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

Jonathan Ryan Wijaya Tumboimbela

Munich, 02.12.2022

Outline

Introduction

Microservice Architecture in Practice

Microservices are everywhere in the industry today!

Evolution of Software Architectures

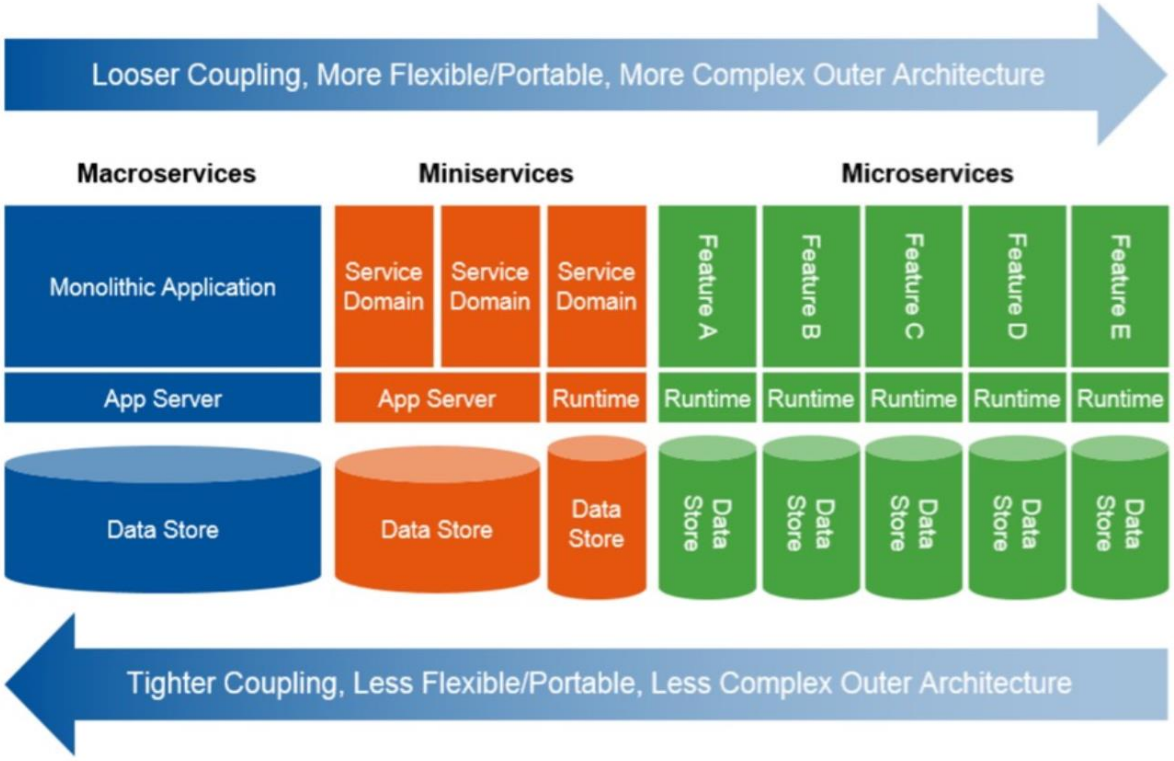


Figure 1: Comparison of software architectures. From “MICROSERVICE ARCHITECTURE : BENEFITS AND CHALLENGES by SmartWave. Retrieved from https://www.smartwavesa.com/blog-articles/microservice-architecture-benefits-and-challenges/. Copyright 2018 by Gartner, Inc.

Microservice Architecture in Practice

The benefits are clear [1]:

- more scalable
- better resiliency — fault isolation
- better time-to-market
- suits agile methodology
- improved productivity for the developers (not always — more on this next!)

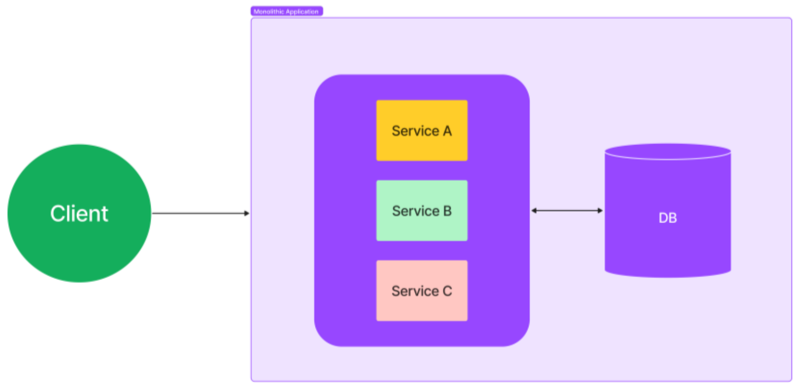


Figure 2: Monolithic Architecture

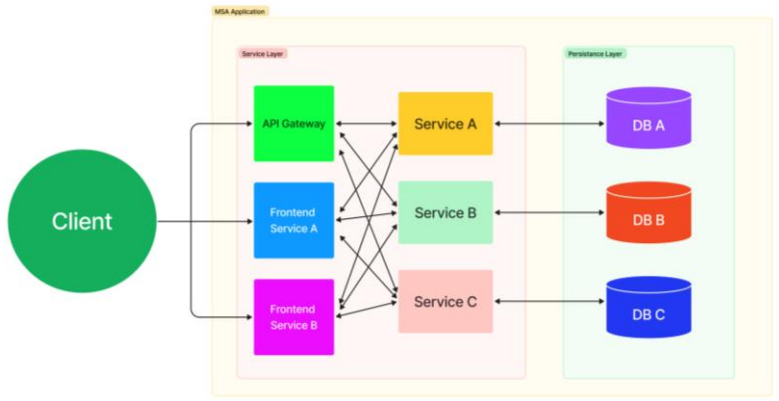


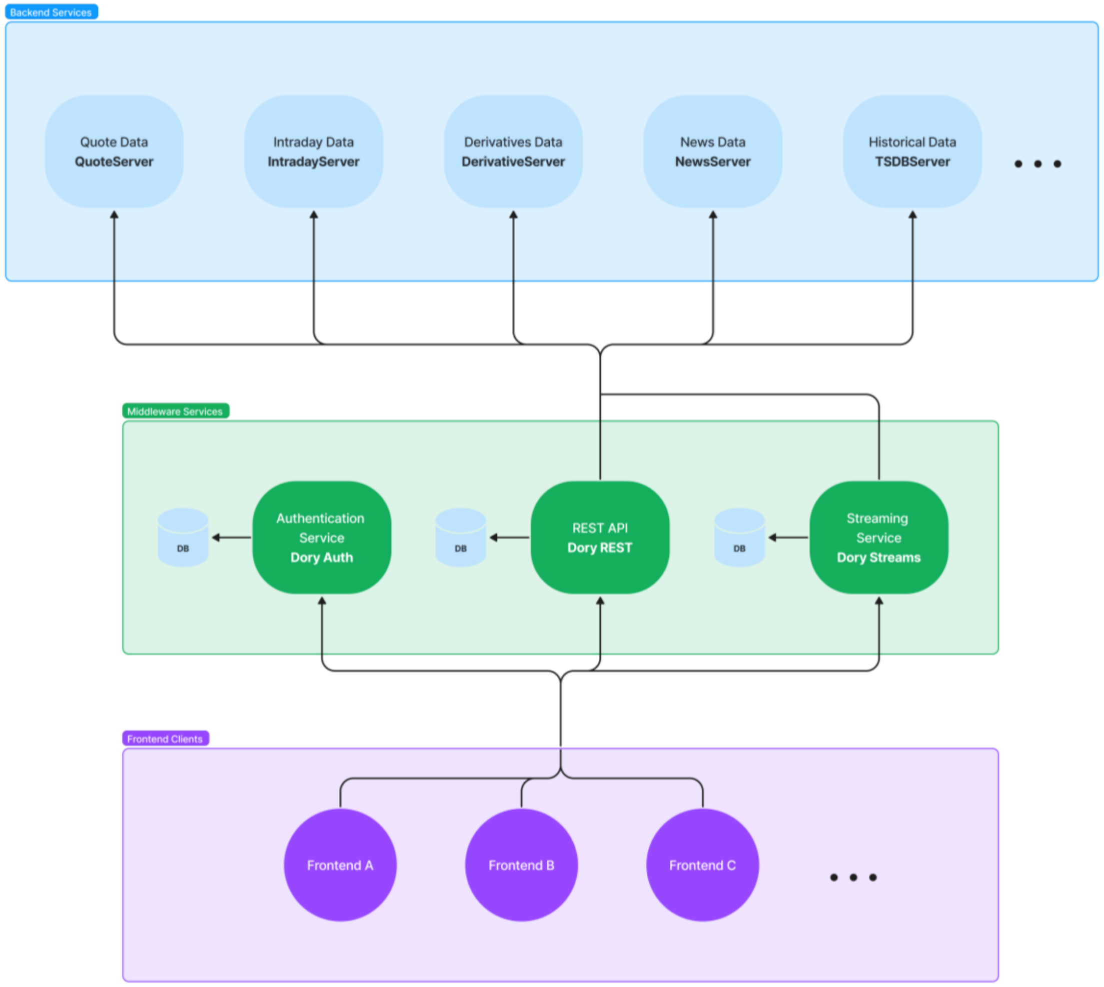
Figure 3: Microservice Architecture

Problem

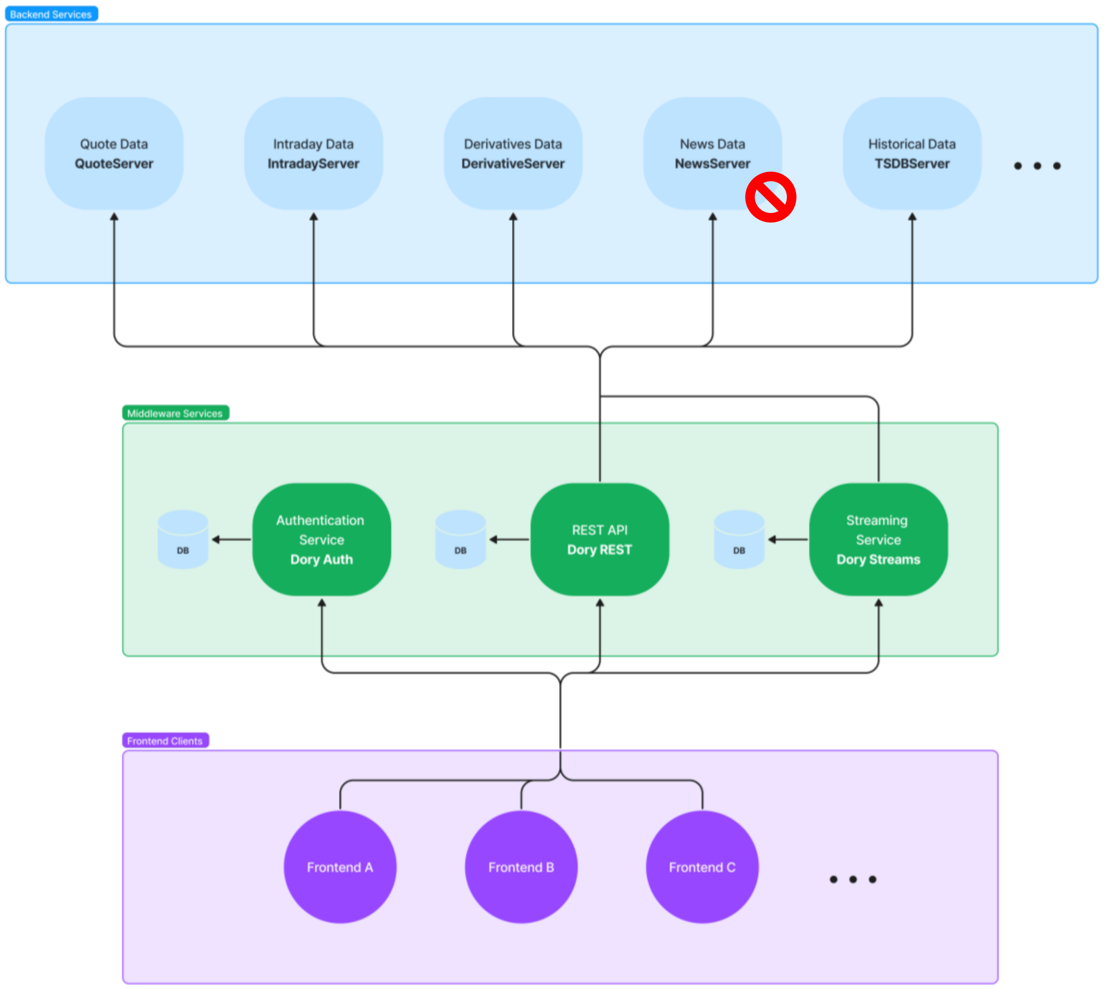
Microservice Architecture (MSA) introduces a whole new set of challenges which are not present with Monolithic Architecture!

- Increased communication complexity between the services
- More resource-hungry
- End-to-end testing is difficult
- Increased troubleshooting complexity for the developers!

