

# Understanding Distributed Systems with OpenTelemetry

Adam Johnson

Apache 2.0 Licensed

- Please find a copy of the slides here:  
<https://bit.ly/3wigdFJ>



# Our agenda

- What is observability?
- How does OpenTelemetry relate to observability?
- What concepts do I need to use OpenTelemetry?
- How do I record data using OpenTelemetry?
- Where can I send my data?



# Slides for today

- You will need to refer to material from the slides during the interactive work period.
- Please find a copy of the slides here:  
<https://bit.ly/3wigdFJ>
- You will need GoLang installed (1.16+)
  - And be able to download something from Github and expose a port on your laptop



# Who am I?

- Adam Johnson,  
Lightstep



# Observability Basics



# Why observability?

- Microservices create complex interactions.
- Failures don't exactly repeat.
- Debugging multi-tenancy is painful.
- Monitoring no longer can help us.



# What is observability?

- We need to answer questions about our systems.

*What characteristics did the queries that timed out at 500ms share in common? Service versions? Browser plugins?*

- Instrumentation produces data.
- Querying data answers our questions.



# Telemetry aids observability

- Telemetry data isn't observability itself.
- Instrumentation code is *how* we get telemetry.
- Telemetry data can include traces, logs, and/or metrics.

All different *views* into the same underlying truth.





# Metrics, logs, and traces, oh my!

- Metrics
  - Aggregated summary statistics.
- Logs
  - Detailed debugging information emitted by processes.
- Distributed Tracing
  - Provides insights into the full lifecycles, aka **traces** of requests to a system, allowing you to pinpoint failures and performance issues.

Structured data can be transmuted into any of these!



## But how do I implement these?

- You need an instrumentation framework!
- and a place to send the data!
- and a way to visualize the data!



