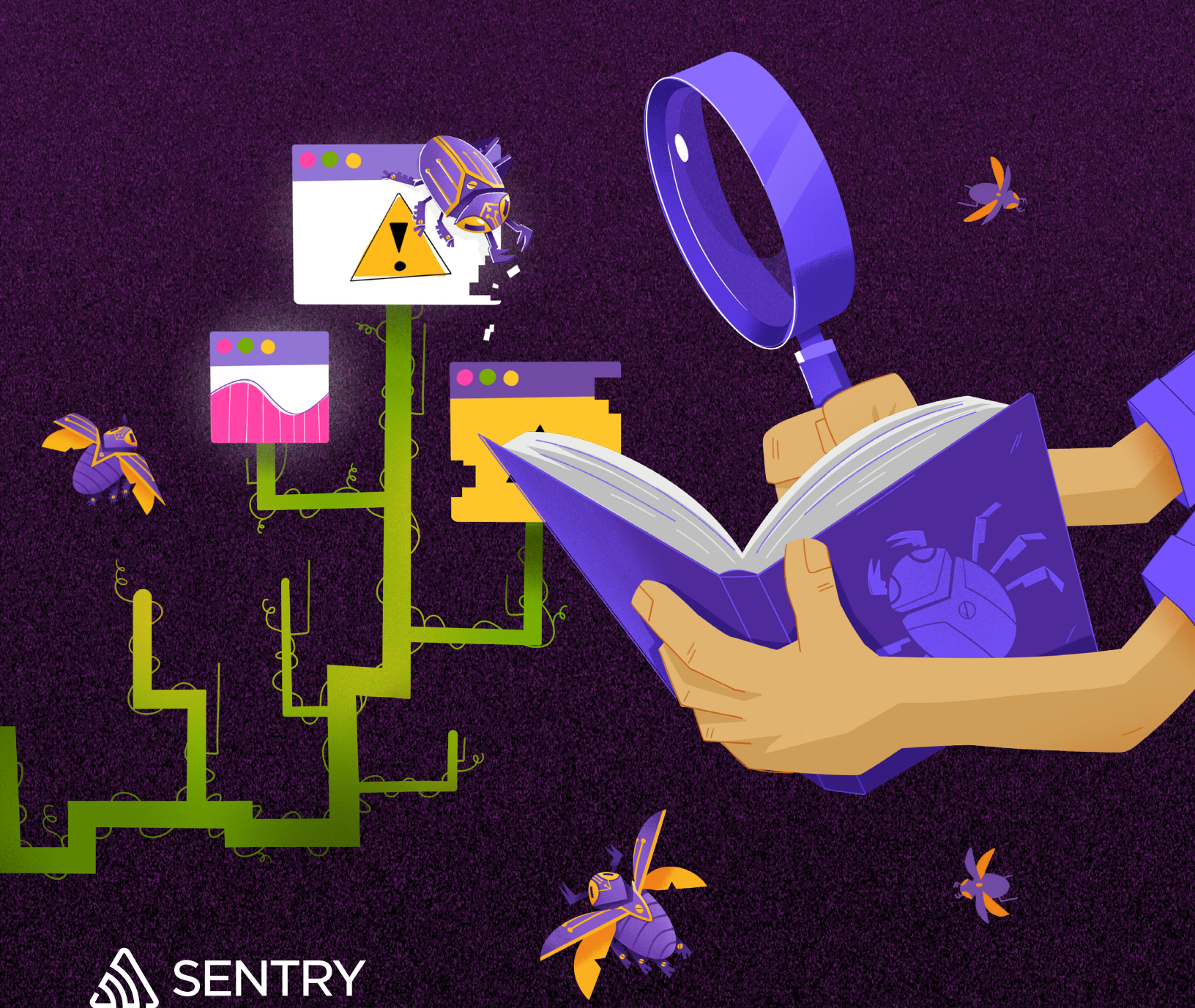
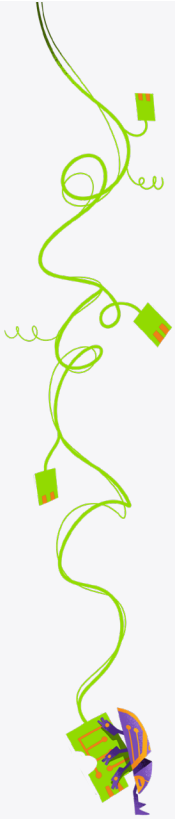


