Gleam v1

Robert Ellen

2024/05/14

Summary

> Gleam is a statically-typed, functional programming language for the Erlang/BEAM and JavaScript ecosystems.
> It just went 1.0 in March 2024, and is considered stable for industrial use.
> Gleam makes some interesting language design choices, opting for simplicity and ergonomics.
> It's further removed from it's Erlang roots more than other popular BEAM languages such as Elixir.
> Gleam is simple without being patronising, and seems like a great choice for building small services where you want to roll up your sleeves and build things from scratch

Gleam

Gleam is a friendly language for building type-safe systems that scale!

https://gleam.run

A functional programming language in the Erlang/BEAM ecosystem with a static type-system designed for simplicity and ergonomics. Oh, and it happens to also target JavaScript.

What is the Erlang Ecosystem?

A group of languages, libraries, frameworks, and applications that are implemented on top of the Erlang virtual machine, the BEAM.

Languages include:

ERLANG

elixir

Erlang

...plus dozens more

What is the Erlang Ecosystem?

Libraries and frameworks include:

OTP

What is the Erlang Ecosystem?

Built around a shared value in:

- massive concurrency
- fault-tolerance
- simplicity
- acknowledging the errors will occur so lets deal with them
- but at the same time, “Let it crash”

Brief history of Erlang

covered in my 2013 talk, but tonight…

Brief history of Erlang

Brief history of Erlang

> developed in the mid 1980s at Ericsson
> to run on next generation telephone switches

- concurrent
- fault-tolerant
- distributed
- soft real-time

> solved web-scale in the '80s

Brief history of Erlang

- fell out of favour at Ericsson in the late '90s
- open-sourced in 1998
- reports of market penetration of > 50% in mobile telephony switches
- reports of 1200k LOC and nine nines of uptime on the AXD301 switch

The BEAM

- Erlang's virtual machine / runtime system
- lightweight processes, SMP
- multi-generational, per-process garbage collector
- asynchronous, location-transparent, message-passing IPC
- hot-code-loading
- powerful REPL and introspection tools
- new JIT compiler (BEAM bytecode to machine code)

The OTP libraries

- Erlang's ”standard library”
- dozens of modules providing various typical stdlib stuff
- core of which are for supervision trees
  - `gen_server`
  - supervisors

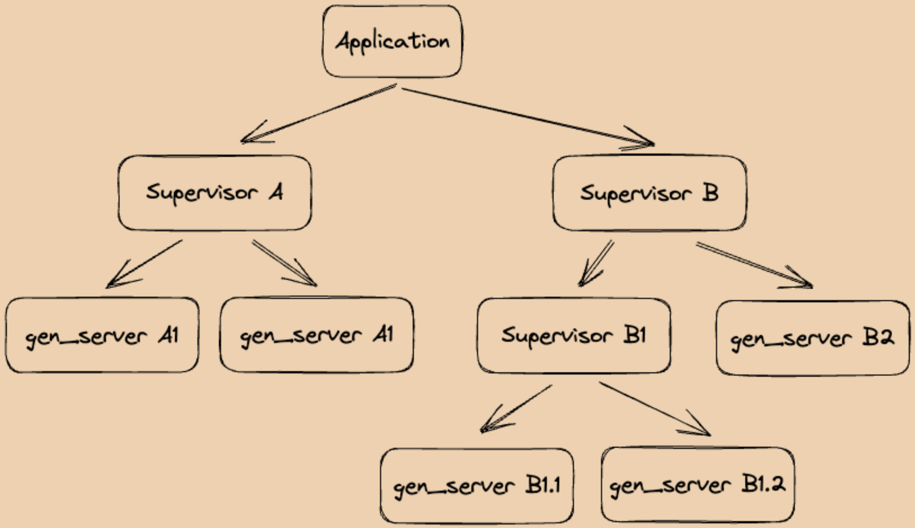
`gen_server`

- generic server process
- implements a behaviour with callbacks:
  - initialisation
  - handling calls (known synchronous messages)
  - handling casts (known asynchronous messages)
  - handling out-of-band messages (e.g. timers)
  - graceful termination
  - hot-code-loading

supervisor

- responsible for starting, stopping, and monitoring child processes
- child specifications dictate how child processes are restarted if they crash, whether other processes are affected by a crash, and if the supervisor itself should shut down

supervision trees



Elixir

There will be some comparing and contrasting of Gleam with Elixir...

