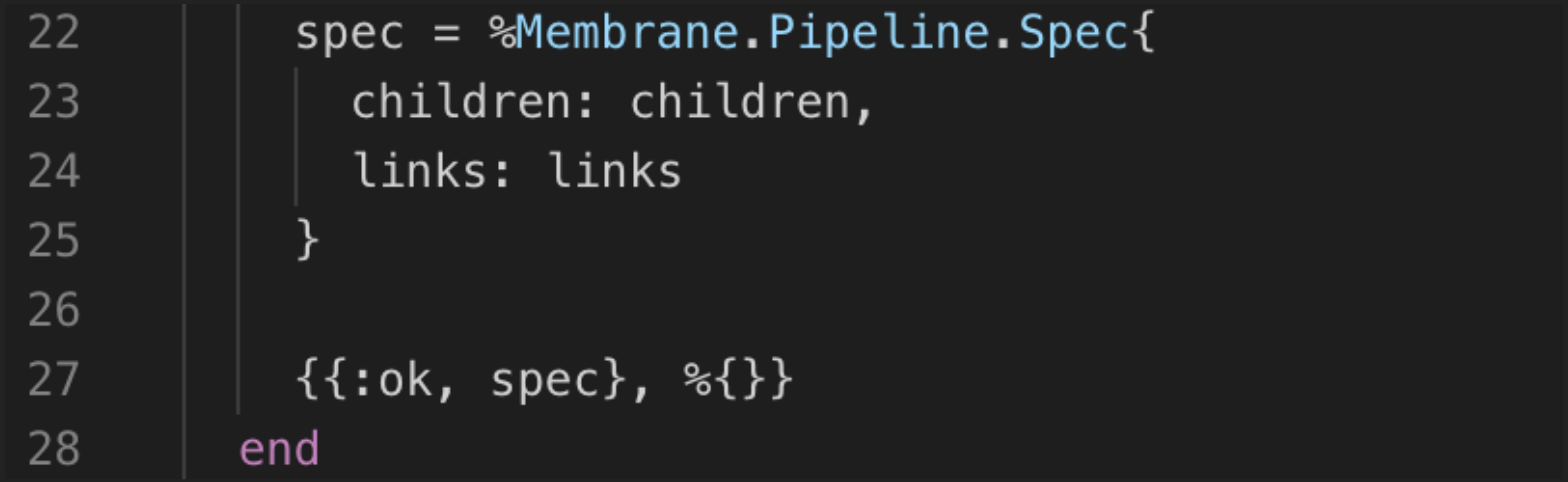
```elixir
  def handle_init(somefile) do
    children = [
      source: %Membrane.Element.File.Source{
        location: somefile
      },
      decoder: Membrane.Element.Mad.Decoder,
      encoder: Membrane.Element.LAME.Encoder{
        bitrate: 128,
      },
      sink: Membrane.Element.File.Sink,
    ]
```

PIPELINE



```elixir
links = %{
  {:source, :output} => {:decoder, :input},
  {:decoder, :output} => {:encoder, :input},
  {:encoder, :output} => {:sink, :input}
}
```

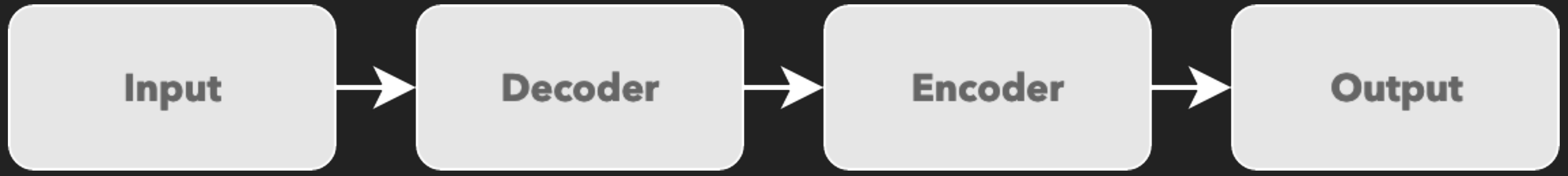
PIPELINE



```
    spec = %Membrane.Pipeline.Spec{
      children: children,
      links: links
    }

    {{:ok, spec}, %{}}
  end
```

PIPELINE

```elixir
defmodule Your.Module.Pipeline do
  use Membrane.Pipeline

  def handle_init(somefile) do
    children = [
      source: %Membrane.Element.File.Source{
        location: somefile
      },
      decoder: Membrane.Element.Mad.Decoder,
      encoder: Membrane.Element.LAME.Encoder{
        bitrate: 128,
      },
      sink: Membrane.Element.File.Sink,
    ]

    links = %{
      {:source, :output} => {:decoder, :input},
      {:decoder, :output} => {:encoder, :input},
      {:encoder, :output} => {:sink, :input}
    }

    spec = %Membrane.Pipeline.Spec{
      children: children,
      links: links
    }

    {{:ok, spec}, %{}}
  end
end
```



PIPELINE

```elixir
alias Membrane.Pipeline
{:ok, pid} = Pipeline.start_link(Your.Module.Pipeline, "/path/to/mp3", [])
Pipeline.play(pid)
```

CHAPTER 5

WHERE TO FIND MORE INFORMATION?

MORE INFORMATION & THANK YOU!

- Website: http://www.membraneframework.org
- Guide: http://www.membraneframework.org/guide
- GitHub: http://github.com/membraneframework
- Discord: https://discord.gg/nwnfVSY
- e-mail: info@membraneframework.org
- Twitter: @ElixirMembrane