

# Assignment #7 - 20231201 - Microservizi Osservabili

## Microservizi Osservabili nell'Arte Pixel Cooperativa

### Descrizione

L'obiettivo di questo compito è applicare praticamente alcuni pattern per creare microservizi pronti per la produzione, come visto nel modulo-2.9, utilizzando le tecnologie discusse in Lab-11-20231201. L'assignment si concentra sull'estensione dell'esempio di Cooperative Pixel Art (Lab-07-20231027). L'esempio attuale include un semplice API Gateway che interagisce con un servizio PixelArtGrid.

Si è esteso l'esempio applicando i seguenti pattern:

1. *API Health Check*
2. *Distributed Logs*
3. *Application Metrics*
4. *Distributed Tracing*

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

### Deliverable

Fornisci un repository Github che includa:

- Codice Sorgente: Organizzato in una struttura di directory adeguata.
- Relazione: Documentazione dei dettagli di implementazione e delle intuizioni acquisite durante il processo.

## 1. API di Health Check.

L'Health Check, o controllo di salute, è un meccanismo essenziale nell'ambito delle architetture di microservizi, progettato per monitorare e garantire il corretto funzionamento dei singoli servizi. Questa pratica consiste nell'esecuzione di verifiche periodiche per assicurarsi che un microservizio sia in uno stato sano e pronto a rispondere alle richieste. Gli Health Check sono utilizzati per migliorare l'affidabilità, la resilienza e la capacità di gestione delle applicazioni distribuite.

Uno dei **principali vantaggi** dell'Health Check è la **tempestiva individuazione di problemi** nelle singole componenti del sistema. Attraverso la verifica regolare dello stato di salute di un microservizio, è possibile identificare precocemente eventuali malfunzionamenti, errori o degrado delle prestazioni. Ciò consente agli sviluppatori e agli operatori di sistema di intervenire prontamente, risolvendo i problemi prima che possano propagarsi e influire sulle prestazioni complessive del sistema.

Un altro **vantaggio** significativo è la capacità di **gestire il traffico** delle richieste. Gli Health Check consentono di dirottare le richieste solo verso i microservizi che sono in uno stato sano, evitando così di inviare traffico a servizi compromessi o non funzionanti. Questo migliora la disponibilità complessiva dell'applicazione e riduce il rischio di interruzioni del servizio.

Tuttavia, l'implementazione degli Health Check introduce anche alcuni **svantaggi**. In primo luogo, i controlli periodici possono causare un **aumento del carico di lavoro** e delle risorse del sistema. Se non gestiti correttamente, gli Health Check troppo frequenti possono comportare un consumo eccessivo di risorse, influenzando negativamente le prestazioni generali dell'applicazione. Pertanto, è fondamentale bilanciare la frequenza degli Health Check in modo ottimale.

Inoltre, l'Health Check **può rivelare informazioni sensibili** sullo stato interno dei microservizi. Questo potrebbe diventare un problema in termini di sicurezza, poiché le informazioni divulgate possono essere utilizzate da potenziali attaccanti per identificare e sfruttare vulnerabilità nel sistema. Di conseguenza, è necessario implementare misure di sicurezza adeguate per garantire che le informazioni rivelate dagli Health Check non possano essere utilizzate malevolmente.

In conclusione, l'Health Check è un elemento cruciale nelle architetture di microservizi, fornendo numerosi vantaggi per la rilevazione precoce dei problemi e la gestione del traffico. Tuttavia, è essenziale considerare attentamente la frequenza di esecuzione e implementare misure di sicurezza per mitigare gli svantaggi associati. Un equilibrio accurato nell'implementazione degli Health Check contribuirà a mantenere un sistema distribuito affidabile e robusto.