# About OpenTelemetry

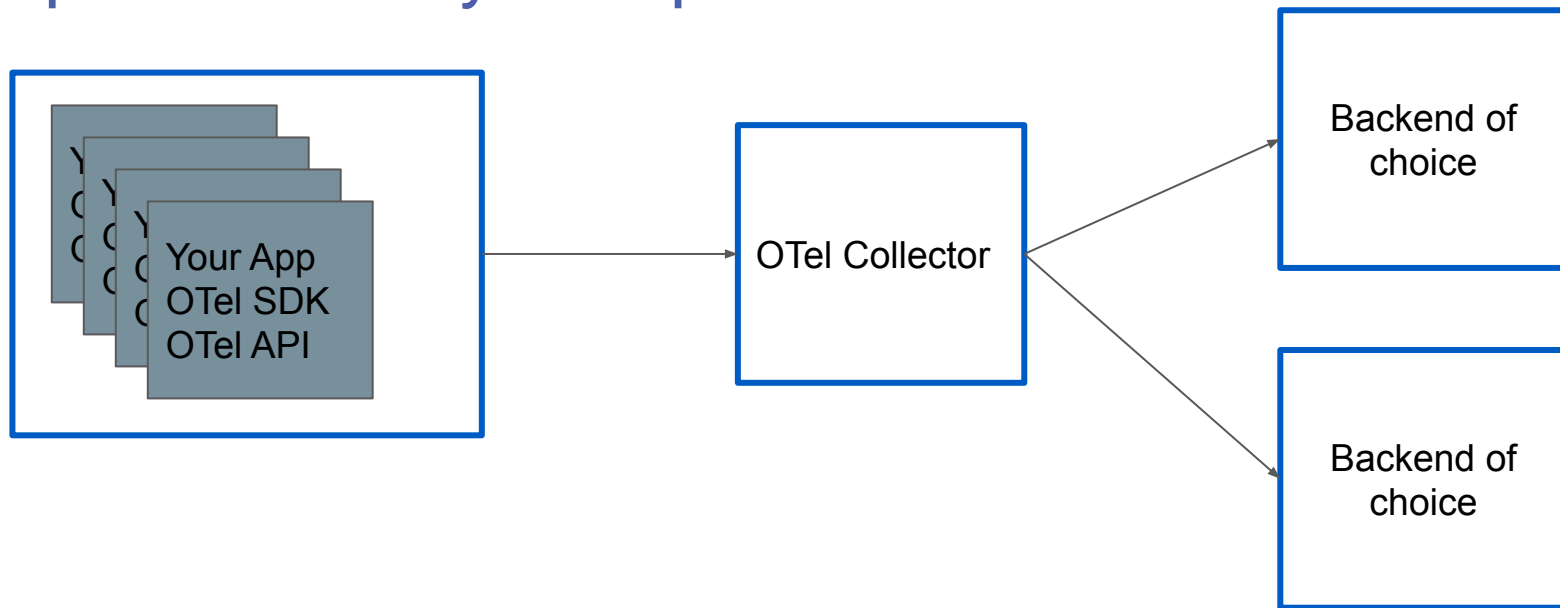


# OpenCensus + OpenTracing = OpenTelemetry

- OpenTracing:
  - Provides APIs and instrumentation for distributed tracing
- OpenCensus:
  - Provides APIs and instrumentation that allow you to collect application metrics and distributed tracing.
- OpenTelemetry:
  - An effort to combine distributed tracing, metrics and logging into a single set of system components and language-specific libraries.



# OpenTelemetry Components



# SDKs, Exporters, and Collector Services, Oh My!

- OpenTelemetry's **SDK** implements trace & span creation.
- An **exporter** can be instantiated to send the data collected by OpenTelemetry to the backend of your choice.
  - E.g. Jaeger, Lightstep, Honeycomb, Stackdriver, etc.
- OpenTelemetry **collector** proxies data between instrumented code and backend service(s). The exporters can be reconfigured without changing instrumented code.



# Vendor-neutral exporters

- Jaeger exporter
  - Jaeger was created at Uber and is now an open-source CNCF project
  - Stores and visualizes traces.
- Prometheus exporter
  - Prometheus is a TSDB inspired by Google's Borgmon
  - Stores time-series metrics. Note: OTel metrics are not finalized.
- stdout/stderr streaming export
  - Inspect what is actually being sent over the wire.
  - No external setup required!



# The OpenTelemetry Collector

- Collectors are useful for more than just proxying traces/metrics.
  - Receive multiple trace formats and marshal them into a new one.
  - Enhance trace/metric data with resources.
  - Perform sampling, filtering, or custom processors to modify attributes.
  - Much more - it's customizable!
- Run them as agents or in standalone mode.
  - A Kubernetes operator exists to aid in deployment.
  - Builds are available for x86/ARM Linux, Darwin, and Windows.





## Microservices

App Code

OTel Auto. Inst.

OTel API

OTel SDK

OTLP

3rd party  
service

OTel  
Collector

Time Series  
Databases

Trace  
Databases

Column  
Stores

Observability  
Frontends  
& APIs

Kubernetes

OTLP

L7 Proxy

OTLP



OTLP

Shared  
Infra

Managed DBs

APIs

Client Instrumentation



# OTel API - packages, methods, & when to call

- Tracer
  - A Tracer is responsible for tracking the currently active span.
- Meter
  - A Meter is responsible for accumulating a collection of statistics.

You can have more than one. Why?

Ensures uniqueness of name prefixes.



# OTel API - Tracer methods, & when to call

- `tracer.Start(ctx, name, options)`
  - This method returns a child of the current span, and makes it current.
- `tracer.SpanFromContext(ctx)`
  - Used to access & add information to the current span



# OTel API - Span methods, & when to call

- `span.AddEvent(ctx, msg)`
  - Adds structured annotations (e.g. "logs") about what is currently happening.
- `span.SetAttribute(label.Key(key).String(value))`
  - Adds a structured, typed attribute to the current span. This may include a user id, a build id, a user-agent, etc.
- `span.End()`
  - Often used with `defer`, fires when the unit of work is complete and the span can be sent



## Code examples: CurrentSpan & Span

- Get the current span
  - `sp := trace.SpanFromContext(ctx)`
- Update the span status
  - `sp.SetStatus(codes.OK)`
- Add events
  - `sp.AddEvent(ctx, "foo")`
- Add attributes
  - `sp.SetAttributes(  
attribute.String("key", "value"))`



# Context Propagation

- Distributed context is an abstract data type that represents collection of entries.
- Each key is associated with exactly one value.
- It is serialized for propagation across process boundaries
- Passing around the context enables related spans to be associated with a single trace.
- W3C TraceContext is the de-facto standard.
  - B3 is more broadly compatible with existing systems.



