

Implementación de Zipkin con RabbitMQ

Índice

- 01** [Introducción](#)
- 02** [Zipkin](#)
- 03** [Dependencias y configuración](#)



01

Introducción

En otro apartado vimos que los servicios se comunican entre sí de manera asincrónica usando Message Brokers. Por lo general, estos mensajes cumplen con las siguientes características:

- Asincrónicos
- De alto volumen
- Múltiples emisores

Hasta ahora, pensamos en estos mensajes como información de negocio (órdenes de compra, productos, etc.). Ahora, vamos a ver otro caso típico, pero que se trata de otro tipo de información.

Logs e información de latencia

Los logs son información de “tranceo” de un sistema. Pueden ser errores (excepciones), información de ejecución, mensajes de Debug, información de latencia, etc. La información de latencia es aquella que identifica los tiempos de entrada y salida de un servicio, es decir, cuánto demora desde que recibe un request hasta que genera una respuesta. Esta información tiene las siguientes características:

- Asincrónica (se loguea la excepción, pero no tiene ningún procesamiento posterior)
- Gran volumen
- Múltiples emisores (toda instancia de microservicio genera información de log todo el tiempo)

A la luz de lo anterior, sería razonable manejar esta información de manera asincrónica, enviando de alguna manera todos los mensajes a un Message Broker (como RabbitMQ) que centralice toda la información.

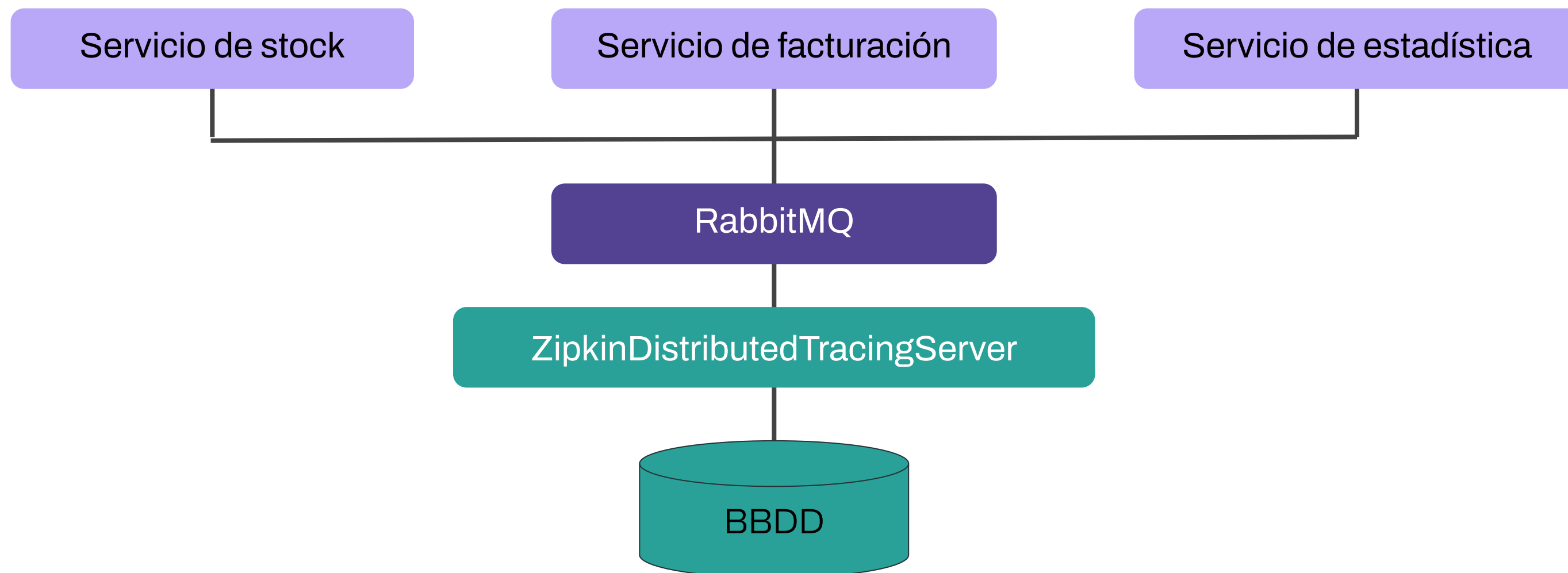
Repasando traceo distribuido

El traceo distribuido es una técnica que se usa para monitorear aplicaciones, especialmente aquellas construidas usando la arquitectura de microservicios.

- Ayuda a identificar los microservicios fallidos o los servicios que tienen problemas de rendimiento cuando hay muchos servicios llamados dentro de una solicitud.
- Es muy útil cuando necesitamos rastrear la solicitud que pasa por múltiples microservicios.
- También se utiliza para medir el rendimiento de los microservicios.