SENTRY

Debugging Microservices and Distributed Systems

From payment processing to notifications

Introduction



This resource explores the complexities of debugging modern web applications, particularly within microservices architectures. It highlights the shift from monolithic to microservices architectures, driven by companies like Netflix and Amazon, to improve scalability, flexibility, and resilience. This resource discusses:

- The benefits of microservices
- The difficulty in debugging microsystems architectures
- The importance of focusing on “debuggability” tools to manage and maintain microservices effectively
- Practical advice on setting up a microservices architecture for success

Gone are the days when you had just a couple of files open in a simple editor and could debug with `print(“Here”)`. Nowadays, even if you’re primarily a frontend developer, you really have to be able to debug errors and performance issues over the entire stack because front end issues often originate in the backend or other services. With a push towards server-side rendering (SSR) and the complex nature of the backend microservices architecture, we’re able to do so much, so quickly, but at the cost of making debugging significantly more challenging.

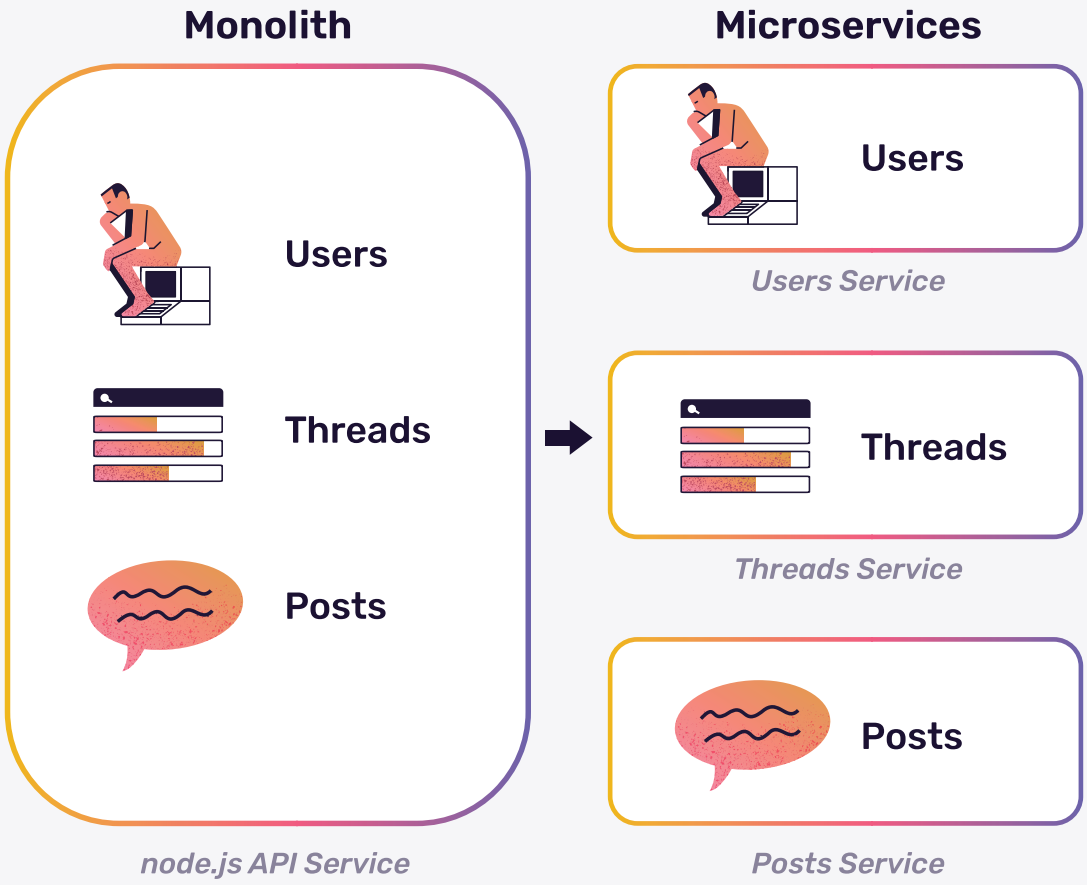
Before you continue building with the best and newest app by calling every single microservice ever created and putting AI everywhere so you can say you’re an “AI-first” company, consider the debuggability of your architecture and the tools you use.

