DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Microservice Architecture in Practice: Debugging the Behaviour of Concurrent Applications at financial.com AG

Jonathan Ryan Wijaya Tumboimbela

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Microservice Architecture in Practice: Debugging the Behaviour of Concurrent Applications at financial.com AG

Microservice-Architektur in der Praxis: Debuggen des Verhaltens nebenläufiger Anwendungen bei financial.com AG

Author:	Jonathan Ryan Wijaya Tumboimbela
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	M.Sc. Emmanouil Giortamis, Dr. rer. nat. Herbert Reiter
Submission Date:	28.11.2022

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 28.11.2022 Jonathan Ryan Wijaya T.

Acknowledgments

Firstly, I would like to express my deepest appreciation to financial.com AG in Munich and to the Chair of Decentralized Systems Engineering at the Technical University of Munich, both of which are pivotal for making this project possible. Words cannot express my gratitude to both of my advisors, Dr. rer. nat. Herbert Reiter from financial.com AG, and Emmanouil Giortamis, M.Sc., who helped and guided me since the beginning of this project with their invaluable feedbacks. Additionally, I am also extremely grateful to Prof. Dr.-Ing. Pramod Bhatotia as the head of the Chair of Decentralized Systems Engineering at TUM for providing me with the opportunity to do this research work on the DSE Chair.

Special thanks to the Green Team at financial.com AG, which supports me with my implementation work on the Dory REST platform with their expert knowledge and guidance, generously sharing feedbacks from their valuable experience in the industry.

Lastly, I would also like to extend my sincere thanks to my family and friends for their unrelenting emotional support. Their belief in me has kept me motivated through this journey, and helps me finish this Thesis well.

Abstract

Microservices Architecture (MSA) has recently been gaining popularity, promising flexibility and modularity — both very important aspects in designing software. They are quickly becoming mainstream as the new paradigm to build complex software systems — with several big tech companies such as Amazon, Microsoft, IBM, and Netflix rapidly adopting MSA as their preferred operating model for developing modern applications [1], [2].

While the benefits of MSA are really felt in practice, they are not the silver bullet which solves all problems of designing complex software solutions. With MSA, the individual services are not controlled and integrated together by a single centralized component as in Service-Oriented Architecture (SOA), as the individual microservices themselves are the one responsible for interacting with each other, and individually deployable as a *single unit*. While this means greater modularity and flexibility, the downside on the other hand is the loss of coherence in the application itself. As the work to fulfill a request is now broken down and distributed between the multiple services — each with its own stacks and implementation languages — the complexity can grow really quickly. The cost of maintaining consistency across the system, monitoring, alarming, and fault tolerance are particularly more expensive and time-consuming [3]. Concretely, this means that the process of troubleshooting and fixing software defects are a lot more complex for the developers.

For this purpose, the research will be divided into two main parts. The first part will focus on uncovering the current state-of-the-art of the process. Particularly, the disadvantages of designing such complex systems using MSA in comparison to the more traditional way will be explored in depth — both from the relevant literatures and from the actual observation at financial.com AG in Munich, which provides the opportunity for conducting this research work. The second part will then turn to focus on the actual solution to the problems uncovered in the first part, and will be done on actual running projects at the respective software company. In the end, the goal of this research work is to give a contribution to how one may approach and solve the biggest problems currently faced with MSA, and the demonstration of such solutions on real industrial projects. The proposed solution will then be evaluated at the later stages of this Master's Thesis, which may lead to further research questions, and can serve as basis for future works.

Contents

1 Introduction

The landscape of software development in business context is changing almost constantly. In particular, the area of software architecture design patterns has been seeing changing trends over the course of past years. This is understandable, as software architecture has long been recognized as one of the key factors for software quality [4], and is therefore very important in the early phases of developing new software projects [5]. Moreover, good software architecture design should offer a great degree of reusability, as this is very important to enable rapid development and deployment of complex software systems, while maintaining minimum engineering efforts and costs. The pressure to have reusable components stems from the fact, that most enterprises today are operating a wide range of different software systems and applications — which may well be from different ages and use different technologies. Concretely, this means integrating multiple products across different platforms can be a really difficult challenge.

This trend can be seen over the last decades, for example, as Service-Oriented Architecture (SOA) has been popular as the standard approach to develop complex enterprise applications. While SOA originally promises greater reusability and loose coupling of individual services by the means of decomposition, it still binds all services to a single general context with the use of Enterprise Service Bus (ESB). This results in the individual services having *interdependencies* with each other — even though the services itself are technically single independent units. In an extreme case, this may lead back to a single large monolithic application to be deployed, diminishing the benefits.

In recent years, Microservice Architecture (MSA) has been emerging as a reimplementation of SOA which offers the true flexibility and modularity of the original concept [1], and are quickly becoming mainstream as the new paradigm to build complex software systems. This trend can be clearly seen with multiple tech companies rapidly adopting MSA to develop their products [1], [2]. While SOA focuses on integrating *"dumb"* services with smart routing mechanisms (e.g. the ESB), Microservice Architecture prefers *smart* service interfaces communicating through simple (dumb) routing mechanisms [6], e.g. REST through HTTP. This of course results in a greater degree of autonomy and naturally, the decoupling of the services in contrast to SOA — delivering the original promise of flexibility and reusability.

1.1 Motivation

As described above, MSA solves the interdependency problem of SOA by providing greater autonomy to the individual *services* — each designed to perform a very specific

task or function well, and contains all the necessary dependencies to carry out their tasks. While the benefits of this architectural style are of course significant, unfortunately it also creates another set of problems not previously encountered. One of the biggest issues of MSA is the difficulty of monitoring/troubleshooting [7]. This is understandable, as the application is not a single monolith anymore where troubleshooting is a lot simpler.

With microservices, the request is broken down and handled by different individual applications. As the individual services are often implemented in different languages or technology stacks, the complexity grows really quickly. In particular, the cost of maintaining consistency across the system, monitoring, alarming, and fault tolerance can tremendously be more expensive and time-consuming [3]. Concretely, the process of troubleshooting and fixing software defects is now a lot more complex for the developers.

The main purpose of this Master's Thesis is then to gain valuable inputs from the industrial scene. The study will be done with the cooperation of financial.com AG — a software company in Munich which develops tailored financial market data solutions for its clients. In the respective company, this trend of developing modern applications according to MSA can be observed in the recent years, and hence suitable for this study. The trade-offs and newly introduced problems in the context of the relatively young architectural style will be explored in depth, and will serve as the problem statement. In the second part of this thesis, possible solutions will then be explored to be later evaluated and presented at the end.

1.2 Research Questions

The following research questions will serve to guide the scope of this Master's Thesis, and are derived from the motivation presented above:

1. *What are the challenges of operating and maintaining microservices at financial.com AG? Which specific problems are uniquely present in developing MSA-based software projects in comparison to older architectural styles such as SOA or monolithic?*

2. *How can one possibly approach and solve the unique challenges introduced by microservice architecture? How does the solution impact the performance of the applications?*

The objective of this Master's Thesis is, first and foremost, to explore the challenges and additionally the possible solutions to the uncovered problems of the current trend of microservices in a real industrial setting — in this case at the company financial.com AG. Therefore, the aim of the first research question is to gain a better understanding of the problem at hand, and will focus on elaborating the current situation of operating microservices based projects at the respective company. The results will then guide the direction of the potential solution being explored in the later stage of this thesis.

The second research question, on the other hand, will focus more on the solution itself to the aforementioned challenges of this architectural style. As the nature of this question is intended to be open-ended and more about the actual implementation of

the solution to real running projects of the aforementioned cooperating company, the possible solutions may dynamically evolve in the course of this research work according to the actual requirements, and will become clearer towards the end of this thesis — as the main goal of this thesis is to gain valuable experiences from the industry.

2 Background

2.1 Evolution of Architectural Trends

The evolving trends of software architecture — especially in an industrial setting — can be clearly observed over the years. This is also true at financial.com AG, in which monolithic software projects are constantly being modernized or replaced by newer, more modern architectures which suit the evolving demands of the customers. In the below sections, the different architectures which are relevant for the respective company will be discussed, to serve as theoretical background for this research work.

2.1.1 Monolithic Architecture

Arguably the oldest architectural style than the other two which are discussed in this study, in a monolithic architecture, a single big program does everything which is needed — from the UI, data access, and whatever else the application has to do [8] — as seen in Figure 2.1. While there are of course benefits of designing such monolithic applications, such as the reduced communication complexity across networks — which means easier debugging, as all of the processing is handled by a single application, lack of network latency or disruption problems, and better security [8] — there are some significant drawbacks. The biggest problem is the fact that pieces of the system, which are responsible for different things, are tightly coupled together. This results in a very rigid architecture, as when developers want to add or change features, they need to guarantee that their changes do not cause the other pieces of the system to stop working [8], [9] — increasing development costs and reducing productivity. Consequently, this also represents a single point of failure as the whole set of functionalities will not be working if there is a fault in the software [9].

Furthermore, a monolithic architecture forces the developers to understand how all the functional units of a system work together from the very beginning of the project, as the costs of redesigning or refactoring at the later stages are high because of the aforementioned tight coupling and little flexibility.

2.1.2 Service-Oriented Architecture

As the application grows in scale — in terms of its users and also its functionalities — monolithic design starts to show its main weakness, which is scalability. Simply scaling up the application instances to handle more users proves to be a challenge. Even when designed as monolithics, software applications are typically composed of lots of

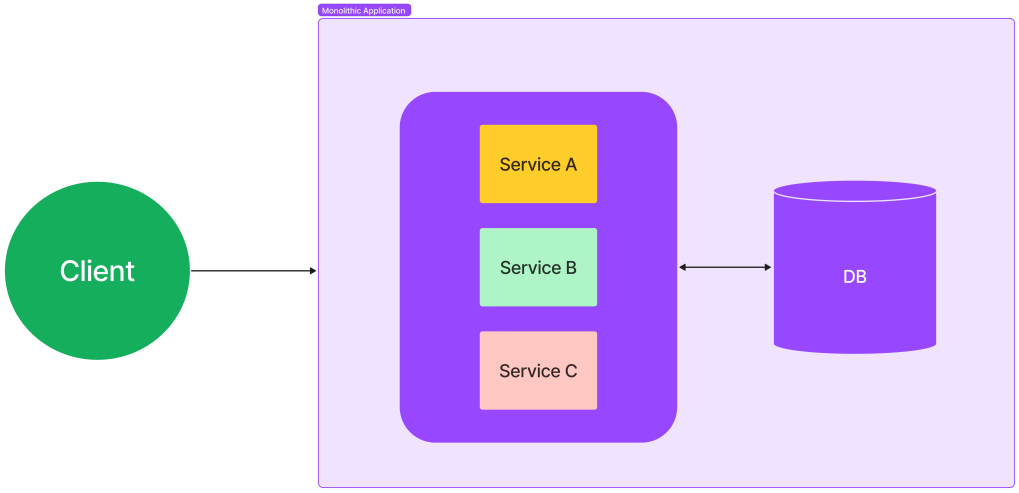

Figure 2.1: Monolithic Architecture.

individual services, which can be thought as single units of logic — some of them can have more demand, and some less. If the demand calls for the application to be scaled up, the less used services will also be scaled at the same time, which is inefficient as the less popular services also consume large amount of resources, even when they are not going to be used as much as the others [9].

Service-Oriented Architecture (SOA) is the first step towards this paradigm of organizing and packaging individual units of functionality as *services* across the network, to be invoked by each other via well-defined interfaces [10]. This can be seen as the natural evolution of the functional level of abstraction — from modules, to objects, and now services. A service is a fundamental building block in developing applications, that *(1)* combines information and behaviour into a single logical piece, *(2)* masks the internal logic from outside access, and *(3)* has well-defined interface to be invoked upon [11]. With the advent of web services, simple client-server distributed architecture evolved into the wider picture that is SOA. This comes from the fact that web services offer architectural benefits such as platform independence, loose coupling, self description, discovery, and formal separation between provider and consumer as it enforces well-defined interfaces [11], [12].

While from a consumer's perspective, the services themselves are black boxes — a consumer describes its request, and receives the result or has its intended side effect operation executed by the service [10] — these functionalities may be composed of a sequence of actions triggered by the request which is for the consumer unimportant. This is exactly the main selling point of SOA — splitting the interface from the implementation itelf, and hence encapsulating a sequence of complex operations which are all part of a specific unit of functionality into an independent unit of software — which solves a

specific business problem and automates the process for solving it, to be reusable across the entire system, and *in theory* means independent deployment and scaling.

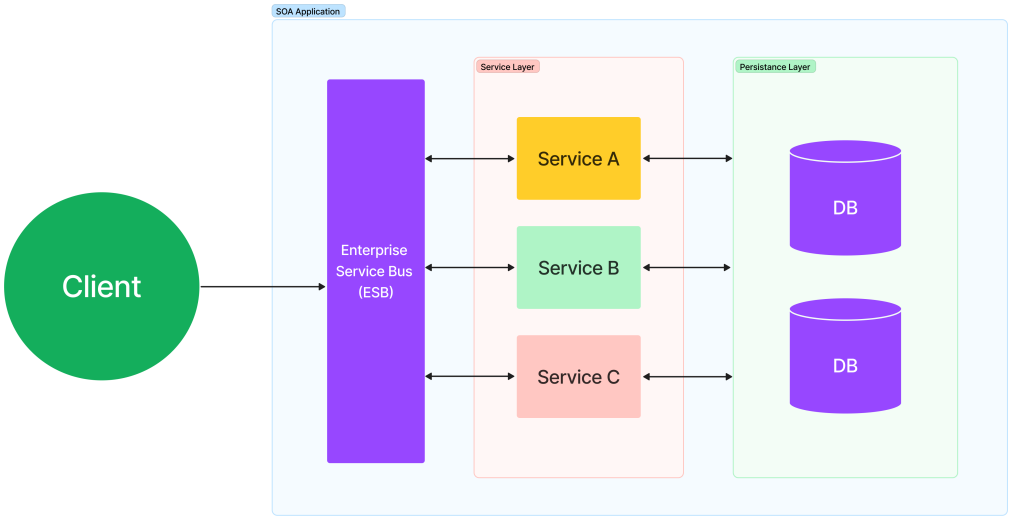

Figure 2.2: SOA Architecture.

For a system to fulfill the SOA paradigm, it needs to provide a means of clients, or service consumers, (1) to find each other (e.g. service discovery), (2) means of interaction through service interfaces, and (3) must produce real world effects to solve the consumer's business problem [10]. Although SOA has been poised to solve the requirements of software companies at that time, which promises better reusability and flexibility unlike monoliths, in practice these implementations are expensive, time consuming, and complex. The challenges of implementing SOA is known in the business and academia, the most prominent one being the difficulty of migrating older monoliths into SOA because it is not at all straightforward [13], [14]. Furthermore, because of the design principles of SOA, there is a need for a centralized control component, usually in the form of Enterprise Service Bus (ESB). ESB is a message backbone component for implementing, deploying, and managing SOA-based solutions. Its two key objectives are to (1) enable loose coupling between the services, and (2) break up the integration logic into distinct, manageable pieces [14] — therefore acting as a *centralized* middleware for the different services. As a consequence, the ESB becomes the single *glue* that integrates all the pieces of a system together. While this may fulfill the needs of some companies back then, when used in this age of modern internet scale applications with millions of users, they become the bottleneck, and worse, a single point of failure in the entire system [9]. Furthermore, ESBs were not designed with the advent of cloud computing in mind. This makes it difficult to add or remove resources from them on demand — therefore complicating the addition of new requirements from end users and requires a

lot of complex configuration, hampering agility and productivity.

2.1.3 Microservice Architecture

In the recent years, microservice architecture (MSA) pattern has been emerging as a more lightweight reimplementation of SOA architecture pattern, in part due to the additional requirements of running internet scale applications. Similarly to SOA, MSA proposes to functionally decompose monolithic applications into a set of loosely coupled, collaborating services [15]. Each service should be:

- Highly maintainable and testable, to enable frequent development and deployment;
- Loosely coupled with other services, increasing developer productivity;
- Can be deployed independently of each other;
- Can be developed and maintained by a small team — hence each team should have every capabilities required including the presentation, business, and persistence layers.

Microservices adopt SOA concepts, but with an architectural style more focused on simplicity, avoiding the centralized orchestration of ESBs — allowing developers to quickly scale and deploy applications, suitable for the new trend of agility and cloud computing in software development [9]. While both agree that decomposing a system into loosely coupled service components is generally desirable — as it lets different parts evolve at different rates — a big difference lies in how both architectural styles enforce *interface constraints*.

In the presentation layer, the services typically communicate with each other using Representational State Transfer (REST) principle. The REST standard specifies several architectural constraints as described in [16], including the central feature which distinguishes itself from other distributed architectural styles — the *uniform* interface constraint — whereas SOA regards interfaces and contracts are inseparable the service definitions themselves. Therefore, axccording to SOA, different services having different interfaces is a normal and *desirable* characteristic of software systems [17].

This difference in interface constraint is exactly the reason why MSA is more *scalable* when compared to SOA. In the case of uniform constraints, a client must only understand the specific service's data contract (e.g. the returned data format and how to handle it). In SOA, the client needs to additionally understand the specifics of *both* the service's interface and data contract [17]. Consequently, it is impractical on large-scale systems for a client to interact with a set of different services because of the possible differences in each service's interface — therefore leading to the development of centralized middleware solutions, such as the Enterprise Service Bus described above. The clients will then interact directly with the ESB, with the ESB taking care of forwarding the requests to the different services with different interfaces. Ironically, this means that the benefits of functional decomposition that SOA initially promises are effectively

diminished. While instead of a big monolithic application it is now composed of a set of services, it is still orchestrated and controlled by a *centralized* ESB component — with complex configurations and a potential single point of failure in the entire system, therefore hampering scalability.

Because REST principle enforces a uniform interface constraint for all services — usually in the form of HTTP protocols — there is no need for a middleware component acting as a *glue* for the different services. In the end, MSA approaches the same problem of monolithic architecture differently as compared to SOA, preferring the concept of *smart endpoints* and *dumb pipes* [18], in which the logic is not implemented in the communication layer (e.g. the *pipes*), but in the endpoints themselves — the service endpoints handle all the required business and application logic. With SOA, this concept is essentially the exact opposite, as the orchestration logic is implemented in the big ESB component, where the services' endpoints are *dumb* — in the sense that they just send/receive data to/from the ESB [6]. The result is a true functional decomposition, with every microservice being independent of each other and loosely coupled.

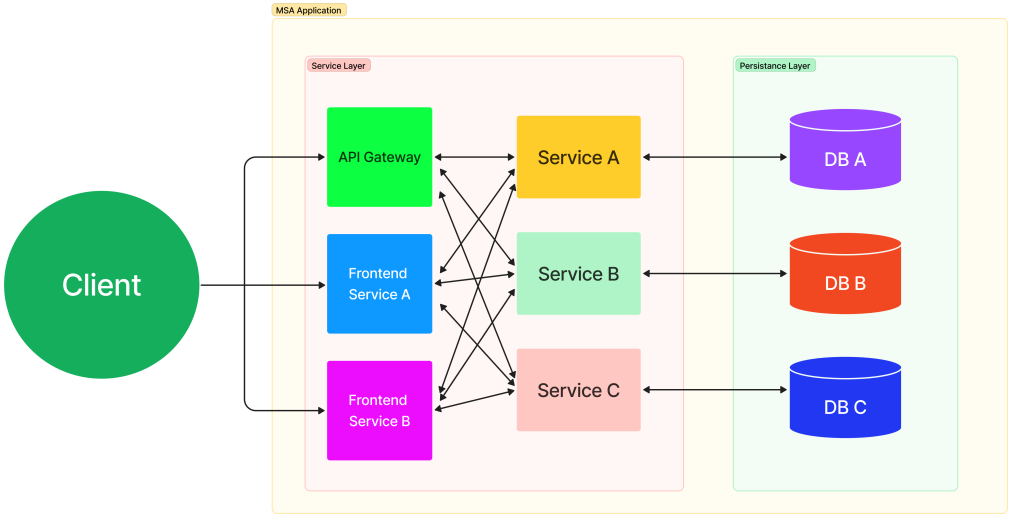

Figure 2.3: Microservice Architecture

While the benefits of MSA are significant and have been felt in the industry [1], [2], they are not without drawbacks. The biggest disadvantage of MSA is the additional complexity for the developers — each service has to be implemented such that they can handle failure of other services, difficulty (and communication costs in coordinating with other teams) in implementing requests that span multiple services, testing, deployment of the services, and also troubleshooting complexity. Because now the logic is distributed across different services in the system, the process of identifying software faults is tremendously more expensive — often involving lots of back-and-forth communication

between different responsible teams. Additionally, even though microservices in general are naturally more scalable and can handle the amount of users now common in large-scale web applications, they can be more resource-hungry because each service instance runs in its own isolated environment (e.g. its own JVM, container, or even individual VM instances) as this isolation is necessary and a central aspect of microservice architecture.

