Microservicios Java: una guía práctica

Última actualización el 12 de diciembre de 2019 - 0 comentarios

Puede usar esta guía para comprender qué son los microservicios de Java, cómo los diseña y construye. Además: un vistazo a las bibliotecas de microservicios de Java y preguntas comunes.

[**Nota del editor** : con casi 7,000 palabras, probablemente no desee intentar leer esto en un dispositivo móvil. Marcarlo y volver más tarde.]

Microservicios Java: los fundamentos

To get a real understanding of Java microservices, it makes sense to start with the very basics: The infamous Java monolith, what it is and what its advantages or disadvantages are.

What is a Java monolith?

Imagine you are working for a bank or a fintech start-up. You provide users a mobile app, which they can use to open up a new bank account.

In Java code, this will lead to a controller class that looks, *simplified*, like the following.

```
@Controller
class BankController {

    @PostMapping("/users/register")
    public void register(RegistrationForm form) {
        validate(form);
        riskCheck(form);
        openBankAccount(form);
        // etc..
    }
}
```

You'll want to:

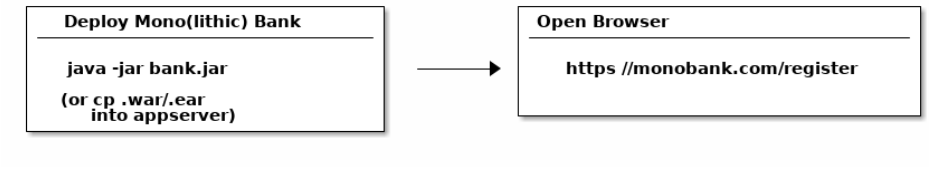
1. Validate the registration form.

2. Do a risk check on the user's address to decide if you want to give him a bank account or not.

3. Open up the bank account

Your BankController class will be packaged up, with all your other source code, into a bank.jar or bank.war file for deployment: A good, old monolith, containing all the code you need for your bank to run. (As a rough pointer, *initially* your .jar/.war file will have a size in the range of 1-100MB).

On your server, you then simply run your .jar file - that's all you need to do to deploy Java applications.



What is the problem with Java monoliths?

At its core, there's nothing wrong with a Java monolith. It is simply that project experience has shown that, if you:

1. Let many different programmers/teams/consultancies...

2. Work on the same monolith under high pressure and unclear requirements...

3. For a couple of years...

Then your small bank.jar file, turns into a gigabyte large code monster, that everyone fears deploying.

How to get the Java monolith smaller?

This naturally leads to the question of how to get the monolith smaller. For now, your bank.jar runs in one JVM, one process on one server. Nothing more, nothing less.

Now you could come up with the idea to say: Well, the risk check service is being used by other departments in my company and it doesn't *really* have anything to do with my Mono(lithic) Bank *domain*, so we could try and cut it out of the monolith and deploy it as its own product, or more technically, run it as its own Java process.

What is a Java Microservice?

In practical terms, this means that instead of calling the riskCheck() method inside your BankController, you will move that method/bean with all its helper

classes to its own Maven/Gradle project, put it under source control and deploy it independently from your banking monolith.

That whole extraction process does not make your new RiskCheck module a *microservice* per se and that is because the definition of microservices is open for interpretation (which leads to a fair amount of discussion in teams and companies).

- Is it micro if it only has 5-7 classes inside?
- Are 100 or 1000 classes still micro?
- Has it even got anything to do with the number of classes?

Instead of theorizing about it, we'll keep things pragmatic and do two things:

1. Call *all separately deployable* services microservices - independent of size or domain boundaries.
2. Focus on the important topic of inter-service communication, because your microservices need ways to talk to each other.

So, to sum up: Before you had one JVM process, one Banking monolith. Now you have a banking monolith JVM process and a RiskCheck microservice, which runs in its own JVM process. And your monolith now has to call that microservice for risk checks.

How do you do that?

How to communicate between Java Microservices?

You basically have two choices: *synchronous communication* or *asynchronous communication*.

(HTTP)/REST - Synchronous Communication

Synchronous microservice communication is usually done via HTTP and REST-like services that return XML or JSON - though this is by no means required (have a look at Google's Protocol Buffers for example).

Use REST communication when you need an immediate response, which we do in our case, as risk-checking is mandatory before opening an account: No risk check, no account.

Tool-wise, check out Which libraries are the best for synchronous Java REST calls?.

Messaging - Asynchronous Communication

Asynchronous microservice communication is usually done through messaging with a JMS implementation and/or with a protocol like AMQP. Usually, because the number of, for example, email/SMTP-driven integrations is not to be underestimated in practice.

Use it when you do not need an immediate response, say the users presses the 'buy-now' button and you want to generate an invoice, which certainly does not have to happen as part of the user's purchase request-response cycle.

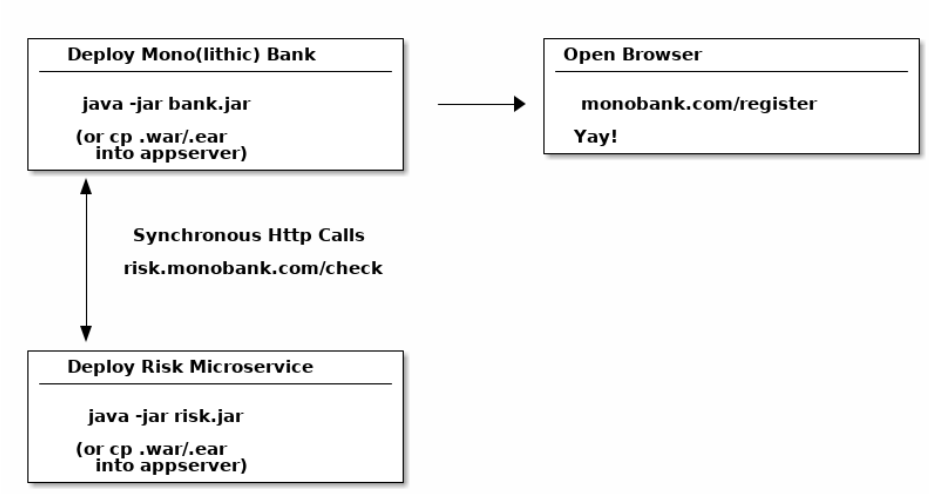
Tool-wise, check out Which brokers are the best for asynchronous Java messaging?.

