Distributed Tracing:

* Tracing is the logging of system activity, a data dump of all the traces that occurred
* Traces are the end-to-end data for a request, from the webpage until the data is returned
* Traces are made up of spans which can have more child spans
  + Spans are the slices of time that represents the beginning to the end of a request
* Mixer sends events/traces to the trace collector
* Trace collector collects data and the relation between traces
* This data is then displayed on an UI

Traces, Logs, Metrics:

* Trace reveals problem in the traffic flow, such as broken flow or latency
  + However, traces don’t explain the details of the error
  + Specific to single operation
  + Factors to consider when using tracing are the language/libraries supported, production operations and community support
* Logs can explain the cause and details on the errors
* Metrics allows deeper analysis into the system faults

Jaeger vs Zipkin:

* Jaeger has better OpenTracing support and more diversity of OpenTracing-compatible clients for different programming languages
* Jaeger has good language coverage for OpenTracing-compatible clients, low memory footprint, and a modern scalable design
* Jaeger is part of the CNCF, so it is more compatible with Kubernetes
* Both supports SpringBoot and SpringBoot cloud applications
* Recommendation is to evaluate Jaeger first and it Jaeger is not a good fit then Zipkin
* Jaeger has good

Prometheus:

* Monitoring is supported in Istio with Prometheus
* Prometheus supports automated monitoring with Alert and Alert Manager

Spans:

* To implement tracing, the app needs to create spans which are exported to Jaeger to be presented in visualization
* Some of the data in a span includes operation name, start timestamp and finish timestamp
* To implement spans, some code changes may be required, or simply include Jaeger in the dependency
  + Include: io.opentracing.contrib:opentracing-spring-cloud-starter:{version} to import libraries for opentracing
  + Include: io.jaegertracing:jaeger-tracerresolver:{version} to import jaeger tracer into the code
  + Every time a call is made, there will be some header info generated for Jaeger tracer to pick up
  + Each call will have 2 spans, one from the server side and other from the client side

Telemetry:

* Istio automatically gathers telemetry for a service and a new metric and log stream will be enabled for calls to services
* A telemetry.yaml file is used to configure Mixer to automatically generate and report a new metric and a new log stream for all traffics in the mesh
* The new configuration controls 3 main function in Mixer:
  + Generate instances, such as metric values and log entries, from Istio attributes
  + Create handlers, in the form of Mixer adapters, that process the generated instances
  + Dispatch the instances to the handlers according to a set of rules
* Metric configuration details: (instance, handler, rule)
  + Kind: metric stanza:
    - Configuration details on the generation of metric instances
    - Includes the metric name
    - Basing off the attributes reported by Envoy during the call, Mixer creates the metric instance and populate its specs
    - In the specs, the value field indicates the amount of records, aka “values”, for each call.
    - Dimensions includes the unique identity that is being reported by the metric, such as source, destination, response code, message, etc.
    - Can also set default values to dimension which will be applied if no value is provided for that dimension
    - Dimension also makes it easier to search through the logs, can use dimension as requirements in the search
  + Kind: prometheus
    - Configures a handler that handles incoming metrics and translate into Prometheus-formatted values that can be process by Prometheus backend
    - The translation is done in the spec metrics configuration
    - The metric is translated into Prometheus metrics, and the details of the Prometheus metric is specified in the spec metric section
    - Prometheus automatically prepends “istio\_” to all metric names
    - The Prometheus metric also has labels which matches the dimension from the inputted metric
  + Kind: rule
    - Rule is configured to direct Mixer to send the metric instances to the prometheus handler
    - It matches the instances name with the handler, sending all instances with that specific name to the correct handler
    - The rule by default will execute for all requests in the mesh unless the namespace or match clause is specified
* Logs configuration details: (instances, handler, rule)
  + Directs Mixer to send log entries to stdout
  + Kind: logentry:
    - Configuration for generating log entry instances, tells Mixer how to generate the log entries basing off the attributes reported by Envoy
    - In spec, the severity parameter is used to indicate the log level, this value is mapped to supported logging level by the handler
    - Timestamp parameter supplies the time for the log entry
    - The variable parameter allows control over what to include in each log entry
    - Can also set default values or expressions for the variables
  + Kind: stdio:
    - Configures a handler that handles the incoming log entry instances
    - The spec section configures how the handler handle the instances, such as mapping security levels to Prometheus security levels
    - Can also control the output of log entries to be in JSON format
  + Kind: rule:
    - Configures the rule to direct Mixer to send all instances with specific name to the designated handler
    - The match parameter is set to true for this rule so it is executed for all requests in the mesh
    - Match parameter is defaulted to true if not specified, otherwise, it could control which requests to apply the rule on