Relazione Assignment #6 - Observable Microservices

Giacomo Totaro
giacomo.totaro2@studio.unibo.it

Novembre 2024

1 Struttura del progetto

L'obiettivo di questo assignment è applicare alcuni pattern per creare microservizi pronti per la produzione utilizzando le tecnologie discusse in aula. Questo compito si concentra sull'estensione dell'esempio di Cooperative Pixel Art. Si è esteso applicando i seguenti pattern:

- **API Health Check**
- **Distributed Logs**
- **Application Metrics**
- **Distributed Tracing**

Inoltre, si è dovuto pensare a metriche che possono essere utilizzate per definire gli scenari di attributi di qualità.

1.1 Distributed Logs

Il microservizio per i Distributed Logs nel progetto implementa una gestione centralizzata dei log generati da ciascun servizio. Con l'invio centralizzato, è possibile monitorare l'attività di ogni componente in modo efficace e diagnosticare eventuali malfunzionamenti. La soluzione adottata prevede l'implementazione di due **API RESTful**: una **POST** per registrare nuovi log e una **GET** per consentire la visualizzazione dei log salvati.
La logica principale di invio dei log a questo microservizio si trova nel metodo *sendLogRequest*. Questo metodo prepara una richiesta JSON con il messaggio di log, imposta l'header appropriato per i contenuti in JSON e invia i dati alla rotta */api/logs*. Utilizzando *Promise¡Void¿*, gestisce la chiamata asincrona, completando la Future solo dopo la risposta positiva o fallimento. Questo garantisce che il flusso resti reattivo e consenta l'invio di log senza bloccare il flusso del programma principale.

```
private Future<Void> sendLogRequest(String messageLog) {
    Promise<Void> p = Promise.promise();

    JsonObject logData = new JsonObject().put("message",
       messageLog);
    client
        .request(HttpMethod.POST, 9003, "localhost", "/api/
           log")
        .onSuccess(request -> {
            // Imposta l'header content-type
            request.putHeader("content-type", "application/
               json");

            // Converti l'oggetto JSON in una stringa e
               invialo come corpo della richiesta
            String payload = logData.encodePrettily();
            request.putHeader("content-length", "" + payload
               .length());
            request.write(payload);
            request.response().onSuccess(resp -> {
                p.complete();
            });

            System.out.println("[Log] Received response with
                status code " + request.getURI());
            // Invia la richiesta
            request.end();
        })
        .onFailure(f -> {
            p.fail(f.getMessage());
        });
    return p.future();
}
```

Il microservizio presenta due API: la prima è l'endpoint **POST /api/logs**, che riceve messaggi di log, li salva in memoria, e risponde con una conferma di avvenuta ricezione. L'API **GET /api/logs** è stata invece progettata per restituire tutti i log accumulati in formato JSON, semplificando così l'accesso ai log e il loro monitoraggio in modo centralizzato.
Il risultato finale accedendo a *'http://localhost:9003/api/logs'* sarà un elenco di tutti i log prodotti da ciascun microservizio. Di seguito viene riportato una parte dei log prodotti:

```
{
  "logs" : [ {
    "timestamp" : 1729864950364,
    "content" : "[API Gateway] - Terminated moveBrushTo!"
  }, {
    "timestamp" : 1729864950364,
    "content" : "[API Gateway] - Terminated SelectPixel!"
  }, {
    "timestamp" : 1729864950364,
    "content" : "[API Gateway] - Terminated moveBrushTo!"
  }, {
    "timestamp" : 1729864950364,
    "content" : "[API Gateway] - Terminated SelectPixel!"
  }, {
    "timestamp" : 1729864950364,
    "content" : "[API Gateway] - Terminated moveBrushTo!"
  }, {
    "timestamp" : 1729864950364,
    "content" : "[API Gateway] - Terminated SelectPixel!"
  } ]
}
```


Figure 1: Distributed logs on the browser

1.2 API Health Check

L'**API Health Check** è un componente critico in architetture moderne e microservizi distribuiti, progettata per verificare lo stato di salute e la disponibilità operativa dei servizi. Implementare un meccanismo di questo tipo nei sistemi complessi, consente di assicurare che i vari componeneti siano sempre attivi e reattivi. Il monitoraggio dello stato di salute è un passo essenziale per garantire la resilienza, l'affidabilità e la gestione proattiva delle applicazioni. Mediante questa API, gli sviluppatori e i sistemi di monitoraggio possono ottenere in tempo reale un feedback sullo stato dell'applicazione, semplificando l'identificazione di eventuale malfunzionamenti e facilitando interventi tempestivi per evitare disservizi.