Example: Calling REST APIs in Java

Assuming we chose to go with synchronous microservice communication, our Java code from above would then look something like this on a low-level. Low-level, because for microservice communication you usually create client libraries, that abstract the actual HTTP calls away from you.

```
@Controller
class BankController {

    @Autowired
    private HttpClient httpClient;

    @PostMapping("/users/register")
    public void register(RegistrationForm form) {
        validate(form);
        httpClient.send(riskRequest,
responseHandler());
        setupAccount(form);
        // etc..
    }
}
```

Looking at the code it becomes clear, that you now must deploy two Java (micro)services. Your Bank and your RiskCheck service. You are going to end up with two JVMs, two processes. The graphic from before will look like this:



That’s all you need to develop a Java Microservices project: Build and deploy smaller pieces, instead of one large piece.

But that leaves the question: How *exactly* do you cut or setup those microservices? What are these smaller pieces? What is the right size?

Let’s do a reality check.

Java Microservice Architecture

In practice, there’s various ways that companies try to design or architect Microservice projects. It depends on if you are trying to turn an existing monolith into a Java microservices project, or if you are starting out with a new Greenfield project.

From Monolith to Microservices

One rather organic idea is to break microservices out of an existing monolith. Note, that "micro" here does not actually mean that the extracted services themselves will, indeed, be micro - they could still be quite large themselves.

Let’s look at some theory.

The Idea: Break a Monolith into Microservices

Legacy projects lend themselves to a microservices approach. Mainly, for three reasons:

1. They are often hard to maintain/change/extend.
2. Everyone, from developers, ops to management ~~wants to make things simpler~~ wants stuff to be simpler.
3. You have (somewhat) clear domain boundaries, that means: You know what your software is supposed to do.

This means you can have a look at your Java bank monolith and try to split it along *domain boundaries* - a sensible approach.

- You could conclude that there should be an 'Account Management' microservice, that handles user data like names, addresses, phone numbers.
- Or the aforementioned 'Risk Module', that checks user risk levels and which could be used by many other projects or even departments in your company.

- Or an invoicing module, that sends out invoices via PDF or actual mail.

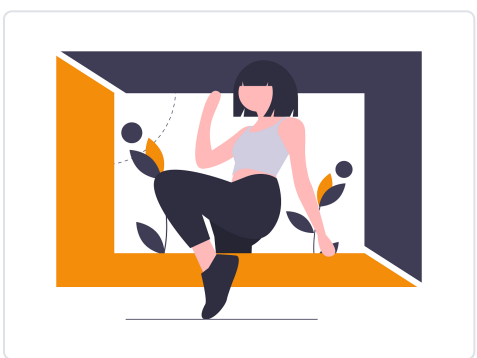
Reality: Let someone else do it

While this approach definitely looks good on paper and UML-like diagrams, it has its drawbacks. Mainly, you need very strong technical skills to pull it off. Why?

Because there is a huge difference between *understanding* that it would be a good thing to extract the, say, highly coupled account management module out of your monolith and *doing it* (properly).

Most enterprise projects reach the stage where developers are scared to, say, upgrade the 7-year-old Hibernate version to a newer one, which is just a library update but a fair amount of work trying to make sure not to break anything.

Those same developers are now supposed to dig deep into old, legacy code, with unclear database transaction boundaries and extract well-defined microservices? Possible, but often a real challenge and not solvable on a whiteboard or in architecture meetings.



This is already the first time in this article, where a quote from @simonbrown on Twitter fits in:

> I'll keep saying this ... if people can't build monoliths properly, microservices won't help.
>
> — Simon Brown

Greenfield Project Microservice Architecture

Things look a bit different when developing new, greenfield Jav aprojects. Now, those three points from above look a bit different:

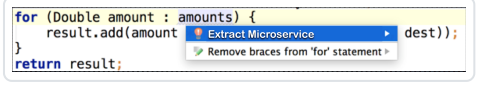
1. You are starting with a clean slate, so there’s no old baggage to maintain.
2. Developers would like things to stay simple in the future.

3. The issue: You have a much foggier picture of domain boundaries: You don't know what your software is actually supposed to do (hint: agile ;))

This leads to various ways that companies try and tackle greenfield Java microservices projects.

Technical Microservice Architecture

The first approach is the most obvious for developers, although the one highly recommended against. Props to Hadi Hariri for coming up with the "Extract Microservice" refactoring in IntelliJ.



While the following example is oversimplified to the extreme, actual implementations seen in real projects are unfortunately not too far off.

Before Microservices

```
@Service
class UserService {

    public void register(User user) {
        String email = user.getEmail();
        String username =  email.substring(0,
email.indexOf("@"));
        // ...
    }
}
```

With a substring Java microservice

```
@Service
class UserService {

    @Autowired
    private HttpClient client;

    public void register(User user) {
        String email = user.getEmail();
        // now calling the substring microservice
via http
        String username =
httpClient.send(substringRequest(email),
responseHandler());
        // ...
    }
}
```

So, you are essentially wrapping a Java method call into a HTTP call, with no obvious reasons to do so. One reason, however, is: Lack of experience and trying to force a Java microservices approach.

Recommendation: Don't do it.

Workflow Oriented Microservice Architecture

The next common approach is, to module your Java microservices after your workflow.

Real-Life example: In Germany, when you go to a (public) doctor he needs to record your appointment in his health software CRM.

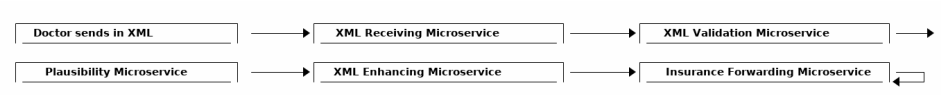
To get paid from the insurance he will send in your treatment data and that of all other patients he treated to an intermediary via XML.

The intermediary will have a look at that XML file and (simplified):

1. Try and validate the file that it is proper XML
2. Try and validate it for plausibility: did it make sense that a 1 year old got three tooth cleanings in a day from a gynecologist?
3. Enhance the XML with some other bureaucratic data
4. Forward the XML to the insurance to trigger payments
5. And model the whole way back to the doctor, including a "success" message or "please re-send that data entry again - once it makes sense"

If you now try and model this workflow with, you will end up with at least six Java microservices.

Note: Communication between Microservices is irrelevant in this example, but could well be done asynchronously with a message broker like RabbitMQ, as the doctor does not get immediate feedback, anyway.