In the course of this thesis, the drawbacks and difficulties of MSA will be elaborated further in the industrial scene, and will serve to guide the direction of this Master's Thesis, in which potential solutions will be presented in the end.

2.2 Current Situation

As described above, in recent years microservice architecture has been becoming even more important at financial.com AG. The main reason is that the respective company operates by developing custom solutions tailored for specific clients/financial institutions. Because every client naturally has varying requirements in terms of the required resources/capacity of the system, we need the flexibility of being able to scale different services independently of each other. This push also coincides with the emergence of containerized applications and automated orchestration system for managing containerized applications, such as Kubernetes (K8s) [1], which places even more importance on microservices in general.

Consequently, observability has been a major problem with microservices. It is common for a company to operate large-scale software systems with millions of users. However thorough the tests are — from the unit tests, integration tests, to the client acceptance tests — there will always be *bugs* that get into production systems, as is common with these complex software systems.

It is therefore critical to ensure client satisfaction that the bugs can be found and fixed quickly. While the general task of troubleshooting/debugging is of course not at all trivial, even for experienced developers, it is a whole new level of complexity with MSA as elaborated above. This means that the developers really need to have a good *understanding* of the inner workings of their applications. This may still be possible in small or medium scale applications, with *appropriately* sized teams, but is quickly becoming impractical with larger scale software systems. No matter how good the onboarding process of a new team member is, this is still an enormous challenge to understand every bit of the code, even worse with the distributed nature of microservices — which are commonly handled not only by a single team of developers. This fact highlights the major problem with MSA, the *loss of coherence* in the entire software system. Therefore, it is very valuable to have the overall system's observability and coherence to be able to track software bugs efficiently — a desirable trait from monolithic software systems.

As more and more software projects are being migrated/developed as microservices, we also noticed increased complexity of debugging/troubleshooting, in line with the

[1] `https://kubernetes.io/`

findings from [7]. Lots of times, when a customer complaint comes, the workflow is usually as follows: The hotline/customer service department first contacts the respective frontend developers to inform of the problem. The frontend team then needs to investigate the root cause of the problem in *their* frontend application. If the problem actually lies on the frontend side, then they can start working on fixing the problem, but more often than not, it is caused by the services behind. They then have to contact the next service layer — the team responsible for maintaining the *middleware* systems. The middleware team then needs to invest resources to first troubleshoot the problem — again on their end. If it turns out that the problem actually is not on the frontend and the middleware side, the middleware team then needs to contact the backend team to inform and troubleshoot the problem together, before the actual bugfixing works can be started. This illustrates the gripes of troubleshooting a problem in the scene of microservices architecture, as lots of valuable resolution time are wasted on the ping-pong communication between lots of teams involved in the software project — hampering effectiveness and reducing customer satisfaction.

In recent years, the industry trend has shifted towards solving this concrete problem of *observability*. What observability concretely means is the ability to understand a system from a black-box (outside) perspective, by allowing the developers to probe it without even knowing its exact inner workings [19]. In order to have such observability properties, the application needs to be *instrumented* properly. What this means is that there has to be some mechanisms for an application to emit telemetry signals which can be properly monitored by the developers [20]. While there were some proposals to solve this problem, mainly from the academia, such as [21], they were often inefficient and complex: requiring high resource usage. There is also the fact that these *automatic* failure diagnostic mechanisms work mainly on *assumptions* — correlating the anomaly scores to locate the fault itself [21].

2.3 Distributed Tracing

Recently, in the context of microservices, there is a relatively new approach to solve this problem — a *distributed tracing* system. In contrast to previous approaches which rely more on assumptions to localize the fault in the code [21], a distributed tracing framework conversely focuses on the system's *observability*. While there are lots of different distributed tracing systems which are production-grade and currently in experimental use across the industry, there are general terminology and terms that trace their origin to early implementations of such concept, such as [22]–[24], which are essential in order to understand what and how distributed tracing works. These general concepts will be elaborated further below [20].

Spans A span encapsulates all the information needed in the context of observability about a *single* operation [25]. They are the building block which enables the construction of distributed *traces*. In the context of a distributed tracing system, each span generally

contains the following information:

- the *operation name*;
- start and end timestamp of the operation;
- *SpanContext*;
- set of *Attributes*;
- an ordered list of *Events*;

SpanContext In order to be able to correlate spans with each other to create the *distributed trace* tree, each span must be able to propagate its current context in terms of a logical operation — the *SpanContext* — to its *child* span, regardless if it is on the same system or on a *remote* system (e.g. microservices). The SpanContext informs the child span of its *parent* (span_ID) and the *trace* it belongs to (trace_ID).

Attributes Attributes are simple key-value pairs that can enrich the details of a span. They help when querying, grouping, or analyzing traces and spans [25].

Events Events represent a specific event that happens at a specific time in the duration of a span's lifecycle [20].

(Distributed) Traces

A distributed tracing system is centered around *traces*, a collection of linked *spans* that can be seen as a "tree" of spans that visualizes the time that each span starts and ends. What differentiates these distributed traces to regular logs is that they are consolidated *across the entire system* — which means crossing process, network, and security boundaries. They record the paths that a client request take as they propagate through a distributed architecture [26] — an ideal solution to the observability problems of MSA previously elaborated.

A span that is not a child of *any* other spans is called the *parent* — or *root* — span of a specific trace, each identified by a *unique* trace ID. This example of a real world use case at financial.com will explain this concept and the benefits it brings to the table better.

At financial.com we are developing customized financial data market applications which are tailored to the needs of the clients. Consider a very simple use-case scenario: a user logs in to the application, and requests the current stock price of a stock, for example. The following simplified diagram 2.4 illustrates the flow of actions that takes place — propagating through multiple independent services.

As can be seen below, each operation triggers another set of operations which represent the work being done during this execution of a workflow, in which the child spans are generated from a specific operation (the parent span). This illustrates the actual benefit

that a distributed tracing system brings to a system composed of microservices — we are getting the *whole* trace of a single workflow as in monolithic applications, even though the individual operations are handled by arbitrarily many, independent microservices.

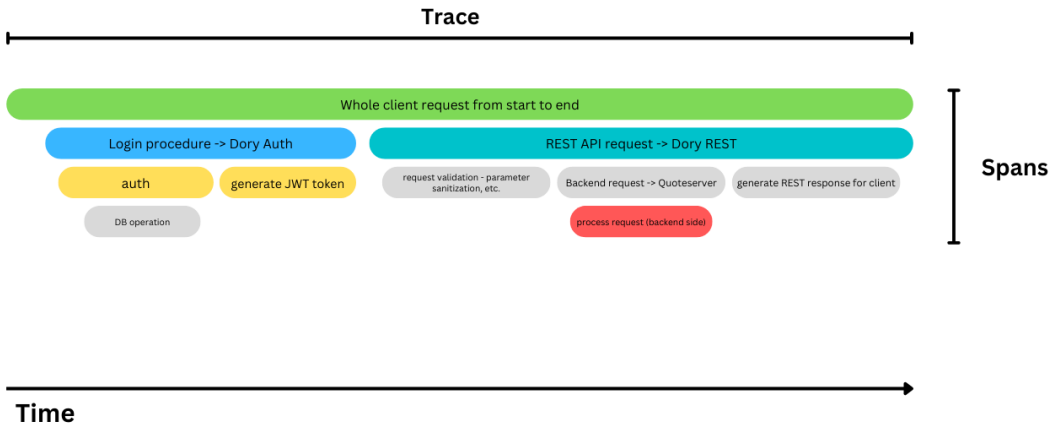

Figure 2.4: Example of a distributed trace.

3 Overview of the Distributed Tracing Approach

3.1 Infrastructure Topology

In this section, the high-level infrastructure topology of the respective platforms in the scope of this Master's Thesis will be elaborated. At financial.com AG, we can also observe the evolution of architectural trends previously explained. As financial.com operates on the financial market data supplier sector, with financial institutions and banks as its clients, we are working with a lot of different types of financial market data. Since the beginning of the infrastructure design, the problem of designing all software components as a monolith was already recognized. This leads to a non-monolithic design of the backend services, which are individual services responsible for handling different financial market data (see Figure 3.1 — not all backend services are illustrated for brevity).

As all these individual backend services handle different types of financial data and implemented with their own querying protocol that may need to be consolidated to give meaningful information to the client, a middleware layer in the middle is needed. This middleware layer — called the *Dory* platform — is developed according to microservice architecture. This platform consists of (1) `Dory REST`, which handles the communication with the different backend services, consolidation, and delivery of the data to the frontends using REST principle, (2) `Dory Auth` which handles authentication service and user persistence, and (3) `Dory Streams` which handles streaming data for the frontend clients. These whole set of services are based on MSA, hence cloud ready, and as of now are run on Kubernetes clusters. For the scope of this Master's Thesis, the focus will be on implementing a distributed tracing solution for the Dory REST platform.

As the needs of individual clients are different, our applications have to be tailored specifically, in which the Dory platform is intended to simplify development efforts at financial.com AG by leveraging the microservice architecture principle. By having the independent middleware layer as microservices, different backend services' interfaces are not baked directly into the logic of the frontend clients. This is a very important aspect at financial.com, because embedding the querying logic directly on the frontend side will incur heavy development and maintenance costs, as every little change on the backend's side of interface needs to be implemented on *every* frontend client — hampering development efficiency [17], which is in line with the reasons that pushed the industry into this current era of microservices architecture.

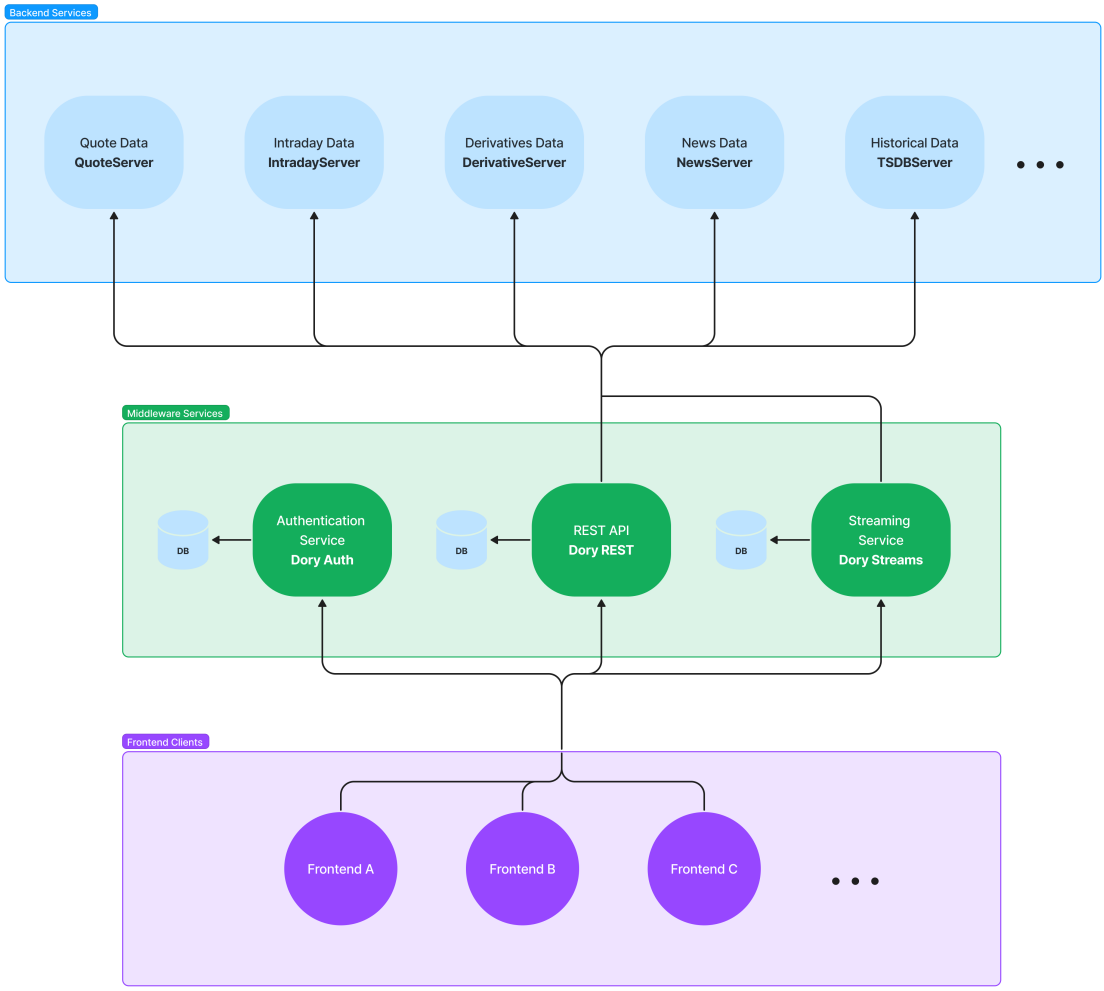

Figure 3.1: Infrastructure Topology at financial.com AG.

3.2 Design Goals

First and foremost, the single most important requirement to integrate a distributed tracing system into financial.com IT infrastructure is that this should be *transparent* to the clients. While this may be obscure at first, concretely this means that it:

- should not interfere with the functional, and non-functional requirements (e.g. performance, resource usage, security),
- should not have unintended side effects,
- should be stable to be deployed to production systems in the future.

In the first phase of this project, `Dory REST` — which powers the middleware REST API part of the financial.com AG infrastructure — and the backend services should be integrated into the planned distributed tracing system as a Proof-of-Concept. In the later chapters of this thesis, the design decisions which are taken regarding this project, and the implementation work itself will be elaborated in depth. In particular, the important design decisions will all be explained, as there might some special considerations in functional and non-functional requirements in regard to production systems.

3.3 Proposed Solution

3.3.1 Distributed Tracing Frameworks

As distributed tracing attempts to solve the problem of a (distributed) system's observability, the code themselves must be able to emit *traces*. The way that these distributed tracing frameworks work is that the traces, which are produced by the instrumented applications (either by manually instrumenting the code or by some form of automatic instrumentation) are then sent to a centralized *observability backend* to be collected and visualized for troubleshooting/debugging purposes [27] — see Figure 3.2.

There is of course the risk of fragmentation in the industry with the rising need for distributed tracing [28], as is common with any relatively young technology. Presently, there are a couple of production-grade distributed tracing frameworks [1] — both open source and commercial — to choose from. Consequently, the way that these individual frameworks instrument the respective applications would naturally vary, as each would have its own libraries and agents for transmitting telemetry data to the tools. This means that there is no standardized data format for these telemetry data, and worse, if for example a company wishes to migrate to another tracing framework, they would need to reinstrument and reconfigure their applications — which of course is impractical in real-world use cases.

This problem led to the cloud community and industry recognizing the need for standardization, and two forerunners were created: *OpenTracing* [2] — a Cloud Native Computing Foundation (CNCF) [3] project, and *OpenCensus* [4] — a Google Open Source project [5]. While OpenTracing solves the problem of data format standardization by providing a vendor-neutral API for sending telemetry data to an observability backend, it does not provide language-specific libraries that developers need to instrument their applications — which is where OpenCensus comes in. These two projects were merged together into *OpenTelemetry* [6] in pursuit of having a single standard in the industry [27].

After our internal discussions, OpenTelemetry was chosen as the distributed tracing

[1] `https://lightstep.com/distributed-tracing#distributed-tracing-tools`
[2] `https://opentracing.io/`
[3] `https://www.cncf.io/`
[4] `https://opencensus.io/`
[5] `https://opensource.google/`
[6] `https://opentelemetry.io/`

platform of choice as this is currently the foremost distributed tracing framework (as of September 2022 a CNCF *incubating* project), with standardized, vendor-agnostic instrumentation libraries that support many programming languages including Java, which is relevant in the context of this project [27]. It is important to note that OpenTelemetry as a framework does not offer an all-in-one solution which includes the observability backend, and hence we chose Jaeger [7] for this purpose as it is an open-source, CNCF graduated project, which is widely used in the industry and is being actively developed by the community. The complete architecture can be seen in the Figure 3.2 below.

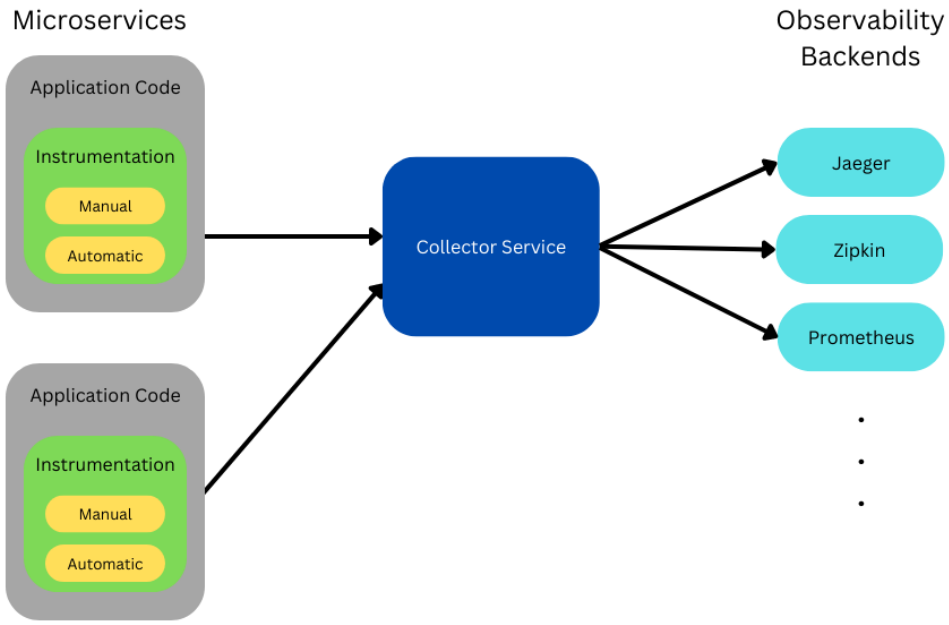

Figure 3.2: Distributed Tracing Architecture, adapted from [20].

[7] https://www.jaegertracing.io/

4 Design Decisions Regarding the Distributed Tracing Implementation

4.1 OpenTelemetry

As described above, in the end we decided that OpenTelemetry is the better suited framework for our project. This section will dive further into the OpenTelemetry framework as a whole. As illustrated in Figure 3.2, the individual microservices are instrumented in order to collect traces for troubleshooting purposes.

OpenTelemetry offers two ways of instrumenting the code (in the context of Java programming language), either by (1) *automatic* instrumentation using the OpenTelemetry Java Agent [1] — which dynamically injects bytecode to the running application to automatically generate and capture telemetry data without requiring any changes to the code itself — or by (2) manually instrumenting the code using the OpenTelemetry SDK and API package [29]. It is decided in the context of this project that *only* manual instrumentation will be used, as there are concerns of unintended side effects with regard to the design goals by simply using and attaching the Java agent to our services in a black-box manner, and also to be able to control which codes should be instrumented as we do not want too much instrumentation signal, or *noise*, being generated by the automatic instrumentation.

4.1.1 OpenTelemetry Collector

The `Collector` component is an independent service which receives, processes, and exports telemetry data to separate observability backends (e.g. Jaeger, Zipkin, etc.). It offers a vendor-agnostic implementation, and hence supports multiple open-source observability data formats from e.g. `Jaeger` and `Prometheus`, to name a few.

The `Collector` service offers two primary deployment methods: either as an (1) *Agent*, which is a lightweight `Collector` instance running side-by-side with the application (e.g. as a `sidecar`, or a `daemonset` deployment in Kubernetes); or as a (2) *Gateway*, which means a standalone `Collector` deployment, typically per cluster, data center, or on a specific region depending on the scope of the project. When deploying the `Collector` service as an Agent, it is acting as the first sink for the raw telemetry data sent from the application, which can enhance telemetry data with relevant metadata such as environment information, and can also offload some responsibilities that the

[1] `https://github.com/open-telemetry/opentelemetry-java-instrumentation/releases/latest/download/opentelemetry-javaagent.jar`

instrumentation client would need to handle, such as batching, compression, encryption, etc. The Gateway deployment mode, on the other hand, means that the `Collector` runs as a standalone service and therefore offers advanced capabilities over the Agent, including the possibility of sampling in order to not process all of the telemetry data. As each Collector instance in a Gateway deployment mode operates independently, it is easy to scale the service based on the needs [30].

The Collector service is composed of three main components which are implemented as pipelines:

- `Receivers`;
- `Processors`;
- `Exporters`.

