El nacimiento de Xorcery - Construir microservicios altamente performantes (1ra. Parte)

on **octubre 27, 2023** in **xorcery** with **2 comentarios**

Using dynamic DNS for service discovery - Rickard Öberg

JavaZone

18:46

Using dynamic DNS for service discovery – Rickard Öberg from JavaZone on Vimeo.

Hoy vengo a contarles uno de los proyectos mas prometedores que se ha empezado a cocinar en con el liderazgo de nuestro compañero Rickard Öberg y miembros del team en exoreaction.

Este proyecto se llama Xorcery. Una librería diseñada para ayudarnos a crecer nuestra solución basada en micro servicios que ya implementa clientes REST API y servidores, así como el streaming de data reactiva de forma minimalista y simple.

A continuación comparto las ideas que ha compartido con nuestro team Rickard Öberg.

Las APIs REST

Revisión del Problema del Lado del servidor

Al revisar las implementaciones recientes de API REST, estas cumplen con devolver datos estructurados y por lo general, en formato JSON. Y si esta estructura no esta definida por un esquema JSON, esta también podría ser un texto sin formato. Lo cual trae como consecuencia que los clientes tengan que hacer mucho trabajo para lograr una buena integración. **Asi que usar un formato JSON es la clave**, porque permite componentes reutilizables para no comenzar desde cero todo el tiempo. Vease la especificación JSON:API v1.1 que es en la que Xorcery se va a basar.

En la mayoría de las API REST no hay enlaces, lo que trae como consecuencia, el tener que implementar interminables estructuras de URL en los clientes, y si estos cambian se produce el caos. También hace imposible que los clientes puedan deducir que posibles acciones se pueden seguir, ya que no hay enlaces a formularios permitidos dado el estado actual del sistema.

Con lo anterior, se puede observar la falta de "descubribilidad" que es uno de los usos claves usar REST. El permitir que un cliente pueda cuando algo es posible o no basandose en la existencia de enlaces con relaciones conocidas.

Finalmente, **no tenemos formularios** como consecuencia de no tener enlaces en los datos transmitidos. Si tuvieramos formularios, los clientes podrían saber que acciones son posibles, pero, al no tenerlas teerminamos en implementaciones CRUD del API simplistas, donde la unica forma de descubrir que una acción no fue posible es enviar una solicitud POST y rechazarla.

Tener formularios vinculados a la estructura de datos permitiría a los clientes no solo saber cuándo son posibles ciertas acciones, sino que también facilitaría la realización de acciones que actualicen solo unos pocos campos en un recurso, en lugar de tener que depender de solicitudes PATCH donde el cliente "sabe" qué hacer.

Conclusión: El lado del servidor de las API REST es un completo desastre en este momento. **Esto se podría solucionar simplemente usando un formato multimedia que incluya todas estas características de forma nativa, como el formato JSON:API, combinado con el esquema JSON para definiciones de las partes personalizadas de la API**.

Solución: Crear un cliente sandbox totalmente automatizado que traduzca una API REST a HTML y permita a un desarrollador interactuar con ella en un navegador.

Revisión del Problema del Lado del cliente

La mayoría de los clientes de API REST estan orientados a solicitudes. Usted construye una URL, hace un GET o POST basado en la documentación del API y luego la envía. Y como cada API tiene su estructura JSON cada cliente es único y debe adaptarse al servicio que está creando. Este estilo de clientes ignora por completo lo que aprendimos de la web cuando se trata de diseño de clientes.

REST hace que el servidor sea sin estado, en el sentido de sesiones de conexión, pero lo hace poniendo esta responsabilidad en manos del cliente. Pero si todo lo que tiene es un cliente REST basado en solicitudes, ¿dónde está el estado? Esta olvidado.

La forma correcta de hacerlo es ver cada interacción de una API REST como una sesión, con estado. Tal como lo hace un navegador web. Una sesión de cliente REST debería permitir tres acciones principales:

- Extraer información de los datos estructurados
- Seguir enlaces
- Enviar Formularios

Esto hace que sea más fácil manejar la recuperación de errores, realizar múltiples pasos, encadenar flujos de procesos y manejar el caso en el que es posible que la interacción nunca termine, en caso de que el servidor no esté disponible durante toda la vida útil del cliente.