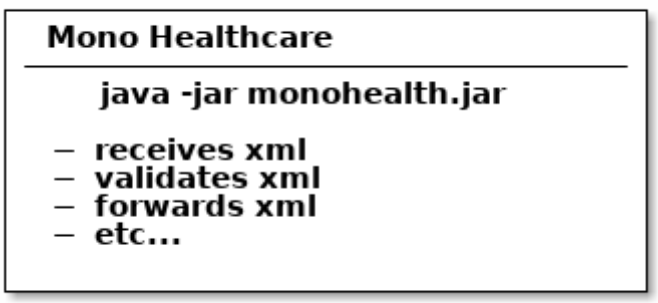
Again, this is something that looks good on paper, but immediately leads to several questions:

- Do you feel the need to deploy six applications to process 1 xml file?
- Are these microservices *really* independent from each other? They can be deployed independently from each other? With different versions and API schemes?
- What does the plausibility-microservice do if the validation microservice is down? Is the system then still running?

- Do these microservices now share the same database (they sure need some common data in a database table) or are you going to take the even bigger hammer of giving them all their own database?

- And a ton of other infrastructure/operations questions.

Interestingly, for some architects the above diagram reads simpler, because every service now has its exact, well-defined *purpose*. Before, it looked like this scary monolith:



While arguments can be about the simplicity of those diagrams, you now definitely have these *additional* operational challenges to solve:

- Don’t just need to deploy one application, but at least six.

- Maybe even databases, depending on how far you want to take it.

- Have to make sure that every system is online, healthy and working.

- Have to make sure that your calls between microservices are actually *resilient* (see How to make a Java microservice resilient?)

- And everything else this setup implies - from local development setups to integration testing

Recommendation:

Unless:

- you are Netflix (you are not)...

- you have super-strong operation skills: you open up your development IDE, which triggers a chaos monkey that DROPs your production database which easily auto-recovers in 5seconds

- or you feel like @monzo in giving 1500 microservices a try, simply because you can.

→ Don’t do it.

In less hyperbole, though.

Trying to model microservices after domain boundaries is a very sensible approach. But a domain boundary (say user management vs invoicing) does not mean taking a single workflow and splitting it up into its tiniest individual pieces (receive XML, validate XML, forward XML).

Hence, whenever you are starting out with a new Java microservices project and the domain boundaries are still very vague, try to keep the size of your microservices on the *lower end*. You can always add more modules later on.

And make sure that you have exceptionally strong DevOps skills across your team/company/division to support your new infrastructure.

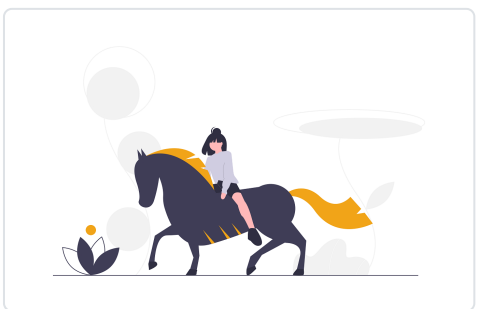
Polyglot or Team Oriented Microservice Architecture

There is a third, almost libertarian approach to developing microservices: Giving your teams or even individuals the possibility to implement user stories with as many languages or microservices they want (marketing term: polyglot programming).

So the XML Validation service above could be written in Java, while the Plausibility Microservice is written in Haskell (to make it mathematically sound) and the Insurance Forwarding Microservice should be written in Erlang (because it *really* needs to scale ;)).

What might look like fun from a developer's perspective (developing a perfect system with your perfect language in an isolated setting) is basically never what an organization wants: Homogenization and standardization.

That means a relatively standardized set of languages, libraries and tools so that other developers can keep maintaining your Haskell microservice in the future, once you are off to greener pastures.



What's interesting: Historically standardization went way too far. Developers in big Fortune 500 companies were sometimes not even allowed to use Spring, because it was 'not in the company's technology blueprint'. But going full-on polyglot is pretty much the same thing, just the other side of the same coin.

Recommendation : If you are going polyglot, try smaller diversity in the same programming language *eco-system*. Example: Kotlin and Java (JVM-based with 100% compatibility between each other), not Haskell and Java.

Deploying and Testing Java Microservices

It helps to have a quick look back at the basics, mentioned at the beginning of this article. Any server-side Java program, hence also any microservice, is just a .jar/.war file.

And there's this one great thing about the Java ecosystem, or rather the JVM: You write your Java code once, you can run it basically on any operating system you want provided you didn't compile your code with a newer Java version than your target JVM's versions).

It's important to understand this, especially when it comes to topics like Docker, Kubernetes or (shiver) *The Cloud*. Why? Let's have a look at different deployment scenarios:

A bare minimum Java microservice deployment example

Continuing with the bank example, we ended up with our monobank.jar file (the monolith) and our freshly extracted riskengine.jar (the first microservice).

Let's also assume that both applications, just like any other application in the world, need a .properties file, be it just the database url and credentials.

A bare minimum deployment could hence consist of just two directories, that look roughly like this:

```
-r-r------ 1 ubuntu ubuntu     2476 Nov 26 09:41
application.properties
-r-x------ 1 ubuntu ubuntu 94806861 Nov 26 09:45
monobank-384.jar

ubuntu@somemachine:/var/www/www.monobank.com/java$
java -jar monobank-384.jar


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
...
```

```
-r-r------ 1 ubuntu ubuntu     2476 Nov 26 09:41
application.properties
-r-x------ 1 ubuntu ubuntu 94806861 Nov 26 09:45
risk-engine-1.jar

ubuntu@someothermachine:/var/www/risk.monobank.com/ja
 java -jar risk-engine-1.jar


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
...
```

This leaves open the question: How do you get your .properties and .jar file onto the server?

Unfortunately, there's a variety of *alluring* answers to that question.

How to use Build Tools, SSH & Ansible for Java microservice deployments

The boring, but perfectly fine answer to Java microservice deployments is how admins deployed *any* Java server-side program in companies in the past 20 years. With a mixture of:

- Your favorite build tool (Maven, Gradle)
- Good old SSH/SCP for copying your .jars to servers
- Bash scripts to manage your deployment scripts and servers
- Or even better: some Ansible scripts.

If you are not fixated on creating a breathing cloud of ever auto-load-balancing servers, chaos monkeys nuking your machines, or the warm and fuzzy-feeling of seeing ZooKeeper's leader election working, then this setup will take you very far.

Oldschool, boring, but working.