Figure 4.1: OpenTelemetry Collector Architecture, from [20].

Receivers

The `Receiver` component is responsible for getting data into the `Collector` service. It can be either *push* or *pull* based. OpenTelemetry officially supports two distributions of the Collector service: the *reference* (1) `Core` [2] implementation, and the *extended* `Contrib` [3] distribution.

[2] `https://github.com/open-telemetry/opentelemetry-collector-releases/pkgs/container/opentelemetry-collector-releases%2Fopentelemetry-collector`

[3] `https://github.com/open-telemetry/opentelemetry-collector-releases/pkgs/container/opentelemetry-collector-releases%2Fopentelemetry-collector-contrib`

The Core distribution only supports the *OpenTelemetry Protocol* (OTLP) format [4], while the Contrib distribution additionally supports extended telemetry data formats from third-party vendors (e.g. telemetry data formats from other observability backends such as Jaeger and Zipkin) [5].

Processors

The Processor components are run on the received telemetry data from the Receivers and process them before being exported further to the observability backends.

While the Contrib distribution supports numerous processors [6], which sometimes are needed and tailored for third-party or commercial tracing systems, the Core distribution only supports two basic processors:

- Batch Processor, which places the telemetry data into batches before exporting them to compress and reduce the traffic required to transmit the data;
- Memory Limiter Processor to prevent out-of-memory situations on the Collector service itself.

Exporters

The third component, Exporters, are naturally responsible for exporting the received telemetry data, and further process data to the observability backends for visualization, as the OpenTelemetry project itself does *not* include an observability backend for this purpose. The Exporters can either be push or pull based. Analogous to the Receiver components, the Core distribution of OpenTelemetry Collector only supports OTLP format, while the Contrib distribution supports multiple data formats from third-party vendors such as Jaeger or Zipkin, and even commercial enterprise tracing systems such as from Amazon WebServices, Google Cloud, or Microsoft Azure.

OpenTelemetry supports exporting the telemetry data through either (1) HTTP, or (2) gRPC transports.

4.1.2 Context Propagation

At the heart of a distributed tracing system is the ability to propagate and correlate events (i.e. Spans) across different service boundaries — a concept known as *Context propagation*. In order to be able to correlate different Spans with each other in a distributed system, the individual components have to be able to receive, process, and further propagate metadata which from now on will be referred to as the Context [20].

[4] https://opentelemetry.io/docs/reference/specification/protocol/design-goals/
[5] https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/receiver
[6] https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/processor

Context

`Context` can be understood as the encapsulation of several pieces of information which are relevant to the current specific workflow (e.g. a client request to an HTTP endpoint), which can be passed between functions in a single process, or inter-process over the propagation mechanisms [20].

There are two types of Contexts:

- **Span Context**
  `Span Context` contains the minimally required data to move and correlate tracing information across service boundaries and consists of the (1) `traceID`, the trace identifier of the whole `Context` (e.g. stays the same between the propagated services within the same `Context`), (2) `spanID`, the span identifier itself (every `Span` has its own unique `spanID`), (3) the `traceFlags` which carries the trace options for the `Span Context`, and the (4) `traceState`, which carries vendor-specific trace data that allows multiple tracing systems from different vendors to participate in the same trace [25], [31].

- **Correlation Context**
  `Correlation Context` (or *Baggage*) in turn contains user-defined properties, and is a key-value store that can be set freely [27]. The use of `Baggage` is entirely optional and is not required for correlating the `Spans` with each other.

Propagation

In order to have a distributed trace, the `Context` has to be *propagated* across service boundaries. The *current* `Context` is first injected into an outgoing request to another service (usually by injecting the relevant values into an HTTP header), and will be extracted by the receiving service in order to correlate the `Spans` with each other.

OpenTelemetry officially supports the following list of `Propagators`:

- W3C TraceContext Propagation [7]
- W3C Baggage (Correlation Context) Propagation [8]
- B3 (Zipkin) Propagation [9]
- Jaeger Propagation Specification [10]

The following Figure 4.2 illustrates how context propagation works in OpenTelemetry.

[7] `w3.org/TR/trace-context/`
[8] `https://w3c.github.io/baggage/`
[9] `github.com/openzipkin/b3-propagation`
[10] `jaegertracing.io/docs/1.37/client-libraries/#propagation-format`

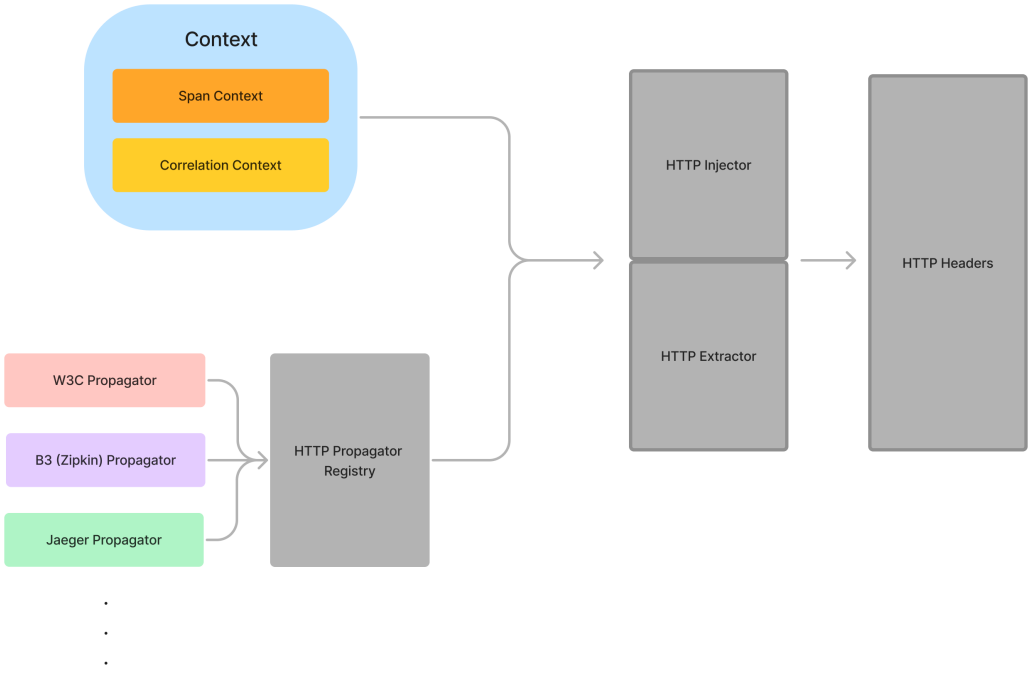

Figure 4.2: Context Propagation in a distributed tracing system, adapted from [20].

4.1.3 Java Manual Instrumentation Using OpenTelemetry SDK

In this section, the manual instrumentation guidelines for a Java application using OpenTelemetry will be explained in a high-level manner, adapted from [29]. The concrete implementation on the `Dory REST` service, which is the main scope of this Master's Thesis, will be elaborated further on Chapter 5 5.

Setting Up the OpenTelemetry Instance

The first step to manually instrument the code is to configure an instance of the `OpenTelemetrySdk` as early as possible in the application, to be injected and reused in the different parts of the codebase in order to generate telemetry data, as illustrated in the example 4.1 below.

```
1 Resource resource = Resource.getDefault()
2   .merge(Resource.create(Attributes.of(ResourceAttributes.SERVICE_NAME,
3         "logical-service-name"))); // Set the corresponding service name
              for the exported spans here.
```

```
4
5  SdkTracerProvider sdkTracerProvider = SdkTracerProvider.builder()
6    .addSpanProcessor(BatchSpanProcessor
7      .builder(OtlpGrpcSpanExporter.builder().build()).build()) // Using gRPC
           transport.
8    // Also possible: OtlpHttpSpanExporter -> using HTTP transport.
9    .setResource(resource)
10   .build();
11
12 // Create the OpenTelemetry instance.
13 // NOTE: we should only have one single instance which will be injected in
       other places where we want to instrument the code!
14 OpenTelemetry openTelemetry = OpenTelemetrySdk.builder()
15   .setTracerProvider(sdkTracerProvider)
16   .setPropagators( // Using W3C Propagator.
17     ContextPropagators.create(W3CTraceContextPropagator.getInstance()))
18   .build();
```

Listing 4.1: Example on setting up the `OpenTelemetrySdk` Instance.

Acquiring the Tracer Object for Generating Spans

After we have the `OpenTelemetry` Bean instance which can be injected to other classes where we want to instrument the code, we need to acquire the `Tracer` instance in order to generate the `Spans` itself. It is important to note, here in comparison to the `OpenTelemetry` instance — which should always be a *singleton* — we are free to create different instances of the `Tracer` object on different classes in order to differentiate where the `Spans` are coming from inside the application boundary.

```
1  import io.opentelemetry.api;
2
3  //...
4
5  // Here the Tracer name and version are purely informational!
6  // The Tracer name will be included in the exported span info for
       debugging purposes.
7  Tracer tracerWithVersionInfo =
8      openTelemetry.getTracer("name-of-class-being-instrumented", "1.0.0");
9  // OR:
10 Tracer tracerNameOnly =
11     openTelemetry.getTracer("name-of-class-being-instrumented");
```

Listing 4.2: Acquiring a `Tracer` Instance.

Generating Spans in the Source Code

With the `Tracer` instance acquired, we are now ready to instrument method calls and generate telemetry data. The recommended pattern is to surround the parts of the code to be instrumented using `try-with-resources` statement, so that the `Span` object can be properly closed after use, avoiding resource leaks.

```
1  // Create a span with the name "my span"
2  Span span = tracer.spanBuilder("my␣span").startSpan();
3
4  // Make the span the current span, with the start time automatically
       recorded
5  try (Scope ss = span.makeCurrent()) {
6      // The code block to be instrumented goes here!
7  } finally {
8      // NOTE: Spans have to be properly closed in order to prevent resource
           leaks!
9      // Close the span and record the end time of the span.
10     span.end();
11 }
```

Listing 4.3: Creating `Spans`.

Recording Thrown Exceptions in a Span

There is also the possibility to record any exceptions thrown within an instrumented code block, such as demonstrated below.

```
1  Span span = tracer.spanBuilder("my␣span").startSpan();
2  try (Scope scope = span.makeCurrent()) {
3      // do something
4  } catch (Throwable throwable) {
5      span.setStatus(StatusCode.ERROR, "Something␣bad␣happened!");
6      span.recordException(throwable)
7  } finally {
8      span.end(); // Cannot set a span after this call
9  }
```

Listing 4.4: Recording `Exceptions` in `Spans`.

Creating Nested Spans

Sometimes it is useful to instrument and correlate nested operations for debugging purposes. OpenTelemetry supports correlating `Spans` within process boundary, and also across remote processes (through Context Propagation — see 4.1.3).

The example below illustrates how one can create and correlate nested `Spans` within the same application [29]:

```
1  void parentTwo() {
2    Span parentSpan = tracer.spanBuilder("parent").startSpan();
3    try(Scope scope = parentSpan.makeCurrent()) {
4      childTwo();
5    } finally {
6      parentSpan.end();
7    }
8  }
9
10 void childTwo() {
11   Span childSpan = tracer.spanBuilder("child")
12     // NOTE: we do not need to setParent(...) explicitly here
13     // parentSpan is automatically added as the parent as childTwo() is
           called within the parent scope!
14     .startSpan();
15   try(Scope scope = childSpan.makeCurrent()) {
16     // do stuff
17   } finally {
18     childSpan.end();
19   }
20 }
```

Listing 4.5: Creating nested `Spans`.

Propagating Spans Across Service Boundaries

As explained in 4.1.2, OpenTelemetry is able to correlate `Spans` across different services, as long as the respective instrumented services emit valid `Context` that can be injected into a *carrier* — e.g. HTTP headers — and extracted at the downstream service. This section will explain how the OpenTelemetry SDK is able to correlate spans with each other to form a distributed *trace*.

The key concepts to be understood in regard to how OpenTelemetry handles context propagation are the `setter/injector` and `getter/extractor` functions. These functions have to be defined in the respective places where we need to correlate the new `Spans` with the propagated `Spans` from the upstream services.

For the `getter` function, we need to create a new `TextMapGetter<T>` object, parametrized with the carrier object itself — the object that *carries* the header value, for example a `javax.servlet.http.HttpServletRequest` — and implement the `Iterable<String> keys(C carrier)` as well as `String get(C carrier, String key)` methods, as illustrated below.

```
1  // Create TextMapGetter on HttpServletRequest Object.
2  // Parameterized with the HttpServletRequest Object.
3  private static final TextMapGetter<HttpServletRequest>
       servletRequestGetter =
4          new TextMapGetter<HttpServletRequest>() {
5              // We have to implement the following methods on the
                   TextMapGetter interface
6
7              // This implementation should return the header names (keys) of
                    the carrier object
8              @Override
9              public Iterable<String> keys(HttpServletRequest carrier) {
10                 // Return the header names using HttpServletRequest.
                       getHeaderNames() method
11                 // NOTE: This implementation will depend on the
                       corresponding carrier object!
12                 return Collections.list(carrier.getHeaderNames());
13             }
14
15             // This implementation should return the value of a given
                   header name.
16             @Nullable
17             @Override
18             public String get(@Nullable HttpServletRequest carrier, String
                   key) {
19                 // Return the header value if exists, else return empty
                       string.
20                 // NOTE: This implementation will depend on the
                       corresponding carrier object!
21                 return carrier.getHeader(key) != null ? carrier.getHeader(
                       key) : "";
22             }
23         };
```

Listing 4.6: Example of a Getter Implementation on `HttpServletRequest` Carrier Object.

Analogous to the `getter` functions, to define a custom `setter` function we need to again create a `TextMapSetter<T>` object, parameterized with the corresponding *carrier* object, and implement the `void set(C carrier, String key, String value)` method.

```
1  // Create a new TextMapSetter on ClientHttpRequest carrier.
2  private static final TextMapSetter<ClientHttpRequest> httpHeaderSetter =
3          new TextMapSetter<ClientHttpRequest>() {
4
```

```
5            // Implement the set method, call the set method on the carrier
                 itself (ClientHttpRequest).
6            @Override
7            public void set(@Nullable ClientHttpRequest carrier, String key,
                 String value) {
8                carrier.getHeaders().set(key, value);
9            }
10       };
```

Listing 4.7: Example of a Setter Implementation on `ClientHttpRequest` Carrier Object.

After having both the `getter` and `setter` method implementations, we are now able to extract `Context` from the incoming requests, correlate it with our `Spans`, and further propagating it to the next service in the call stack.

```
1  // Extract the request Context from client
2  // The relevant API is TextMapPropagator.extract(Context context, C
       carrier, TextMapGetter<C> getter)
3  // Here we pass Context.current() as the first argument, to merge the
       extracted Context data with our current context
4  // The method returns a new Context object which contains the merged
       context data.
5
6  // The following is taken from the OpenTelemetry SDK Javadoc:
7  /* Extracts data from upstream. For example, from incoming http headers.
       The returned Context
8   * should contain the extracted data, if any, merged with the data from
        the passed-in Context.
9   *
10  * If the incoming information could not be parsed, implementations MUST
        return the original
11  * Context, unaltered.
12  *
13  * @param context the {@code Context} used to store the extracted value.
14  * @param carrier holds propagation fields. For example, an outgoing
        message or http request.
15  * @param getter invoked for each propagation key to get data from the
        carrier.
16  * @param <C> the type of carrier of the propagation fields, such as an
        http request.
17  * @return the {@code Context} containing the extracted data.
18  */
19 Context context = openTelemetry.getPropagators().getTextMapPropagator().
       extract(Context.current(), httpServletRequest, servletRequestGetter);
20 // Generate a new span, and sets the extracted context as the (!)Parent
```

```
21      Span span = openTelemetry.getTracer("class-name")
22              // set the incoming Request URI as the Span name
23              .spanBuilder(httpServletRequest.getRequestURI())
24              // Set the extracted context as parent
25              .setParent(context).setSpanKind(SpanKind.SERVER)
26              .startSpan();
27  try (Scope scope = span.makeCurrent()) {
28      // do stuff...
29      // Example on further injecting the current context to an outgoing
            request
30      // Here the example carrier is an org.springframework.http.client.
            ClientHttpRequest
31      ClientHttpRequest request = httpRequestFactory.createRequest(uri,
            httpMethod);
32
33      // Now inject the current context using our defined setter method
34      openTelemetry.getPropagators().getTextMapPropagator().inject(Context.
            current(), request, httpHeaderSetter);
35  } finally {
36      span.end() // end span as normal
37  }
```

Listing 4.8: Extracting and further propagating the `Context` across service boundaries.

4.2 Instrumentation of the Dory REST Service

As explained in the previous section, the `Dory REST` service will be *manually* instrumented instead of using automatic instrumentation agent that the OpenTelemetry framework offers. The main consideration for not using automatic instrumentation is the fact that the Java agent which can be attached to a running Java application will dynamically inject bytecode at runtime in order to generate telemetry data [29]. Consequently, the instrumentation is then done in a black-box manner, with the agent's behaviour outside of our control, raising concerns about unintended side-effects or system stability/resource usage issues. Concretely, this means that code changes are needed in the `Dory REST` service in order to instrument the application manually via the OpenTelemetry SDK, which forms the bulk of the implementation works in this thesis.

`Dory REST` is a Spring Boot [11] service, handling the consolidation and processing of data from the different backend services (Figure 3.1), and the delivery to the frontend clients using REST principle. As `Dory REST` acts as the middleware between frontend and backend, with regards to instrumentation we are interested in (1) extracting the `Context` from incoming frontend requests, (2) generating *Spans* which correlate to

[11] `https://spring.io/projects/spring-boot`

the root parent `Context` from the incoming requests, and (3) further injecting and propagating the `Context` information through the outgoing requests to different types of backend services, which will be further explored in the next sections.

It is also important to note that there are a couple of different services under our team's responsibility which are classified as *backend* services, as they serve as one of the many sources of data for `Dory REST`. As these services also use Spring Boot, some of the following design approaches discussed below will apply to them as well. The concrete implementations for these services will be explained separately in Chapter 5.

4.2.1 Instrumenting the Incoming Requests

The `Dory REST` service is responsible for communicating with different backend services, and providing the data to the frontend clients via REST endpoints. In order to understand the design decisions taken in the course of this project, it is helpful to illustrate the generic three-layer architecture of a Java REST API Service.

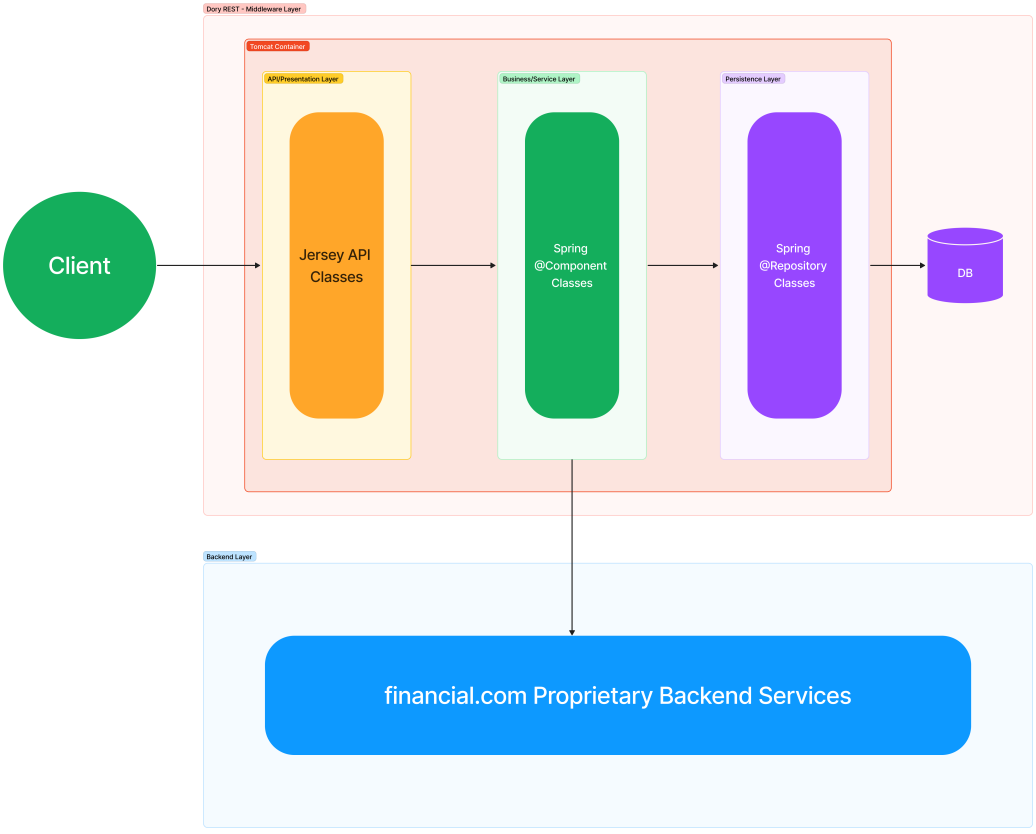

Figure 4.3: High Level Dory REST Architecture.

