

Automatic Instrumentation With OpenTelemetry for Software Visualization

An Evaluation for JavaScript Applications in the Context of
ExplorViz

Roman Hemens

Bachelor's Thesis
March 25, 2024

Software Engineering Group
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Wilhelm Hasselbring
Additional Advisor, M.Sc. Malte Hansen

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

Abstract

This thesis examines and evaluates the implementation of OpenTelemetry for automatic instrumentation within ExplorViz, a software visualization tool. Motivated by ExplorViz's limitation to Java applications, the study aims to expand its scope by integrating JavaScript data. Utilizing OpenTelemetry, the research generates, collects, and exports telemetry data like traces and metrics, which are essential for software system observation and monitoring. The investigation encompasses conceptualizing automatic instrumentation, its implementation into ExplorViz, and evaluating the resulting software visualization expansion. Findings suggest that OpenTelemetry's solution for instrumenting JavaScript lacks applicability and functionality in ExplorViz, particularly in the comprehensive visualization of instrumented traces. The work contributes to software engineering by demonstrating OpenTelemetry's automatic instrumentation of JavaScript, which, without further enhancements, is not suitable for software visualization in the context of ExplorViz.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Goals	2
1.2.1	Goal 1: Conceptualize and Implement the Automatic Instrumentation of OpenTelemetry	2
1.2.2	Goal 2: Implementation of JavaScript Instrumentation into ExplorViz	2
1.2.3	Goal 3: Conduct a Constructive Evaluation	2
1.3	Document Structure	3
2	Foundations and Technologies	5
2.1	OpenTelemetry	5
2.1.1	General Overview of OpenTelemetry	5
2.1.2	Automatic Instrumentation in OpenTelemetry	5
2.1.3	Traces in OpenTelemetry	5
2.1.4	Metrics in OpenTelemetry	6
2.1.5	The Collector in OpenTelemetry	6
2.2	ExplorViz	9
2.3	InfluxDB	12
3	Approach	15
3.1	Automatic Instrumentation	15
3.2	Implementation into ExplorViz	16
3.2.1	JavaScript Traces into ExplorViz	16
3.2.2	JavaScript Metrics into ExplorViz	17
4	Implementation	21
4.1	Implementation of Automatic Instrumentation	21
4.1.1	Instrumenting Traces of a NodeJS Application	21
4.1.2	Instrumenting Traces of a Web Application	23
4.1.3	Instrumenting Metrics of a NodeJS Application	25
4.1.4	Instrumenting Metrics of a Web Application	26
4.2	Implementation of Traces in ExplorViz	27
4.2.1	NodeJS Traces visualized in ExplorViz	27
4.2.2	Web Traces visualized in ExplorViz	30
4.3	Implementation of Metrics in ExplorViz	31
4.3.1	Metric-Service	32

Contents

4.3.2	Integration into ExplorViz’s Frontend	34
5	Evaluation	37
5.1	Structure and Procedure	37
5.2	Neutral Analysis of Responses	38
5.3	Evaluative Summary	40
5.4	Threats of Validity	42
5.4.1	Internal Validity	42
5.4.2	External Validity	43
5.4.3	Construct Validity	43
6	Related Work	45
7	Conclusions and Future Work	47
7.1	Conclusions	47
7.2	Future Work	48
	Bibliography	49

Introduction

When encountering the challenge of explaining the structure of software to somebody who has not been working on the project, it can turn out to be a difficult task. Therefore, the approach of visualizing software has emerged, which is easier said than done. What should be the appropriate design? What should be visualized, and what is rather distracting and unnecessary? ExplorViz is one possible solution to this question. [Hasselbring et al. 2020] It visualizes the underlying structure and communication lines of Java applications and frameworks. The first step is the automatic instrumentation of the code, which means observing the traces, for example, an operating software leaves when communicating between classes and packages. Automatic means, in this sense, that the original code of the software is not altered at all. ExplorViz processes the insights of such traces, gaining valuable architectural and connective information and visualizing them at multiple levels. [Fittkau et al. 2017] As mentioned before, ExplorViz works only for Java applications since the tool that instruments the required data is specialized in Java. Therefore, this thesis is occupied with evaluating another tool for automatic instrumentation named OpenTelemetry, which can instrument more languages. One language that is well documented and marked as stable for traces and metrics by the developers is JavaScript. [OpenTelemetry Contributors 2024] This thesis aims to answer whether the visualization of traces and metrics, automatically instrumented by OpenTelemetry, is sufficient for software visualization using ExplorViz's academic standards as guidelines.

1.1 Motivation

ExplorViz provides a research solution for visualizing software. It currently uses the tool InspectIT Ocelot to automatically gather the required data, which unfortunately provides this service only for the programming language Java and its corresponding frameworks. It has been suggested that the portfolio should be expanded by finding additional software that can automatically instrument traces and metrics from various languages and technologies. Here, OpenTelemetry might be one answer to the question since it is open source and contributes solutions for multiple languages. The overall goal is, therefore, to evaluate its usefulness for software visualization based on JavaScript and its different frameworks. ExplorViz represents visualizing software; its possible extension

1. Introduction

through OpenTelemetry motivates this work.

1.2 Goals

1.2.1 Goal 1: Conceptualize and Implement the Automatic Instrumentation of OpenTelemetry

The first goal is to understand the broad concepts of OpenTelemetry and ExplorViz (described more thoroughly in Chapter 2) and how automatically instrumenting a JavaScript application within a small environment works. The latter should be carried out practically to export trace and metric data to third-party backends successfully.

1.2.2 Goal 2: Implementation of JavaScript Instrumentation into ExplorViz

The implementation is focusing on one hand on the traces and the other on the metrics. The overall goal concentrates on successfully implementing and visualizing both in ExplorViz.

Goal 2.1: Implementing the Automatic Instrumentation of Traces

One sub-goal is the successful automatic instrumentation of traces from at least one scaled and complex JavaScript application. This should result in a visualization in ExplorViz, which can be evaluated.

Goal 2.2: Implementing the Automatic Instrumentation of Metrics

The other sub-goal is automatically instrumenting metrics from the same applications as in Goal 2.1. This also includes successfully implementing a new microservice into ExplorViz's software landscape, which processes metric data. This data should be displayed in the ExplorViz user interface as additional visualization.

1.2.3 Goal 3: Conduct a Constructive Evaluation

If all preceding goals have been achieved, the evaluation tries to answer whether reaching them improves ExplorViz. In this question, improving means adding a new feature while satisfying the requirement of better software comprehension through visualization. The objective of the analysis is to find a qualitative answer or at least a tendency with feedback on improvement. Nonetheless, this thesis evaluates OpenTelemetry as a tool for automatic instrumentation for software visualization, for which ExplorViz provides an applicable solution.

1.3 Document Structure

After explaining the fundamentals of the relevant technologies for the following chapters in Chapter 2, the approaches taken to achieve the defined goals from Section 1.2 are described in Chapter 3. Chapter 4 is concerned with their implementations, which results in reviewable visualizations of both traces and metrics. These are then evaluated, and their outcome is presented in Chapter 5 with an analysis that aims to answer the thesis's primary research question. Before concluding and pointing to future work in Chapter 7, related work to the topic is presented in Chapter 6.

Foundations and Technologies

2.1 OpenTelemetry

2.1.1 General Overview of OpenTelemetry

OpenTelemetry is a set of APIs, SDKs, tools, and integrations designed to create and manage telemetry data such as traces, metrics, and logs. It aims to provide a comprehensive toolkit for software developers and operators to instrument (manually and automatically), generate, collect, and export telemetry data. This enables observability and monitoring within software applications. This data primarily encompasses traces and metrics, offering an exhaustive portrayal of a software system's operational dynamics. [Thakur and Chandak 2022] OpenTelemetry is a Cloud Native Computing Foundation (CNCF) project with contributions from various companies and individuals, underscoring its community-driven approach to solving observability challenges in modern distributed systems. [OpenTelemetry Contributors 2024]

2.1.2 Automatic Instrumentation in OpenTelemetry

Automatic instrumentation in OpenTelemetry simplifies integrating observability into applications by automatically capturing telemetry data, such as metrics, logs, and traces, without requiring developers to insert instrumentation code manually. This feature is especially useful in complex systems where manual instrumentation would be time-consuming and prone to errors. Unlike manual instrumentation, which requires explicit code modifications to capture telemetry data, automatic instrumentation relies on pre-built libraries and frameworks to automatically instrument an application, significantly reducing the initial setup time and maintenance effort. [OpenTelemetry Contributors 2024]

2.1.3 Traces in OpenTelemetry

Tracing is a fundamental aspect of OpenTelemetry, providing insights into the behavior and performance of applications by recording the journey of requests as they traverse through the various components of a system. A trace consists of a series of spans, each representing a single operation or piece of work. Spans include metadata such as start and end times, logs, and attributes that provide context about the operation. OpenTelemetry

2. Foundations and Technologies

traces are designed to be lightweight and efficient to capture, allowing developers to gain visibility into request execution paths and identify performance bottlenecks and issues. [OpenTelemetry Contributors 2024]

2.1.4 Metrics in OpenTelemetry

Metrics in OpenTelemetry offer a way to quantitatively measure an application's behavior, operations, and health. Metrics data are numerical values collected over time intervals, providing aggregated statistical insights. This could include counters, gauges, histograms, and summaries, which are crucial for understanding the performance and reliability of applications. One single metric datum is called an event. This includes, but is not limited to, data on request counts, error rates, and resource utilization. OpenTelemetry provides a robust framework for capturing, processing, and exporting metrics data, enabling developers and operators to monitor system health, usage patterns, and operational efficiency. [OpenTelemetry Contributors 2024]

2.1.5 The Collector in OpenTelemetry

The OpenTelemetry Collector plays a pivotal role in the telemetry data pipeline, acting as an intermediary that receives, processes, and exports telemetry data. Designed to be vendor-neutral, the Collector supports receiving telemetry data from various sources, processing and transforming it as required, and then exporting it to multiple backends for analysis and observability. This modular and extensible approach allows for a centralized collection strategy that can scale with the needs of modern, distributed applications. [OpenTelemetry Contributors 2024]

The Collector's architecture revolves around a pipeline model consisting of receivers, processors, and exporters (see in Figure 2.1). Receivers accept data in various formats, processors apply batching, filtering, and enrichment transformations, and exporters send the processed data to specified backend systems. This pipeline model provides flexibility and customization, enabling users to tailor the data collection and exportation processes to meet their specific observability requirements. [OpenTelemetry Contributors 2024]

Usually, the OpenTelemetry SDK (which describes the instrumentation by the tool) sends traces and metrics to the collector, who exports them to a specific backend, leading him to be perceived as an agent between gathering and using the data. These operations require the OpenTelemetry protocol (short: OTLP), which acts as the encoding, transport, and delivery mechanism of telemetry data. The protocol uses gRPC and HTTP 1.1 transports while utilizing a Protocol Buffer (Protobuf) schema for serializing the data, resulting in better performance. While gRPC sends "exportService" requests, HTTP sends a POST request. The former's default port is 4317; the latter's is 4318. In both transportation ways,

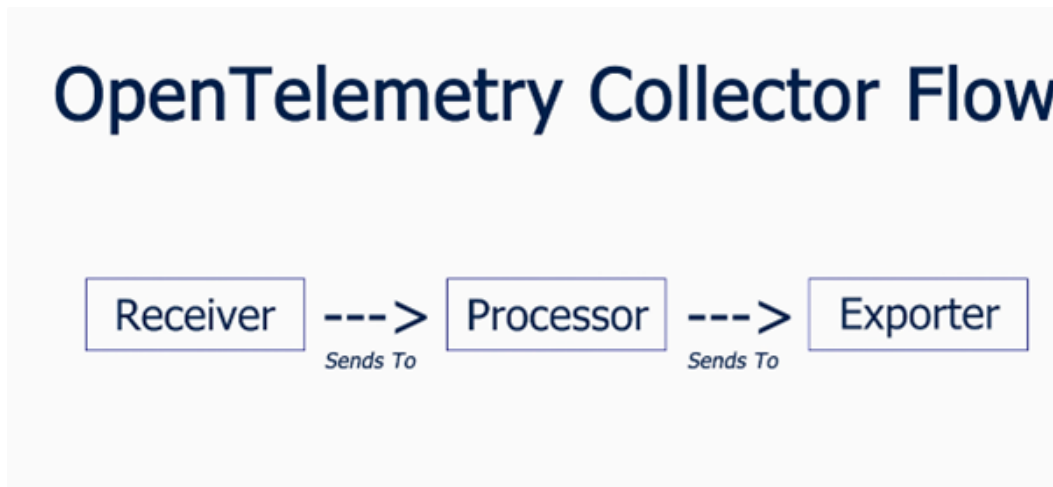


Figure 2.1. Data in an OpenTelemetry collector flows from the receiver over the processor to the exporter.

