Navigating the depths of observability: a journey into modern software systems

Introduction

In order to grow my backend skills and my understanding of modern software systems, I decided to take on a challenging project. My goal was to expand my technical abilities and to make this project a valuable asset in my pursuit of a job as a developer.

As I explored potential topics for this project, observability for modern distributed systems catched my attention. I perceived it as an important area of expertise, with a great potential for growth, and essential for understanding the intricate systems I was trying to understand myself. So I made it my chosen focus.

Each challenge I encountered in my research served as a stepping stone for my professional growth, giving me a holistic understanding of modern software systems and enriching my capabilities in troubleshooting and optimizing complex architectures.

During this journey, I gained extensive knowledge in navigating distributed systems and understanding their intricacies, tackled the challenges of microservice architectures, efficiently managed containerized applications with Docker, and learnt to implement observability solutions using the OpenTelemetry toolset.

Looking back, this adventure wasn't just about coding and completing a project. It was a journey of pushing both my personal and professional capabilities and witnessing my skills evolve.

Discovering distributed architectures

My goal was to implement an observability solution for a microservices application, but I didn't have one to test my experiments on, so I decided to build one myself.

Up until this point all the applications I previously built were based on monolithic architectures. That means that all the components shared the same database, business logic and computing resources. The only *distribution* of tasks in those apps were the separation of the logic into layers (controller, service, data access, view, etc). But that was still all inside the monolith, with layers that still depend on each other.

In my application I decided to have three entities or services called bar, brewery and inventory. These would be my microservices. But before I progressed any further, I needed to understand some essential concepts.

Communication between microservices

A microservice is a small, standalone, loosely coupled, distributed service. Several microservices work together to form an overall application. One of the biggest challenges in microservices-based applications is the communication mechanism.

Unlike monoliths, in distributed architectures there could be so many services that need to communicate with each other. The two protocols used most commonly for service communication in microservice architectures are HTTP based API calls and lightweight messaging systems.

In the API calls over HTTP communication model, one service sends a request to another service and waits for the response. Because HTTP is a synchronous blocking protocol, the message exchange blocks the thread and the client code can only continue its task when it receives the response. Therefore this model, although valid, doesn’t provide the full potential of the microservices architecture.

In contrast, message-based communication utilizes message brokers. Unlike HTTP communication, the services don't directly communicate with each other; instead, they use messaging channels. This middleware accepts and forwards messages, enabling services to exchange information in an asynchronous, non-blocking manner, providing an excellent way of communication.

[CHALLENGE: API calls vs msg brokers]
I had extensive experience developing and consuming API endpoints, but didn’t know much about message brokers; also, I learnt that message brokers were a widely used standard for microservice communication, so I opted for this over the API calls model.

Message brokers offer two basic message distribution patterns called Point to Point messaging (each message in the queue is dispatched to only one recipient and is consumed only once) and Publisher/Subscriber (pub/sub) messaging (a broadcast-style distribution method, where publishers post messages into message broker systems without knowing which subscribers are there).

[CHALLENGE: Point to Point vs pub/sub]
In my app, since the communication needed between services is a direct one-to-one message exchange, I decided to implement a point to point pattern, where messages are sent to known queues to be consumed by a single service.

[CHALLENGE: what broker to choose]
Among the popular message brokers I chose RabbitMQ. This open-source message broker server is indicated for communication and integration within, and between applications where a system simply needs to notify another part of the system to start to work on a task.

Implementing RabbitMQ in my application

There are different ways to use the RabbitMQ server. You can easily download it from the official website and install it on your machine as it's open source. As a MacOS user, the easiest way to install RabbitMQ is using the package manager `Homebrew` (follow the instructions detailed here).

Alternatively there is an easier way to experiment with RabbitMQ, using a Docker image maintained by the community. With a simple command we deploy a local RabbitMQ instance in a Docker container:

```
docker run -p 5672:5672 -p 15672:15672 -d --name rabbit rabbitmq:3-management.
```

It is relevant to mention that `rabbitmq:3-management` on the command indicates the version and plugin usage of the RabbitMQ Docker image.

