Tracing

Servisno orijentisane arhitekture

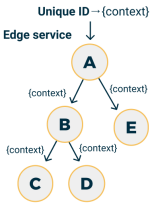
Univerzitet u Novom Sadu
Fakultet tehničkih nauka

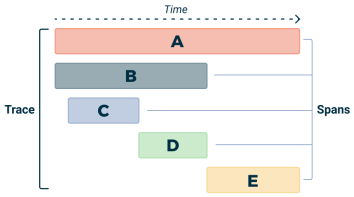
Problem

- U aplikacijam koje su distribuirane po svojoj prirodi, vrlo je bitno da znamo šta se desilo tokom svakom korisničkog zahteva
- Tradicionalne aplikacije jednostavnije je nadgledati jer se stanje nalazi na jednom mestu i direktno iz prikupljenih logova možemo rekonstruisati kada se i zašto nešto desilo
- Ovakav posao znatno je teže obaviti kod distribuiranih aplikacija, jer nije jednostavno utvrditi da li se događaj desio pre ili posle nekog drugog

Tracing

- **Tracing** je metoda koja omogućava nadgledanje distribuirane infrastrukture
- Svaki **trace** predstavlja jedan graf poziva, iniciran korisničkim zahtevom
- Sastoji se iz jednog ili više **span**-ova, svaki span predstavlja jedno parče koda (na primer funkcija) na čijem početku i kraju treba naznačiti da je span počeo, odnosno završio se





Registrovanje Exporter-a

```
func newExporter(address string) (*jaeger.Exporter, error) {
        exp, err := jaeger.New(
                jaeger.WithCollectorEndpoint(
                        jaeger.WithEndpoint(address)))
        if err != nil {
                return nil, err
        }
        return exp, nil
}
```

Registrovanje Tracer Provider-a

```
func newTraceProvider(exp sdktrace.SpanExporter) *sdktrace.TracerProvider {
        r, err := resource.Merge(
                resource.Default(),
                resource.NewWithAttributes(
                        semconv.SchemaURL,
                        semconv.ServiceNameKey.String("ordering-service"),
                ),
        )

        if err != nil {
                panic(err)
        }

        return sdktrace.NewTracerProvider(
                sdktrace.WithBatcher(exp),
                sdktrace.WithResource(r),
        )
}
```

Registrovanje Tracer-a

```
func main() {
        ...
        exp, err := newExporter(cfg.JaegerAddress)
        if err != nil {
                log.Fatalf("failed to initialize exporter: %v", err)
        }
        tp := newTraceProvider(exp)
        defer func() { _ = tp.Shutdown(ctx) }()
        otel.SetTracerProvider(tp)
        tracer := tp.Tracer("ordering-service")
        otel.SetTextMapPropagator(propagation.TraceContext{})
        ...
}
```

Kreiranje span-ova

- Start metoda kreira novi span
- Ako prosleđena context promenljiva sadrži informacije o postojećem span-u, novi span će biti njegovo dete, u suprotnom kreira se korenski span

```
func (h OrderHandler) GetOrder(w http.ResponseWriter, r *http.Request) {
        ctx, span := h.Tracer.Start(r.Context(), "OrderHandler.GetOrder")
        defer span.End()
        ...
}
```

Injektovanje span-a u HTTP zahtev

- Kada jedan servis treba da kontaktira drugi, potrebno je da prosledi informacije o span-u kako bismo ispravno formirali graf poziva
- Dodajemo header-e koje će drugi servis moći da pročita i na osnovu njih kreira novi span

```
...
req, err := http.NewRequest(http.MethodGet, getProductUrl, nil)
if err != nil {
        span.RecordError(err)
}

otel.GetTextMapPropagator().Inject(ctx, propagation.HeaderCarrier(req.Header))

res, err := http.DefaultClient.Do(req)
...
```

Ekstrakcija span-a iz HTTP zahteva

- Kada servisu pristigle zahtev, potrebno je da iz zahteva izvuče span, ukoliko je on prosleđen
- Registrujemo middleware koji će u kontekst zahteva dodati informacije o span-u, kako bismo ga kasnije u handler-u mogli pročitati

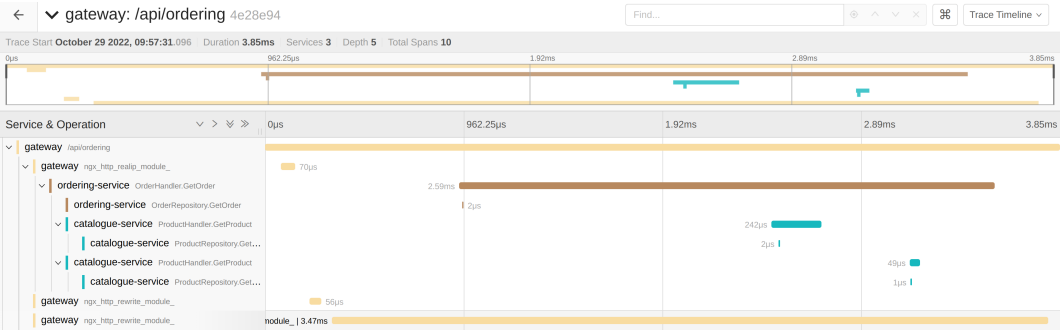
```
func ExtractTraceInfoMiddleware(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
                ctx := otel.GetTextMapPropagator().Extract(
                        r.Context(),
                        propagation.HeaderCarrier(r.Header),
                        )
                next.ServeHTTP(w, r.WithContext(ctx))
        })
}
```

NGINX konfiguracija

- NGINX mora biti proširen dinamičkim modulom koji će da generiše span-ove, šalje ih kolektoru, koji ih zatim šalje Jeager-u
- Za to su nam potrebna dva nova konfiguraciona fajla:
  - opentelemetry_module.conf - podešava OpenTelemetry modul NGINX-a
  - otel-collector-config.yaml - konfiguracija kolektora u kojoj navodimo da prikupljene span-ove šaljemo Jeager-u
- Pored novih fajlova, potrebno je izmeniti i postojeći Dockerfile NGINX-a, tako da sada učitava OpenTelemetry modul
- U docker-compose fajl treba dodati kolektor servis i proslediti mu odgovarajuću konfiguraciju

Vizuelizacija

- Primer jednog trace-a koji se formira kada pošaljemo zahtev za dobavljanje porudžbine



Injektovanje span-a u gRPC zahtev

- Kada kontaktiramo gRPC servis, treba da registrujemo interceptor koji će da obavi injekciju span informacija u zahtev

```
conn, err := grpc.DialContext(
        context.Background(),
        cfg.GreeterServiceAddress,
        grpc.WithBlock(),
        grpc.WithTransportCredentials(insecure.NewCredentials()),
        grpc.WithUnaryInterceptor(otelgrpc.UnaryClientInterceptor()),
)
if err != nil {
        log.Fatalln("Failed to dial server:", err)
}
client := greeter.NewGreeterServiceClient(conn)
```

Ekstrakcija span-a iz gRPC zahteva

- Kada servis prihvati zahtev, treba da ekstrahuje span informacije i doda ih u kontekst

```
grpcServer := grpc.NewServer(
        grpc.UnaryInterceptor(otelgrpc.UnaryServerInterceptor()),
)
```