Protobuf is used, and therefore, it must be explained due to its relevance in this work, especially in terms of metrics.

Protocol Buffers (Protobuf) is a method developed by Google for serializing structured data, similar to XML or JSON. It's used for storing and exchanging data between systems or applications. Protobuf is designed to be simpler and more efficient than XML and JSON in terms of its format and the speed of serialization and deserialization. The process involves defining data structures (what fields are present and their data types) in a ".proto" file. [Protocol Buffers 2024] The Protobuf compiler then uses this file to generate source code in supported languages like Java, C++, Python, etc. This generated code provides APIs to serialize structured data to a binary format (for storage or transmission) and deserialize it back into a usable form. [Tilkov and Vinoski 2010]

As mentioned, a pipeline implements the collector flow from Figure 2.1, and OpenTelemetry provides opportunities to configure the single parts. A YAML file, usually called "collector-config", which needs to be mounted on the collector's Docker container, is an adjustment format. The structure of this file embraces the collector flow by using a service that implements the definition of each operation, resulting in the pipeline.

The type of receiver depends on the expected data's transport protocol and is mandatory to set up. If the default is addressed, the OpenTelemetry receiver does not require a configured endpoint, only allowing the used transport protocol (gRPC or HTTP). In this way, it is eligible for traces, metrics, and logs, like in Listing 2.1. Another receiver, like Prometheus, can only receive metrics from a Prometheus backend, yet others can only receive traces from their respective backend. [OpenTelemetry Contributors 2024] Further

2. Foundations and Technologies

Listing 2.1. Possible configuration file of the collector.

```
1 receivers:
2   otlp:
3     protocols:
4       grpc:
5       http:
6
7 processors:
8   batch:
9   attributes:
10    actions:
11     - key: environment
12       value: production
13       action: insert
14     - key: db.statement
15       action: delete
16
17 exporters:
18   otlp:
19     endpoint: otelcol:4317
20   otlphttp:
21     endpoint: otelcol:4318
22   prometheus:
23     endpoint: 0.0.0.0:9464
24   zipkin:
25     endpoint: http://zipkin:9411/api/v2/spans
26
27
28 extensions:
29   health_check:
30   pprof:
31   pages:
32
33 service:
34   extensions: [health_check, pprof, zpages]
35   pipelines:
36     Traces:
37       receivers: [otlp]
38       processors: [batch, attributes]
39       exporters: [otlp, zipkin]
40     metrics:
41       receivers: [otlp]
42       processors: [batch, attributes]
43       exporters: [otlphttp, prometheus]
```


configurations for this part are not necessary in this work.

In contrast, the processor is optional since using him means configuring the spans or events, which is not the default. However, it allows for bringing more individual structure to the whole process by combing the data into batches or using a memory limiter to prevent out-of-memory situations on the collector. Furthermore, the talked about enrichment happens by adding attributes to and deleting attributes from the span or the event as shown in Listing 2.1. [OpenTelemetry Contributors 2024] This mechanism becomes practical later in Section 4.2 and Section 4.3 when the instrumented data needs to be identified and assigned to the correct visualization.

The exporter is necessary, but multiple exports can coexist, and the pipeline can export the same data to different destinations. OpenTelemetry provides various exporters; next to otlp and otlphttp, Zipkin, and Prometheus are relevant for this thesis. The first two can send traces and metrics to configured services or other collectors using gRPC or HTTP and are therefore seen as customizable exporters. In contrast, the latter two send them specifically to their respective backend, which must also be hosted. This is useful for testing the instrumentation before integrating it into ExplorViz's architecture. Listing 2.1 displays all four possible exporters and what an example endpoint might look like. Additionally, the security settings can be configured via "TLS", which is necessary for otlp and otlphttp because otherwise, the exporter does not consider the destined backend trustworthy. [OpenTelemetry Contributors 2024]

As the name suggests, the extensions are not crucial, but they help to review some performance statistics and debug the code when the pipeline is not running as it should be. However, being the most important section, the service component implements the pipelines for traces and metrics as shown in Listing 2.1. The described configuration rules also apply to logs, but since they are not stable at the time, they are not included in the list. The collector must separate traces and metrics since it automatically treats both differently. The reason is that it accepts both at different addresses in the receiver section. Therefore, it does not transform instrumented metrics if the appropriate pipeline is not set up. [OpenTelemetry Contributors 2024]

2.2 ExplorViz

ExplorViz, an open-source software visualization and comprehension tool, stands out with its focus on dynamic analysis and live trace visualization of software landscapes. Its unique ability lies in its use of advanced visualization techniques to enable comprehensive and interactive exploration of software operations. This tool is particularly useful in the academic and research sectors, where it aids in exploring and comprehending complex software systems and architectures. It offers a detailed view of software landscapes, highlighting

2. Foundations and Technologies

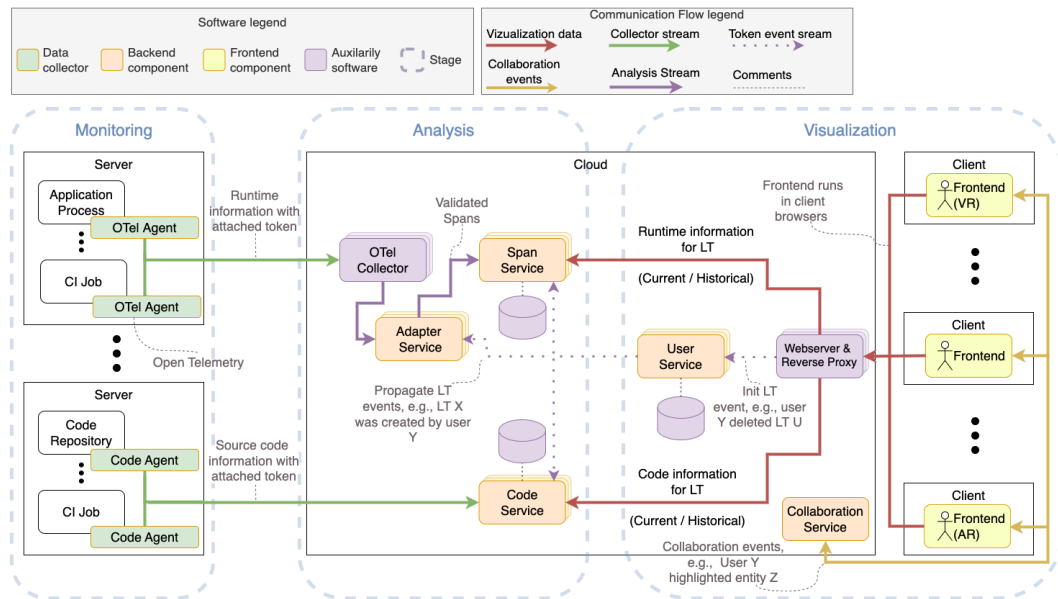


Figure 2.2. Software architecture of ExplorViz, describing the data flow from monitoring to analysis to visualization. Source: <https://explorviz.dev/3-architecture/> Accessed: [20.03.2024]

its utility for individual applications and broader software ecosystems. The collaborative capabilities of ExplorViz allow multiple users to interact and analyze software landscapes simultaneously. [Hasselbring et al. 2020]

ExplorViz’s primary mission is to address the challenges of managing and understanding large software landscapes. These landscapes often become complex due to architectural erosion, frequent modifications, and the integration of new systems. ExplorViz provides a solution by facilitating a deeper understanding of software structures and behaviors through immersive and interactive visualizations. It applies modern technologies such as AR and VR for collaborative analysis and exploration, enhancing the comprehension and management of complex software projects. [Hasselbring et al. 2020]

ExplorViz’s multi-level visualization approach is a standout feature. It offers a high-level overview of the entire software landscape and the intricacies of individual applications. This hierarchical abstraction is a practical feature that allows users to navigate complex software ecosystems efficiently. It aids in the identification of architectural patterns, dependencies, and potential issues, thereby enhancing the management and comprehension of software projects. The tool’s functionality is centered around live trace visualization, which captures and visualizes the runtime information of software systems. This approach helps monitor the software’s execution in real-time, providing immediate insights into its performance and behavior. [Fittkau et al. 2017]

One of the tool’s advantages is enhanced comprehension, which enables users to grasp

2.2. ExplorViz

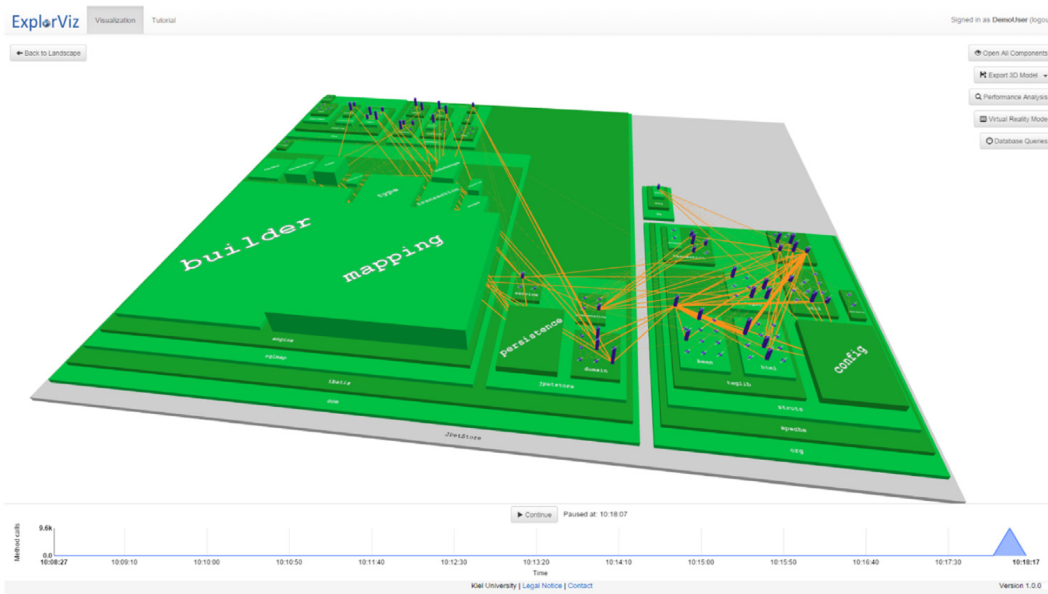


Figure 2.3. Example software visualization in ExplorViz showing the underlying structure and communication between the methods. Source: [Hasselbring et al. 2020]

complex software architectures and behaviors quickly through intuitive and interactive visualizations. Furthermore, collaborative exploration becomes possible supported by VR, making it a valuable tool for team-based projects. [Krause et al. 2018] However, this requires specialized hardware, which may not be readily available to all users. The focus lies on research and academic purposes and the allowance for customization, extension, and collaboration. [Fittkau et al. 2015]

In later chapters, ExplorViz’s software architecture, displayed in Figure 2.2, becomes somewhat relevant, and thus, a brief description of it is provided here. This work focuses mainly on monitoring, which means gathering traces and metrics from JavaScript applications utilizing OpenTelemetry. Enlarging the spans with custom attributes like a landscape token and token secret is critical since they make it possible to maintain the span’s identification to which visualized software landscape they belong. The frontend generates both attributes whenever a new software landscape is created in the user interface. ExplorViz receives traces with a configured OpenTelemetry collector via the gRPC transport protocol and passes them to a Kafka exporter. The adapter-service receives the spans transported by Kafka and deserializes them from Protobuf data structures to readable objects. Defining every span as a new object, with the remaining attributes being the only important ones for further use, is the second task of the adapter-service. Kafka transports these rewritten spans to the span-service. The span-service is responsible for detecting the underlying structure

2. Foundations and Technologies

and dynamic information all spans of an application together reveal, which are stored with the set landscape token and token secret as identifiers in a Cassandra database. When the user opens the software landscape in ExplorViz's user interface, the frontend requests the dynamic and structural data from the span-service via HTTP, including the time stamps, which display the temporal course of the spans when gathered. This leads to a visualization that might look like the example visualization of a Java application shown in Figure 2.3.

In conclusion, ExplorViz is a research and academic tool in software visualization and comprehension. Its innovative use of hierarchical visualizations and VR technology offers significant advantages in understanding and collaborating on complex software projects. This work aims to use ExplorViz as the guiding principle of software visualization when reviewing OpenTelemetry as an automatic instrumentation tool for this purpose.