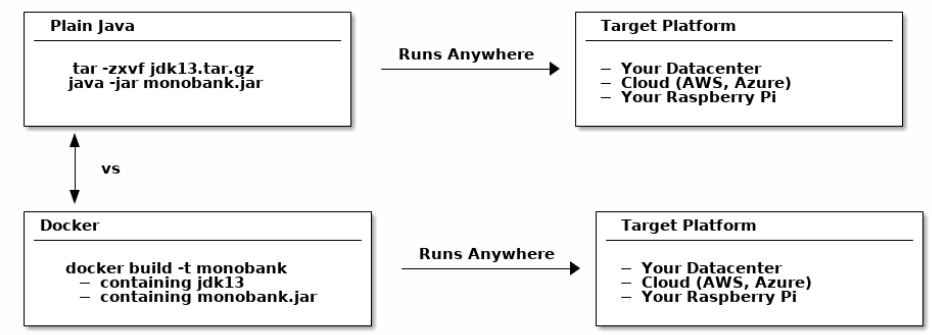
How to use Docker for Java microservice deployments

Back to the alluring choices. A couple of years ago, Docker or the topic of containerization hit the scene.

If you have no previous experience with it, this is what it is all about for end-users or developers:

1. A container is (simplified) like a good old virtual machine, but more lightweight. Have a look at this Stackoverflow answer to understand what lightweight means in this context.

2. A container guarantees you that it is portable, it runs anywhere. Does this sound familiar?



Interestingly, with the JVM's portability and backwards compatibility this doesn't sound like major benefits. You could just download a JVM.zip on any server, Raspberry Pi (or even mobile phone), unzip it and run any .jar file you want.

It looks a bit different for languages like PHP or Python, where version incompatibilities or deployment setups historically were more complex.

Or if your Java application depends on a ton of other installed services (with the right version numbers): Think a database like Postgres or key-value store like Redis.

So, Docker's primary benefit for Java microservices, or rather Java applications lies in:

- Setting up homogenized test or integration environments, with tools like Testcontainers.
- Making complex deployables "simpler" to install. Take the Discourse forum software. You can install it with one Docker image, that contains everything you need: From the Discourse software written in Ruby, to a Postgres database, to Redis and the kitchen sink.

If your deployables look similar or you want to run a nice, little Oracle database on your development machine, give Docker a try.

So, to sum things up, instead of simply scp'ing a .jar file, you will now:

- Bundle up your jar file into a Docker image
- Transfer that docker image to a private docker registry
- Pull and run that image on your target platform
- Or scp the Docker image directly to your prod system and run it

How to use Docker Swarm or Kubernetes for Java microservice deployments

Let’s say you are giving Docker a try. Every time you deploy your Java microservice, you now create a Docker image which bundles your .jar file. You have a couple of these Java microservices and you want to deploy these services to a couple of machines: a *cluster*.

Now the question arises: How do you manage that cluster, that means run your Docker containers, do health checks, roll out updates, scale (brrrr)?

Two possible answers to that question are Docker Swarm and Kubernetes.

Going into detail on both options is not possible in the scope of this guide, but the reality takeaway is this: Both options in the end rely on you writing YAML files (see Not a question: Yaml Indentation Tales) to manage your cluster. Do a quick search on Twitter if you want to know what feelings that invokes in practice.

So the deployment process for your Java microservices now looks a bit like this:

- Setup and manage Docker Swarm/Kubernetes
- Everything from the Docker steps above
- Write and execute YAML until ~~your eyes bleed~~ things are working

How to test Java microservices

Let’s assume you solved deploying microservices in production, but how do you integration test your n-microservices during development? To see if a complete workflow is working, not just the single pieces?

In practice, you’ll find three different ways:

1. With a bit of extra work (and if you are using frameworks like Spring Boot), you can wrap all your microservices into one launcher class, and boot up all microservices with one Wrapper.java class - depending if you have enough memory on your machine to run all of your microservices.
2. You can ~~try to~~ replicate your Docker Swarm or Kubernetes setup locally.
3. Simply don’t do integration tests locally anymore. Instead have a dedicated DEV/TEST environment. It’s what a fair numbers of teams actually do, succumbing to the pain of local microservice setups.

Furthermore, in addition to your Java microservices, you'll likely also need an up and running message broker (think: ActiveMQ or RabbitMQ) or maybe an email server or any other messaging component that your Java microservices need to communicate with each other.

This leads to a fair amount of underestimated complexity on the DevOps side. Have a look at Microservice Testing Libraries to mitigate some of that pain.

In any case, this complexity leads us to common Microservice issues:

Common Java Microservice Questions

Let's have a look at Java specific microservices issues, from more abstract stuff like resilience to specific libraries.

How to make a Java microservice resilient?

To recap, when building microservices, you are essentially swapping out JVM method calls with synchronous HTTP calls or asynchronous messaging.

Whereas a method call execution is basically guaranteed (with the exception of your JVM exiting abruptly), a network call is, by default, unreliable.

It could work, it could also not work for various reasons: From the network being down or congested, to a new firewall rule being implemented to your message broker exploding.

To see what implications that has, let's have a look at an exemplary *BillingService* example.

HTTP/REST Resilience Patterns

Say customers can buy e-books on your companies' website. For that, you just implemented a billing microservice, that your webshop can call to generate the actual PDF invoices.

For now, we'll do that call synchronously, via HTTP. (It would make more sense to call that service asynchronously, because PDF generation doesn't have to be instant from a user's perspective. But we want to re-use this very example in the next section and see the differences.)

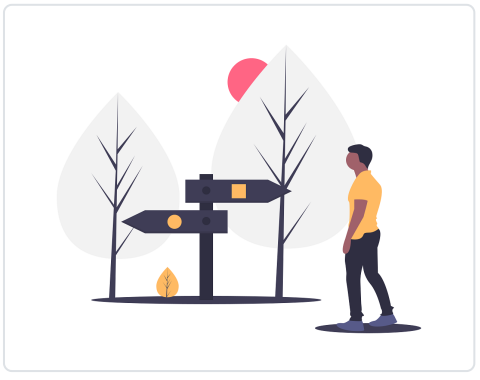
```
@Service
class BillingService {

    @Autowired
    private HttpClient client;

     public void bill(User user, Plan plan) {
        Invoice invoice = createInvoice(user,
plan);

httpClient.send(invoiceRequest(user.getEmail(),
invoice), responseHandler());
        // ...
    }
}
```

Think about what kind of possible results that HTTP call could have. To generalize, you will end up with three possible results:

1. **OK**: The call went through and the invoice got created successfully.
2. **DELAYED**: The call went through but took an unusually long amount of time to do so.
3. **ERROR**: The call did not go through, maybe because you sent an incompatible request, or the system was down.

Handling errors, not just the happy-cases, is expected for any program. It is the same for microservices, even though you have to take extra care to keep all of your deployed API versions compatible, as soon as you start with individual microservice deployments and releases.

And if you want to go full-on chaos-monkey, you will also have to live with the possibility that your servers just get nuked during request processing and you might want the request to get re-routed to another, working instance.



