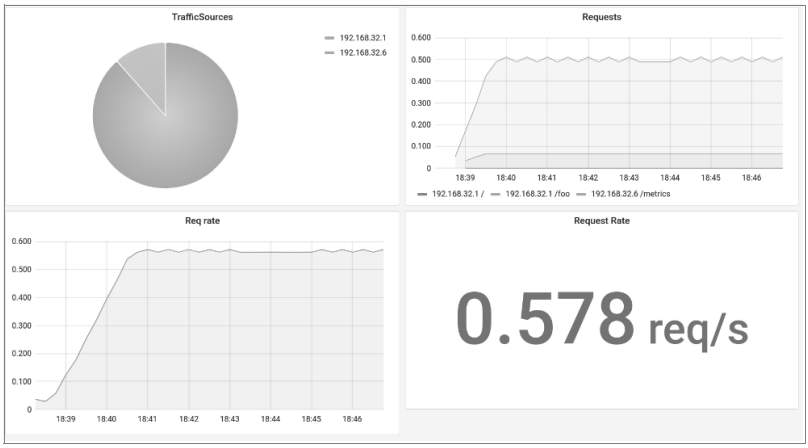

Abb. 19–2
Beispielvisualisierung der gesammelten Metriken mit Grafana

19.5.2 Tracing

Tracing können wir als eine spezielle Form des Loggings ansehen. Dabei werden alle auftretenden Events, die zu einem Businessobjekt gehören, durch eine eindeutige ID gekennzeichnet, sodass wir diese später korrelieren können. Da wir aber mit einer Microservice-Architektur arbeiten, ist es gut möglich, dass die einzelnen Services auf unterschiedlichen Servern laufen. Das macht uns die konsistente Aggregation von Log-Einträgen schwierig. Verteiltes Tracing ist hier ein Lösungsansatz. Dabei werden *Traces* an einen logisch zentralen Service von den einzelnen Services gesendet. Das hat Vor- und Nachteile gegenüber Alternativen wie Aggregation von Logdaten in Systemen wie dem ELK-Stack. Während generell aufgrund des vergleichsweise großen Aufwands geringere Performance erreicht wird, so werden andere Probleme beseitigt: Wir erhalten einen konsistenten Blick auf die Verarbeitung über verschiedene Services hinweg, ohne dass wir uns über Probleme wie abweichende Systemzeiten oder Ähnliches Gedanken machen müssen. Wir verwenden im Folgenden OpenTelemetry, weswegen wir auch deren Terminologie ab hier verwenden. Wie bereits erwähnt gibt es das Konzept des Trace, das den Weg der Verarbeitung verfolgt. Ein Trace besteht aus einem Baum aus *Span*s (meist Codebereiche), wobei ein einzelner Span einem beliebig großen Block an Arbeit entspricht. In unserem Beispiel kann je ein Trace einer Kontaktanfrage zugeordnet werden. Der Wurzelknoten des Trace ist der erste Span, hier also das Ankommen im Webservice. Kindknoten sind damit Spans auf den beiden anderen Microservices. Jeder Span kann beliebig viele Kindknoten haben. So könnte der Service `database-sink` das Speichern in die Datenbank als eigenen Span realisieren. Für unsere Microservices sieht das wie folgt aus:

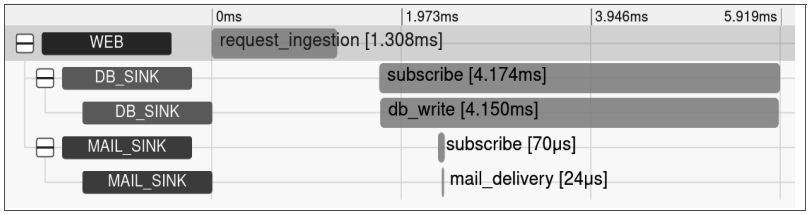

Abb. 19–3
Visualisierung der gesammelten Trace-Daten mit Zipkin

Wir benutzen, um die Verwendung von Tracing exemplarisch zu demonstrieren, die Implementierung mit dem Namen *Zipkin* (eine Alternative wäre *Jaeger*). Die obige Visualisierung ist ein Beispiel für die Möglichkeiten dieser Software. Sie können diese wie gewohnt über einen Docker-Container in unserem Compose-File mit folgendem Befehl starten:

```
docker-compose up zipkin
```

Hintergrund

Das verwendete Crate `opentelemetry-rust` verwendet hinter den Kulissen globalen Zustand. Während das die Interaktion mit manchen Teilen der API stark vereinfacht, werden Sie doch feststellen, dass bei einem solchen Programmiermodell sehr schnell eine Aktion vergessen werden kann.