2.3 InfluxDB

InfluxDB is an open-source time series database developed by InfluxData that handles high write and query loads. It is well-suited for operations monitoring application metrics, Internet of Things (IoT) sensor data, and real-time analytics. The core strength of InfluxDB lies in its ability to efficiently store and retrieve time series data, which is collected at and ordered by time. Written in Rust, it offers cross-platform compatibility and is licensed under MIT. [Kirešová et al. 2023]

InfluxDB uses an SQL-like language with built-in time-centric functions, making it straightforward for those familiar with SQL to query data. The data structure in InfluxDB consists of points, series, and measurements. Each point includes a timestamp and key-value pairs (field set), and points are indexed by time. A set of key-value pairs is known as the tag set. This design allows for efficient data retrieval based on time and tags. Moreover, InfluxDB supports data integrity through retention policies and continuous queries and has no external dependencies. [Nasar and Kausar 2019]

Setting up InfluxDB involves deciding whether to use the cloud version or an on-premise installation. The cloud version, InfluxDB Cloud, offers a fully managed service that automatically scales storage and compute resources to meet demand, allowing for a serverless experience. For those preferring or requiring data to reside on their infrastructure, InfluxDB offers an on-premises version known as InfluxDB Clustered, which is designed for enterprise-grade workloads. The cloud version is deployed in this work because the required data storage will not match enterprise levels and is easier to dispose of. Starting InfluxDB for the first time requires an account in the cloud and defining an organization and bucket. After that, the cloud interface returns a token, which needs to be saved immediately since it is only given once and cannot be retrieved later. Together with the URL, the organization, the bucket, and the token identify any operations when interacting with the database provided by the cloud service. [InfluxData 2024]

Using the line protocol format, data can be ingested into InfluxDB via its HTTP, TCP,

2.3. InfluxDB

or UDP interfaces. This protocol is simple and efficient, making sending data to InfluxDB from various sources easy. Once ingested, data can be queried using InfluxDB's SQL-like query language, Flux, which includes time-centric functions to extract meaningful insights from time series data easily. [Kirešová et al. 2023]

After all, InfluxDB's design as a time series database offers unique advantages for handling time-stamped data. Its SQL-like query language and powerful data ingestion and integration capabilities make it a robust time series data management solution. InfluxDB provides scalable, efficient, and flexible data storage and analysis tools for modern data needs, whether deployed on-premises or in the cloud. As in Section 3.2.2 explained in more detail, accomplishing Goal 2.2 requires a new service with the need for a database storing metrics for which InfluxDB meets the requirements.

Approach

As described in Section 1.2, both traces and metrics should be automatically instrumented and exported to the backend of ExplorViz. To illustrate this, example applications will be added. The initial phase involves exploring and experimenting with OpenTelemetry to gain familiarity with the tool and apply it using small applications. This part pertains to the potential challenges that may arise in automatically instrumenting JavaScript. After that, the approach of integrating it for traces and metrics into ExplorViz while scaling the applications to a more complex state is presented. This chapter aims to clearly state the idea behind the first two goals presented in Section 1.2.

3.1 Automatic Instrumentation

While this part is concerned with first understanding the basics and key elements, it focuses on implementing automatic instrumentation with JavaScript for small examples while closely following the documentation of OpenTelemetry. The documentation splits the automatic instrumentation of JavaScript into NodeJS and browser context, and therefore, this work does the same, resulting in two example applications. [OpenTelemetry Contributors 2024] It is necessary to consider not only the instrumentation but also the integration of the collector logic introduced in Section 2.1, which does not differ between both applications. While later the traces and the metrics are planned to be exported to the ExplorViz backend, in this part, they should be exported to a fitting third-party framework like Zipkin for traces and Prometheus for metrics. [Prometheus Authors 2024] [Zipkin Community 2024] When the instrumentation and exportation work as expected, the integration into ExplorViz becomes easier to debug. The only difference is that the exporter changes.

The collector flow from Figure 2.1 goes as follows: The collector should receive the traces and metrics from the OpenTelemetry SDK, which is responsible for instrumenting the data. Afterward, the chosen third-party exporters send the spans and events to their respective backends. This leaves out the processor, which is not essential here since the focus is on the instrumentation. All components (the application, the collector, Zipkin, and Prometheus) are deployed via Docker in the same network. This approach should lead to a clear understanding of how the automatic instrumentation by OpenTelemetry technically works for both NodeJS and browser applications.

3. Approach

3.2 Implementation into ExplorViz

When implementing the automatic instrumentation of traces and metrics into ExplorViz, both must be processed differently as described in Section 2.1. Therefore, the approach for each is presented in two different subsections.

3.2.1 JavaScript Traces into ExplorViz

Traces of Java applications are already being received in ExplorViz by its configured OpenTelemetry collector. The spans stem currently from the automatic instrumentation by InspectIT Ocelot. Since this framework provides its service only for Java applications, this work uses OpenTelemetry as outlined in Section 1.1. This means when looking at the architecture, the collector of the deployed JavaScript application collects the traces and exports them to the collector of ExplorViz. The last part alters the setup from Section 3.1, where Zipkin's backend received the traces. The focus is on the interfaces of both collectors, as the instrumentation in use has been established and is functioning well. Adding extra information to each exported span plays a role in this configuration. ExplorViz's backend expects every received span to contain a token and a value, which are important for assigning the traces to the respective visualization and identification within its database. In Section 2.1, the processing phase within the collector flow was discussed, allowing for inserting additional attributes to the spans to fulfill ExplorViz's requirements. Once the connection has been established, a visualization is expected to appear in the user interface of ExplorViz.

The previous steps use the simple examples mentioned earlier in Section 3.1, but scaling the example applications to a fair state is next. ExplorViz uses an example website application to monitor a fictional pet clinic, a Java Spring project, shown in Figure 3.1. There exists a version of this pet clinic project that deploys an Angular project ¹, which contains the frontend of the software and relies on a backend provided through a REST-API, which is based on Java Spring ². The chosen solution leverages the visualization of the front and backend, enabling communication between the two, which was one of the reasons for its selection. Another reason is that the well-distributed example software with all its different versions has proven to be a wise example for testing ExplorViz's capabilities. [Krause et al. 2021] The plan is to automatically instrument the Angular project while the backend runs via a supplied Docker image. After that, InspectIT Ocelot is set to instrument the backend, resulting in a distributed visualization.

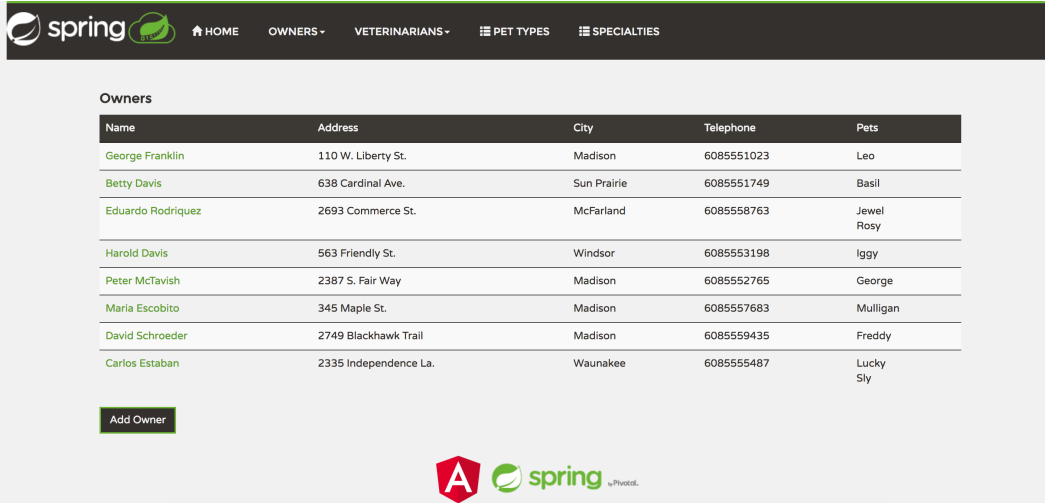
The Angular application includes the Web instrumentation of OpenTelemetry. As a result, a second example based on NodeJS is necessary. An open-source project that offers a web-based MongoDB admin interface has been selected. The project is built using NodeJS and Express, with a MongoDB backend. ³ The application meets the requirement of being

¹<https://github.com/spring-petclinic/spring-petclinic-angular>

²<https://github.com/spring-petclinic/spring-petclinic-rest>

³<https://github.com/mongo-express/mongo-express>

3.2. Implementation into ExplorViz



Name	Address	City	Telephone	Pets
George Franklin	110 W. Liberty St.	Madison	6085551023	Leo
Betty Davis	638 Cardinal Ave.	Sun Prairie	6085551749	Basil
Eduardo Rodriguez	2693 Commerce St.	McFarland	6085558763	Jewel Rosy
Harold Davis	563 Friendly St.	Windsor	6085553198	Iggy
Peter McTavish	2387 S. Fair Way	Madison	6085552765	George
Maria Escobito	345 Maple St.	Madison	6085557683	Mulligan
David Schroeder	2749 Blackhawk Trail	Madison	6085559435	Freddy
Carlos Estaban	2335 Independence La.	Waunakee	6085555487	Lucky Sly

Add Owner


 by Pivotal

Figure 3.1. Pet clinic application, giving an overview of different variables.

not too complicated to set up and deployable as a Docker container. Furthermore, the traces and, with foresight, the metrics are significantly more complex due to multiple views, documents, and collections the user can create, alter, or delete. The exemplary homepage of the software shows an overview of databases within the MongoDB instance and the server status displayed in Figure 3.2.

3.2.2 JavaScript Metrics into ExplorViz

The functionality for exporting metrics to Prometheus is assumed to be operational for the primary use cases described in Section 3.1. Contrary to Section 3.2.1, ExplorViz does not precisely process events. It lacks a service that does. Since handling metrics in an architecture focused on traces as explained in Section 2.2, adjusting the existing services can become too complicated. This leads to the approach of designing a new microservice whose focus lies solely on metrics transformation. This has the advantage of free configuration but can also become quite complex and challenging. For these particular reasons, the implementation will be pretty simple, and the aim, first and foremost, is to pass the events to ExplorViz's frontend successfully. They will be visualized there in the simplest ways possible, for example, in a table view. Of course, there are better procedures to implement and visualize the automatic instrumented metrics of a JavaScript application. However, the focus lies on the nature of events received from a complex application and their potential to aid in the comprehension of software. Unfortunately, the kind of metrics to expect is not explicitly specified in OpenTelemetry's documentation, which means it is impossible to determine whether the scale of the applications impacts the outcome of the instrumentation.

3. Approach

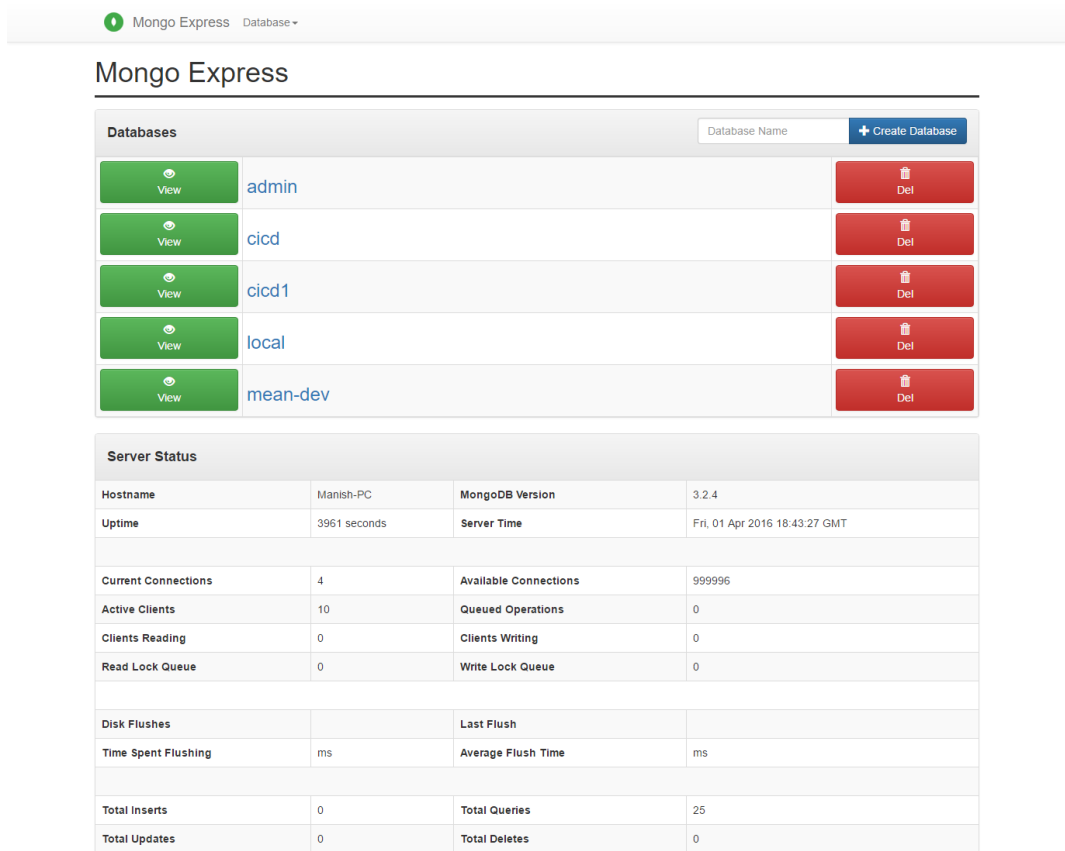


Figure 3.2. The homepage of a MongoDB admin panel application, showing the existing databases and the server status below. Source: <https://github.com/mongo-express/mongo-express> Accessed: [21.03.2024]

[OpenTelemetry Contributors 2024] Therefore, planning extensive visualizations is too dependent on the unknown, and deciding it later in this work seems logical.