Estos son algunos de los problemas de las API REST, tanto en el cliente como en el servidor, que queremos solucionar con **Xorcery**, facilitando la creación de API REST basadas en los principios de la web y facilitando en crear clientes REST que tengan sesiones con estado que puedan abordar las falacias de la computación distribuida de una manera repetible y natural.

Descubrimiento distribuido, orquestación y conectividad

Al comienzo de este post encontrará un vídeo de la charla de Rickard en el JavaZone del 2023 sobre como Using dynamic DNS for service discovery - Rickard Öberg que explica este punto.

Una segunda area que necesita ayuda es la colaboración de servicios. Los servicios deben poder descubrirse entre sí y descubrir donde está disponible un servicio en particular que se necesita, mediante las relaciones definidas en la descripción del servicio y JSON;API y JSON Schema para luego interactuar con ellos.

Si queremos escalar un servidor para arriba o para abajo, debemos poder hacerlo sin un mecanismo externo. Debe estar intergrado en cada servidor para que los servicios puedan detectarse entre sí, decidir como organizar las colaboraciones y gestionar las conexiones entre servicios.

Es importante, hacer la distinción entre **servidores**, **contenedores** y **servicios** que realizar una tarea particular. Un servidor puede tener uno o veinte servicios, y la razón para hacer uno u otro debe basarse en las necesidades, no en un diagrama de diseño preconstruido que puede o no ser aplicable al sistema en ejecución. Si un servidor puede ejecutar todos los servicios, normalmente lo llamamos "**monolito**". En Xorcery tal vez sea mejor considerarlo como "**un sistema que aún no ha tenido suficiente presión como para requerir una división**".

Siempre que los servicios estén diseñados de tal manera que no estén codificados con la dependencia de que otros servicios se ubiquen o no en el mismo servidor, tenemos la libertad de elegir cuándo realizar dichas divisiones.
Si bien es posible ejecutar siempre servidores con un solo servicio cada uno, pero, cada uno con su propio contenedor Docker ejecutándose en la misma máquina virtual, esto no es particularmente rentable ni práctico. Es una complejidad innecesaria.

Al crear cuidadosamente métodos de descubrimiento y orquestación, que utilizan los principios de REST para permitir ignorar las ubicaciones físicas de los servidores, podemos hacer que sea más fácil comenzar poco a poco y crecer cuando sea necesario, a medida que aumentan las presiones sobre el sistema, en lugar de tener que diseñar por adelantado estas delineaciones.

Finalmente, analizando que el principal medio de comunicación entre servicios actualmente es, a traves, de API REST, en algunas situaciones, no son tan útiles. En particular cuando se trata de enviar flujos de datos entre sí. Ahí es mejor utilizar una forma de abstracción de transmisión y con el descubrimiento del patrón reactivo en especial los flujos reactivos. Entonces, concluimos que se debe permitir a los servicios publicar en flujos de datos reactivos y se suscriban y consuman estos flujos facilmente. Esto para nosotros es la herramienta que nos falta.

En Xorcery estamos abordando esto proporcionando una función incorporada de flujo reactivo de procesamiento por lotes, que se implementa mediante

websockets y la biblioteca Disruptor, para que sea más fácil posible la creación de flujos de datos que eventualmente progresen, que puedan procesarse por lotes y sean recuperables, en caso de errores.

Esta es una mejora importante con respecto al uso de endpoints de API REST para transferir flujos continuos de datos de un servicio a otro. Como se trata de una característica nativa, combina bien con las características de descubrimiento y orquestación para permitir que los servicios se encuentren fácilmente entre sí y decidan quién debe conectarse a qué. Al utilizar websockets y, posteriormente, aprovechar el **HTTP/3**, estamos reutilizando toda la infraestructura existente que necesitamos para estas conexiones, incluido el cifrado, la autenticación y la autorización SSL.

