Hello and welcome to Go as a Top Level SDK!

I'm Robert Burke, a Senior Software Engineer at Google, and a committer on the Apache Beam project.
I've been working with the Go SDK on behalf of Google for over 4 years now.

Today I'm going to focus on what makes a Beam SDK, with a focus on how we implemented one with Go.
A language's features affects it's design as a Beam SDK, and Go is very different from Python and Java.

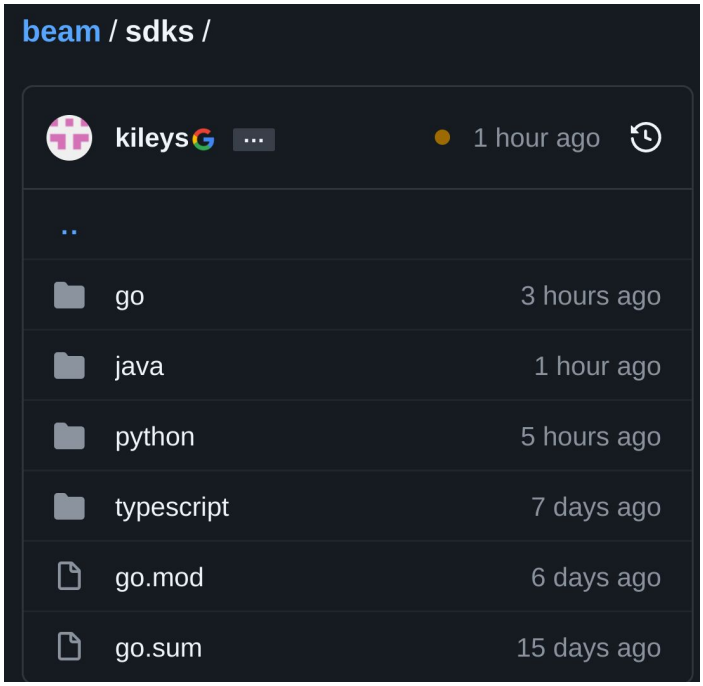
But First: What does a Top Level SDK mean?

What's a Top Level SDK?

What the heck does a top level SDK even mean?

When it was suggested I do this talk, I originally didn't have a clue.
It's not a term we use in Beam.
It doesn't show up in the glossary

So ultimately, this talk is what I feel being a top level Beam SDK is about as far as the user perspective is concerned.



If I'd have to draw a line to start, a Top Level SDK is one that lives in the Beam github repository, and is maintained by the Beam community.

But that doesn't quite cover it exactly.
We currently have 4 SDKs in the repo Java, Python, Go , and now, Typescript!

Typescript was only added to the repo a week ago, and is experimental. So while it's in the Beam repository, I wouldn't call it a top level SDK yet.

That leaves the other three.
Java was first, Python came second, and after a long period of being experimental, came the Go SDK.

So that gives us a few requirements to start with.

Top Level SDK Requirements

From a user perspective

- Implements the Beam Model
- In Beam Repository
- Maintained by Beam Community
- Not Experimental

A Top Level SDK is an SDK that:
Implements the Beam Model
Lives in the Beam Repository
Maintained by the Beam Community
And is Not Experiemental.

These are useful from the User point of view since users are part of the community, and they're empowered to change and improve the SDKs.
This is the value of Open Source.

Implementing the Beam Model to some degree is required. SDKs that don't do this, can't be Beam SDKs, let alone a Top Level one.
Being in the Beam repo makes them easier to find and contribute to.
Not being experimental means the users can trust that the SDK is tested and isn't likely to introduce breaking changes.
Working code written should continue to work between updates.

But this doesn't feel like it's enough, does it?

The Go SDK is all of these. But what else makes it a Top Level SDK?

Perhaps we'll figure out more features of a Top Level SDK as we go on.
For that we'll need to look deeper.

So for those of you who don’t know…

The Go Programming Language

Learn more at go.dev

- Fast Compiles
- Simple Syntax
- Robust Standard Library
- Strongly typed
- Statically Compiled
- First class functions
- Interfaces
- Built-in Maps & Slices
- Built-in Concurrency
- Non-inheritance based type system
- No function or method overloading
- ~~No~~ Generics (since Go 1.18)

Go is a Programming Language.

The apocryphal story is that Go was originally conceived of during a long C++ compilation.
The Go Authors took that as a sign that programming could be faster, and focused on making a new language that's easier to read, easier to write, compiles quickly and executes quickly.

This ethos drove the design of the language, leading to one that's Strongly typed, statically compiled, avoids inheritance and overloading.