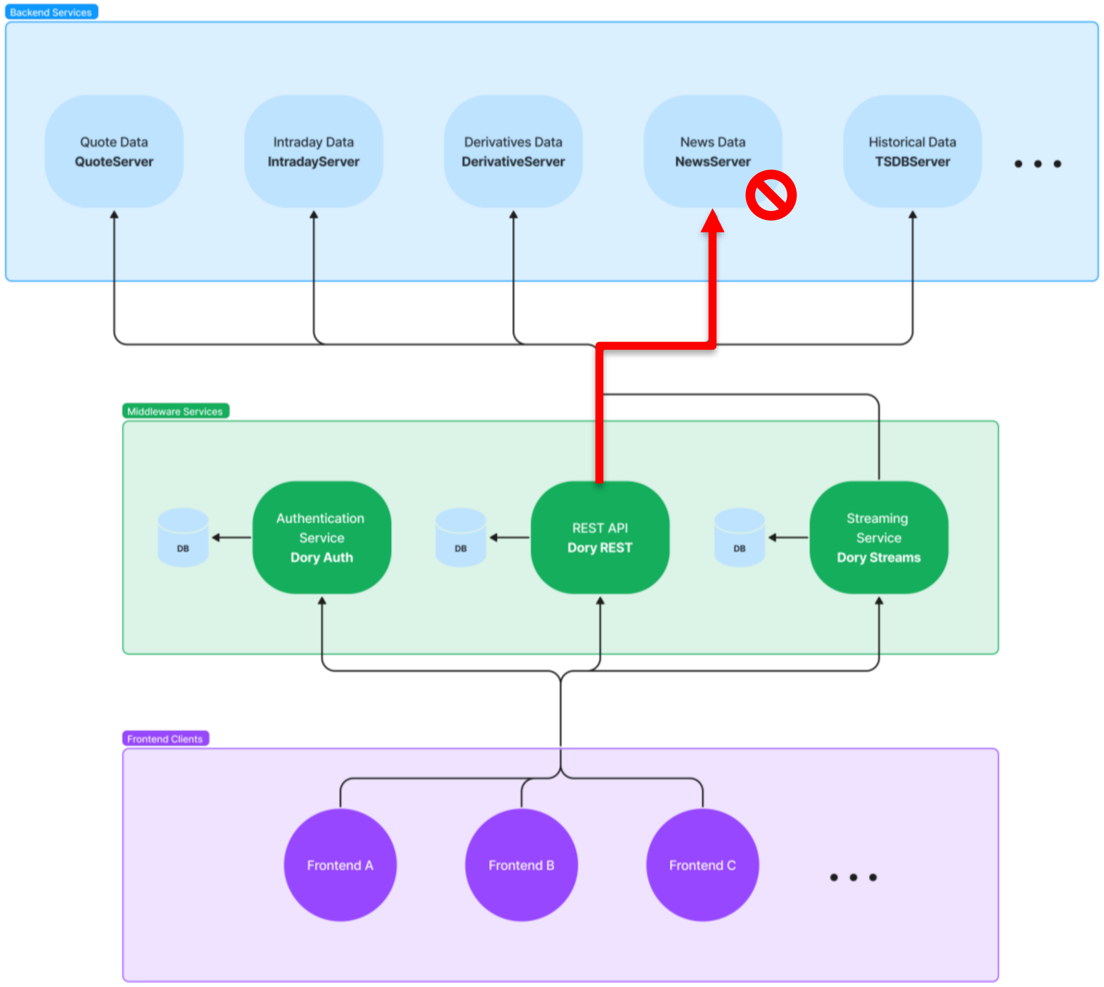
Problem Illustration



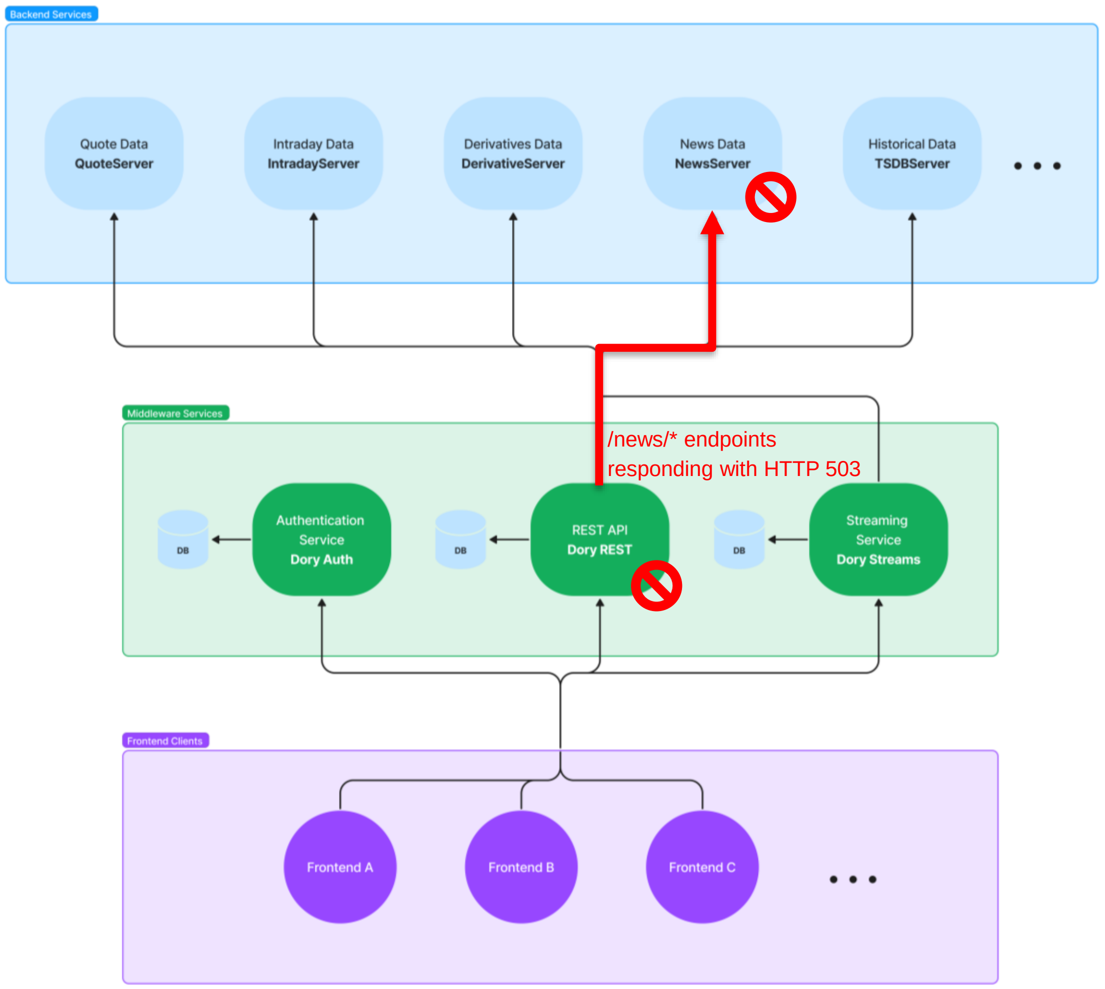
Problem Illustration



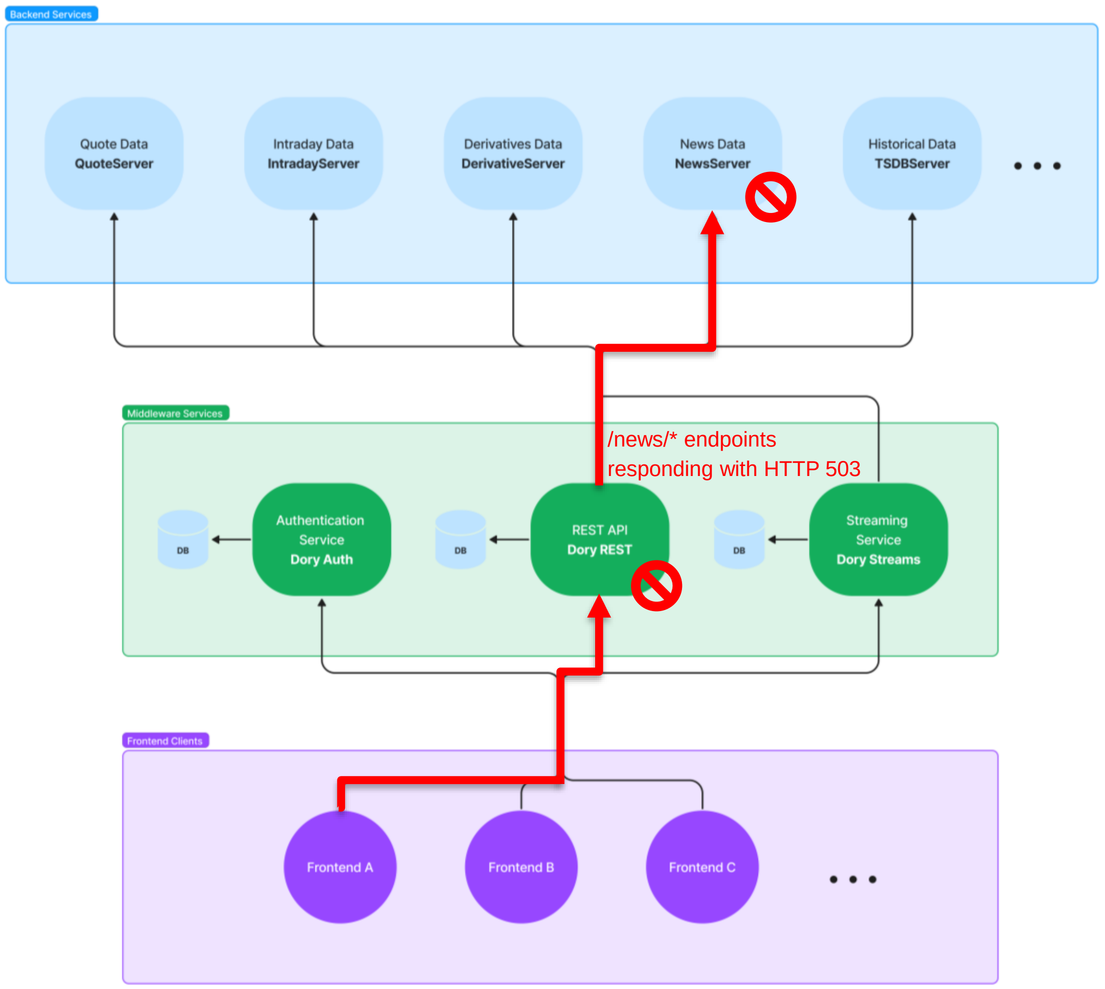
Problem Illustration


Backend Services
Quote Data QuoteServer
Intraday Data IntradayServer
Derivatives Data DerivativeServer
News Data NewsServer
Historical Data TSDBServer
Middleware Services
DB
Authentication Service Dory Auth
DB
REST API Dory REST
DB
Streaming Service Dory Streams
Frontend Clients
Frontend A
Frontend B
Frontend C