In favor of simplicity, the new service should be based on the NodeJS framework Express. Furthermore, InfluxDB is used to store the metrics. Express has the advantage of being lightweight and high-performance with intensive Input/Output applications, which meets the key requirements of the planned service. Last, it does not add unnecessary complexity to the simple creation of Web APIs. As for the communication between the frontend and the metric service, HTTP requests suffice since this is the existing and functioning method the frontend employs. InfluxDB is chosen as the storage system for metric data because it is optimized specifically for time series data, offering various benefits as discussed in

3.2. Implementation into ExplorViz

Section 2.3. This makes it ideal for metrics but differs from ExplorViz's usual database, Cassandra. InfluxDB, as a cloud service, serves the immediate need to set up a database that can handle high write and read for time series data. Cassandra might not be unfitting for this purpose, but integrating would be more difficult, missing the focus of this work.

Essentially, the new service should be able to process the received events, store them accordingly, and appropriately transform them to be visualized by ExplorViz's frontend. There, a simple table view should suffice at first, and if the metrics have the potential, an extensive display can also be implemented. All in all, this should lead to visualizing specific useful metrics when instrumenting small JavaScript applications. After this pipeline functions as planned, integrating the two more complex examples from Section 3.2.1 becomes the next step. It is assumed that scaling the application may not lead to the necessary adjustments in instrumentation. Meanwhile, configuring the collector to export the metrics to the new service is required. Interesting to see will be whether the metric types differ between the size and complexity of the instrumented software.

Implementation

This chapter focuses on implementing the described approaches in Chapter 3 to reach Goal 1 and 2 from Section 1.2. The first section mainly concerns how automatically instrumenting small applications with OpenTelemetry works. It lays the base of how traces and metrics are being automatically instrumented from JavaScript applications in general, while Section 4.2 and Section 4.3 focus on how to integrate the instrumentation in the visualization provided by ExplorViz with scaled software.

4.1 Implementation of Automatic Instrumentation

This section starts by implementing the procedure of automatically instrumenting traces when working with a NodeJS and a Web application. After that, the same sequence is repeated for metrics. The examples used for NodeJS and Web come from the OpenTelemetry guide for JavaScript¹.

4.1.1 Instrumenting Traces of a NodeJS Application

As mentioned in Section 3.1, a small and simple application comprised of a single file is used to test the instrumentation's functionality. The system exports the spans to Zipkin, a third-party backend that facilitates viewing traces within a Web-based UI. [Zipkin Community 2024] This work uses a file called "tracer.js" to specify the instrumentation required to initiate the process. For frameworks based on NodeJS, OpenTelemetry provides a package named "SDK-Node". The library allows the configuration of the destination of instrumented spans and the type of instrumentation to be used. This thesis aims to implement an automatic instrumentation system using the "auto-instrumentation-node" package provided by OpenTelemetry. The logic is imported as a constant and then assigned to the instrumentation variable of the "SDK-Node" constructor in the instrumentation section.

The other essential part of the "tracer.js" is the "OTLPTraceExporter". It determines to which endpoint the instrumented traces are being sent. The exporter, imported with an OpenTelemetry-specific package, exports the traces to the in Section 2.1 explained

¹<https://opentelemetry.io/docs/languages/js/getting-started/>

4. Implementation

Listing 4.1. Instrumentation file for collecting traces from NodeJS applications.

```
1 // Required dependencies
2 const { NodeSDK } = require("@opentelemetry/sdk-node");
3 const { OTLPTraceExporter } = require("@opentelemetry/exporter-trace-otlp-proto");
4 const { getNodeAutoInstrumentations } = require("@opentelemetry/auto-
   instrumentations-node");
5
6 const sdk = new NodeSDK({
7   traceExporter: new OTLPTraceExporter({
8     url: "http://node-collector:4318/v1/traces",
9   }),
10
11   instrumentations: [getNodeAutoInstrumentations()],
12 });
13
14 // initialize the SDK and register with the OpenTelemetry API
15 // this enables the API to record telemetry
16 sdk.start();
```

collector by default. For consistency, all utilized exporters in this work are based on HTTP/Protobuf since ExplorViz expects spans serialized in proto, and the later relevant metric-service will work mainly with HTTP transports, as shown in Section 3.2.2. Therefore, the receiver of the collector must allow HTTP transport. This way, the adjustments in the later implementation stages become more manageable. The targeted URL is "http://node-collector:4318/v1/traces". It consists of the Docker container name of the collector, the default HTTP receiver port of the collector, and the suffix "/v1/traces", which specifies the traces endpoint of the collector, where traces are expected and processed. More configuration of the automatic instrumentation of traces from NodeJS is not required. Listing 4.1 shows how the described code could look in the base form. However, the instrumentation file must be integrated into the application's build process.

Usually, when working with a NodeJS application, it is necessary to inject the instrumentation file into the command when starting the application. So, adding the flag "--require tracer.js" to the command for starting the application or even to the CMD when building a Docker container with a Dockerfile is sufficient. For example, in this case, the starting command in the Dockerfile looks like this: " node --require ./tracer.js app.js ". It is recommended to store the "tracer.js" in the root where the main file (like "app.js" here) is placed.

This leads to exporting instrumented spans to the collector, which receives them accordingly and exports them to any configured destination, as defined in Section 2.1. Due to the absence of a need for span adjustment, the optional processor is bypassed. The exporter's

4.1. Implementation of Automatic Instrumentation

endpoint is configured to "http://zipkin:9411/api/v2/spans" like in Listing 2.1 to export the traces from the collector to Zipkin. This requires running Zipkin in a Docker container, and the application, the collector, and Zipkin must run within the same Docker network. The latter is critical for the whole instrumentation process; otherwise, the communication between all three containers malfunctions. Then, the instrumentation file cannot reach the collector, which, in turn, cannot build a connection to Zipkin. If everything in the "collector-config.yaml" was correctly configured like explicated in Section 2.1 and "tracer.js" is correct, all traces are visible in Zipkin's UI.

4.1.2 Instrumenting Traces of a Web Application

Another example application provided by OpenTelemetry, which initially only uses HTML code, is deployed to instrument traces of a Web application. NodeSDK does not work without adding extra packages to the source code, so instrumenting a Web application differs from the presented way in Section 4.1.1. The reason is that changing existing code violates the definition of automatic instrumentation from Section 2.1.2, which is why other libraries are required. It is necessary to embed a slightly different configuration compared to Listing 4.1, although some things do not change, like setting up the "OTLPTraceExporter". This time, the collector's hostname is "collector"; therefore, the target address differs in the configuration. The automatic instrumentation is configured by an imported "provider", which focuses on Web traces. Adding a so-called "spanProcessor" enables the implementation of the adjusted exporter. Still, the provider lacks the context of the instrumented spans; therefore, a context manager and propagator are also implemented. These are responsible for connecting spans to its child spans and gathering more information about a system across services [OpenTelemetry Contributors 2024], which helps to instrument useful spans. [OpenTelemetry Contributors 2024] Imported libraries provide every mentioned functionality, as seen in the dependency part of Listing 4.2. [OpenTelemetry Contributors 2024]

To utilize the provider's capabilities, one must apply a built-in function called "registerInstrumentation()" in the instrumentation file. It records the configured provider as a value for the variable "tracing" and sets up the automatic Web instrumentation. It is possible to customize this by limiting the instrumentation to specific targets. For example, if the goal is to focus solely on page-loading events in a Web application, limiting the instrumentation to "document load" is possible. The other possible focus packages are called "fetch", "user-interaction" and "xml-http-interaction". [OpenTelemetry Contributors 2024] However, limiting the instrumentation means missing out on some piece of information ExplorViz might need for a better visualization. Therefore, the automatic Web instrumentation is not customized further. How the complete resulting "tracer.js" looks like is shown in Listing 4.2

Another difference from the previous solution is the injection of the instrumentation into the software. Above in Section 4.1.1, the instrumentation file with the command "--require"

4. Implementation

Listing 4.2. Instrumentation file for collecting traces from Web applications.

```
1 // Required dependencies
2 const { WebTracerProvider } = require('@opentelemetry/sdk-trace-web');
3 const { getWebAutoInstrumentations } = require('@opentelemetry/auto-
  instrumentations-web');
4 const { OTLPTraceExporter } = require('@opentelemetry/exporter-trace-otlp-proto');
5 const { SimpleSpanProcessor } = require('@opentelemetry/sdk-trace-base');
6 const { registerInstrumentations } = require('@opentelemetry/instrumentation');
7 const { ZoneContextManager } = require('@opentelemetry/context-zone');
8 const { B3Propagator } = require('@opentelemetry/propagator-b3');
9
10 const exporter = new OTLPTraceExporter({
11   url: "http://collector:4318/v1/traces",
12 });
13
14 const provider = new WebTracerProvider({
15 });
16
17 provider.addSpanProcessor(new SimpleSpanProcessor(exporter));
18 provider.register({
19   contextManager: new ZoneContextManager(),
20   propagator: new B3Propagator(),
21 });
22
23 registerInstrumentations({
24   tracerProvider: provider,
25   instrumentations: [
26     getWebAutoInstrumentations({
27     }),
28   ],
29 });
```


4.1. Implementation of Automatic Instrumentation

was added to the CMD call within the Dockerfile, but this differs for Web applications. The reason is that the flag `--require` depends on NodeJS, which cannot be used here since the example is incompatible. The documentation suggests that one line of code should be added to the software's HTML file. The line in question is: `<script type="module" src="tracer.js"></script>`. In general, this means the instrumentation file is imported as a module into the application's main file. However, this means breaking the one rule of automatic instrumentation, which does not alter the original code, defined in Section 2.1.2. Still, there is no other way since the method from Section 4.1.1 only works using NodeSDK. This one line does not affect how the code operates and, hence, is as minimal as possible but deviates from pure automatic instrumentation.

4.1.3 Instrumenting Metrics of a NodeJS Application

As explained in Section 2.1, metrics differ in their structure fundamentally from traces, so, surprisingly, the code for instrumentation stays nearly the same. In Section 4.1.1, the essential `tracer.js` was thoroughly examined, including its critical parts, along with the setup process within the project. The overall foundation remains constant for automatically instrumenting metrics, but a metric-specific exporter is configured. This one is called `OTLPMetricsExporter`; it utilizes HTTP as the transport mechanism and Protobuf as the serialization protocol. As explained in Section 4.1.1, the HTTP/Proto exporter is chosen for consistency. The task is to integrate the new exporter into the `NodeSDK` constructor and ensure that all relevant metrics are automatically tracked and measured. Here, the only unexpected change is the necessity of a metric reader, imported within the package `SDK-metrics`. The `PeriodicExportingMetricReader` is used since metrics should be exported in periods and not without a system. Within this reader, the metric exporter to the collector is specified, which sends the events to the address `http://node-collector:4318/v1/metrics`, where the collector, deployed like in Section 4.1.1, can receive them. In Listing 4.3, all the new code lines are displayed in addition to Listing 4.1, except for `NodeSDK`, which is shown for context.