## Implementazione nel progetto

Si è scelto di implementare nel progetto il sistema di Health Check per verificare la validità delle API prima del loro utilizzo in modo tale da fornire al progetto la capacità di riadattarsi sullo stato dei vari microservizi. Il formato seguito per il checking dello stato dei vari microservizi è quello basato su **microprofiling** con seguente formato **JSON**:

```
INFORMAZIONI: Body: {
  "createBrush" : true,
  "getCurrentBrushes" : true,
  "getBrushInfo" : true,
  "destroyBrush" : true,
  "moveBrushTo" : true,
  "changeBrushColor" : true,
  "selectPixel" : true,
  "getPixelGridState" : true,
  "status" : "UP"
}
```

All'interno del file vengono riportati i metodi utilizzati dagli handler delle varie route forniti dall'API. Ciò che viene fatto è: fare eseguire ciascuno di questi metodi e verificare il loro completamento, se questo va a buon fine ritornano uno stato di true, altrimenti di false.

Nella fase di validazione dello status di health check si imposta il risultato finale come congiunzione del risultato dei vari metodi se tutti ritornano true viene impostato ad **"UP"** altrimenti, se anche uno dovesse fallire a **"DOWN"**.

Questo controllo viene fatto periodicamente poiché viene eseguito prima di effettuare una qualsiasi route dell'API in modo tale da prevenire l'esecuzione del microservizio in caso in cui qualcosa non funzionasse. Di seguito viene riportato il nuovo API REST: "

```
/* configure the HTTP routes following a REST style */

//Every incoming request will first go through the handleRouteRequest method, which performs the health check.
router.route().handler(this::handleRouteRequest);
//If you want to check manually through localhost decomment this line of code.
//router.route(HttpMethod.GET, "/health").handler(this::healthCheck);

router.route(HttpMethod.POST, path: "/api/brushes").handler(this::createBrush);
router.route(HttpMethod.GET, path: "/api/brushes").handler(this::getCurrentBrushes);
router.route(HttpMethod.GET, path: "/api/brushes/:brushId").handler(this::getBrushInfo);
router.route(HttpMethod.DELETE, path: "/api/brushes/:brushId").handler(this::destroyBrush);
router.route(HttpMethod.POST, path: "/api/brushes/:brushId/move-to").handler(this::moveBrushTo);
router.route(HttpMethod.POST, path: "/api/brushes/:brushId/change-color").handler(this::changeBrushColor);
router.route(HttpMethod.POST, path: "/api/brushes/:brushId/select-pixel").handler(this::selectPixel);
router.route(HttpMethod.GET, path: "/api/pixel-grid").handler(this::getPixelGridState);
this.handleEventSubscription(server, path: "/api/pixel-grid/events");
```

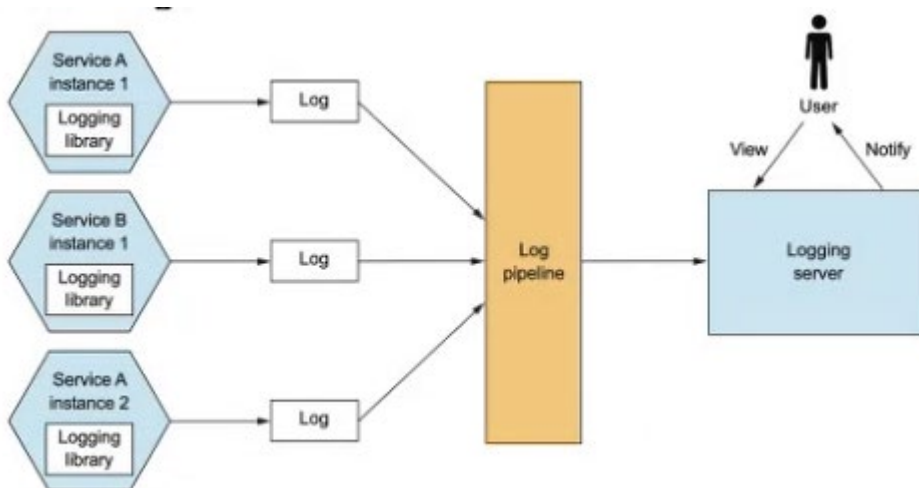
Che procede all'invocazione di un metodo di nome "handleRequest" che non è altro che un controllo per dividere in step la gestione della route. Appare in questo modo:

```
1 usage  🍷Marzoli Leo
private void handleRouteRequest(RoutingContext context) {
    if (healthCheck(context)) {
        // Proceed with the actual route handling logic
        context.next();
    } else {
        // Respond with a service unavailable status
        sendServiceUnavailable(context.response());
    }
}
```