Die OpenTelemetry-API für Tracing funktioniert unabhängig von der Technologie für das Tracing selbst. Die API ist so gestaltet, dass wir an jeder Stelle unseres Programms Zugriff auf den aktuell aktiven Span haben. Diesem können wir beliebige Events hinzufügen, die Vorgängen in unserem Programm entsprechen, vergleichbar mit einzelnen Log-Einträgen. Der Zugriff auf das aktuelle Span verlangt nicht, dass wir eine Variable durch Ihre Methoden mit hindurchschleifen.

Tipps und Tricks

Bevor Sie weiterlesen, überlegen Sie sich, wie Sie selbst diese Problemstellung lösen würden. Ihr Code müsste die folgenden Punkte erfüllen:

- globale Funktionen zum Beginnen und Beenden eines Spans
- Zustand, der sicherstellt, dass immer nur ein Span aktiv ist
- Hinzufügen von Events zum aktuellen Span, ohne eine Referenz darauf vorzuhalten

Mit den bisher vermittelten Kenntnissen aus diesem Buch können Sie Code schreiben, der diese Anforderungen erfüllt. Bedenken Sie auch, dass sich mehrere Threads gleichzeitig in mehreren Spans befinden können.

Um OpenTelemetry mit Rust und Zipkin verwenden zu können, müssen wir die beiden Komponenten konfigurieren. Genauer gesagt, müssen wir OpenTelemetry derart konfigurieren, dass es das Zipkin-Backend verwendet. Für Zipkin müssen Einstellungen, wie die URL der Zipkin-Instanz, gesetzt werden. Diese Konfiguration nehmen wir global vor und zeigen Sie exemplarisch.

Listing 19–23
Erzeugen des globalen Tracers

```
global::set_text_map_propagator(opentelemetry_zipkin::Propaga-
tor::new());

let _tracer = opentelemetry_zipkin::new_pipeline()
    .with_service_name("web")
    .with_collector_endpoint("http://zipkin:9411/api/v2/spans")
    .install_simple()
    .expect("Could not install tracer");
```

Stöbern wir durch den Quellcode der Bibliothek, stellen wir fest, dass im Zuge von `install_simple` ein `tracer_provider` hinter einem `RwLock` gesetzt wird. Dieses `RwLock` befindet sich in einer globalen Variablen, realisiert mit `lazy_static`. Dadurch kann unser konfigurierter Tracer von jeder Stelle aus ihrem Code heraus verwendet werden. Hierfür gibt es zwei grundlegende Möglichkeiten: das Markieren eines Start- bzw. Endpunkts eines Spans von Hand oder durch eine mit dem Crate gelieferte Funktion. Wir betrachten die beiden Ansätze nachfolgend getrennt. Die folgenden Codestücke sind erweiterte Versionen des Codes zu Beginn des Kapitels. Zuerst schauen wir auf das Codestück im Request-Handler. Hier starten wir einen neuen Span, da diese Stelle den Eintritt eines Requests in unser System darstellt. Wir geben die `trace_id` aus, sodass wir diese später manuell vergleichen können.

Listing 19–24
Starten eines Spans und Registrieren eines Events darin

```
let mut span = global::tracer("Handler")
    .start("request_ingestion");
debug!("Handler with tid: {}",
span.span_context().trace_id().to_hex());

span.add_event("Handler call".to_string(), vec![
    KeyValue::new("Remote",Value::from(request.email.clone()))
]);
```

In der Variablen `span` halten wir eine Referenz auf den aktuellen Span vor, die wir später verwenden können, um Events hinzuzufügen.

Alternativ können wir auch folgende Mechanik verwenden:

Listing 19–25
Zugriff auf den Thread-eigenen Context zum Setzen eines Spans

```
let _span = trace::mark_span_as_active(span);
…
Context::current().span()
    .add_event("Payload parsed".to_string(),vec![]);
```

Hierbei setzen wir `span` als aktiv. Bis wir einen neuen Span als aktiv setzen oder den aktuellen Span manuell beenden, bleibt dieser aktiv. Über den aktuellen Kontext `Context::current()` können wir den aktuellen Span abrufen. Wir sollten dabei im Hinterkopf behalten, dass `Context` pro Thread existiert. Das hat den Vorteil, dass wir in mehreren Threads auch gleichzeitig mehrere Spans aktiv halten können. Wenn das Business-Objekt beispielsweise per `mpsc` den aktiven Thread wechselt, wird nicht automatisch der aktive Span übertragen. Da wir das mit asynchronen Tasks nur sehr umständlich realisieren können, bietet OpenTelemetry hier eine Hilfsfunktion für Futures.

```
publish_to_kafka(request, producer)
        .with_context(Context::current_with_span(span))
        .await
```

Listing 19–26
Verwendung der Funktion with_context(), um die Ausführung des asynchronen Tasks zu verfolgen

Wir müssen glücklicherweise in der Funktion `publish_to_kafka` nichts anpassen, sondern können direkt `Context::current()` verwenden, um den korrekten Span zu erhalten.