This means taking the code from Listing 4.1, adding the metric reader, and starting the application results in the automatic metric instrumentation. However, the collector does not receive the metrics due to the difference between traces and metrics. This requires an extra pipeline to export metrics as explained in Section 2.1, illustrated in Listing 2.1. The pipeline looks similar to the one for the traces with the distinction of another exporter. The software tool Prometheus is deployed for monitoring purposes, along with a recommended backend from the OpenTelemetry Website. This enables the examination of metrics via a user interface. [Prometheus Authors 2024] Necessary to note is that the corresponding container should be running within the same Docker network as the collector and the application to sustain the collector flow. The exporter address is `0.0.0.0:9464`, and port 9464 needs to be provided by the collector container. In addition to the container, Prometheus requires a file called `prometheus.yml` to configure the scraping target to fetch the metrics.

4. Implementation

Listing 4.3. Adjustements to instrumentation file for traces, to gather metrics from NodeJS applications.

```
1 const { PeriodicExportingMetricReader, ConsoleMetricExporter } = require("@opentelemetry/sdk-metrics");
2 const { OTLPMetricExporter } = require('@opentelemetry/exporter-metrics-otlp-proto');
3
4 const sdk = new NodeSDK({
5   metricReader: new PeriodicExportingMetricReader({
6     exporter: new OTLPMetricExporter({
7       url: "http://node-collector:4318/v1/metrics",
8     }),
9   }),
10 });
```

[Prometheus Authors 2024] This target is the address of the collector container with port 9464, which means the collector exports the metrics to its own provided port using the Prometheus exporter, and from there, Prometheus gets the automatically captured metrics. This ".yaml" file is mounted as volume onto the Prometheus docker container for this to happen. If everything is configured correctly, the automatic instrumented metrics from the example application of Section 4.1.1 can be examined at "http://localhost:9090", the port of the Prometheus Docker container.

4.1.4 Instrumenting Metrics of a Web Application

In the context of Web and browser applications, OpenTelemetry focuses mainly on the traces so far. The metrics instrumentation for client-side JavaScript software is less developed than server-side JavaScript software and other frameworks like NodeJS. This leads to fewer integrated solutions. In contrast, manual metrics instrumentation is possible, which must be defined explicitly to track specific metrics. However, this is not the aim of this work, as it aims to automatize the capturing part without interfering with the source code. An additional line needs to be added to the Web example to instrument the traces as shown in Section 4.1.2, which already violates the definition of automatic instrumentation. To further alter the code to capture valuable and useful metrics is not part of the specified goals in Section 1.2, which leads to the conclusion this work cannot provide metrics from Web or browser applications written in JavaScript instrumented via OpenTelemetry. Since it is an open-source and active project, one might expect automatic instrumentation of metrics in this context as a feature.

4.2 Implementation of Traces in ExplorViz

While Section 4.1 concentrates on implementing the automatic instrumentation pipelines for both traces and metrics, this section focuses on further visualizing them in ExplorViz and scaling the example applications. This section covers the steps necessary for implementing NodeJS and Web software traces into the existing logic of ExplorViz, which is covered in Section 2.2. Section 4.2.1 tackles NodeJS instrumentation and Section 4.2.2 the Web instrumentation.

4.2.1 NodeJS Traces visualized in ExplorViz

As described in Section 3.2.1, sending the traces of the small application from Section 4.1.1 for visualization to ExplorViz is the first step. This is easier to debug, knowing the instrumentation works fine. The spans should be sent to ExplorViz's backend and not to Zipkin. While this may initially look like a simple alteration of the endpoint, there are things to look out for. Zipkin as an exporter is configured and provided by OpenTelemetry, which means any further adjustment other than the container must run in the same Docker network as the collector is not required.

In contrast, ExplorViz expects additional attributes to process the spans and mark them as valid. Here, the logic of ExplorViz's architecture becomes essential to remember as described in Section 2.2. Every span must include the generated landscape token and token secret, the application name, its instance ID, and the programming language used. Otherwise, the frontend cannot allocate the data to a software landscape.

Fortunately, the processor within the explained collector flow allows the insertion of attributes to the processed spans. When adding an attribute, it is necessary to define the key, such as the landscape token, along with the corresponding value generated by the frontend. What to do with this key-value pair is also determined, which is "insert" in this case. Then, the attributes need to be added to the pipeline. How this would look is displayed in Listing 2.1. Furthermore, configuring a new exporter becomes the next obstacle. As described in Section 2.1, OpenTelemetry provides a modifiable exporter using its protocol, and since ExplorViz's collector expects spans via the gRPC transport protocol, the customizable otlp exporter is chosen. ExplorViz determines the endpoint. These adjustments result in ExplorViz receiving the spans, accepting them, and visualizing them successfully in the browser. However, the visualization is fundamental and does not contain much information, which different reasons can explain. It is possible that one factor contributing to this issue could be the relatively low complexity of the instrumented software. As a result, integrating the MongoDB admin panel application., presented in Section 3.2.1, is next.

After the application is set up and running, instrumentation configuration from Section 4.1.1 is integrated to extract the spans automatically. Here, the advantage of automatic instrumentation is displayed because, in general, adjusting the content of "tracer.js" is unnecessary, but changing the file name to "tracer.cjs" is required. This means it becomes a

4. Implementation

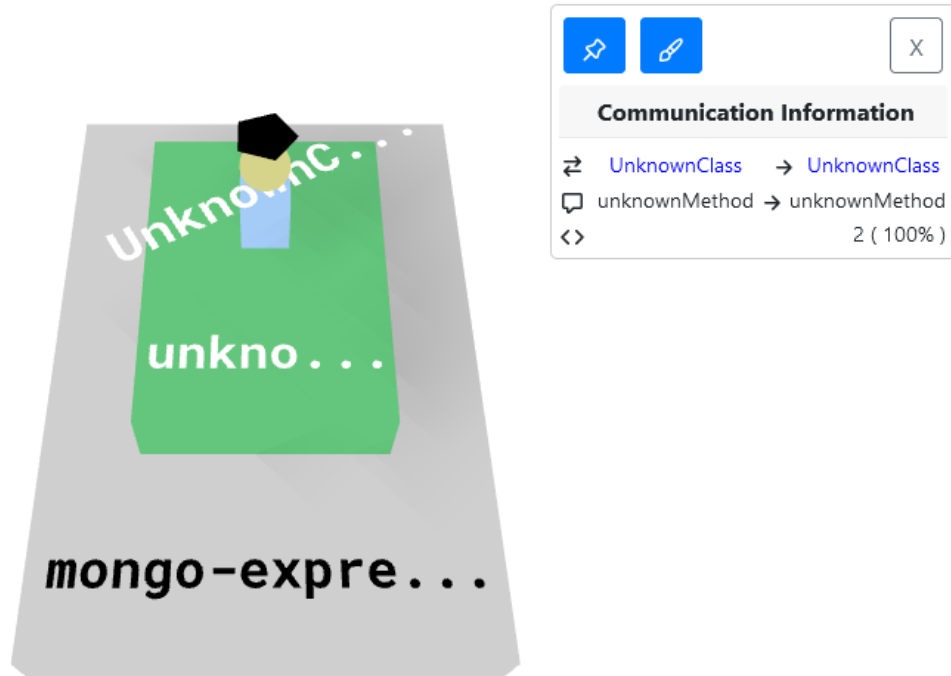


Figure 4.1. Visualized Mongo-Example without adapter-service adjustment, displaying "unknowns".

CommonJS file, which is necessary since the chosen NodeJS server-side application treats each file as a separate module, which is a JavaScript extension. However, this change does not influence the instrumentation file itself. It is essential to integrate the tracer file into the Dockerfile and configure the collector as a Docker container in the same network as the application and ExplorViz. This is no different than explained in Section 4.1.1.

After fulfilling the expectations of ExplorViz for the incoming spans and instrumenting a complex application, nothing stands in the way that it can visualize the NodeJS example in more depth. However, ExplorViz usually receives the fully qualified name of the origin of the span. This is not a mandatory attribute but plays a significant role in the visualization. The fully qualified name consists of a package name, a class name, and the method's name, which identifies the exact point in the application where a span was generated. Explorviz uses this to get valuable information about the software's structure and communication as described in Section 2.2. It usually receives traces from Java applications, instrumented via InspectIT Ocelot, which adds the correct name to each span. The problem is not with adding further attributes to the span, as "tracer.js" could be altered by using a custom

4.2. Implementation of Traces in ExplorViz



Figure 4.2. Visualized Mongo-Example with adapter-Service adjustment, displaying communication between span IDs.

span processor, which opens the possibility of inserting a key-value pair at the moment of instrumentation. The issue is the fully qualified name, as JavaScript is structurally different from Java. Therefore, getting any method or function-specific name at the exact time when the span is created is not possible. Thus, the visualization of the MongoDB admin panel software looks like in Figure 4.1.

As it can be seen, due to the missing fully qualified name, the landscape consists of "unknowns," as this is the default value. The adapter-service, responsible for receiving and deserializing the spans, validating them, and transforming them into a format so the following span-service can work with them, sets this as no other information exists. In this procedure, the adapter service identifies the extra-added attributes and processes

4. Implementation

them further. Since, in this case, there is no such thing as a fully qualified name, the adapter-service uses the default value of "unknownPkg.unknownClass.unknownMethod" for transforming, which results in the visualization in Figure 4.1. To enhance the aesthetics of the landscape, it is possible to consider renaming it as "package.class.<the span ID of every processed span>". This could facilitate communication to some extent, as the span IDs may function as child spans and interact with the parent span. This is achieved by adding a small clause to the adapter-service in the class to process the spans and ask whether the span has a fully qualified name. If it has, the service does nothing; if not, it sets the value to the mentioned target value and ensures it is used. This leads to a slightly different visualization in Figure 4.2 that displays at least some hint of communication between the spans. Furthermore, Figure 4.2 is also the content evaluated in Chapter 5.

4.2.2 Web Traces visualized in ExplorViz

Since visualizing traces from a Web application is tackled after the progress from Section 4.2.1, this section focuses on scaling the example application from Section 4.1.2. The more exciting part is that this approach does not only automatically instrument one application but technically two, although the backend and frontend are meant. In Section 3.2.1, the well-used sample software for a pet clinic is presented, which consists of a JavaScript Angular frontend and a Java Spring backend. However, to stay modularly, this part automatically instruments the Angular project first, requiring the source code, since injecting the "tracer.js" by adding it to the build process within the Dockerfile is unfortunately not enough, as pointed out in Section 4.1.2. Here, the source code consists of more than one HTML file, which is why the importation of "tracer.js" as a module happens in the main file by adding it to the required dependencies ("import './tracer';"). The backend provides an API Docker image, which enables the visualization of a working Web frontend without needing the source code of the backend. This results in a similar visualization in Figure 4.2, with a different application name and other span IDs.

So, the next step is to instrument the backend using InspectIT Ocelot, visualizing the whole software in ExplorViz. As justified in Section 3.2.1, using OpenTelemetry for instrumenting the backend would also go beyond this work's research area. Therefore, an existing configuration from other examples by ExplorViz is employed, which requires building the backend locally and not deploying the existing image provided by DockerHub. A Dockerfile that clones the project via Git suffices since the resulting "app.jar" is connected to a file named "inspectit.yml", which covers the InspectIT Ocelot instrumentation code. [NovaTec Consulting GmbH 2024] In this file, the exportation address to ExplorViz's backend is configured as well, the same as the collector uses for traces of the frontend. What is done by the processor of the collector flow is accomplished by InspectIT Ocelot with tags. The landscape token and the token secret must have the same values as the frontend to be allocated to the same visualization. To ensure consistency, employing environment

4.3. Implementation of Metrics in ExplorViz

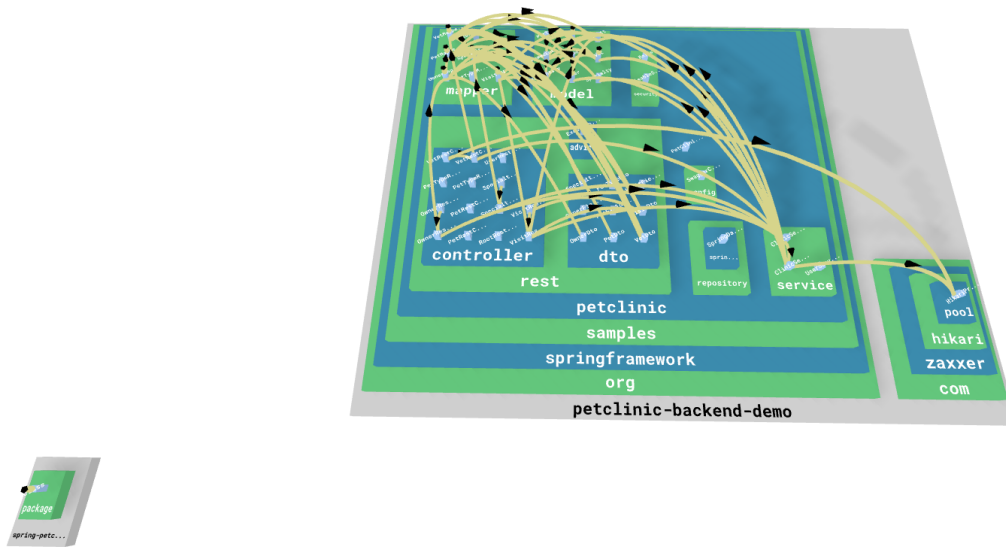


Figure 4.3. Visualization of the pet clinic example with OpenTelemetry instrumenting the frontend and InspectIT Ocelot the backend.

variables is the best and safest practice. ExplorViz can differentiate between the spans since they have different application names, which results in two landscapes like in Figure 4.3.