L'implementazione **a livello architetturale** del sistema di Health Check riguarda sia il servizio di "API Gateway", che quello di "CooperativePixelArtService". In quanto voglio sia poter verificare lo stato dell'API Gateway e quello del servizio da esso chiamato che implementa la business logic. Questo perché in caso di malfunzionamento uno sviluppatore, può prontamente intervenire nella sostituzione di uno o dell'altro, in caso in cui questo compito non sia già delegato ad un'altra componente del sistema.

## 2. Distributed Logs.

Allo scopo di monitorare l'andamento dell'applicazione si è scelto di optare per lo sviluppo di un nuovo microservizio "DistributedLogService" in grado di collegarsi ai precedenti microservizi attraverso il seguente stile architetturale:



Difatti, ogni microservizio dispone di un metodo "sendLogRequest" all'interno di quella che deve essere la sua business logic così che quando viene invocato produce dei Log da mandare al nuovo microservizio. Questo avrà il compito di mettere in pipeline i vari Log e l'utente poi sarà in grado di visualizzare i risultati monitorando così l'andamento del

sistema. Il Metodo che ciascuno microservizio deve implementare si presenta nel seguente modo:

```
//Part of the Pattern for the Distributed Log.
10 usages  🐼 Marzoli Leo
private Future<Void> sendLogRequest(String messageLog) {
    Promise<Void> p = Promise.promise();

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

            // Converti l'oggetto JSON in una stringa e invialo come corpo della richiesta
            String payload = logData.encodePretty();
            request.putHeader(name: "content-length", value: "" + 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 espone due API Rest una di GET e una di POST, rispettivamente una per la visualizzazione dei messaggi messi in pipeline e una per catturare tutti i messaggi di Log che vengono inviati a runtime di ciascun microservizio.

```
/* configure the HTTP routes following a REST style */
router.route(HttpMethod.GET, path: "/api/log").handler(this::sendLogsList);
router.route(HttpMethod.POST, path: "/api/log").handler(this::printLogMessage);
```

Il risultato finale accedendo a "http://localhost:9003/api/log" sarà un elenco di tutti i log prodotti da ciascun microservizio, di seguito viene riportato una parte dei Log prodotti:

```
{
  "logs" : [ {
    "content:" : "{\r\n  \"message\" : \"[API Gateway] - Init of the PixelArtServiceProxy.\"\r\n}"
  }, {
    "content:" : "{\r\n  \"message\" : \"[CooperativePixelArtService] - Init the RestPixelArtServiceControllerVerticle!\"\r\n}"
  }, {
    "content:" : "{\r\n  \"message\" : \"[DashBoard] - Init of the PixelArtServiceProxy.\"\r\n}"
  }, {
    "content:" : "{\r\n  \"message\" : \"[DashBoard] - Terminated createBrush!\"\r\n}"
  }, {
    "content:" : "{\r\n  \"message\" : \"[API Gateway] - Terminated CreateBrush!\"\r\n}"
  }, {
    "content:" : "{\r\n  \"message\" : \"[CooperativePixelArtService] - Terminated createBrush!\"\r\n}"
  }, {
    "content:" : "{\r\n  \"message\" : \"[DashBoard] - Terminated subscribePixelGridEvent!\"\r\n}"
  }, {
    "content:" : "{\r\n  \"message\" : \"[API Gateway] - Terminated subscribePixelGridEvent!\"\r\n}"
  }, {
    "content:" : "{\r\n  \"message\" : \"[CooperativePixelArtService] - Terminated handleEventSubscription!\"\r\n}"
  }, {
  }
```

## 2. Application Metrics.

Per lo scopo si è scelto di utilizzare Prometheus esso è un sistema di monitoraggio open-source progettato per raccogliere e analizzare metriche di sistema e applicazioni in ambienti informatici complessi, in questo caso microservizi. Offre numerosi vantaggi.

Il sistema adotta un modello di tipo "pull", dove il server Prometheus raccoglie attivamente i dati da servizi da monitorare. La sua architettura distribuita permette la scalabilità e la ridondanza, essenziali per ambienti complessi e su larga scala.

Prometheus utilizza il linguaggio di query PromQL, semplificando l'analisi dei dati raccolti. Supporta un sistema di allerta che avvisa gli utenti quando le metriche superano limiti prestabiliti o soddisfano condizioni specifiche, con notifiche attraverso vari canali.

In sintesi, Prometheus è un potente strumento di monitoraggio e gestione delle prestazioni, noto per la sua facilità d'uso, integrazione con l'ambiente cloud native e capacità di analisi avanzata delle metriche raccolte.

## Implementazione

Per poter implementare il software nel progetto si è dovuto creare un file di configurazione prometheus.yml da associare al momento della messa in esecuzione.

Questo file appare nel seguente modo:



```

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: 'PrometheusTest'
    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9400']

  - job_name: 'APIGatewayService'
    scrape_interval: 5s
    static_configs:
      - targets: [ 'localhost:9010' ]

  - job_name: 'DashboardService'
    scrape_interval: 5s
    static_configs:
      - targets: [ 'localhost:9011' ]

  - job_name: 'CooperativePixelArtService'
    scrape_interval: 5s
    static_configs:
      - targets: [ 'localhost:9012' ]

```

Definisce al suo interno tre nuovi “job” oltre a quello di prova “PrometheusTest” che rappresentano ciascuno un microservizio dell’applicazione, viene definito per ognuno di essi uno scrape di 5s. Quindi il server Prometheus rimarrà in ascolto sui vari endpoint e aggiornerà le metriche al finire di quell’intervallo.

Si è scelto di creare dei jobs separati in modo da distinguere la logica di ciascun microservizio, ma questi potevano effettivamente essere inseriti tutti nello stesso job mettendo nella sezione “targets” più elementi.

A questo punto facendo partire l’eseguibile di Prometheus e si potrà accedere tramite browser Chrome al localhost:9090 per visualizzare la GUI. Andando nella sezione “targets” ciò che apparirà sarà il seguente:

# Targets

All scrape pools

AllUnhealthyCollapse All

Filter by endpoint or labels

APIGatewayService (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9010/metrics	UP	instance="localhost:9010" job="APIGatewayService"	594.000ms ago	2.790ms	

CooperativePixelArtService (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9012/metrics	UP	instance="localhost:9012" job="CooperativePixelArtService"	175.000ms ago	8.298ms	

DashboardService (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9011/metrics	UP	instance="localhost:9011" job="DashboardService"	1.400s ago	7.944ms	

L'immagine raffigura i tre microservizi del sistema, che, come si può vedere, risiedono nello stato di UP, dunque sono monitorati. Accedendo a ciascuno di essi verranno esposte le metriche, come nel seguente esempio:

```
# HELP API_Gateway_http_requests_total Total number of HTTP requests
# TYPE API_Gateway_http_requests_total counter
API_Gateway_http_requests_total{method="POST",path="/api/brushes",status="success"} 1.0
# HELP jvm_buffer_pool_capacity_bytes Bytes capacity of a given JVM buffer pool.
# TYPE jvm_buffer_pool_capacity_bytes gauge
jvm_buffer_pool_capacity_bytes{pool="direct"} 1.2591151E7
jvm_buffer_pool_capacity_bytes{pool="mapped"} 0.0
jvm_buffer_pool_capacity_bytes{pool="mapped - 'non-volatile memory'"} 0.0
# HELP jvm_buffer_pool_used_buffers Used buffers of a given JVM buffer pool.
# TYPE jvm_buffer_pool_used_buffers gauge
jvm_buffer_pool_used_buffers{pool="direct"} 9.0
jvm_buffer_pool_used_buffers{pool="mapped"} 0.0
jvm_buffer_pool_used_buffers{pool="mapped - 'non-volatile memory'"} 0.0
# HELP jvm_buffer_pool_used_bytes Used bytes of a given JVM buffer pool.
# TYPE jvm_buffer_pool_used_bytes gauge
jvm_buffer_pool_used_bytes{pool="direct"} 1.2591152E7
jvm_buffer_pool_used_bytes{pool="mapped"} 0.0
jvm_buffer_pool_used_bytes{pool="mapped - 'non-volatile memory'"} 0.0
```

Inoltre, si è dovuto modificare il codice pre-esistente di ciascun microservizio per avere delle metriche personalizzate. Per esempio: l'API Gateway introduce un Prometheus counter che incrementa ad ogni chiamata API, la Dashboard utilizza un Prometheus Histogram per monitorare il tempo di risposta alle varie chiamate API. E il PixelArtCooperativeService implementa tre Prometheus gauge per monitorare la CPU, l'heap e il non heap.

Tramite il monitoraggio di questi aspetti si sono voluti garantire quelle che sono dei quality attributes del sistema, ovvero:



- **Reliability:** in quanto il sistema opererà come previsto entro un certo intervallo di tempo che viene rappresentato dal tempo di risposta dell'API della Dashboard.
- **Usabilità:** il sistema è user-friendly in quanto la GUI messa a disposizione dalla Dashboard è facilmente comprensibile e espone tutte le funzionalità in maniera chiara e concisa.
- **Verificabile:** il sistema è stato esteso con una serie di softwares, come Prometheus, Zipkin, HealthCheck e DistributedLogs che permettono di ricavare facilmente metriche utili alla misurazione della performance.
- **Maintainability:** l'estensioni apportate facilitano il mantenimento del software e il suo continuo versioning.
- **Repairability:** è facilmente riparabile in un tempo relativamente breve.
- **Portability:** il sistema è eseguibile in diversi OS.

### 3. Distributed Tracing

Zipkin è uno strumento open source specializzato nel **distributed tracing**, progettato per fornire una visione dettagliata e analitica delle richieste che attraversano diversi servizi all'interno di un'architettura a microservizi. Questa tecnologia è particolarmente utile per individuare e risolvere problemi di latenza e prestazioni in sistemi distribuiti.

Difatti questo strumento offre una visione dettagliata e analitica delle richieste che attraversano diversi servizi all'interno di un'architettura a microservizi. Questa tecnologia è particolarmente utile per individuare e risolvere problemi di latenza e prestazioni in sistemi distribuiti.

## Implementazione

### Configurazione Iniziale

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.

```

var sender = OkHttpClient.create("http://127.0.0.1:9411/api/v2/spans");
var zipkinSpanHandler = AsyncZipkinSpanHandler.create(sender);

var tracing = Tracing
    .newBuilder()
    .localServiceName("APIGatewayService")
    .addSpanHandler(zipkinSpanHandler)
    .build();

var tracer = tracing.tracer();

var span = tracer.startScopedSpan("APIGatewayService-main");

