OT/OC important merge items.

This document lists the members/aspects that are important as part of our merge prototyping - hence items such as logs/annotations or tags/attributes that will be relatively simple to unify, have been left out.

Observe this only covers the API/code side, no governance or other areas.

1. Context/Span in-process propagation - Conclusion: If we really need this we will consider to add, otherwise we stay with the current version. In other languages we may need to add a notion of Context (like OC python).
2. Tags/Baggage - Conclusion: Keep them separate from the SpanContext.
3. SpanContext - Proposal: We can keep the TraceId (32 hex), SpanId (16 hex), TraceOptions (2he,t,.io
4. Jfjm.j..9
5.
    a. x with last bit representing theL,,m recording) like w3c + trace-state. SpanContext not final - allow users maybe to overwrite for lku
    b. J
    c. 9k
    d.
    e. 00u9 5,gg,5hg 5 r hmm,

    some impl3 qn577ementation.
6. Inter-process propagation - Proposal: Have tracer specific http/binary formats on the propagation component, and use these as default for integration (but allow overwrite for specific cases like http calls to Azure Functions). Also provide on the propagation component some standard formats like b3, w3c.
7. MessageEvent - Proposal: explore how to make structured timed events (such as MessageEvents) an extensible API [BD]
8. Span state management - Not Now
9. Packages/API organization - Only API has NoOp or not
10. Sampling - Not Now (happen in the implementation meeting)
11. Separation of core tracing from context - Not Now (happen in the implementation meeting)
12. Async propagation - Not Now (happen in the implementation meeting)

Context/Span in-process propagation

OT: Defines a `ScopeManager` abstraction that both the user and the Tracer use to set/propagate the active `Span` for the current context. This abstraction is also used in other

languages, such as Python, to handle propagation for asynchronous frameworks such as gevent or Tornado.
OC: Uses `io.grpc.Context` directly as an explicit context layer to set/propagate the active Span, as well as the currently active Tag context. Their supported languages/platform seem to support thread-local scenarios only.

Feedback from ElasticAPM:

> They have an existing context propagation mechanism in place, which doesn't implement the OT interfaces, and hence bridges are built. This propagation mechanism is leveraged for their auto instrumentation mode too, and it co-exists with the manual mode ("mix and match mode" is called as shown here).
>
> If Scope objects are created by an in-place propagation mechanism, they would be detached from the ones the auto agent creates.
>
> Implementation wise, they use thread-local data, but they don't use an implicit stack of `Scope` objects, but an explicit one (it keeps all the previously active `Scope` objects).

Feedback from OT Python:

> The `ScopeManager` abstraction has been helpful to add/handle in-process propagation in frameworks for which thread-local data is not enough (gevent, asyncio, tornado, etc). There is even a set of built-in instances for each of these frameworks, and they can be used along any Tracer.

Possible approaches

1. **Keep the ScopeManager abstraction and let the Tracer (implicitly) and the user (explicitly) make use of it to propagate both Span and TagContext objects,** providing a default implementation on top of thread-local storage or io.grpc.Context.
    a. **Advantages**: Possible to easily exchange storage/logic for a given Tracer instance (such as when using a ref-counting manager for continuation-based frameworks, such as Akka/Play); easy integration for existing codebases (useful for APM agents).
    b. **Disadvantages**: To have an extra abstraction that could confuse users, or end up to be essentially useless (or not needed at least).
2. **Use io.grpc.Context to propagate Span/Tag, and let users override the storage if needed** (https://grpc.io/grpc-java/javadoc/io/grpc/Context.Storage.html). The Tracer interface/class would provide a small set of hooks for letting for allowing extra/custom logic.
    a. **Advantages**: A simple, well defined, unique way to do in-process propagation.

b. **Disadvantages**: Limited override/customization of functionality; not obvious way to override the storage (need to have a specific class in the classpath providing such storage override); tricky to properly integrate with existing in-process propagation mechanism (old APM code bases).

Tags/Baggage

OT: Tied to the Span, and part of SpanContext (which is immutable).

OC: Exist independently, can be propagated without a Span.

Possible approaches:

1. **Have Baggage/Tags decoupled from the Span/SpanContext and have it stand independently** (NOTE: this seems to be the path we will take)**.**
    a. **Advantage**: Simplicity, decoupling.
    b. **Disadvantage**: Would need to provide a backwards layer for OT-compliant Tracers using the current approach.
2. **Have Baggage/Tags moved to SpanContext**.