However, the visualization seems enriched at first, but realizing that no communication between the backend and frontend is tracked makes it seem like two different applications are visualized in the same picture. The reason for that is most likely the difference in focus between both technologies since InspectIT Ocelot concentrates on internal processes more than on communication with the outside. [NovaTec Consulting GmbH 2024] It might result in a visualization that captures the communication between the frontend and backend if both instrumenting tools are the same, but this may work in the future.

4.3 Implementation of Metrics in ExplorViz

As described in Section 4.1.3, metrics are collected nearly the same way as the traces for JavaScript applications based on NodeJS. The only difference is that the collector processes them differently; "tracer.js" sends them to a different collector address. Contrary to Section 4.1.3, the exporter does not send the metrics to Prometheus but to ExplorViz. As pictured in Section 3.2.2, this invokes the need to construct a new service to process metrics and transform them so the frontend can visualize them. Therefore, Section 4.3.1 goes more deeply into how the metrics get to the service and how they are initially transformed. After

4. Implementation

that, Section 4.3.2 takes a closer look at visualizing them on ExplorViz's frontend. This section uses the scaled application of the MongoDB admin panel, presented in Section 3.2.1 and already deployed in Section 4.2.1. This also counts for the collector configuration, meaning attributes for identification are added.

4.3.1 Metric-Service

The idea of the metric-service is to receive the metrics, convert them to JSON, and then filter out the essential data relevant to the visualization. InfluxDB stores this data, which is sent to ExplorViz's frontend when requested.

Receiving the metrics involves more than creating an address within the container to which the collector from the example application exports the metrics. In Section 4.1.3, the metrics are exported to the third-party backend Prometheus to simulate the exportation process. The configuration is more straightforward since Prometheus is a known exporter address for OpenTelemetry, already configured and only requiring a valid endpoint. One of the configurable otlp exporters sends the metrics to the metric-service. Section 2.1 already discussed the types of exporters and their protocols.

In Section 4.2, the regular otlp exporter is sufficient since ExplorViz receives the spans via gRPC and sends them to Kafka, for which a fitting exporter exists. In this section, an HTTP request transmits the events to the metric-service, enabling the utilization of the otlp HTTP exporter from OpenTelemetry, which uses a POST request to export the metrics. The metrics are sent to the endpoint address with the suffix `/v1/metrics`. This means the service requires an address for POST-request. Since the service runs on port 8085, the URL is `http://localhost:8085/v1/metrics`, but because the metric-service and the example application are running within the same Docker network, `localhost` can be replaced by `metric-service`. The events are serialized in the Protobuf schema, as explained in Section 2.1, which leads to the metric-service's first task, deserializing the data.

First, the package `protobufs` is installed to perform Proto-specific operations with the data. Since the used framework is Express, the expected standard format of incoming data is JSON; hence, adjusting the service's middleware to expect Proto data is necessary. The Protobuf metrics structure is not universally the same, but OpenTelemetry has defined one for its purpose. Therefore, a GitHub repository² of the metric's pattern is provided. Adding it to the service enables the transformation of the incoming metrics into JSON format. The only configuration within those Proto files might be the import paths of the other files. Since the structure of the files embedded in the service can differ from the repository, the paths may need updating. All relevant Proto files should be checked because they are intertwined.

The process of how the incoming data is decoded using the library `protobufs` works like this: In-build methods load the specified Protocol Buffer schema, which contains the

²<https://github.com/open-telemetry/opentelemetry-proto/tree/main/opentelemetry/proto>

4.3. Implementation of Metrics in ExplorViz

structure of the incoming data, and define the decoding process. The service uses this definition to decode the incoming binary data encapsulated within the body of the POST request into usable JavaScript objects. After that, another in-built method converts the decoded message into standard JSON object format, facilitating the manipulation and access of the metrics data within the application.

After decoding, the readable metrics are validated by a function named "validateMetrics()", which checks whether the incoming data meets the expected schema and is disposed of essential attributes for further use. These are the metric itself, the landscape token, and the token key, which are both required to identify to which visualized landscape the metric belongs. Additionally, multiple metrics may be received within the same file, meaning the validator iterates over them before the final judgment. An extension to the function would be the further examination of the landscape token and its secret to not only its existence in the metrics but also in the underlying database of ExplorViz.

As the incoming data has been decoded and validated, it is stored in an InfluxDB, running in the background. How it works, respectively, and what is required to set it up is broadly described in Section 2.3. The reason behind the usage of InfluxDB for this kind of data is further explained in Section 3.2.2, which leads to the actual operation to store the metric events. The data is filtered for specific values and attributes that are relevant to the visualization. This helps to eliminate irrelevant information and focus only on what is required. Namely, these are the metric name, value, the time when it was instrumented, the corresponding unit and description, the landscape token, and the token secret. The first four are the relevant parameters for a helpful metric in any visualization. At the same time, the description further explains the metric's information since the metric names might not always be self-explanatory. The token and its secret assign the metric to a specific application from which it originates. This is important for the frontend to get the right metrics for the correct software landscape.

It was discussed how the metrics are processed when sent as an HTTP-POST request to the service, with the final step being storing the critical attributes in a fitting database. Section 4.3.2 dives deeper into the actual visualization, but first, this subsection defines how the frontend gets the correct metrics they request via HTTP. This happens via a standard Flux request on the database, which is nothing more than InfluxDB's scripting language, further explained in Section 2.3. The frontend adds the time stamp and the landscape token to the request body since it asks for metrics for a specific landscape gathered around a particular time. The goal is to respond with an array of all the metrics that fit these criteria. Due to the systematic gathering by the "PeriodicExportingMetricReader" described in Section 4.1.3, which exports metrics only in periods and not constantly, it makes sense to filter for metrics within the last minute before the given time stamp. Flux achieves this by ranging between a start and a stop date. To subtract one minute from the given time stamp, the functionality of the "date" package is imported within the query itself. After that, the metrics are filtered further for the landscape token. Due to the additional storage of the

4. Implementation

time and how long it took to store the data, which InfluxDB does with every saved data point, a constraint of the kept columns is induced. These are the same as the ones stored above: the name (or in InfluxDB measurement), the time, the value, the unit, the token, and the metric's description. The described Flux-query can be found in Listing 4.4, and one observes that the structure is similar to a SQL-query, which is the base of Flux. If the range of functions is expanded, authentication with a bearer token can be added, for which the token secret might be helpful. However, the visualization does not depend on it, so the token secret is filtered out. In the end, all metrics that fit the criteria of the landscape token and the correct period are collected in an array and sent back to the frontend as a response. All these steps have an error handling in case anything irregular occurs. However, a request that invokes an empty response is not treated as an error since some applications might not be instrumented for metrics or metrics were not gathered at the asked time.

Listing 4.4. Flux-Query to get the metrics, gathered within the minute before the given time stamp.

```
1 const fluxQuery = flux'  
2   import "date"  
3   from(bucket: "${bucket}")  
4   |> range(start: date.sub(d: 1m, from: ${timestamp}), stop: ${timestamp})  
5   |> filter(fn: (r) => r.landscape_token == "${landscapeToken}")  
6   |> keep(columns: ["_measurement", "_time", "_value", "unit", "  
   landscape_token", "description"])  
7   |> yield(name: "filtered_last_min")'
```

4.3.2 Integration into ExplorViz's Frontend

Before altering the frontend, the following question needs to be answered: what is a pleasant way to visualize metrics? It depends on the metric types and what kind of operations with the metrics are useful and possible. In this case, the metrics are predetermined by OpenTelemetry and mostly consist of request monitoring, which means focusing on HTTP requests as a whole and not specific types. Therefore, a comparison makes little sense, and a tabular view, displaying the metric with its value, the corresponding unit, and the time it was gathered, is chosen. A classical tabular view seems safe, listing the metric with its value, the corresponding unit, and the time it was gathered. How this is perceived, the evaluation shows in Chapter 5.

Visualizing the metrics in a table makes integrating the adjustments into the frontend easier. The last paragraph in the above in Section 4.3.1 details how the logic behind a GET request to the metric service is handled if no errors are thrown. So, the expected response, an array consisting of JSON objects, is implemented as an interface so that every incoming data is processed as instances of it. The table is visualized in the existing sidebar as additional column "Metrics", which shows an empty table with the respective columns

4.3. Implementation of Metrics in ExplorViz

Metric Data			
Load Metrics			
Metric Name	Timestamp	Value	Unit
db.client.connections.usage	2024-02-23 09:09:09	0	{connection}
db.client.connections.usage	2024-02-23 09:09:09	2	{connection}
db.client.connections.usage	2024-02-23 09:09:09	0	{connection}
db.client.connections.usage	2024-02-23 09:09:09	2	{connection}
db.client.connections.usage	2024-02-23 09:09:14	0	{connection}
db.client.connections.usage	2024-02-23 09:09:14	2	{connection}
db.client.connections.usage	2024-02-23 09:09:18	0	{connection}
db.client.connections.usage	2024-02-23 09:09:18	2	{connection}
db.client.connections.usage	2024-02-23 09:09:19	0	{connection}
db.client.connections.usage	2024-02-23 09:09:19	2	{connection}
db.client.connections.usage	2024-02-23 09:09:29	0	{connection}
db.client.connections.usage	2024-02-23 09:09:29	2	{connection}
http.client.duration	2024-02-23 09:09:09	197.61171399999998	ms
http.client.duration	2024-02-23 09:09:09	265.087874	ms
http.client.duration	2024-02-23 09:09:14	199.79958299999998	ms
http.client.duration	2024-02-23 09:09:18	270.18364299999996	ms
http.client.duration	2024-02-23 09:09:19	280.32424999999995	ms
http.client.duration	2024-02-23 09:09:29	268.592834	ms
http.server.duration	2024-02-23 09:09:09	1070.670098	ms
http.server.duration	2024-02-23 09:09:09	1114.2573859999995	ms
http.server.duration	2024-02-23 09:09:09	86.05129600000001	ms
http.server.duration	2024-02-23 09:09:09	107.60295	ms
http.server.duration	2024-02-23 09:09:09	193.87650100000002	ms
http.server.duration	2024-02-23 09:09:09	1070.670098	ms
http.server.duration	2024-02-23 09:09:09	1114.2573859999995	ms

Figure 4.4. Exemplary table view of instrumented metrics, displaying the name, time, value, and unit.

"Metric Name", "Timestamp", "Value" and "Unit". Furthermore, a button exists, which, on click, triggers the described GET request for the time stamp that is currently viewed in the landscape visualization. The received JSON objects are then mapped in a tracked array; therefore, any updates on its content are directly visible. The reason for that is the connection of the body of the table with the said array. Additionally, an icon on the left of the metric name is created, which displays the description of that metric when hovered over. All this results in a view like in Figure 4.4 that illustrates metrics corresponding to the shown software landscape, which depend on when they were instrumented. This leads to a small dynamic within this visualization since the presentation changes conditionally at the chosen time to load the metrics. However, there might be times when no metrics are shown due to the types of metrics, which primarily emerge from actions in the user interface. It has to be noted that for the NodeJS example application of the MongoDB admin panel, only three types were generated. These are evaluated in the next chapter to determine whether they are helpful for further software understanding.