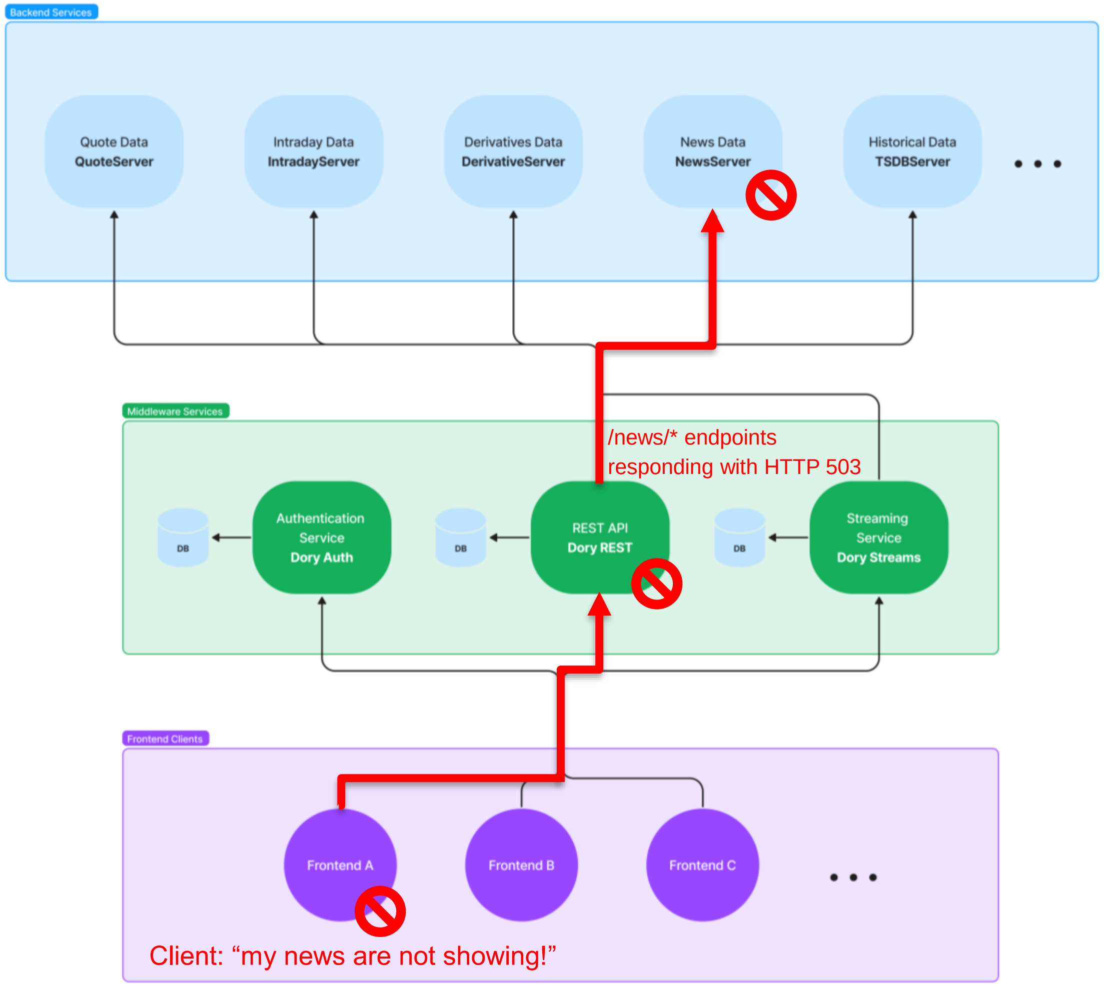
Problem Illustration



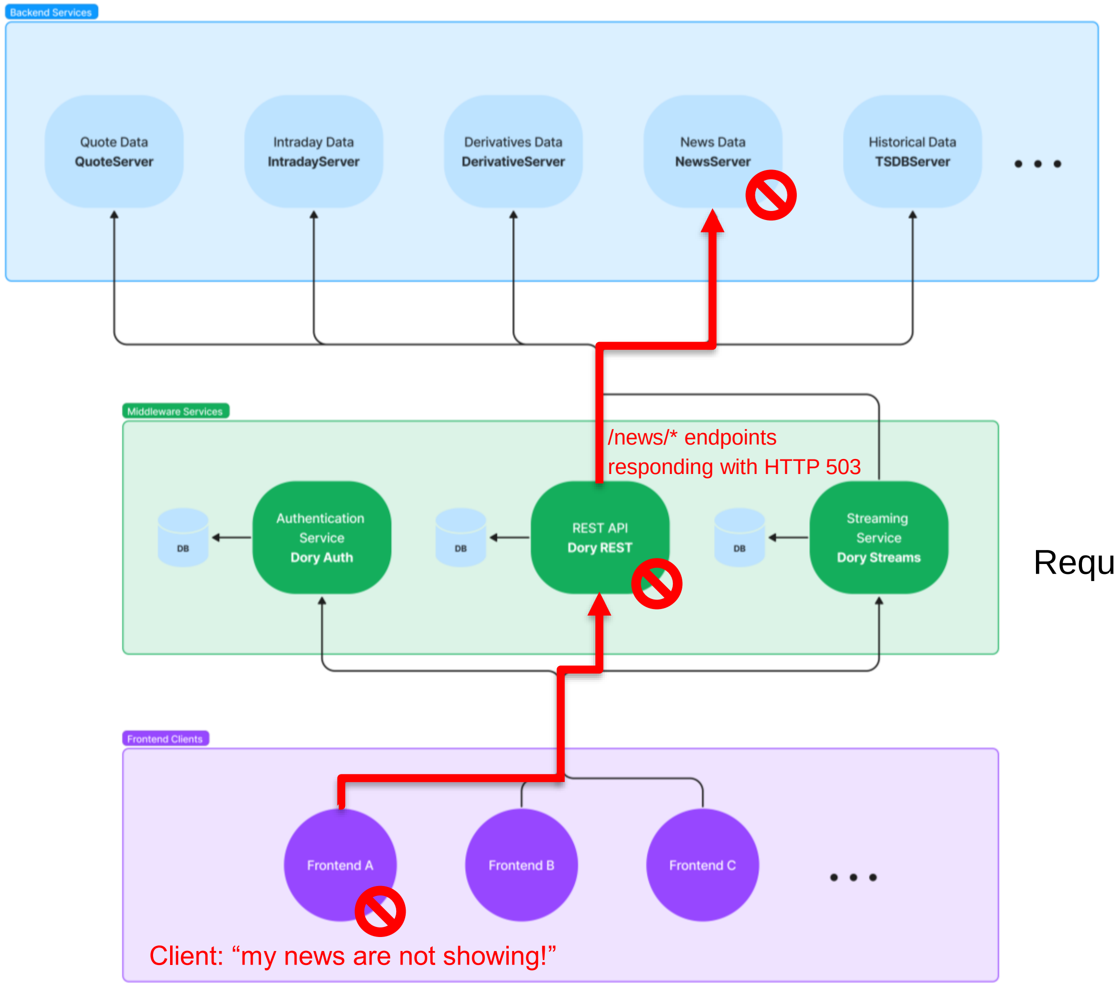
Problem Illustration



Problem Illustration


Backend Services
Quote Data QuoteServer
Intraday Data IntradayServer
Derivatives Data DerivativeServer
News Data NewsServer
Historical Data TSDBServer
Middleware Services
DB
Authentication Service Dory Auth
DB
REST API Dory REST
DB
Streaming Service Dory Streams
/news/* endpoints responding with HTTP 503
Frontend Clients
Frontend A
Frontend B
Frontend C
Client: “my news are not showing!”

Problem Illustration



Backend Services
Quote Data QuoteServer
Intraday Data IntradayServer
Derivatives Data DerivativeServer
News Data NewsServer
Historical Data TSDBServer
. . .
Middleware Services
/news/* endpoints responding with HTTP 503
Authentication Service Dory Auth
DB
REST API Dory REST
DB
Streaming Service Dory Streams
DB
Frontend Clients
Frontend A
Frontend B
Frontend C
. . .
Client: "my news are not showing!"

Difficulty in troubleshooting!

Requires lots of debugging efforts between different responsible teams.

Increased Debugging Efforts is the Main Problem at financial.com AG!

- First customer contact point: Hotline/Customer Service division!
- Hotline then inform the devs, usually first contacting the frontend teams.
- Ping-pong communications between the different teams responsible for different sets of microservices to troubleshoot the problem.
- Increased problem resolution time → decreased customer satisfaction!

Background

The Problem of Debugging Complexity in MSA can be Solved in Many Ways!

- Centralized logging efforts as currently being used at financial.com AG (i.e., Kibana — ELK Stack) and the advent of smart orchestration systems as Kubernetes do help, but the extent of their effectiveness is limited.
- Localize the fault by analyzing anomalous software behaviour [2],
- or, the one which has been enjoying increased popularity in the industry, *distributed tracing* [3].

Distributed Tracing

Distributed tracing essentially makes a distributed system more *observable* through the concept of *context propagation*.

In order to be observable, the microservices must be able to emit telemetry *signals* in the form of Spans.

The Spans are then correlated together *across* service boundaries, creating a *distributed trace* [4].

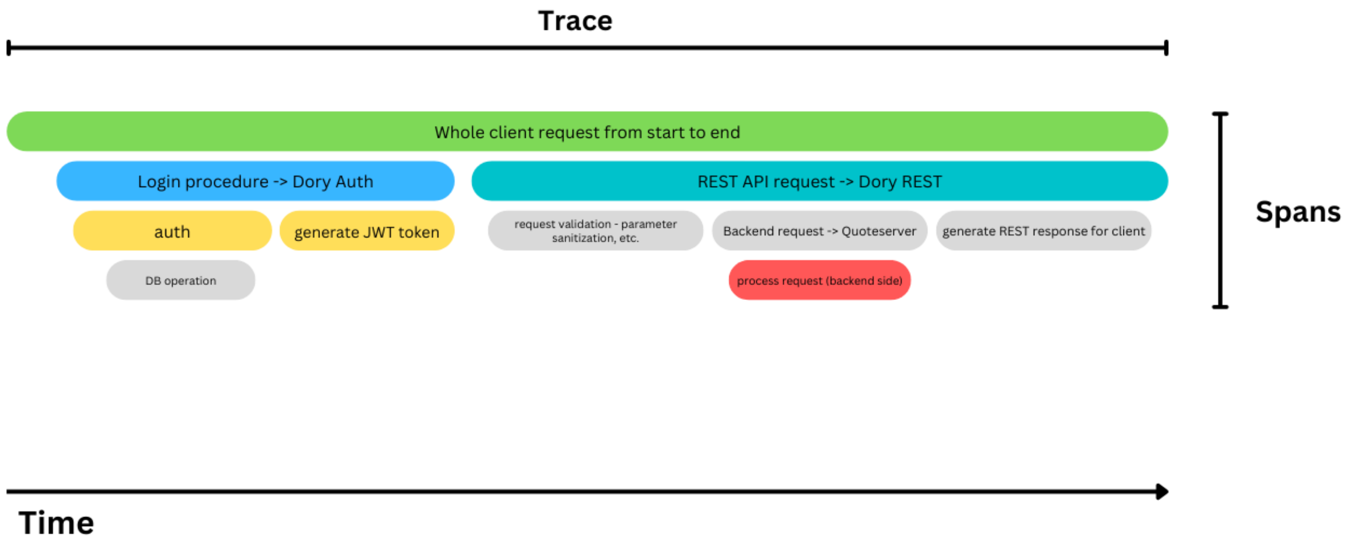

Trace
Whole client request from start to end
Login procedure -> Dory Auth
REST API request -> Dory REST
auth
generate JWT token
request validation - parameter sanitization, etc.
Backend request -> Quoteserver
generate REST response for client
DB operation
process request (backend side)
Spans
Time

Figure 4: Structure of a distributed *trace.*

Context Propagation

To have a complete *distributed trace* across a distributed system, the Spans from different services relating to a common *Context* have to be correlated together.

How? By propagating metadata information (the Context) across microservices!

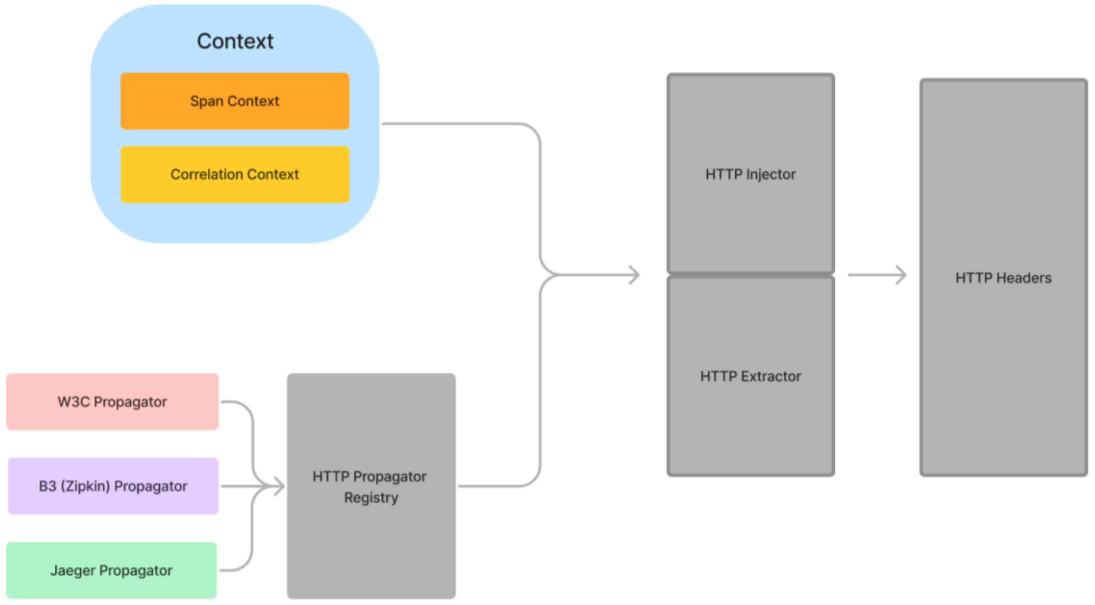


Figure 6: Example of Context propagation through HTTP header

Context Propagation

Note: Usually through embedding the information in a *carrier* object, e.g. the HTTP header (traceparent header value) from the W3C recommendation, but not always!

Basically any object which acts as a *carrier* can propagate the context to another services, depending on how they communicate (which protocols are being used?)