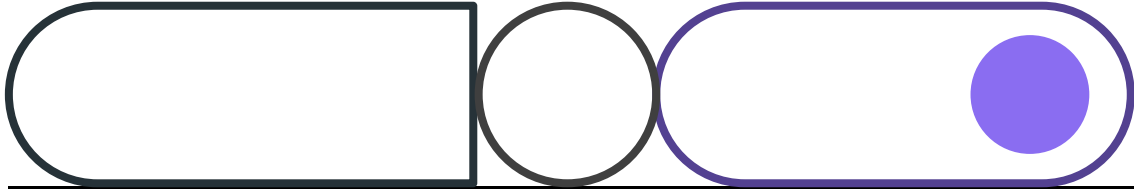
Traceo distribuido con Zipkin y RabbitMQ

ZipkinDistributedTracingServer se conecta a una base de datos en memoria. Todos los microservicios colocarán la información de traceo (errores, latencia, debug, etc.) a través de mensajes en el servidor RabbitMQ, desde donde **ZipkinDistributedTracingServer** consume los mismos.

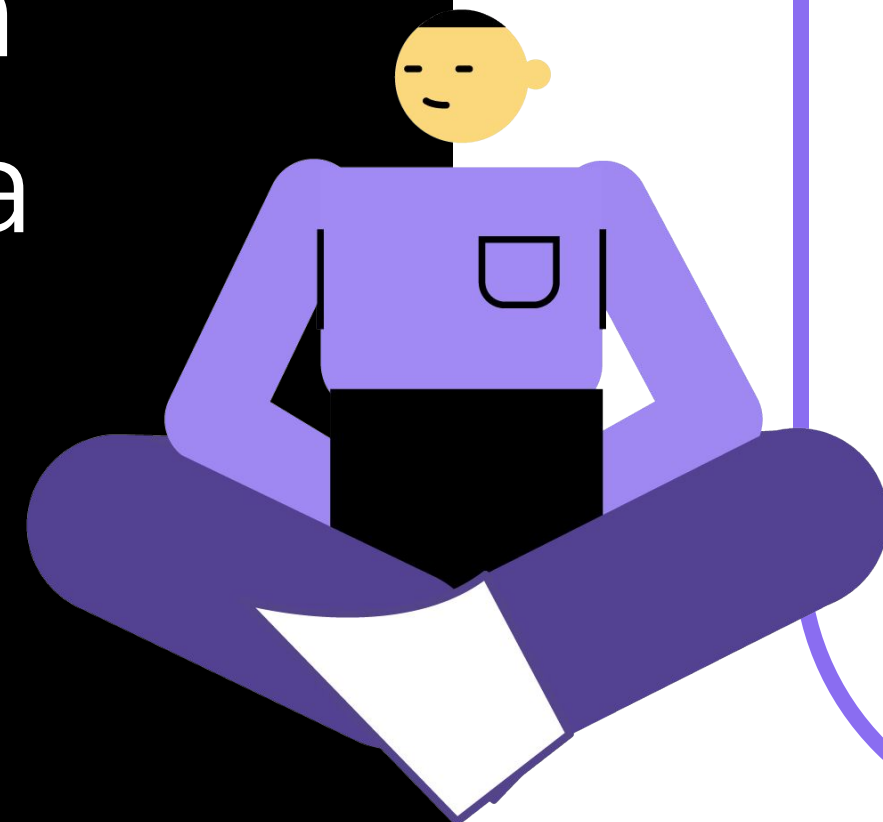


02

Zipkin



Zipkin es un sistema de
tracing distribuido, cuya
función principal consiste en
la recopilación y la búsqueda
de esta información.



Zipkin

Es una aplicación que permite identificar problemas de latencia (demoras) en los requests. Al agregar las dependencias de Zipkin en una app Java, este intercepta los requests y va generando información que será recolectada para analizar problemas de latencia y errores. Para más información pueden ingresar en: zipkin.io.

Vale aclarar que hay muchos sistemas de traceo distribuidos, como Jaeger, New Relic (uno de los más conocidos), Splunk, AWS X-Ray. Todos funcionan de la misma manera, aunque tienen diferentes funciones y mejoras en la interfaz. Se optó por Zipkin por su facilidad de uso.

¿Cómo instalar Zipkin local?