Also, the exposed port `15672` is the default port for the management UI for the `rabbitmq_management` plugin. The management plugin provides a browser-based UI for monitoring and managing RabbitMQ, which is accessible on `http://localhost:15672/` (use `guest` for both the username and the password when prompted). This interface visualizes your RabbitMQ instance and shows, for example, the current message rate, queues and exchanges created, and the bindings between them. Besides the UI, the `rabbitmq_management` plugin also includes a HTTP-based API

that we can use to interact with RabbitMQ, which is accessible on http://localhost:15672/api.

Finally, port `5672` is the default port for RabbitMQ when using the AMQP protocol (Advanced Message Queuing Protocol). In my case I use the `amqplib` library in my application, a widely used AMQP client library for Node.JS, so it makes sense to expose the port `5672`.

Later we'll learn how to deploy a RabbitMQ instance in a Docker container using `docker-compose`.

The steps required to use RabbitMQ are simple, especially if the communication model is point to point: a producer (microservice A) sends a message with a task to be accomplished by a consumer (microservice B). For example to update an inventory or to fetch some data from a database, etc.

To do that we need to create a RabbitMQ server connection (a connection is a TCP connection between your application and RabbitMQ) using an AMQP client (in my case `amqplib`); create a communication channel (a channel is a virtual connection inside a connection) and assert a new queue with a name of our choice. When using a local instance of RabbitMQ (as in my case, running it on a Docker container), the default URL location of the broker connection is `amqp://guest:guest@localhost:5672`. If we were using a different connection method, like the one provided by message broker managers (e.g. CloudAMQP), this URL would be different and would probably be stored in an environment variable.

Finally we need to indicate the service to `consume` from that queue, so microservice A would be listening to messages sent to that queue in that channel.

To send a message out to a consumer, we sent a message to the *known* queue of that consumer. So we are making microservice A to send a message to microservice B on its known queue name. The messages to and from the queues in RabbitMQ are sent using a buffer data structure. Buffers refer to a region in memory used to store data temporarily while it is being moved from one place to another, and are usually used in FIFO (first in, first out) methods (like the ones used by message brokers).

[REFERENCE HERE RABBITMQ CODE FROM MY APP]

Designing a microservice architecture

Now that we understand how different services in a microservice architecture communicate with each other and we have decided to use RabbitMQ as a message broker, we need to understand how to organize our codebase.

Micro-services

A fundamental principle to follow when designing microservice based applications, is the SRP or Single Responsibility Principle. That means avoiding the creation of services responsible for numerous tasks. Simplifying our task involves identifying distinct business capabilities and constructing corresponding services for each. A key advantage of microservice architectures lies in the autonomy of each service, allowing independent teams to manage them. Services can be developed using languages and toolsets tailored to their specific needs and then implement ways for different services to communicate with each other, as we've seen in previous sections.

We already know our application will consist of three server-side services: an inventory service that will persist data in a mongoDB database, a bar

service that will fetch data from the inventory, and a brewery service that will feed new data to the inventory for storage. Since the focus of this project lies on backend development, we haven't implemented any frontend oriented service.

Once we understand how our application should be divided in separate responsibilities, architecting these responsibilities into separate services is as simple as treating them as individual projects, located in separate directories, with their own dependencies and versioning (I use Node.js, so I need a separate `package.json` for each service). Each service listens to a different port and runs separately in different terminal windows. Although they are separate projects, they can all make part of an unique application directory and therefore an unique repository.

Code organization / version control

[*CHALLENGE: monorepo vs poly repo*]
When deciding how to structure a microservice architecture in terms of version control systems, like GitHub, there are two main options: a Mono-repo and a Poly-repo.

In a Mono-repo setup, the entire project resides in a single repository, with each service placed in separate directories. On the other hand, a Poly-repo approach involves placing each microservice in its own separate repository.