Generally, a REST API service can be divided into three layers: (1) the REST API

Gateway (*presentation*) layer, which is responsible for receiving incoming client requests and redirecting the requests to the responsible resources on the next layer — (2) the business/service layer, which is composed of the *resources*, or Components (in the sense of Spring `@Component`) which holds the business logic of the requests, and the (3) persistence layer, which is responsible for the data access to persistence storage mediums, such as a database.

In our case, the REST API layer is handled by the API Java classes annotated by the Jersey [12] annotation `@Path` [13] to identify the URI path that these classes will serve requests for, as illustrated in the example below.

```
1  @Path("")
2  @Api(description = "the␣quote␣API")
3  public class QuoteApi extends AbstractResource {
4      @GET
5      @Path("" + "/quote" + "/info")
6      public Response quoteInfoGet(...) {
7          // Calling the respective Component which implements the business
               logic here.
8      }
9      ...
10 }
```

Listing 4.9: Jersey API Class Example.

As the REST endpoints will naturally have different data sources and can be tailored to specific use-cases/clients, they can have custom implementations, and consequently different Spring component classes (annotated with `@Component` [14]) which holds the business logic. The high level architecture for the component classes in the business/service layer can be seen in Figure 4.4 below.

The `handleRequest()` method which returns a `javax.ws.rs.core.Response` object is where the processing of the request happens, and where the business logic is implemented. This method provides a good starting point to instrument incoming requests from the client, as it encapsulates the whole request call. As previously explained in Chapter 4.1.3, we need to surround the code to be instrumented in a `try-with-resources` block. Consequently, this means that the naive solution by instrumenting the `handleRequest()` methods will produce a lot of repeated `try-finally` patterns as there can be arbitrarily many `SpecificComponents` that implement/override the `handleRequest()` method (Figure 4.4) — making this approach brittle and error-prone.

The better approach is to leverage the Java EE (now: Jakarta EE) Filters [15] which

[12] `https://eclipse-ee4j.github.io/jersey/`
[13] `https://javadoc.io/doc/com.sun.jersey/jersey-bundle/latest/javax/ws/rs/Path.html`
[14] `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html`
[15] `https://jakarta.ee/specifications/platform/9/apidocs/jakarta/servlet/filter`

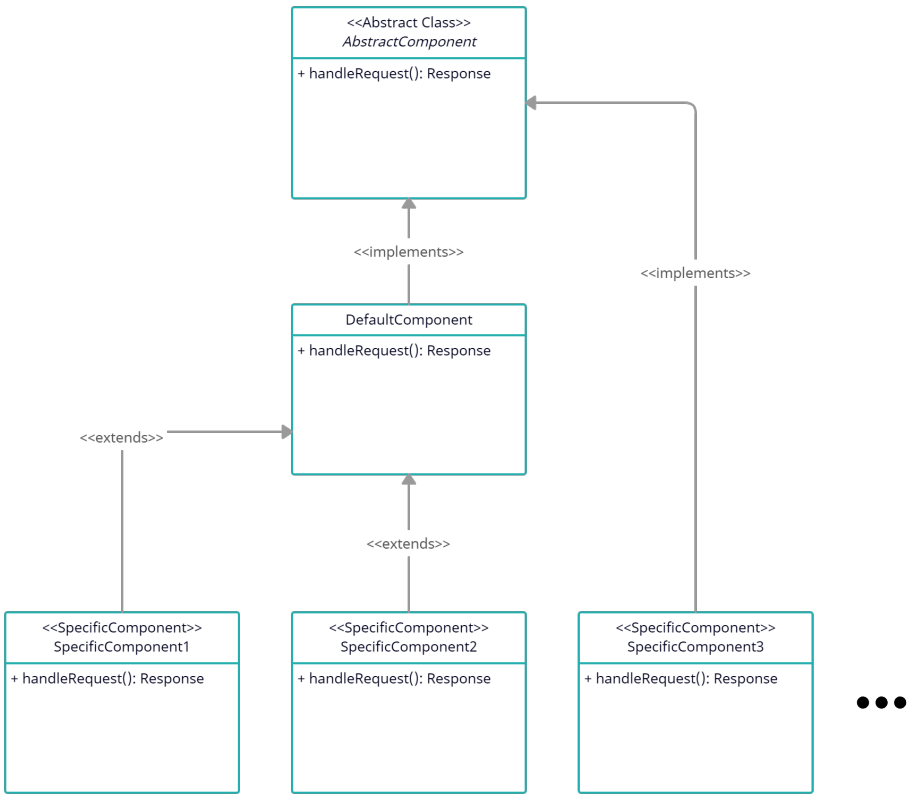

Figure 4.4: Component Diagram — Service Layer.

can act as a pre-processing step on the requests before being processed by the Jersey servlet. The example 4.10 illustrates how a custom `Filter` can be implemented, with the possibility of manual code instrumentation.

```
1  public class MyCustomFilter implements Filter {
2
3      // Need to implement the doFilter method.
4      @Override
5      public void doFilter(ServletRequest httpServletRequest, ServletResponse
           httpServletResponse, FilterChain filterChain) {
6          // do stuffs, here we can access the HTTP Servlet Request and
               Response (after the processing) objects.
7          // We can access e.g. the header values from the request object.
8
9          // In the following an example on how a manual instrumentation
               might look like is shown.
10         // Generate a new span
11         Span span = traceContext.getTracer(MyCustomFilter.class.
```

```
            getCanonicalName())
12              .spanBuilder(httpServletRequest.getRequestURI())
13              .setParent(context).setSpanKind(SpanKind.SERVER)
14              .startSpan();
15      try (Scope scope = span.makeCurrent()) {
16          // Continue the processing through the filter chain.
17          // This method call encapsulates the actual request processing
                through the business logic until returning of the response.
18          filterChain.doFilter(httpServletRequest, httpServletResponse);
19      } catch (Throwable throwable) {
20          span.setStatus(StatusCode.ERROR, throwable.getMessage());
21          span.recordException(throwable);
22          throw throwable;
23      } finally {
24          // Set attributes and end span
25          HttpServletResponse servletResponse = (HttpServletResponse)
                httpServletResponse;
26          span.setAttribute("http.status", servletResponse.getStatus());
27          span.setStatus(StatusCode.OK);
28          span.end();
29      }
30  }
31 }
```

Listing 4.10: Example of a custom `Filter` implementation.

As illustrated in example 4.10, implementing the incoming request instrumentation on a custom `Filter` provides three immediate benefits:

- it provides an ideal place in the code to extract the `Context` from the incoming requests (embedded in the *traceparent* header [16]),
- it also provides a central place where the instrumentation for all incoming requests can happen, as we only need to surround the `doFilter` method in a `try-with-resources` (see 4.3) statement to instrument the whole processing of the request, thereby solving the previous problem of duplicated patterns on the `Component` classes for the endpoints,
- being able to record the status code of the *Response* in the Spans, as we are able to access the `HttpServletResponse` object after the processing of the request.

Additionally, as every exceptions thrown within the business logic are encapsulated in the `doFilter` method call, a custom `Filter` implementation also brings the benefit of being able to record the thrown exceptions in this central place (see 4.4).

[16] `https://www.w3.org/TR/trace-context/`

4.2.2 Instrumenting the Outgoing Requests

With the design problems regarding instrumenting the incoming requests solved, we now need to turn to the other end — how we can elegantly instrument the outgoing requests from `Dory REST`. As `Dory REST` is a middleware REST API platform, it naturally creates outgoing requests to downstream backend services in order to consolidate/process the raw data to be returned to the frontend clients.

These outgoing requests can be broadly classified into three types according to their implementations:

1. Outgoing requests to the financial.com backend services through the `fcom_base` proprietary framework, which is based on a custom protocol implemented on top of the low level socket communications;
2. *Generic* outgoing HTTP requests;
3. Outgoing HTTP requests to ElasticSearch [17] services, which uses its own REST client implementation.

While there are other types of outgoing request calls, such as database queries to the persistence layer, we decided that we will initially implement tracing solution only on these types of requests explained above. The reason is that we want to have a meaningful distributed tracing system, without having too much irrelevant tracing information, even though we might implement more instrumentation on our infrastructure in the future.

In this case, having a central place to instrument all outgoing requests will be difficult, as these distinct types of requests have very different types of client implementations. The following sections will explain more about the proposed solutions to instrument these outgoing requests.

financial.com Proprietary Backend Services

Making up the most of the outgoing requests from Dory REST are the requests to the financial.com backend services. `Dory REST` and the backend services communicate through a proprietary protocol which is based on low level TCP sockets. At `Dory REST`, we are using an internal financial.com Java framework — the `fcom_base` — to communicate with the backend services.

To understand the design decisions which will be explained in a moment, it is helpful to understand how we are using the `fcom_base` framework in Dory REST. To put it simply, this is an internal framework, developed as a way for the frontend clients to access the backend APIs. At `Dory REST`, we are initializing custom `FClients` to communicate with the different backend services. The carrier object, the `FRequest`, which represents a request to a backend service already supports embedding header information, analogous to HTTP headers.

[17] `https://www.elastic.co/`

The implementation work on Dory REST side to instrument these first type of requests is arguably the most straightforward, as all the backend requests are already managed in a single place on the Dory REST codebase — hence already providing a suitable, central place where we can inject the `Context` in the `FRequest` *and* also for instrumenting the request execution — eliminating the need to do refactoring works.

Generic HTTP Requests

Not all backend services and the underlying data sources depend on the proprietary financial.com backend services, as previously explained above. These *generic* HTTP requests are making up the second part of the outgoing requests from `Dory REST`. Here the situation is a bit more complicated, as these requests are scattered through the code, lacking a centralized place where all outgoing requests can be easily instrumented. Furthermore, the usage of HTTP clients in `Dory REST` is still not uniform, as there are two types of HTTP client currently being used: (1) the lower level Apache HTTP client [18] (through the Spring `HttpComponentsClientHttpRequestFactory` [19]), and (2) the Spring `RestTemplate` client [20], which provides higher level abstractions.

After consultation with the other developers responsible for maintaining the `Dory REST` platform, it was decided to solve this problem in two steps. The (1) first task will first focus to unify the HTTP client usage across `Dory REST`, whereas the (2) second task is then to refactor the code so that all of the outgoing HTTP requests are done through a central component, thereby solving the problem of repeated instrumentation code blocks scattered all over the codebase.

Regarding the first task, we have the choice to either migrate to the Apache HTTP client, to the Spring `RestTemplate` as previously explained above, or even to another suitable HTTP client altogether. There is also a newer web client implementation from the Spring framework which is meant to replace the `RestTemplate` client — the Spring `WebClient` [21]. After discussing the alternatives, in the end we decided to migrate all HTTP requests to use the Apache HTTP client, with the implementation works explored more in-depth in Chapter 5 — our consideration is as follows:

- As since Spring version 5.0 upwards the `RestTemplate` client will be in maintenance mode — meaning there will be only bug fixes going forward and will possibly be deprecated in future versions — we decided to migrate from `RestTemplate`. This leaves us with two options: the Apache HTTP client or the new `WebClient` implementation from Spring;

[18] `https://hc.apache.org/httpcomponents-client-5.1.x/`

[19] `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/client/HttpComponentsClientHttpRequestFactory.html`

[20] `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html`

[21] `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/reactive/function/client/WebClient.html`

- Migrating to the newer `WebClient` implementation does not really bring much benefit as this will involve lots of code refactorings to adapt to the new API;
- The Apache HTTP client operates on the lowest abstraction level in comparison to the `RestTemplate` and the `WebClient`, thereby offering most flexibility and functionality. While the lower-level nature of the Apache HTTP client means that we are losing some convenience of `RestTemplate`, the benefits of having uniform HTTP client usage across Dory REST is far more valuable;
- There are fewer uses of `RestTemplate` in our codebase, therefore sparing more refactoring works;

As we now have unified usage of Apache HTTP client across `Dory REST`, the next design consideration is on how we can have a central place where we can inject the `Context` to an outgoing request, and also in the process instrument the whole request execution. Currently, the usage of the HTTP client is configured through the standard Spring `@Configuration` class, where the `@Beans` are initialized, configured, and returned — as illustrated in the code snippet 4.11 below.

```
1  @Bean
2  @Primary
3  // Configure the primary HTTP client bean, not proxied.
4  public HttpComponentsClientHttpRequestFactory clientHttpRequestFactory() {
5      HttpComponentsClientHttpRequestFactory clientHttpRequestFactory = new
           HttpComponentsClientHttpRequestFactory();
6      clientHttpRequestFactory.setHttpClient(HttpClients.createDefault());
7
8      return clientHttpRequestFactory;
9  }
10
11 @Bean
12 @Qualifier(ApplicationConfiguration.HTTP_FACTORY_WITH_PROXY_BEAN_ID)
13 // Configure another HTTP client bean with proxy configuration for
       outgoing requests to the public internet.
14 public HttpComponentsClientHttpRequestFactory
       clientHttpRequestFactoryWithProxy(RestConfig restConfig) {
15     HttpComponentsClientHttpRequestFactory clientHttpRequestFactory = new
           HttpComponentsClientHttpRequestFactory();
16
17     clientHttpRequestFactory.setHttpClient(HttpClientBuilder.create().
           setProxy(new HttpHost(restConfig.getProxy(), restConfig.getPort(),
           "http")).build());
18
19     return clientHttpRequestFactory;
```

```
20 }
```

Listing 4.11: Apache HTTP client configuration.

These configured client `Beans` can then be injected in other classes in order to make outgoing HTTP requests from `Dory REST`. Going back to the original problem, the consideration is to have a central place where all generic HTTP requests are made, instead of injecting the clients everywhere and executing the requests directly — which of course leads to scattered request executions across the codebase.

The most pragmatic solution in this case is to implement a *wrapper* class/component, where we adapt the interface to our use cases, and in turn to inject this new wrapper class everywhere we need to run an outgoing HTTP request. The benefit is directly clear: the actual request execution will then happen in this wrapper class, giving us the opportunity to instrument the request execution without repeated `try-with-resources` blocks scattered across our code. The detailed implementation of the design approaches discussed here will be explained in Chapter 5.

ElasticSearch HTTP Requests

Lastly, we turn to the third type of outgoing requests which is relevant in the context of instrumentation scope — HTTP requests to the *ElasticSearch* services. ElasticSearch is being used to power several backend services in order to provide enterprise search capabilities and certain types of data feed, for example `NewsServer` and `SymbolSearch` for searching stock tickers.

While ElasticSearch provides REST interface to interact with the service as standard, these type of requests are not covered in the previous section 4.2.2 as the outgoing requests on `Dory REST` are executed through the `org.elasticsearch.client.RestHighLevelClient` instead of using the Apache HTTP client because it provides a more convenience high-level functionalities to interact with the Elastic Search API [22].

The `RestHighLevelClients` are similarly configured with a `@Configuration` class where the multiple beans (for different ElasticSearch hosts) can be configured.

```
1  // Default RestHighLevelClient for ElasticSearch requests
2  @Scope("singleton")
3  @Bean(destroyMethod = "close")
4  @Primary
5  public RestHighLevelClient client(ElasticsearchConfig elasticsearchConfig)
       {
6
7      RestHighLevelClient client = new RestHighLevelClient(
8              RestClient.builder(new HttpHost(elasticsearchConfig.getHost(),
                   elasticsearchConfig.getPort(), ELASTICSEARCH_PROTOCOL))
9      );
```

[22] `https://www.elastic.co/guide/en/elasticsearch/reference/current/search-search.html`

```
10      return client;
11  }
12
13  // Example for creating another client with different configurations
14  @Bean(name = "AnotherESClientConfiguration", destroyMethod = "close")
15  public RestHighLevelClient anotherClient(ElasticsearchConfig anotherConfig
        ) {
16
17      return new RestHighLevelClient(
18              RestClient.builder(new HttpHost(
19                      anotherConfig.getHost(),
20                      anotherConfig.getPort(), ELASTICSEARCH_PROTOCOL)));
21
22  }
```

Listing 4.12: ElasticSearch `RestHighLevelClient` configurations.

The problem is then analogous to the the previous problem regarding instrumentation of generic HTTP requests. Similarly, we decided to modularize the request executions into a central component.

In contrast to the generic HTTP requests, here we will *not* be injecting the current `Context` into the ElasticSearch requests, but instead only encapsulating the request in an instrumentation block and recording the exceptions that may be thrown in the course of the execution. The reason is that the ElasticSearch service will not be able to be manually instrumented anyway, and hence the important issue to focus is to instrument the request from *client* side — in this case `Dory REST`.

4.3 Infrastructure Design

With the most pressing design and implementation issues already discussed in Chapter 4.2, this section will now focus on the infrastructure design considerations. First consideration to be decided upon is about the distributed tracing architecture — how the individual components that make up the whole tracing system will be organized and deployed. As previously shown in Figure 3.2, it is common to have one `Collector` service deployed in an environment, for example, which will be ingesting and processing the telemetry data from the instrumented applications, and further sending them to an (or multiple) observability backend(s), e.g. `Jaeger`.

At financial.com AG, we are already migrating major parts of our DevOps infrastructure to Kubernetes clusters since past couple of years — in line with the industrial trend that was supported by the advent of microservice architecture (MSA). At first, our consideration was on how the OpenTelemetry `Collector` service should be configured and deployed into the `dev` environment of our Kubernetes cluster. It was decided that we should have a `Collector` instance on each environment with the *Gateway* development mode (see Chapter 4.1.1).

In the context of Kubernetes, there is a possibility to deploy the `Collector` instance using a Kubernetes Operator implementation from OpenTelemetry [23]. A Kubernetes Operator is an extension which allows custom logics to be implemented in order to manage applications and their components — in a sense replacing a *human operator* which controls and manages the K8s deployments. This option turned out to be out of the question, as the IT Operations division at our company didn't like having a Kubernetes Operator in the internal cluster managing the deployment of the instrumentation services themselves, deeming it a security concern.

Since `Jaeger v1.35` there is also another option to bypass the OpenTelemetry Collector completely, as the newer versions support receiving OpenTelemetry trace information via the OpenTelemetry Protocol directly [24] (see 4.1.1). In the end, we decided to go with this option, as this means that we do not need to additionally deploy a separate service (the OpenTelemetry `Collector`) — hence reducing maintenance work and resource usage. For the storage solution, our `Jaeger` deployment will employ an ElasticSearch storage backend for storing the `Span` data [25], as ElasticSearch is recommended by the Jaeger team for large scale production deployment, and we plan on bringing this distributed tracing solution to our `production` systems in the future.

With the instrumentation infrastructure in place, the early Proof-of-Concept versions of the Spring Boot services under our team's responsibility will be deployed on the `dev` environment on our Munich Kubernetes cluster. These early versions will then be continuously developed and evaluated in the course of this Master's Thesis — with more detailed explanation on Chapter 5.

[23] `https://opentelemetry.io/docs/k8s-operator/`

[24] `https://medium.com/jaegertracing/introducing-native-support-for-opentelemetry-in-jaeger-eb661be8183c`

[25] `https://www.jaegertracing.io/docs/1.18/deployment/#storage-backends`

5 Implementation Details

While Chapter 4 previously focuses on explaining the abstract design considerations which need to be addressed before actual implementation tasks can be started, in this chapter the concrete implementations will be explained in depth. This chapter will focus on implementation works on `Dory REST` and several other backend services which are operated under the responsibility of our division at financial.com AG. Nevertheless, the main emphasis will be placed upon `Dory REST` as the implementation works on these backend services will largely follow the same approaches previously discussed in Chapter 4.

5.1 Implementing the Distributed Tracing Solution on Dory REST

This section will now elaborate the implementation details of integrating the OpenTelemetry distributed tracing solution into the `Dory REST` codebase. The chapter will be structured in a similar way as the design approaches discussed in Chapter 4.2.

5.1.1 Configuring and Creating the OpenTelemetry Instance

First we need to instantiate the `OpenTelemetry` bean in our Spring Boot application as early as possible in the runtime, as explained in Chapter 4.1.3. To do this, we will need to create a Spring `@Configuration` class which will create the OpenTelemetry `@Bean` which is managed by Spring. This way, we can inject the instance at other places in the codebase where instrumentation is required. Additionally, we implemented the `OpenTelemetryConfiguration.java` class which will provide the opportunity of configuring whether the instrumentation should be enabled or disabled, the `Collector` service's endpoint, and the environment name to be embedded as additional information on the `Spans`.

