cloudbees.com

Comparing Elixir and Go

23–29 minutes

Elixir and Go have both grown significantly in popularity over the past few years, and both are often reached for by developers looking for high concurrency solutions. The two languages follow many similar principles, but both have made some core tradeoffs that affect their potential use cases.

Let's compare the two by taking a look at their backgrounds, their programming styles, and how they deal with concurrency.

Foundations

Go/Golang has been developed by Google since 2009 and runs as a compiled native binary for the architecture where it's deployed. It began as an experiment to create a new programming language that addressed the main criticisms of other programming languages while maintaining their strengths.

Go does an excellent job of achieving the balance of development speed, concurrency, performance, stability, portability, and maintainability that it set out to reach. As a result, Docker and InfluxDB are built with Go while many major companies including Google, Netflix, Uber, Dropbox, SendGrid and SoundCloud are using it for assortments of tools.

Elixir has been developed by Jose Valim at Plataformatec since 2011 and runs on the BEAM virtual machine, also known as the Erlang VM.

Erlang has been developed by Ericsson since 1986 for use in highly available, distributed phone systems. It has since expanded

into numerous other areas, such as web servers, and has achieved nine 9s of availability (31 milliseconds/year of downtime).

Elixir was designed to enable higher extensibility and productivity in the Erlang VM while maintaining compatibility with the Erlang ecosystem. It achieves that goal by allowing use of Erlang libraries in Elixir code and vice versa.

> For purposes of this article and to avoid repetition, we'll refer to Elixir/Erlang/BEAM as "Elixir."

A number of companies are using Elixir in production, including Discord, Bleacher Report, Teachers Pay Teachers, Puppet Labs, Seneca Systems, and FarmBot. Many other projects are built with Erlang, including WhatsApp, Facebook's chat service, Amazon's CloudFront CDN, Incapsula, Heroku's routing and logging layers, CouchDB, Riak, RabbitMQ, and about half of the world's phone systems.

Programming Style

The core principles of each run-time need to be understood to make a solid comparison of Elixir and Go, as these building blocks are where everything else is derived.

Go is a language that will be more familiar to people coming from a traditional C-style programming background, even though the language makes some decisions that favor a functional programming style. You'll see static typing, pointers, and structs that will feel very familiar.

Functions can be created and attached to struct types, but the style is more composable to facilitate the growth of projects over time. Rather than embedding the functions within an Object that must be expanded, the function can be created anywhere and attached to the type.

If a method needs to be called with more than one type of struct, an interface can be defined for the method to provide greater flexibility.

Unlike interfaces from typical object-oriented programming languages where an object must be defined initially to implement a specific interface, interfaces in Go are automatically applied to anything that matches them. Here is a good example of Go interface code.

Elixir favors a more functional style but blends some principles from object-oriented languages that make the transition seem less foreign.

Variables are immutable, and message passing is used so no pointers are passed around, meaning that function calls are very literal in their operation. Arguments are passed in, a result is returned with no side effects. This simplifies a number of aspects of development, including testing and code readability.

Due to immutable data, common operations such as for-loops aren't available because incrementing a counter isn't an option. Instead, recursion is used for these operations, although the Enum library provides common iterative patterns in a manner that is comfortable.

Because recursion is so heavily used, Elixir also utilizes tail call optimization. The compiler detects the last function call in every code path and transforms the last call to the jump so the call stack doesn't grow.

Elixir utilizes pattern matching everywhere, which is very similar to the way that Go leverages interfaces. With Elixir, a function can be defined as:

```
def example_pattern(%{ "data" => %{ "nifty" => "bob", "other_thing" => other}}) do
  IO.puts(other)
end
```

Using a map pattern as an argument, that function will only be called if a map with a key of `data` is passed in that has a nested map which contains a `nifty` key that has a value of `"bob"` and an

`other_thing` key. The variable `other` would be set to its value.

Pattern matching is used everywhere, from function arguments to variable assignments and especially recursion. Here are a few good examples of pattern matching to make sense of it. Structs can be defined as types and then utilized in pattern matching as well.

These approaches are both very similar in principle. Both separate data structures from data operations. Both use matching to define function calls, Go via interfaces and Elixir via pattern matching.

Even though Go allows functions to be called by a specific type, `g.area()`, it's essentially the same thing as calling `area(g)`. The only difference in the two is that with Elixir, `area()` would have to return a result while Go could potentially manipulate a reference in memory.

Both languages are very composable because of this approach, meaning that large inheritance trees don't have to be manipulated, extended, injected, or rebuilt over the life-cycle of a project. This is a significant boon for large projects over time.

The biggest difference here is that with Go, the patterns are defined outside of a function for reuse but can result in creating a lot of duplicate interfaces if they aren't well organized. Elixir can't reuse the patterns as easily, but the pattern is always defined in the exact place where it is used.