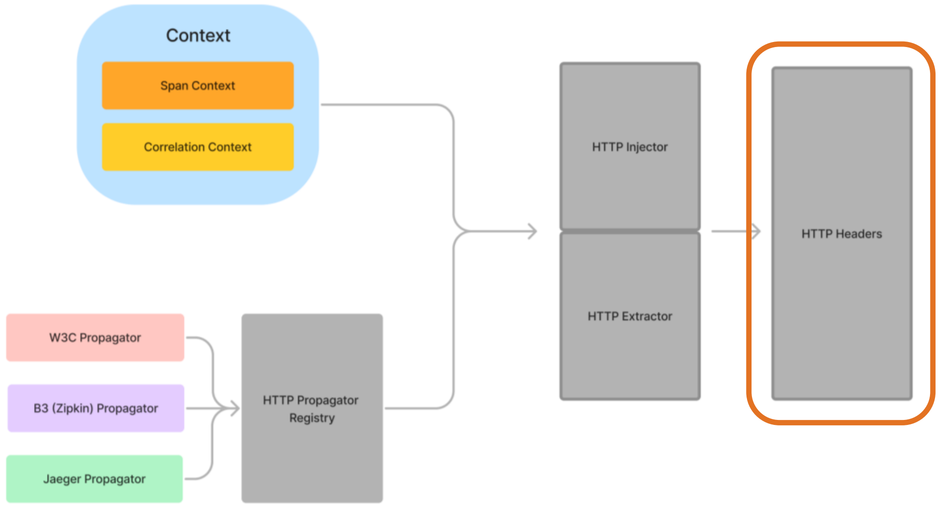


Figure 6: Example of Context propagation through HTTP header

Solution Approach

OpenTelemetry Framework

A Cloud Native Computing Foundation (CNCF) incubating project.

Founded in April 2019 after merger of two distributed tracing projects: *OpenTracing* (CNCF) and *OpenCensus* (Google Open Source).

- OpenTracing provides data format standardization through OpenTelemetry Protocol (OTLP), a vendor-neutral telemetry data API.
- OpenCensus provides language-specific libraries that developers need to start instrumenting their applications.

Instrumenting a Java Application

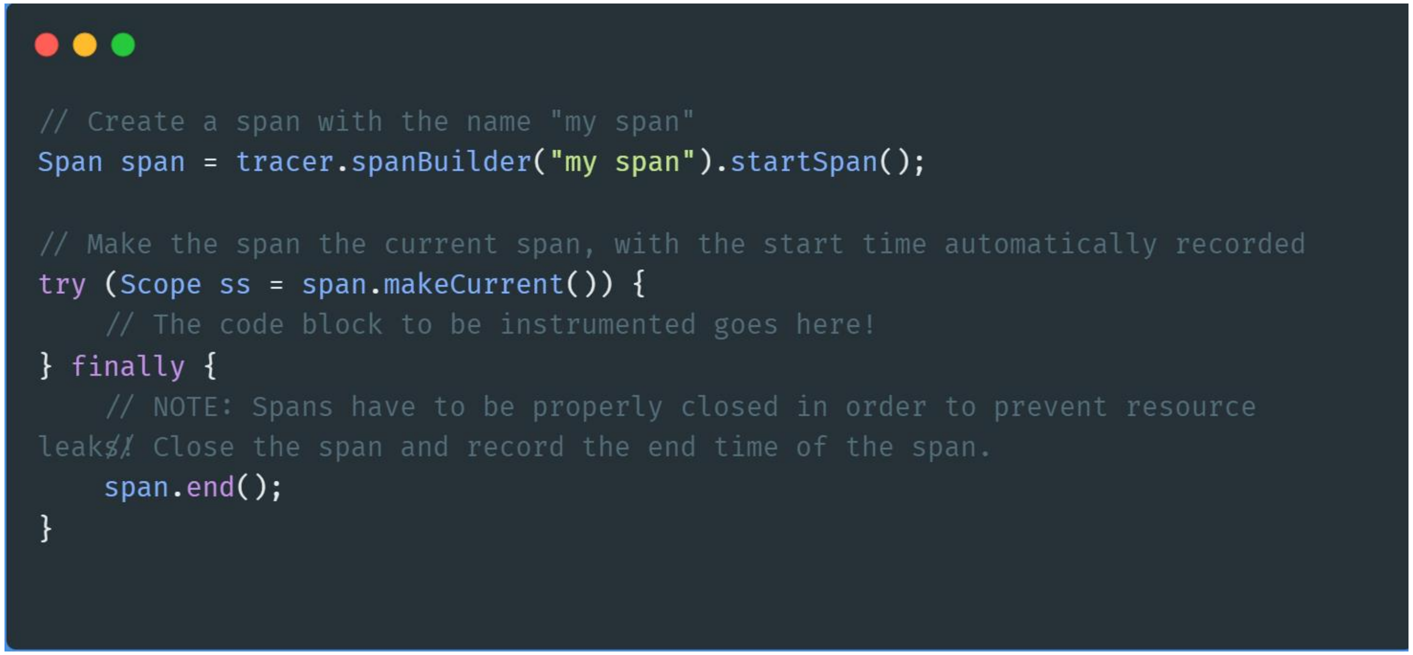
OpenTelemetry offers two ways of instrumenting a Java application:

- *Automatic* Instrumentation through the OpenTelemetry Java Agent — dynamically injecting bytecode in runtime to generate telemetry data, or,

Instrumenting a Java Application

OpenTelemetry offers two ways of instrumenting a Java application:

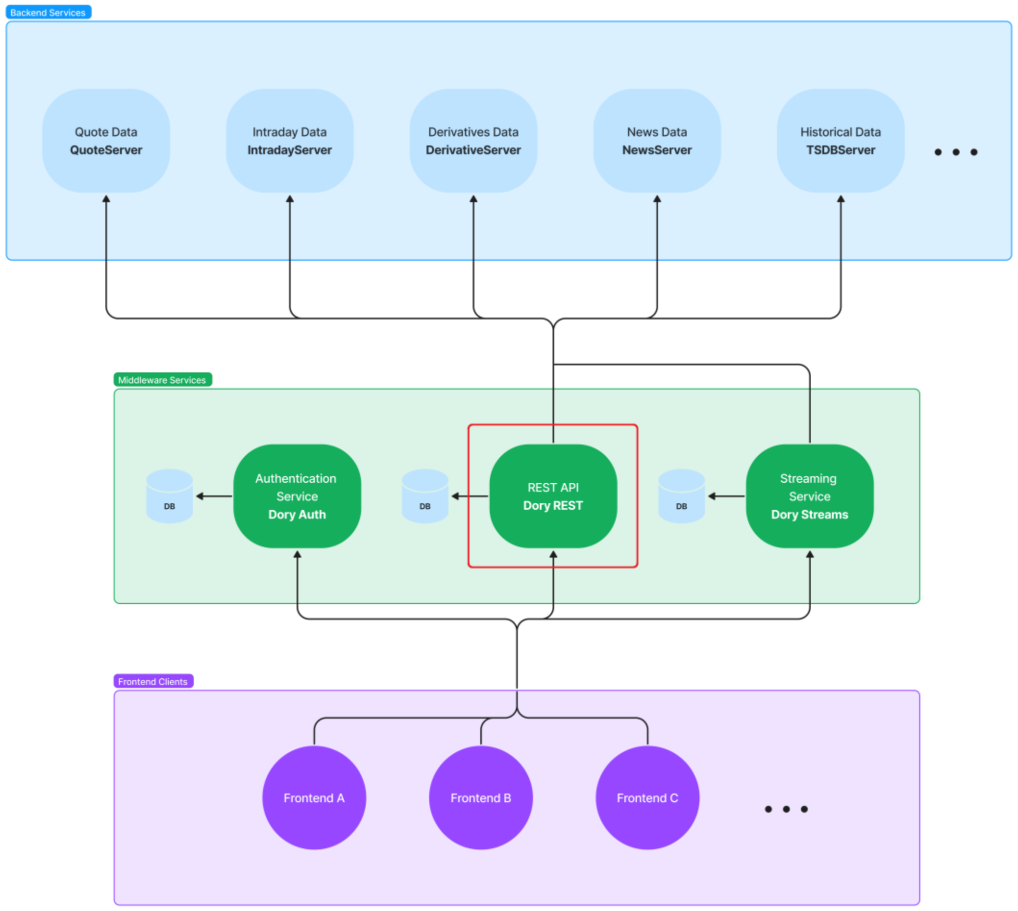
- *Automatic* Instrumentation through the OpenTelemetry Java Agent — dynamically injecting bytecode in runtime to generate telemetry data, or,
- *Manual* Instrumentation through the OpenTelemetry SDK and its APIs — requiring code changes!



```java
// Create a span with the name "my span"
Span span = tracer.spanBuilder("my span").startSpan();

// Make the span the current span, with the start time automatically recorded
try (Scope ss = span.makeCurrent()) {
    // The code block to be instrumented goes here!
} finally {
    // NOTE: Spans have to be properly closed in order to prevent resource
leaks// Close the span and record the end time of the span.
    span.end();
}
```

Main Focus: Instrumentation of the Dory REST Service



Design Requirements

- Automatic instrumentation dynamically injects bytecode at runtime, raising concerns of side effects — stability is important in production!
- We would also not have control over the telemetry signals emitted — *which parts of code should be/are interesting to be instrumented, which parts aren't?*
- As previously demonstrated, manual instrumentation requires modifications to the code — *where, why, and how* specific parts of the Dory REST code base have to be modified to achieve instrumentation?

Dory REST Architecture

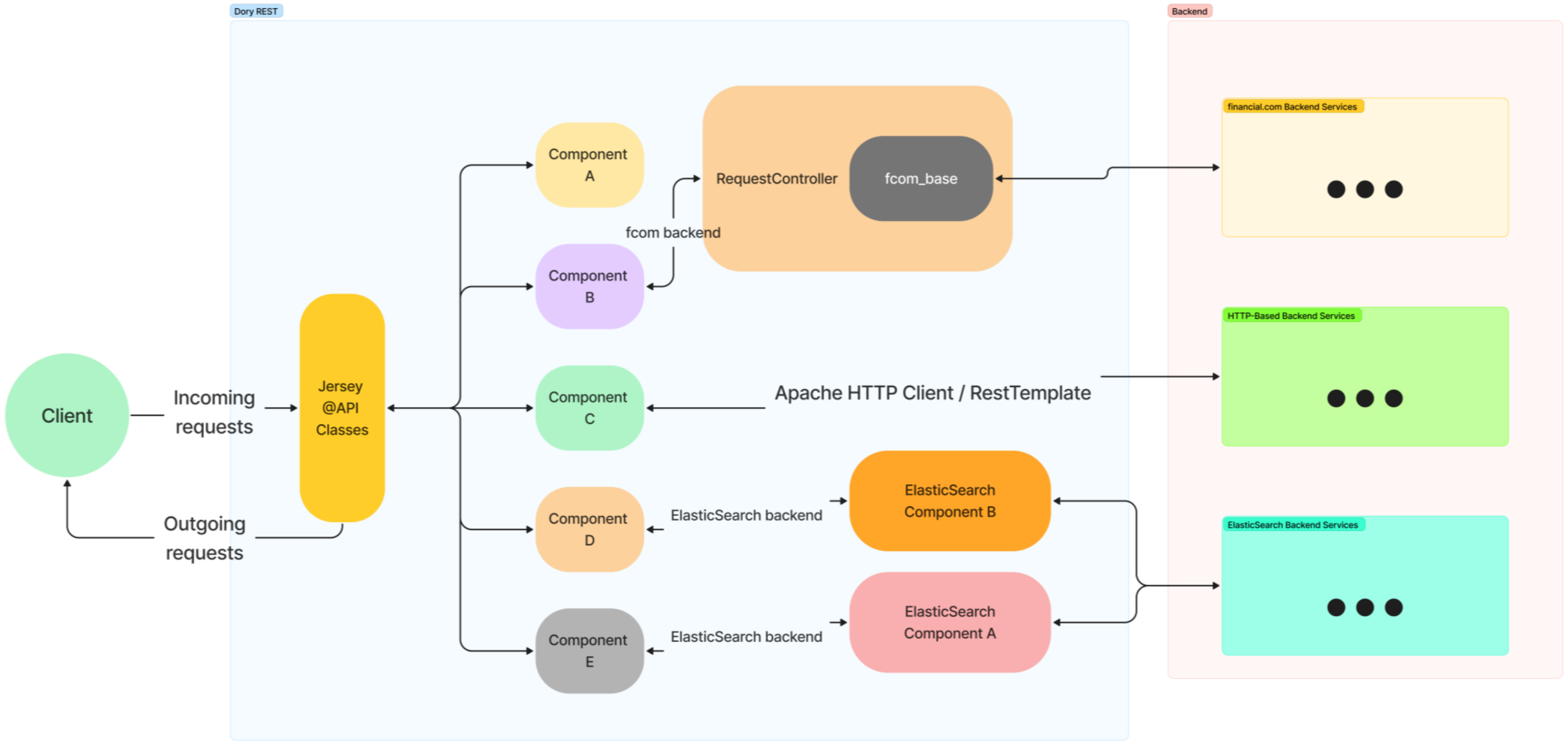


Figure 8: Dory REST architecture.

