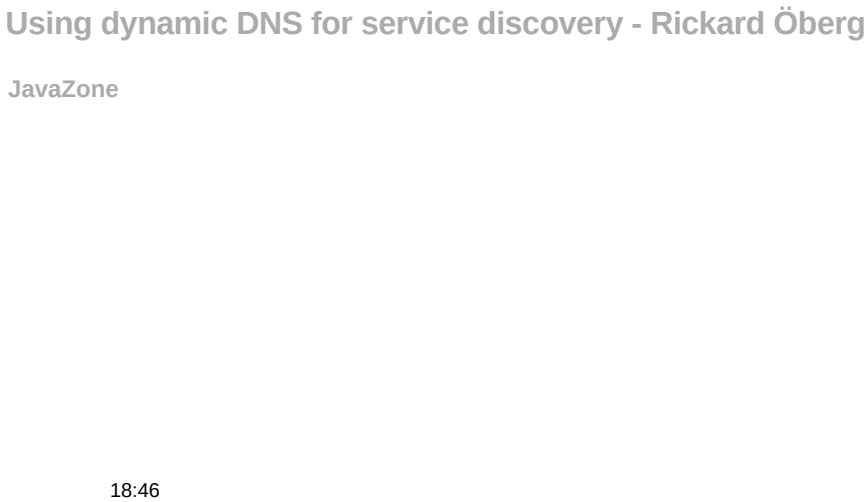
XORCERY IS BORN - BUILD HIGH-PERFORMANCE MICROSERVICES

on **octubre 27, 2023** in **xorcery** with **No hay comentarios.**



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

Today I come to tell you about one of the most promising projects that has begun to be cooked up with the leadership of our colleague Rickard Öberg and members of the team at exoreaction.

This project is called Xorcery. A library designed to help us grow our microservices-based solution that already implements REST API clients and servers, as well as reactive data streaming in a minimalist and simple way.

Below I share the ideas that Rickard Öberg has shared with our team.

REST APIs

Server Side Issue Review

When reviewing recent implementations of REST APIs, they comply with returning structured data, usually in JSON format. If this structure is not defined by a JSON schema, it could also be plain text. This means that clients have to do a lot of work to achieve a good integration. **So using a JSON format is the key because it allows reusable components so you don't start from scratch all the time**. See the JSON: API v1.1 specification, which is what Xorcery will be based on.

In most REST APIs there are no links, which results in having to implement endless URL structures in clients, and if they change, chaos ensues. It also makes it impossible for clients to deduce what possible actions can be taken since there are no links to forms allowed given the current state of the system.

Finally, we do not have forms as a consequence of not having links in the transmitted data. If we had forms, clients could know what actions were possible, but without them, we end up in simplistic CRUD implementations of the API, where the only way to discover that an action was not possible is to send a POST request and reject it.

Having forms bound to the data structure would allow clients to not only know when certain actions are possible but would also make it easier to

perform actions that update just a few fields in a resource, rather than having to rely on PATCH requests where the client "knows" what to do.

Conclusion: The server side of REST APIs is a complete mess right now. This could be solved by simply using a media format that includes all of these features natively, such as the JSON: API format, combined with the JSON schema for definitions of the custom parts of the API.

Solution: Create a fully automated sandbox client that translates a REST API to HTML and allows a developer to interact with it in a browser.

Client Side Problem Review

Most REST API clients are request-oriented. You construct a URL, do a GET or POST based on the API documentation, and then submit it. And since each API has its JSON structure, each client is unique and must adapt to the service it is creating. This style of clients completely ignores what we learned from the web when it comes to client design.

REST makes the server stateless, in the sense of connection sessions, but it does so by putting this responsibility in the hands of the client. But if all you have is a request-based REST client, where's the state? It is forgotten.

The correct way to do this is to view each interaction of a REST API as a session, with state. Just like a web browser does. A REST client session should allow three main actions:

- Extract information from structured data
- Follow links
- Submit Forms

This makes it easier to handle error recovery, perform multiple steps, chain process flows, and handle the case where the interaction may never finish, in case the server is unavailable for the entire life of the server. customer.

These are some of the problems with REST APIs, both on the client and on the server, that we want to solve with **Xorcery**, making it easier to create REST APIs based on web principles and making it easier to create REST clients that have stateful sessions that can address the fallacies of distributed computing in a repeatable and natural way.

Distributed discovery, orchestration and connectivity

At the beginning of this post you will find a video of Rickard's talk at JavaZone 2023 on Using dynamic DNS for service discovery - Rickard Öberg that explains this point.

A second area that needs help is service collaboration. Services must be able to discover each other and discover where a particular service that is needed is available, using the relationships defined in the service description and JSON;API and JSON Schema and then interact with them.

If we want to scale a server up or down, we must be able to do so without an external mechanism. It must be integrated into each server so that services can detect each other, decide how to organize collaborations, and manage connections between services.

It is important to make the distinction between servers, containers and services that perform a particular task. A server can have one or twenty services, and the reason for doing one or the other should be based on needs, not a pre-built design diagram that may or may not be applicable to the running system. If a server can run all services, we usually call it a "monolith". In Xorcery it's perhaps best thought of as “a system that hasn't had enough pressure yet to require a split.”