The code listing 5.1 illustrates the concrete implementation with the explanations.

```
1 // Annotate as a Spring configuration class so that the created instances
      are managed by Spring
2 @Configuration
3 // Make the OpenTelemetry configurable via a configuration .properties
      file
4 @PropertySource(value = "file:/etc/financial.com/dory-rest/config/otel.
      properties", ignoreResourceNotFound = true)
```

```
5  @ConfigurationProperties(prefix = "otel", ignoreUnknownFields = true)
6  public class OpenTelemetryConfig {
7
8      private static final Logger logger = LoggerFactory.getLogger(
           OpenTelemetryConfig.class);
9
10     private static final String SERVICE_NAME = "dory-rest"; // Set service
           name to be included in Spans
11
12     // Controls whether the instrumentation should be enabled or disabled.
           By default it is disabled.
13     private boolean enabled = false;
14
15
16     private String endpoint = "http://localhost:4317"; // Default OTLP
           endpoint -> configurable through otel.properties
17
18     private String environmentName = ""; // Configurable environment name
           to be embedded in Span attributes
19
20     // Getter and setter
21     public String getEndpoint() {
22         return endpoint;
23     }
24
25     public void setEndpoint(String endpoint) {
26         this.endpoint = endpoint;
27     }
28
29     public String getEnvironmentName() {
30         return environmentName;
31     }
32
33     public void setEnvironmentName(String environmentName) {
34         this.environmentName = environmentName;
35     }
36
37     public boolean isEnabled() {
38         return enabled;
39     }
40
41     public void setEnabled(boolean enabled) {
```

```
42          this.enabled = enabled;
43      }
44
45      // Create the OpenTelemetry instance (Spring managed @Bean)
46      // This instance can then be injected everywhere else in the code
47      @Bean("openTelemetry")
48      public OpenTelemetry initOpenTelemetry() {
49
50          // If instrumentation should be disabled, we return the
                OpenTelemetry.noop() bean, which is a no operation (dummy)
                OpenTelemetry instance.
51          if (!enabled) {
52              logger.info("OpenTelemetry Instrumentation is disabled.");
53              return OpenTelemetry.noop();
54          }
55
56          // Instrumentation is enabled -> configure
57          OtlpGrpcSpanExporter exporter = OtlpGrpcSpanExporter.builder()
58                  .setEndpoint(endpoint).build();
59
60          // Configure the batch span processor which exports the Spans to
                the Collector service in batches
61          BatchSpanProcessor batchSpanProcessor = BatchSpanProcessor.builder(
                exporter)
62                  .setMaxExportBatchSize(512)
63                  .setMaxQueueSize(2048)
64                  .setExporterTimeout(30, TimeUnit.SECONDS)
65                  .setScheduleDelay(5, TimeUnit.SECONDS)
66                  .build();
67
68          // Instantiate the SdkTracerProvider with the service.name
                attribute (in this case dory-rest)
69          SdkTracerProvider sdkTracerProvider = SdkTracerProvider.builder()
70                  .setResource(Resource.create(Attributes.of(AttributeKey.
                        stringKey("service.name"), SERVICE_NAME)))
71                  .addSpanProcessor(batchSpanProcessor).build();
72
73          // Build the OpenTelemetrySdk instance itself
74          OpenTelemetrySdk openTelemetrySdk = OpenTelemetrySdk.builder()
75                  .setTracerProvider(sdkTracerProvider)
76                  .setPropagators(ContextPropagators.create(
                        W3CTraceContextPropagator.getInstance()))
```

```
77                .build();
78
79        // Shutdown the SDK when the application exits
80        Runtime.getRuntime()
81                .addShutdownHook(
82                        new Thread(
83                                () -> {
84                                    logger.warn(
85                                            "*** forcing the Span Exporter to
                                                shutdown and process the
                                                remaining spans");
86                                    sdkTracerProvider.shutdown();
87                                    logger.warn("*** Trace Exporter shut down"
                                        );
88                                }));
89
90        return openTelemetrySdk;
91    }
92 }
```

Listing 5.1: `OpenTelemetryConfig.java` class.

An example `otel.properties` file to configure the OpenTelemetry setup in Dory REST is as follows:

```
1 # Example setting the endpoint for the collector service
2 otel.enabled=true
3 otel.endpoint=http://localhost:4317
4 otel.environmentName=local
```

Listing 5.2: Example `otel.properties` configuration file.

5.1.2 Wrapping the OpenTelemetry API Methods

With the `OpenTelemetry` bean instance now readily available for use through the Spring Dependency Injection (DI) mechanism using the `@Autowired` annotation, the naive approach would be to simply inject the configured `OpenTelemetry` bean everywhere in our code where needed. We decided that it is instead better to encapsulate the `OpenTelemetry` API with a *wrapper* class — the `TraceContext.java` class — as will be explained in Listing 5.3 below, which will give us the benefits of simpler code and better readability, while additionally also providing a suitable location to hold the `hostname` information to be embedded in the `Span` attributes.

Furthermore, in order to be able to generate the `Spans` and propagate the `Context` further, (1) we first need to configure the `Tracer` instance, and then the (2) `TextMapPropagator` to propagate the `Context`, which is provided by the `TraceContext` class.

```java
// Annotate as Spring Component
@Component
// The component should be request scoped, i.e. a new instance will be made at every incoming client request
@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode = ScopedProxyMode.TARGET_CLASS)
public class TraceContext {

    // Inject the OpenTelemetry to this wrapper class
    @Autowired
    private OpenTelemetry openTelemetry;

    // Save hostname here to avoid multiple calls to the Java API InetAddress.getLocalHost().getHostName()
    private String instance;

    @PostConstruct
    public void init() {
        try {
            this.instance = InetAddress.getLocalHost().getHostName();
        } catch (UnknownHostException e) {
            this.instance = "";
        }
    }

    // Simple wrapping of the OpenTelemetry APIs to get the Tracer and TextMapPropagator
    public Tracer getTracer(String name) {
        return openTelemetry.getTracer(name);
    }

    public TextMapPropagator getTextMapPropagator() {
        return openTelemetry.getPropagators().getTextMapPropagator();
    }

    public String getInstance() {
        return instance;
    }
}
```

Listing 5.3: `TraceContext.java` wrapper class for the OpenTelemetry API.

The `TraceContext` component will in turn be injected at other places to instrument the code, instead of injecting the `OpenTelemetry` bean directly. With the wrapper class

implemented, we can now turn to the actual implementation details on the manual instrumentation.

5.1.3 Implementation of the Incoming Requests Instrumentation

As previously explained, in the end the most elegant way to have the whole instrumentation of incoming requests from the client is through the implementation of a custom `Filter`.

The requirements for this custom `Filter` implementation is as follows:

- the `Filter` needs be able to extract the propagated `Context` (if present) from the incoming requests, which is embedded in the value of the `traceparent` header [31];
- the extracted `Context` needs to be stored and made readily available at other places in the code;
- the `Filter` must be able to generate a `Span` which is correlated to the propagated `Context`, and also to encapsulate the whole request execution;
- the `Filter` must also record relevant information on the `Span`, such as the HTTP status code, and the exceptions being thrown if exist;

It is better to explain the implementation directly with the code snippet as in Listing 5.4 below.

```
1  @Component
2  public class TraceFilter implements Filter {
3
4      protected static final Logger logger = LoggerFactory.getLogger(
           TraceFilter.class);
5
6      // Inject the TraceContext wrapper class as previously explained
7      @Inject
8      private TraceContext traceContext;
9
10     // Inject the OpenTelemetryConfig bean to get the configured
           environment name
11     @Inject
12     private OpenTelemetryConfig openTelemetryConfig;
13
14     // Define the TextMapGetter for HttpServletRequest carrier
15     // Here the carrier is HttpServletRequest as this object contains the
           traceparent header value (W3C standard)
16     private static final TextMapGetter<HttpServletRequest>
           servletRequestGetter =
```

```
17              new TextMapGetter<HttpServletRequest>() {
18                  // Need to implement the method which gives all the HTTP
                        header keys
19                  @Override
20                  public Iterable<String> keys(HttpServletRequest carrier) {
21                      // Here the method HttpServletRequest.getHeaderNames()
                            gives out the set of header keys
22                      return Collections.list(carrier.getHeaderNames());
23                  }
24
25                  // Here we need to implement the method for getting an HTTP
                        header value!
26                  @Nullable
27                  @Override
28                  public String get(@Nullable HttpServletRequest carrier,
                        String key) {
29                      // If the header key exists return the value, else
                            return empty string
30                      return carrier.getHeader(key) != null ? carrier.
                            getHeader(key) : "";
31                  }
32              };
33
34      // The custom filter implementation must always override the Filter.
            doFilter() method
35      @Override
36      public void doFilter(ServletRequest httpServletRequest, ServletResponse
             httpServletResponse, FilterChain filterChain) throws IOException,
            ServletException {
37          HttpServletRequest servletRequest = (HttpServletRequest)
                httpServletRequest;
38
39          // NOTE: we want to only instrument the requests to our REST API in
                 which the URI starts with /api
40          if (servletRequest.getRequestURI().startsWith("/api/")) {
41              instrumentRequest(servletRequest, httpServletResponse,
                    filterChain);
42          } else {
43              // Else we do not need to instrument the request
44              filterChain.doFilter(httpServletRequest, httpServletResponse);
45          }
46      }
```

```
47
48      private void instrumentRequest(HttpServletRequest httpServletRequest,
            ServletResponse httpServletResponse, FilterChain filterChain)
            throws ServletException, IOException {
49          // Extract the request Context from client
50          Context context = traceContext.getTextMapPropagator().extract(
                Context.current(), httpServletRequest, servletRequestGetter);
51          logger.info("Extracted context from TraceFilter with context: {}",
                context);
52
53          // Generate a new span, set the propagated context (if exists!) as
                the parent
54          Span span = traceContext.getTracer(TraceFilter.class.
                getCanonicalName())
55                  .spanBuilder(httpServletRequest.getRequestURI())
56                  .setParent(context).setSpanKind(SpanKind.SERVER)
57                  .startSpan();
58          // Set relevant span attributes
59          span.setAttribute("instance", traceContext.getInstance());
60          span.setAttribute("env", openTelemetryConfig.getEnvironmentName());
61
62          // Instrument the request execution
63          try (Scope scope = span.makeCurrent()) {
64              // Here the doFilter method encapsulates the whole request
                    execution until the response generation
65              filterChain.doFilter(httpServletRequest, httpServletResponse);
66          } catch (Throwable throwable) {
67              // Record exceptions which are thrown at execution for
                    debugging purposes
68              span.setStatus(StatusCode.ERROR, throwable.getMessage());
69              span.recordException(throwable);
70              throw throwable;
71          } finally {
72              // Set the HTTP response code for this request execution
73              // NOTE: this status code is the one returned to the client
                    from Dory REST (as the server)
74              HttpServletResponse servletResponse = (HttpServletResponse)
                    httpServletResponse;
75              span.setAttribute("http.status", servletResponse.getStatus());
76              span.setStatus(StatusCode.OK);
77              // Important to always close the span to avoid resource leaks!
78              span.end();
```

```
79          }
80      }
81  }
```

Listing 5.4: `TraceFilter.java`, a custom JEE Filter implementation which is responsible for instrumenting the incoming requests.

This concrete implementation follows the design approaches previously explained in Chapter 4.2.1.

One interesting point to be noted here is that we do not want to instrument *every* incoming HTTP requests to the `Dory REST` API service — which may include irrelevant HTTP requests such as the periodic probe requests to the Spring Boot Actuator's `/health` endpoint for evaluating the application health [1] — as this is irrelevant for our distributed tracing system. We can easily do that by filtering the requests and evaluating the `URI` string, with the relevant REST API calls to the `/api/` path (see Listing 5.4 above).

5.1.4 Implementation of the Outgoing Requests Instrumentation

Now we will turn to the implementation details of instrumenting the outgoing requests from `Dory REST`. This section will be divided into the three request types as previously elaborated in Chapter 4.2.2.

Requests to financial.com Backend Services

The requests to the financial.com backend services are already managed in a central component — the `RequestController.java` Component. The implementation is relatively straightforward, as the carrier object — the financial.com proprietary `FRequest.java` implementation which represents a backend request object to the corresponding services — already supports embedding header values.

The following code snippet will explain the code changes to the `RequestController.java` class which are relevant for the instrumentation implementation. As the focus of this Master's Thesis is the implementation of the distributed tracing system on the `Dory REST` platform, some part of the code on the `RequestController.java` class will be left out as they are irrelevant for the tracing implementation topic at hand, and is part of the proprietary financial.com AG intellectual property (IP) which may reveal the internal workings of our systems.

```
1  @Component
2  public class RequestController {
3      ...
4
5      // Need to inject TraceContext class for instrumentation
6      @Inject
```

[1] `https://docs.spring.io/spring-boot/docs/1.3.5.RELEASE/reference/html/production-ready-monitoring.html`

```
7     private TraceContext traceContext;
8
9     // Define the TextMapSetter for FRequest object as the carrier
10    private static final TextMapSetter<FRequest> fRequestSetter =
11            new TextMapSetter<FRequest>() {
12                // Implement the set method
13                @Override
14                public void set(@Nullable FRequest carrier, String key,
                      String value) {
15                    // Call the FRequest.putHeader(String key, String value)
                          method
16                    // The FRequest object supports embedding header values
                          in the request
17                    carrier.putHeader(key, value);
18                }
19            };
20
21    // The main method which handles the request execution
22    public FResponse execute(FClient server, String query) throws
          BackendQueryException {
23        FRequest request = server.createRequest(query);
24
25        // Inject the current context to the carrier (FRequest object)
              using the Setter function
26        logger.info("Injecting␣current␣context:␣{}␣to␣a␣csf␣backend␣request
              ␣to␣{}.", Context.current(), server.getType());
27        traceContext.getTextMapPropagator().inject(Context.current(),
              request, fRequestSetter);
28        FResponse<SuperByteString> response = null;
29
30        int statusCode = 200;
31
32        // Generate the Span
33        Span span = traceContext.getTracer(RequestController.class.
              getCanonicalName())
34                .spanBuilder("OutgoingCsfBackendRequest")
35                .setSpanKind(SpanKind.CLIENT)
36                .startSpan();
37        // Set relevant span attributes
38        span.setAttribute("instance", traceContext.getInstance());
39        span.setAttribute("env", openTelemetryConfig.getEnvironmentName());
40
```

```
41          // Make the current Span active
42          try (Scope scope = span.makeCurrent()) {
43              response = request.execute();
44          } catch(ExecutionException e) {
45              // Record thrown exceptions
46              span.setStatus(StatusCode.ERROR, e.getMessage());
47              span.recordException(e);
48              if (e.getCause() instanceof TimeoutException) {
49                  statusCode = 504; // Gateway timeout
50                  throw new BackendQueryException(server.getType(), e,
                        statusCode);
51              } else {
52                  statusCode = 500;
53                  throw new BackendQueryException(server.getType(),e);
54              }
55          } catch (SendException | InterruptedException |
                NullPointerException e) {
56              // Record thrown exceptions
57              span.setStatus(StatusCode.ERROR, e.getMessage());
58              span.recordException(e);
59              statusCode = 500;
60              throw new BackendQueryException(server.getType(),e);
61          } finally {
62              // Set the status code attribute in the Span and end
63              span.setAttribute("status.code", statusCode);
64              span.end();
65          }
66          return clientBackendResponse;
67      }
68  }
```

Listing 5.5: `RequestController.java`, the central component which handles the execution of requests to the financial.com backend services.

Generic HTTP Requests to Downstream Services

As we now turn to generic HTTP requests, this section will mainly focus on two main tasks as previously explained in Chapter 4.2.2. Because in the end we will only use the centralized `@Component` where we are managing the request execution *including* instrumentation, the more pragmatic way regarding the implementation is to first wrap the Apache HTTP Client, here concretely through the `DoryHttpClientWrapper.java` class, and *then* refactoring the code to replace the `RestTemplate` by the newly implemented `DoryHttpClientWrapper`. The following Listing 5.6 will explain the concrete implemen-

tation.

```
1  @Component
2  public class DoryHttpClientWrapper {
3
4      // Define the HTTP Header setter function
5      private static final TextMapSetter<ClientHttpRequest> httpHeaderSetter
           =
6              new TextMapSetter<ClientHttpRequest>() {
7                  @Override
8                  public void set(@Nullable ClientHttpRequest carrier, String
                       key, String value) {
9                      // ClientHttpRequest.getHeaders().set(String key, String
                           value) will set a HTTP header on the
                          ClientHttpRequest object
10                     carrier.getHeaders().set(key, value);
11                 }
12             };
13
14     // Inject the TraceContext object
15     @Autowired
16     private TraceContext traceContext;
17
18     @Autowired
19     private OpenTelemetryConfig openTelemetryConfig;
20
21     // Inject the actual Apache HTTP Client
22     @Autowired
23     private HttpComponentsClientHttpRequestFactory httpRequestFactory;
24
25     // Another Apache HTTP Client bean configured with proxy
26     @Autowired
27     @Qualifier(ApplicationConfiguration.HTTP_FACTORY_WITH_PROXY_BEAN_ID)
28     private HttpComponentsClientHttpRequestFactory
           httpRequestFactoryWithProxy;
29
30     public ClientHttpResponse executeRequest(URI uri, HttpMethod httpMethod
           , String downstreamServiceName, boolean proxied, HttpHeaders
           httpHeaders, byte[] httpBody) {
31         // Generate the Span
32         Span span = traceContext.getTracer(DoryHttpClientWrapper.class.
               getCanonicalName())
33                 .spanBuilder("DoryOutgoingHttpRequest")
```

```
34              .setSpanKind(SpanKind.CLIENT)
35              .startSpan();
36          // Set Span attributes
37          span.setAttribute("http.url", uri.toString());
38          span.setAttribute("http.method", httpMethod.toString());
39          span.setAttribute("component", "http");
40          span.setAttribute("instance", traceContext.getInstance());
41          span.setAttribute("env", openTelemetryConfig.getEnvironmentName());
42
43          // Instrument the request execution
44          try (Scope scope = span.makeCurrent()) {
45              ClientHttpRequest request;
46
47              try {
48                  if (proxied) {
49                      request = httpRequestFactoryWithProxy.createRequest(uri,
                            httpMethod);
50                  } else {
51                      request = httpRequestFactory.createRequest(uri,
                            httpMethod);
52                  }
53
54                  // Insert the HTTP body to the request if exists
55                  if (httpBody != null) {
56                      FileCopyUtils.copy(httpBody, request.getBody());
57                  }
58
59                  // Set the HTTP Headers
60                  if (httpHeaders != null) {
61                      request.getHeaders().addAll(httpHeaders);
62                  }
63
64                  // Inject Context to the outgoing request
65                  logger.info("Injecting␣current␣context:␣{}␣to␣an␣outgoing␣
                        HTTP␣request.", Context.current());
66                  traceContext.getTextMapPropagator().inject(Context.current()
                        , request, httpHeaderSetter);
67
68              } catch (IOException e) {
69                  span.setStatus(StatusCode.ERROR, e.getMessage());
70                  span.recordException(e);
71
```

```
72                 logger.info(e.getMessage());
73                 throw new BackendServerClientException(
                       BackendServerClientErrorCode.GENERAL_BACKEND_ERROR);
74             }
75
76             try {
77                 // Execute the actual request
78                 return request.execute();
79             } catch (IOException e) {
80                 span.setStatus(StatusCode.ERROR, e.getMessage());
81                 span.recordException(e);
82
83                 BackendServerClientErrorIdException be = new
                       BackendServerClientErrorIdException(
                       BackendServerClientErrorCode.
                       BACKEND_SERVER_NOT_AVAILABLE_ERROR,
                       downstreamServiceName);
84                 logger.info(be.getLogMsg(), e);
85                 throw be;
86             }
87         } finally {
88             span.end();
89         }
90     }
91 }
```

Listing 5.6: `DoryHttpClientWrapper.java` class for wrapping the Apache HTTP Client API.

The `RestTemplate` is a high level HTTP Client, which may also use Apache HTTP Client (or another supported HTTP Clients) underneath. It offers a higher level API and convenience methods, mainly for serializing and deserializing a Java `Object` to- and from `byte` array. When we migrate to the lower level Apache HTTP Client, this means that we need to handle the serialization and deserialization by ourselves. `RestTemplate` also does *client-side* error handling, in which the `4xx` error codes being thrown as `HttpClientErrorException` — providing a simpler error handling. Consequently, this means that we need to additionally handle the `4xx` client errors when we migrate to the Apache HTTP Client, as these errors are perfectly valid HTTP Responses from the view of the Apache Client and should not throw `Exceptions`.