Existen dos maneras de instalarlo:

01 Ejecutando la aplicación directamente de Java:

```
curl -sSL https://zipkin.io/quickstart.sh | bash -s
```

→ Descarga el JAR

```
java -jar zipkin.jar
```

→ Ejecuta el JAR

02 Ejecutando un container Docker:

```
docker run -d -p 9411:9411 openzipkin/zipkin
```

←

→

🔄

📄


localhost:9411/zipkin/

🔖

☆

👤

⋮

 Zipkin

🔍

Find a trace

🔗

Dependencies

🌐

ENGLISH

📄

🔍

Search by trace ID

+

🔄

RUN QUERY

⚙️

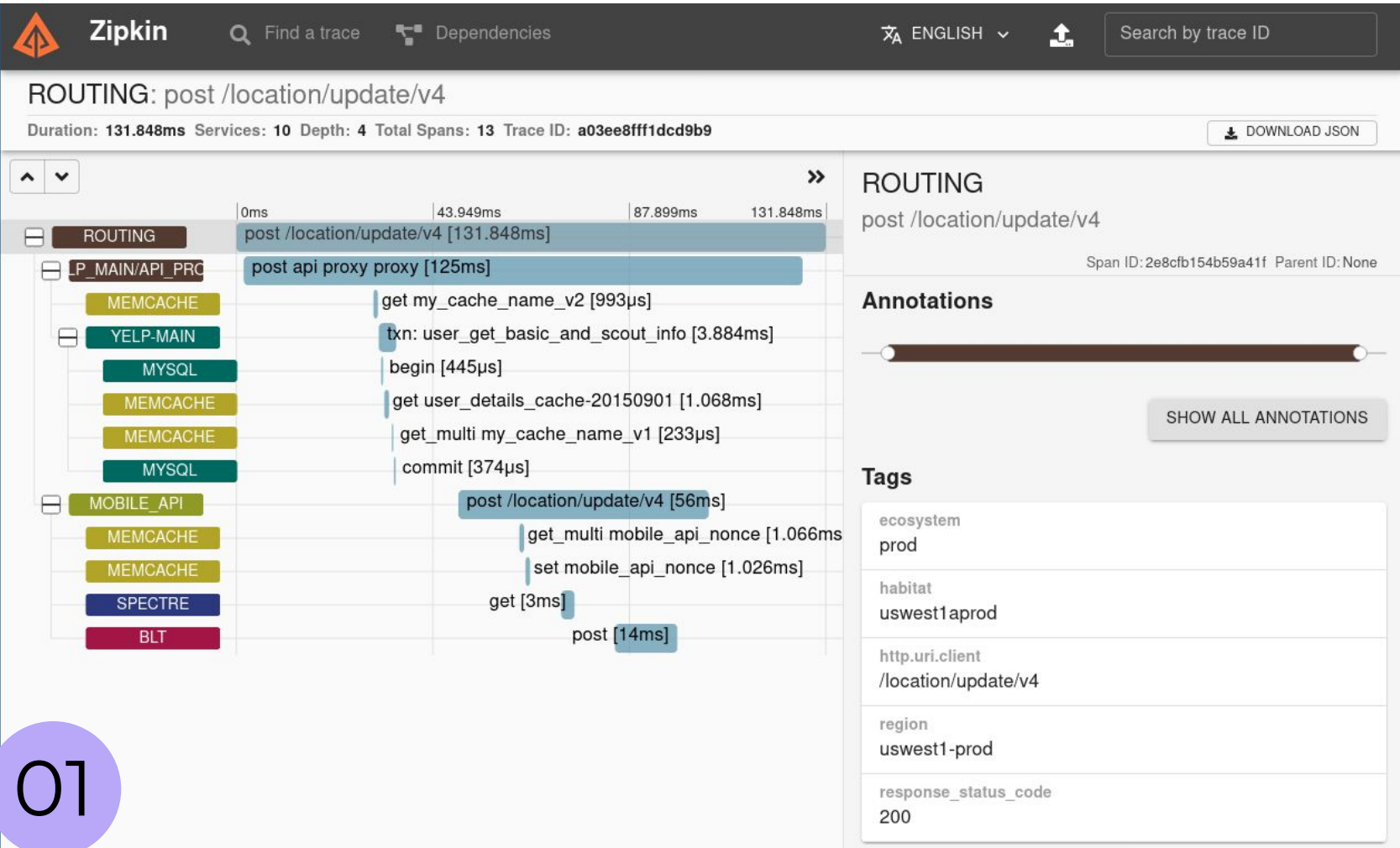
🔍

Search Traces

Please select criteria in the search bar. Then, click the search button.

