JRebel

WHITE PAPER

Developer's Guide to Microservices Performance

Introduction

More Java developers are working within microservices-based applications than ever before. And, with developers taking a greater responsibility for application performance, developing performant microservices has never been more important.

In this white paper, we look at some of the unique ways that microservices can cause unexpected performance issues — with a focus on common inter-service performance issues and patterns that can help increase resilience while decreasing the chance of cascading and catastrophic failures.

Contents

Finding Microservices Performance Issues

At a symptomatic level, there can be clear warning signs of performance issues within your application: slow services, failing services, and, if the application isn't engineered resiliently, application failure.

Ideally, the developer is looking at application (and individual service) performance throughout the development pipeline with a variety of tools suited to the stage.

APM SOLUTIONS AND SERVICE MESHES

For applications already in production, APM solutions like Dynatrace or AppDynamics can help developers to assess availability and performance of their applications and services. These tools focus on helping companies identify issues that have appeared in their application and mitigate the risk presented by these performance issues. APM tools typically provide an automated process called rollback which will revert your application to the last working version of your application if a performance issue presents itself.

Meanwhile, service mesh solutions like Istio and Linkerd can help to streamline inter-service communication and provide insights into service health, latencies, and request volume. Those insights can also help with data-driven feature rollout via canary or blue/green deployments.

Application performance monitoring and service mesh solutions provide big benefits during production due to their ability to identify issues that are affecting the application in production. They can also provide quick band-aid solutions that help to mitigate those issues. But developers still need tools that can give insight into service performance during development in order to truly address those problems.

ANALYZING CODE AND DATA STORE QUERIES

Another critical part of developing performant microservices-based applications is in early stage analysis and optimization.

Developers should regularly look at the performance of individual services and the combined application during development (including data store queries and third-party services). By doing so, the developer gets better insight into how their code is interacting with other services and can better contribute to the application at large.

This is particularly true as more development teams are adopting the DevOps methodology. Previously, developers never cared about performance as it was "someone else's job". With DevOps methodology, engineers are increasingly responsible for the way in which their application is delivered and how well it performs.

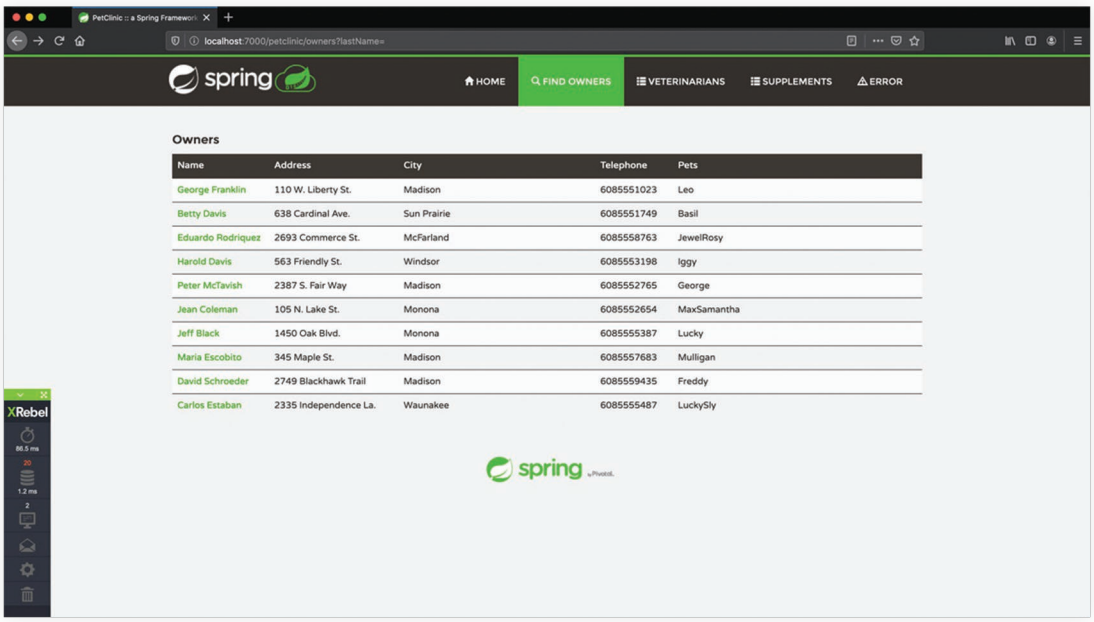
PROFILING TOOLS FOR JAVA MICROSERVICES

Profiling tools like JProfiler, VisualVM, YourKit, or Stackify Prefix can give greater visibility into your Java microservices application. These tools are typically plugged into the test environment to provide the engineers with a tool to address performance issues like memory leaks and threading issues.

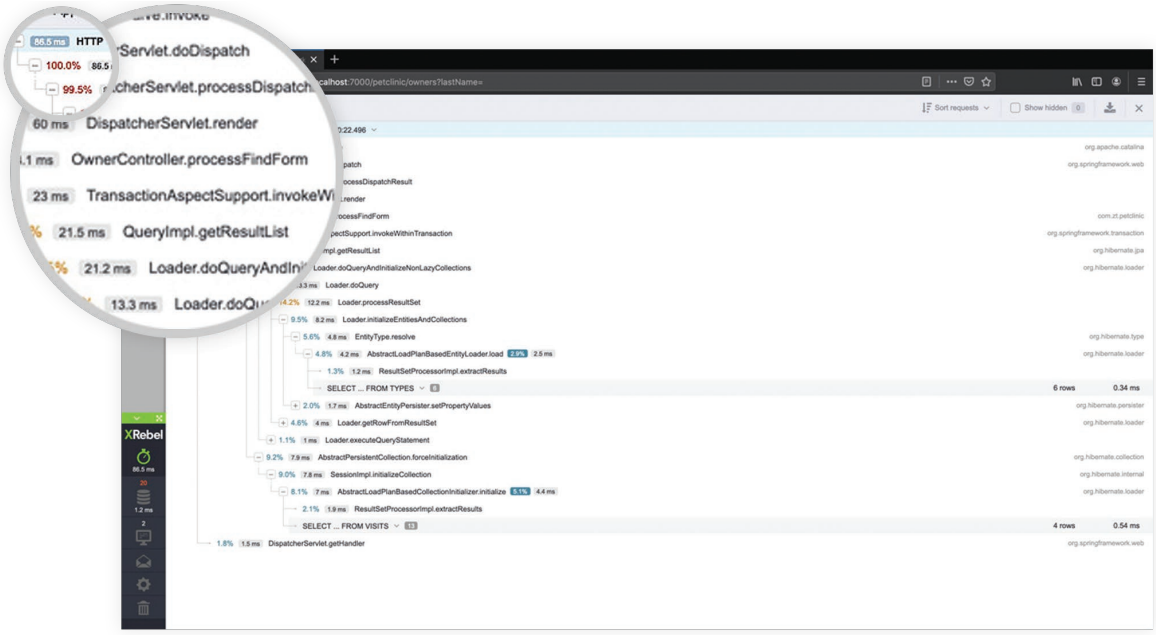
Identifying Microservices Performance Issues With XRebel

If you're debugging Java microservices applications during development, XRebel can help to easily find performance issues. Whether that's spotting a slow service, finding inefficient queries to a data store, or tracing parallel CompletableFuture requests, XRebel can be an invaluable tool for Java developers.