# Automatic Instrumentation

OpenTelemetry has wrappers around common frameworks to propagate context and make it accessible.

```
import "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"

mux.Handle("/", otelhttp.NewHandler(otelhttp.WithRouteTag("/",
http.HandlerFunc(rootHandler)), "root", otelhttp.WithPublicEndpoint()))

func h(w ResponseWriter, req *Request) {
    ctx := req.Context()
    span := trace.SpanFromContext(ctx)
}
```



# Manual Instrumentation

- Sometimes automatic instrumentation isn't descriptive or good enough
  - Or just doesn't exist!
- For manual instrumentation, there are several core steps that we can follow to create our own instrumentation





# Manual Instrumentation

- You need some common established items when starting your tracing
  - A resource definition - This identifies the resource or machine creating the telemetry
    - Will define standard tags and minimum identifiers
  - An exporter - Somewhere to send the data
    - You can have many exporters
  - A TracerProvider - This unites spans that are made, the resource definition for all spans produced, and the exporter
    - You must register the TracerProvider with the sdk:  
`otel.SetTracerProvider(tracerProvider)`



# Manual Instrumentation

- Finally, just add spans!

```
func (m Model) persist(ctx context.Context) {  
    tr := otel.Tracer("dbHandler")  
    ctx, span := tr.Start(ctx, "database")  
    defer span.End()  
  
    // Persist the model to the database...  
    [...]  
}
```



# Automatic Instrumentation

Metric instrumentation looks a little different

- Often agents are in place to collect metrics automatically
  - Just configure to export to an OTEL collector with a configured Receiver
- OTEL collector can both PUSH and PULL!
  - Works with existing Prometheus libraries



# Manual Instrumentation

Metrics use a similar process as setting up a tracer:

- A resource definition
  - Can use the same definition as your tracer!
- An exporter
- Instead of a TracerProvider, metrics use a MetricController
- As a final step, you register the controller with the SDK:
  - `mglobal.SetMeterProvider(metricController)`
- You can then register metrics to your controller and start measuring!



# Our interactive work today



# Clone the Git repository

- <https://github.com/lightstep/otel-workshop>
  - \$ git clone https://github.com/lightstep/otel-workshop.git
- Ensure you have go installed:

\$ go version

\$ go version go1.17.2 darwin/amd64      //This workshop relies on go 1.16 or newer

\$ cd /download/dir/otel-workshop/src

\$ go mod download



# Our example application

- `mux.Handle("/", http.HandlerFunc(rootHandler))`
  - Prints "Hello, World!"
- `mux.Handle("/favicon.ico", http.NotFoundHandler())`
  - 404s
- `mux.Handle("/fib", http.HandlerFunc(fibHandler))`
  - Returns `/fib?i=n-1 + /fib?i=n-2`



Our job is to instrument this. How?





# Set up modules and initialize SDK

- The SDK initialization will give us the foundations for creating traces and passing context between our services
- When you see the execute icon, you should be able to run your application successfully with ``go run .`` from your `/src` directory



# Add OTel imports

```
import ( ...  
    "Context"  
    "runtime"  
    "syscall"  
    "time"  
    "go.opentelemetry.io/otel"  
    sdktrace "go.opentelemetry.io/otel/sdk/trace"  
    "go.opentelemetry.io/otel/sdk/resource"  
    "go.opentelemetry.io/otel/attribute"  
    semconv "go.opentelemetry.io/otel/semconv/v1.7.0"  
)
```



# Add a resource definition

```
func newResource() *resource.Resource {  
    r, _ := resource.Merge(  
        resource.Default(),  
        resource.NewWithAttributes(  
            semconv.SchemaURL,  
            semconv.ServiceNameKey.String("fib"),  
            semconv.ServiceVersionKey.String("v0.1.0"),  
            attribute.String("environment", "workshop"),  
        ),  
    )  
    return r  
}
```