Elixir utilizes "strong" typing rather than static typing, and much of it is inferred. Within Elixir, there is no operator overloading, which can seem confusing at first if you want to use a + to concatenate two strings. In Elixir you would use <> instead.

This will seem tedious if you don't understand the reason behind it. The compiler is able to use the explicit operators to infer that whatever is on either side of that plus sign must be a number. Likewise, either side of the <> must be a string.

Strong typing essentially means dynamic typing where the compiler (via dialyzer) can catch virtually every type, with the exception of

ambiguous arguments in a pattern match (using an _ operator to indicate a variable or argument that isn't used to avoid allocated memory for it, just as with Go). Code comments can be used to define types in these exception cases. The benefit of this is that you gain most of the benefits of static typing without losing the flexibility and meta programming perks that come from dynamic typing.

Elixir files can use a .ex extension for compiled code or a .exs extension for scripts that are compiled at run-time, such as shell scripts and so on. Go is always compiled, however the Go compiler is so fast that even with huge code bases it can feel almost instantaneous.

Concurrency

Concurrency is where the meat of the comparison comes from, and now that you have a basic overview of the language styles, the rest of this will make more sense.

Traditionally, concurrency has involved threads, which were more heavyweight. More recently, languages have taken to using "light threads" or "green threads," which essentially use a scheduler inside of a single thread to let different logic take turns.

This type of concurrency pattern is much more memory efficient but relies on the run-time to dictate its flow. JavaScript has utilized this style for years in the browser. As a simple example, when you hear the term "non-blocking I/O" with JavaScript, it means that code executing in the thread relinquishes control back to the scheduler to do something else when an I/O operation begins.

Cooperative versus preemptive scheduling

Both Elixir and Go utilize a scheduler to achieve their concurrency models, although both languages naturally spread across multiple processors, while JavaScript does not.

Elixir and Go both implement scheduling differently. Go utilizes cooperative scheduling, which means that running code must

relinquish control back to the scheduler for another operation to have a turn. Elixir utilizes preemptive scheduling, in which each operation has a preset execution window that will be enforced no matter what.

Cooperative scheduling is more efficient in terms of benchmarking, as preemptive scheduling creates additional execution overhead to enforce. Preemptive scheduling is more consistent, meaning that millions of small operations can't be delayed by a single large operation that doesn't relinquish control.

Go programmers have the ability to insert `runtime.Gosched()` in their code to force more check-ins with the scheduler as a precautionary measure for potential problem code. Run-time enforcement allows more trust of third-party libraries and real-time systems.

Goroutines versus processes

In order to execute a concurrent operation in Go, we use a goroutine, which is as simple as typing `go` before your method...any method. So we can go from:

```
hello("Bob")
// To...
go hello("Bob")
```

Elixir is very similar in this regard. Instead of goroutines, you spawn processes (not OS processes, for the sake of clarity). Also note that functions must be inside modules in Elixir.

```
# From...
HelloModule.hello("Bob")
# To...
spawn(HelloModule, :hello, ["Bob"])
# Or by passing a function
spawn fn -> HelloModule.hello("Bob") end
```

The main difference between what's happening here is that the `go`

operation returns nothing while the `spawn` operation returns an id for the process.

Both systems utilize a similar communication style with these routines via messages queues. Go calls them channels, while Elixir has process mailboxes.

With Go, a channel can be defined so that anything can pass messages to if it has the channel reference. With Elixir, messages are sent to a process either via the process id or a process name. Channels in Go are defined with types for the messages, while process mailboxes in Elixir utilize pattern matching.

Sending messages to an Elixir process is equivalent to sending a message to a Go channel monitored by a goroutine. Here's a simple example:

Go channel

```
messages := make(chan string) // Define a channel that accepts strings
go func() { messages <- "ping" }() // Send to messages
msg := <-messages // Listen for new messages
fmt.Println(msg)
```

Elixir process mailbox

```
send self(), {:hello, "world"}
receive do
  {:hello, msg} -> msg # This reciever will match the pattern
  {:world, msg} -> "won't match"
end
```

Both have the ability to set timeouts when listening for messages as well. Because Go has shared memory, goroutines can also directly transform an in-memory reference, although a mutex lock must be used to avoid contention. Ideally, a single goroutine should listen on a channel for updates to shared memory to avoid the need for a mutex lock.

Beyond this functionality is where things start to expand.

Erlang defines a set of patterns for best practices when utilizing concurrency and distribution logic all bundled under "OTP". In most cases with Elixir code, you'll never touch the raw `spawn` and `send/receive` functions, deferring to the abstractions for this functionality.

Wrappers include `Task` for simple `async/await` style calls; `Agent` for concurrent processes which maintain and update a shared state; `GenServer` for more complex custom logic.