Service meshes (Mallas de Servicios)

Cuando un sistema de microservicios crece hasta "un cierto tamaño", inevitablemente llegará la pregunta de si se debe utilizar una malla de servicios o no. Esto se debe en parte a los problemas mencionados anteriormente con el descubrimiento, la orquestación y la conectividad, pero si los manejamos permitiendo que los servidores tengan esas características de forma nativa, ¿qué nos podría ofrecer?

Nos quedan problemas operativos, principalmente de registro, métricas y gestión de certificados para SSL. Dado que estamos tratando con el registro y las métricas del ecosistema Java, ambos tienen herramientas que podemos integrar con el servidor mismo y, utilizando las funciones de flujos reactivos, estos se pueden enviar de manera eficiente a servidores centralizados para su almacenamiento y análisis.

El problema restante relacionado con la administración de certificados lo podemos manejar creando un servicio que se ejecute en cada servidor Xorcery y que actualice periódicamente el certificado instalado localmente para habilitar SSL, tanto para nuestras API REST como para conexiones websocket, además de manejar algunos escenarios de autenticación y autorización donde los certificados de cliente son suficientes.

Con esto esperamos que no haya necesidad de los engorrosos service meshes que ya no serían necesarios. Menos piezas móviles, menos cosas que actualizar, menos cosas que puedan fallar y menos conexiones de red dan como resultado menos dolores de cabeza relacionados con las falacias de la computación distribuida.

El crecimiento de un sistema

La parte más compleja de la creación de sistemas de microservicios tiene que ver con afrontar el crecimiento. O más específicamente, lidiar con la presión.

¿Qué presiones tenemos en un sistema distribuido que podría ser útil analizar? Aquí hay algunas opciones posibles:

- Tamaño de la data
- Tamaño de la característica del sistema
- Operaciones por segundo
- Tamaño del equipo

Por ejemplo, con tamaños de datos pequeños podría ser suficiente tener un único servidor con los datos. A medida que crezca es posible que queramos replicarlo. A medida que crezca aún más, es posible que queramos fragmentarlo.

Con un equipo pequeño, es posible que queramos que todas las funciones se reúnan en un solo servicio. A medida que el equipo crece o se divide, es posible que deseemos dividir los servicios en consecuencia para facilitar la gestión del control de cambios.

Todas estas cosas son presiones que pueden cambiar la forma en que crecen nuestros sistemas. **Pero, por definición, no podemos conocer el futuro**. Entonces, ¿cómo diseñaría un sistema para manejar estas presiones sin saber cuáles serán? Cómo será todo de aquí a un año?, en cinco años, en diez años, etc.

Al leer artículos de ingenieros de Facebook y Twitter sobre cómo han evolucionado sus sistemas, queda claro que lidiar con los cambios en las presiones es el principal dolor de cabeza con el que todos tienen que lidiar.

Y así, la conclusión a la que llegamos, resulta tan obvia para mencionarla. ¿**Por qué no diseñar el sistema de manera que pueda hacer frente a estas presiones cuando se produzcan?** En lugar de intentar diseñar desde el principio lo que creemos que serán en el futuro, podríamos implementar mecanismos que nos permitan cambiar la estructura del sistema sobre la marcha, como reacción a estas presiones y no de manera proactiva. Aunque no sepamos lo que nos deparará el futuro, entonces sabremos que tenemos opciones sobre cómo lidiar con las diversas posibilidades cuando se presenten frente a nosotros.

Tengo la esperanza de que, al abordar todas las cuestiones antes mencionadas de manera inteligente, la capacidad de reaccionar ante las presiones se convierta en un resultado natural y no en algo que deba abordarse específicamente. **Este es el objetivo**

de diseño de todas las demás partes: facilitar el uso de Xorcery desde el principio hasta el final del ciclo de vida de una arquitectura de microservicios.

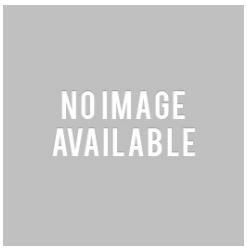
En los siguientes articulos ire compartiendo con todos ustedes detalles de la implementación y tutoriales de como usar Xorcery.

Enjoy!