Dory REST Architecture

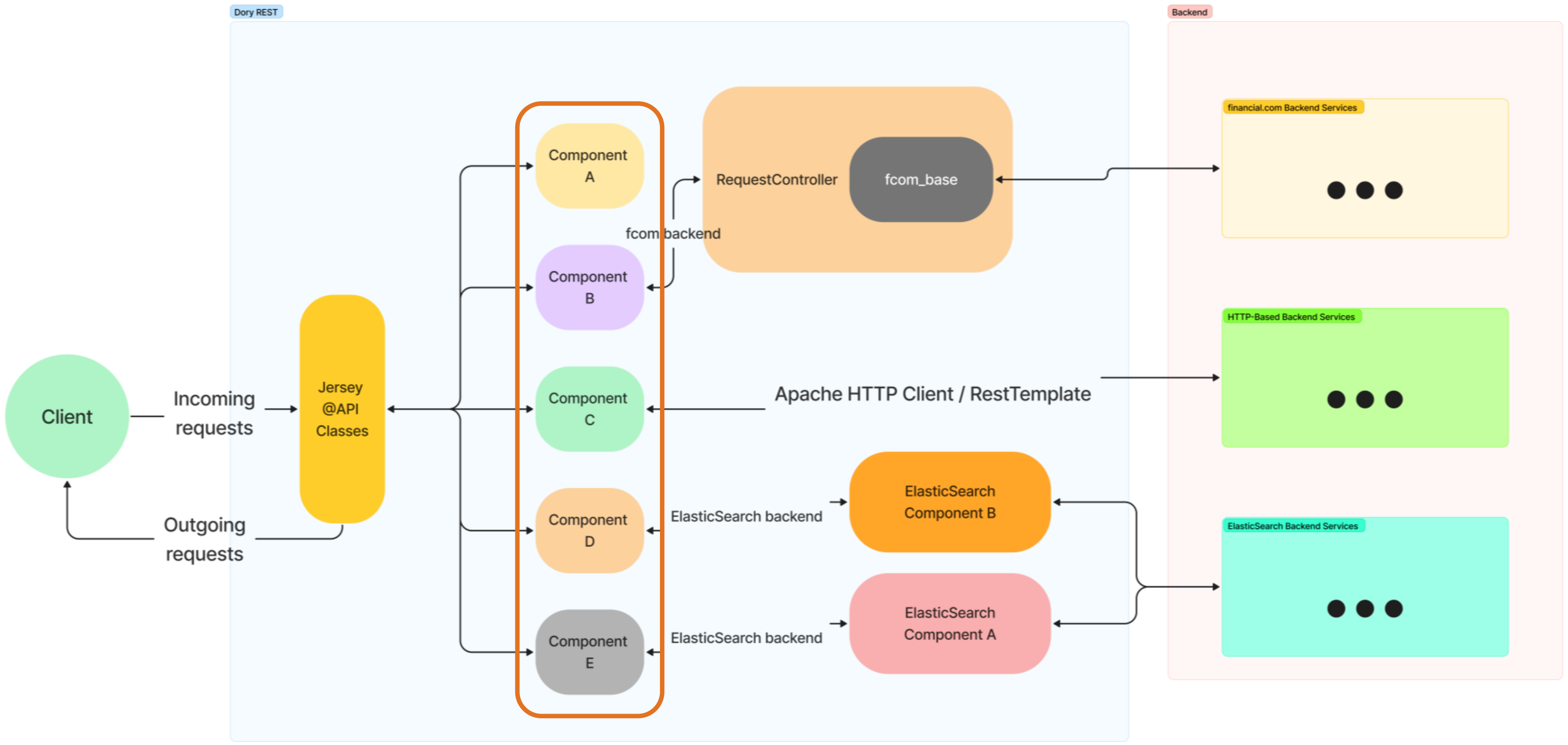


Figure 8: Dory REST architecture.

Instrumenting the Incoming Requests

Instrumenting the Components directly is inconvenient!

- must always remember to instrument the implementation of new components,
- repeated code patterns.

Better through the use of Java EE filters:

- encapsulates the entire request execution chain, until the sending of the response.
- provides a central place to instrument **all** incoming requests → no repeated codes!

Instrumenting the Incoming Requests

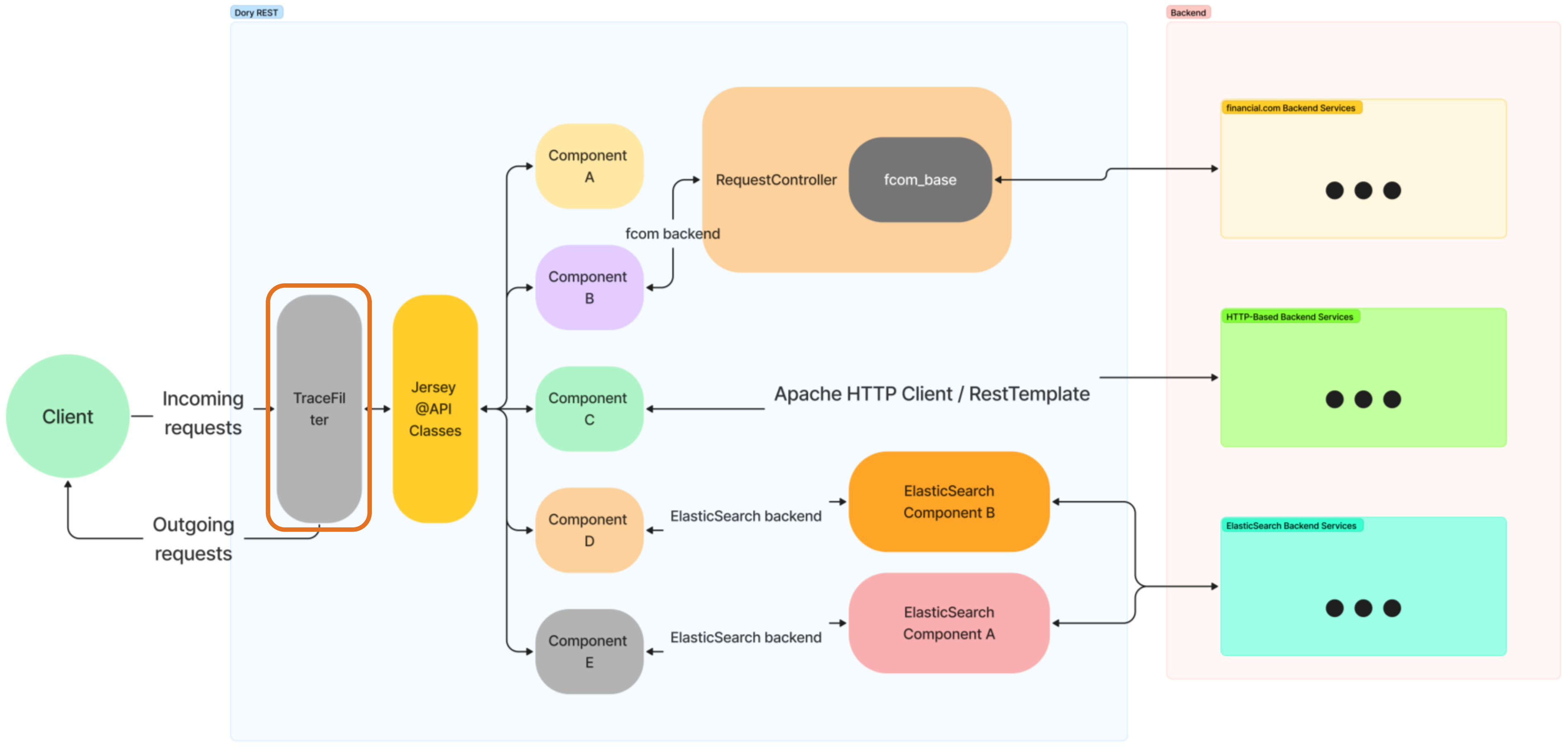


Figure 9: TraceFilter implementation.

Instrumenting the Outgoing Requests

Three broad types of outgoing requests from Dory REST:

1. to financial.com AG backend services through proprietary protocol based on sockets — through the fcom_base framework,
2. to generic HTTP backend services,
3. to ElasticSearch REST services.

Instrumentation of the Dory REST Service

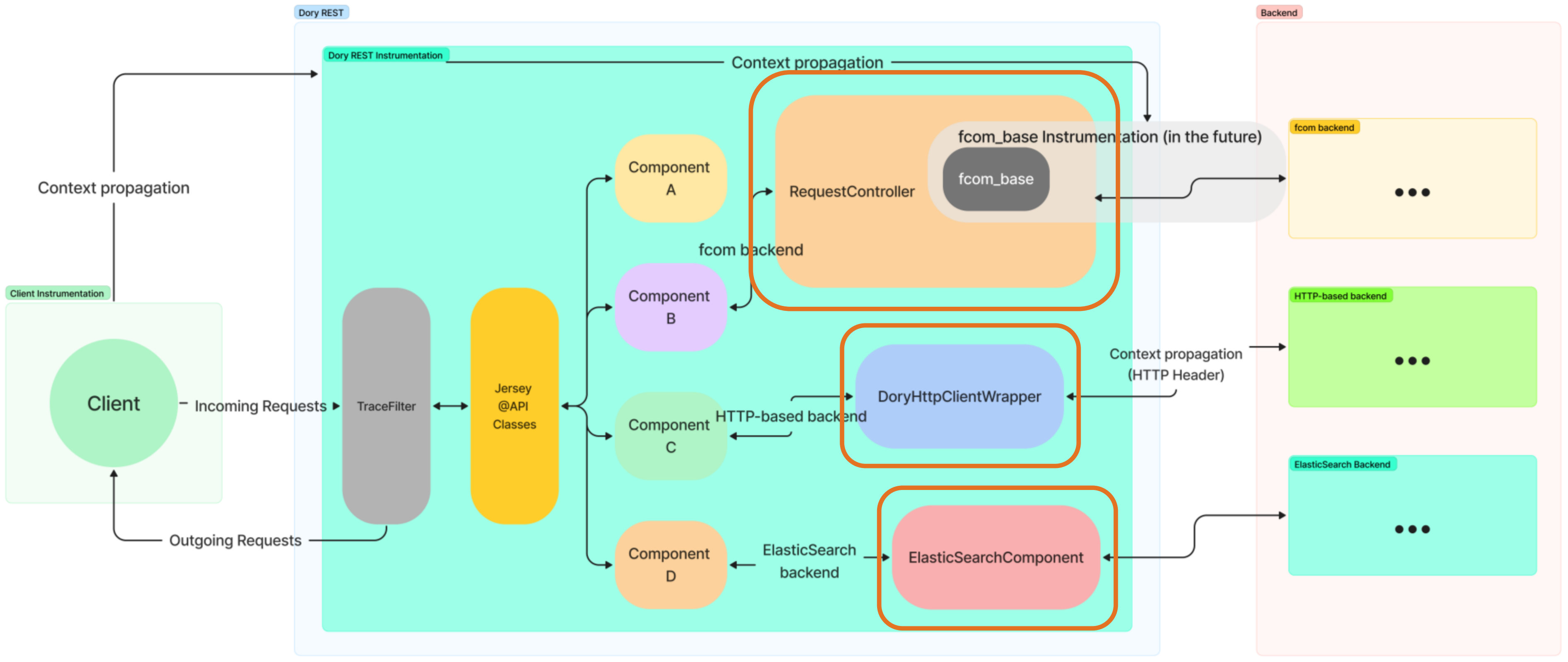


Figure 14: High level instrumentation of the Dory REST service.

Evaluation

Load Testing

Most important principle of the OpenTelemetry API:
→ Performance impact to the application being instrumented should be very minimal!

Nevertheless, we need to do initial performance testing before promoting this Proof-of-Concept to higher tier environments (e.g., staging, UAT, production), using previous load testing scenario for a client’s capacity planning.

Load Testing

We will be comparing these two versions of Dory REST:

- Dory REST with distributed tracing implementations, and the instrumentation features are enabled — configuration (1);
- Equivalent Dory REST version, *without* any distributed tracing implementations on the codebase — plain Dory REST version, configuration (2).

The initial load testing is organized in two phases:

- **Normal Load Scenario:**
  Testing typical normal load, comparing the response times of both configurations.
- **Limit Testing Scenario:**
  Testing the limits of the system, comparing the maximum amount of users/sec we can achieve in both configurations before requests start failing (over limit).

Normal Load Scenario Results

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 1: Mean response times for configuration (1).

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 2: Mean response times for configuration (2).

Limit Testing Results

Here the results are also effectively the same!

Configuration (1):
- Maximum users per second: 12;
- Requests per second: ~130.891 Req/s;
- Peak number of concurrent users: ~291 users;

Configuration (2):
- Maximum users per second: 12;
- Requests per second: ~130.891 Req/s;
- Peak number of concurrent users: ~273 users;

Conclusions

- Main problem of adopting MSA at financial.com: Increased troubleshooting complexity → longer problem resolution time → decreased customer satisfaction!
- Distributed tracing attempts to solve the abstract problem of increased debugging difficulty in MSA by making the distributed services itself more *observable* by the concept of Context propagation.
- Essentially propagating metadata (the *Context*) across service boundaries, usually embedded in the HTTP headers (*traceparent* — W3C), but doesn't always have to — any *carrier* object which supports embedding header values can work!
- In the context of this project, only manual instrumentation is used as there are concerns of side-effects/system stability.
- Initial performance testing results are promising, next step is to promote the Proof-of-Concept version to *staging* environment.