In order to constrain maximum concurrency to a particular queue, Go channels implement buffers which receive a defined number of messages (blocking the sender of at the limit). By default, channels block until something is ready to receive the messages, unless that buffer has been set.

Elixir process mailboxes default to unlimited messages but can utilize `Task.async_stream` to define max concurrency for an operation and blocking senders in the same way as limited buffers on a channel.

Routines in both languages are inexpensive: goroutines are 2KB each, while Elixir processes are 0.5KB each. Elixir processes have their own isolated heap spaces which are individually reclaimed when the process finishes, while goroutines utilize shared memory and an application-wide garbage collector to reclaim resources.

Error handling

This is probably the single biggest difference in the languages. Go is very explicit about error handling at all levels, from function calls to panics. In Elixir, error handling is considered "code smell." I'll take a second to let you read that again.

So how does this all work? Remember earlier when we talked about Elixir's `spawn` call returning a process ID? That's used for more than just sending messages. It can also be used to monitor the process and check whether it's still alive.

Because processes are so inexpensive, the standard mode of operation in Elixir is to create two. One to run the process and another to supervise the process.

This approach is called the supervisor pattern, and Elixir applications tend to operate within a supervision tree. The supervisor spawns the process with a different function called `spawn_link` under the hood that will crash the spawning process if the spawned process itself crashes. Supervisors handle these and instantly restart the process.

Here is a simple example using a supervised process to do division. Dividing by zero crashes the process, which the supervisor immediately restarts, allowing future operations to continue. It's not that error handling doesn't exist, it's just implemented by supervisors transparently.

By contrast, Go has no way of tracking the execution of individual goroutines. Error handling is very explicit at every level, leading to a lot of code that looks like this:

```
b, err := base64.URLEncoding.DecodeString(cookie)
if err != nil {
  // Handle error
}
```

The expectation here is that error handling will exist at the point where an error can occur, whether in a goroutine or not.

Goroutines can pass error cases to channels in the same way. However, if panics occur, each goroutine is responsible for having its own recovery condition met, or it will crash the entire application. Panics are not equivalent to exceptions in other languages as they are intended to be system level "stop everything" events.

This condition is perfectly encapsulated by an Out of Memory error. If a goroutine triggers an Out of Memory error, the entire Go application will crash even with proper error handling because of the shared memory state of the run-time.

With Elixir, because each process has its own heap space, a max heap size can be set per process that would crash the process if reached but would then be independently garbage-collected and restarted without affecting anything else.

That's not to say that Elixir is bullet proof. The VM itself can still run out of memory via other means. But it's a containable problem within processes.

This isn't intended to be a knock against Go either. This is a problem faced by virtually every language with shared memory, meaning most every language you've ever heard about. It's specifically a strength of the way that Erlang/Elixir was designed.

Go's approach forces developers to handle errors directly at the point that they would occur, which requires explicit design thought and can lead to very well-thought-out applications.

The main point of the Elixir model is an application that can be expected to run forever, as Joe Armstrong put it. Just as you can manually call the scheduler with Go, you can also manually implement a version of supervisors in Go via the suture library.

Note: Within handlers that you'll implement with Go for most servers, panics are already addressed. Therefore, a crash within a web request that wasn't critical enough to kill the entire application won't. You'll still have to address it on your own goroutines though. Don't let this explanation imply that Go is fragile, because it is not.

Mutable versus Immutable

Understanding the trade-offs that come from mutable versus immutable data is important in a comparison between Elixir and Go.

Go uses the same style of memory management that most programmers are experienced with via shared memory, pointers, and data structures that can be changed or reassigned. For dealing with transformations of large data structures, this can be much more efficient.

Immutable data within Elixir utilizes copy-on-write. That means that within the same heap space it really is just passing a pointer to the data, but as soon as you want to do something to it, a new copy is created.

A list of values, for example, would pass a list of pointers to those immutable values in memory, while sorting would return a list of pointers in a different order since the values in memory themselves can be relied on to go unchanged. Changing a value in the list would return a new list of pointers, including a pointer to the different value. If I want to pass that list to another process however, the entire list including values would be copied to the new heap space.

Clustering

The other trade-off that comes from mutable versus immutable data comes from clustering. With Go, you have the ability to make remote procedure calls very seamlessly if you want to implement them, but because of pointers and shared memory, if you call a method on another box with an argument that references to something on your machine, it can't be expected to function the same way.

With Elixir, because everything operates with message passing, the entire application stack can be clustered across any number of machines. Data is passed into a function that returns a response. There is no in-memory transformation that occurs as the result of any function call, which allows Elixir to call functions on different heap spaces, different machines, or different data centers entirely in the exact same way as any other function in its local heap space.