Sarah Guthals, PhD
Director of Developer Relations, Sentry

What are microservices?

Maybe you've started reading this and thought, "Oh, this isn't for me, I don't even use microservices". Boy, would you be wrong (probably).

In the early 2010s, companies like Netflix and Amazon moved from a monolithic architecture (all the code for all the projects and APIs and services in one place) towards a microservices architecture to improve scalability, flexibility, and resilience for the on-demand content being streamed worldwide. But it really wasn't until the mid-2010s when Fowler and Lewis wrote "Microservices: a definition of this new architectural term" that we saw the emergence of tooling and frameworks to take advantage and fortify this new shift to a microservices architecture to really become mainstream.



Now, in the mid-2020s, you probably use microservices for:

- **User Authentication:** With services like OAuth and OpenID, the safety and privacy of your users and their data is fairly simple.
- **Payment Processing and Order Management:** Stripe, PayPal, Square, and Shopify lead the way in making it easier than ever to spin up an e-commerce site your customers can trust.
- **Notification Services:** Sending emails, SMS, push notifications, and in-app messages to users based on triggers or schedules can be quickly done with services such as Twilio, Firebase Cloud Messaging, and SendGrid.
- **Content Management:** Leveraging a headless CMS like Contentful makes publishing content manageable.
- **Customer Support:** Keeping customers happy, even when something goes wrong, isn't too much of a hassle with services like Zendesk.
- **AI:** And of course, everyone's favorite topic, incorporating AI through services such as Microsoft, Google, and Amazon's text-to-speech (and vice-versa), recommendation systems, image recognition, and, of course, natural language processing has become a popular way to "make an AI-first app" these days.

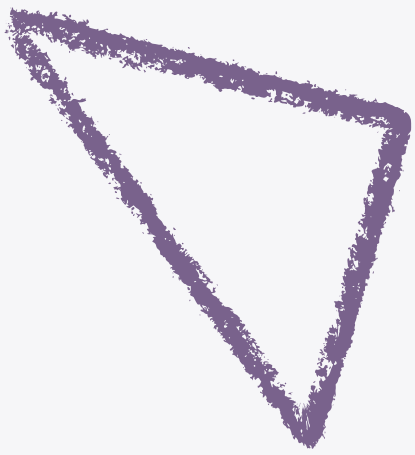
The benefits of microservices

At its core, a microservice is a single-responsibility, well, service that performs a very specific function. The benefit is, in theory, that it is much easier to develop, test, deploy, and debug a small, single responsibility service than it is to debug a full, monolithic application.

Having a microservices architecture makes it easy to remain flexible, which can enable your team to keep the entire system up to date and even enable a faster integration of new technologies. Microservices make it easy to swap out one microservice for another or isolate improvements (bug fixes and performance issues).

In terms of flexibility, microservices are usually preferred compared to monolithic applications, where changes require complex unentaglement.

Isolated testing and reverting is also made much easier with a microservices architecture. Tests can range from testing an individual unit to testing across the entire system, which makes it much easier to find the root cause of an issue because the smaller units are being tested in isolation. And, when an issue does come up, an individual unit can be reverted quickly without affecting the entire system.



The challenges of microservices

As you can imagine, however, the challenges with microservices is the rapid increase in complexity for the application as a whole. While everyone's use case is unique and therefore has unique challenges, there are a few ways in which microservices challenges all of us.

Increased complexity

Keeping data (and types!) consistent across the entire codebase, as microservices change or get swapped out, can pose a challenge. With individual pieces of a system being tested, sometimes the more complex edge cases that are system-wide get missed. Deploying also becomes increasingly difficult as you need solid orchestration and automation tools to ensure they're each up and running and ready in the right order and correctly.

Communication overhead

Perhaps most challenging is the huge communication overhead both within the system and the engineering teams themselves. This overhead affects performance, quality, and reliability of the application. There is only so much you can do to optimize each individual service you're using, and oftentimes you don't even have control over the performance of said service because it's a third-party service. Sometimes the effects of one service show up in another part of your application unexpectedly, making debugging issues your users are experiencing a very stressful scavenger hunt.

Microservices in the wild

That which removes the complexity of building an application, can sometimes bring it back tenfold when maintaining and debugging the same application.

We're now seeing products like Amazon Prime Video moving away from a microservice architecture, improving performance and reducing operational costs by over 90%. The costs of debugging have probably also been significantly reduced since tracing and consistency is likely much easier with a more traditional, monolithic architecture.