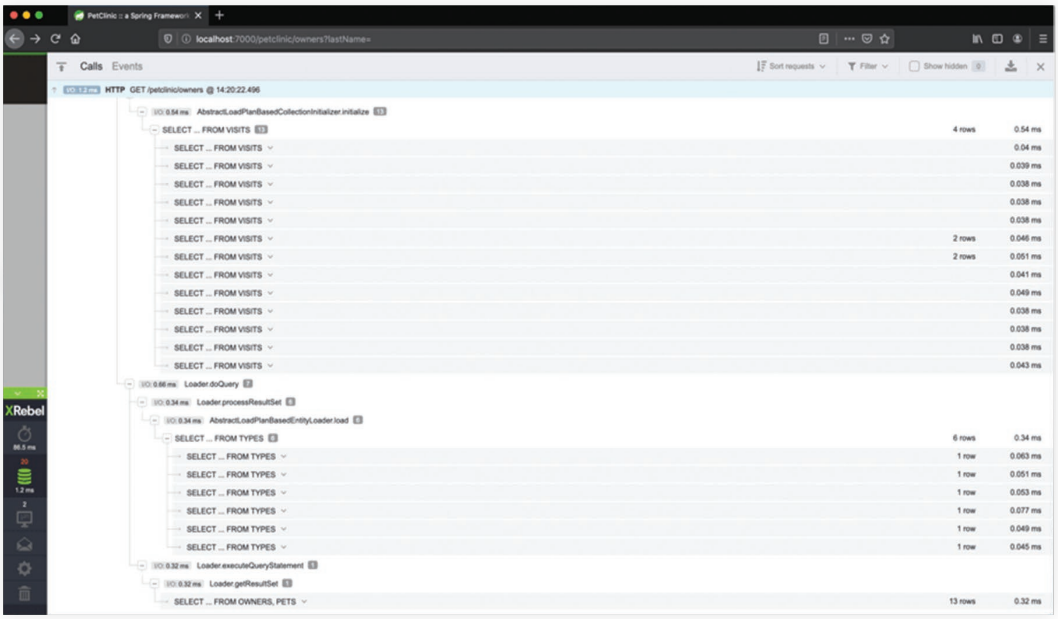
In the example below, we see XRebel helping to identify a performance issue tied to a series of database requests.



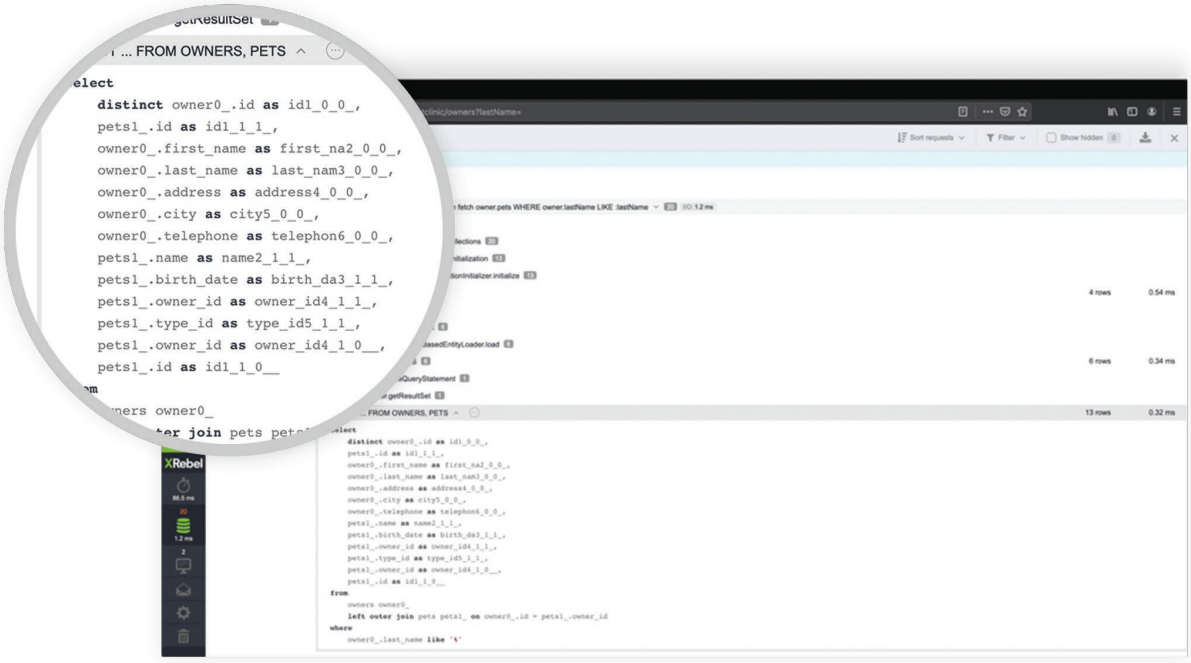
Looking closer, we understand that we are spending about the same amount of time initializing the Pets as we are initializing the Visits within the OwnerController.processFindForm.



Although this request only takes 86.5ms, we can see that we are taking twice as long processing both sets of query tables. Next, we'll proceed to the I/O view where we can further investigate the queries and determine if there is an issue.



Looking at the queries we can see that we are calling the find all owners, and that we are making a single query that is returning 13 rows of data.



Looking the at the visits table, we see that we are calling all visits that a pet has. In order to fetch the visits, a separate query is being issued for each pet. If, for example, an owner has 3 separate pets, then 3 additional queries will be executed to fetch the visits information from the pets. This is known as an N + 1 problem — where a single query is designed to fetch all N pets with an additional N queries to fetch all visits of those pets.

Fixing Microservices Performance Issues

Not all microservices performance issues are created equally. Some, like the N+1 Problem, can be as simple as changing a fetch type. Unfortunately, not all are so easy.