Evaluation

This chapter evaluates the implemented features of OpenTelemetry into ExplorViz, described in Chapter 4. However, the code is not assessed; it is the altered interface of ExplorViz, focusing on the new example landscapes based on JavaScript. Mainly, this evaluation tries to answer whether the new features enhance software visualization by ExplorViz. Most importantly, it tries to draw a first-draft conclusion on whether OpenTelemetry is an adequate tool for automatically instrumenting JavaScript software to visualize it. First, the structure and procedure of the evaluation are presented, then a neutral analysis of the responses is conducted, and the findings are qualitatively summarized. At last, the threats of validity are discussed. The raw data of the evaluation can be seen [here](#).

5.1 Structure and Procedure

The implementation is evaluated through a qualitative survey using open-ended questions. This is more helpful since only six people participate in it, most of them being developers of ExplorViz, and are therefore qualified as "experts". This thesis about the automatic instrumentation of JavaScript software through OpenTelemetry and its resulting display is an entirely new approach compared to existing work in ExplorViz. Hence, the evaluation should aim to fundamentally answer whether this approach could improve software visualization in ExplorViz as defined as a goal in Section 1.2.3. This is done best with people with high expertise in ExplorViz and broad programming knowledge, allowing them to answer the main evaluative question proficiently. Everyone executes the evaluation on their own.

First, everyone gets a quick insight into the purpose of this evaluation and the topic of this thesis before they perform the survey. The survey consists of nine questions; the first two reflect the previous knowledge described by each participant. After that, every participant is presented with the NodeJS example application described in Section 3.2.1 and the corresponding visualization from Section 4.2.1. At the same time, the metrics from Section 4.3 are introduced, and where they can be found. Additionally, they are informed that metrics only exist for this application due to reasons specified in Section 4.1.4. Everyone is free to interact with everything presented as long as needed and proceed with the survey. This exploration is not supervised and is not part of the evaluation. The participants can

5. Evaluation

switch between survey and visualization without restriction. The survey continues with one block of two questions focusing solely on the visualized landscape of the NodeJS software, following a second and similar structured block, which targets evaluating the metrics and their display. Both parts ask if it helps to understand the software and what the participants suggest regarding improvement. For the metrics, a question is asked in addition to how satisfied the participants were with the display of the metrics as a table view.

When finished, the pet clinic example from Section 3.2.1 and its visualization in ExplorViz from Section 4.2.2 are shown. Here, the explicit request is that this part of the evaluation focuses on the ensemble of front and backend since this is the significant difference between both examples. This section repeats the questions from the previous part concerning overall software understanding and suggested improvements. Again, the participants can inspect the application and visualization as long as they wish to and even come back while writing and answering the questions. After that, the survey is finished.

To facilitate a comprehensive analysis of the responses, it is necessary to provide a detailed account of the visualizations reviewed by all participants during the evaluation process. The software's user interfaces are exemplary, displayed in Figure 3.1 and Figure 3.2. A possible visualization in ExplorViz of the MongoDB admin interface can be seen in Figure 4.2 with the corresponding metrics shown in Figure 4.4. The pet clinic visualization can be seen in Figure 4.3. The screenshots of the visualizations accurately represent what the participants evaluate since, for the whole process, ExplorViz's demo supplier was used. The built-in service mocks the backend and provides recorded data when the frontend requests it. This means that only these two services are running during the evaluation, which is beneficial because every participant works with the same visualizations. On the downside, the interactive part of generating spans and events by clicking through the example applications gets lost, and no relation between action and visualization of the software and its metrics persists. This might affect the answers, but guaranteeing everyone the same circumstances seemed more critical.

Every participant did the survey on the same computer provided by the Software Engineering Group of the University of Kiel. Four tabs were open simultaneously: one for the survey hosted by LimeSurvey, one for ExplorViz's frontend, and one for each example software interface. The primary evaluator sat in the same room during every evaluation, answering technical unclarities without interfering in the formation of opinion.

5.2 Neutral Analysis of Responses

The survey was conducted among six participants with diverse programming experiences, three at intermediate and three at advanced levels. Intermediate means being comfortable with several programming languages and concepts, while advanced was labeled as having

5.2. Neutral Analysis of Responses

extensive experience with software development and complex systems. Their familiarity with ExplorViz or similar tools varied, with most being developers of ExplorViz, while others have had limited or no source code and visualization exposure so far.

The survey responses regarding the usefulness of the added visualization of JavaScript applications in understanding software structures indicate a consensus that it falls short in its current form. Respondents pointed out the inadequacy of the visualization when it displays only a generic class or a single package. While some respondents noted that the ability to view function calls was beneficial, they also expressed that such a feature did not compensate for the overall lack of depth in the visualization. Respondents were critical of its inability to illustrate the complexity of an actual software application, such as the connection to a database. They noted missing essential elements like method calls and communication between classes. There was a feeling that the visualization needed to go beyond the basic structure to include detailed insights into how different parts of the software interact. The current visualization was seen as too simplistic, providing an incomplete picture that does not represent the intricacies and dynamics of the software's architecture one might expect.

Suggestions for improvements were focused on enhancing the communicative aspects of the visualization. Participants advocated for the visualization to include communication lines within and between the software and external components like databases. They also requested the inclusion of meaningful names for classes, packages, and methods to facilitate a more comprehensive understanding of the software.

The participant's feedback on the presentation of metrics in ExplorViz reveals a spectrum of satisfaction, with most respondents expressing neutral to satisfied sentiments and one indicating dissatisfaction. The responses to the open-ended questions suggest that while presenting metrics provides some utility, there is a clear division in user experience and expectations. Respondents who found the metrics helpful appreciated the ability to understand the frequency and duration of HTTP requests and database connections, pointing to a clear layout and the helpfulness of explanatory icons. However, there is an acknowledgment that the utility of these metrics could be case-dependent, with some questioning the necessity of specific data points, such as duration values, unless they highlight significant performance concerns.

The insights provided by the metrics were recognized as valuable in understanding client-server interactions, but there were concerns regarding the realism of the data. The use of mocked data drew criticism for not allowing an authentic experience of ExplorViz's responsiveness and alignment with actual runtime behavior. Some respondents felt the data did not effectively contribute to understanding the software due to a lack of evaluation and relation to the software's internal processes.

Ideas for improvement were directed towards enhancing clarity and functionality in the metrics visualization. Participants requested features such as filtering, sorting, and

5. Evaluation

more explicit delineation of the metrics, such as whether duration values pertained to processing times or connection durations. The need for aggregated views, such as graphs or cumulative data representations, was highlighted to improve readability and provide a more comprehensive overview. Additionally, there was a call for the visualization to address discrepancies in the presentation of metrics, such as the inconsistent display of timestamps and function requests. There were also calls for features that would enhance the interactivity and informativeness of the metrics, such as linking metrics to specific classes or components within the visualization, average values, anomaly detection, and more straightforward representation of metric summaries. This feedback indicates a desire for the metrics visualization to present raw data with the freedom to filter and to serve as a feature for diagnosing and understanding the deeper aspects of software performance and behavior.

As indicated in the responses regarding the separate visualization of front and backend in ExplorViz, the participants found the distinction between the two to be conceptually clear but lacking in depth and connectivity. The predominant view is that the visualization does not provide a meaningful connection between the front and backend components, essential for a comprehensive understanding of the overall software system.

Most respondents expressed that the visualization, in its current form, fails to offer significant insights into how the front and backend interact. The lack of visible communication lines was a common critique, with users noting that this omission necessitates additional knowledge beyond what ExplorViz provides. There was an appreciation for the module representation within each front and backend section. Still, the visualization was seen as too minimalistic and failed to reflect the actual software structure and behavior, especially for the frontend component.

In terms of suggested improvements, there is a strong call for the visualization to incorporate elements demonstrating the interaction and data flow between the front and backend. Respondents seek enhancements such as visualizing communication lines, including meaningful names and structural information for frontend elements, and an intuitive representation that clarifies the relationship between the two parts of the software. Additionally, suggestions were made to improve the navigation of the software landscape with more intuitive labels, icons, and representations that can help users differentiate between the front and backend more easily.

5.3 Evaluative Summary

The analysis of participant feedback reveals an overarching perspective that the current visualization of JavaScript software, instrumented by the tool OpenTelemetry, in ExplorViz has significant areas for improvement to be truly effective. Participants agree that its present capabilities are not sufficient for in-depth software understanding. The consensus that the visualization fails to represent complex software architectures underpins the need

5.3. Evaluative Summary

for evolving beyond a simplistic structural display. There is a critical demand for more detailed visualization that captures the nuances of software interaction, especially the communication between classes and methods that are currently absent.

The underlying problem this evaluation has been pointing out is the lack of layers of the source software. ExplorViz requires a meaningful "fully qualified name" that identifies the origin of a span in the code and gives insights into the structure. While the possibility of adding custom attributes to the span exists, no feasible solution for getting such information during the instrumentation exists, as described in Section 4.2.1. This alone hinders ExplorViz's ability to build a multi-layer landscape with connections between methods. Hence, the question needs to be asked whether OpenTelemetry's proposed way of automatically instrumenting JavaScript software lacks this feature or if JavaScript as a programming language is creating this difficulty. Both could be true as well, but it is unfavorable that the fundamental structure of JavaScript deviates much from Java's, which ExplorViz is very familiar with.

The evaluation of metrics displayed within ExplorViz indicates a spectrum of user satisfaction, though even satisfied users acknowledge room for improvement. The clear layout and presence of explanatory icons are praised; however, the actual utility of metrics like HTTP request durations and database connections is debated. The feedback highlights a desire for metrics to be data points and a narrative that contributes to the story of the shown software's performance and behavior, guiding the user to actionable insights. Participants suggest that the metrics visualization should function as an interactive and informative diagnostic add-on, with the ability to filter, sort, and better understand the significance of presented data.

However, this shows why automatic instrumented metrics from JavaScript software by OpenTelemetry cannot meet ExplorViz's standard of supporting software understanding. During all test runs and multiple tries, only three types of metrics were automatically gathered, and these were shown in the evaluation, and the participants doubted their usefulness. If this base criteria cannot be satisfied, then any other way of visualizing it will not solve the root problem. Since automatic instrumentation is the goal, there is no further influence on the metrics that can be visualized. The critique that the metrics are just presented without possible filter and sorting options does make sense. However, it is questionable that such options would outweigh the lack of quality of the pure metrics. The suggestion that the metrics should be directly connected to the software visualization is desirable, but this is feedback for future times when meaningful metrics are instrumented. Then, configurations in the frontend for a more pleasing presentation become more significant.

The separate visualization of the front and backend received mixed reactions. While the conceptual distinction is clear, the absence of visual cues linking the two segments leads to a fragmented understanding of the system. Users need an integrated visualization

5. Evaluation

representing the individual components and their interactions, which is pivotal to comprehending the overall software system. Respondents called for specific improvements, such as visualization of communication lines between front and backend and enhanced navigational elements to traverse the software landscape intuitively. The feedback indicates a need for a visualization that conveys the individual aspects of software architecture and the dynamic flows of information and control between these elements.

These flows are not detected due to several problems. First, two different instrumenting technologies are used, and second, InspectIT Ocelot focuses more on detecting the internal processes. Either way, this might have been fixed by instrumenting the backend with OpenTelemetry. However, as said in Section 3.2.1, the goal of this work remains to inspect the instrumentation of JavaScript software for visualization. Therefore, pursuing the idea of instrumenting both the backend and frontend with OpenTelemetry is categorized as future work. However, the criticized and sparse frontend visualization will probably persist. However, the evaluation made this very clear: without distinct communication lines between both parts of the software, the visualization of the front and backend cannot help viewers better understand the software.

In summary, the participants' responses illustrate that the solution of instrumenting JavaScript software with OpenTelemetry for visualization purposes must undergo significant enhancements to meet the needs of its users. The tool must provide more intricate and connected data so the view of software systems in ExplorViz supports users' understanding of complex architectures. This would involve enriching the spans with more detail, improving the functionality and interpretability of metrics, and ensuring the data's alignment with actual software behavior. Addressing these aspects would elevate OpenTelemetry to a tool that could be an indispensable asset for software visualization in the context of ExplorViz.

5.4 Threats of Validity

This section focuses on the evaluation's threats to validity and is structured in terms of internal, external, and construct validity.

5.4.1 Internal Validity

Internal validity pertains to the extent to which a causal relationship between the evaluation and its outcome can be established. It is crucial to ensure that the study's design and execution allow for accurate conclusions. [Bhandari 2019]

- ▷ **Confirmation Bias:** Participants, due to their expertise and involvement with ExplorViz, may inadvertently seek information or interpret questions in ways that affirm their preconceived notions about the software's performance. This bias could influence their evaluations, affecting the study's internal validity. However, the goal was to conduct a

5.4. Threats of Validity

qualitative survey since this work presents a new feature in the context of ExplorViz. So, the value of qualitative and longer answers from participants with expert knowledge outweighs this possible bias.