An interesting 'warning' case is the delayed case. Maybe the responding's microservice hard-disk is running full and instead of 50ms, it takes 10 seconds to respond. This can get even more interesting when you are experiencing a certain load, so that the unresponsiveness of your BillingService starts

cascading through your system. Think of a slow kitchen slowly starting the block all the waiters of a restaurant.

This section obviously cannot give in-depth coverage on the microservice resilience topic, but serves as a reminder for developers that this is something to actually *tackle* and *not ignore* until your first release (which from experience, happens more often than it should)

A popular library that helps you think about latency and fault tolerance, is Netflix's Hystrix. Use its documentation to dive more into the topic.

Messaging Resilience Patterns

Let's take a closer look at asynchronous communication. Our BillingService code might now look something like this, providing we use Spring and RabbitMQ for our messaging.

To create an invoice, we now send a message to our RabbitMQ message broker, which has some workers waiting for new messages. These workers create the PDF invoices and send them out to the respective users.

```
@Service
class BillingService {

    @Autowired
    private RabbitTemplate rabbitTemplate;

     public void bill(User user, Plan plan) {
        Invoice invoice = createInvoice(user,
plan);
        // converts the invoice to,for example,
json and uses it as the message's body
        rabbitTemplate.convertAndSend(exchange,
routingkey, invoice);
        // ...
    }
}
```

Now the potential error cases look a bit different, as you don't get immediate OK or ERROR responses anymore, like you did with synchronous HTTP communication. Instead, you'll roughly have these three error cases:

1. Was my message delivered and consumed by a worker? Or did it get lost? (The user gets no invoice).

2. Was my message delivered just once? Or delivered more than once and only processed exactly once? (The user would get multiple invoices).

3. Configuration: From "Did I use the right routing-keys/exchange names", to is "my message broker setup and maintained correctly or are its queues overflowing?" (The user gets no invoice).

Again, it is not in the scope of this guide to go into detail on every single asynchronous microservice resilience pattern. More so, it is meant as pointers in the right direction, especially as it also depends on the actual messaging technology you are using. Examples:

- If you are using JMS implementations, like ActiveMQ, you could want to trade speed for the guarantees of two-phase (XA) commits.
- If you are using RabbitMQ you at least want to make sure to have read and understood this guide and then think hard about acknowledgements, confirms and message reliability in general.
- And also have someone with experience in setting up e.g. Active or RabbitMQ servers and configuring them properly, especially when used in combination with clustering and Docker (network splits, anyone? ;))

Which Java microservice framework is the best?

On one hand you have established and very popular choices like Spring Boot, which makes it very easy to build .jar files that come with an embedded web server like Tomcat or Jetty and that you can immediately run anywhere.

Recently though, and partially inspired by parallel developments like reactive programming, Kubernetes or GraalVM, a couple of, dedicated microservice frameworks have arisen.

To name a few: Quarkus, Micronaut, Vert.x, Helidon.

In the end, you will have to make your own choice, but this article can give some, maybe unconventional, guidance:

With the exception of Spring Boot, all microservices frameworks generally market themselves as *blazingly fast*, *monumentally quick startup time*, *low memory footprint*, able to *scale indefinitely*, with impressive graphs comparing themselves against the Spring Boot behemoth or against each other.

This is clearing hitting a nerve with developers who are maintaining legacy projects that sometimes take minutes to boot-up or cloud-native developers who

want to start-stop as many micro-containers as ~~they now can or want~~ they need in 50ms.

The issue, however, is that (artificial) bare metal startup times and re-deploy times barely have an effect on a project's overall success, much less so than a strong framework ecosystem, strong documentation, community and strong developer skills.

You'll have to look at it this way.

If until now:

- You let your ORMs run rampage and generate hundreds of queries for simple workflows.
- You needed endless gigabytes for your moderately complex monolith to run.
- You added so much code and complexity that (disregarding potentially slow starters like Hibernate) your application now need minutes to boot up.

Then adding *additional* Microservice challenges (think: resilience, network, messaging, DevOps, infrastructure) *on top* will have a *much heavier* impact on your project, than booting up an empty hello world. And for hot redeploys during development, you finally might want to look into solutions like JRebel or DCEVM.

To go back to Simon Brown's quote: If people cannot build (fast & efficient) monoliths, they will be having a hard time building (fast & efficient) microservices - no matter the framework.

So, choose your framework wisely.

Which libraries are the best for synchronous Java REST calls?

A los aspectos más prácticos de llamar a las API REST de HTTP. En el aspecto técnico de bajo nivel, probablemente terminará con una de las siguientes bibliotecas de cliente HTTP:

HttpClient propio de Java (desde Java 11), HttpClient de Apache u OkHttp .

Tenga en cuenta que estoy diciendo 'probablemente' aquí porque también hay miles de millones de otras formas, desde los buenos clientes JAX-RS hasta los clientes modernos de WebSocket .

En cualquier caso, existe una tendencia hacia la generación de clientes HTTP, en lugar de perder el tiempo con las llamadas HTTP. Para eso, desea echar un vistazo al proyecto OpenFeign y su documentación como punto de partida para futuras lecturas.

¿Qué agentes son los mejores para la mensajería asincrónica de Java?

Comenzando con la mensajería asincrónica, es probable que termines con ActiveMQ (Classic o Artemis) , RabbitMQ o Kafka . Nuevamente, esta es solo una elección popular.

Sin embargo, aquí hay un par de puntos aleatorios:

- ActiveMQ y RabbitMQ son corredores de mensajes tradicionales y completos. Esto significa un corredor bastante inteligente y consumidores tontos.
- Históricamente, ActiveMQ tenía la ventaja de una fácil integración (para pruebas), que se puede mitigar con las configuraciones RabbitMQ / Docker / TestContainer
- Kafka *no* es un corredor tradicional. Es todo lo contrario, esencialmente un almacén de mensajes relativamente 'tonto' (piense en un archivo de registro) que necesita consumidores más inteligentes para el procesamiento.

Para comprender mejor cuándo usar RabbitMQ (o los corredores de mensajes tradicionales en general) o Kafka, eche un vistazo a la publicación de blog correspondiente de Pivotal como *punto de partida* .

Sin embargo, en general, trate de descartar *cualquier* motivo de rendimiento artificial al elegir su corredor. Hubo un momento en que los equipos y las comunidades en línea discutieron muchísimo sobre lo rápido que era RabbitMQ y lo lento que era ActiveMQ.