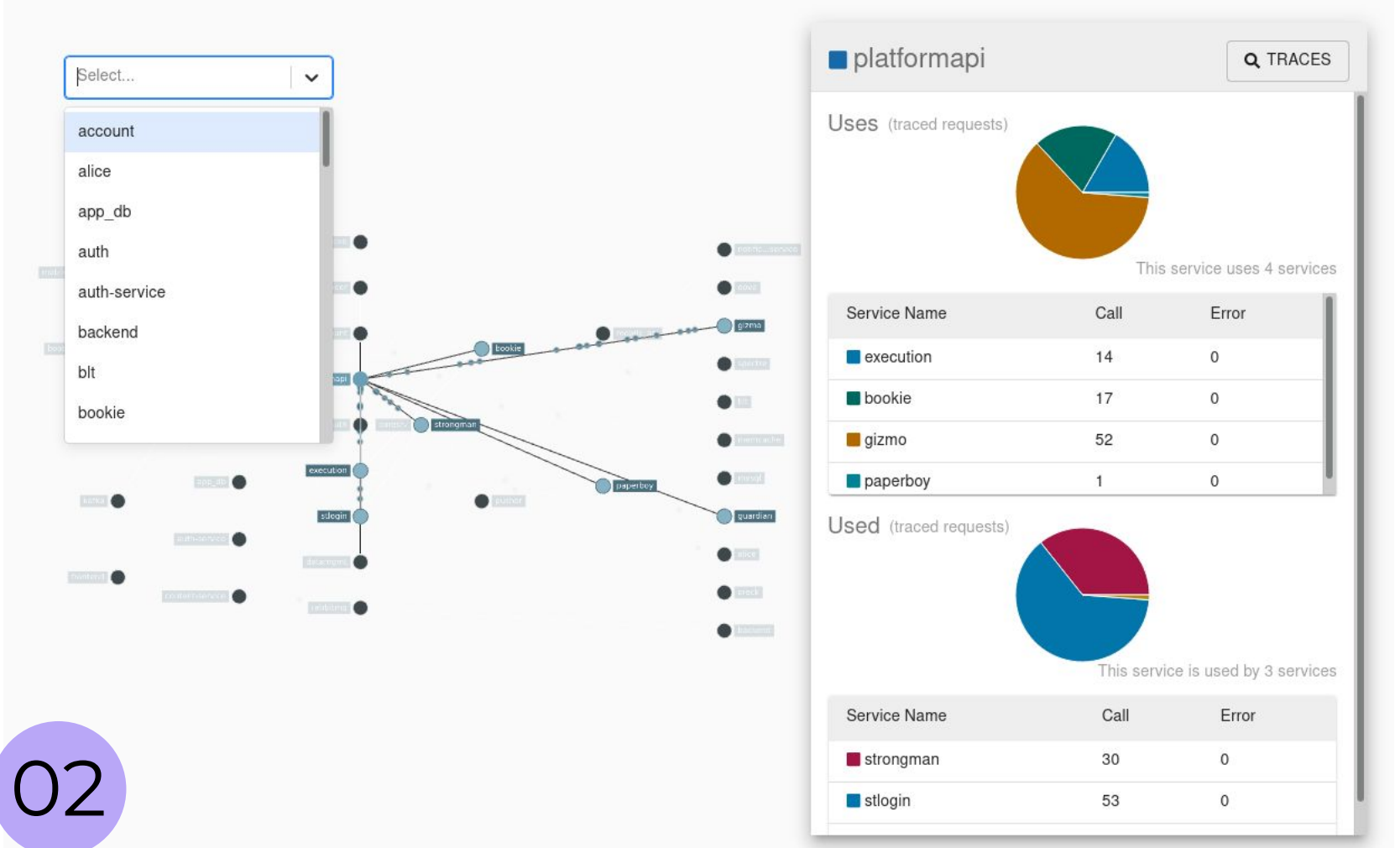
Una vez instalado podrán acceder a la consola de Zipkin a través del navegador en: <http://localhost:9411>.

¿Cómo funciona?



01

En este gráfico de Gantt, se puede observar la secuencia de llamadas y el tiempo que demoró cada una. Con esta información se pueden identificar los “cuellos de botella”.



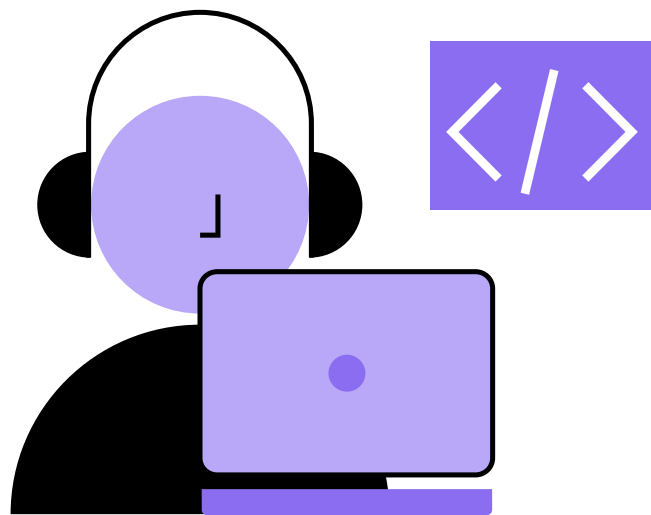
02

En este gráfico se muestra el mapa de dependencias entre servicios. Esto nos permite determinar qué servicios se ven afectados al ocurrir una falla en algún punto del sistema.