Ist ein Span aktiv, wird automatisch beim Erstellen eines neuen Spans dieses als Kindknoten im Span-Baum eingetragen. Da hier das korrekte Starten und Stoppen von Spans kompliziert werden kann, bietet OpenTelemetry auch dafür eine Hilfsfunktion an. Betrachten wir das nächste Beispiel aus dem Mailer-Service:

```
let span = …
let _span = trace::mark_span_as_active(span);
match …message…{
    Ok(message) =>  tracer.in_span(
        "mail_delivery",
        |_cx|  self.send_mail(message)),
    Err(error) => …
}
```

Listing 19–27
Senden der E-Mail mit Tracing

Hier starten wir mit `in_span` automatisch für die Funktion `send_mail` einen Span, setzen ihn als aktiv und lassen ihn am Ende der Funktion automatisch beenden. Die Verwendung der Hilfsfunktionen verhindert hier natürlich auch, dass wir vergessen können, den Span zu beenden. Sollten wir nicht die Hilfsfunktion verwenden, so wird unser Span beendet, sobald die Referenz darauf zerstört wird, beispielsweise durch `drop()` oder weil die Referenz alleine den Gültigkeitsbereich verlässt. Das ist beim vorherigen Beispiel mit `_span` der Fall.

Die einzige Herausforderung, die uns noch bleibt, um verteiltes Tracing umzusetzen, ist eine Möglichkeit, einen Span auf einem System zu starten, der ein Kindknoten eines Span-Baumes ist, welcher wiederum auf einem anderen System gestartet wird. Wir benötigen dieses Konstrukt, wenn wir einen Trace einer Kontaktanfrage im Web-

service gestartet haben, diese Anfrage im E-Mail-Service bearbeiten und der Trace selbstverständlich der gleiche bleiben soll. In unserem Beispielprojekt lösen wir das Problem, indem wir die Trace-ID in die Header-Daten der einzelnen Kafka-Nachrichten hinzufügen. Da Metainformationen wie die Trace-ID von der Fachlogik unabhängig sind, haben wir uns dagegen entschieden, diese Information in die Nachricht selbst mitaufzunehmen. Schauen wir uns dieses Verhalten am Beispiel einer dafür zuständigen Funktion aus dem Mailer-Service an:

```
let sc = serde_json::from_slice::<SpanContext>(header)
…
let tracer =  global::tracer("mail_handler");
let mut span_builder = tracer.span_builder("subscribe");
span_builder.trace_id = Some(sc.trace_id());
let mut span = tracer.build(span_builder);
return Some(span);
```

Wir geben `span` aus der Funktion zurück, um zu vermeiden, dass der aktive Span direkt wieder geschlossen wird. Hier gestaltet sich das Handling des Fehlerfalls besonders einfach: Bekommt die aufrufende Funktion `None` zurück, so kann diese einfach einen neuen Root-Span starten. Das kann vorkommen, wenn eine Kontaktanfrage manuell in Kafka geschrieben wurde.

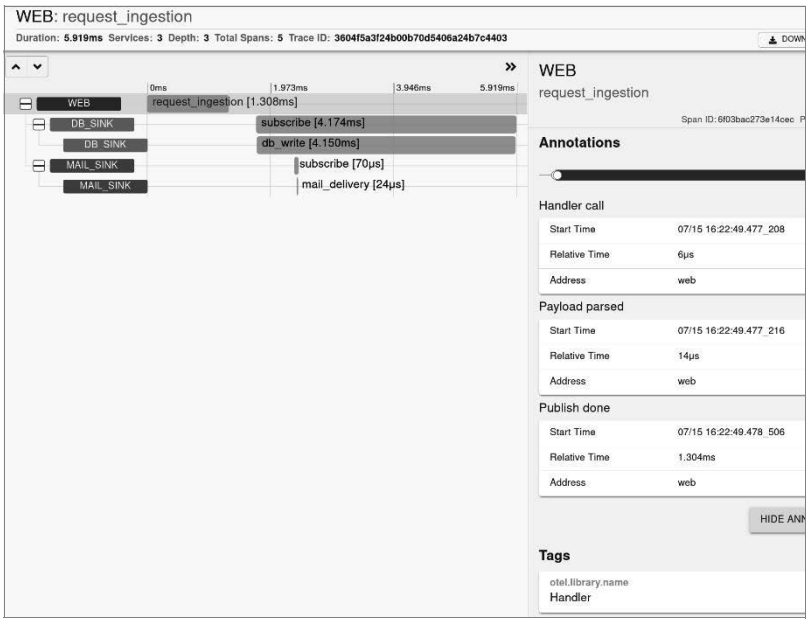

Abb. 19–4
Extrahieren des SpanContext aus dem Header