Ahora tiene los mismos argumentos sobre que RabbitMQ es lento con *solo* 20-30K / mensajes consistentes cada.single.segundo. Kafka es citado con 100K mensajes / segundo. Por un lado, este tipo de comparaciones convenientemente dejan de lado que usted está, de hecho, comparando manzanas y naranjas.

Pero aún más: ambos números de rendimiento pueden estar en el lado inferior o medio para Alibaba Group , pero su autor *nunca* ha visto proyectos de este tamaño (*millones* de mensajes por minuto) en el mundo real. Definitivamente existen, pero estos números no son motivo de preocupación para el otro 99% de los proyectos comerciales regulares de Java.

Por lo tanto, ignore la exageración y elija sabiamente.

¿Qué bibliotecas puedo usar para las pruebas de microservicio?

Dependiendo de su pila, puede terminar usando herramientas específicas de Spring (ecosistema de Spring), o algo así como Arquillian (ecosistema de JavaEE).

Querrá echar un vistazo a Docker y la muy buena biblioteca Testcontainers , que lo ayuda, por ejemplo, a configurar fácil y rápidamente una base de datos Oracle para sus pruebas de desarrollo o integración local.

Para burlarse de servidores HTTP completos, eche un vistazo a Wiremock . Para probar la mensajería asincrónica, intente incrustar (ActiveMQ) o acoplar (RabbitMQ) y luego escribir pruebas con el Awaitility DSL .

Aparte de eso, todos sus sospechosos habituales se aplican, como Junit , TestNG a AssertJ y Mockito .

Tenga en cuenta que esta no es una lista completa y, si le falta su herramienta favorita, publíquela en la sección de comentarios y la recogeré en la próxima revisión de esta guía.

¿Cómo habilito el registro para todos mis microservicios Java?

Iniciar sesión con microservicios es un tema interesante y bastante complejo. En lugar de tener un archivo de registro que puede menos o grep, ahora tiene archivos n-log, que le gustaría ver combinados.

Un excelente punto de partida para todo el ecosistema de registro es este artículo . Asegúrese de leerlo, especialmente la sección Registro centralizado en términos de microservicios.

En la práctica, encontrará varios enfoques:

- Un administrador de sistemas que escribe algunos scripts que recopilan y combinan archivos de registro de varios servidores en un solo archivo de registro y los coloca en servidores FTP para que los descargue.

- Ejecute cat / grep / unig / sort combos en sesiones paralelas SSH. Puede decirle a su gerente: eso es lo que Amazon AWS hace internamente .

- Use una herramienta como Graylog o ELK Stack (Elasticsearch, Logstash, Kibana)

¿Cómo se encuentran mis microservicios?

Hasta ahora, supusimos que todos nuestros microservicios se conocen entre sí, conocen su IPS correspondiente. Más de una configuración estática. Entonces, nuestro monolito bancario [ip = 192.168.200.1] sabe que tiene que hablar con el servidor de riesgos [ip = 192.168.200.2], que está codificado en un archivo de propiedades.

Sin embargo, puede optar por hacer las cosas mucho más dinámicas:

- Ya no podría implementar archivos application.properties con sus microservicios, en su lugar use un servidor de configuración en la nube desde donde todos los microservicios extraigan su configuración.

- Debido a que sus instancias de servicio pueden cambiar sus ubicaciones dinámicamente (piense en las instancias de Amazon EC2 que obtienen direcciones IP dinámicas y escala automática elástica de la nube), pronto podría estar buscando un registro de servicios, que sepa dónde viven sus servicios qué IP y puede enrutar en consecuencia.

- Y ahora, dado que todo es dinámico, tiene nuevos problemas, como la elección automática de líderes: ¿Quién es el *maestro* que trabaja en ciertas tareas para, por ejemplo, no procesarlas dos veces? ¿Quién reemplaza al líder cuando falla? ¿Con quien?

En términos generales, esto es lo que se llama *orquestación de microservicios* y otro gran tema en sí mismo.

Las bibliotecas como Eureka o Zookeeper intentan "resolver" estos problemas, como clientes o enrutadores que saben en qué servicios están

disponibles. Por otro lado, introducen una *gran cantidad de* complejidad adicional.

Simplemente pregúntele a cualquiera que haya ejecutado una configuración de ZooKeeper.

¿Cómo hacer la autorización y autenticación con microservicios Java?

Otro gran tema, vale su propio ensayo. Una vez más, las opciones van desde la autenticación básica HTTPS codificada con marcos de seguridad autocodificados, hasta ejecutar una configuración Oauth2 con su propio servidor de autorización .

¿Cómo me aseguro de que todos mis entornos se vean iguales?

Lo que es cierto para las implementaciones que no son de microservicio también es cierto para las implementaciones de microservicios. Intentará una combinación de Docker / Testcontainers, así como scripting / Ansible.

Intenta que sea sencillo.

No es una pregunta: Cuentos de sangría de Yaml

Haciendo un corte duro de preguntas específicas de la biblioteca, echemos un vistazo rápido a Yaml. Es el formato de archivo que se utiliza como formato de archivo de facto para "escribir la configuración como código". Desde herramientas más simples como Ansible hasta los poderosos Kubernetes.

Para experimentar el dolor de sangría de YAML, intente escribir un archivo Ansible simple y vea con qué frecuencia necesita volver a editar el archivo para que la sangría funcione correctamente, a pesar de varios niveles de soporte IDE. Y luego regrese para terminar esta guía.

```
Yaml:
  - is:
    - so
    - great
```

¿Qué pasa con las transacciones distribuidas? ¿Pruebas de rendimiento? ¿Otros temas?

Desafortunadamente, esos temas no llegaron a esta revisión de esta guía. Mantente sintonizado para más.

Desafíos conceptuales del microservicio

Además de los problemas específicos de microservicio de Java, también hay problemas que vienen con *cualquier* proyecto de microservicio. Estos son más desde una perspectiva organizacional, de equipo o de gestión.

Frontal / backend no coinciden

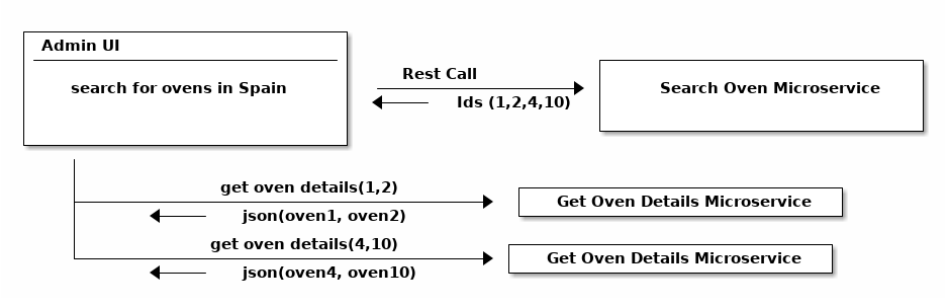
Algo que ocurre en muchos proyectos de microservicios es lo que yo llamaría la falta de coincidencia de microservicios de interfaz de usuario. Qué significa eso?