Nel progetto *Cooperative Pixel Art*, l'API di health check è stata implementata sia nel microservizio principale, *CooperativePixelArtService*, sia nell'*API Gateway*. Questo permette di monitorare le operazioni interne e le connessioni tra i microservizi. L'endpoint */health* è stato configurato per eseguire un controllo approfondito di tutte le operazioni e delle loro dipendenze, valutando l'accessibilità e lo stato di ogni componente essenziale. Quando l'API viene invocata, effettua una serie di controlli sulle operazioni chiave per verificare che siano operative e in grado di rispondere correttamente. Se tutte le operazioni risultano attive e i controlli di integrità passano, l'API restituisce uno stato **UP**, indicando un funzionamento ottimale. Viceversa, se una o più operazioni non rispon-

dono correttamente o presentano errori, l'API segnala uno stato **DOWN**, allertando gli amministratori e i sistemi di monitoraggio della necessità di un intervento.
Tra i principali vantaggi dell'implementazione di un'API di health check vi è la capacità di rilevare e isolare rapidamente i problemi, riducendo i tempi di inattività del sistema. Questo è particolarmente importante in architetture distribuite dove un singolo componente non funzionante potrebbe compromettere l'intera catena di servizi. Inoltre, l'API contribuisce a migliorare l'affidabilità complessiva del sistema grazie a controlli costanti e a una gestione proattiva delle problematiche.
Il formato seguito per il checking dello stato dei vari microservizi è quello basato su **microprofiling** con seguente formato **JSON**:

Figure 2: Body of health API

1.3 Application Metrics

Nel contest dell'applicazione 'Cooperative Pixel Art', l'integrazione di **application metrics** rappresenta un elemento essenziale per il monitoraggio e il miglioramento continuo delle prestazioni del sistema. Questo monitoraggio è stato implementato con **Prometheus**, una piattaforma open-source progettata per la raccolta e gestione delle metriche. Prometheus raccoglie e archivia in tempo reale i dati relativi all'andamento di vari servizi, rendendo accessibili informazioni su prestazioni, stato e utilizzo delle risorse di ciascun microservizio.
Prometheus opera configurando i microservizi per esporre endpoint specifici di metrics in formato leggibile su una porta dedicata. In questa implementazione, ogni microservizio, dalla **dashboard** al **cooperative pixel art** servizi, passando per l'**api gateway** e il **distributed logging service**, esponde un endpoint *metrics* che Prometheus interroga a intervalli regolari per raccogliere i dati. L'architettura del progetto permette, in questo modo, di avere una visione centralizzata dei comportamenti dei singoli compo-

nenti, consentendo un'analisi efficace per l'individuazione di colli di bottiglia o problemi di performance.
Per implementare il sistema di monitoraggio con Prometheus all'interno del progetto, è stato necessario creare un file di configurazione denominato *prometheus.yml*. Questo file è stato associato durante la fase di esecuzione del software, garantendo la corretta integrazione del servizio di monitoraggio con i microservizi in uso. La configurazione di *prometheus.yml* è strutturata come segue:

```
global:
  scrape_interval:     15s # By default, scrape targets every 15
     seconds.

  # Attach these labels to any time series or alerts when
     communicating with
  # external systems (federation, remote storage, Alertmanager).
  external_labels:
    monitor: 'codelab-monitor'

# A scrape configuration containing exactly one endpoint to scrape
   :
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any
     timeseries scraped from this config.
  - job_name: 'APIGatewayService'
    scrape_interval: 5s
    static_configs:
      - targets: [ 'host.docker.internal:9010' ]

  - job_name: 'DashboardService'
    scrape_interval: 5s
    static_configs:
      - targets: [ 'host.docker.internal:9011' ]

  - job_name: 'CooperativePixelArtService'
    scrape_interval: 5s
    static_configs:
      - targets: [ 'host.docker.internal:9012' ]
```