On top of that it bring concurrency into the language itself, instead of leaving it to libraries.
It also has a standard library that is useful for the internet age.

None of these ideas were new, but they had not come together in this way before.

But others seem to have agreed, and it's become the Language Upon Which The Cloud Is Built, powering Docker and Kubernetes.

Until very recently, Go did not have Generics.
This talk may become out of date quickly as a result, as we're already considering how we can change the SDK with Generics in Go!

Like with any programming language, Go’s features and affordances affect how people design APIs and the resulting SDKs.
That directly affects the user experience, from initial authoring, to how to deploy it to production.

Deploying Go Pipelines to Production

Let's start with that last point: deploying to production, and move backwards.

In the interest of time I'm going to omitting points of how Beam is architected, nor will this talk teach Go generally.
There are plenty of other talks that have covered that before.
There should be time for questions after the talk though,
and I'll be answering questions in the Beam College Slack afterwards.

Right. Deployment!

Statically Compiled Executables

- Executable is OS + Architecture Specific
  - Developing on the same OS + Architecture? Same executable used as the worker binary
  - Mix? Cross compile to target architecture.
- Cross Compiling Go Pipelines

```
$ go build path/to/my/pipeline -o output/launcher

$ GOOS=linux GOARCH=amd64 go build path/to/my/pipeline -o output/worker

$ ./output/launcher --worker_binary=output/worker --runner=...
```

For more, see https://beam.apache.org/documentation/sdks/go-cross-compilation/

Of the SDK languages; Go is the only one that must be statically compiled into an architecture specific executable.

Python can be interpreted, and Java compiles to bytecode executed on the JVM. Both have resources dynamically loaded in at runtime, via modules or JARs.

But for Go, everything the binary needs to run is included in the executable at compilation.
This is great for portability!
No managing downloads of libraries, or preloading them into containers.
That is, assuming you're developing on the same OS and architecture that your pipeline's workers are running on.

Does this mean you can't deploy to the linux based cloud from your M1 Mac?
No. Of course not.
It just means that your code needs to compile twice: once for the platform launching the pipeline, and once for the platform used by the pipeline workers.

Java's "Compile Once, Run Anywhere" this is not.

Go has robust cross compile support for different architectures, making this pretty easy.
Simply set the target architecture, with the GOOS and GOARCH environment variables, and run `go build`.

The SDK will even attempt to do this for you in some cases too, targeting the common Linux + AMD64 platform.
Either way: voila! A Go binary for your target architecture.

To use it, set the worker_binary flag to the executable when starting your pipeline.

The binaries are otherwise multi-modal, and will behave as workers or as your main program as needed.

You can see the go-cross-compilation page on the beam site for more information.

And that’s about it. Other deployment details should be taken care of by the runner in question.

Building Go Pipelines

Next, is how the language affects Pipeline Construction.

This is where Go's previous lack of generics came into play, along with its lack of inheritance, it's lack of function overloading, and it's lack of annotations.

How these intersect with strong typing, first class functions, full type reflection, and multiple return values, lead to this current SDK.

What's in an SDK?

An SDK is how users express the **PTransforms** they want to apply to the **PCollections** in their Pipeline.

For Beam, the SDK is an implementation of the Beam abstract model.

A simple interpretation of this is, is how users express the PTransforms they want to apply on the PCollections in their Pipeline.

The SDK is there to make sure everything makes sense, and that it can execute your DoFns at Pipeline Execution time, while catching potential runtime errors earlier at Pipeline Construction time.

Let's see how that looks for Go.

PTransforms and PCollections

```
p, s := beam.NewPipelineWithRoot()

lines := textio.Read(s, *input)
counted := CountWords(s, lines)
formatted := beam.ParDo(s, formatFn, counted)
textio.Write(s, *output, formatted)

if err := beamx.Run(context.Background(), p); err != nil {
    log.Fatalf("Failed to execute job: %v", err)
}
```

Annotated example at
https://github.com/apache/beam/blob/master/sdks/go/examples/wordcount/wordcount.go

This snippet of code is from the wordcount example.

We get our pipeline object and root scope.

We pass each transform the scope and their input, and they in turn, return a PCollection that can be used down stream.
When the pipeline is constructed, we can pass it to the Run function to be sent off to some runner for execution.

As per Go style, types on variables don't need to be specified manually in many cases, though everything is checked by the compiler.
This makes Go code light weight to change.

Though, you may just need to take my word for it that lines, counted, and formatted are beam.PCollections.