Brief history of Elixir

> created by José Valim starting in 2012
> inspired by Erlang, Ruby, and, to a lesser extent, Lisp
> like Erlang, the language is quite stable

Features of Elixir

- Ruby-like syntax while retaining most of Erlang's semantics
  - ..., immutable data, HoF, side-effects anywhere, dynamically-typed, ...
- interoperate with Erlang
- hygienic macros
- highly ergonomic build tool: `mix`
- modern package manager: `hex`
- excellent unit test tool: exunit
- opinionated formatter
- drops strict SSA in favour of rebinding

Gleam

Gleam

```gleam
// https://gleam.run
import gleam/io

pub fn main() {
  io.println("hello, friend!")
}
```

Gleam

> created by Louis Pilfold (https://github.com/lpil)
> functional: immutable data and HoF, but impure
> statically-typed: Hindley-Milner type system
> strongly-typed
> strict semantics

Gleam

```gleam
import gleam/io
import gleam/list

fn sum(xs: List(Int)) -> Int {
  list.fold(xs, 0, fn(acc, x) { acc + x })
}

pub fn main() {
  io.debug(sum([1, 2, 3])) // prints "6"
}
```

Gleam

- compiles to BEAM or JavaScript
- FFI to JS and Erlang/Elixir - both at the same time with fallback
  - TS type annotations
  - can't call Elixir macros - need to wrap
- all tooling provided by `gleam`
- `gleam` is written in Rust
- went v1.0 in March 2024
- has a fantatic interactive language tour
  - https://tour.gleam.run/

Gleam philosophy

“Gleam has no null, no implicit conversions, no exceptions, and always performs full type checking. If the code compiles you can be reasonably confident it does not have any inconsistencies that may cause bugs or crashes.”

“Gleam lacks exceptions, macros, type classes, early returns, and a variety of other features, instead going all-in with just first-class-functions and pattern matching.”

(https://gleam.run)

Type-inference

```gleam
import gleam/list

fn sum(xs) {
  list.fold(xs, 0, fn(acc, x) { acc + x })
}

pub fn main() {
  sum(["a", "b", "c"])
}
```

Type-inference

```gleam
import gleam/list

fn sum(xs) {
  list.fold(xs, 0, fn(acc, x) { acc + x })
}

pub fn main() {
  sum(["a", "b", "c"])
}

// Produces the compile-time error ...
// 8 │   sum(["a", "b", "c"])
//   │       ^^^^^^^^^^^^^^^
// Expected type:
//     List(Int)
// Found type:
//     List(String)
```

Type aliases...are not a new type

```gleam
import gleam/io

pub type SpecialString = String

pub fn main() {
  let normal: String = "I'm a string"
  let special: SpecialString = "I'm a special string"

  io.debug(normal == special) // False
}
```

Opaque types with smart constructors

```gleam
import gleam/io

pub opaque type OpaqueString {
 OpaqueString(inner: String)
}

pub fn make(str: String) -> OpaqueString {
  OpaqueString(str <> " (totally a string)")
}

pub fn main() {
  let normal: String = "I'm a string"
  let special: OpaqueString = make("I'm not a string")

  io.debug(normal == special) // Compile error
}
```

Data types

```gleam
import gleam/io
import gleam/string

pub type Person {
  Person(name: String, age: Int)
}

pub fn main() {
  let ashley = Person("Ashley", 42)
  let message =
    ashley.name <> " is " <> string.inspect(ashley.age) <> " years old"
  io.println(message)
  // "Ashley is 42 years old"
}
```

Data types

```gleam
pub type Shape {
  Square(side: Float)
  Rectangle(length: Float, width: Float)
  Circle(radius: Float)
}

pub fn area(shape: Shape) -> Float {
  case shape {
    // note the *. operator
    Square(side: s) -> s *. s
    Rectangle(length: l, width: w) -> l *. w
    Circle(radius: r) -> 3.14 *. r *. r
  }
}
```

Data types - no pattern matching in function head

```
pub type Shape {
  Square(side: Float)
  Rectangle(length: Float, width: Float)
  Circle(radius: Float)
}

// something like this adapted from Erlang/Elixir won't compile
pub fn area(Square(side: s)) {
    s *. s
}

pub fn area(Rectangle(length: l, width: w)) {
    l *. w
}
```

Type variables - parametric polymorphism but not HKT

```gleam
import gleam/io
import gleam/string

fn my_fold(collection: List(a), accumulator: b, reducer: fn(a, b) -> b)
  case collection {
    [] -> accumulator
    [x, ..xs] -> { // expression block
      let new_acc = reducer(x, accumulator)
      my_fold(xs, new_acc, reducer)
    }
  }
}

pub fn main() {
  io.debug(my_fold([1, 2, 3], "", fn(a, b) { string.inspect(a) <> b }))
  // "321"
}
```