La sezione *global* definisce l'intervallo generale di scraping a 15 secondi, ossia ogni quanto Prometheus interroga gli endpoint per raccogliere metriche.
Nella sezione *scrape_configs*, ogni servizio è configurato come job con un'intervallo specifico di 5 secondi. Ogni job include il proprio nome e un indirizzo con una porta dedicata. In particolare, l'API gateway è stato dotato di un contatore Prometheus che si incrementa ad ogni chiamata API, fornendo così un'analisi quantitativa delle richieste gestite. La Dahsboard, d'altra parte, utilizza un istogramma Prometheus per monitorare i tempi di risposta associati alle diverse chiamate API, consentendo un'osservazione dettagliata delle performance in tempo reale. Infine, il servizio Cooperative Pixel Art Service im-

plementa tre misure di tipo gauge di Prometheus, le quali monitorano costantemente l'utilizzo della CPU, la memoria heap e la memoria non heap, offrendo un quadro esaustive delle risorse di sistema in uso.

1.3.1 Deployment

Prometheus è stato implementato come un servizio containerizzato tramite Docker, facilitando la sua gestione e distribuzione. Una volta avviato tramite *docker compose*, l'utente può accederee all'interfaccia della dashboard di Prometheus navigando all'indirizzo *http://localhost:9090*.

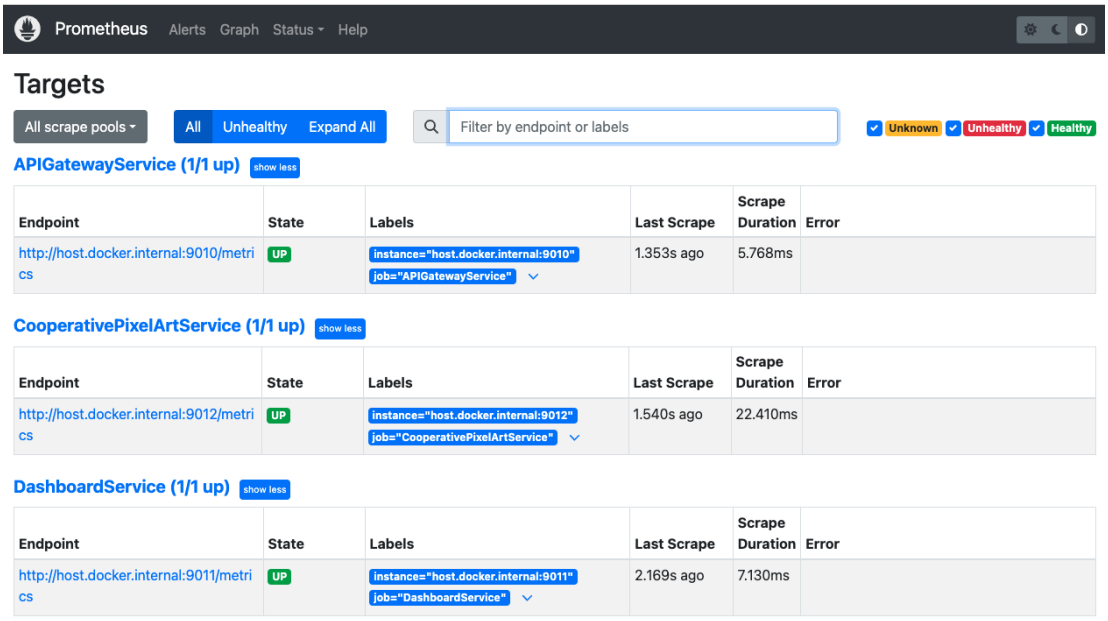

Figure 3: Prometheus Dashboard

Grafana è una piattaforma open-source avanzata per l'analisi e la visualizzazione di dati di monitoraggio. Viene utilizzata per creare dashboard interattivi e grafici dettagliati, facilitando l'osservabilità di sistemi e applicazioni. In questo progetto, Grafana è stato incluso nel file *docker-compose.yml* per connettersi a Prometheus, che funge da data source per raccogliere ed elaborare metriche. Questa integrazione consente di visualizzare i dati raccolti in modo efficace, migliorando il monitoraggio delle performance e l'identificazione di eventuali anomalie o problemi nel sistema. Qui di seguito viene mostrata un istogramma basato sui dati delle metriche di risposta delle API. La grafica mostra quante risposte delle API rientrano nei diversi intervalli di tempo. Ogni legenda nella lista fa riferimento a un endpoint specifico con i relativi dettagli di istanza e job che identifica il servizio.