PCollections

- Implicitly typed
- Pipeline typechecked at runtime during Pipeline Construction, like Python
- No Generics, means no explicit pipeline type checking by the compiler

In the Go SDK, PCollections maintain the types from their parent DoFns implicitly so the pipeline can be type checked at construction time.

The lack of Generics meant that we couldn't have the compiler ensure this for us, so much of the front end of the SDK appropriately binds types and verifies that the pipeline fits together.

In practice Go's fast compiles, and the SDKs ever improving error messages makes mistakes easy enough to fix.

PTransforms

```
var s beam.Scope

side := beam.ParDo(s, doFn, inputPCol)

out2 := beam.ParDo(s, doFn2, inputPCol, beam.SideInput{ side })

outA, outB, outC := beam.ParDo3(s, doFn3, out2)

outAB := beam.Flatten(s, outA, outB)

gbk := beam.GroupByKey(s, outC)
```

For more, see https://s.apache.org/beam-go-sdk-design-rfc

On to PTransforms:

Instead of applying Transforms to PCollections, Go's low weight variables are declared and passed down as needed to build a pipeline.

The lack of function overloading, but multiple return parameters means that Go went with a different approach.

To use transforms with multiple output PCollections, we have "arity" specialized ParDo functions, so users can indicate the number of output PCollections.

This allows at least some aspects of pipeline construction to be managed by the compiler, with the construction time checks to clean up afterwards.

Other primitive transforms like flattening and Grouping By Key, follow the same pattern.

Composite PTransforms

```
func CountWords(s beam.Scope, lines beam.PCollection) beam.PCollection
{
    s = s.Scope("CountWords")
    col := beam.ParDo(s,
        &extractFn{SmallWordLength: *smallWordLength},
        lines)
    return stats.Count(s, col)
}
```


Composite PTransforms are also implicit.

Since we can't have an `expand` method on some abstract class, the Go SDK allows for composites using a Scope variable.

Composite PTransforms are nothing more than regular Go functions that happen to take in a scope parameter, the input pcollections or other configuration, along with returning any output PCollections.

The scope name can be defined if desired, and then passed to the subtransforms.

Otherwise there's no special handling from the SDK.
One simply calls the functions as needed to add them to a pipeline.

Writing Go DoFns

Finally, down to the nitty gritty: How did Go's language features affect DoFn authoring?

Considerations in DoFn API Design

An incomplete list

- What type does the DoFn process?
- Mapping Side inputs to the input PCollections
  - Also Access Pattern: iterator vs Map access
- Mapping Emitters to Output PCollections
- Does the DoFn observe Windows?
- Is the DoFn Splittable?
- Is the DoFn Unbounded?
- Does the DoFn use State and Timers?

When figuring out a DoFn API for a language, there are several things that need to be accounted for.
If we don't, we could end up having the SDK being unable to implement more advanced features of the model.

So we need to plan for growth early, even if we aren't implementing them.

Much of this is "How do we tell the Runner and Pipeline Graph what to expect?"

If we don't expose input and output types, we can't verify the PCollections match until it fails at Pipeline Execution time, let alone at Construction time or compile time.
Side inputs need to be mapped to their inputs.
Output PCollections need to be declared somehow.

Access patterns and other things need to be known, so the Runner can take advantage of a DoFn's abilities and optimize its execution.

DoFns - Funcs or Structs

```
// DoFns are either configurable structs with a ProcessElement method

type myDoFn struct{
   Config string
}

func (fn *myDoFn) ProcessElement(...) (...) { ... }

// Or a function for simpler non configurable DoFns.

func simpleDoFn(...) (...) { ... }


func init() {
  beam.RegisterType(reflect.TypeOf((*myDoFn)(nil)))
  beam.RegisterFunc(simpleDoFn)
}
```


So how do we define a DoFn?

Go doesn't have inheritance or Duck typing, so there are no Base classes to extend or override for something to be a DoFn.

Instead, DoFns are structs that have a ProcessElement method.

During pipeline Construction, reflection is used to pull out the method's parameters, to figure out what the DoFn needs as input and provides as outputs so we can correctly type check the pipeline.

But not all DoFns need configuration, or to support the Bundle Lifecycle.

For that, plain functions can be treated as a ProcessElement call by themselves.

Both approaches should be registered with the SDK in an init block, to make sure they can be accessed by at Execution time.

So what do the parameters look like?

Simple Elements

- Passed in as is
- Returned out as is.

```
func prefixFn(s string) string {
    return “http://” + s
}

func extractValue(v *Pair) int {
    return v.Value
}
```


Simple elements are passed in as is, and in the simplest cases, returned out after any transforms.

Key Value pairs needed some special handling.