# Set up SDK

```
func main() { ...  
  
    // Define TracerProvider  
  
    tracerProvider := sdktrace.NewTracerProvider(  
  
        sdktrace.WithSampler(sdktrace.AlwaysSample()),  
  
        sdktrace.WithResource(newResource()),  
  
    )  
  
    // Set TracerProvider  
  
    otel.SetTracerProvider(tracerProvider)
```



# Add trace spans to the logic

- We're not adding any traces to our TracerProvider yet!
- We will instrument our main() and helper functions
- `mux.Handle("/", http.HandlerFunc(rootHandler))`
  - Wrap rootHandler with HTTP plugin
  - Add dbHandler internal span.
- `mux.Handle("/fib", http.HandlerFunc(fibHandler))`
  - Returns `/fib?i=n-1 + /fib?i=n-2`
    - Wrap the handler
    - Add attributes for the parameters
    - Create spans for each parallel client call
    - Propagate the context to downstream calls.



# otelhttp instrumentation of root handler

```
import (  
  
    "context"  
  
    "go.opentelemetry.io/otel/trace"  
  
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"  
  
    //"go.opentelemetry.io/contrib/instrumentation/net/http/httptrace/otelhttptrace"  
  
)  
  
func main() {  
    mux.Handle("/", otelhttp.NewHandler(  
        http.HandlerFunc(rootHandler), "root"))  
  
    func rootHandler(...) {  
        ctx := req.Context()  
        trace.SpanFromContext(ctx).AddEvent(ctx, "Ran root handler.")  
    }  
}
```



# Internal spans & context propagation

```
func rootHandler([...]) {  
    ctx := req.Context()  
  
    trace.SpanFromContext(ctx).AddEvent("annotation within span")  
  
    _ = dbHandler(ctx, "foo")  
}
```

```
func dbHandler(ctx context.Context, color string) int {  
  
    tr := otel.Tracer("dbHandler")  
  
    ctx, span := tr.Start(ctx, "database")  
  
    defer span.End()  
}
```

**\*\*Note that context is being passed as a param! Doing this over the network typically involves Inject/Extract**



## Configure output to stdout

- Now we are creating spans and attaching them to our Tracer
- However, we need to register an exporter that tells the TracerProvider where to send these spans!
- Set up an exporter to stdout that will write our spans to the console (`stdouttrace.New`)





# Configure output to stdout

```
import "go.opentelemetry.io/otel/exporters/stdout/stdouttrace"

func main() {

    // stdout exporter
    std, err := stdouttrace.New(stdouttrace.WithPrettyPrint())
    if err != nil {
        log.Fatal(err)
    }

    // Define TracerProvider
    tracerProvider := sdktrace.NewTracerProvider(
        sdktrace.WithSampler(sdktrace.AlwaysSample()),
        sdktrace.WithBatcher(std),
        sdktrace.WithResource(newResource()),
    )
}
```

We are now initializing err.  
For later references, you  
may need to change the  
assignment operators



# What you should see...

```
{
  "SpanContext": {
    "TraceID": "9850b11fa09d4b5fa4dd48dd37f3683b",
    "SpanID": "1113d149cffffa942",
    "TraceFlags": 1
  },
  "ParentSpanID": "e1e1624830d2378e",
  "SpanKind": "internal",
  "Name": "dbHandler/database",
  "StartTime": "2019-11-03T10:52:56.903919262Z",
  "EndTime": "2019-11-03T10:52:56.903923338Z",
  "Attributes": [],
  "MessageEvents": null,
  "Links": null,
  "Status": 0,
  "HasRemoteParent": false,
  "DroppedAttributeCount": 0,
  "DroppedMessageEventCount": 0,
  "DroppedLinkCount": 0,
  "ChildSpanCount": 0
}
{
  "SpanContext": {
    "TraceID": "9850b11fa09d4b5fa4dd48dd37f3683b",
    "SpanID": "e1e1624830d2378e",
    "TraceFlags": 1
  },
  "ParentSpanID": "ff33261fd1178603",
  "SpanKind": "server",
  "Name": "go.opentelemetry.io/plugin/otelhttp/root",
  "StartTime": "2019-11-03T10:52:56.903886026Z",
  "EndTime": "2019-11-03T10:52:57.330677729Z",
  "Attributes": [
    {
      "Key": "http.host",
      "Value": {
        "Type": "STRING",
        "Value": "opentelemetry-instructor.glitch.me"
      }
    }
  ],
  {