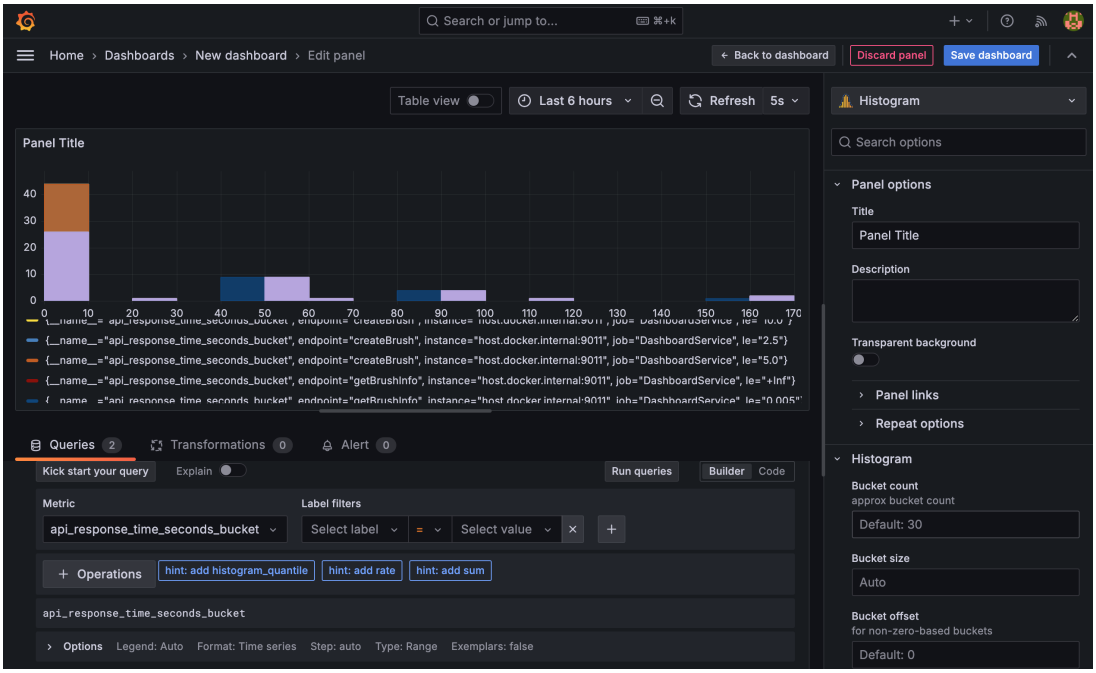

Figure 4: Grafana Dashboard

1.4 Distributed Tracing

É stato implementato un sistema di **distributed tracing** per monitorare e analizzare le interazioni tra i vari microservizi. A tal fine, è stata adottata la soluzione **Zipkin**, un sistema open source per la raccolta e l'analisi delle tracce di esecuzione delle applicazioni distribuite. Zipkin consente di tracciare le chiamate tra i microservizi, facilitando l'identificazione dei colli di bottiglia e delle problematiche di latenza.
La configurazione di Zipkin è stata implementata in modo standard all'interno dei microservizi. Nel launcher di ciascun microservizio è stato creato un sender per inviare i dati di tracciamento a Zipkin utilizzando il protocollo HTTP tramite l'istanza di OkHttpSender. Successivamente, è stato creato un gestore di span asincrono (AsyncZipkinSpanHandler) associato al sender.

```
public class APIGatewayServiceLauncher {

    public static void main(String[] args) {
        JvmMetrics.builder().register();
        Counter counter = Counter.builder()
                .name("API_Gateway_http_requests_total")
                .help("Total number of HTTP requests")
                .labelNames("method", "path", "status")
                .register();
        OkHttpSender sender = OkHttpSender.create("http://
           localhost:9411/api/v2/spans");
        AsyncReporter<Span> reporter = AsyncReporter.create(
           sender);
        SpanHandler zipkinSpanHandler = ZipkinSpanHandler.
           newBuilder(reporter).build();
        Tracing tracing = Tracing.newBuilder()
            .localServiceName("APIGatewayService")
            .addSpanHandler(zipkinSpanHandler)
            .currentTraceContext(
               ThreadLocalCurrentTraceContext.newBuilder().
               build())
            .build();
        Tracer tracer = tracing.tracer();
        ScopedSpan span = tracer.startScopedSpan("
           APIGatewayService-main");
        APIGatewayService service = new APIGatewayService();
        try (HTTPServer server = HTTPServer.builder().port
           (9010).buildAndStart()){
            System.out.println("[API Gateway] HTTPServer
               listening on port http://localhost:" + server
               .getPort() + "/metrics");
            service.launch(counter);
            Thread.currentThread().join();
        } catch (Exception e) {
            span.error(e);
        } finally {
            span.finish();
        }
        tracing.close();
        reporter.close();
        sender.close();
    }
}
```

