Introduction to Elixir and Phoenix

Alex Wynter
awochna@email.arizona.edu
UA IT Summit, Oct 18th 2016

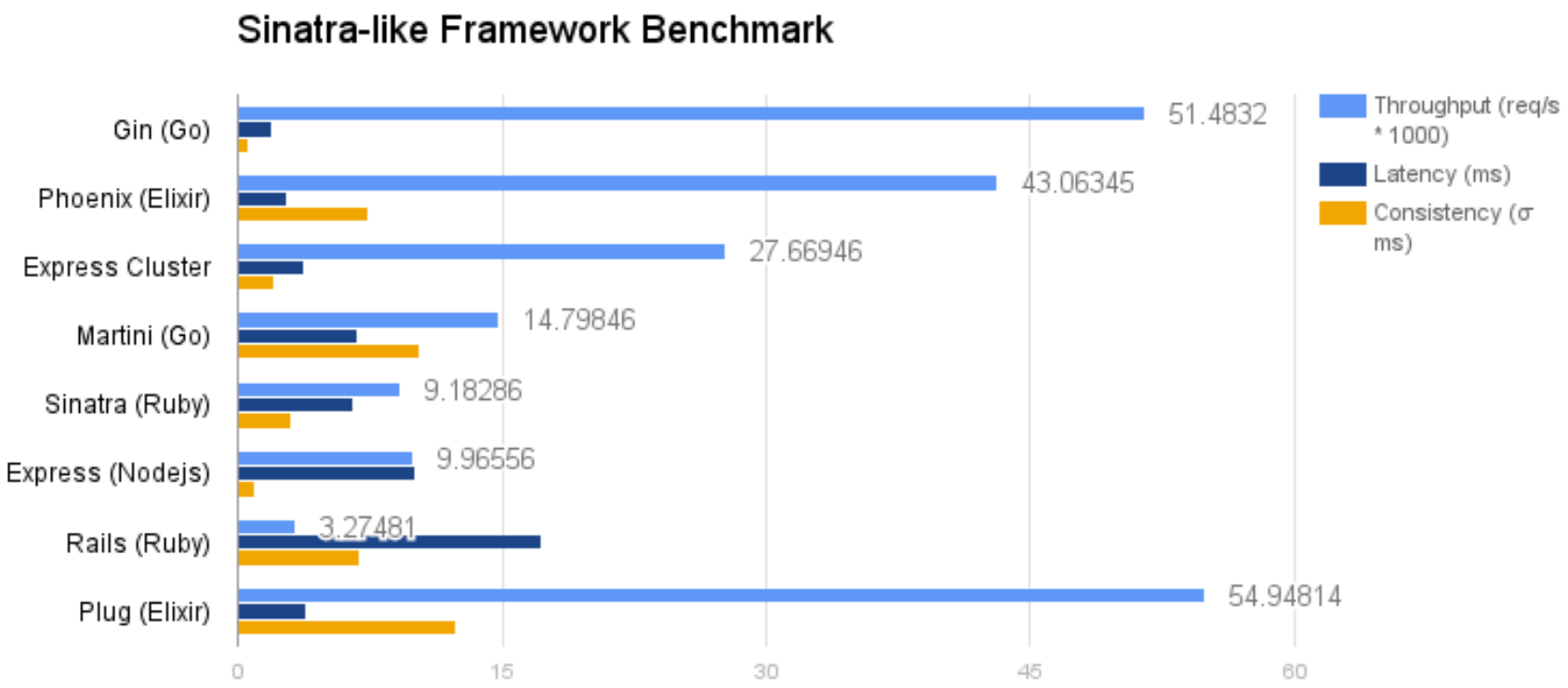
demo.awochna.com

Why Elixir and Phoenix?

- Programming as data processing
- Multi-process concurrency
- Resilient, fault-tolerant systems using supervisors
- Fast; compiles to Erlang bytecode.
- Readable syntax with metaprogramming
- Simple, easy to test layers
- Service providers that automatically restart
- Multiple calls to multiple data sources
- Really good web socket support
- Lots of connections (up to 2 million!)

Why NOT Elixir and Phoenix?

- Functional programming feels different
- There are no objects passing messages back and forth!
- You have to learn a new programming language and **another** framework
- It’s easier to reason about than OO
- They’re conceptually replaced by processes.
- Sorry!



Benchmark from from https://github.com/mroth/phoenix-showdown, chart is own creation.

Elixir: A Programming Language

Elixir: History

Created by José Valim in 2012

Created to solve performance on multi-core systems

Built on the Erlang Virtual Machine (BEAM)

What’s an Erlang?

Created in 1986 by Joe Armstrong, Robert Virding and Mike Williams.

Designed for software telecom systems.

Extremely fault-tolerant and highly available.

Powers GPRS, 3G, and LTE mobile networks worldwide.