```



# Understanding the output

JSON formatted info, output in order End() was called.

```
"SpanContext": {  
  "TraceID": "9850b11fa09d4b5fa4dd48dd37f3683b",  
  "SpanID": "1113d149cffffa942",  
  "TraceFlags": 1  
},  
"ParentSpanID": "e1e1624830d2378e",  
"SpanKind": "internal",  
"Name": "dbHandler/database",  
"StartTime": "2019-11-03T10:52:56.903919262Z",  
"EndTime": "2019-11-03T10:52:56.903923338Z",  
"Attributes": [],  
"MessageEvents": null,  
"Links": null,  
"Status": 0,  
"HasRemoteParent": false,  
"DroppedAttributeCount": 0,  
"DroppedMessageEventCount": 0,  
"DroppedLinkCount": 0,  
"ChildSpanCount": 0
```



# Attributes & MessageEvents

```
"Attributes": [  
  {  
    "Key": "http.host",  
    "Value": {  
      "Type": "STRING",  
      "Value": "opentelemetry-instructor.glitch.me"  
    }  
  },  
  {  
    "Key": "http.status_code",  
    "Value": {  
      "Type": "INT64",  
      "Value": 200  
    }  
  }  
],  
"MessageEvents": [  
  {  
    "Message": "annotation within span",  
    "Attributes": null,  
    "Time": "2019-11-03T10:52:56.903914029Z"  
  }  
],
```



## Now, let's instrument /fib.

- Let's instrument the meat of our project
- First, we will want to use our auto instrumentation libraries to capture our HTTP tracing:

`"go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"`

- Because we are going to be passing Context between HTTP calls, we will need to make sure the text propagator is declared: `otel.SetTextMapPropagator`
- Within /fib, you will want to start a span early in the code



## Now, let's instrument /fib.

- In our recursive function, you will need to inject your span context to the new request, so subsequent calls can appropriately make connected spans: `otelhttptrace.Inject`
- Try adding attributes to each span that will track what fib number the span is calculating and what the returned result is



# Now, let's instrument /fib.

```
import "go.opentelemetry.io/otel/propagation"

"go.opentelemetry.io/otel/codes"

func main() {
...

otel.SetTextMapPropagator(propagation.NewCompositeTextMapPropagator(propagation.TraceContext{},
propagation.Baggage{}))

mux.Handle("/", otelhttp.NewHandler(otelhttp.WithRouteTag("/", http.HandlerFunc(rootHandler)), "root"))

...

mux.Handle("/fib", otelhttp.NewHandler(otelhttp.WithRouteTag("/fib", http.HandlerFunc(fibHandler)),
"fibonacci"))

...
```



# Now, let's instrument /fib.

```
func fibHandler(w http.ResponseWriter, req *http.Request) {  
  
    ctx := req.Context()  
  
    tr := otel.Tracer("fibHandler")  
  
    var err error  
  
    ...  
  
    if err != nil {  
  
        fmt.Fprintf(w, "Couldn't parse index '%s'.", req.URL.Query()["i"])  
  
        w.WriteHeader(503)  
  
        return  
  
    }
```





## Now, let's instrument /fib.

```
go func(n int) {  
  
    err := func() error {  
  
        ictx, sp := tr.Start(ctx, "fibClient")  
  
        defer sp.End()  
  
        url := fmt.Sprintf("http://127.0.0.1:3000/fibinternal?i=%d", n)  
  
        trace.SpanFromContext(ictx).SetAttributes(attribute.String("url", url))  
  
        trace.SpanFromContext(ictx).AddEvent("Fib loop count",  
trace.WithAttributes(attribute.Int("fib-loop", n)))  
  
        req, _ := http.NewRequestWithContext(ictx, "GET", url, nil)  
  
        ictx, req = otelhttptrace.W3C(ictx, req)  
  
        otelhttptrace.Inject(ictx, req)
```



# Now, let's instrument /fib.

```
resp, err := strconv.Atoi(string(body))

    if err != nil {

        trace.SpanFromContext(ictx).SetStatus(codes.Error, "failure parsing")

        return err

    }

    trace.SpanFromContext(ictx).SetAttributes(attribute.Int("result", resp))

    mtx.Lock()

    defer mtx.Unlock()

    ret += resp

    return err

}
```