- ▷ **Influence of Introductory Information:** Providing preliminary information about the study's goals and the thesis topic might prime participants to answer in ways that align with the expected outcomes of the evaluation. This priming effect can influence participants' objectivity, potentially skewing the results and compromising internal validity by introducing a systematic bias in how information is interpreted and reported. It is improbable that the participants' proficiency is affected by introductory information as it only provides context, which is insignificant given their extensive familiarity with ExplorViz.

5.4.2 External Validity

External validity refers to the generalizability of the study's findings beyond the immediate context of the research setting. [Bhandari 2020]

- ▷ **Sample and Observer Bias:** With a small group of six participants, most of whom are experts in ExplorViz, there is a significant risk that the findings may not apply to a broader audience. These participants' specialized knowledge and experience may give similar expert users highly relevant insights. Still, this limitation restricts the generalization of the evaluation results to other contexts, affecting the evaluation's external validity. However, as pointed out before, to get an impression of whether automatic instrumentation with OpenTelemetry is suitable for software visualization in ExplorViz, it makes sense to do a qualitative evaluation with a few specialized participants. This leads to further revisions before conducting a more generalizable study on the topic, especially since this is the first work done in the environment of ExplorViz.

5.4.3 Construct Validity

Construct validity involves the degree to which the evaluation measures what it intends to measure, ensuring that the investigated constructs are accurately operationalized. [Bhandari 2022]

- ▷ **Loss of Interactive Element in the Evaluation:** By displaying only the software's user interfaces without live backend interaction or the capability to generate dynamic events, the evaluation may not fully capture the essence of using the software in a real-world scenario. This limitation can lead to a partial understanding of the software's functionality and performance, thus affecting the construct validity. However, the focus lies on ensuring the same evaluation environment for every participant. Otherwise, supervision would be essential for this evaluation, and the results would become

5. Evaluation

difficult to compare due to different experiences. The interactive element could have led to circumstances that would also violate the construct validity, especially for the metrics, since the values differ from time to time and depend on the user's action.

Related Work

Unfortunately, another work that uses OpenTelemetry for software visualization has not been published. Therefore, this work is compared with react-bratus, an interactive web-based component hierarchy visualization for React-based projects. [Boersma and Lungu 2021] The reason for that is that theoretically similar software visualization approach to ExplorViz is deployed. Furthermore, the gathering process of data from a JavaScript framework is related to automatic instrumentation, since react-bratus uses pre-built libraries as well.

React-bratus primarily targets novice developers grappling with the complexities of React applications. This tool stands out for its capability to parse the source code of React projects, meticulously extracting detailed information about components and their intricate relationships, thereby facilitating a deeper understanding and reverse-engineering process of React applications. [Boersma and Lungu 2021]

This thesis pursues a similar path, using two tools instead of one. While OpenTelemetry instruments data, ExplorViz visualizes the resulting insights into structure and dynamics. React-bratus offers both functionalities in one tool, focusing on only one JavaScript framework, while this thesis spans the complete programming language.

Comparing the data extraction methodology might give insights into other solutions for automatic instrumenting JavaScript applications. React-bratus parses JavaScript and JSX to produce an Abstract Syntax Tree (AST) in line with the ESTree specification using the library Babel. With this method, specific AST nodes recognize React components, with the criterion that a piece of code is considered a component if it returns JSX elements. Furthermore, the parser adeptly identifies relationships between components, assuming a project-wide uniqueness of component names. [Boersma and Lungu 2021]

However, JSX (JavaScript XML, formally JavaScript Syntax eXtension) is adapted mainly by React and some other Web frameworks, which means this technique does not apply to server-side frameworks. Therefore, the idea behind react-bratus cannot be adopted broadly. Furthermore, the data format does not conform with classical trace data, which would result in a severe architecture adjustment when planning to integrate it into ExplorViz. Although the theoretical approach of software visualization looks pretty similar, there is an argument that ExplorViz might provide more profound insights by using multi-layer hierarchies in practice.

This work aimed to evaluate automatic instrumentation by OpenTelemetry of JavaScript

6. Related Work

applications as a tool for software visualization. This would have included a broad sample of JavaScript frameworks if it had successfully worked out. So, different to react-bratus [Boersma and Lungu 2021], the aim was always to find a solution for visualizing JavaScript software, not micro-focusing on one framework out of many.

Conclusions and Future Work

7.1 Conclusions

This work focused on evaluating the automatic instrumentation of JavaScript software by OpenTelemetry for software visualization in the context of ExplorViz. The evaluation revealed that while it is technically possible to visualize traces and metrics in the user interface, the visualizations lack depth of information. This is primarily the case because ExplorViz requires the fully qualified name of the span's origin to gain structural knowledge, which is not provided by JavaScript's base architecture during automatic instrumentation. This limitation makes the expressed improvements challenging to achieve at present. Additionally, OpenTelemetry does not offer more customization options during the instrumentation, which hampers the creation of custom solutions. On the contrary, many options exist that satisfy many requirements and let one shape the collector flow as individually as one prefers. Also, a minor interference in the source code is necessary to deploy instrumentation for Web applications, which technically violates automatic instrumentation.

Furthermore, this thesis found that the automatic instrumentation of metrics via OpenTelemetry, followed by processing in a newly written service in ExplorViz's architecture and visualization in its frontend, was adequate. However, the evaluation revealed that the instrumented metrics, while functioning well, do not significantly enhance the understanding of the software. This is partly due to the limited types of metrics available for visualization and the lack of connection to the visualized software landscape.

In conclusion, this work has demonstrated that OpenTelemetry's automatic instrumentation of JavaScript software is unsuitable for software visualization in ExplorViz. The limitations of the JavaScript base architecture, which has no method of getting the fully qualified name to every span while ExplorViz requires it, pose significant challenges. The instrumented metrics lack quantity and variety, which makes it difficult to connect them to the trace visualization. While the goals defined in Section 1.2 were achieved, the evaluation highlighted the disadvantages of the instrumentation when visualized. Therefore, this work concludes that the tool is not currently viable for software visualization in ExplorViz when instrumenting JavaScript.

7. Conclusions and Future Work

7.2 Future Work

The missing "fully qualified name" is identified as the main problem of the instrumentation. As mentioned, this might be primarily rooted in the underlying structure of JavaScript itself, and a possible future work is not to overwork the programming language. However, when following the approach that at least function names could be fetched, OpenTelemetry needs to offer more customization of the automatic instrumentation process. This should include gathering more information at the moment of instrumentation and not only inserting additional attributes manually. Spans with deeper insight could lead to a software visualization that gives the user a broad understanding of the structure and communication between components.

The same goes for the automatic instrumented metrics, which are very performance-sided and may not differ much from the insight a monitoring plug-in of a common browser might give. If more meaningful metrics that concern internal processes are gathered by automatic instrumentation, they will better support any software visualization linking them to single structural components. Currently, metrics are not sufficiently gathered automatically for Web frameworks like Angular, which could be something OpenTelemetry adds in the near future, broadening their portfolio. The instrumentation of the pet clinic example could be enhanced by using OpenTelemetry as a tool for both the front and backend. One might hope for communication lines to become visible between both components, but that is unfortunately not for sure. Another addition might be optimizing the metric-service by using the token secret to authenticate for GET requests from the frontend. Also, connecting it to the technologies used in ExplorViz Kafka and Cassandra would result in a better validation of processed spans since the landscape token and the secret could be checked to see whether the frontend created them.

Finally, this work has evaluated OpenTelemetry's proposed automatic instrumentation of JavaScript software for visualization. Future work might involve testing it with other programming languages like Python or PHP. This would also answer the question of whether the problem of this work stems from the language or the way the libraries function.

Bibliography

- [Bhandari 2019] P. Bhandari. *Internal validity in research | definition, threats & examples*. <https://www.scribbr.com/methodology/internal-validity/>. Revised in 2023. Accessed: [17.03.2024]. 2019. (Cited on page 42)
- [Bhandari 2020] P. Bhandari. *External validity | definition, types, threats & examples*. <https://www.scribbr.com/methodology/external-validity/>. Revised in 2023. Accessed: [17.03.2024]. 2020. (Cited on page 43)
- [Bhandari 2022] P. Bhandari. *Construct validity | definition, types, & examples*. <https://www.scribbr.com/methodology/construct-validity/>. Revised in 2023. Accessed: [17.03.2024]. 2022. (Cited on page 43)
- [Boersma and Lungu 2021] S. Boersma and M. Lungu. React-bratus: visualising react component hierarchies. In: *2021 Working Conference on Software Visualization (VISSOFT)*. IEEE. 2021, pages 130–134. (Cited on pages 45, 46)
- [Fittkau et al. 2017] F. Fittkau, A. Krause, and W. Hasselbring. Software landscape and application visualization for system comprehension with explorviz. *Inf. Softw. Technol.* 87 (2017), pages 259–277. DOI: 10.1016/j.infsof.2016.07.004. (Cited on pages 1, 10)
- [Fittkau et al. 2015] F. Fittkau, S. Roth, and W. Hasselbring. Explorviz: visual runtime behavior analysis of enterprise application landscapes (2015). DOI: 10.18151/7217313. (Cited on page 11)
- [Hasselbring et al. 2020] W. Hasselbring, A. Krause, and C. Zirkelbach. Explorviz: research on software visualization, comprehension, and collaboration. *Software Impacts* 6 (2020). DOI: 10.1016/j.simpa.2020.100034. (Cited on pages 1, 10, 11)
- [InfluxData 2024] InfluxData. *Influxdb time series platform*. Accessed: [09.03.2024]. 2024. URL: <https://www.influxdata.com/>. (Cited on page 12)
- [Kirešová et al. 2023] S. Kirešová, M. Guzan, B. Sobota, V. Fedák, R. Bača, and D. Bakši. The use of time series database in measurements. In: *2023 International Conference on Electrical Drives and Power Electronics (EDPE)*. IEEE. 2023, pages 1–8. (Cited on pages 12, 13)
- [Krause et al. 2021] A. Krause, M. Hansen, and W. Hasselbring. Live visualization of dynamic software cities with heat map overlays. *arXiv preprint arXiv:2109.14217* (2021). URL: <https://arxiv.org/abs/2109.14217v1>. (Cited on page 16)
- [Krause et al. 2018] A. Krause, C. Zirkelbach, and W. Hasselbring. Simplifying software system monitoring through application discovery with explorviz. *Softwaretechnik-Trends* 39 (2018), pages 46–48. (Cited on page 11)

Bibliography

- [Nasar and Kausar 2019] M. Nasar and M. A. Kausar. Suitability of influxdb database for iot applications. *International Journal of Innovative Technology and Exploring Engineering* 8.10 (2019), pages 1850–1857. (Cited on page 12)
- [NovaTec Consulting GmbH 2024] NovaTec Consulting GmbH. *Inspectit ocelot documentation*. <https://inspectit.github.io/inspectit-ocelot/docs/doc1>. Accessed: [15.03.2024]. 2024. (Cited on pages 30, 31)
- [OpenTelemetry Contributors 2024] OpenTelemetry Contributors. *Opentelemetry documentation*. Accessed: [07.03.2024]. 2024. URL: <https://opentelemetry.io/docs/>. (Cited on pages 1, 5–7, 9, 15, 18, 23)
- [Prometheus Authors 2024] Prometheus Authors. *Prometheus - overview*. Accessed: [23.03.2024]. 2024. URL: <https://prometheus.io/docs/introduction/overview/>. (Cited on pages 15, 25, 26)
- [Protocol Buffers 2024] Protocol Buffers. *Protocol buffers - google's data interchange format*. Accessed: [08.03.2024]. 2024. URL: <https://protobuf.dev/>. (Cited on page 7)
- [Thakur and Chandak 2022] A. Thakur and M. B. Chandak. A review on opentelemetry and http implementation. *International Journal of Health Sciences* 6.S2 (2022), pages 15013–15023. DOI: 10.53730/ijhs.v6nS2.8972. URL: <https://doi.org/10.53730/ijhs.v6nS2.8972>. (Cited on page 5)
- [Tilkov and Vinoski 2010] S. Tilkov and S. Vinoski. Protocol buffers: an efficient serialization format. *IEEE Internet Computing* 14.6 (2010). (Cited on page 7)
- [Zipkin Community 2024] Zipkin Community. *Zipkin - distributed tracing system*. Accessed: [23.03.2024]. 2024. URL: <https://zipkin.io/pages/community.html>. (Cited on pages 15, 21)