Picking the wrong data store for a service, for example, can mean additional hardware cost, higher risk of timeouts and unavailability, and a bad end-user experience.

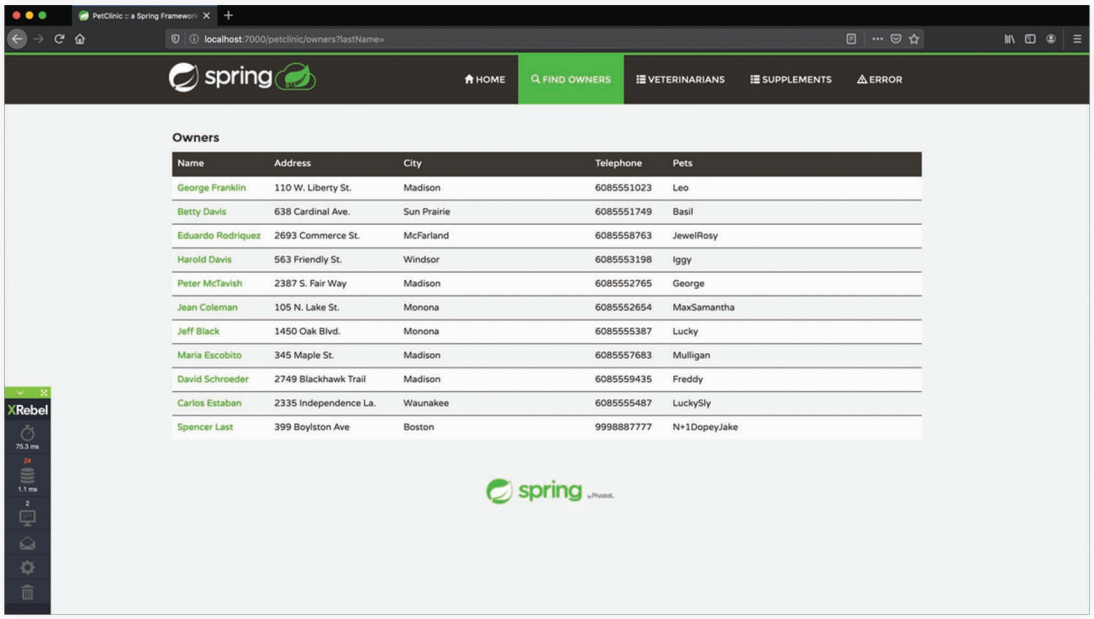
Even fixes, as we detail in our section on antipatterns, can create unintended performance consequences for your application.

SOLVING N+1 PROBLEMS

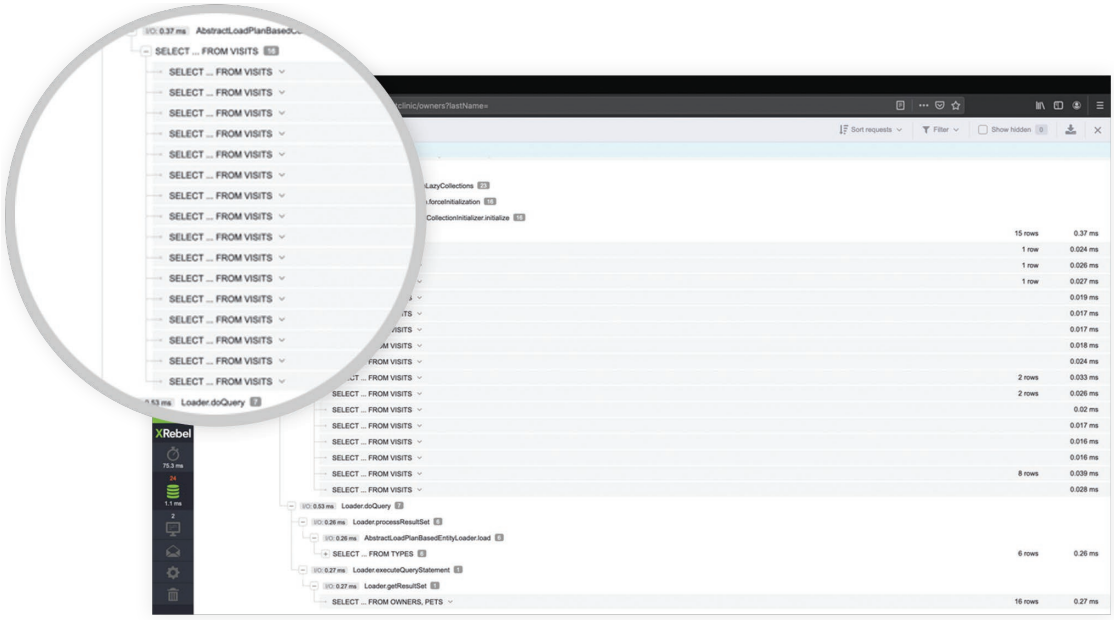
Object oriented languages like Java often need to work with relational databases. That either means a developer or database administrator needs to write (optimized) SQL requests, or they need to use an intermediary layer, like an ORM framework, that generates compatible requests for that database. While functionally great, ORM frameworks have a reputation for creating unoptimized queries — including N+1 queries.

What is an N+1 Problem?

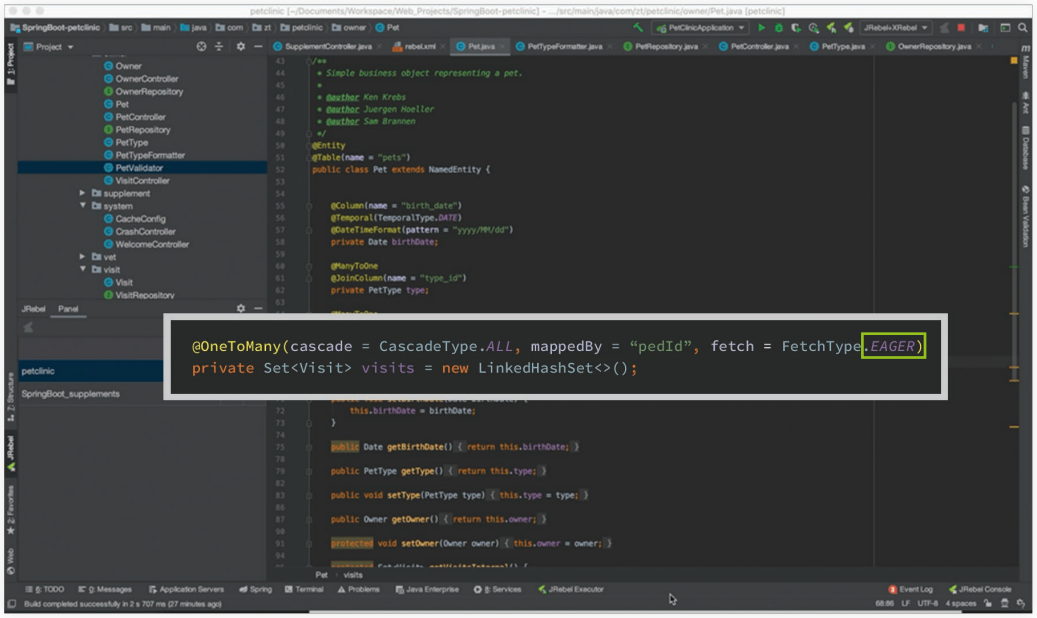
An N+1 Problem, also known as an N+1 Select Problem or N+1 Query, happens when a service requests a number of rows (N) of data from a database, then individually requests dependent data for each of those N items.