# Now, let's instrument /fib.

```
trace.SpanFromContext(ctx).SetAttributes(attribute.Int("result", ret))
```

```
fmt.Fprintf(w, "%d", ret)
```

```
}
```



# Configure context propagation

- We're using HTTP headers to propagate context.
- We need to mark our public endpoint as a trace boundary.
- To solve this, `otelhttp.WithPublicEndpoint()` can be passed to `otelhttp.NewHandler()`

```
mux.Handle("/", otelhttp.NewHandler(otelhttp.WithRouteTag("/", http.HandlerFunc(rootHandler)), "root",  
otelhttp.WithPublicEndpoint()))
```

...

```
mux.Handle("/fib", otelhttp.NewHandler(otelhttp.WithRouteTag("/fib", http.HandlerFunc(fibHandler)),  
"fibonacci", otelhttp.WithPublicEndpoint()))
```



# Go look for your trace!

Look in your console to ensure that traces are being written out.

You should be able to see the parent ID as well as any attributes associated with that trace



Getting data out more usefully...



# Analysing your traces

From our earlier slides:

- Telemetry data isn't observability itself.
- Instrumentation code is *how* we get telemetry.

Let's take a look at our slides in an observability platform



## Some motivating challenges:

- How many times is `/fibinternal?i=2` called when `/fib?i=5` is called?
- Can you find the overhead of DNS compared to the overhead of server HTTP?
- Can you add another parameter to the root HTTP request, and send the value of that parameter as an Attribute to the backend?





# Plugging in your own exporter

- Initialize a custom exporter with an API key
  - examples: Stackdriver, Lightstep, Honeycomb, etc.
  - A current list of vendors working with OpenTelemetry can be found here:  
<https://opentelemetry.io/vendors/>



# Lightstep

- Sign up for Lightstep Community
  - [http://app.lightstep.com/signup/developer?signup\\_source=ato](http://app.lightstep.com/signup/developer?signup_source=ato)
  - We also have a Discord at [tstp.run/discord](https://tstp.run/discord)
- Get your access token and configure OTLP export.
  - Click 'Settings' gear icon, then look for 'Access Tokens' and click the copy icon. Export this value as an environment variable:

```
% export LS_ACCESS_TOKEN=$YOUR_ACCESS_TOKEN
```



# Lightstep

Include these libs

We want to add an exporter. We will then need to register the exporter with the TracerProvider

```
Import{ ...  
  "go.opentelemetry.io/otel/exporters/otlp/otlptrace"  
  "go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracegrpc"  
  "google.golang.org/grpc/credentials"  
  "google.golang.org/grpc/encoding/gzip"  
}
```



# Lightstep

The exporter definition should look like this for an otlp exporter sending to Lightstep

```
Func main() {  
    ..**..  
    //OTLP exporter  
    ls_access_token, _ := os.LookupEnv("LS_ACCESS_TOKEN")  
    exporter, err := otlptrace.New(context.Background(), otlptracegrpc.NewClient(  
        otlptracegrpc.WithTLSCredentials(credentials.NewClientTLSFromCert(nil, "")),  
        otlptracegrpc.WithEndpoint("ingest.lightstep.com:443"),  
        otlptracegrpc.WithHeaders(map[string]string{"lightstep-access-token":ls_access_token}),  
        otlptracegrpc.WithCompressor(gzip.Name),),  
    )  
    if err != nil {  
        log.Fatalf("Could not start web server: %s", err)  
    }  
    ..**..  
}
```



# Lightstep

We have to register our new exporter with our TracerProvider

```
// Define TracerProvider
tracerProvider := sdktrace.NewTracerProvider(
    sdktrace.WithSampler(sdktrace.AlwaysSample()),
    sdktrace.WithBatcher(std),
    sdktrace.WithBatcher(exporter),
    sdktrace.WithResource(newResource()),
)
```



## Lightstep analysis

Adding our new exporter now provides us with a new output! Look in Lightstep and navigate to 'Explorer' with the clock icon on the left

You should see each of your spans listed out. Click on one of these to see how your spans look



## OTLP Metrics - Alpha

Let's try sending metrics now.