These following code blocks, 5.7 and 5.8 will explain the refactoring works needed by comparing the usage of the two clients. As these refactorings will more or less always be the same across the different components, the code listings will only aim to illustrate the implementation tasks, and will be elaborated as examples. The important points to be noted here are the differences regarding the serialization and deserialization of the

request body and the exception handling, which are highlighted below.

```
1  // Example component where we do outgoing HTTP requests
2  @Component
3  public class MyComponent {
4      ...
5
6      // The injected RestTemplate bean
7      @Inject
8      @Qualifier(ApplicationConfiguration.REST_TEMPLATE_WITH_HTTP_FACTORY)
9      private RestTemplate restTemplate;
10
11     // The method which handles the client request
12     @Override
13     public Response handleRequest() {
14         ... // Irrelevant logic here
15
16         // Setting the HTTP Headers for the outgoing request
17         HttpHeaders headers = new HttpHeaders();
18         headers.put("Content-Type", Arrays.asList("application/json"));
19
20         // Create the RequestEntity Object
21         // As an example, the Object to be embedded on the request body is
               an instance of MyClass.java class
22         MyClass myInstance = new MyClass();
23         HttpEntity<MyClass> requestEntity = new HttpEntity<>(myInstance,
               headers);
24
25         ResponseEntity<Object> response = null;
26         try {
27             response = restTemplate.exchange(<Request URL>, HttpMethod.POST,
                   requestEntity, Object.class);
28         } catch (HttpClientErrorException ex) {
29             // Handle 4xx client errors here
30             fLogger.info("Error during request: {}", ex.
                   getResponseBodyAsString());
31             return Response.status(ex.getRawStatusCode()).entity(ex.
                   getResponseBodyAsString()).build();
32         } catch (Exception ex) {
33             // Handle 5xx server errors
34             fLogger.error(ex.getMessage(), ex);
35             return Response.status(Response.Status.INTERNAL_SERVER_ERROR).
                   build();
```

```
36          }
37      }
38  }
```

Listing 5.7: With `RestTemplate`.

```
1  // Example component where we do outgoing HTTP requests
2  @Component
3  public class MyComponent {
4      ...
5
6      // Inject the new DoryHttpClientWrapper class
7      @Autowired
8      private DoryHttpClientWrapper doryHttpClientWrapper;
9
10     // The method which handles the client request
11     @Override
12     public Response handleRequest() {
13         ... // Irrelevant logic here
14
15         // Setting the HTTP Headers for the outgoing request
16         HttpHeaders headers = new HttpHeaders();
17         headers.put("Content-Type", Arrays.asList("application/json"));
18
19         MyClass myInstance = new MyClass();
20         // Here we need the ObjectMapper to manually serialize/deserialize
               to- and from byte array
21         ObjectMapper objectMapper = new ObjectMapper();
22         byte[] body;
23         try {
24             // Try to serialize the MyClass instance to byte array
25             body = objectMapper.writeValueAsBytes(requestEntity);
26         } catch (JsonProcessingException e) {
27             throw new BackendServerClientException(
                   BackendServerClientErrorCode.GENERAL_BACKEND_ERROR, e.
                   getMessage());
28         }
29
30         /* The following is the method signature of DoryHttpClientWrapper.
               executeRequest()
31          * public ClientHttpResponse executeRequest(URI uri, HttpMethod
                httpMethod, String downstreamServiceName, boolean proxied,
                HttpHeaders httpHeaders, byte[] httpBody)
32          */
```

```
33          ClientHttpResponse response = doryHttpClientWrapper.executeRequest
                (<URL>, HttpMethod.POST, "
                downstreamServiceNameForInstrumentation", false, headers, body)
                ;
34
35          try {
36              // Try to deserialize the body to a Map object for the
                    response
37              Map<String, Object> resultBody = objectMapper.readValue(
                    response.getBody(), Map.class);
38
39              if (response.getStatusCode().isError()) {
40                  // Manually handle 4xx and 5xx errors
41                  logger.info("Error during request: {}", response.
                        getStatusText());
42                  return Response.status(response.getRawStatusCode()).entity(
                        resultBody).build();
43              } else {
44                  // Return response body on successful request
45                  responseBuilder.withData(resultBody);
46                  responseBuilder.withStatusCode((response.getStatusCode().
                        value()));
47              }
48
49          } catch (IOException e) {
50              // Throw a general backend error on IOException
51              throw new BackendServerClientException(
                    BackendServerClientErrorCode.GENERAL_BACKEND_ERROR, e.
                    getMessage());
52          }
53      }
54  }
```

Listing 5.8: With `DoryHttpClientWrapper`.

With these implementations explained above, we are now able to generate `Spans` for generic outgoing HTTP requests. Concrete examples on how this works will be shown more in Chapter 6.

HTTP Requests to ElasticSearch Services

The last major implementation work to be done is regarding the instrumentation of outgoing HTTP requests to ElasticSearch services, and the encapsulation of the request execution into a `Span`. Analogous to the previous problem regarding generic HTTP

request above, we decided the most pragmatic solution is also to modularize the request execution into a central component, where the instrumentation can be managed easily.

Currently, there are a couple of ElasticSearch `@Components` on our codebase, where there are repeated patterns as illustrated in the Example 5.9 below.

```
1  // Example ElasticSearch service class
2  @Scope("singleton")
3  @Component
4  public class MyElasticSearchService {
5      @Inject
6      RestHighLevelClient client;
7
8      ... // Irrelevant code parts
9
10     public SearchResponse search(<arbitrary method arguments to query
           ElasticSearch, depends on the index>) {
11         /* Build the ElasticSearch SearchRequest object.
12          * This logic to build the SearchRequest depends on the use case of
                the service itself.
13          * Here the example serves only to illustrate how such custom
                methods are implemented on our code base.
14          */
15         SearchRequest searchRequest = buildSearchRequest(<arbitrary method
               arguments to query ElasticSearch, depends on the index>);
16         SearchResponse searchResponse = runSearch(searchRequest);
17         return searchResponse;
18     }
19
20     // The actual method where the request execution happens
21     protected SearchResponse runSearch(SearchRequest request) {
22         SearchResponse searchResponse;
23         try {
24             searchResponse = client.search(request, RequestOptions.DEFAULT);

25         }
26         catch (ElasticsearchException e) {
27             fLogger.error("Elasticsearch error: " + e.getMessage());
28             throw new BackendServerClientException(
                   BackendServerClientErrorCode.
                   BACKEND_SERVER_NOT_AVAILABLE_ERROR, ELASTICSEARCH_MSG);
29         }
30         catch (Exception e) {
31             if (e.getCause() instanceof UnknownHostException)
```

```
32                 fLogger.error("UnknownHostException: " + e.getMessage());
33             else
34                 fLogger.error("Exception: " + e);
35             throw new BackendServerClientException(
                   BackendServerClientErrorCode.
                   BACKEND_SERVER_NOT_AVAILABLE_ERROR, ELASTICSEARCH_MSG);
36         }
37         return searchResponse;
38     }
39
40 }
```

Listing 5.9: Example of ElasticSearch `RestHighLevelClient` usage pattern.

The `runSearch` method illustrated above is more or less the same across different ElasticSearch service implementations in `Dory REST`, and hence can be modularized well into a central `Component`. The concrete *wrapper* class implementation — `ElasticSearchComponent.java` — can be seen in Listing 5.10, commented for clarity.

```
1  @Component
2  public class ElasticSearchComponent {
3      @Autowired
4      private RestHighLevelClient client;
5
6      @Autowired
7      @Qualifier("AnotherClientConfiguration")
8      private RestHighLevelClient anotherClient;
9
10     ... // Here arbitrary amount of RestHighLevelClient which requires
           custom configuration can be injected
11
12     @Autowired
13     private TraceContext traceContext;
14
15     @Autowired
16     private OpenTelemetryConfig openTelemetryConfig;
17
18     /* Modularize the runSearch method
19      *
20      * @param request the ElasticSearch SearchRequest object
21      * @param requestOptions ElasticSearch request options
22      * @return serviceName service name for deciding which
            RestHighLevelClient to be used and instrumentation purposes
23      */
```

```
24      public SearchResponse runSearch(SearchRequest request, RequestOptions
            requestOptions, String serviceName) {
25          // Generate the span and set relevant attributes
26          Span span = traceContext.getTracer(ElasticSearchComponent.class.
                getCanonicalName())
27                  .spanBuilder("ElasticSearchRequest")
28                  .setSpanKind(SpanKind.CLIENT)
29                  .startSpan();
30          // Set relevant span attributes
31          span.setAttribute("instance", traceContext.getInstance());
32          span.setAttribute("env", openTelemetryConfig.getEnvironmentName());
33
34          RestHighLevelClient searchClient;
35
36          // Example of differentiating between multiple ElasticSearch
                services (which requires usage of custom RestHighLevelClient
                configuration)
37          if (serviceName.equals("some␣condition")) {
38              logger.info("Using␣AnotherClientConfiguration");
39              searchClient = anotherClient;
40          } else {
41              logger.info("Using␣generic␣REST␣client␣for␣elasticsearch␣
                    request");
42              searchClient = client;
43          }
44
45          SearchResponse searchResponse;
46          // Instrument the request execution
47          try (Scope scope = span.makeCurrent()) {
48              // Execute ES search request
49              // Call the RestHighLevelClient.search(SearchRequest request,
                    RequestOptions requestOptions)
50              searchResponse = searchClient.search(request, requestOptions);
51          }
52          catch (ElasticsearchException e) {
53              logger.error("Elasticsearch␣error:␣" + e.getMessage());
54              // Record relevant exceptions
55              span.setStatus(StatusCode.ERROR, e.getMessage());
56              span.recordException(e);
57              throw new BackendServerClientException(
                    BackendServerClientErrorCode.
                    BACKEND_SERVER_NOT_AVAILABLE_ERROR, ELASTICSEARCH_MSG + "-"
```

```
                + serviceName);
58          }
59          catch (Exception e) {
60              if (e.getCause() instanceof UnknownHostException)
61                  logger.error("UnknownHostException: " + e.getMessage());
62              else
63                  logger.error("Exception: " + e);
64
65              span.setStatus(StatusCode.ERROR, e.getMessage());
66              span.recordException(e);
67              throw new BackendServerClientException(
                    BackendServerClientErrorCode.
                    BACKEND_SERVER_NOT_AVAILABLE_ERROR, ELASTICSEARCH_MSG + "-"
                    + serviceName);
68          } finally {
69              span.end();
70          }
71          return searchResponse;
72      }
73  }
```

Listing 5.10: `ElasticSearchComponent.java` class.

The last step is then to simply inject the `ElasticSearchComponent` to our ElasticSearch service classes and refactor the code to use the centralized request execution on `ElasticSearchComponent.runSearch(...)` method (compare with 5.9 above).

```
1  // Example ElasticSearch service class
2  @Scope("singleton")
3  @Component
4  public class MyElasticSearchService {
5      // Use our ElasticSearchComponent instead of RestHighLevelClient
           directly
6      @Inject
7      ElasticSearchComponent elasticSearchComponent;
8
9      ... // Irrelevant code parts
10
11     public SearchResponse search(<arbitrary method arguments to query
           ElasticSearch, depends on the index>) {
12         SearchRequest searchRequest = buildSearchRequest(<arbitrary method
               arguments to query ElasticSearch, depends on the index>);
13         // Run the search through ElasticSearchComponent
14         SearchResponse searchResponse = elasticSearchComponent.runSearch(
               searchRequest, RequestOptions.DEFAULT, "elasticsearch-service-
```

```
            name");
15          return searchResponse;
16      }
17  }
```

Listing 5.11: Refactoring the ElasticSearch service classes.

This concludes the implementation of the distributed tracing feature on the `Dory REST` side. The complete system diagram can be seen in Figure 5.1 below.

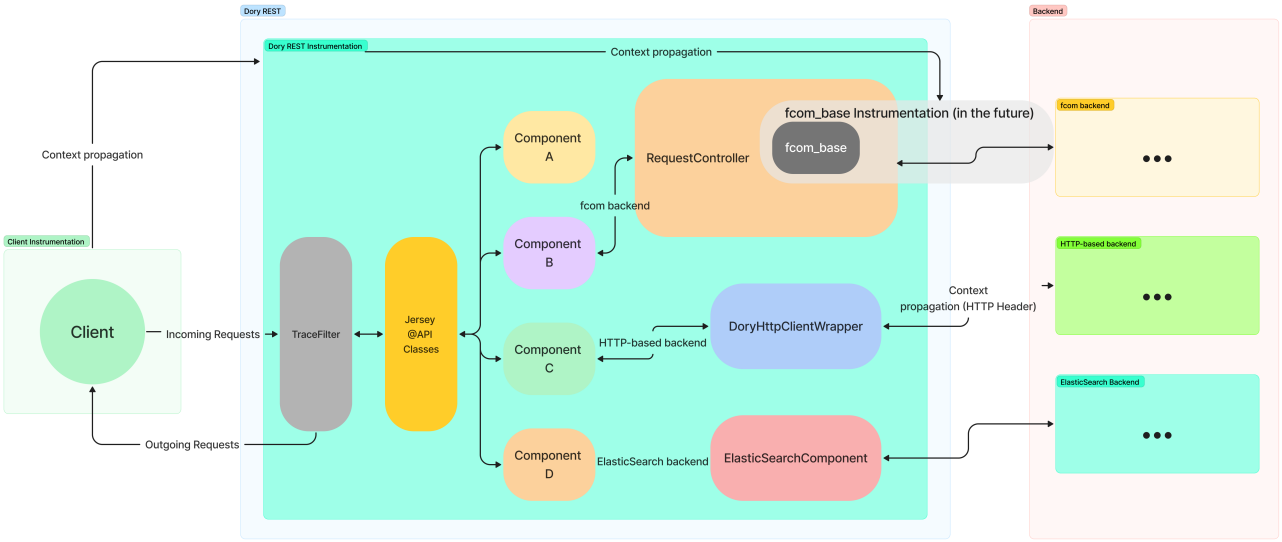

Figure 5.1: `Dory REST` instrumentation.

5.2 Integrating Other Services Into the Distributed Tracing Implementation

As previously explained above in the beginning of Chapter 5, we want to integrate a couple of *backend* services which falls under our team's responsibility to the distributed tracing solution, to have a Proof-of-Concept (PoC) and initial evaluations. It is important to note that the backend services that are being discussed here are Spring Boot services that communicate via REST interface, not the financial.com AG backend services that communicate via proprietary protocols on top of low-level Socket API through our internal `fcom_base` framework (see 4.2.2).

Because of this fact, the implementation is relatively straightforward and analogous to what was explained in 5.1.

A couple of key points to be noted here are as follows:

- The configuration and initialization of the `OpenTelemetry` instance are completely analogous to the one from `Dory REST`, configurable through the `otel.properties`

file;

- In case of these backend services, only the *incoming* requests will be instrumented, as they are acting as the data source;
- The instrumentation of the incoming requests is completely analogous to 5.1.3, implemented by the `TraceFilter.java` class.

6 Evaluation

6.1 Setting Up the Infrastructure

The evaluation of the distributed tracing system that was implemented in Chapter 5 will be done on our `development` environment of our internal Kubernetes Cluster. In the following section the configuration of the core services for the distributed tracing system will be explained.

6.1.1 Setting Up the Collector Service

We opted to set up the `jaeger-collector` service to act as the `Collector`, thereby eliminating the need for a standalone `OpenTelemetry Collector` deployment, as explained in Chapter 4.3.

Here we need to set a couple of relevant environment variables for the `jaeger-collector` service as follows:

```
1 SPAN_STORAGE_TYPE = elasticsearch # Use ElasticSearch storage backend
2 ES_SERVER_URLS = http://elasticsearch-otel-dev-es-elasticsearch:9200 # Set
      the ElasticSearch URL to our ES instance
3 COLLECTOR_OTLP_ENABLED = true # Enables the OTLP support on jaeger-
      collector
```

Listing 6.1: Environment variables to be set on the `jaeger-collector` service.

By default, the following ports also need to be exposed, with their respective functionalities explained below.

- Port `14268`: for receiving `Spans` in the `jaeger.thrift` format over Jaeger's binary thrift protocol using HTTP;
- Port `14250`: for receiving `jaeger.thrift` format `Spans` over gRPC;
- Port `4317`: for receiving OTLP (OpenTelemetry Protocol) `Spans` over gRPC;
- Port `4318`: for receiving OTLP `Spans` over HTTP;
- Port `14269`: for administrative purposes, such as health check, metrics, etc.

6.1.2 Setting Up the Observability Backend

We will also need to deploy the `jaeger-query` service to act as our *observability backend*. Additionally, it also provides a JavaScript UI for viewing the distributed traces. In this case we also need to set the following environment variables:

```
1 SPAN_STORAGE_TYPE = elasticsearch # Use ElasticSearch storage backend
2 ES_SERVER_URLS = http://elasticsearch-otel-dev-es-elasticsearch:9200 # Set
      the ElasticSearch URL to our ES instance
```

Listing 6.2: Environment variables to be set on the `jaeger-query` service.

Per default settings, the `jaeger-query` exposes the following ports:

- Port `16685`: for the gRPC QueryService;
- Port `16686`: exposes the `/api/*` Jaeger endpoints and the Jaeger UI at `/`;
- Port `16687`: to be used as the admin port, which is for health checking at `/` and for viewing the service's metrics at `/metrics`.

6.1.3 Setting Up the Dory REST and the Backend Services

In this section, the relevant configuration for deploying and configuring distributed tracing on `Dory REST` and other services will be elaborated. In this case, there is nothing to do except configuring the `otel.properties` file for the `Collector` service URL configuration — in this case the `jaeger-collector` located at `http://jaeger-collector-dev.dev-preview-green.svc.cluster.local:4317` — and the environment name in which it was deployed to, in this case our `dev` environment.

```
1 # Enable the instrumentation feature
2 otel.enabled=true
3 # Set the collector service endpoint
4 otel.endpoint=http://jaeger-collector-dev.dev-preview-green.svc.cluster.
      local:4317
5 # Set the environment name for informational purposes
6 otel.environmentName=dev
```

Listing 6.3: The otel.properties configuration file for the services.

6.2 Evaluating the Distributed Tracing System

With the individual components that make up the planned distributed tracing system now pieced together, we are ready to evaluate it in practice. The first phase is to evaluate how the system behaves on the `dev` environment, before doing performance tests against the `Dory REST` service — to see if there are any obvious performance impact.

6.2.1 Demonstration of the Distributed Tracing System

The evaluation on the `dev` environment will be relatively brief and straightforward, to see how the distributed tracing system and the services behaves and interacts with each other, and to test the `Context` propagation between the services.

As of now, there are three services configured and integrated into the distributed tracing system:

- `Dory REST`: the REST API middleware;
- `Dory Importers`: A helper backend service which is responsible for importing a specific data from a partner data source;
- `StoredChains`: A backend service which caches the data for special financial market *symbols*, which represents a *chain* of related instruments, for example the group of stocks on the DAX index.

`Dory REST` is the middleware REST API service for the frontends, and the other three are categorized as backend services — i.e. they act as *data sources* (Figure 6.1). In the following, we will be making test requests to `Dory REST` to certain endpoints, which will in turn invoke outgoing request to the other backend services mentioned above — to see the distributed tracing framework in action.

The test requests to demonstrate how `Context` propagation works in practice are made to the following endpoints, with relevant example request parameters explained below:

- `/quote/storedchain`: Retrieves chain instrument data from the `StoredChains` service.
  Parameters:
  - `rics`: comma separated list of chain Reuters Instrument Codes (RICs) [1];
    Example: `0#GFB.OL,0#ETF.OL`
  - `fids`: comma separated list of Field IDs (FIDs) — available financial data information on the corresponding RICs;
    Example: `q._DSPLY_NAME` — shows the display name of the financial instrument.
- `/quote/info`: Retrieves real-time quote information for the corresponding RICs.
  Parameters:
  - `rics`: comma separated list of generic Reuters Instrument Codes (RICs);
    Example: `AAPL.O` — RIC for Apple Inc.
  - `fids`: comma separated list of FIDs;
    Example: `q._TRDPRC_1` — shows the most recent trade price of the corresponding RIC.

[1] `https://en.wikipedia.org/wiki/Reuters_Instrument_Code`

- `/product/ubs/symbols`: An endpoint developed specifically to tailor the requirements of a specific client which depends on the `Dory Importers` service. Returns preconfigured financial data from a specific set of financial instruments without query parameters.

The requests were made through Postman API testing tool [2], where we can inject custom `traceparent` HTTP header values to test the `Context` extraction on Dory REST and the further injection to the backend services.