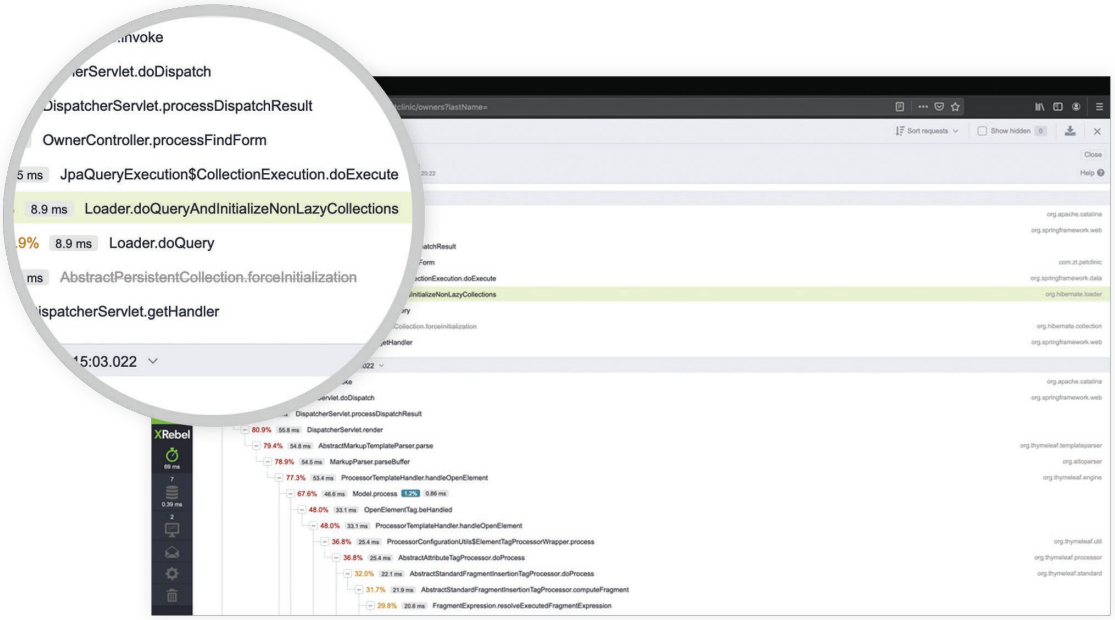
Let's return to our earlier N+1 problem. We've added a new owner, Spencer Last, and with that new owner added three new pets N+1, Dopey and Jake. From the I/O we will see that we have increase the query count to 16.



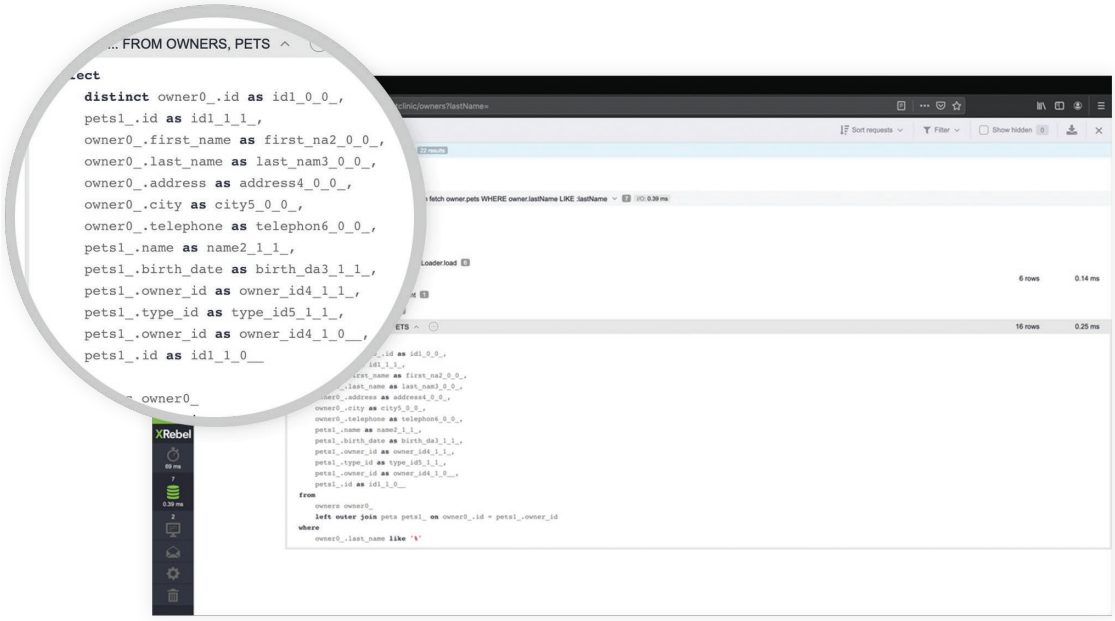
To remedy this we will be changing our fetching strategy in the Pet Class from Eager to Lazy.



After making the change we will return to the application refresh the page and look at our results.



When comparing the previous request with the updated code we can immediately see that we have reduced to the time spent in the Loader.doQueryAndInitializeNonLazyCollections method trace by over 2x.



We can also see in the XRebel Comparison view that we have removed branch from the request call tree AbstractPersistentCollection.forceInitialization. Next, if we proceed to the I/O view we can see that we have significantly reduced the number of queries to 7!