Type variables - in type definitions too

```
...
pub type Option(a) {
  Some(a)
  None
}
...
pub type MyEither(a, b){
  Left(a)
  Right(b)
}
...
```

Nil - the unit type

```gleam
// gleam_stdlib/src/gleam/io.gleam
...
pub fn println(string: String) -> Nil {
  do_println(string)
}

@external(erlang, "gleam_stdlib", "println")
@external(javascript, "../gleam_stdlib.mjs", "console_log")
fn do_println(string string: String) -> Nil
...
```

todo - “top” type

```gleam
import gleam/io

// prints "one" then errors at two()
pub fn main() {
  one()
  two()
  three()
}

pub fn one() {
  io.println("one")
}

pub fn two() {
  todo as "two() is not implemented yet!"
}
```

use - a monad if you squint

```
import gleam/io
import gleam/list
import gleam/string

fn upcase_duplicate(strings) {
  use str <- list.map(strings)
  let dup = string.append(to: str, suffix: str)
  string.uppercase(dup)
}

pub fn main() {
  ["hello", "world!!"]
  ▷ upcase_duplicate()
  ▷ io.debug()
}
```

use - a monad if you squint

```gleam
import gleam/io
import gleam/result
import gleam/string

pub fn main() {
  let res = {
    use data <- result.try(read_data())
    use record <- result.map(find_record(data))
    format(record)
  }

  case res {
    Ok(formatted) -> io.println(formatted)
    Error(error) -> io.println("ERROR: " <> error)
  }
}
```

use - a monad if you squint

```
fn read_data() {
  Ok(#("Ashley", 42))
}

fn find_record(record) {
  Ok(record)
}
```

use - a monad if you squint

```gleam
import gleam/io
import gleam/result
import gleam/string

// prints "#("Ashley", 42)"
pub fn main() {
  let res = {
    use data <- result.try(read_data())
    use record <- result.map(find_record(data))
    format(record)
  }

  case res {
    Ok(formatted) -> io.println(formatted)
    Error(error) -> io.println("ERROR: " <> error)
  }
}
```

use - a monad if you squint

```
fn read_data() {
  Error("couldn't read data")
}

fn find_record(record) {
  Error("not found")
}

fn format(record) {
  string.inspect(record)
}
```

use - a monad if you squint

```gleam
import gleam/io
import gleam/result
import gleam/string

// prints "ERROR: couldn't read data"
pub fn main() {
  let res = {
    use data <- result.try(read_data())
    use record <- result.map(find_record(data))
    format(record)
  }

  case res {
    Ok(formatted) -> io.println(formatted)
    Error(error) -> io.println("ERROR: " <> error)
  }
}
```

No type classes, traits, or interfaces

https://mckayla.blog/posts/all-you-need-is-data-and-functions.html

```gleam
import gleam/io

pub type MyDebug(a) {
  MyDebug(debug: fn(a) -> String)
}

pub fn main() {
  let my_dbg = MyDebug(debug: fn(str) { io.debug("The string is: " <> st
  my_dbg.debug("Hello, World!!!")
}
```

Some gleam libraries - no magic

- Lustre: Elm-inspired frontend framework
  - https://github.com/lustre-labs/lustre
- cgi: CGI in gleam
  - https://github.com/lpil/cgi
- wisp: web framework
  - https://github.com/gleam-wisp/wisp

OTP concurrency

Gleam being statically-typed does not wrap OTP's concurrency entities wholesale like Elixir. Instead providing an OTP-interoperable library built using basic BEAM primatives.

- https://github.com/gleam-lang/otp
- Actor hierarchy
  - Process - a wrapper around BEAM processes, all other actors based on process
  - Actor - like a `gen_server`, receives messages, updates state
  - Task - run a function and quit
  - Supervisor - manage other processes, provides fault-tolerance

OTP and BEAM ecosystem

> there are several other libraries that wrap OTP features, like ETS
> if something from the OTP libaries can be made type-safe, it's an FFI away

Where to use Gleam

> Gleam is simple without being patronising, no magic

> seems like a great choice for building small services

> you'll have to roll up your sleeves and build more things from scratch

> small-to-medium web apps with less CRUD

> isomorphic apps (share common types across FE and BE packages)

> learning static-typing, polymorphism, immutable data structures

Thank you

I'll post slides soon.