References

[1] C. Richardson. "Microservices | Pattern: Microservices Architecture." https://microservices.io/patterns/microservices.html (accessed 2022-07-05, 2022).
[2] 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," presented at the 2009 13th European Conference on Software Maintenance and Reengineering, 2009.
[3] I. Lightstep. "Lightstep | OpenTelemetry | What is Distributed Tracing?" https://lightstep.com/opentelemetry/tracing (accessed 2022-07-11, 2022).
[4] OpenTelemetry. "OpenTelemetry Docs | Observability Primer." https://opentelemetry.io/docs/concepts/observability-primer/ (accessed 2022-07-08, 2022).

Backup Slides

Demonstration

Live demonstration of the distributed tracing system on dev environment.

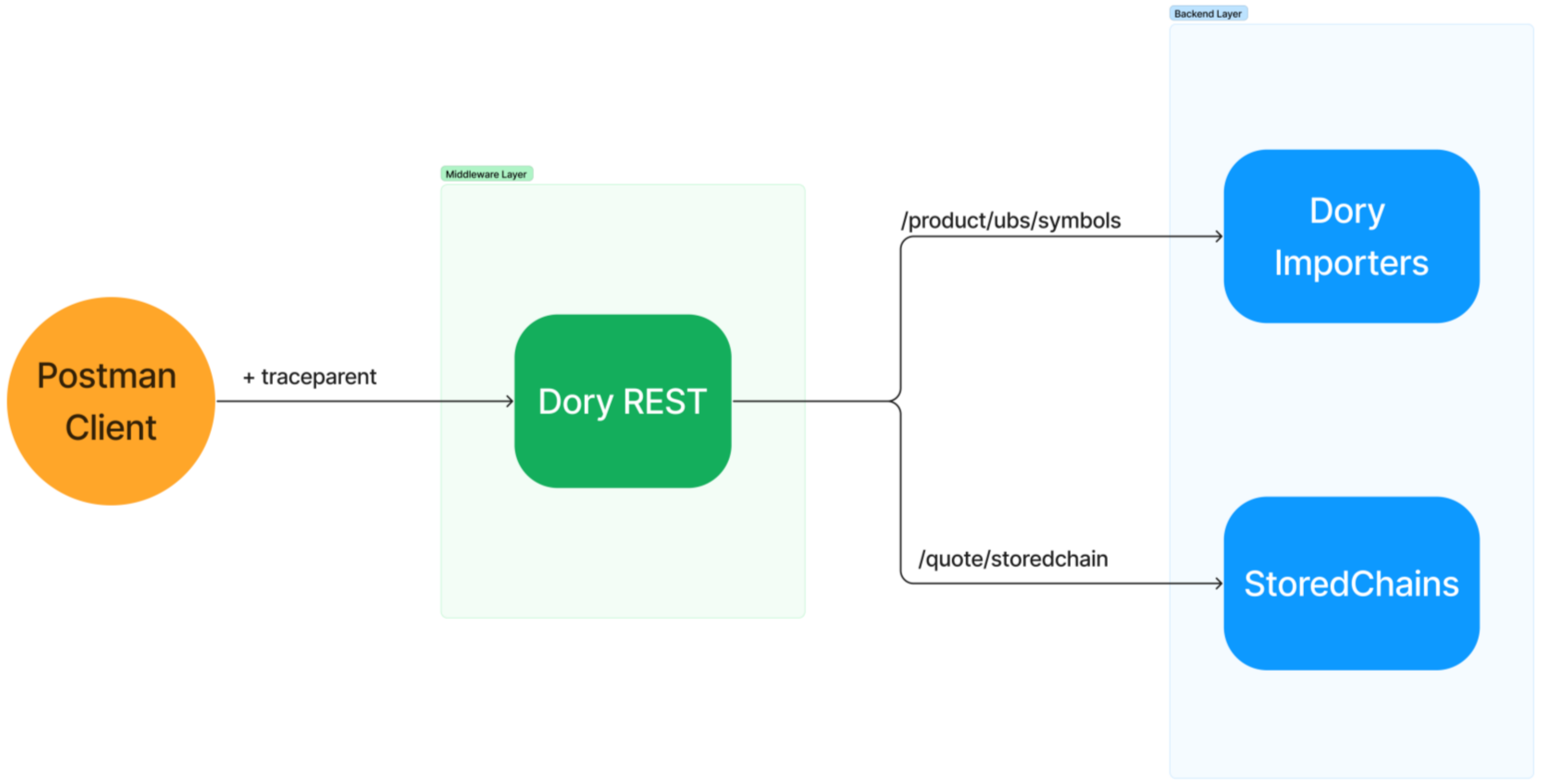


Figure 15: Demonstrating the Context propagation to the Dory Importers and StoredChains backend services.

Dory REST Architecture

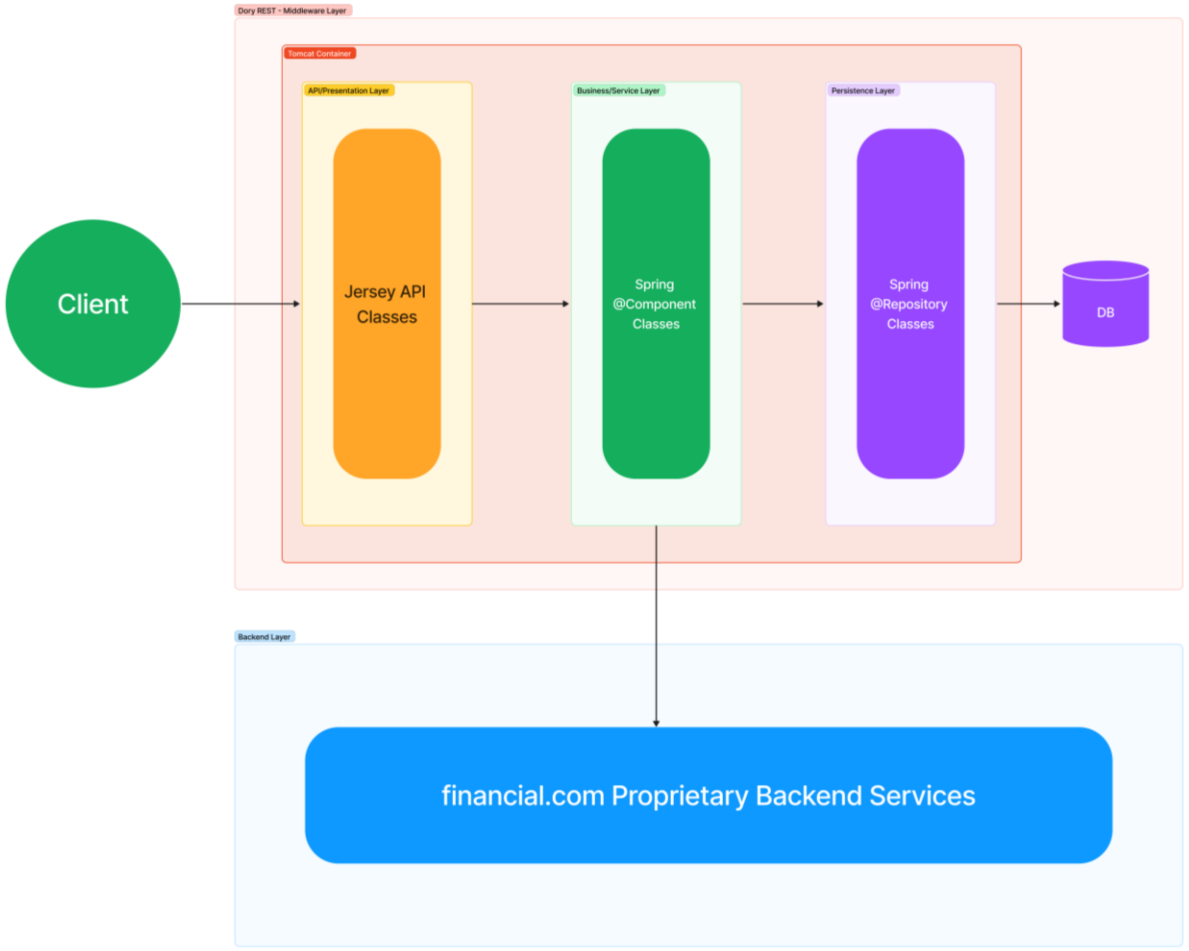

Dory REST - Middleware Layer
Tomcat Container
API/Presentation Layer
Business/Service Layer
Persistence Layer
Client
Jersey API Classes
Spring @Component Classes
Spring @Repository Classes
DB
Backend Layer
financial.com Proprietary Backend Services

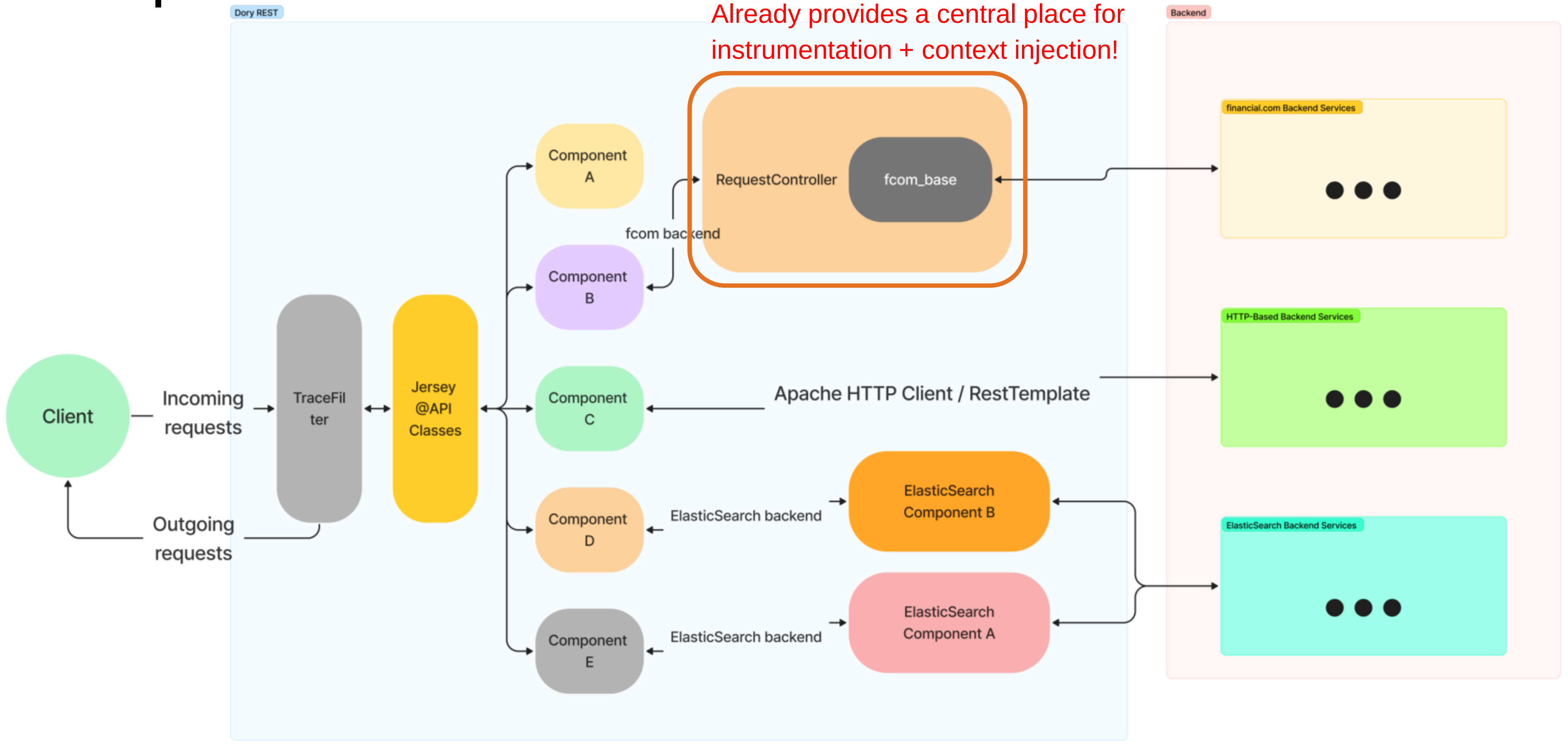
Figure 7: Dory REST high level architecture.