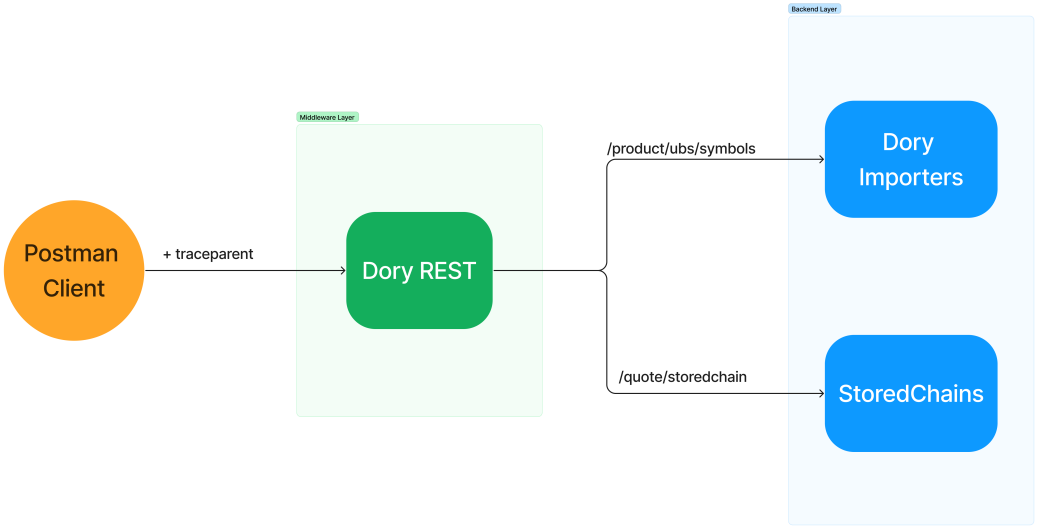

Figure 6.1: Demonstrating distributed tracing on the `dev` environment.

The format of the `traceparent` header is as follows:

- `version`: 1-byte value representing one 8-bit unsigned integer. The current specification from [31] assumes the current version `00`;
- `trace-id`: 16-byte array, uniquely identifying a single `trace` (refer to Chapter 2.3);
- `span-id`: Also called `parent-id` in some tracing systems, identifies a single `Span`;
- `trace-flags`: 8-*bit* field that controls the tracing flags, e.g. sampling, tracing level, etc.

An example of valid traceparent header value is shown below, taken from [31].

```
Value = 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
base16(version) = 00
base16(trace-id) = 4bf92f3577b34da6a3ce929d0e0e4736
```

[2] `https://www.postman.com/`

```
base16(parent-id) = 00f067aa0ba902b7
base16(trace-flags) = 01 // sampled
```

Listing 6.4: Example traceparent value.

For testing purposes, we will be demonstrating the following REST requests against the endpoints described above to the `Dory REST` service deployed on our `dev` environment (reachable internally at `https://dory-dev.financial.com/rest`), with mock — but *valid* according to the specifications described in [31] — traceparent values. While there are of course guidelines on how to generate the values within the `traceparent` header, for demonstration purposes only simple values will be used for simplicity.

- `GET https://dory-dev.financial.com/rest/api/quote/storedchain?rics=0%23GFB.OL%2C0%23ETF.OL&fids=q._DSPLY_NAME` — request to `StoredChains` backend service.
  Parameters:
  - `rics: 0#GFB.OL,0#ETF.OL`
  - `fids: q._DSPLY_NAME`

  Headers:
  - `traceparent: 00-11110000000000000000000000001111-1111000000001111-01`
  - `jwt: <jwt_access_token>`
- `GET https://dory-dev.financial.com/rest/api/quote/info?rics=AAPL.O&fids=q._TRDPRC_1` — request to `ExtraServer`, a financial.com proprietary backend service maintained by the backend team.
  Parameters:
  - `rics: AAPL.O`
  - `fids: q._TRDPRC_1`

  Headers:
  - `traceparent: 00-22220000000000000000000000002222-2222000000002222-01`
  - `jwt: <jwt_access_token>`
- `GET https://dory-dev.financial.com/rest/api/product/ubs/symbols` — request to the `Dory Importers` backend service.
  Headers:
  - `traceparent: 00-33330000000000000000000000003333-3333000000003333-01`
  - `jwt: <jwt_access_token>`

To view the generated telemetry data, we are using the `jaeger-ui` observability backend, with its built-in web interface. The `jaeger-ui` service on our `dev` environment is configured to be (internally) reachable at the URL `https://telemetry-dev.preview.financial.com/`.

We will now essentially be testing these points:

- If the injected `traceparent` value is correctly extracted at the entry point (Dory REST);
- If the extracted `Context` is further propagated to the downstream backend services;
- If the generated `Spans` from different services are properly correlated to form a *distributed* `trace` — thereby providing an end-to-end view of the request processing.

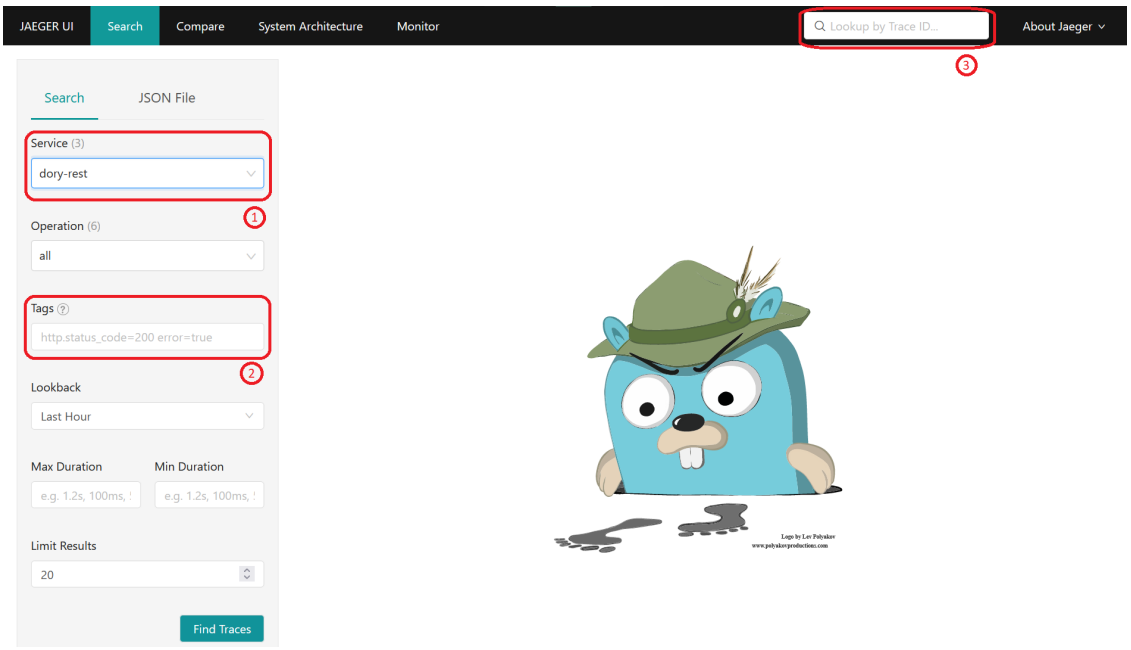

Figure 6.2: Jaeger web interface UI.

The `jaeger-ui` interface is shown in Figure 6.2, where we have the possibility of (1) selecting the instrumented service, (2) filtering the `Spans` according to the tags, and (3) searching the `traces` by its `trace-id`.

The distributed `traces` from the test requests (Chapter 6.2.1) are shown in Figure 6.3, with the expanded views shown beneath.

From the figures, it can be noted that:

- The `Spans` within one `Context` (i.e. having same `trace-id`) are properly correlated with each other, forming a distributed `trace` — with timing information and different colours on the `jaeger-ui` to signify different independent microservices — providing an end-to-end view of the whole request chain;
- The mock `traceparent` values embedded on the HTTP header are correctly extracted, and further propagated to the downstream services — with the first 7 hex digits of the `trace-id` shown in the `traces` correctly correspond to our mock `traceparent` values — as shown in Figures 6.4, 6.5, and 6.6;

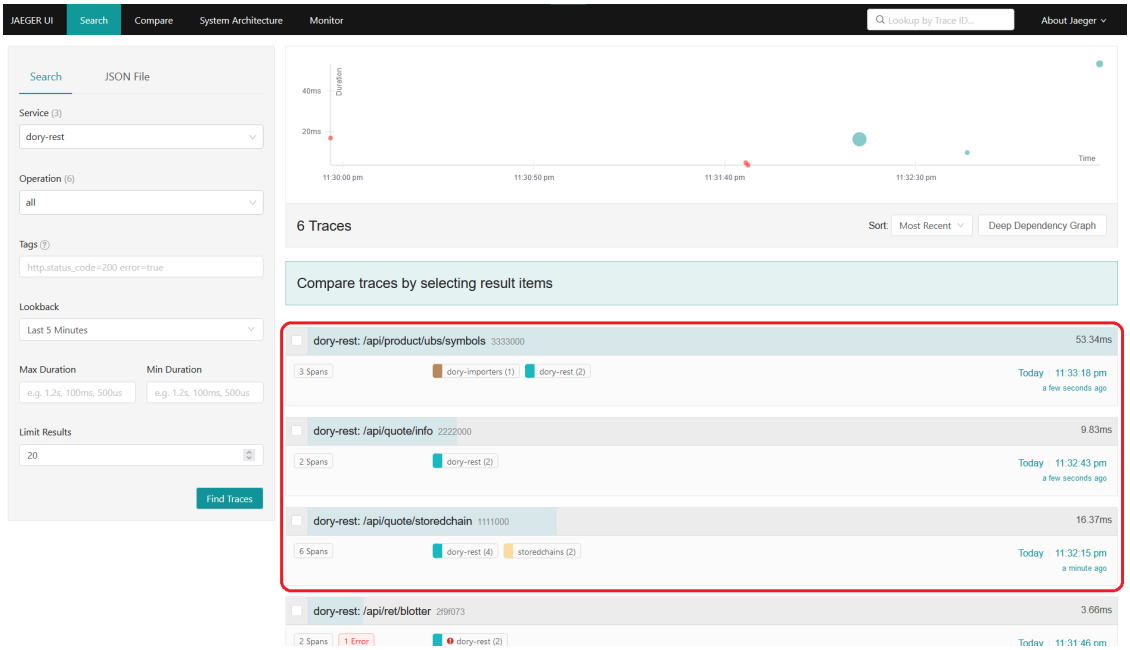

Figure 6.3: Resulting distributed traces of the example requests against `Dory REST`.

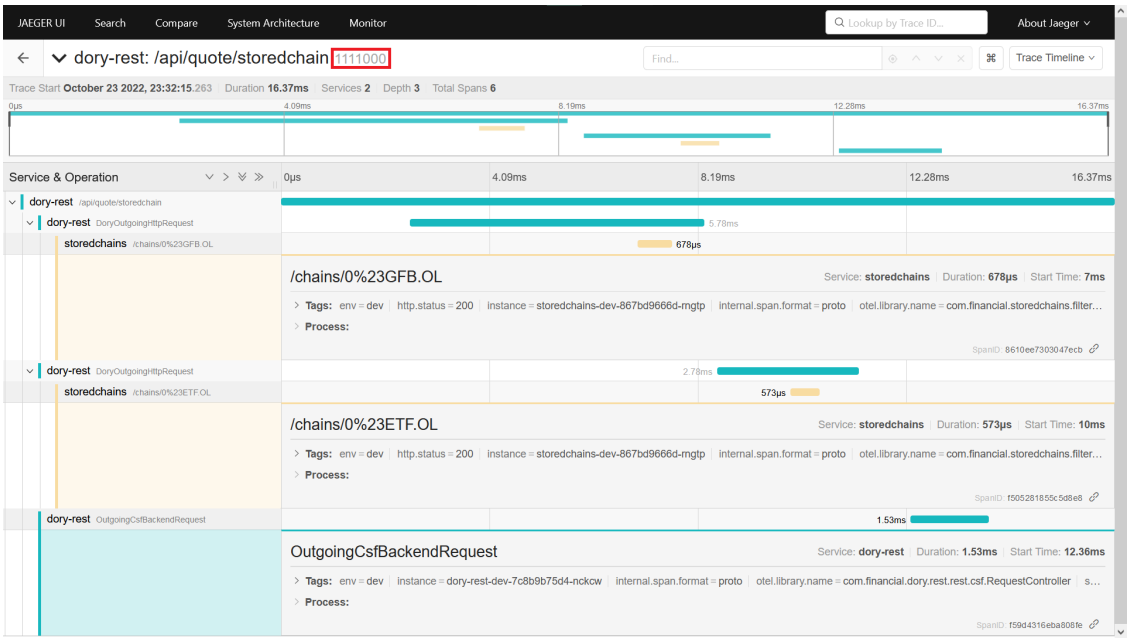

Figure 6.4: Detailed trace view of the `/quote/storedchains` request.

- The `Span` attributes are correctly set, which provides us the possibility of filtering the `traces`.

Therefore, the implemented distributed tracing solution fulfills the initial functional

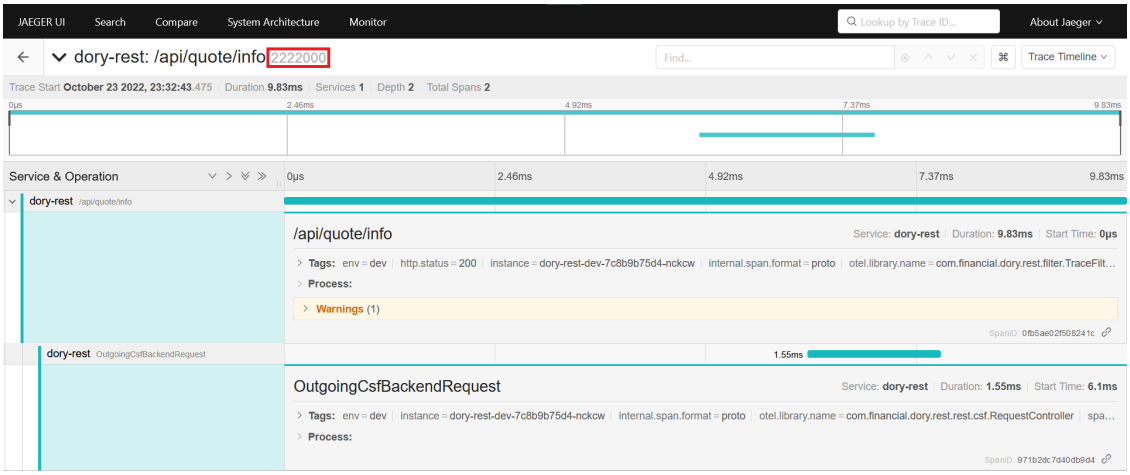

Figure 6.5: Detailed trace view of the `/quote/info` request.

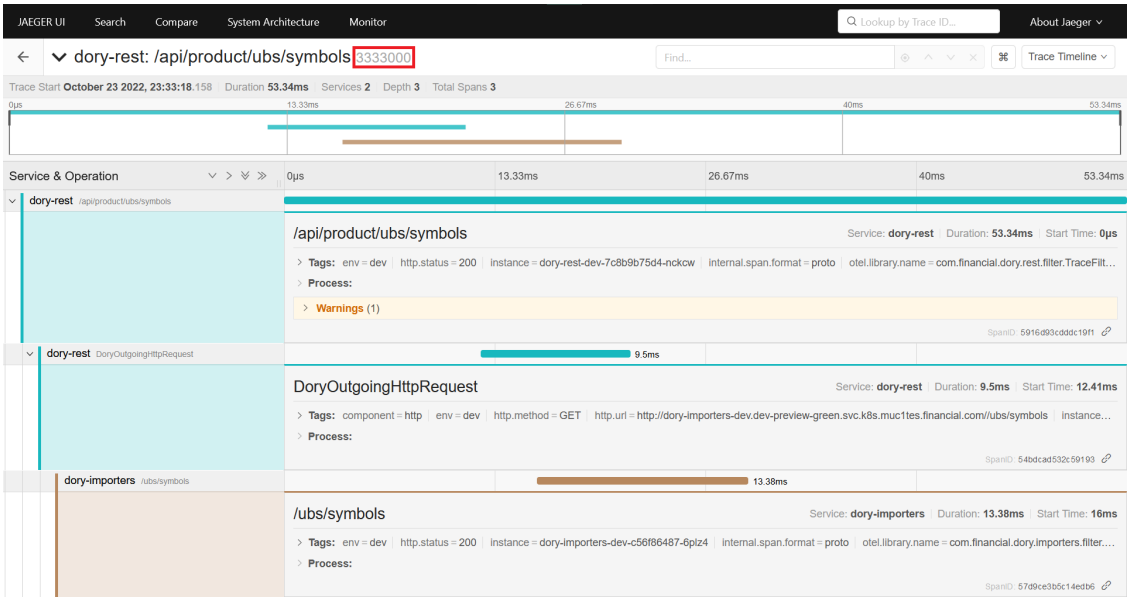

Figure 6.6: Detailed trace view of the `/product/ubs/symbols` request.

requirements defined at the start of the implementation works.

6.2.2 Evaluating the Performance Impact

While one of the key principles of the OpenTelemetry API is that the API should not degrade the end-user application — in terms of performance — as much as possible, there is inevitable overhead to achieve distributed tracing [3]. In this section, we will

[3] `https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/performance.md`

focus on doing initial performance testing on the `dev` environment, to get a rough idea on how much performance impact we can expect. If the results are deemed acceptable, this version will then be promoted to the `staging` environment, which is being actively used by the frontend teams for development purposes, and therefore should be kept relatively stable.

Internally at our middleware team, we are routinely using Gatling [4] for load testing purposes. The load testing scenarios are scripted in Scala language, which give us greater customization possibilities. The implementation of the load test classes will not be shown here for the sake of simplicity as we are using a load test scenario developed for one of our clients, which are designed to simulate real-world use case scenario with relevant request parameters.

The Dory REST service on the `dev` environment is configured as a Kubernetes `deployment` with 2 *pods* — each pod configured with 2 CPU cores, and `3500MiB` (3500 *Mebibytes* — 3.5 GiB) of memory.

The Gatling load test is organized in multiple scenarios, which correspond to the endpoint(s) being tested. Each scenario will be run with constant load of 1 user per second, for 300 seconds — totaling 300 executions in *one scenario*. Each scenario will have varying usage patterns derived from an actual performance testing suite developed for the purpose of capacity planning for a client.

Precisely, the test suite consists of the following scenarios:

- `UbsSymbology` scenario: Load tests the `/product/ubs/symbols` endpoint, which depends on the Dory Importers service.
- `UbsQuicksearch` scenario: Load tests the `/product/ubs/quicksearch` endpoint, which depends on a pre-configured ElasticSearch index.
- `Historical` scenario: Load tests the `/timeseries/historical` endpoint for historical market data — depends on the financial.com proprietary backend service.
- `Intraday` scenario: Load tests the `/timeseries/intraday` endpoint for *intraday* market data — depends on the financial.com proprietary backend service.
- `QuoteInfo` scenario: Load tests the `/quote/info`, which gives general information about a financial instrument — depends on the financial.com proprietary backend service.
- `QuoteStoredchain` scenario: Load tests the `/quote/storedchain` endpoint, which depends on the StoredChains backend service.
- `News` scenario: Load tests the `/news/stories` endpoint, which gives updates on global financial news. Depends on the financial.com proprietary backend service.

[4] `https://gatling.io/`

- `StreetEvents` scenario: Load tests the `/events/streetevents` endpoint for *events* information on a specific company, for example an earnings call — depends on the financial.com proprietary backend service.

In the following section, we will be comparing two configurations of the `Dory REST` service:

- `Dory REST` with distributed tracing implementations and instrumentation features enabled through the `otel.enabled=true` flag on the `otel.properties` configuration file — *configuration (1)*;
- Equivalent current `Dory REST` version, *without* any distributed tracing implementations on the codebase — that is, our *plain* version of the `Dory REST` service without the integration of OpenTelemetry APIs — *configuration (2)*.

For the load test configuration, we decided to deploy the *plain* versions of the `Dory Importers` and `StoredChains` service without the modifications regarding OpenTelemetry, so that we can measure the pure performance impact of the instrumentation on the `Dory REST` side.

The load test will be run for 3 times for each configuration in order to have more reliable results, which effectively means 900 runs *per scenario* (`3 * 300 executions`). The following results are taken directly from the Gatling load test reports, with only the average response times for each run of the test suite shown for simplicity, as the full reports attached in the submission of this Master's Thesis. The average of the mean response times on each run will then give us a rough idea on the response times of each scenario, as shown in Table 6.1 and 6.2 below.