First, let's instrument our application to collect metrics

Comment in the func `updateDiskMetrics` and the call in `main()` to run this once



# OTLP Metrics - Alpha

Walking through the function -

First we have our metric declarations:

```
meter := mglobal.Meter("container")
    mem, _ := meter.NewInt64Counter("mem_usage",
        metric.WithDescription("Amount of memory used."),
    )
...
```

We are naming our metrics and defining the type, name, and description for these metrics





# OTLP Metrics - Alpha

Next we are fetching the runtime measurements that our application makes available:

```
var m runtime.MemStats
for {
    runtime.ReadMemStats(&m)

    var stat syscall.Statfs_t
    wd, _ := os.Getwd()
    syscall.Statfs(wd, &stat)

    all := float64(stat.Blocks) * float64(stat.Bsize)
    free := float64(stat.Bfree) * float64(stat.Bsize)
```

The system stats provide us with the data to create more actionable metrics




# OTLP Metrics - Alpha

Finally, we are defining what our metric batch looks like:

```
meter.RecordBatch(ctx, []attribute.KeyValue{
    appKey.String(os.Getenv("PROJECT_DOMAIN")),
    containerKey.String(os.Getenv("HOSTNAME"))},
    used.Measurement(all-free),
    quota.Measurement(all),
    mem.Measurement(int64(m.Sys)),
    goroutines.Measurement(int64(runtime.NumGoroutine()))),
    time.Sleep(time.Minute)
```

Note how we are assigning our metric definitions from part 1 the measurements we took from part 2.

 Also note the time.Sleep - we are taking these measurements once per minute

## OTLP Metrics - Alpha

Now that our metrics are being created and recorded - we need to define an exporter to send them to our system. We can send these to Lightstep as well



# OTLP Metrics - Alpha

Include our new libraries for metrics:

```
"go.opentelemetry.io/otel/exporters/otlp/otlpmetric"  
"go.opentelemetry.io/otel/exporters/otlp/otlpmetric/otlpmetricgrpc"  
"go.opentelemetry.io/otel/metric"  
mglobal "go.opentelemetry.io/otel/metric/global"  
controller "go.opentelemetry.io/otel/sdk/metric/controller/basic"  
processor "go.opentelemetry.io/otel/sdk/metric/processor/basic"  
selector "go.opentelemetry.io/otel/sdk/metric/selector/simple"
```



# OTLP Metrics - Alpha

Let's start with our GRPC client -

```
//Establish metrics client
metricClient := otlpmetricgrpc.NewClient(
    otlpmetricgrpc.WithTLSCredentials(credentials.NewClientTLSFromCert(nil, "")),
    otlpmetricgrpc.WithEndpoint("ingest.lightstep.com:443"),
    otlpmetricgrpc.WithHeaders(map[string]string{"lightstep-access-token":ls_access_token}),
    otlpmetricgrpc.WithCompressor(gzip.Name),
)
```

Just like our tracing set up, we need an exporter with the context and client

```
//Declare metrics exporter
mExporter, err := otlpmetric.New(
    ctx,
    metricClient,
)
```



# OTLP Metrics - Alpha

Instead of a TracerProvider, metrics use a controller that configures the export settings (we are using a push model)

```
//Set metric controller
metricController := controller.New(
    processor.NewFactory(
        selector.NewWithHistogramDistribution(),
        mExporter,
    ),
    controller.WithResource(newResource()),
    controller.WithExporter(mExporter),
    controller.WithCollectPeriod(2*time.Second),
)

mglobal.SetMeterProvider(metricController)
```



# OTLP Metrics - Alpha

The last thing to do is to start the controller to begin pushing our metrics!

```
if err := metricController.Start(ctx); err != nil {  
    log.Fatalf("could not start metric controller: %v", err)  
}  
  
defer func() {  
    ctx, cancel := context.WithTimeout(ctx, time.Second)  
    defer cancel()  
    // pushes any last exports to the receiver  
    if err := metricController.Stop(ctx); err != nil {  
        otel.Handle(err)  
    }  
}()
```



## Need any hints?

- Instructor code is at <https://github.com/lightstep/otel-workshop-go-instructor>
- You can see how we did things there.





Thank you!