OpenTelemetry Framework

Distributed tracing essentially makes a distributed system more *observable* through the concept of *context propagation* (more on this later!)*.*

In order to be observable, the microservices have to be able to emit telemetry *signals.*

Instrumenting the financial.com Backend Requests



Dory REST
Already provides a central place for instrumentation + context injection!
Backend
financial.com Backend Services
RequestController
fcom_base
Component A
fcom backend
Component B
HTTP-Based Backend Services
Client
Incoming requests
TraceFilter
Jersey @API Classes
Component C
Apache HTTP Client / RestTemplate
ElasticSearch Component B
ElasticSearch Backend Services
Outgoing requests
Component D
ElasticSearch backend
ElasticSearch Component A
Component E
ElasticSearch backend

Figure 10: RequestController class for financial.com backend instrumentation.

Instrumenting the fcom Backend Requests

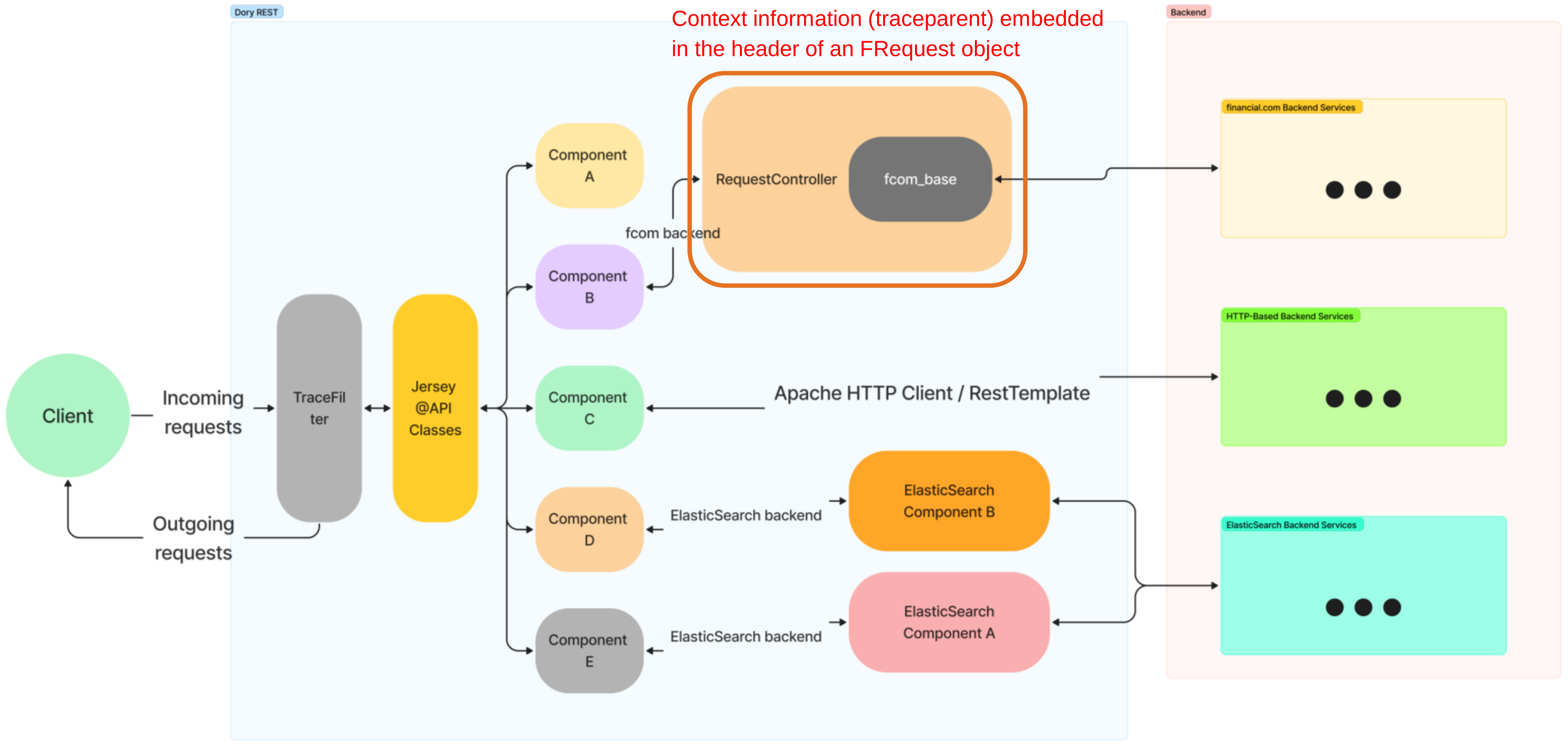

Dory REST
Context information (traceparent) embedded in the header of an FRequest object
Backend
financial.com Backend Services
RequestController
fcom_base
fcom backend
Component A
Component B
Component C
Component D
Component E
HTTP-Based Backend Services
Client
Incoming requests
TraceFilter
Jersey @API Classes
Apache HTTP Client / RestTemplate
ElasticSearch backend
ElasticSearch Component B
ElasticSearch backend
ElasticSearch Component A
ElasticSearch Backend Services
Outgoing requests

Figure 10: RequestController class for financial.com backend instrumentation.

Instrumenting the Generic HTTP Requests

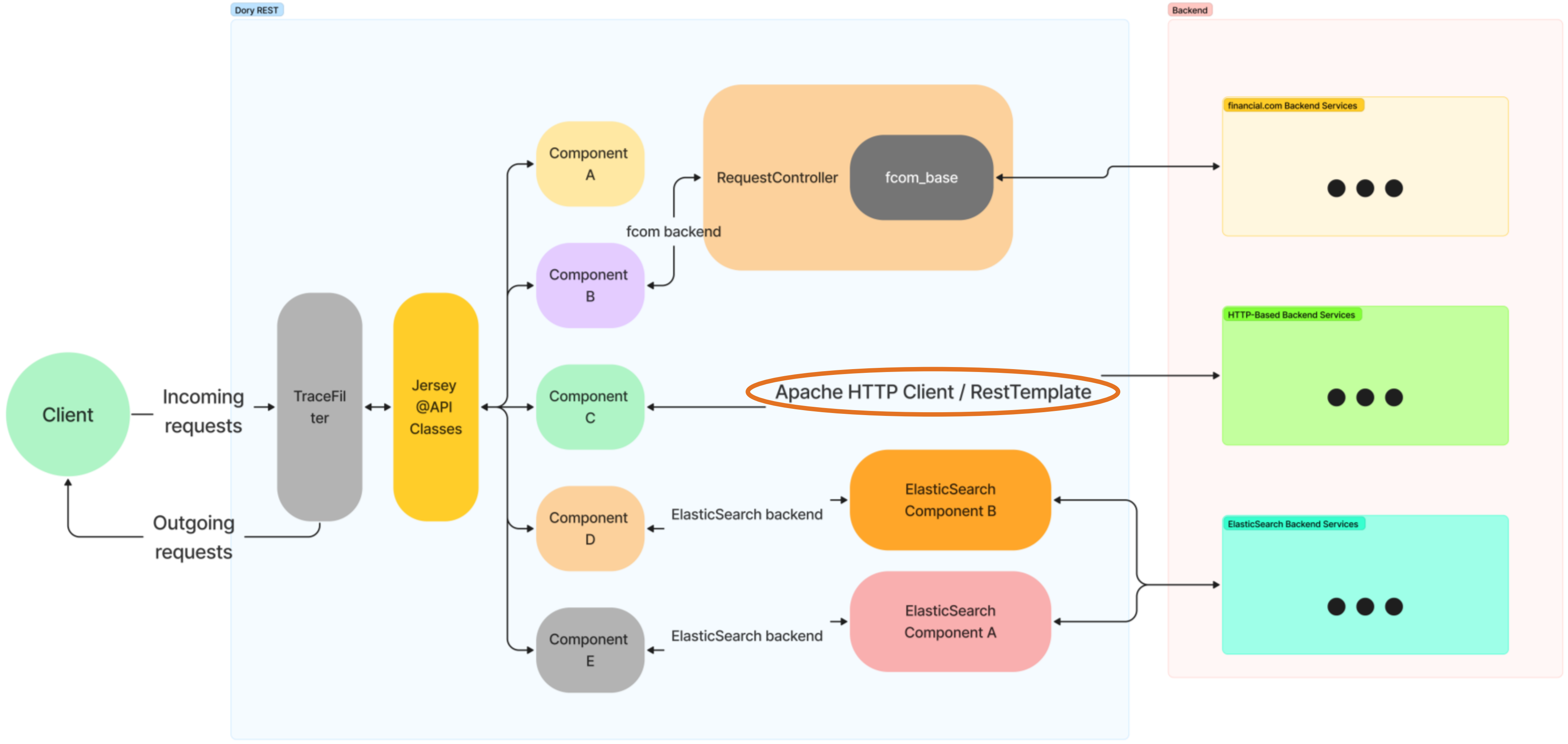


Figure 11: Instrumenting the outgoing HTTP requests

Instrumenting the Generic HTTP Requests

Problem: usage of HTTP client is not uniform across Dory REST!

Solution:

- Unify client usages, and encapsulate the use through a *wrapper* component/class,
- This wrapper class provides a centralized place to instrument the outgoing HTTP requests!

Instrumenting the Generic HTTP Requests

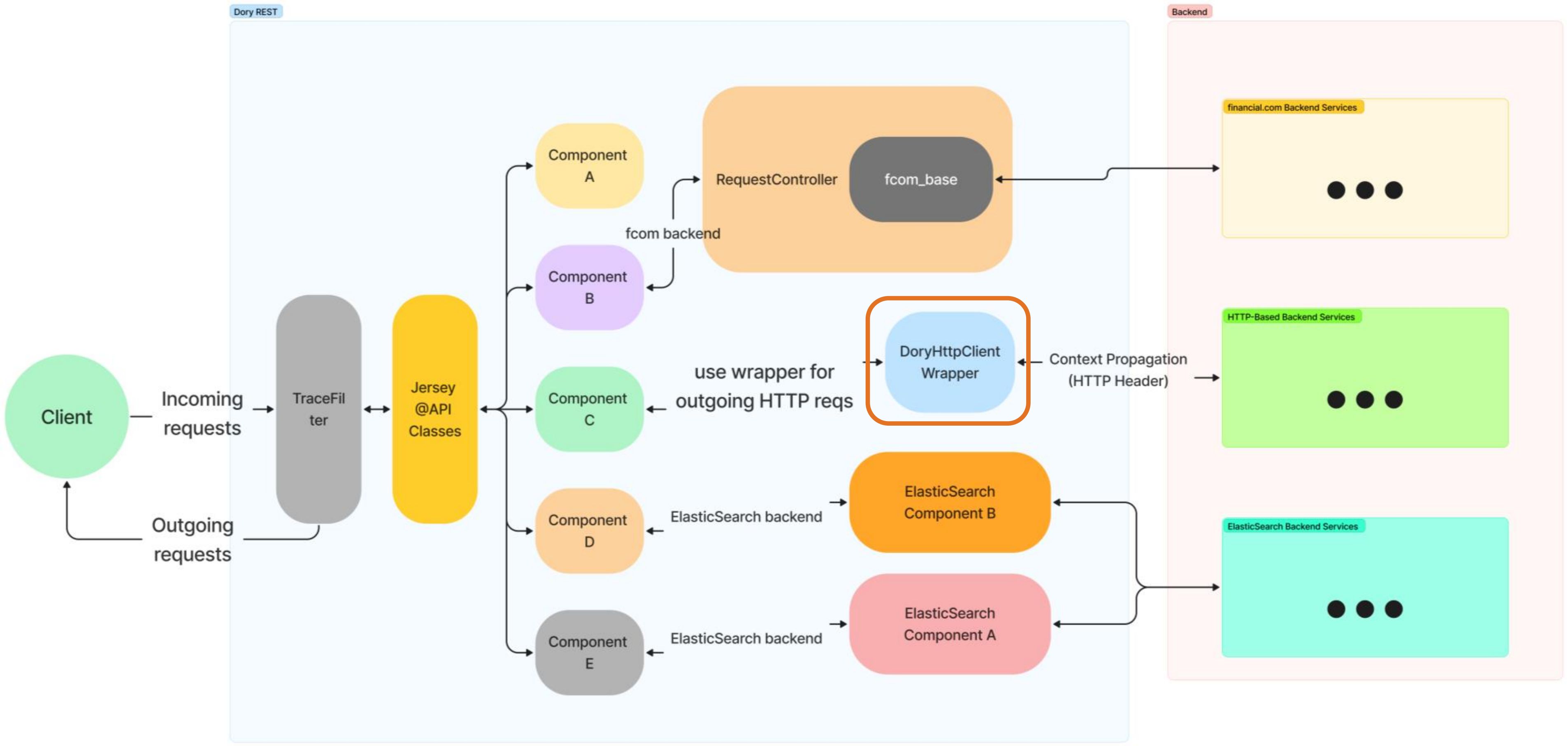


Figure 12: DoryHttpClientWrapper implementation.

Instrumenting the ElasticSearch Requests

- In Dory REST, ElasticSearch HTTP requests uses its own client implementation from Elastic — the RestHighLevelClient.
- Similar problem to above, multiple ElasticSearch component classes are implemented to tailor for the different client requirements!
  → consolidate in a central component to instrument the requests!