03

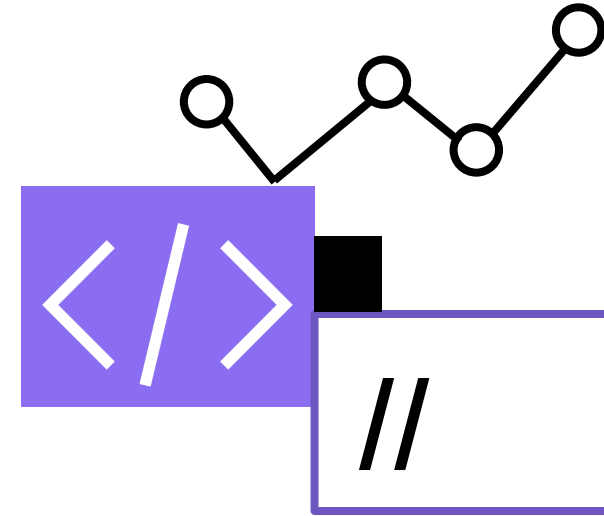
Dependencias y configuración

Agregamos dependencias al POM.XML



```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth
</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.amqp</groupId>
  <artifactId>spring-rabbit</artifactId>
</dependency>
```



Configuración .yml

Se pueden personalizar valores por defecto, como puertos, URL de zipkin, etc.

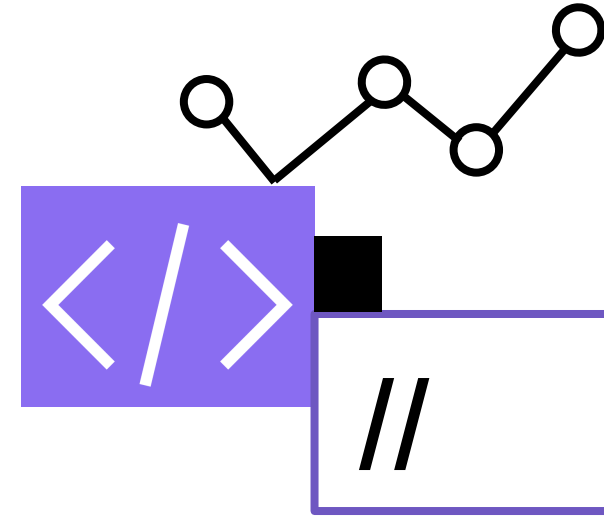
```
spring:
  sleuth:
    sampler:
      probability: 1.0
  rabbitmq:
    addresses:localhost:5672
    virtual-host: app_vhost
    username:username
    password:password
  zipkin:
    baseUrl: http://localhost:9411/
    enabled:true
    sender:
      type:rabbit
    enabled:true
```


Si ejecutan desde Java, van a ver lo siguiente. Si lo hacen desde Docker, quedará dentro de la consola del container.

```
*****  
**      **  
  
*       *  
  
**      **  
  
**      **  
  
**      **  
  
*****  
****  
***  
***  
***  
***  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```