Key Value Pairs (KVs)

- “Exploded” into 2 parameters or return values

```
// formatFn takes in a KV<string,int> and returns a string
func formatFn(w string, c int) string {
    return fmt.Sprintf("%s: %v", w, c)
}

// unpair takes in a Pair, and returns a KV<string,int>
func unpair(v Pair) (string, int) {
    return v.Key, v.Value
}
```


Go didn’t have Generics, nor does it have “Tuples”.
So the original SDK authors opted to explode out KVs into two parameters.

So to have your DoFn accept KVs, you simply have it take in the key and value types as parameters.
To have your DoFn return them, it’s the same in reverse: return two values!

Emitters (Output PCollections)

```
// formatFn takes in a KV<string,int> and returns a string
func formatFn(w string, c int, emit func(string)) {
    emit(fmt.Sprintf("%s: %v", w, c))
}

// unpair takes in a Pair, and returns a KV<string,int>
func unpair(v Pair, emit func(string, int)) {
    emit(v.Key, v.Value)
}

// Split takes in a Pair, and returns a PCollection for keys and
// values
func Split(v Pair, emitK func(string), emitV func(int)) {
    emit(v.Key)
    emit(v.Value)
}
```


Next up is emitters, for defining output PCollections.

One thing Go does have is first class functions!

This is pretty close to as generic as Go could get previously.

Not all DoFns are 1:1 so having a function as a parameter is handy.
DoFns can emit zero or more times to these emitter functions.
Naturally they also support KVs.

Or even have several emitter functions, which will map to the same order as the returned PCollections by the ParDo functions.

Iterators (Side Inputs & GBKs)

```
func match(v string, iter func(*string) bool,
    emit func(string, string)) {

    var s string
    for iter(&s) {
        if matches(v, s) {
            emit(v, s)
        }
    }
}

func lookup(v string, hasCached func(string) func(*int) bool, emit …)
{}
```


Similarly, we have Iterables for handling Side Inputs and the results of GroupByKeys.

Like emitters, these are functions, with a slightly different signature. They take in the pointer of the type and return a boolean, and reuse a variables's allocation, and can be concisely used in loops.

Map Side inputs let you look values for a given key.

The Go SDK uses a Positional API

In the end, the positions of Side Inputs and Emitters will match the order needed in the beam.ParDo calls in pipeline construction.

We call this the Positional API.

It’s not the only way the API could have looked, but it’s the one we’ve got for now.

Who knows? maybe as Go continues to evolve, we’ll have a different API in the future!

Usable by Go Community

To tie things off. Community.

Integrating the SDK with Language Specific Tools

- Are there versioning or release conventions?
- Standard Packaging?
- Package Managers?
- Dependency Management?
- Environment conventions?
- Documentation?

When it comes to a new SDK, how Beam integrates with that languages tools is also important.

If we don't, then users would be forced to handle things manually or with ad hoc scripts.
This makes it harder to take on dependencies or integrate with other tools.

The Go SDK isn't new, but Go's mechanisms for these have only been prevalent for about a year or three.

For Go, a lot of this is very simple.

We have the go tool, which handles all of this for us, from building, and testing, to the rest.

Releases happen when the SDK is tagged in Github with the new version. That's it.

Documentation is available on pkg.go.dev.

Dependency management happens with Go Modules.

Top Level SDK Requirements

From a user perspective

- In Beam Repository
- Maintained by Beam Community
- Not Experimental
- Implements the Beam Model idiomatically for the language.
- Integrates with the languages tools

As for the Top Level SDK Question, we can modify our requirements a little.

The Beam Model should be implemented idiomatically for the language.
It needs to be familiar to users of that language, even if they aren't familiar with Beam yet.

It also needs to integrate with the languages tools, so they're accessible to the community for that language.

I think that's a pretty reasonable bar.

These days, I think the ability to support Cross Language transforms is also needed, but that's for the community to decide.

Additional Resources

Go Website: go.dev

Effective Go: go.dev/doc/effective_go

Original Beam Go SDK RFC: s.apache.org/beam-go-sdk-design-rfc

SDK Overview: beam.apache.org/documentation/sdks/go/

Beam Programming Guide: https://beam.apache.org/documentation/programming-guide/

SDK Comparison metrics: https://s.apache.org/beam-community-metrics

Just before I finish up…

Thank you!

Thank you very much!

Go as a Top Level SDK

Robert Burke
Senior Gopher Wrangler
Google Cloud, Data Analytics
@lostluck

https://twitter.com/lostluck
lostluck@apache.org
https://github.com/lostluck/
lostluck.dev