Instrumenting the ElasticSearch Requests

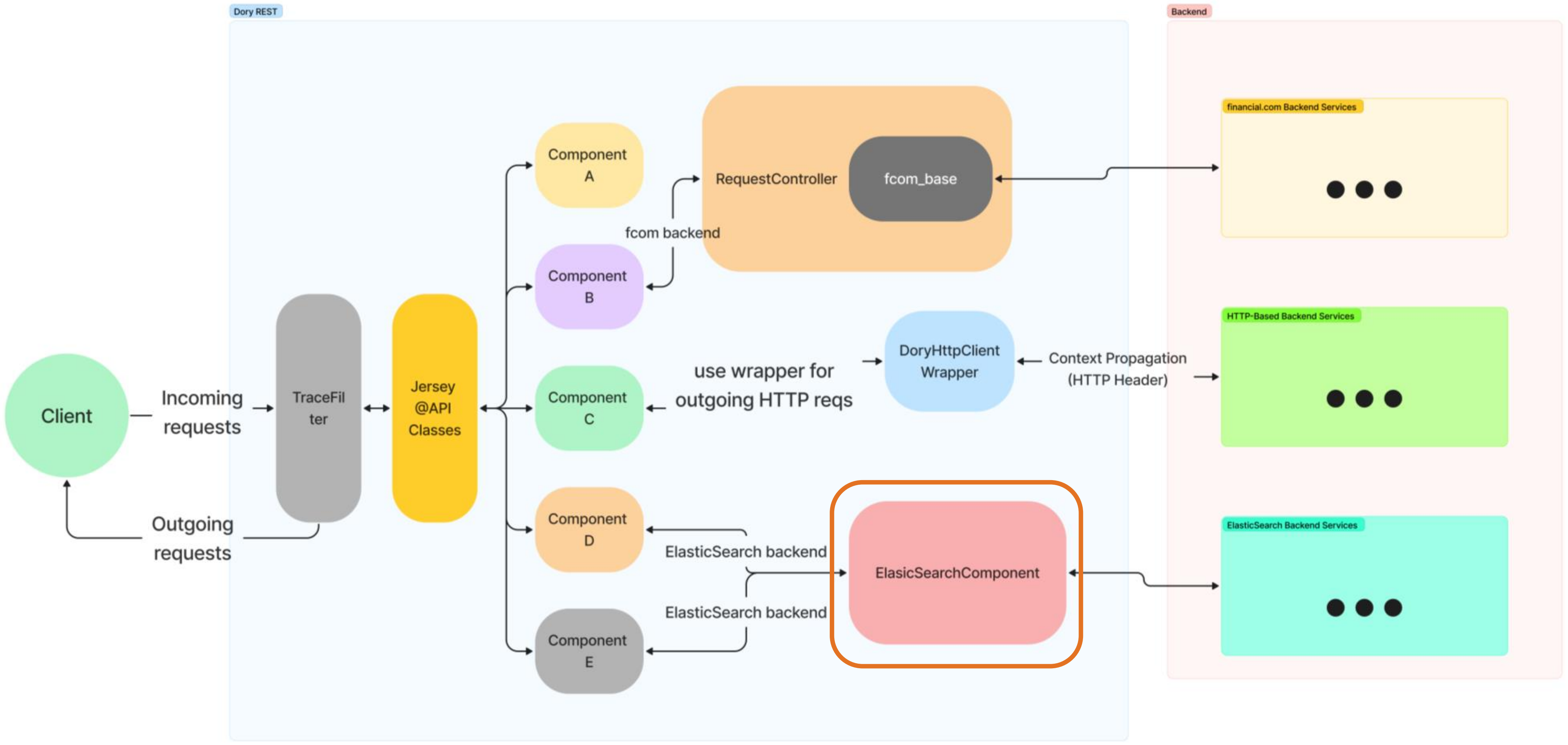


Figure 13: ElasticSearchComponent implementation.

Distributed Tracing

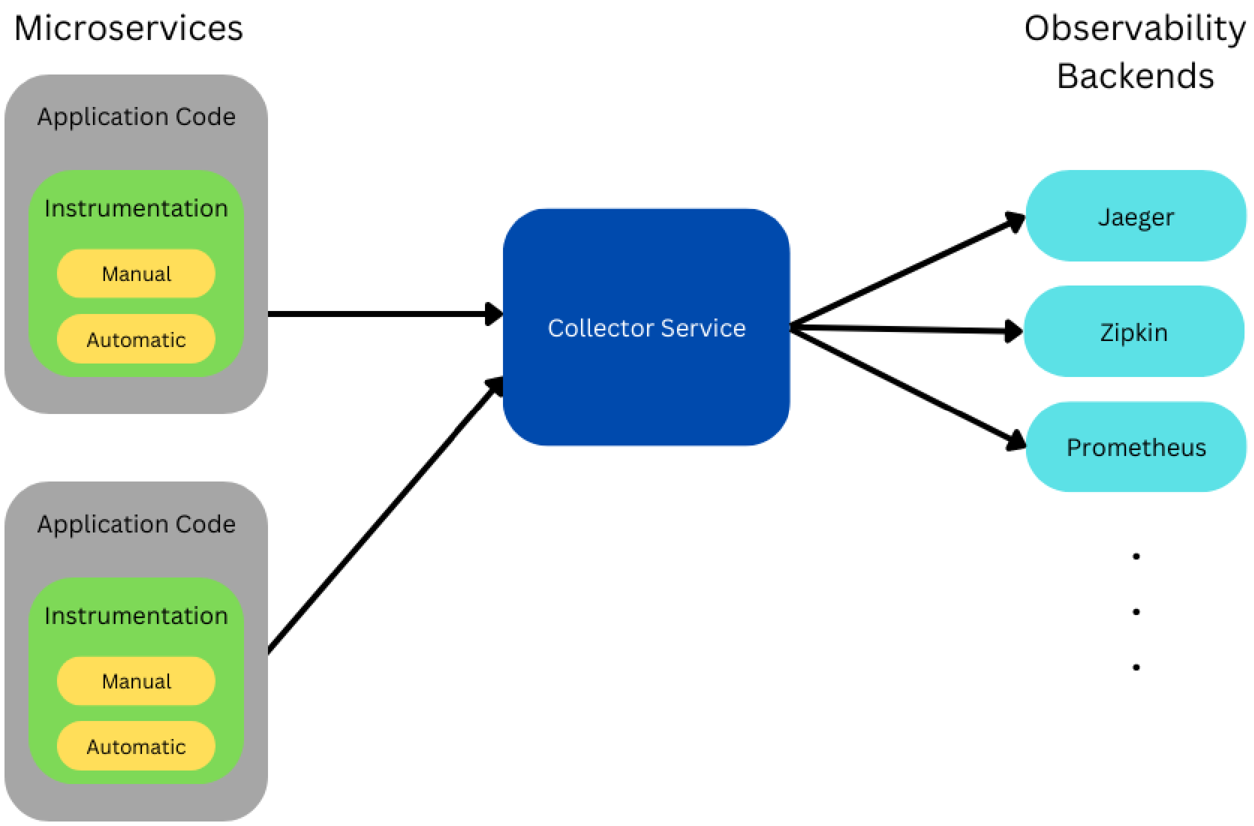


Figure 5: Distributed tracing infrastructure topology.

Load Test Suite

Consists of the following scenarios:

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

Normal Load Scenario Results

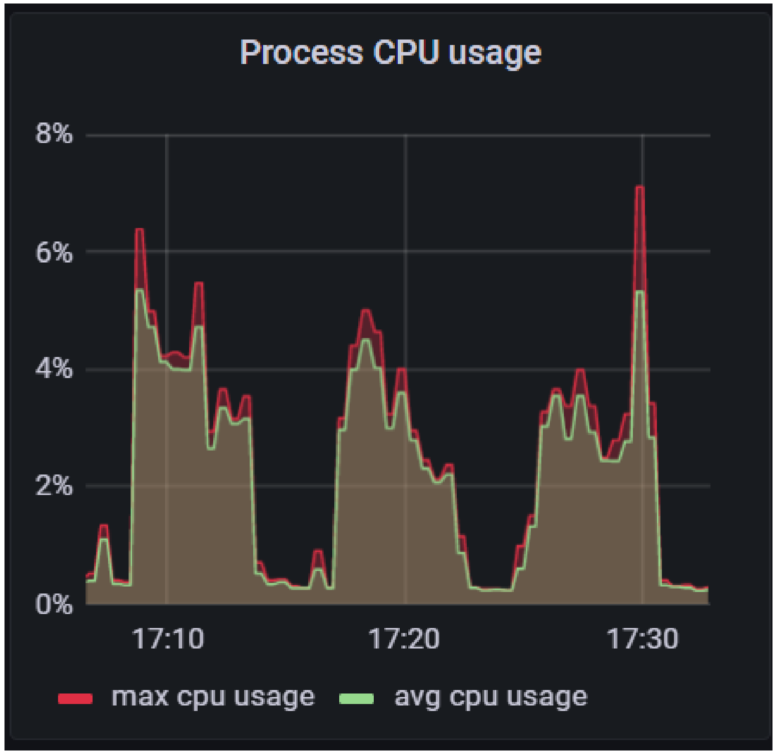


Figure 15: CPU usage of configuration (1) during the load tests.

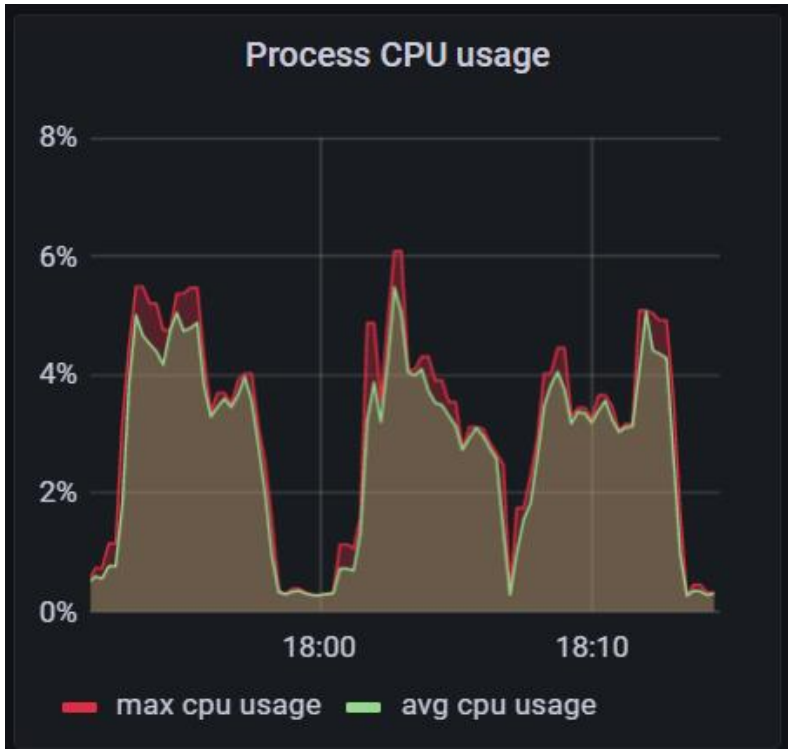


Figure 16: CPU usage of configuration (2) during the load tests.

Future Works

- Integrating the frontends into the distributed tracing system:

  Recall that at the start of this project the focus was mainly on the *middleware* layer (Dory REST) and the *backend* side (the Dory Importers and StoredChain services, also the financial.com backend services in the future through the instrumentation of the *fcom_base* framework).
  There are also plans in the future to also integrate the frontends into the distributed tracing system to have a full end-to-end view of the infrastructure.

Future Works

- Exploring the possibility of mixing automatic and manual instrumentation:

  There are a couple of benefits of using automatic instrumentation:
  - Lots of internal Spring libraries are already instrumented, for example eliminating the need to implement the instrumentation of incoming and outgoing HTTP requests as previously explained — with more information automatically embedded in the Spans.
  - JDBC calls can also be automatically instrumented.
  - Uniform attributes in the Spans as defined by the OpenTelemetry Semantic Conventions.

  Need to clarify on the concerns about side-effects first!

Future Works

- Instrumenting other types of outgoing calls from Dory REST:

  There are other outgoing calls from Dory REST which are not yet instrumented, e.g., JDBC or Redis calls.

  Need to evaluate first if the mix of automatic and manual instrumentation works well in practice (in this case we will have these calls instrumented already)!

Conclusions

- OpenTelemetry is the leading observability framework in the industry, CNCF project.#
- OpenTelemetry offers two way of instrumentation:
  - Automatic: through the OpenTelemetry Java Agent which dynamically injects bytecode in runtime to generate telemetry data;
  - Manual: through the OpenTelemetry SDK and its APIs which requires manual code changes!
- In the context of this project, only manual instrumentation as there are concerns of side-effects/system stability.
- Initial performance testing results are promising, next step is to promote the Proof-of-Concept version to *staging* environment.