José Díaz

Share:

Related Posts:



Xorcery is born - Build high-performance microservices

Xorcery usa Jetty & Jersey e implementación de mTLS y soporte a JWT

Xorcery implements Reactive Streams - Parte 1

Xorcery implementa Reactive Streams - Parte 1

Xorcery uses Jetty & Jersey and mTLS implementation and JWT support

Entrada más reciente Página Principal Entrada antigua

2 COMENTARIOS:

Daniel Amoretty 2:37 p. m., octubre 27, 2023

Muy buen aporte, muchas gracias.

Responder

Hector 12:00 p. m., octubre 28, 2023

Gracias por el aporte.

Responder

Para dejar un comentario, haz clic en el botón de abajo para acceder con Google.

ACCEDER CON GOOGLE

JOEDAYZ.PE

Cursos

POPULAR POSTS

Xorcery implementa Reactive Streams - Parte 1

¿Qué es Reactive Streams? Para poder entender como nos permite Xorcery trabajar

ABOUT

CATEGORIES

#github #java (1)

#guatejug (1)

con reactive streams , tenemos que saber que es Reactive ...

Paso de la Oración

Este 10, 11 y 12 de Septiembre ha sido un fin de semana enriquecedor para Miryan y para mí. Despues de varios años fuimos a la convivencia...

Blog Archive

git (1)

github (2)

google-format (1)

google-style (1)

grails (5)

groovy (3)

hangouts (1)

highchart-export-server (2)

huacho (1)

hudson (1)

hyperledger-composer (1)

hyperledger-fabric (1)

i-educa (1)

iBATIS (2)

icescrum (1)

informatica (1)

Intigas (1)

ITP_JAVA (1)

jakartaee (4)

jakartaee10 (1)

JasperReports (1)

java (3)

JavaCard (1)

JavaDayUNI (1)

JavaOne (1)

jhipster (2)

jmeter (1)

joedayz (46)

JOERP (4)

jpa (1)

jquery (1)

kafka (3)

kotlin (2)

Kubernetes (3)

lombok (1)

m2eclipse (1)

mac (2)

Matt Raible (1)

Maven (3)

microprofile (6)

microprofile-jwt jakartaee (2)

microprofile-jwt jdbc-realm jakartaee (1)

microprofile-jwt jdbc-realm payara (1)

microservicios (1)

Ministerio del Interior (1)

MJN (5)

móvil (1)

mysql (1)

namespaces (1)

navidad (1)

NET (4)

Nextel (1)

Novell (1)

ocjp (1)

Opentaps (2)

Oracle (1)

oraclecloud (1)

oraclefunctions (1)

oracleopenworld (1)

OSUM (1)

OSX (1)

p6spy (1)

Payara (5)

personal (1)

perujug jconfperu joedayz (2)

php (1)

play (1)

PMP (1)

podcasts (1)

PostgreSQL (8)

programacion (1)

pubsub (1)

PUCP (4)

quadim (2)

quarkus (1)

rackspace (1)

rails (2)

redis (1)

refactoring (2)

Reniec (1)

renovatebot (2)

Rider (1)

ruby (4)

rust (1)

scala (1)

SCD2010 (1)

SCJP (1)

Scrum (3)

Scrum evaluacion (1)

seminarios (1)

Setup (1)

SourceRepo (1)

spring (13)

spring 3.1 (1)

spring android (1)

spring mobile (1)

spring social (1)

spring-boot (9)

spring-boot-admin (2)

spring-cloud (1)

spring-cloud-config (2)

SpringCommunityDay (1)

SpringRoo (2)

springsource (1)

sqlserver (2)

start-up (1)

STS (1)

Subclipse (1)

Subversion (1)

SUN (1)

SUNAT (2)

synergyj (2)

Syscom (1)

Talleres (21)

Telefonica (1)

thedevconf (1)

thymeleaf (1)

Trac (1)

try-with-resources (1)

twitter (1)

Tye (1)

ubuntu (3)

UNI (3)

UNMSM (1)

UPC (1)

videos (1)

vimeo (1)

weblogic (2)

Workspace (1)

WPF (1)

xorcery (6)

xsd (1)

YaRetail (1)

YoMeQuedoEnCasa (1)

zuul (2)