Que en los viejos monolitos, los desarrolladores frontend tenían una fuente específica para obtener datos. En proyectos de microservicio, los desarrolladores frontend de repente tienen *n-fuentes* para obtener datos.

Imagine que está creando un proyecto de microservicios Java-IoT. Digamos que está vigilando máquinas, como hornos industriales en toda Europa. Y estos hornos le envían actualizaciones periódicas de estado con sus temperaturas, etc.

Ahora, tarde o temprano, es posible que desee buscar hornos en una interfaz de usuario de administrador, tal vez con la ayuda de un microservicio de "horno de búsqueda". Dependiendo de cuán estrictos puedan interpretar sus colegas del backend las leyes de *microservicio* o *diseño de dominio* , podría ser que el microservicio de "horno de búsqueda" solo le devuelva ID de hornos, ningún otro dato, como su tipo, modelo o ubicación.

Para eso, los desarrolladores frontend podrían tener que hacer una o n llamadas adicionales (dependiendo de su implementación de paginación), para obtener un microservicio "obtener detalles del horno", con los identificadores que obtuvieron del primer microservicio.



Y si bien esto es solo un ejemplo simple (pero tomado de un proyecto de la vida real (!)), Demuestra el siguiente problema:

Los supermercados de la vida real obtuvieron una gran aceptación por una razón. Porque no tienes que ir a 10 lugares diferentes para comprar verduras, limonada, pizza congelada y papel higiénico. En cambio, vas a un lugar.

Es más simple y rápido. Es lo mismo para desarrolladores frontend y microservicios.

Expectativas de la gerencia

Este problema es un efecto secundario desafortunado de desarrolladores individuales, revistas de programación o empresas en la nube que impulsan microservicios:

La gerencia tiene la impresión de que ahora puede incorporar una cantidad infinita de desarrolladores al proyecto (general), ya que los desarrolladores ahora pueden trabajar de manera *completamente* independiente entre sí, cada uno en su propio microservicio. Con solo un *pequeño* trabajo de integración necesario, al final (es decir, poco antes de la puesta en marcha).

Veamos por qué esta mentalidad es un problema en los próximos párrafos.

Piezas más pequeñas no significan piezas mejores

Una cuestión bastante obvia es que *20 piezas más pequeñas* (como en los microservicios) en realidad no significan *20 piezas mejores* . Puramente desde una perspectiva de calidad técnica, podría significar que sus servicios individuales aún ejecutan 400 consultas de Hibernate para seleccionar un Usuario de una base de datos a través de capas y capas de código inservible.

Para volver a la cita de Simon Brown , si las personas no pueden construir monolitos correctamente, tendrán dificultades para construir microservicios adecuados.

Especialmente la resistencia y todo lo que sucede *después de* la puesta en marcha es una idea de último momento en muchos proyectos de microservicios, que da un poco de miedo ver los microservicios funcionando en vivo.

Sin embargo, esto tiene una razón simple: porque los desarrolladores de Java generalmente ~~no~~ están ~~interesados,~~ no están capacitados adecuadamente en resiliencia, redes y otros temas relacionados.

Las piezas más pequeñas conducen a piezas más técnicas.

Además, existe la desafortunada tendencia a que las historias de los usuarios se vuelvan cada vez más técnicas (y por lo tanto estúpidas), a medida que se vuelven más micro y abstraídas del usuario.

Imagine que se le pide a su equipo de microservicios que escriba un microservicio técnico de inicio de sesión contra una base de datos que es más o menos esto:

```
@Controller
class LoginController {

    // ...

    @PostMapping("/login")
    public boolean login(String username, String
password) {
        User user =
userDao.findByUserName(username);
        if (user == null) {
            // handle non existing user case
            return false;
        }
        if
(!user.getPassword().equals(hashed(password))) {
            // handle wrong password case
            return false;
        }
        // 'Yay, Logged in!';
        // set some cookies, do whatever you want
        return true;
    }
}
```

Ahora su equipo podría decidir (y tal vez incluso convencer a los empresarios): eso es demasiado simple y aburrido, en lugar de un servicio de inicio de sesión, escribamos un microservicio UserStateChanged realmente capaz, sin ningún requisito comercial real y tangible.

Y debido a que Java está actualmente fuera de moda, escribamos el microservicio UserStateChanged en Erlang. Y tratemos de usar árboles rojo-negros en alguna parte, porque Steve Yegge escribió que necesita conocerlos de adentro hacia afuera para solicitar Google.

Desde una perspectiva de integración, mantenimiento y proyecto general, esto es tan malo como escribir capas de código de espagueti dentro del mismo monolito.

¿Ejemplo fabricado y exagerado? Sí.

Desafortunadamente, tampoco es raro en la vida real.

Las piezas más pequeñas conducen a una comprensión más pequeña.

Luego está este tema de comprender el sistema completo, sus procesos y flujos de trabajo, si usted como desarrollador solo es responsable de trabajar en microservicios aislados [95: login-101: updateUserProfile].

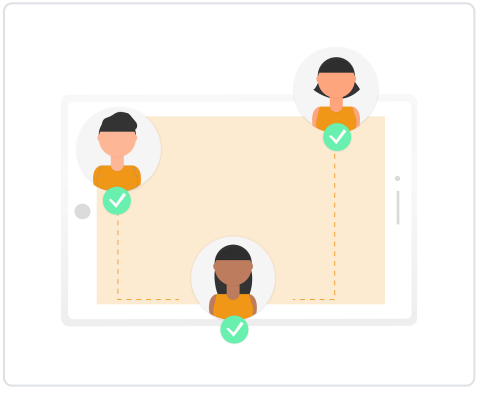
Se combina con el párrafo anterior, pero dependiendo de su organización, confianza y niveles de comunicación, esto puede llevar a muchos encogimientos de hombros y culpas, si una parte aleatoria de toda la cadena de microservicios se rompe, sin que nadie acepte por completo responsabilidad más.

No solo insinúa la mala fe, sino el problema de que en *realidad es realmente difícil* entender n-cantidad de piezas aisladas y su lugar en el panorama general.

Comunicación y mantenimiento

Lo que se combina con el último problema aquí: Comunicación y mantenimiento. Lo que obviamente depende en *gran medida* del tamaño de la empresa, con la regla general: cuanto más grande, más problemático.