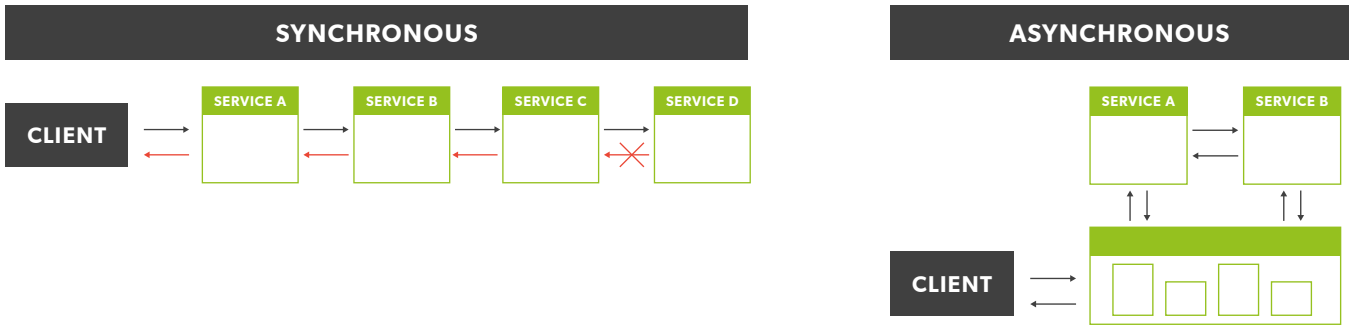
Now we are only calling the Owners and Pets in one query and returning the Pet types in another.

Using Asynchronous Requests

Determining when to use synchronous vs asynchronous calls has a large impact on application performance. And, depending on the circumstance, calling a service synchronously can cause significant performance bottlenecks for other services and for the combined application.

By using asynchronous requests, a service can make a request to another service and return immediately while that request is fulfilled. That allows for more concurrent work within individual services, and more efficient requests for the combined application.

Keep in mind, developers still need to make sure that the receiving service can fulfill those asynchronous requests within an acceptable timeframe, and scale to accommodate request load.



ASYNCHRONOUS MESSAGING TECHNOLOGIES

Some of the most popular open source asynchronous messaging systems used in microservices architectures:

Apache Kafka is an open source stream processing software platform. It allows developers to publish, process, and store streams of data in distributed and replicable clusters.

RabbitMQ is a high scale, high availability open source message broker used for message queuing, routing and more.

ActiveMQ is the most popular, multi-platform Java-based messaging server. It's used for load balancing, availability fail safes, and more.

Mind Your Antipatterns

Sometimes trying to solve a problem can create a bigger problem. For example, adding timeout and retry functionality to a service sounds like a good idea, but if another service it calls is chronically slow and always triggers the timeout, the retry will put additional stress on an already overloaded service, causing a bigger latency issue than the original fix tried to resolve.

Before implementing resiliency techniques in one service, carefully consider how it will impact other services and the application as a whole. Service meshes like Istio can make overall resiliency easier by enforcing consistency and avoiding one-off implementations.

AN ASYNCHRONOUS ANTIPATTERN

As we discussed in the last section, asynchronous calls can help to avoid a single slow response slowing down the entire response chain. But developers also need to be careful to avoid antipatterns with these asynchronous calls.

For example, a developer puts a message queue between two services to handle short term call bursts. This helps the service to handle more calls without getting overloaded, but it doesn't fix the underlying issue — the service is still slow.

In the end, the message queue quickly maxes out, calls start to fail, and the dependent services are more difficult to restart.

To make a bad situation worse, making upgrades to the receiving service is now more difficult because messages in an older format may need to be processed alongside a newer format.

Throttling Overactive Services

Is one of your microservices receiving too many requests to handle? Throttling requests or using fixed connection limits on a service by service basis can help your receiving services keep up. Throttling also helps with fairness by preventing a few hyperactive services from starving others.

While throttling does ensure availability of the service for your application, it will make it work slower. But it's a better alternative than having the application fail altogether.

TECHNOLOGIES FOR THROTTLING, LOAD BALANCING AND SCALING

Developers don't need to reinvent the wheel with every microservice or microservices application. Using a service mesh like Istio or Linkerd can help developers to create better performing microservices — without the overhead of in-house solutions for throttling, load balancing, and scaling. At a logistical level, they can help add network configuration, security, traffic management, and telemetry to your application.

At the application level, these services can help to apply resilience patterns like load balancing, retries, failover, and circuit breaker.

For deployment, these services can help support canary and blue/green releases for better overall application quality.

Istio provides a dedicated layer that facilitates communications between microservices, including security, observability, and traffic management.

Linkerd is a service mesh that provides runtime debugging, observability, security, and traffic management via proxies attached to individual services.

Managing Third-Party Requests

Even if your microservices are running efficiently with one another, sometimes the limitations of a third-party service or API can cause significant issues for an application.

Using text detection in images? Your requests to the Google API will play a role in your application performance. Authenticating your users with Facebook? If they're having a slow response, now you are too. Using Amazon Polly for voice recognition? You get the picture.

With the increasing presence of third-party services and APIs within applications, it's important that developers take proper action to ensure these services and APIs don't lead to application failure.

KNOW THE LIMITATIONS

It's important for developers to understand the limitations of a third-party service before relying on them at scale. Can they keep up with your expected demand while maintaining the performance you require? Is their stated SLA compatible with yours? For example, if you promise 99.99% uptime, but one of your service providers only guarantees 99.9%, your customers will eventually be disappointed and blame you.

ENSURING RESILIENCY

Developers also need to be proactive. Applications must be resilient to slow third-party requests by utilizing best practices like caching, pre-fetching, or using resiliency patterns like the circuit breaker to prevent services from causing cascading failures.

Avoiding Application Ceiling

Even properly configured and optimized services can have performance ceilings.

If you've already determined that all your requests are necessary and optimized, and you're still overloading your service, consider load balancing across additional containers to improve scalability.

You might even consider autoscaling to dynamically adjust to incoming request load by adding and removing containers as necessary. If you go this route, be sure to implement a maximum container count and have a plan for defending against DDoS (Distributed Denial of Service) attacks, especially if your application is deployed in a public cloud.

That said, even the best-planned and coded applications and services have hardware ceilings. Applications and services that need to maintain large databases with true ACID transactions, for example, will always need to have enough processing power to fulfill requests to those databases — even if that load is balanced across multiple servers.