2014: Facebook buys WhatsApp for $19B.

What does an Erlang do?

- Uses shared-nothing, Actor Model of concurrency.
- Hot code reloading.
- Processes can supervisor each other.
- Has tools to introspect a running system.
- No leaky state.
- Take advantage of multiple cores.
- Avoid downtime for code upgrades.
- Debug multiple processes easily.
- Process failure is no longer as scary!

Why not use the Erlang?

Elixir has

- Better Syntax
- Improved abstractions
- A core build tool
- A cool web framework (Phoenix)

Elixir

Example Code

```
defmodule Issues.GithubIssues do
  ...
  def fetch(user, project) do
    Logger.info "Fetching user #{user}'s project #{project}"
    issues_url(user, project)
    |> HTTPoison.get(@user_agent)
    |> handle_response
  end

  def handle_response({:ok, %{status_code: 200, body: body}}) do
    Logger.info "Successful response"
    Logger.debug fn -> inspect(body) end
    {:ok, Poison.Parser.parse!(body)}
  end

  def handle_response({_, %{status_code: status, body: body}}) do
    Logger.error "Error #{status} returned"
    {:error, Poison.Parser.parse!(body)}
  end
end
```

Elixir

Pretty, Ruby-like syntax:

- Code in do/end blocks
- No ending semi-colons
- Implicit return statements
- Optional parenthesis
- Stabby lambdas (fn -> stuff)

```
defmodule Issues.GithubIssues do
  ...
  def fetch(user, project) do
    Logger.info "Fetching user #{user}'s project #{project}"
    issues_url(user, project)
    |> HTTPoison.get(@user_agent)
    |> handle_response
  end

  def handle_response({:ok, %{status_code: 200, body: body}}) do
    Logger.info "Successful response"
    Logger.debug fn -> inspect(body) end
    {:ok, Poison.Parser.parse!(body)}
  end

  def handle_response({_, %{status_code: status, body: body}}) do
    Logger.error "Error #{status} returned"
    {:error, Poison.Parser.parse!(body)}
  end
end
```

Elixir

Cool, new features

- Pipes (|>)
- Multiple function definitions
- Pattern matching
- Unused variables (_)
- Parameter destructuring

```
defmodule Issues.GithubIssues do
  ...
  def fetch(user, project) do
    Logger.info "Fetching user #{user}'s project #{project}"
    issues_url(user, project)
    |> HTTPoison.get(@user_agent)
    |> handle_response
  end

  def handle_response({:ok, %{status_code: 200, body: body}}) do
    Logger.info "Successful response"
    Logger.debug fn -> inspect(body) end
    {:ok, Poison.Parser.parse!(body)}
  end

  def handle_response({_, %{status_code: status, body: body}}) do
    Logger.error "Error #{status} returned"
    {:error, Poison.Parser.parse!(body)}
  end
end
```

Another example

Looks like it might be Rails?

It’s actually a Phoenix Router

The browser pipeline gets run before hitting a controller.

```
defmodule WebApp.Router do
  use WebApp.Web, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
    plug WebApp.Auth, repo: WebApp.Repo
  end

  scope "/", Rumbl do
    pipe_through :browser

    get "/", PageController, :index
    resources "/users", UserController
    resources "/sessions", SessionController
    get "/watch/:id", WatchController, :show
  end
end
```

Phoenix: A Web Framework

Phoenix: History

- Created in 2014 by Chris McCord
- Wanted to make WebSockets easier
- Released 1.0.0 in August 2015
- After 1.0.0, got up to 2 million WebSockets on a single server (40 cores / 128 GB RAM)

Phoenix: Basics

- Written in Elixir, based on Plug
- MVC-style architecture
- Separate Model and Repository
- EEx templates look like ERB or EJS templates
- Phoenix Channels create a process per websocket connection
- Elixir build tool manages Elixir stuff; Nodejs build tools manage CSS & JS

Phoenix: Controller

Functions defined for REST actions

```
defmodule Rumbl.UserController do
  use Rumbl.Web, :controller
  plug :authenticate_user when action in [:show]

  alias Rumbl.User

  def show(conn, %{"id" => id}) do
    user = Repo.get(Rumbl.User, id)
    render conn, "show.html", user: user
  end

  def new(conn, _params) do
    changeset = User.changeset(%User{})
    render conn, "new.html", changeset: changeset
  end

  def create(conn, %{"user" => user_params}) do
    changeset = User.registration_changeset(%User{}, user_params)
    case Repo.insert(changeset) do
      {:ok, user} ->
        conn
        |> Rumbl.Auth.login(user)
        |> put_flash(:info, "#{user.name} created!")
        |> redirect(to: user_path(conn, :index))
      {:error, changeset} ->
        render(conn, "new.html", changeset: changeset)
    end
  end
end
```