Each option has its advantages and drawbacks, and the decision rests entirely with the developer team, based on their requirements. In my situation, considering the project's size and simplicity, I opted for a monorepo as it offered better manageability and fulfilled the project's needs effectively. Larger and more complex projects may find the isolation provided by a poly repo beneficial, but this wasn't applicable in my case.

The codebase of each service will maintain a well-defined directory structure. This may typically include directories for controllers, models, services, and more, with different code files within. These files encapsulate the business logic of the microservice, focusing tasks such as handling incoming requests, data processing, database interactions, and any other service-specific functionality.

Moreover, each microservice's codebase will define and expose different API endpoints. These endpoints implement routes or handlers responsible for processing incoming requests and returning appropriate responses. This can involve RESTful API endpoints or alternative communication mechanisms like message brokers, as demonstrated previously.

Finally, it’s worth mentioning that each individual service can be containerized using technologies such as Docker. While this approach offers benefits in terms of consistency and portability across various environments, it's not mandatory. We will revisit the implementation of our services in Docker containers at a later stage in the project.

Discovering Observability

Monitoring traditional systems vs *observing* modern systems

Traditional system architectures are primarily monolithic, characterized by a static and long-running configuration of nodes, containers, or hosts, usually managed by a centralized team. Managing the potential issues that may arise in such systems during production is relatively manageable. Developers address these challenges by proactively identifying and resolving errors during development, implementing monitoring solutions to alert them about *known* issues and challenges.

The conventional monitoring practices are built upon numerous implicit assumptions about the systems they aim to monitor (monolithic application, single data store, access to system metrics, control over containers or virtual machines, etc). As developers increasingly implement modern approaches to deploying software systems, the limitations of these assumptions become more apparent.

In modern distributed systems the prevalence of microservices architectures, SaaS services, and third-party APIs has brought in a new level of dynamism and flexibility, but also extra layers of complexity. These systems, often outside the direct control of a single team, with numerous services involved interacting with each other, are difficult to understand, and issues with one of these services can be challenging to identify when they occur.

The traditional practice of monitoring systems presenting metrics, such as response time, latency, and error rates, through intricate dashboards, to understand and anticipate every possible failure isn't very useful with modern distributed systems. In the new scenario failures can occur in unpredictable ways, making it really hard to monitor for every potential failure mode.

Observability techniques, with the introduction of distributed tracing and the combination of traces, metrics and logs, helped in tackling these issues.

Observability systems, instead of collecting and analyzing metric data, are based on the idea of preserving as much of the context around any given request as possible, so that developers can reconstruct the environment and circumstances that triggered the bug that led to the failure.

Observability isn’t about sending an alert once some issue on the system is impacting the user experience, but rather, the ability to examine the entire

system and user experience, in real time, to detect anomalies before it affects the user experience and gaining the ability to identify what failed, when did it fail and where did the failure happen.

Identifying the origin of an error, pinpointing where latency is introduced, determining where data was altered, understanding which component is consuming the most processing time, and assessing whether wait time is evenly distributed across all users or experienced by only a subset are critical questions addressed through observability.

Despite all of that, it’d be necessary to clarify that monitoring systems still continue to provide valuable insights. We can say that monitoring is best suited to evaluating the health of systems (the infrastructure a developer needs to run in order to support the software they want to run), while observability is best suited to evaluating the health of software (the code that delivers valuable services to customers).

Telemetry data

Telemetry data is a collection of data entries that represent information about an application's state, and can be used for monitoring and observability purposes. There are three main types of telemetry data: metrics, logs and traces.

Metrics

A metric is an aggregated numerical value that represents a system state at the particular interval of time when it was recorded (e.g. % of memory in use, requests per second, average response time for components to render, latency, error rates, status code frequency, …). Metrics are usually represented in charts and graphs in monitoring dashboards and help

understand the overall performance of an application. Metrics are intended to provide statistical information in aggregate.

Metrics are helpful for assessing the system's health status at a macro level, allowing questions like “is the application slower than expected?” or “are users encountering errors?” to be answered, but don’t provide the detailed context of individual transactions or requests or the root of an issue.

