DISTRIBUTED AUDIO
USING BEAM, GLEAM, AND THE WEB AUDIO API
WHOAMI
WHOAMI

tag hayleigh

profile frontend elm developer
WHOAMI

Hayleigh

frontend elm developer

phd student
WHOAMI

Hayleigh
Frontend Elm developer
PhD student
Gleam community person
DISTRIBUTED AUDIO
WHAT ARE WE MAKING?
DISTRIBUTED AUDIO

WHAT ARE WE MAKING?

- collaborative step sequencer
- grid of notes/steps
- some sound controls
- all clients in sync
OVERVIEW

- why not x?
- why gleam?
- making sounds
- rendering web apps
- serving static files
- client <-> server communication
OVERVIEW

- why not x?
- why gleam?
- making sounds
- rendering web apps
- serving static files
- client <-> server communication
WHY NOT JAVASCRIPT?

- mutable
- dynamically typed
- error prone
- can’t decide whether or not to use semi colons
WHY NOT ELM?

• restrictive FFI
• what to choose for the backend?
• unfamiliar syntax (not for me, but)
WHY NOT ELIXIR?

- still needs a lot of js for the audio
- i’m a type nerd
- i don’t know elixir...
OVERVIEW

- why not x?
- why gleam?
- making sounds
- rendering web apps
- serving static files
- client <-> server communication
WHY GLEAM?

- same language for the front end and back end
- sharable types
- functional but familiar
- amazing interoperability
- leverage existing libraries
- otp makes multiplayer easy
- #1 bdfl
  - consortium? how about one dude. (idk maybe bad joke)
OVERVIEW

- why not x?
- why gleam?
- making sounds
- rendering web apps
- serving static files
- client <-> server communication
A PRIMER ON THE WEB AUDIO API

- low-ish level API for making sounds
- audio nodes are connected in a graph
- signal processing is handled by native code
const audioContext = new AudioContext()
const osc = audioContext.createOscillator()
const amp = audioContext.createGain()
const dac = audioContext.destination

osc.frequency.value = 880
osc.type = 'square'
amp.gain.value = 0.5

osc.connect(amp)
amp.connect(dac)
osc.start()
pub type Node {
  Node(
    t: String,
    params: List(Param),
    connections: List(Node)
  )
}

pub type Param {
  Param(name: String, value: Float)
  Property(name: String, value: Dynamic)
}
osc([ freq(880.0), waveform("square") ], [
    amp([ gain(0.5) ], [
        dac
    ])
])
OVERVIEW

- why not x?
- why gleam?
- making sounds
- rendering web apps
- serving static files
- client <-> server communication
LUSTRE
type Model {
  Model(
    ctx: AudioContext,
    nodes: List(Node),
    rows: List(Row),
    step: Int,
    step_count: Int,
    waveform: Waveform,
    delay_time: DelayTime,
    gain: Float,
  )
}
fn render_step(step, name, active_column) {
    let #(idx, is_active) = step
    let msg = UpdateStep(name, idx, !is_active)
    let bg = case idx == active_column, is_active {
        True, True -> "bg-faff-200 animate-bloop"
        True, False -> "bg-charcoal-200 scale-[0.8]"
        False, True -> "bg-faff-300"
        False, False -> "bg-charcoal-700 scale-[0.8]"
    }

element.div(
    [class("p-2")],
    [button.box(bg <> " hover:bg-faff-100", msg)]
)
```javascript
fn update(model, msg) {
  case msg {
    UpdateStep(name, idx, is_active) -> {
      let rows = list.map(model.rows, fn(row) {
        let steps = case row.name == name {
          True -> map.insert(row.steps, idx, is_active)
          False -> row.steps
        }

        Row(.,row, steps: steps)
      }

      Model(.,model, rows: rows)
    }

    ...
  }
}
```
Hello, FOSDEM
OVERVIEW

• why not x?
• why gleam?
• making sounds
• rendering web apps
• serving static files
• client <-> server communication
glisten provides a supervisor which manages a pool of acceptors. Each acceptor will block on accept until a connection is opened. The acceptor will then spawn a handler process and then block again on accept.
mist is a pure Glem web server. It provides a simple HTTP server that can be configured to support WebSockets and SSL connections.
fn serve(app: App) -> Result(Nil, glisten.StartError) {
    let port = 8080
    let handler = {
        use req <- mist_handler.with_func

        case req.method, request.path_segments(req) {
            Get, [] -> {
                let res = serve_static_asset("index.html")
                mist_handler.Response(res)
            }