As long as services are designed in such a way that they are not coded with a dependency on whether or not other services are located on the same server, we are free to choose when to make such splits.

While it is possible to always run servers with a single service each, each with its own Docker container running in the same virtual machine, this is not particularly cost-effective or practical. It is unnecessary complexity.

By carefully creating discovery and orchestration methods, which use REST principles to allow ignoring the physical locations of servers, we can make it easier to start small and grow when necessary as pressures on the system increase. , instead of having to design these delineations in advance.

Finally, analyzing that the main means of communication between services currently is, through REST APIs, in some situations, they are not so useful. Particularly when it comes to sending data streams to each other. There it is better to use a form of transmission abstraction and with the discovery of the reactive pattern especially the reactive flows. So, we conclude that services should be allowed to publish to reactive data streams and subscribe to and consume these streams easily. This for us is the tool we are missing.

At Xorcery we are addressing this by providing a built-in batch reactive stream feature, implemented using websockets and the Disruptor library, to make it as easy as possible to create data streams that eventually progress, can be batched, and are recoverable, in case of errors.

This is a major improvement over using REST API endpoints to transfer continuous streams of data from one service to another. Since this is a native feature, it combines well with discovery and orchestration features to allow services to easily find each other and decide who should connect to what. By using websockets and then leveraging HTTP/3, we are reusing all of the existing infrastructure we need for these connections, including SSL encryption, authentication, and authorization.

Service meshes

When a microservices system grows to "a certain size", the question will inevitably arise whether to use a service mesh or not. This is partly due to the aforementioned issues with discovery, orchestration and connectivity, but if we handle them by allowing servers to have those features natively, what could that offer us?

We are left with operational issues, mainly logging, metrics, and certificate management for SSL. Since we are dealing with Java ecosystem logging and metrics, both have tools that we can integrate with the server itself and using the reactive streams features these can be efficiently sent to centralized servers for storage and analysis.

The remaining problem related to certificate management we can handle by creating a service that runs on each Xorcery server and periodically updates the locally installed certificate to enable SSL, both for our REST APIs and websocket connections, in addition to handling some authentication and authorization where client certificates are sufficient.

With this we hope that there will be no need for cumbersome service meshes that would no longer be necessary. Fewer moving parts, fewer things to update, fewer things to fail, and fewer network connections result in fewer headaches related to the fallacies of distributed computing.

The growth of a system

The most complex part of building microservices systems has to do with dealing with growth. Or more specifically, dealing with pressure.

What pressures do we have in a distributed system that might be useful to analyze? Here are some possible options:

- Data size
- System Feature Size
- Operations per second
- Team size

For example, with small data sizes it might be sufficient to have a single server with the data. As it grows we may want to replicate it. As it grows even more, we may want to break it up.

With a small team, we may want all functions to be brought together in a single service. As the team grows or splits, we may want to split services accordingly to make change control easier to manage.

All of these things are pressures that can change the way our systems grow. But, by definition, we cannot know the future. So how would you design a system to handle these pressures without knowing what they will be? What will everything be like a year from now?, in five years, in ten years, etc.

Reading articles from Facebook and Twitter engineers about how their systems have evolved, it's clear that dealing with changing pressures is the biggest headache everyone has to deal with.

And so, the conclusion we reached is too obvious to mention. Why not design the system so that it can cope with these pressures when they occur? Instead of trying to design from the beginning what we think they will be in the future, we could implement mechanisms that allow us to change the structure of the system on the fly, in reaction to these pressures rather than proactively. Even if we don't know what the future holds, then we will know that we have options for how to deal with the various possibilities when they present themselves in front of us.

It is my hope that by addressing all of the above issues intelligently, the ability to react to pressures becomes a natural outcome and not something that needs to be specifically addressed. This is the design goal of all other parts: to make it easy to use Xorcery from the beginning to the end of the lifecycle of a microservices architecture.

In the following articles I will share with all of you implementation details and tutorials on how to use Xorcery.

Enjoy!

José Díaz

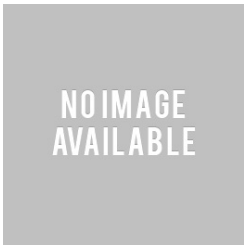
Share:

Related Posts:

Xorcery uses Jetty & Jersey and mTLS implementation and JWT support

Xorcery implements Reactive Streams - Parte 1

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



Xorcery is born - Build high-performance microservices

Xorcery implementa Reactive Streams - Parte 1

Entrada más reciente Página Principal Entrada antigua

0 comentarios:

Publicar un comentario

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

ACCEDER CON GOOGLE

JoeDayz.pe

Cursos

About

Popular Posts

Xorcery implementa Reactive Streams - Parte 1

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

Categories

#github #java (1)

#guatejug (1)

#historiasdeprogramador (1)

aaii (1)

academia web (2)

acr (1)

para mí. Despues de varios años fuimos a la convivencia...

Blog Archive

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)