Logs

A log is a timestamped record of an action or event that occurs in a system or application during its operation. Logs are the go-to place for developers to try to understand system behavior, as they provide insights into raw system information about what a piece of software is doing and how it’s behaving in real-time, including code execution, performance, errors, warnings, and other relevant events.

Unfortunately, unlike traces, logs are not associated with a particular request, lacking contextual information, such as where they were called from, so they are not very useful to track code execution. Logs help with understanding an issue once you know where to look.

Traces

Distributed traces, or simply traces, are detailed chronological records of the progress of a single operation (i.e. a request) through the different components of a system, revealing its path (where it starts, where it goes to next, and where it finally terminates), performance (behavior), and relationships between services.

Traces complement logs. While logs provide information about what happened inside the service, distributed tracing tells you what happened between services/components and their relationships. This is extremely important for microservices, where many issues are caused due to the failed integration between components.

Traces are made of spans; events that represent a specific operation or processing step that a request makes (e.g. a single method call, a database query, an API call, etc.). Spans are linked together into traces thanks to context propagation, a technique that allows certain metadata representing a shared state (a context) to be passed along as a header to each hop of a request execution. By adding context in the form of many key-value pairs to each event, developers can create a wide, rich view of what is happening in all the parts of their system.

Distributed traces provide you with visibility into the system and across all services and the relationships between them. You can see the journey requests went through, how long they took, insights into system health, and more. You can use distributed tracing not only for identifying why a problem occurred, but also to avoid problems with ongoing observability and tracking.

With observability solutions, instead of sending spans to a tracing tool, logs to a log management, and metrics to a monitoring tool, we send all that telemetry data to one central place, an observability backend.

Observability with OpenTelemetry

After understanding the basis of observability, it was time to learn how to implement it in my application. That led me to discover OpenTelemetry

(OTel), a powerful tool for instrumenting applications to generate telemetry data, that has become the industry standard in the practice of observability.

OTel has libraries available in a number of programming languages. Each language gets an API that implements the OTel Specification, plus a software development kit (SDK) that builds on top of the API. OTel's SDKs include auto-instrumentation for things like HTTP or gRPC requests, database queries, and many framework-specific features. One of the advantages of OTel is that telemetry data can be collected from an application using automatic instrumentation, without needing to modify the source code. That out of the box instrumentation gets added to the code as a dependency.

Automatic instrumentation is a good place to start, but the highest-value instrumentation will be specific to the needs of your individual application. To enrich the instrumentation of applications, and get customized observability data, manual instrumentation should be considered. Manual instrumentation allows things like adding tracers, creating multi level spans using tracing context, adding custom attributes to spans to add extra information about the current operation that the span is tracking, etc. Manual instrumentation is not mutually exclusive to automatic instrumentation. You can use both side by side.

Once we have gained the ability to generate and collect telemetry data from our application or services, we need a place to send that data to to be stored, displayed and analyzed.

Distributed tracing backend solutions

Distributed tracing backend solutions act as the central hub for processing and analyzing telemetry data collected from applications, providing a comprehensive and real-time view of the performance and interactions

within a distributed system. Their role is crucial for achieving observability and facilitating effective troubleshooting and optimization efforts in complex, modern application architectures.

Several backend solutions are available for implementing distributed tracing:

Open-source solutions

Assembling an entire telemetry pipeline involves integrating multiple open-source observability components, which demands thorough research and understanding of each piece and how they fit together. This process entails manual maintenance, deployments, and scaling, resulting in a steep learning curve. However, it offers the flexibility to incorporate business-specific features, ensuring bespoke solutions and complete data ownership. While some components may be inexpensive or free to obtain, the investment in engineering time for setup and maintenance can be considerable. Available components include OpenTelemetry, Jaeger, Zipkin, ELK, Fluentd, Timescale, Influxdata, Prometheus, and Grafana.