```
:: Powered by Spring Boot ::          (v2.1.4.RELEASE)
```

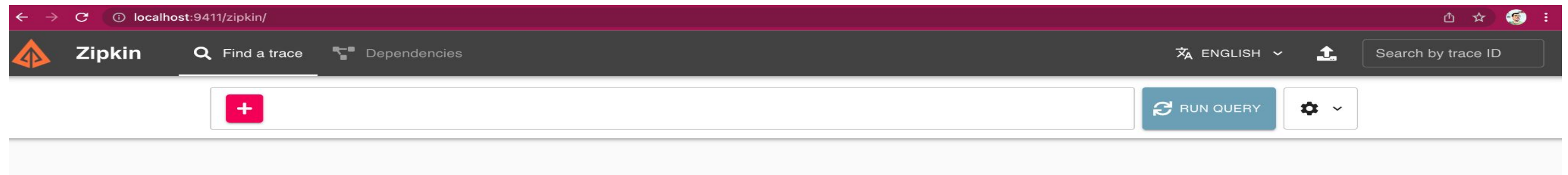
```
2022-02-10 11:56:05.786 INFO 18556 --- [main] z.s.ZipkinServer : Starting ZipkinServer on DESKTOP-KRCLJMG with PID 18556 (C:\Users\Adriano\Downloads\zipkin-server-2.12.9-exec.jar started by Adriano in C:\Users\Adriano\Downloads)  
2022-02-10 11:56:05.790 INFO 18556 --- [main] z.s.ZipkinServer : The following profiles are active: shared  
2022-02-10 11:56:06.958 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.verboseExceptions: false (default)  
2022-02-10 11:56:06.958 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.verboseSocketExceptions: false (default)  
2022-02-10 11:56:06.960 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.verboseResponses: false (default)  
2022-02-10 11:56:06.963 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.useEpoll: false (default)  
2022-02-10 11:56:07.113 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.useOpenSsl: true (default)  
2022-02-10 11:56:07.114 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.maxNumConnections: 2147483647 (default)  
2022-02-10 11:56:07.115 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.numCommonWorkers: 12 (default)  
2022-02-10 11:56:07.115 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.numCommonBlockingTaskThreads: 200 (default)  
2022-02-10 11:56:07.117 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultMaxRequestLength: 10485760 (default)  
2022-02-10 11:56:07.118 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultMaxResponseLength: 10485760 (default)  
2022-02-10 11:56:07.118 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultRequestTimeoutMillis: 10000 (default)  
2022-02-10 11:56:07.119 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultResponseTimeoutMillis: 15000 (default)  
2022-02-10 11:56:07.119 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultConnectTimeoutMillis: 3200 (default)  
2022-02-10 11:56:07.120 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultServerIdleTimeoutMillis: 15000 (default)  
2022-02-10 11:56:07.120 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultClientIdleTimeoutMillis: 10000 (default)  
2022-02-10 11:56:07.121 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultHttp2InitialConnectionWindowSize: 1048576 (default)  