L'integrazione di Zipkin nelle operazioni del microservizio è stata effettuata utilizzando l'oggetto Tracer fornito da Zipkin. Ad esempio, nel metodo **createBrush**, è stato creato uno span associato al tracer prima dell'esecuzione delle richieste necessarie.

```
protected void createBrush(RoutingContext context) {
    ScopedSpan span = this.tracer.startScopedSpan("
       createBrush");
    try {
        this.counter.labelValues("POST", "/api/brushes", "
           success").inc();
        logger.log(Level.INFO, "CreateBrush request - " +
           context.currentRoute().getPath());
        JsonObject reply = new JsonObject();
        serviceAPI.createBrush()
           .onSuccess((String brushId) -> {
               try {
                       reply.put("brushId", brushId);
                       sendReply(context.response(), reply);
               } catch (Exception ex) {
                       sendServiceError(context.response());
               }
        }).onFailure((e) -> {
                sendServiceError(context.response());
                this.counter.labelValues("POST", "/api/
                   brushes", "error").inc();
        });
    } finally {
        span.finish();
    }
}
```

Tutti i dati ottenuti dal tracciamento sono visualizzabili sulla dashboard di Zipkin.

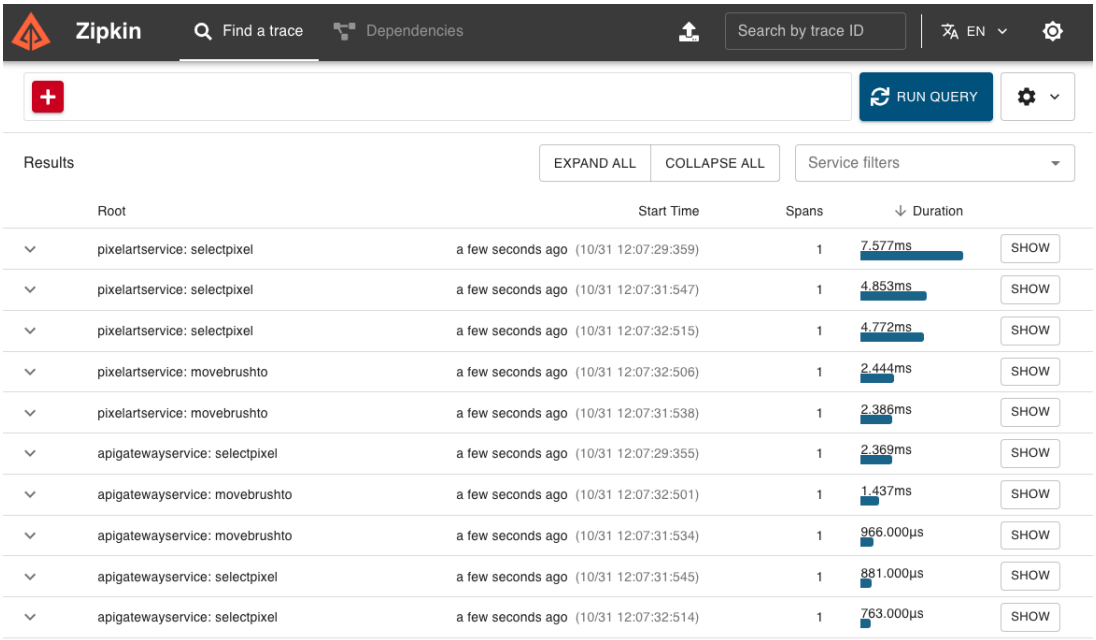

Figure 5: Zipkin Dashboard

1.4.1 Deployment

In questo progetto ci siamo servizi dell'immagine docker **openzipkin/zipkin** che permette di visualizzare la dashboard all'indirizzo *http://localhost:9411*. Nella home della dashboard si possono visualizzare i vari servizi su cui sta venendo effettuato il tracciamento.