            Get, _ -> {
                let res = serve_static_asset(req.path)
                mist_handler.Response(res)
            }
        }
    }

    mist.serve(port, handler)
}
fn serve_static_asset(path: String) -> Response(HttpResponseBody) {
    let path = static <> "/" <> path
    let file =
        path
            |> file.read_bits
            |> result.map(bit_builder.from_bit_string)

    let res = case file {
        Ok(bits) -> Response(200, [], BitBuilderBody(bits))
        Error(_) -> Response(404, [], not_found)
    }

    case list.last(string.split(path, ".")) {
        Ok("html") -> response.set_header(res, "content-type", "text/html")
        Ok("svg") -> response.set_header(res, "content-type", "image/svg+xml")
        Ok("css") -> response.set_header(res, "content-type", "text/css")
        Ok("js") -> response.set_header(res, "content-type", "application/javascript")
        _ -> response.set_header(res, "content-type", "text/plain")
    }
}

OVERVIEW

• why not x?
• why gleam?
• making sounds
• rendering web apps
• serving static files
• client <-> server communication
fn serve(app: App) -> Result(Nil, glisten.StartError) {
    ...
    case req.method, request.path_segments(req) {
        Get, ["ws"] -> upgrade_websocket(app)
        ...
    }
}

mist.serve(port, handler)
fn upgrade_websocket(app) {
    let handler =
        WebSocketHandler(
            on_init: Some(on_ws_open(_, app)),
            on_close: Some(on_ws_close(_, app)),
            handler: fn(message, client) {
                case message {
                    TextMessage(json) ->
                        Ok(on_ws_message(client, json, app))
                } ->
                    Error(Nil)
            },
        )

    mist_handler.Upgrade(handler)
}
lustre_websocket is a package that makes it trivial to set up websockets on the client. We just need to call `ws.init` and let it handle everything for us.
fn init() {
    #(Model(..), ws.init("/ws", WebSocket))
}

fn update(model, msg) {
    case msg {
        WebSocket(OnOpen(conn)) ->
            #(Model(..).model, ws: Some(conn)), cmd.none()
        WebSocket(OnClose(_)) ->
            #(Model(..).model, ws: Some(conn)), cmd.none()
        WebSocket(OnMessage(msg)) -> {
            // Handle messages from the backend here
            ...
        }
        ...
    }
}
lustre_websocket is a package that makes it trivial to set up websockets on the client. We just need to call ws.init and let it handle everything for us.
```haskell
pub type ToBackend {
  Play
  Stop
  UpdateDelayTime(DelayTime)
  UpdateStep(#(String, Int, Bool))
  UpdateWaveform(Waveform)
}
```
OnMessage(_, _, UpdateWaveform(waveform)) -> {
  let shared = shared.State(state.shared, waveform: waveform)
  broadcast(state.clients, SetWaveform(waveform))

  State(state, shared: shared)
}

OnMessage(_, _, UpdateStep(#(name, idx, is_active))) -> {
  let rows = {
    use row <- list.map(state.shared.rows)
    let steps = case row.name == name {
      True -> map.insert(row.steps, idx, is_active)
      False -> row.steps
    }

    Row(row, steps: steps)
  }

  let shared = shared.State(state.shared, rows: rows)

  broadcast(state.clients, SetRows(rows))
  State(state, shared: shared)
}

...
pub type ToFrontend {
  SetDelayTime(DelayTime)
  SetGain(Float)
  SetRows(List(Row))
  SetState(State)
  SetStep(Int)
  SetStepCount(Int)
  SetWaveform(Waveform)
}

fn on_message(state, message) {
    let shared = state.shared

    case json.decode(message, to_frontend.decoder) {
        Ok(SetState(shared)) ->
            State(..state, shared: shared)

        Ok(SetRows(rows)) -> {
            let shared = shared.State(..shared, rows: rows)
            State(..state, shared: shared)
        }

        Ok(SetStep(step)) -> {
            let shared = shared.State(..shared, step: step)
            State(..state, shared: shared)
        }

        ...

        Error(_) -> state
    }
}
DEMO
RECAP

- fullstack gleam app
- otp server on the backend
- react app on the frontend
- liveview style communication
RECAP

- fullstack gleam app
- otp server on the backend
- react app on the frontend
- liveview style communication
$ cloc src
  15 text files.
  15 unique files.
  0 files ignored.

github.com/AlDanial/cloc v 1.90  T=0.02 s (920.4 files/s, 106209.0 lines/s)

<table>
<thead>
<tr>
<th>Language</th>
<th>files</th>
<th>blank</th>
<th>comment</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gleam</td>
<td>13</td>
<td>265</td>
<td>134</td>
<td>1219</td>
</tr>
<tr>
<td>JavaScript</td>
<td>1</td>
<td>17</td>
<td>9</td>
<td>84</td>
</tr>
<tr>
<td>CSS</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>SUM:</td>
<td>15</td>
<td>282</td>
<td>143</td>
<td>1306</td>
</tr>
</tbody>
</table>
THANKS FOR LISTENING!