Scenario	Average Response Time (ms)			
	#1	#2	#3	Mean
UbsSymbology	191	197	174	187.3
UbsQuicksearch	110	120	104	111.3
Historical	161	170	143	158
Intraday	191	179	159	176.3
QuoteInfo	119	126	103	116
QuoteStoredchain	176	182	166	174.7
News	233	245	223	233.7
StreetEvents	113	126	107	115.3

Table 6.1: Mean response times for Configuration 1.

The response times figures gave us some relevant information to be observed. Firstly, it is interesting to see that the *plain* `Dory REST` version without instrumentation — configuration (2) — gives very similar response time figures in comparison to configuration (1) with the instrumentation features enabled. While at first sight this may mean that the

Scenario	Average Response Time (ms)			
	#1	#2	#3	Mean
UbsSymbology	190	195	193	192.7
UbsQuicksearch	109	109	108	108.7
Historical	163	148	154	155
Intraday	172	178	178	176
QuoteInfo	109	115	115	113
QuoteStoredchain	175	177	177	176.3
News	235	247	231	237.7
StreetEvents	114	118	111	114.3

Table 6.2: Mean response times for Configuration 2.

differences are simply because of the variability of the data and that the instrumentation features should not have a noticeable overhead on the performance, we need to see the CPU load figures from the time of our load tests to confirm this hypothesis.

We are monitoring our application metrics using our internal Grafana [5] boards to see the actual CPU usage of the service. As seen in Figure 6.7 and Figure 6.8 which are taken from our Grafana board at the time of the load tests, the actual CPU usage between the 2 configurations compared above is very similar, hovering at about 4% CPU usage, both reaching about 6% CPU usage at maximum. These initial results support the hypothesis that the instrumentation should not have a noticeable impact on the performance.

To be sure that the performance impact is indeed minimal, we decided to run a second suite of load tests which tests the system at its limit — to see the amount of maximum requests per second that we can achieve for both configurations. In this case, `Dory REST` will be deployed as a *single* pod — configured with 2 CPUs and 3.5 GiB of memory same as above — to test the system's limit. It is important to note that we need to disable the Readiness Probe [6] here, so that Kubernetes doesn't kill the pod because the health check calls to Spring Boot's `/actuator` endpoint are responded slowly under load.

In this case, we will be running tests and tweaking the number of users per second, as well as the test duration as the load testing parameters, to see which is the limit before the system is overwhelmed by the amount of the requests. Here we will try to get the highest possible amount of requests per second for one pod of the `Dory REST` service while keeping the percentage of failed requests at 0% — meaning we will not be accepting results at the point where the requests starts failing, in other words the system is already over its limit. It is also interesting to see, that the percentage of failed requests will start going up exponentially when the system has exceeded its limit as we

[5] `https://grafana.com/`

[6] `https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/`

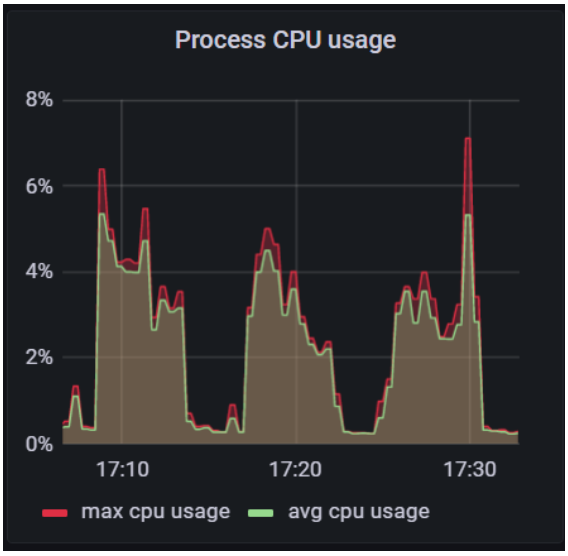

Figure 6.7: CPU usage from load testing on configuration 1.

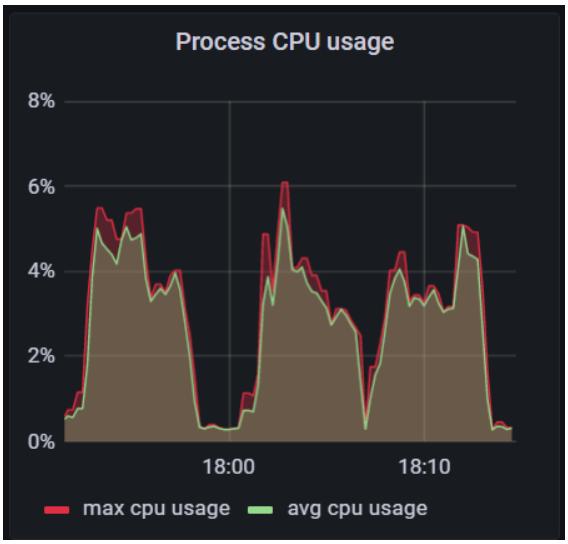

Figure 6.8: CPU usage from load testing on configuration 2.

increase the number of users per second, providing us with a relatively good indicator on the limits of each configuration. We opted to keep the test duration to 900 seconds (15 minutes) to see if the system still behaves stably in a relatively longer load scenario, while adjusting the number of users per second to find the limits of each configuration described above.

After running multiple load tests on each configuration to have a reproducible result, we converged to the following figures (refer to the attached load testing reports for more information):

- Configuration (1) — instrumentation enabled,
  Maximum users per second: `12`;
  Requests per second: $\sim$ `130.891 Req/s`;
  Peak number of concurrent users: $\sim$ `291 users`;
  Increasing the amount of users to 13 per second results in requests start failing (see attached load testing reports).

- Configuration (2) — plain `Dory REST`,
  Maximum users per second: `12`;
  Requests per second: $\sim$ `130.891 Req/s`;
  Peak number of concurrent users: $\sim$ `273 users`;
  Increasing the amount of users to 13 per second starts leading to failing requests.

The results from load testing a single pod to its limit as described above tells a similar story. Here the results are effectively the same, again supporting the hypothesis that instrumentation should not have a noticeable impact on overall system's performance.

The initial load testing figures gave us the confidence that the integration of the OpenTelemetry API, and the manual instrumentation itself should not give a noticeable performance impact *under normal situations*, which is one of the design goals explained in Chapter 3.2. The next step is then to deploy this Proof-of-Concept version of `Dory REST` with manual instrumentation to our `staging` environment at our next release window, where we can observe the system with real load for a couple of months, before promoting it to our `production` environments.

7 Related Works

7.1 Different Approach to Pinpoint Failures on a Distributed System

In the end, what OpenTelemetry aims to solve is the difficulty of debugging applications in a distributed system. While OpenTelemetry and other similar distributed tracing system attempt to solve this abstract problem through the concept of *Observability* — meaning to make the distributed application more observable in order to be able to pinpoint the fault quickly — of course it's not the only solution.

One other approach from Marwede et al. [21] is to detect the failure by analyzing the software behaviour — in this case the *timing* behaviour. The system attempts to localize faults by (1) computing the *anomaly scores* of the components, and then (2) correlating the anomaly scores *globally* (at the whole distributed system level) to pinpoint the bug.

At its simplest, the *RanCorr* system analyzes the timing behaviour anomaly. In the first step, the normal system behaviour is profiled from the historical timing behaviours — for example the response times. This will then be compared to the *current* system behaviour in order to detect fault indicators. The second step is then to localize the failure's root-cause based on the computed anomaly scores obtained from comparing the ideal system behaviour and the current.

While the solution is perfectly valid, there are a couple of disadvantages with this approach:

- Timing behaviour anomaly may be propagated across the system, i.e. a component that shows anomaly signs might only be anomalous because it depends on another component that *might* be the real root-cause — hence complicating the detection problem;
- Consequently, the individual anomaly scores of the components need to be correlated with each other to compensate such propagation effects, and in the end provide more accurate fault localization;
- The approach is resource-intensive, as the ideal system state has to be profiled first, and the correlation algorithm is complex and depends on the system's dependency graph. Furthermore, the system works mainly on *assumptions*, i.e. comparing the current system state and the ideal to make educated guesses on the root-cause.

8 Summary

8.1 Summary

In the end, the main focus of this Thesis is on the challenges of operating microservices on an industrial scale. In the first phase, the challenges are elaborated in a real world situation at the cooperating company — financial.com AG. As elaborated in Chapter 2.2, the major problem with Microservice Architecture (MSA) is the loss of coherence in the entire system. This in turn causes a whole other set of challenges, one of them the difficulty of troubleshooting and quickly pinpointing the location of the software fault in the entire system — a trait which is not found in the previous age of *monolithic* architecture, as the application is a single, big unit, consisting of multiple components responsible for different tasks, thereby simplifying the troubleshooting process.

While this concrete problem can be solved in many ways, in this project we opted to solve it through the distributed tracing approach — in essence making the entire distributed system more *observable*. Distributed tracing solves this problem by utilizing the concept of Context *propagation*, in the end passing the current `Context` (with regard to the current request from the client) further to *downstream* services, which results in `Spans` that can be correlated with each other, regardless of the service boundaries.

We decided to utilize the `OpenTelemetry` framework as the backbone of our distributed tracing system. This decision is driven by the fact that OpenTelemetry is the foremost distributed tracing framework which attempts to unify the instrumentation standard in the industry, and is supported by the Cloud Native Computing Foundation (CNCF).

In the next phase, the design decisions to integrate OpenTelemetry into our services were discussed in Chapter 4, which highlights the specific decisions being taken in consideration of the real-world use cases, as this project is implemented on the microservices which are being actively developed at financial.com AG in Munich, and will be promoted to the `production` systems in the future. Consequently, we need to be very sure that the performance impact is minimal and not noticeable to the end-users. For this purpose, we performed initial load testing using the Gatling stress test tool with realistic use-case scenarios developed internally, which were originally designed for capacity planning, and hence simulates real-world usage patterns.

As shown in Chapter 6.2.2, initial load testing on the `dev` environment shows promising results, with the three Dory REST configurations described above giving essentially very similar response times — which indicates that the manual instrumentation has no noticeable performance impact, fulfilling the design goals. This early Proof-of-Concept version of the Dory REST (and the other backend services under our team's responsibility as described in Chapter 6.2.1) will be promoted to our `staging` system at our next

release window, so that we can observe how the system behaves under semi-realistic load as it is used for frontend development, with the intent of deploying the feature into our `production` systems in the future.

8.2 Conclusion

In the beginning, it was shown that the industrial trend is moving towards the microservice architectural style. This is mainly driven by the boom in web technologies since the start of the Internet age, which pushes the requirement regarding scalability of the software systems. It was observed that *decomposing* the application into independent units which specifically handles a set of functionalities helps with the problem of scalability, and gives the benefit of being able to scale the individual services independently according to the usage patterns. This results in the evolution of the architectural style, from the traditional *monoliths*, to Service-Oriented Architectures (SOA), and most recently to the relatively young Microservice Architecture (MSA).

While MSA brings the benefits which are crucial to power our current modern web applications, the biggest weakness which is not present in monolithic architecture is the difficulty of troubleshooting the software system, as the application is decomposed into multiple services which communicates with each other — significantly increasing the complexity of pinpointing the fault. This situation is also observed in the cooperating company for this study, financial.com AG. While there are different approaches to solve this problem, one approach — distributed tracing — stands out in the industry, with the trend clearly heading in the direction.

Distributed tracing attempts to solve the difficulty of troubleshooting for the developers by making the whole distributed system *observable*. The main idea is to propagate a set of metadatas in a specific scope — the *Context* — across service boundaries. As demonstrated in Chapter 6.2.1, this gives developers the possibility of tracing a client request across service boundaries in a distributed system, and pinpointing the exact location of the software fault — greatly reducing the costs and efforts required to troubleshoot the problem, which commonly involves multiple different teams that are responsible for different services. In the end, with the concrete goal of increasing efficiency and therefore customer satisfaction.

In the course of this project, we chose OpenTelemetry as our distributed tracing framework of choice, with the design considerations elaborated in previous chapters. This project has shown that OpenTelemetry is able to solve our concrete problem described above, and the initial concept fulfills the design requirements defined at the start of this project (Chapter 3.2).

While there are still future works to do, such as to observe how the system behaves on our `staging` environment, and also rolling them out to the `production` systems in the future, we can conclude that the result from this project is promising and exactly what we need in this current age of microservices, therefore fulfilling the scope of this project.

9 Future Works

9.1 Integrating the Frontends into the Distributed Tracing System

Recall that the works being discussed in this Master's Thesis mainly focuses on the *server* side instrumentation — on the Dory REST as the middleware, and the backend services — but not on the frontend side of the equation. There may be plans to also integrate the multiple frontends in our infrastructure into the distributed tracing system.

The OpenTelemetry framework also supports JavaScript, and *client-side*, or browser instrumentation [1]. Frontend instrumentation would be very desirable, as we would have a complete end-to-end observability, vastly simplifying the process of localizing the fault in a distributed setup.

There is one challenge, though, as financial.com AG develops tailored frontends for many different clients — hence there are relatively numerous JavaScript frontend components. Consequently, the implementation efforts for these frontends will of course be higher, as it involves many software projects and the different teams responsible for them.

There may be future works being planned in order to have the frontends integrated into our distributed tracing system — assuming the efforts and costs are feasible.

9.2 Exploring the Possibility of Mixing Automatic and Manual Instrumentation

In the beginning of this project, it was explained that there are concerns of using OpenTelemetry automatic instrumentation which is supported on the Java platform, as automatic instrumentation works by dynamically modifying the Java bytecode after compilation, and there are concerns of unintended side-effects. The other reason is that the automatic instrumentation works according to black-box principle, with no control over the telemetry data generation itself. Concretely, this means that we may get a lot more instrumentation data — sometimes also at places that do not need or aren't relevant to be instrumented, generating more *noises*.

Even so, there are a couple of benefits of using automatic instrumentation, namely:

- We are able to get a lot of internal Spring libraries instrumented, for example we do not need to implement the instrumentation of incoming and outgoing

[1] `https://opentelemetry.io/docs/instrumentation/js/getting-started/browser/`

HTTP requests as previously explained in Chapter 5 manually — with a lot more information automatically embedded in the `Spans`;

- Until now, we haven't considered to instrument calls to the persistence layer (the Database). By instrumenting the application automatically using the Java agent, these JDBC calls are also instrumented automatically — possibly giving us greater insights into the distributed application;
- The attributes are uniformly set on the `Spans`, compliant to the OpenTelemetry Semantic Conventions [2];
- The Java agent also supports environment variables which controls which components/libraries are automatically instrumented — thereby providing us with the possibility of fine-tuning the automatic instrumentation in order to avoid the problem of too much *noise* on the traces as described above.

Considering these benefits, there are plans on our team to explore this possibility of mixing *both* manual and automatic instrumentations on Dory REST, to see how it behaves and if it fulfills our requirements.

9.3 Instrumenting Other Types of Outgoing Calls From Dory REST

Lastly, in addition to the outgoing request types previously explained in Chapter 4.2.2, there are also other outgoing calls being made from Dory REST, for example to the Database (JDBC calls) and to Redis in-memory data stores [3]. It is then interesting to explore the possibilities of instrumenting these types of requests not covered in the scope of this Master's Thesis, either by manually implementing them, or through the mix of automatic and manual instrumentation described above.

[2] `https://opentelemetry.io/docs/reference/specification/metrics/semantic_conventions/`
[3] `https://redis.io/`

List of Figures

List of Tables

Bibliography

[1] C. P. Josuttis, O. Zimmermann, M. Amundsen, J. Lewis, and N., "Microservices in practice, part 1: Reality check and service design," *IEEE Software*, vol. 34, pp. 91–98, 2017, ISSN: 1937-4194. DOI: 10.1109/MS.2017.24.

[2] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 21–30, 2017. DOI: 10.1109/ICSA.2017.24.

[3] W. Hasselbring and G. Steinacker, *Microservice architectures for scalability, agility and reliability in e-commerce*, Conference Paper, 2017. DOI: 10.1109/icsaw.2017.11.

[4] B. J. Williams and J. C. Carver, "Characterizing software architecture changes: A systematic review," *Information and Software Technology*, vol. 52, no. 1, pp. 31–51, 2010, ISSN: 09505849. DOI: 10.1016/j.infsof.2009.07.002.

[5] K. Gos and W. Zabierowski, "The comparison of microservice and monolithic architecture," *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pp. 150–153, 2020. DOI: 10.1109/MEMSTECH49584.2020.9109514.

[6] T. Cerny, M. J. Donahoo, and M. Trnka, *Contextual understanding of microservice architecture: Current and future directions*, Conference Paper, 2017. DOI: 10.1145/3183628.3183631.

[7] D. Taibi, V. Lenarduzzi, C. Pahl, and A. Janes, *Microservices in agile software development: A workshop-based study into issues, advantages, and disadvantages*, Conference Paper, 2017. DOI: 10.1145/3120459.3120483.

[8] R. Stephens, *Beginning Software Engineering*. John Wiley and Sons, 2015.

[9] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," *2015 10th Computing Colombian Conference (10CCC)*, pp. 583–590, 2015. DOI: 10.1109/ColumbianCC.2015.7333476.

[10] K. B. Laskey and K. Laskey, "Service oriented architecture," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 1, no. 1, pp. 101–105, 2009, ISSN: 1939-5108 1939-0068. DOI: 10.1002/wics.8.

[11] D. Sprott and L. Wilkes, "Understanding service-oriented architecture," *The Architecture Journal*, vol. 1, pp. 10–17, 2004.

[12] J. McGovern, O. Sims, A. Jain, and M. Little, *Enterprise Service Oriented Architectures: Concepts, Challenges, Recommendations*. Springer, 2006, vol. 20.

[13] J. Hutchinson, G. Kotonya, J. Walkerdine, P. Sawyer, G. Dobson, and V. Onditi, "Evolving existing systems to service-oriented architectures: Perspective and challenges," *IEEE International Conference on Web Services (ICWS 2007)*, 2007. DOI: 10.1109/ICWS.2007.88.

[14] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research challenges," *Computer*, vol. 40, pp. 38–45, 2007. DOI: 10.1109/MC.2007.400.

[15] C. Richardson. "Microservices | pattern: Microservices architecture." (2014), [Online]. Available: https://microservices.io/patterns/microservices.html (visited on 07/05/2022).

[16] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Thesis, 2000.

[17] S. Vinoski, "Rest eye for the soa guy," *IEEE Internet Computing*, vol. 11, pp. 82–84, 2007.

[18] A. Khan, P. Steckmeyer, N. Peck, E. Shanmugam, S. Muraleedharan, and L. Arcega. "Running containerized microservices on aws." (2017), [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/running-containerized-microservices/smart-endpoints-and-dumb-pipes.html (visited on 07/06/2022).

[19] OpenTelemetry. "Opentelemetry docs | observability primer." (2022), [Online]. Available: https://opentelemetry.io/docs/concepts/observability-primer/ (visited on 07/08/2022).

[20] I. Lightstep. "Lightstep | opentelemetry docs." (2022), [Online]. Available: https://opentelemetry.lightstep.com/ (visited on 07/13/2022).

[21] N. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring, *Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation*, Conference Paper, 2009. DOI: 10.1109/csmr.2009.15.

[22] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online modelling and performance-aware systems," *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[23] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI 07)*, 2007.

[24] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010.

[25] OpenTelemetry. "Opentelemetry docs | specification." (2022), [Online]. Available: https://opentelemetry.io/docs/reference/specification/overview/ (visited on 07/11/2022).

[26] I. Lightstep. "Lightstep | opentelemetry | what is distributed tracing?" (2022), [Online]. Available: https://lightstep.com/opentelemetry/tracing (visited on 07/11/2022).

[27] OpenTelemetry. "Opentelemetry docs | concepts." (2022), [Online]. Available: https://opentelemetry.io/docs/concepts/what-is-opentelemetry/ (visited on 08/31/2022).

[28] I. Lightstep. "Distributed tracing: A complete guide." (), [Online]. Available: https://lightstep.com/distributed-tracing#distributed-tracing-tools (visited on 08/29/2022).

[29] OpenTelemetry. "Opentelemetry docs | instrumentation | java." (2022), [Online]. Available: https://opentelemetry.io/docs/instrumentation/java/ (visited on 09/01/2022).

[30] OpenTelemetry. "Opentelemetry docs | collector." (2022), [Online]. Available: https://opentelemetry.io/docs/collector/ (visited on 09/03/2022).

[31] W. W. W. C. (W3C). "W3c recommendation | trace context." (2021), [Online]. Available: https://www.w3.org/TR/trace-context/ (visited on 09/04/2022).