2022-02-10 11:56:07.121 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultHttp2InitialStreamWindowSize: 1048576 (default)  
2022-02-10 11:56:07.122 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultHttp2MaxFrameSize: 16384 (default)  
2022-02-10 11:56:07.123 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultHttp2MaxStreamsPerConnection: 2147483647 (default)  
2022-02-10 11:56:07.124 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultHttp2MaxHeaderListSize: 8192 (default)  
2022-02-10 11:56:07.125 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultHttp1MaxInitialLineLength: 4096 (default)  
2022-02-10 11:56:07.126 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultHttp1MaxHeaderSize: 8192 (default)  
2022-02-10 11:56:07.126 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultHttp1MaxChunkSize: 8192 (default)  
2022-02-10 11:56:07.127 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultUseHttp2Preface: true (default)  
2022-02-10 11:56:07.127 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultUseHttp1Pipelining: false (default)  
2022-02-10 11:56:07.127 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultBackoffSpec: exponential=200:10000,jitter=0.2 (default)  
2022-02-10 11:56:07.128 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.defaultMaxTotalAttempts: 10 (default)  
2022-02-10 11:56:07.128 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.routeCache.maximumSize=4096 (default)  
2022-02-10 11:56:07.128 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.compositeServiceCache.maximumSize=256 (default)  
2022-02-10 11:56:07.128 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.parsedPathCache.maximumSize=4096 (default)  
2022-02-10 11:56:07.129 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.headerValueCache.maximumSize=4096 (default)  
2022-02-10 11:56:07.129 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.headers.cacheHeaders: authority,scheme,method,accept-encoding,content-type (default)  
2022-02-10 11:56:07.130 INFO 18556 --- [main] c.l.a.c.Flags : com.linecorp.armeria.annotatedServiceExceptionVerbosity: unhandled (default)  
2022-02-10 11:56:07.132 INFO 18556 --- [main] c.l.a.c.Flags : /dev/epoll not available: java.lang.IllegalStateException: Only supported on Linux
```



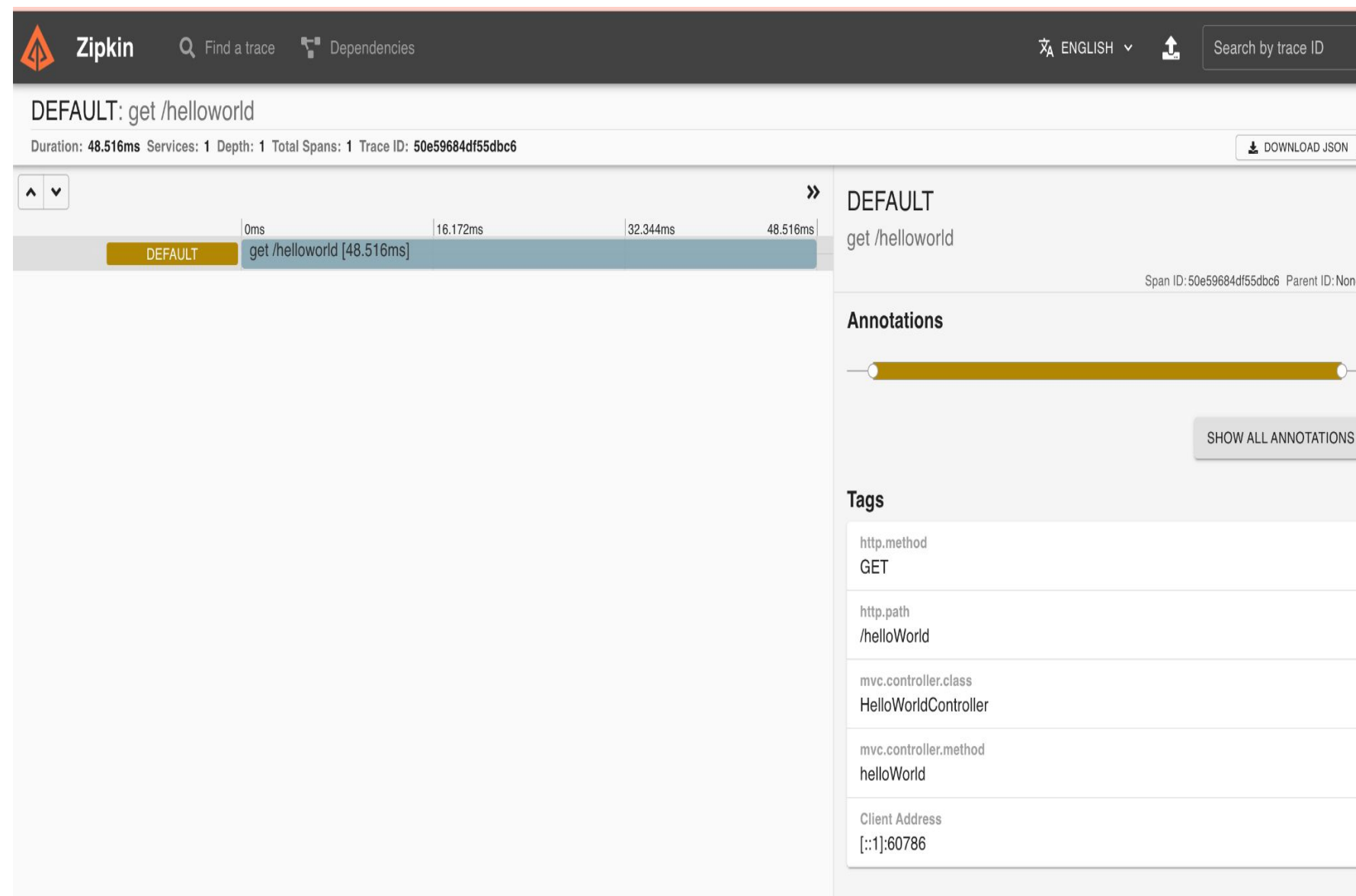
Si observamos estos logs, podremos identificar en qué puerto se está ejecutando este servidor.