And for companies like monday.com who choose to continue using microservices, it becomes critical to use debuggabilty tools, like Sentry, to seamlessly trace across all distributed services, despite the complexity of a growing architecture.

Setting up a Microservices Architecture for success

If you've made it this far, you have a good reason for still wanting to have a Microservices Architecture. Truthfully, there are many reasons why it is better to build with agility and flexibility in mind. Not every project can have the funding and stability of Amazon Prime Video.

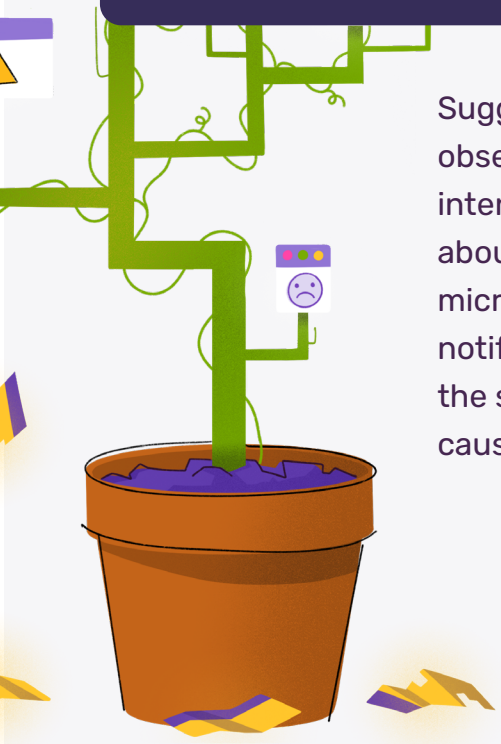
There are ways to mitigate the challenges microservice architectures have, and support effective debugging workflows that make the system relatively easy to maintain.

Focus on Debuggability

Building a healthy system with a microservices architecture heavily relies on one thing:

Having the right tools

When choosing the right tools, consider what you need to know about your entire system, what context you need to have about the issue and who, what, where, when, and why it's happening so that you can find the right engineer to quickly and simply fix the problem.



Suggested tools are often in the IDE, logging, monitoring, and observability spaces. But I suggest you think more monolithically (pun intended) about your actual debugging workflow. Because, I don't know about you, but trying to follow logs for a system that uses a different microservice for authorization, payments, inventory, orders, content, notifications, customer support, analytics, and AI and decipher where the slowdown is, or which customers it's affecting, or which release caused the issue sounds… impossible.

Configure for Debuggability

Let's set up your system's environment. But just like with choosing which microservices to include in your system, you should take time to choose the development environment that is tailored to what you need, not just the newest trend because it's fun (unless, of course, that is what you want to be exploring, which is completely fine).

Here are some basic suggestions for your development environment:

- **IDE:** There are a few IDEs and editors that have become popular depending on what you're building, but according to the 2024 Stack Overflow Developer Survey, Visual Studio Code remains the most popular choice. This is probably because of the flexibility and customization you can have with VS Code. Plus, the options for integrating the rest of the tools that will help you with debuggability make this IDE a common go-to.

- **Source Code Control:** There are many options for source code control, but one of the more popular choices is GitHub. With the integration of GitHub to VS Code, the second most popular AI developer tool with GitHub Copilot, community focus with GitHub Discussions, and strong integrations with other tools throughout your workflow, you can merge to main with confidence.

- **Deployment Environments:** Once you have the basics, you should make sure you have development and staging environments that mirror the production environment as closely as possible. This will help you ensure you can confirm any fixes will actually improve your users' experience.

- **Code Testing:** Making sure your code is covered with your test suite is critical to making sure you are shipping with confidence. Tools like Codecov make that easy, integrating with your other tools, like GitHub.