SpanContext

OT: SpanContext exposes Trace Identifiers as strings, hiding the native representation.

OC: SpanContext exposes the underneath Trace Identifiers implementation, as well as TraceState and sampling information.

Feedback from ElasticAPM:

> Span contains a lot of implementation (details), and SpanContext contains Tracestate. Could this be (or be allowed to be) a custom implementation?

Possible approaches:

1. **Have SpanContext abstracting the underneath implementation of the Trace Identifiers and related members** (https://github.com/bogdandrutu/opencensus-java/pull/3)
    a. **Advantages**: Let existing Tracer implementations have/handle their own Tracer Identifiers implementation.
    b. **Disadvantages**: To have each Tracer implementation re-do a lot of Trace Identifiers logic.

2. **Have SpanContext expose Trace Identifiers and other members that are covered by the W3 context effor**t.
    a. **Advantages**: Reuse of the ongoing Trace Identifiers effort.
    b. **Disadvantages**: Can become tricky for existing codebases that do not plan to support W3 Trace Identifiers in the near future, and/or would like to keep their own implementation.

Inter-process propagation

OT: Defines a Format and a carrier for injecting/extracting a given `SpanContext`. The defined formats are TEXT_MAP, HTTP_HEADERS, and a simple BINARY (using `ByteBuffer`), and the implementation if left to the Trace implementers.

OC: Defines support for the specific W3C Tracing Format and a specific Binary (using byte arrays) format implementation.

Possible approaches: TBD

MessageEvent (important for now?)

OT: Nothing like this exists.

OC: Defines this additional API.

Possible approaches:

1. **Implement this is a specialized form of log/annotation** (NOTE: Bogdan doesn't like this. Might need to do further research of how this is handled underneath).
2. **Keep this as an additional/separated API, but keep it abstract**, so each Tracer implementation can choose whatever way to handle they want.
3. **Do not expose it at the public level**, but keep it as a OC specific feature.

Span state management (RESOLVED)

OT: Nothing is done, as OT only defines a set of interfaces, and each Tracer implementation is left to decide/handle these details.

OC: The `impl_core` artifact defines a `EventQueue` interface (source) that handles when/how a Span is passed to the exporter layer upon being finished. As an example of a basic implementation, there is a `SimpleEventQueue` (source), which simply passes the finished `Span` right away to the exporter.

Possible approaches:

1. Create a custom EventQueue, if/as needed, to have specialized handling, tailored to each vendor's needs.

Resolution: Will not be iterated now (as part the initial prototyping).

Packages/API organization

OT: Has **api**, **mock**, **noop** and **util**. Essentially utility classes, and no implementation bit.

OC: Has **api** (along with basic components, such as thread-local based in-process propagation layer), **spi** (which has the exporters api), **impl_core** (the base implementation), **impl** (the full implementation, based on impl_core), **impl_lite** (lite implementation, based on impl_core), and **exporters** (the actual export implementations).

Feedback from Elastic APM:

> There's a lot of value in integrating with OT/OC but it would be better for them to not commit to an API and use it natively.
>
> The core API seems to be still in flux and committing to it would be a risk for them. It is easier/cleaner to implement bridges to external APIs, and have full control of their own internal representation.

Possible approaches:

1. Have the standard be defined as part of the **OC api** package **only**, and have the rest as optional/reference implementation bits (including not touching the exporters package). Extra: possibly have in-process propagation as a separated package, as Ben Sigelman mentioned in the past.

Additional: Having a no-op implementation in place?

Sampling (RESOLVED)

Concern: OpenCensus does not expose control over the sampler to exporters. Datadog requires all traces to be sampled and sent to the datadog agent (where the sampling decision takes place).

Who raised it: Andrew Kent (Datadog)

Alternate proposal: To be defined.

Resolution: Will not be iterated now (as part of the initial prototyping).

Separation of core tracing from context (RESOLVED)

Concern: Core trace concepts (span, exporter, sampler) live in the same artifact as context management (scope, scope manager). Separating the two would simplify the api and implementation.

Who raised it: Andrew Kent (Datadog)

Resolution: Will not be iterated now (as part of the initial prototyping).

Async Propagation (RESOLVED)

Concern: How are we expected to propagate context for async traces? Must we always keep a span open?

Who raised it: Andrew Kent (Datadog)

Resolution: Will not be iterated now (as part of the initial prototyping).