SaaS solutions

Besides the open-source solutions there are many Software-as-a-Service commercial solutions out there, such as DataDog, NewRelic, Honeycomb, AWS CloudWatch, etc. These are out-of-the-box solutions with minimal setup required; the pipeline is managed by a third party, so there is no need to maintain, scale or deploy anything. Usually these solutions present a very rich variety of features, which sometimes can be overkilling for small applications. Also these can get quite expensive as soon as the app grows and data generation and management grows too. The other problem they present is that your data will reside on a third-party provider’s infrastructure, with little control on your hands.

Instrumenting my application to collect traces

In the first version of my application, observability was implemented in the most basic and direct possible way: traces automatically generated from the application services, using the OpenTelemetry SDK and some auto instrumentation for different modules (http, express, rabbitmq, mongodb…), and from there exported directly to a observability backend solution, in my case Jaeger.

The benefit of this direct-send approach is that I didn't need to run any additional software alongside my application; there were no ports to configure or processes to manage. The downside is that my application had to manage the entire workload of collecting and sending the collected data, besides its normal operation.

In order to implement this model I created a simple tracer file (`tracer.js`) where a number of OTel libraries are used to generate, process and export telemetry data. To make the tracer do its work on the different services of my application, the `tracer.js` must be required and initialized at the top of the service file, before any other code is run.

Jaeger-all-in-one

Jaeger-all-in-one is an executable designed for quick local testing. It launches the Jaeger and all its components (user interface, collector, query and agent), with an in memory storage component.

The simplest way to start the all-in-one is to use the pre-built image published to DockerHub. This can be achieved with the following command, which provides a self-contained environment for exploring and testing Jaeger, an open-source, end-to-end distributed tracing system.

```
docker run --rm --name jaeger \
  -e COLLECTOR_ZIPKIN_HOST_PORT=:9411 \
  -p 6831:6831/udp \
  -p 6832:6832/udp \
  -p 5778:5778 \
  -p 16686:16686 \
  -p 4317:4317 \
  -p 4318:4318 \
  -p 14250:14250 \
  -p 14268:14268 \
  -p 14269:14269 \
  -p 9411:9411 \
  jaegertracing/all-in-one:1.53
```

This command launches the Jaeger all-in-one container, exposing various ports for different agents and collectors components of the Jaeger system. Once the container is running, you can access the Jaeger UI at `http://localhost:16686` to visualize and analyze traces.

After setting Jaeger up and having an instance of its backend running on our local environment, we are ready to receive trace data. There is no need for further configuration of Jaeger. The Jaeger backend can receive trace data from the OpenTelemetry SDKs when using the native OTel Protocol (OTLP).

In the `tracer.js` file I use the OTel Collector Exporter for web and node, which is a module that provides a trace-exporter for OTLP protocol. This module can be required in our code with `@opentelemetry/exporter-trace-otlp-http`.

With this module in place it is no longer necessary to configure the OTel SDKs with Jaeger exporters, nor deploy the OTel Collector between the SDKs and the Jaeger backend. The traces will be sent directly from the SDK to Jaeger in the correct format.

To see this in practice we first need to run instances of RabbitMQ and Jaeger-all-in-one, as explained in previous sections, and then run the three services in separate terminal windows. Note that the message broker and tracing backend we are using are running in Docker containers, and to use Docker commands, you need to install Docker on your system.

After that, when an API call is made to any of the services endpoints, the application will generate a number of traces (http calls, express calls, mongoDB calls, rabbitMQ calls, etc) which are exported to Jaeger and can be seen in the Jaeger UI on `http://localhost:16686/`. This procedure is pretty basic and standard to test observability in local environments. And I've managed to make it work, sweet!

Jaeger UI

[Insert screenshots from my app's Jaeger]

The Jaeger user interface is a powerful tool that helps us understand distributed architecture. Once again, once we have a jaeger instance running correctly, the UI is accessible on `http://localhost:16686/`.

There we'll see a search pane, where we can choose which service we want to see traces for. Within each service, we can search for all operations the system traces or for specific ones. In my application for example I can search for traces generated by the bar service and specifically the message broker spans; or I can choose to search for the inventory service and all the spans of any kind included in the tracing. We can also refine the search by using tags, defining time frames, etc.