- ¿Quién está trabajando en el microservicio número 47?
- ¿Acaban de implementar una nueva versión de microservicio incompatible? ¿Dónde se documentó esto?
- ¿Con quién debo hablar para solicitar una nueva función?
- ¿Quién va a mantener ese microservicio Erlang después de que Max dejó la empresa?
- ¡Todos nuestros equipos de microservicios trabajan no solo en diferentes lenguajes de programación, sino también en diferentes zonas horarias! ¿Cómo nos coordinamos correctamente?



El tema general aquí es que, de manera similar a las habilidades de DevOps, un enfoque completo de microservicios en una empresa más grande, tal vez incluso internacional, viene con una tonelada de desafíos de comunicación adicionales. Como empresa, debe estar preparado para eso.

Aleta

Después de leer este artículo, puede concluir que su autor recomienda estrictamente contra los microservicios. Esto no es del todo cierto: estoy tratando principalmente de resaltar puntos que se olvidan en el frenesí de los microservicios.

Los microservicios están en un péndulo.

Hacer microservicios Java completos es un extremo de un péndulo. El otro extremo sería algo así como cientos de buenos viejos módulos Maven en un monolito. Tendrás que encontrar el equilibrio correcto.

Especialmente en proyectos greenfield, no hay nada que le impida adoptar un enfoque más conservador y monolítico y construir menos módulos Maven mejor definidos en lugar de comenzar de inmediato con veinte microservicios listos para la nube.

Los microservicios generan una tonelada de complejidad adicional

Tenga en cuenta que, cuantos más microservicios tenga y menos talento DevOps realmente fuerte tenga (no, ejecutar algunos scripts de Ansible o implementar en Heroku no cuenta), más problemas tendrá más adelante en la producción.

Leer la sección Preguntas comunes sobre microservicios de Java de esta guía ya es agotador. Luego piense en *implementar* soluciones para todos estos desafíos de infraestructura. De repente te darás cuenta de que nada de esto tiene que ver con la programación empresarial (lo que te pagan), sino con una fijación de más tecnología en aún más tecnología.

Siva lo resumió perfectamente en su blog :

> No puedo explicar lo horrible que se siente cuando el equipo pasa el 70% del tiempo luchando con esta configuración de infraestructura moderna y el 30% del tiempo en la lógica comercial real.
>
> — Siva Prasad Reddy

¿Debería crear microservicios Java?

Para responder a esa pregunta, me gustaría terminar este artículo con un teaser de entrevista muy descarado, similar a Google. Si conoce la respuesta a esta pregunta por *experiencia* , aunque aparentemente no tiene nada que ver con microservicios, entonces podría estar listo para un enfoque de microservicios.

Guión

Imagine que tiene un monolito de Java que se ejecuta solo en la máquina dedicada Hetzner más pequeña . Lo mismo ocurre con su servidor de base de datos, también se ejecuta en una máquina Hetzner similar.

Y supongamos también que su monolito Java puede manejar flujos de trabajo como registros de usuarios y que no genera cientos de consultas de base de datos por flujo de trabajo, sino solo un puñado razonable (<10).

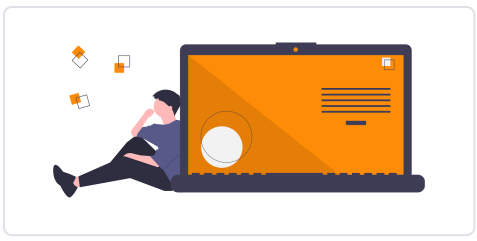
Pregunta

¿Cuántas conexiones de base de datos debe abrir su monolito Java (grupo de conexiones) a su servidor de base de datos?

¿Por qué? ¿Y a cuántos usuarios activos concurrentes cree que su monolito puede (aproximadamente) escalar?

Responder

Publique su respuesta a estas preguntas en la sección de comentarios. Estoy esperando todas las respuestas.



Ahora, toma tu propia decisión

Si todavía estás aquí conmigo: ¡Gracias por leer!

Compartir:

Hay más de donde vino eso

Te enviaré una actualización cuando publique nuevas guías. Absolutamente no spam, nunca. Darse de baja en cualquier momento.

e.g. martin@fowler.com ¡Quiero más!

Comentarios

Iniciar sesión

Add a comment

M ↓ COMENTA ANÓNIMAMENTE AGREGAR COMENTARIO

MARKDOWN

Votos más nuevos Más antiguos

? **Anónimo**
1 punto · hace 2 días

¿Cuántas conexiones de base de datos debe abrir su monolito Java (grupo de conexiones) a su servidor de base de datos? pool_size = (número de núcleos de la CPU * 2) + número de husos efectivo número de husos efectivo = 1 si SSD

METRO **Marco Behler**
0 puntos · Hace 1 dias

Tenemos un ganador. Eso fue demasiado rápido :)

? **Anónimo**
0 puntos · Hace 1 dias

Creo que también debería mencionar el tiempo de espera de http. La mayoría de los desarrolladores suelen olvidarse de establecer el tiempo de espera. El tiempo de espera predeterminado de la biblioteca predeterminada y del cliente http de Apache es infinito. https://docs.oracle.com/javase/7/docs/api/java/net/URLConnection.html)

METRO **Marco Behler**
0 puntos · Hace 1 dias

Muy buen punto! Lo agregará en una futura revisión.

? **Anónimo**
0 puntos · Hace 1 dias

> Muy interesante, gracias.

> ? **Anónimo**
> **0 puntos** · hace 2 días
>
> Excelente artículo !

> ? **Anónimo**
> **0 puntos** · Hace 1 dias
>
> Gracias bonito articulo

> ? **Anónimo**
> **0 puntos** · hace 2 días
>
> Great article. I wish technical leadership (architects and managers) at my client would have considered some of these points.
>
> I hope a future version of this article mentions the pain of dealing with legacy network infrastructure and cloud deployments. Proxy config, whitelisting IOS, firewall rules, etc.
>
> Another fun topic is deploying to multiple environments and the configuration changes needed for each (local, QA, integration, prod validation, prod, ...)
>
> Again, very well written article.

> ? **Anonymous**
> **0 points** · 1 days ago
>
> Hombre de mi compañía anterior, sería crucificado por tales declaraciones ;-) En cuanto a la pregunta al final: depende. Depende de cuántos registros tenga por hora, digamos. ¿Cuántos usuarios activos simultáneamente puede manejar? Bueno, el registro es una acción única, por lo que el número de usuarios activos simultáneamente no importa a este respecto. Pero para dar un número, le da 100 000 usuarios activos activos :-)