```
: Serving HTTP at /[0:0:0:0:0:0:0:0]:9411 - http://127.0.0.1:9411/  
: Armeria server started at ports: {[0:0:0:0:0:0:0:0]:9411=ServerPort(/[0:0:0:0:0:0:0:0]:9411, [http])}
```

O más sencillo, podemos ingresar a **http://localhost:9411** desde el navegador y ver esta página.

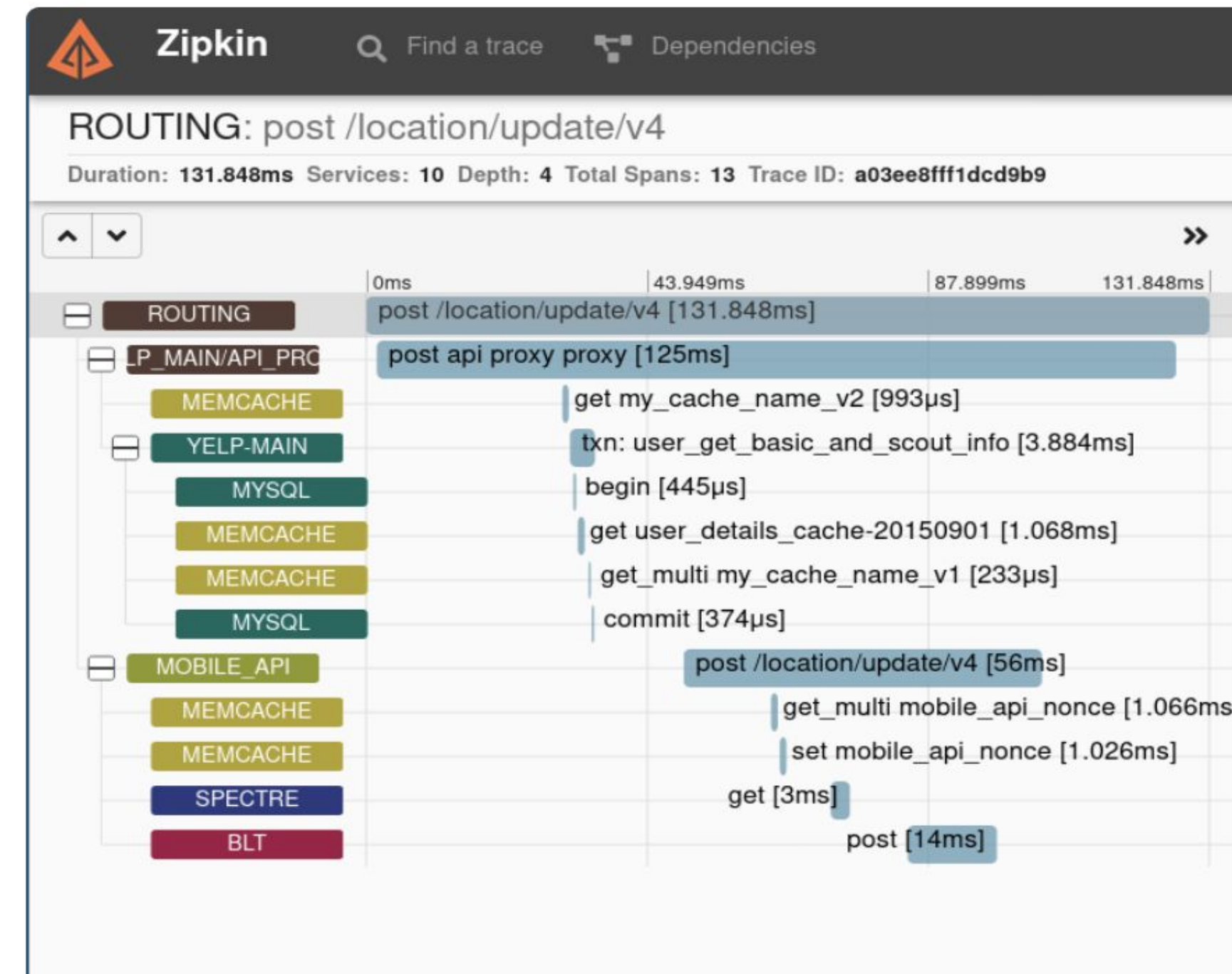
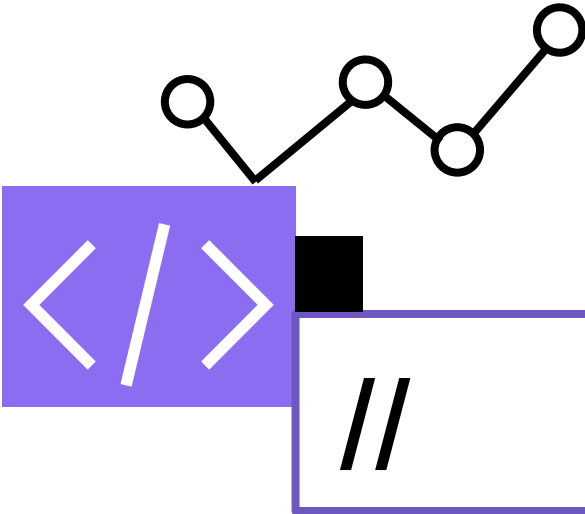


Monitoreando las invocaciones a los servicios



En este ejemplo se puede identificar el ciclo de vida de nuestra request. Esta entra al controller y sale del mismo con un “HelloWorld”, no hay pasos intermedios. Podemos apreciar el tiempo de entrada y salida en milisegundos.

El código de este ejemplo se encuentra [aquí](#).



En este ejemplo más completo, se puede identificar el ciclo de vida de una request. La primera línea es el enrutamiento (recordemos que una request nace cuando entra en el enrutamiento y muere cuando sale de él). Luego, las líneas subsiguientes son distintos procesos, microservicios y bases de datos que se llaman dentro del ciclo de vida de esa request. Cada uno con su inicio y fin, pero siempre dentro del ciclo que se marca en la primera línea.

¡Muchas gracias!