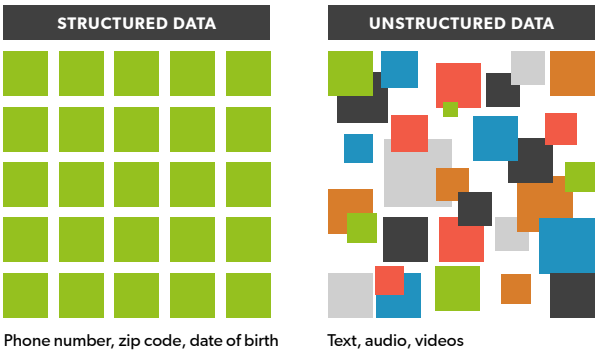
Consider clustering technology and potentially moving some of your services to NoSQL solutions that can offer scale higher than an RDBMS. However, be prepared to deal with eventual consistency and compensating operations if you need ACID-like transactions across services.

Choosing the Right Data Store

Microservices give the flexibility to use multiple data stores within a single application. But picking the wrong kind of storage can cause significant performance issues.

Imagine the hardware cost for a streaming video service if they used a RDBMS to dynamically update content recommendations for 169 million users based on their individual viewing habits!

It's important for developers to choose data stores for microservices at a service by service level and to make sure that the selected data store is the best tool for each particular job.



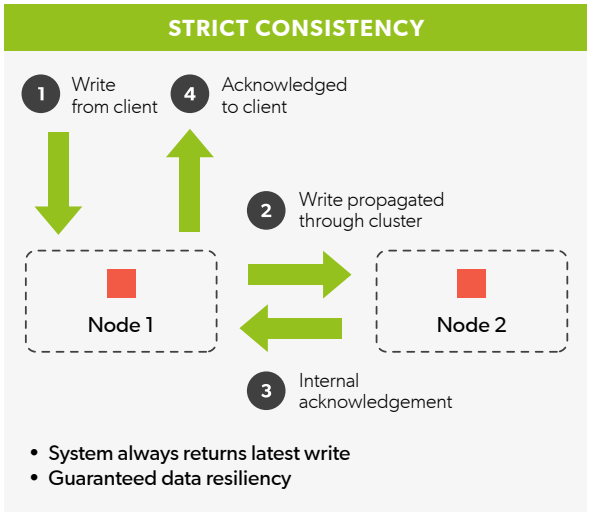
Structured vs Unstructured Data

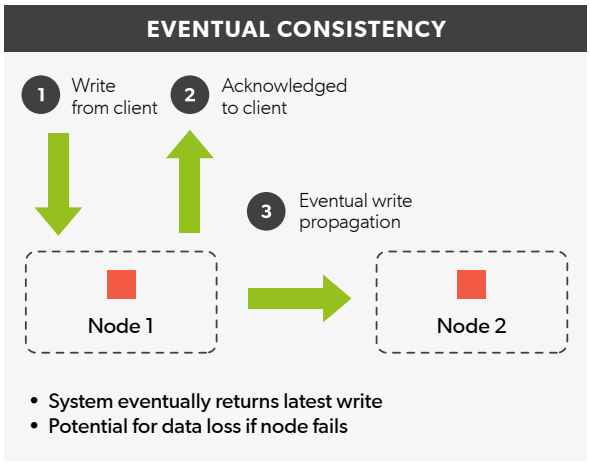
DATA STORES FOR CHANGING, UNSTRUCTURED DATA

Using a RDBMS for a service that processes a large amount of changing, unstructured data will mean working against the primary benefit of the relational database architecture – consistency. Keeping data consistent across these rapidly changing databases would be unnecessary and expensive.

In these cases, it's better to use a scalable and schemaless NoSQL data store like MongoDB or Cassandra.

DATA STORES FOR CONSISTENT DATA





Strict vs. Eventual Consistency

Traditional relational databases choose to be consistent even if it means becoming unavailable during a hardware or software failure. If your use case allows for occasional downtime and ACID transactions are paramount, or eventual consistency causes more problems than it's worth, consider a tried and true RDBMS. These workhorse databases don't grab headlines like they used to, but they're still as valuable as ever when the use case fits.

Caching Database Calls

When a service requests a field of data across multiple databases, each of those databases has the capacity to hold up that request.

If that information is frequently accessed by hat service, consider caching that information in an easily accessible place that doesn't rely on multiple databases.

Making sure your request is targeted at a single, cached database, with rules that add request destinations upon a set request time limit can help make sure your database calls never timeout, but also don't cause excessive calls.Memcached is used in high performance and distributed systems to store arbitrary data in-memory – allowing for better utilization of available memory. Memcached is used primarily for key-value memory structures.

Redis, like Memcached, is used for high performance in-memory data storage but is more functionally robust. It supports Hash, List, String, Set, and Sorted Set data types, and can also swap cached memory to disk if not used frequently enough to warrant storage in-memory.

Some database systems offer native in-memory caching. Cassandra, for example, can be configured to store data in-memory for Key and Row data types while compacting SSTables by default.

CACHING AND PROJECTIONS

In Event-Driven Architectures (EDA), the system of record may be an ordered collection of events that already happened (e.g., "customer created", "order shipped", "added $10 to account ABC").

To determine someone's current account balance, you might have to look through millions of records to sum all the deposits and withdrawals for a particular account, which is obviously too slow for a waiting user.

In this case, you might create a "Projection" from the main event stream that contains only account balance-related transactions and store it in another data store more suited for quick lookups. At that point, you could specifically cache account balances in an in-memory data store like memcached or Redis for even faster queries if that query becomes your primary bottleneck to performance.

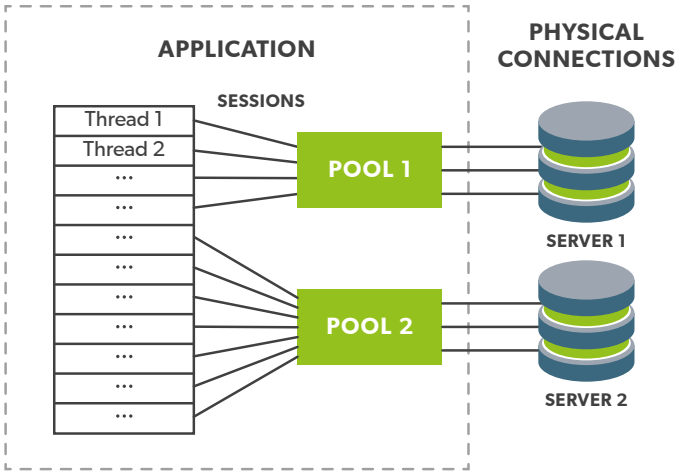
POPULAR CACHING TECHNOLOGIES

Caching can be complicated, but open-source technologies like Memcached and Redis can make caching easier to integrate.

Memcached is used in high performance and distributed systems to store arbitrary data in-memory – allowing for better utilization of available memory. Memcached is used primarily for key-value memory structures.