try {
    APIGatewayService service = new APIGatewayService();
    service.launch();
} catch (Exception ex) {
    span.error(ex);
} finally {
    span.finish();
}

```

## Tracciamento nelle Operazioni

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

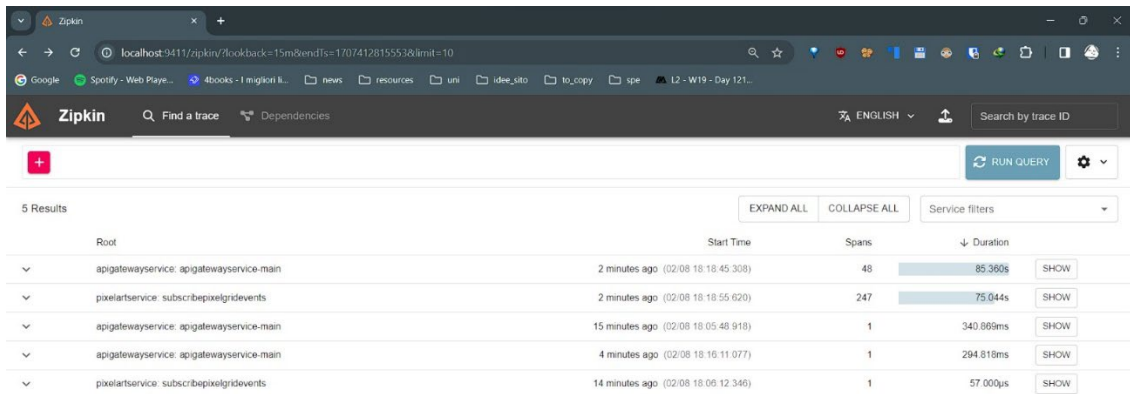
```

protected void getBrushInfo(RoutingContext context) {
    var span = tracer.startScopedSpan("getBrushInfo");
    try {
        logger.log(Level.INFO, "Get Brush info request: " + context.currentRoute().getPath());
        String brushId = context.pathParam("brushId");
        JsonObject reply = new JsonObject();
        serviceAPI
            .getBrushInfo(brushId)
            .onSuccess((JsonObject info) -> {
                try {
                    reply.put("brushInfo", info);
                    sendReply(context.response(), reply);
                } catch (Exception ex) {
                    sendServiceError(context.response());
                }
            })
            .onFailure((e) -> {
                sendServiceError(context.response());
            });
    } finally {
        span.finish();
    }
}

```

Tutti i dati ottenuti dal tracciamento sono visualizzabili sulla dashboard di Zipkin. In questo progetto ci siamo serviti dell'immagine docker **opnzipkin/zipkin** che permette di

visualizzare la dashboard sull'indirizzo **localhost:9411**. Nella home della dashboard si possono visualizzare i vari servizi su cui sta venendo effettuato il tracciamento.



The screenshot shows the Zipkin dashboard interface. At the top, there's a navigation bar with the Zipkin logo, a search bar labeled 'Find a trace', and a 'Dependencies' link. On the right, there's a language selector set to 'ENGLISH' and a 'Search by trace ID' input field. Below the navigation bar, there's a 'RUN QUERY' button and a settings icon. The main content area displays '5 Results' in a table. The table has columns for 'Root', 'Start Time', 'Spans', and 'Duration'. Each row represents a search result with a 'SHOW' button to the right. The results are as follows:

Root	Start Time	Spans	Duration	SHOW
apigateway-service: apigateway-service-main	2 minutes ago (02/08 18:18:45.308)	48	85.360s	SHOW
pixelart-service: subscribepixelgridevents	2 minutes ago (02/08 18:18:55.620)	247	75.044s	SHOW
apigateway-service: apigateway-service-main	15 minutes ago (02/08 18:05:48.918)	1	340.889ms	SHOW
apigateway-service: apigateway-service-main	4 minutes ago (02/08 18:16:11.077)	1	294.818ms	SHOW
pixelart-service: subscribepixelgridevents	14 minutes ago (02/08 18:06:12.346)	1	57.000µs	SHOW

Selezionando un servizio diventa possibile visualizzare tutte le richieste che quel servizio effettua e le rispettive metriche annesse.