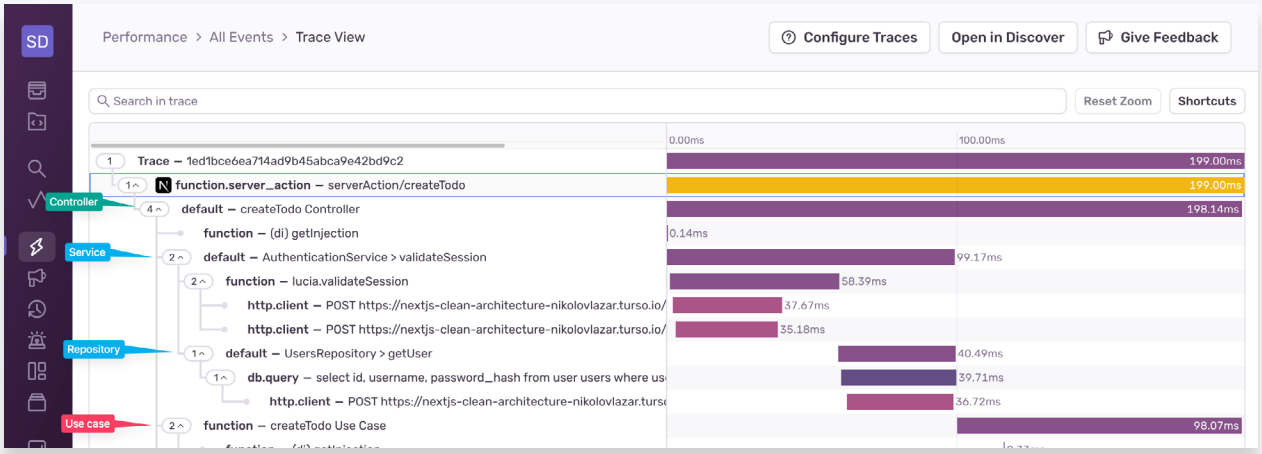
- **Monitoring vs. Observability vs. Debuggability:** Whether you're looking for monitoring, observability, or something more actionable and contextualized like debuggability, being able to stay up to date on performance issues and errors, prioritize those issues quickly, and trace issues throughout your entire stack, microservices and all, is absolutely imperative to your success at debugging a microservices architecture. But more on that later.

Common microservices architecture debugging techniques

When trying to debug an application built on a microservices architecture, you have to be ready to employ many debugging techniques. The distributed nature of the system makes the root cause of the issue harder to find.

Analyzing logs and trace data

The three pillars of observability are metrics, logs, and traces. Metrics can often be the initial indicator that something is going wrong. If there is a major change in how users are interacting with your application, or a drop in clicks or visits to specific areas of your website, you know there is likely an issue that is triggering that change in user behavior. From there, you might have to dig through logs and try to decipher where an issue is occurring. Once you think you've found the culprit, or at least the general area where it might be, leveraging tracing can help you fully visualize and understand where data is moving across your system.



Root Cause Analysis

Debugging with root cause analysis is easier with the support of metrics, logs, and traces. While you are systematically ruling out potential causes, you can leverage the data from observability tools to decide whether a part of the system needs further investigation, or can be confirmed as operational. But what is often missing from traditional observability tools is the context across all dependencies, configurations, users, and code changes.

A tool centered around debuggability, like Sentry, will help you resolve your issues even faster. While you're exploring any span along the trace, Sentry provides context that is relevant for developers to not only pinpoint where the issue is coming from but also determine whether this particular issue should be prioritized over others.

Debugging API calls and response issues

If, through your initial scoping of the issue, you discover that there is likely an issue with a specific API you are calling, it is probably a good idea to test whether the API requests are successful, performant, and whether the requests and responses are conforming to the expected formats. You can use tools like Postman or curl to test endpoints.

See how Dan Mindru shaved off 22.3 seconds on an API call by tracing his system with Sentry's Trace View.

Identifying performance bottlenecks with a performance profiler

If the issue is more of a performance issue, you should be using a profiler. Profiling tools help you monitor CPU, memory, and I/O usage, identify services or endpoints with high latency or resource consumption, and help you analyze the performance of individual services across your system.

Reproduce the bug and find the code causing the issue

Once you have determined where the issue is within the system, you are likely going to want to dive deeper into the code itself. Leveraging an IDE that supports debugging with breakpoints and stepping through code will make this a lot simpler. Running the code in debug mode within an IDE will enable you to reproduce the issue and actually see what is happening in the code at that exact moment.

But what happens when the issue isn't locally reproducible? For example, maybe the issue is a performance issue, and you aren't able to reproduce it because you're on a high-end device with a solid Internet connection.

Debuggability tools, like Sentry, can be leveraged for this as well with features such as Session Replay. Replays are a video-like reproduction of what was happening on the clients device when an error or slowdown happened. Luckily, Session Replay leverages the data that Sentry has across your entire application and contextualizes the issue for you.