Redis, like Memcached, is used for high performance in-memory data storage but is more functionally robust. It supports Hash, List, String, Set, and Sorted Set data types, and can also swap cached memory to disk if not used frequently enough to warrant storage

Some database systems offer native in-memory caching. Cassandra, for example, can be configured to store data in-memory for Key and Row data types while compacting SSTables by default.



Database Connection Pool Example

Database Connection Pools

One of the most effective ways to reduce overhead in microservices that access and alter databases (aside from caching) is to pool their connections to that database.

A typical service will establish a connection to a data store, issue some queries, and close the connection very quickly. Connecting and disconnecting adds quite a bit of overhead to a short-lived connection, thus limiting the amount of work that can be done. A connection pool typically establishes a fixed set of connections to a data store when it starts up and lets the calling service re-use an existing connection from the pool instead of opening and closing them with every request. The result is a much more efficient service.

CONNECTION POOLING FRAMEWORKS

While in the past, some hardy developers may have needed to create their own connection pooling services, there are now frameworks that make connection pooling in Java easier to implement.

Popular connection pooling frameworks for Java include:

- Apache Commons DBCP – **Implements database connection pooling for JDBC.**
- HikariCP – **High-performance JDBC connection pooling.**
- C3PO – **Library that augments JDBC drivers for the enterprise.**

When Microservices Fail

Despite all the planning and painstaking development involved in creating a microservices application, microservices can and will fail. With enough services and load on the system, the application will always be in a state of partial failure and recovery due to hardware issues, networking glitches, virtual machine crashes, bursty traffic causing timeouts, and the like. It's no longer a matter of "if" or "when" something will go wrong — that's a given — but rather how to architect for automatic self-healing.

It's up to development teams to create applications that can handle failure gracefully. What that means depends on the application. For some, that may mean maintaining full functionality and adequate performance during service failures. For others, it may mean just preventing one failure from causing cascading failure of the application.

How developers and application architects prevent those failures varies from project to project. But the core concept of application resiliency, and the resiliency patterns and techniques used to preven these failures, should be a major topic of discussion for every development team.

What Are Resiliency Patterns?

Resiliency patterns are a type of service architecture that help to prevent cascading failures and to preserve functionality in the event of service failure. Common resiliency patterns used in application development include the bulkhead pattern and circuit breaker pattern.

In this section of the white paper, we'll look common failure points in microservices applications, how to predict where your application may fail, and look at the resiliency patterns you can use to help prevent cascading and otherwise catastrophic failures for your application.

PREDICTING RESILIENCE ISSUES

Many of the performance issues and fixes we've looked at so far in this white paper can be looked at as band-aids for performance problems. Just because you have optimized requests from one service to another service doesn't mean you've addressed the underlying architectural issue within the application.

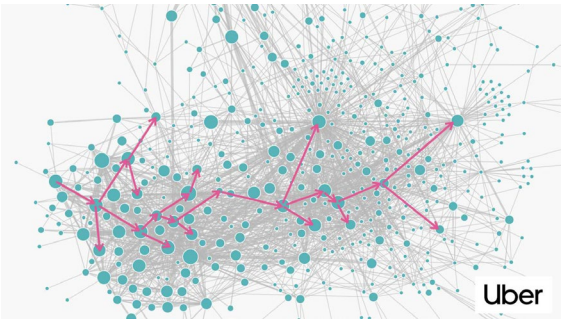
As Mark Richards points out in his book, Finding Structural Decay in Architectures, prevalent enough issues — like unintended static coupling between services — can be indicative of a larger architectural mismatch.

Because many, if not most, microservices applications are transitioned from existing monolithic applications, there are bound to be a few cases where making that transition was the wrong decision.

Identifying Failure Points

If you have been troubleshooting your application and individual service performance, you have already likely identified a few services that either receive or send a lot of requests. Optimizing those requests is important and can help to prolong availability. But, given a high enough load, the services sending or receiving those requests are likely failure points for your application.

> "100% is the wrong reliability target for basically everything (pacemakers and anti-lock brakes being notable exceptions)."
>
> - Betsy Beyer, Site Reliability Engineering: How Google Runs Production Systems



Source: https://www.infoq.com/presentations/uber-microservices-distributed-tracing/

For enterprise microservices applications like Uber, where engineers are using thousands upon thousands of microservices, tracing requests across these services can be hopelessly complex – with traces that have hundreds of thousands of spans.

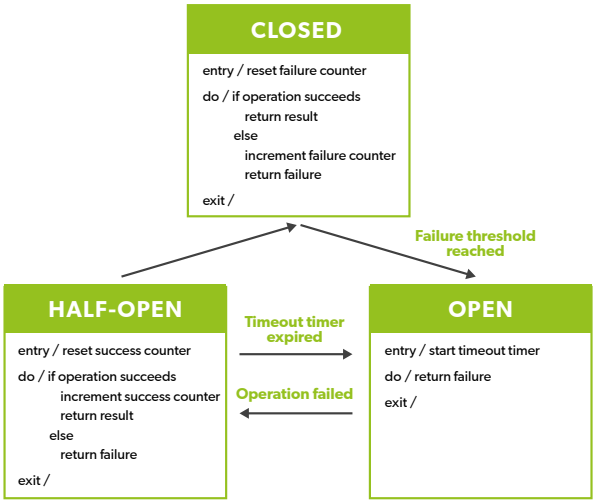
Using visualization tools to make sense of those complex traces and how they progress through the microservices that comprise the application helps engineers to identify and bolster application failure points proactively instead of reactively.

USING RESILIENCE PATTERNS

In the next section, we'll cover three resilience patterns that can help developers to engineer failure-resistant microservices applications.

Circuit Breaker Pattern

Applications often rely on remote resources, like third-party services, as a key component of their program. But what happens when one of those remote resources times out upon request? Does your microservice continue calling that resource in an endless loop until it fulfills that request? What happens when multiple services are requesting that same remote resource?