Once we have defined our search and click on Find Traces, all the traces that meet the search requirements will be displayed in chronological order.

Each item on the list corresponds to a trace and will show some basic information, like the service and operation names, the duration of the execution and the time it was created. For more detailed information we click on the desired trace so we can drill down on its details. Here we can see specific information about execution times, which calls were made and their durations, specific properties like http status code, route path and more.

Diving Deeper: The OTel collector

The OTel Collector (OTelCol), a key component of the OTel project, functions as a versatile, vendor-agnostic agent designed for collecting, processing, and exporting telemetry data consistently across various services and systems.

From no collector to collector gateway deployment

Initially I implemented my observability solution without using the additional OTel collector. I learnt that sending your data directly to a backend is a great way to get value quickly. Also, in a development or small-scale environment I could get decent results without a collector.

While the simplest approach for experimenting with OTel involves not utilizing a collector, it is generally advisable to use a collector alongside your service.

The goal would be double: to offload the processing of telemetry data from the application, enabling it to focus solely on business logic, and leave the manipulation of the data (filtering, sampling, attribute manipulation, etc.) in the hands of the collector, right before the data gets exported to a database or tracing backend.

The collector also supports multiple input and output formats, allowing it to integrate with different data stores, monitoring systems, and analysis tools.

Collector pipeline: configuration and deployment

The OTel Collector employs pipelines for data receiving, processing, and exporting. Each pipeline can be configured with components that include Receivers, for accepting telemetry data from applications, Processors, for shaping or altering the data, and Exporters, for sending data to desired observability tools in the appropriate format.

Now I needed to decide what collector deployment pattern was more suited for my app. OTel collector can be deployed as a standalone agent or as a gateway.

The decision to deploy the OpenTelemetry (OTel) Collector as either a gateway or an agent depends on the architecture and specific requirements of your observability infrastructure. As a gateway, the collector serves centrally to gather and process telemetry data from multiple applications, reducing individual service loads, optimizing network traffic, and ensuring consistent system-wide configuration. This centralized approach facilitates exporting telemetry data to various backends, making it ideal for environments with numerous microservices or centralized management needs.

In contrast, deploying the OTel Collector as an agent adopts a more decentralized approach, with each service managing its own telemetry data export. This provides flexibility for tailoring configurations to individual service requirements and simplifies integration with existing deployment

practices. The lightweight nature of this deployment model allows for customizing configurations according to service needs.

In some cases, a hybrid approach combining both gateway and agent deployments may be employed to strike a balance between centralization and decentralization, accommodating diverse system needs. Ultimately, the choice between gateway and agent deployment depends on factors such as network topology, scalability, and desired telemetry data control levels.

For my project I chose the collector gateway pattern.

Deploying collector: docker-compose and other YAML configuration files

[*CHALLENGE: HOW TO DEPLOY COLLECTOR*]
There are numerous methods to deploy the OTel Collector across various operating systems and architectures. It can be manually deployed using an installation package or via container images for Docker or Kubernetes. Alternatively, we can incorporate the Collector into a `docker-compose.yaml` file and configure it separately in another YAML file.

Docker Compose serves as a tool for defining and running multi-container applications, simplifying control over the entire application stack. It facilitates management of services, networks, and volumes through a single, easily comprehensible YAML configuration file. With a single command, `docker-compose up`, all services from the `docker-compose.yaml` configuration file are created and initiated.

In my case I chose to deploy the collector as a containerized instance using docker-compose.

[*CHALLENGE: JAEGER NOT IDENTIFYING SERVICES FROM COLLECTOR*]

As the complexity grew, so did the challenges. Configuring the docker compose and YAML files for the various services proved to be a daunting task. The real test came when setting up Jaeger alongside the collector. Despite successfully visualizing traces with the initial basic setup (the no-collector scenario), integrating the collector in my architecture introduced unexpected issues. To resolve this, I delved into documentation, experimented with configurations, and eventually identified a compatibility issue. Downgrading one of the OTel dependencies proved to be the key, and the traces from my microservices were finally displayed in Jaeger as intended.