Many applications don't require clustering, but there are a number of applications that benefit from it significantly, such as communications systems like chats where users are connected from different machines or horizontally distributed databases. Both have common solutions available via the Phoenix framework's

channels and Erlang's Mnesia database respectively. Clustering is critical to any application's ability to horizontally scale without depending on bottlenecked central relay points.

Libraries

Go has an extensive standard library which will allow most developers to do virtually anything without needing to reach for third-party packages.

The Elixir standard lib is more concise but also includes the Erlang standard lib, which is more thorough and includes three built-in databases, ETS/DETS/Mnesia. Other packages must be pulled in from third-party libraries.

Both Elixir and Go have plenty of third-party packages available. Go uses a direct `go get` command to import packages remotely, while Elixir uses Mix, a build tool that invokes the Hex package manager in a manner that will be familiar for users of most languages.

Go is still working to standardize a full package-management solution across the language. Between that and the extensive standard library, most people in the Go community seem to favor sticking with the standard lib where possible. There are several package-management tools already available.

Deployment

Go deployments are straightforward. A Go application is compiled into a single binary including all of its dependencies that can then be run natively, cross platform, wherever it's going. The Go compiler can compile binaries for just about any destination architecture, regardless of the type of machine that you're running on. This is one of Go's greatest strengths.

Elixir actually comes with a lot of deployment options, but the primary method is via the excellent distillery package. This wraps up your Elixir application into a binary with all of its dependencies

that can be deployed to its destination.

The biggest difference between the two languages is that compilation for the destination architecture has to be done on that same architecture with Elixir. The documents include several workarounds for this scenario, but the simplest method is to build your release within a Docker container that has the destination architecture.

With both of those approaches, you just stop the currently running code, replace the binary, and restart it as you would with most blue-green style deployments today.

Hot reloading

Elixir also comes with another deployment option that the BEAM makes possible. It's a little bit more complex, but certain types of applications can greatly benefit from it. It's called a "hot reload" or "hot upgrade."

Distillery goes out of its way to make this easy by just letting you tac on the `--upgrade` flag to your release build command, but that still doesn't mean you should always use it.

Before talking about when you'd use it, we need to understand what it does.

Erlang was developed to power phone systems (OTP stands for Open Telecom Platform) and currently powers about half of them on the planet. It was designed to never go down, which is a complicated problem for deployments when you have active phone calls running through the system.

How do you deploy without disconnecting everybody on the system? Do you stop new traffic from coming to that server and then politely wait for every single call to finish?

The answer is no, and that's where hot reloads come in.

Because of the heap isolation between processes, a release upgrade can be deployed without interrupting existing processes.

The processes that aren't actively running can be replaced, new processes can be deployed side by side with currently running processes while absorbing the new traffic, and the running processes can keep on trucking until they finish their individual jobs.

That allows you do deploy a system upgrade with millions of calls and let the existing calls finish on their own time without interruption. Imagine replacing a bunch of bubbles in the sky with new bubbles...that's basically how hot reloading works; the old bubbles hang around until they pop.

Understanding that, we can potentially see some scenarios where it might come in handy:

1. A chat system utilizing websockets where users are connected to specific machines
2. A job server where you may need to deploy an update without interrupting jobs in progress
3. A CDN with huge transfers in progress on a slow connection next to small web requests

For websockets specifically, this allows deployments to a machine that might have millions of active connections without immediately bombarding the server with millions of reconnection attempts, losing any in progress messages. That's why WhatsApp is built on Erlang, by the way. Hot reloading has been used to deploy updates to flight computers while planes were in the air.

The drawback is that hot reloading is more complicated if you need to roll back. You probably shouldn't use it unless you have a scenario where you really do need it. Knowing you have the option is nice.

Clustering is the same way; you don't always need it, but it's indispensable when you do. Clustering and hot reloads go hand in hand with a distributed system.

Conclusions

This article has been a long journey, but hopefully it provides a solid overview of the differences between Elixir and Go. The most helpful way to think about these two languages that I've found is to think of Elixir as an operating system and Go as a specialized program.

Go works extremely well for engineering exceptionally fast and very focused solutions. Elixir creates an environment where many different programs can coexist, operate, and talk to each other without interfering with each other even during deployments. You'd utilize Go to build individual microservices. You'd build multiple microservices within a single Elixir umbrella.

Go is more focused and simpler to pick up. Elixir is very straightforward once you get the hang of it, but the world of OTP and expanse of Erlang can be intimidating if your goal is to learn it all before you use it.

Both are excellent languages that are at the top of my list of recommendations to do virtually anything in programming.

For very focused code, portable system level tools, performance intensive tasks, and APIs, Go is very hard to beat. For full-stack web applications, distributed systems, real-time systems, or embedded applications, I'd reach for Elixir.