Code testing

Often deprioritized when trying to ship quickly and often, testing is a critical component to being able to maintain and fix complex, distributed systems that use a lot of microservices. Ensuring your test framework is up to date and you are always able to check on your code coverage, maybe using a tool like Codecov, will also give you quick insights into coding errors and help you identify where there might be gaps in the system-wide assumptions made on how data flows through and use cases for your application.

Debugging microservices with Sentry

Before you start signing up for a million other tools (in addition to leveraging all of the microservices you're using), consider using a full debuggability tool like Sentry. With just a couple of lines of code and a few minutes, you can make sure your frontend, backend, mobile app… your entire system, is being monitored and giving you the data, context, and tools you need to quickly take action to debug and continue building those newer features.

One thing to be aware of is that not all microservices are called the same way, so it's important to understand how your microservices are being called so that you can handle passing the trace header correctly, so ensure an uninterrupted tracing experience. Check out this resource to learn more about the different ways in which microservices communicate.

Set up Sentry

The first thing you will need to do is set up Sentry. Make sure to check out our sign up page for information on which type of account would be good for you. As an individual developer you can get started with the **free Developer** plan. And don't worry, we have plans that make sense for all size teams and projects; Sentry scales with you.

There are a myriad of Sentry **SDKs that support over 100 languages and frameworks**. Create a project in Sentry for each part of your application (e.g. each microservice, the frontend vs backend), so that you can track errors and performance metrics independently. Don't worry, you will still be able to trace errors and performance issues between Sentry projects.

Integrate Sentry across your entire application, using the appropriate SDK for each part. For example, if you have a Python-based microservice, you would first install the Sentry SDK

```
pip install sentry-sdk
```

Then you would initialize Sentry in your application:

```
from fastapi import FastAPI

import sentry_sdk
sentry_sdk.init(
    dsn="https://<key>@sentry.io/<project>",

    # Set traces_sample_rate to 1.0 to capture 100%
    # of transactions for Tracing.
    # We recommend adjusting this value in production,
    traces_sample_rate=1.0,

    # If you wish to associate users to errors (assuming you are using
    # django.contrib.auth) you may enable sending PII data.
    send_default_pii=True,
)

app = FastAPI()
```

It's really that easy. If you don't believe me, see why Dan realized he had waited way too long to install Sentry, considering how fast, easy, and useful it was.

Enable Distributed Tracing

Distributed Tracing is a must when you're trying to debug a system that leverages a lot of microservices. To get started with Distributed Tracing, it's fairly straightforward. In fact, the init code above already had tracing enabled with:

```
traces_sample_rate=1.0
```

Enabling tracing allows you to be able to trace from an issue your users are facing all the way to a database call and back, and everything in between. The best part is, with Sentry, Distributed Tracing just works out of the box if you're using one of our SDKs. You can check, for example, which Python frameworks we support out of the box on our docs here. And if you're using something else, you can also do custom instrumentation for distributed tracing.

Use Sentry

Ok, this might feel trivial, but truly, the next step is to just… use Sentry. Let your application be used by people and then watch as Sentry captures errors, performance issues, and related contextual data. Then, regardless of how you're being notified of an issue, you can trace down to the line of code causing the issue, automatically assign to the codeowner, discover the pull request that introduced the bug or slowdown, and quickly see what subset of users are being affected.

Sign up for Sentry on any of our plans, starting from a free developer plan to affordable team, business, and even enterprise plans, which you can explore here. Then, get started by installing the SDKs you need for your stack. In a matter of minutes, you will be able to start benefiting from the debuggability support Sentry provides.

For more information on how to get started with tracing specifically, check out these resources:

Docs
Sentry Tracing

Blog
See how Dan Mindru cut 22.3 seconds off an API call using Sentry's Trace View

Customer Story
See how monday.com reduced errors by 60% with Sentry in their microservices architecture

Workshop
Check out this workshop recording on using tracing to identify the root cause of frontend issues

From me

We're always looking to stay connected to our community, drop into one of our communities to ask questions and give feedback, or get a quick demo from a Sentry expert to get your specific questions answers.

Still have questions about how Sentry can help you make sense of your distributed system? Get a demo from a Sentry expert.

To learn more:

Join our Discord

Check out GitHub

REQUEST A DEMO