Circuit breaker pattern example

USING THE CIRCUIT BREAKER PATTERN

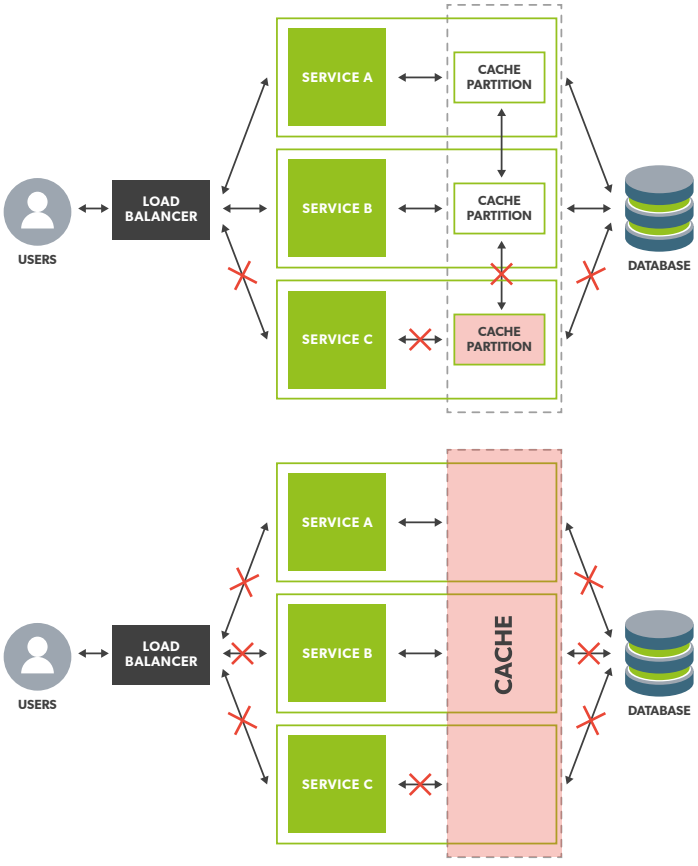
The circuit breaker pattern, just like the electrical engineering concept, would prevent subsequent requests from occurring – preventing the service from

getting overloaded. Additionally, by monitoring how many requests to that service have failed, a circuit breaker pattern can prevent additional requests from coming into the service for an allotted time, or untiL the amount of failed requests by time have reaches a certain threshold.

When recovering from failure, the circuit can return incrementally to full functionality to prevent overloading the service with a large amount of requests.

WHEN TO USE THE CIRCUIT BREAKER PATTERN

The circuit breaker pattern can add significant overhead to your application, so it's best to use sparingly, and only in cases where you're accessing a remote service or shared resource prone to failure. If you user a service mesh like Istio, it's easy to experiment with various resiliency patterns, including Circuit Breaker, to see which techniques work best for each service.



Bulkhead pattern applied via cache partition

Bulkhead Pattern

When architecting microservice-based applications, it's easy to overload specific services within the application. As we've outlined throughout this white paper, overloading these services can lead to slow applications at best, and catastrophic failures at worst.What is the Bulkhead Pattern?

The bulkhead pattern is an application resiliency pattern commonly employed in microservices applications. It's used to isolate services and consumers via partitions in order to prevent cascading failures, give sliding functionality when services fail vs total failure, and to prioritize access for more important consumers and services.

In the example below, we see the bulkhead pattern applied to a database cache. In the non-partitioned database cache, the failure of the cache leads to the services interacting directly with the database, leading to application failure. But in partitioned example, we can see that the partition for service c has failed independently, resulting in the failure of service c without immediately causing failure in the other two services.

If those cache partitions were configured as a cluster sharing the same data in each instance, then the service could be configured to access one of the remaining partitions upon failure.

TIPS FOR USING BULKHEAD PATTERN

While useful for preventing catastrophic failures, the bulkhead pattern does add complexity to your application. For teams working on services independently, understanding where each service fits within the bulkhead pattern and how their service is partitioned relative to other services is crucial to a performant application.

Additionally, keep in mind that this pattern — while great for resiliency — can add another performance hurdle for your application.

Stateless Services

What happens if a service being called upon fails in your microservices application? If there isn't an alternative database or service that can fulfill that request, additional services can fail, leading to a cascading failure of the entire application.

But what if you could have a copy of that service ready to go if the primary fails? Or another one that could be spun up on demand instantly if that second service fails?

USING STATELESS SERVICES FOR RESILIENCY

Stateless services can achieve that function Because they depend on inputs, and don't actually hold data, any copy of that service can serve just as well as the original.

In addition, these services can be instantiated dynamically as the need arises instead of existing permanently within the application – meaning less resource usage for the service and application.

As an alternative, you can imagine a service that needs to serve many requests for non-persistent data, but the demand is lumpy, coming in bursts. But when those requests are issued, using asynchronous requests or request buffers can't fulfill the requests fast enough for a good end-user experience.

An engineer could allocate a large chunk of resources to that service, add additional hardware to accommodate in case of high loads, or they could employ a stateless, scalable service that can spin up new services to fulfill requests during heavy load, using minimal extra resources and only when scaled.

WHEN TO USE STATELESS VS STATEFUL SERVICES

Stateless services are ideal for high availability, scalability, and performance, but obviously can't be used for everything. In fact, they're only possible if the data they act upon is provided by the caller. For example, converting a video from one format to another, text-to-speech generation, or image recognition.If you do need to store state, as the majority of services do, carefully consider where to put it. If you store it in the local container, access will be fast, but what happens if the container crashes and the data is corrupted? If you store in a NAS or SAN, will there be consistency issues if multiple services read and write to it concurrently? If you use a cloud storage service, how will that affect latency? As with most questions, the answers are specific to each service — use the best tool for the job as opposed to forcing every problem to look like a nail if you own a really good hammer.

Closing Thoughts

In this white paper, we've talked about just some of the ways that developers can improve performance in microservices.

But it's important to note that while many of these ideas can improve application performance, giving your developers greater visibility into the application as individual services are developed, and continually communicating about changes happening within the application, can also help teams to create more performant applications.

Lastly, all the troubleshooting in the world won't fix an architectural mismatch. Microservices bring many benefits, but they're not a one-size-fits-all solution.

If you have any questions about the material covered within this white paper, be sure to reach out to us on Twitter, LinkedIn, or via the JRebel contact us page.

Thanks for reading,

The JRebel/XRebel Team

Find Performance Issues With XRebel, Fix Them Faster With JRebel

Want to see how XRebel and JRebel combine for lightning-fast performance improvements during Java application development? Request a trial and see the difference for your team.

REQUEST TRIAL

https://www.jrebel.com/products/free-trial

About Perforce

Perforce powers innovation at unrivaled scale. With a portfolio of scalable DevOps solutions, we help modern enterprises overcome complex product development challenges by improving productivity, visibility, and security throughout the product lifecycle. Our portfolio includes solutions for Agile planning & ALM, API management, automated mobile & web testing, embeddable analytics, open source support, repository management, static & dynamic code analysis, version control, and more. With over 15,000 customers, Perforce is trusted by the world's leading brands to drive their business critical technology development. For more information, visit www.perforce.com.